Реальные проблемы и решения

2-е издание

Java™ и XML

O'REILLY®

Бретт Мак-Лахлин

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-018-9, название «Java и XML» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Java & XML

Second Edition

Brett McLaughlin

O'REILLY®

Java и XML

Второе издание

Бретт Мак-Лахлин

Символ

Санкт-Петербург
2002

Бретт Мак-Лахлин

Java и XML

Перевод Т. Морозовой

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*М. Зислис*
Редактор	*В. Овчинников*
Корректура	*С. Беляева*
Верстка	*А. Дорошенко*

Мак-Лахлин Б.
Java и XML. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 544 с., ил.
ISBN 5-93286-018-9

Java и XML. Эти две технологии уже давно привлекают внимание разработчиков. И не зря. Они идеально подходят для создания веб-ориентированных корпоративных приложений, обеспечивают независимость от платформы, расширяемость, возможность повторного использования кода, а также поддержку стандарта Unicode. Их соединение позволяет создавать веб-сайты с динамически обновляемыми страницами, разрабатывать корпоративные приложения, снижающие затраты на совместное использование информации, и находить простые и эффективные решения проблемы переносимости данных. Автор описывает применение всего арсенала инструментов и средств XML и Java. Здесь и DTD и пространства имен, XML Schema и XPath, XSL и различные API (SAX, DOM, JDOM). Рассматривается связывание данных, разработка приложений при помощи XML-RPC и SOAP, использование систем веб-публикации (например, Apache Cocoon). Не оставлены вниманием создание веб-служб с применением SOAP, UDDI и WSDL, каналы RSS, динамические данные и XSP.

Второе издание «Java и XML» дополнено главами о расширенных возможностях SAX и DOM, а также о SOAP и связывании данных. Эта книга станет незаменимым спутником для тех, кто пишет программы на Java и собирается применять XML (или планирует заниматься этим), участвует в движении peer-to-peer, разрабатывает программное обеспечение для электронной коммерции либо использует службу сообщений или веб-службы.

ISBN 5–93286–018–9
ISBN 0–596–00197–5 (англ)

© Издательство Символ-Плюс, 2002
Authorized translation of the English edition © 2001 O'Reilly & Associates Inc. This translation is published and sold by permission of O'Reilly & Associates Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 27.05.2002. Формат 70x100 1/16 . Печать офсетная.
Объем 34 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 9

Структура этой книги . . . 9
Для кого предназначена эта книга? . . . 12
Программное обеспечение и версии . . . 13
Типографские соглашения . . . 14
Комментарии и вопросы . . . 14
Благодарности . . . 15

1. Введение . . . 17

XML имеет значение . . . 17
Что важно? . . . 20
Основы . . . 23
Что дальше? . . . 26

2. Основы технологии . . . 27

Основы . . . 28
Ограничения . . . 40
Преобразования . . . 48
И еще... . . . 58
Что дальше? . . . 58

3. SAX . . . 59

Подготовительные операции . . . 60
SAX-совместимые анализаторы . . . 62
Обработчики содержимого . . . 69
Обработчики ошибок . . . 87
Советы разработчикам . . . 93
Что дальше? . . . 97

4. Расширенный SAX . . . 98

Свойства и возможности . . . 99
И опять обработчики . . . 107
Фильтры и объекты Writer . . . 113
И снова обработчики . . . 120

Советы разработчикам 125
Что дальше? 128

5. Объектная модель документа 129

Объектная модель документа 129
Сериализация 135
Изменяемость 148
Советы разработчикам 149
Что дальше? 151

6. Расширенный DOM 152

Изменения 153
Пространства имен 164
Модули DOM Level 2 168
DOM Level 3 183
Советы разработчикам 187
Что дальше? 188

7. JDOM 189

Основы 189
Класс PropsToXml 195
Класс XMLProperties 206
Является ли JDOM стандартом? 217
Советы разработчикам 219
Что дальше? 221

8. Расширенный JDOM 222

Полезная информация по внутренней организации JDOM 222
JDOM и фабрики 228
Классы Wrapper и Decorator 233
Советы разработчикам 246
Что дальше? 248

9. JAXP 249

API или абстракция 249
JAXP 1.0 251
JAXP 1.1 260
Советы разработчикам 270
Что дальше? 272

10. Системы веб-публикации 273

Выбор системы публикации 275
Установка 278
Применение системы публикации 283

XSP . . . 300
Cocoon 2.0 и выше . . . 316
Что дальше? . . . 319

11. XML-RPC . . . 320

RPC и RMI: за и против . . . 321
Простейшее приложение . . . 324
Переложение нагрузки на сервер . . . 337
Реальные ситуации . . . 353
Что дальше? . . . 356

12. SOAP . . . 357

Начинаем . . . 357
Установка . . . 361
«Пачкаемся» . . . 366
Идем дальше . . . 375
Что дальше? . . . 384

13. Веб-службы . . . 385

Веб-службы . . . 385
UDDI . . . 387
WSDL . . . 389
Собираем все вместе . . . 392
Что дальше? . . . 411

14. Объединение содержимого . . . 412

Библиотека Foobar . . . 413
Компания mytechbooks.com . . . 423
Рассылка или запрос . . . 433
Что дальше? . . . 443

15. Связывание данных . . . 444

Основные принципы . . . 446
Castor . . . 452
Zeus . . . 460
JAXB . . . 469
Что дальше? . . . 477

16. Взгляд в будущее . . . 478

XLink . . . 478
XPointer . . . 480
Отображения для схем XML . . . 485
И прочее... . . . 486
Что дальше? . . . 487

A. Справочник по API 488

SAX 2.0 488
DOM Level 2 501
JAXP 1.1 510
Пакет javax.xml.transform.stream 516
JDOM 1.0 (Beta 7) 516
Пакет org.jdom 516

B. Возможности и свойства SAX 2.0 528

Основные возможности 528
Базовые свойства 530

Алфавитный указатель 532

Предисловие

Когда я чуть больше года назад писал предисловие к первому изданию «Java и XML», я даже и не представлял себе, с чем связываюсь. Я отпускал шуточки о том, что XML скоро появится на шапках и футболках; сейчас, когда я пишу эти строки, на мне футболка с вышитой на ней надписью «XML», а еще у меня есть кепка с надписью XML (на самом деле у меня их даже две!). Так что XML, без сомнения, оправдал все ожидания. И это хорошо.

Но это означает, что каждый день ведутся новые разработки, и ландшафт XML расширяется с такой скоростью, которую я не мог себе представить даже в самых смелых предположениях. И хотя для XML это здорово, при обращении к первому изданию это ввергает в уныние: почему все настолько устарело? Я говорил о SAX 2.0 и DOM Level 2 как о чем-то новом. Теперь это промышленные стандарты. Я только знакомил читателей с JDOM, а теперь он входит в JSR (Sun's Java™ Specification Request, запрос на разработку или изменение спецификации Java-технологии). Я даже не рассматривал SOAP, UDDI, WSDL и связывание данных XML. В этом издании им отведено три главы! Все изменилось, и это не преувеличение.

Даже тем, кто только подозревает, что в ближайшие несколько месяцев, возможно, будет работать с XML, эта книга, весьма вероятно, будет полезна. А если где-то на вашем рабочем столе есть первое издание этой книги, то просмотрите и новое издание; думаю, вы обнаружите, что оно все же актуально. Я избавился от чрезмерных описаний основных концепций, собрал материал об основах XML в одной главе и переписал практически все примеры. Кроме того, я добавил много новых примеров и глав. Другими словами, я попытался создать подробную, многогранную техническую книгу. Начинающим на ее изучение потребуется немного больше времени, поскольку помощи стало меньше, зато знаний будет гораздо больше.

Структура этой книги

Эта книга очень четко структурирована: в ее первой половине – главах с 1 по 9 – основное внимание уделяется изучению основ XML и ключевых интерфейсов прикладного программирования Java для обработки XML. Каждому из трех интерфейсов для работы с XML (SAX, DOM

и JDOM) автор посвящает по главе, в которой рассматривает базовые понятия, и еще по главе, описывающей более сложные концепции. Глава 10 – переходная, здесь начинается подъем по «стеку» XML. В ней рассматривается JAXP, представляющий собой уровень абстракции над SAX и DOM. Остальная часть книги – главы с 11 по 15 – посвящена специфическим вопросам, которые постоянно поднимаются в конференциях и руководствах, с которыми я связан, и предназначена для желающих научиться применять XML в своих приложениях. Среди этих тем новые главы по SOAP, связыванию данных и новый взгляд на приложения business-to-business. Наконец, есть еще два приложения, завершающие книгу. Итак, содержимое книги таково:

Глава 1 «Введение»

Посмотрим, о чем, собственно, шум, изучим начала XML и уделим время обсуждению причин, по которым XML настолько важен для настоящего и будущего корпоративных разработок.

Глава 2 «Основы технологии»

Это ускоренный курс основ XML, от XML 1.0 до DTD и от XML Schema до XSLT и пространств имен. Для тех, кто читал первое издание, это сумма всех глав, посвященных работе с XML, и кое-какой дополнительный материал.

Глава 3 «SAX»

В этой главе вводится и рассматривается простой API для XML (SAX), первый Java API для обработки XML. Подробно рассматривается жизненный цикл процесса анализа, а затем демонстрируются события, которые SAX может перехватывать и которые могут использоваться разработчиками.

Глава 4 «Расширенный SAX»

Здесь мы еще глубже познакомимся с SAX, рассматривая реже используемые, но тем не менее очень мощные элементы API. Узнаем, как применять фильтры XML для объединения в цепи обратных вызовов, как применять классы `XMLWriter` и `DataWriter` для вывода XML при помощи SAX. Кроме того, мы рассмотрим некоторые из менее распространенных обработчиков SAX, такие как `LexicalHandler` и `DeclHandler`.

Глава 5 «DOM»

В этой главе читатель движется дальше по ландшафту XML к следующему API Java и XML – DOM (Document Object Model, объектная модель документа). Вы изучите основы DOM, выясните, что находится в текущей спецификации (DOM Level 2), и узнаете, как читать и создавать деревья DOM.

Глава 6 «Расширенный DOM»

Продолжая изучать DOM, вы узнаете о различных модулях DOM, например Traversal, Range, Events, CSS и HTML. Также мы посмот-

рим, что собой предоставляет новая версия DOM Level 3 и как применять эти новые возможности.

Глава 7 «JDOM»

Эта глава содержит введение в JDOM и описывает его отличия от DOM и SAX, не оставляя без внимания их сходства. Рассматривается чтение и создание XML при помощи этого API.

Глава 8 «Расширенный JDOM»

Углубляя знакомство с JDOM, мы рассмотрим практические приложения API – интеграцию с JAXP и то, как JDOM обеспечивает использование фабрики с вашими собственными подклассами JDOM. Кроме того, вы увидите, как XPath действует в паре с JDOM.

Глава 9 «JAXP»

В настоящее время JAXP – это развившийся API, поддерживающий анализ и преобразования, и он заслуживает отдельной главы. Тут мы рассмотрим обе версии – 1.0 и 1.1, и вы узнаете, как использовать этот API во всей полноте.

Глава 10 «Системы веб-публикации»

В этой главе рассказано, чем является система веб-публикации, почему она имеет к вам отношение и как выбрать хорошую систему. Затем мы остановимся на системе Apache Cocoon, более внимательно рассмотрим множество ее свойств и то, как она может применяться для создания динамического наполнения веб-страниц.

Глава 11 «XML-RPC»

Здесь рассматриваются удаленные вызовы процедур (Remote Procedure Calls, RPC), их уместность в распределенном окружении по сравнению с RMI и то, как XML делает RPC жизнеспособным решением некоторых проблем. Затем мы перейдем к применению Java-библиотек XML-RPC и созданию клиентов и серверов XML-RPC.

Глава 12 «SOAP»

В этой главе мы остановимся на использовании данных конфигурирования в формате XML и увидим, почему этот формат столь важен для кроссплатформенных приложений, особенно по отношению к распределенным системам и веб-службам.

Глава 13 «Веб-службы»

Продолжая говорить о SOAP и веб-службах, в этой главе мы познакомимся с двумя важными технологиями – UDDI и WSDL.

Глава 14 «Объединение содержимого»

Не покидая русла межкорпоративных (business-to-business) приложений, эта глава представляет еще один способ взаимодействия приложений – при помощи соглашений о содержимом. Вы узнаете о Rich Site Summary, создании информационных каналов и даже немного о Perl.

Глава 15 «Связывание данных»

И снова вверх по стеку XML. В этой главе говорится об одном из высокоуровневых интерфейсов для Java и XML, а именно интерфейсе связывания данных XML. Вы узнаете, что такое связывание данных, как оно позволяет существенно упростить работу с XML, а также осознаете существующие возможности. Я рассмотрю три системы: Castor, Zeus и недавнюю версию JAXB (Java Architecture for XML Data Binding, архитектура Java для связывания данных XML) от Sun.

Глава 16 «Взгляд в будущее»

В этой главе я указываю на несколько интересных технологий, появившихся на горизонте, и привожу немного дополнительной информации по каждой из них. Некоторые из моих догадок могут быть совершенно неверными, другие же могут оказаться действительно чем-то интересным.

Приложение A «Справочник по интерфейсам прикладного программирования»

В приложении подробно описаны все классы, интерфейсы и методы, доступные в интерфейсах SAX, DOM, JAXP и JDOM.

Приложение B «Возможности и свойства SAX 2.0»

В этом приложении подробно рассмотрены возможности и свойства, доступные в реализациях синтаксических анализаторов, поддерживающих SAX 2.0.

Для кого предназначена эта книга?

Эта книга основана на предпосылке, что XML очень быстро становится (и в некоторой степени уже стал) важным аспектом программирования на Java. Материал книги учит вас применять XML и Java, и нигде, кроме как в первой главе, не задается вопрос «*следует ли* применять XML». Разработчикам на Java несомненно следует использовать XML. Вот почему эта книга адресована программистам на Java, а также тем, кто собирается программировать на Java, имеет в подчинении программистов на Java либо связан с проектом, в котором используется Java. Если вы хотите совершенствоваться как разработчик, создавать более понятный код или хотите, чтобы проект был завершен вовремя и не вышел за рамки бюджета, если вам необходим доступ к данным устаревших систем, если вам нужно распределение системных компонентов либо если вы просто хотите знать, почему XML уделяется столько внимания, то эта книга для вас.

Я старался как можно меньше представлять себе конкретного читателя; я не верю, что следует усложнять введение в XML настолько, чтобы невозможно было войти в курс дела. Тем не менее, я также пола-

гаю, что если вы потратили деньги на эту книгу, то вам нужны не просто основы. Поэтому я предполагал лишь, что вы знакомы с языком Java и понимаете основные концепции серверного программирования (скажем, из области Java-сервлетов и Enterprise JavaBeans™). Если вы никогда прежде не писали на Java или только начинаете изучать язык, то, возможно, вам стоит прочесть книгу Пэта Нимайера (Pat Niemeyer) и Джонатана Кнудсена (Jonathan Knudsen) «Learning Java», O'Reilly (Изучаем Java), прежде чем приступать к этой книге. Я не предполагаю, что вы знаете что-либо об XML, поэтому начинаю с простого. Но я полагаю, что вы хотите хорошо поработать и быстро научиться; поэтому базовые понятия мы рассмотрим быстро, а основной объем книги будет посвящен работе с более сложными вещами. Материал не повторяется, если нет явной необходимости, так что кому-то, возможно, понадобится перечитать предшествующие разделы или полистать страницы в обоих направлениях, поскольку рассмотренные ранее понятия используются в последующих главах. У тех, кто хочет изучить XML, немного знаком с Java и готов набирать код примеров в своем любимом текстовом редакторе, не должно возникнуть особых проблем при чтении этой книги.

Программное обеспечение и версии

В этой книге рассмотрен стандарт XML версии 1.0 и различные словари XML по состоянию на июль 2001 года. Поскольку многие затронутые спецификации XML еще не приняты в окончательном варианте, могут существовать незначительные несоответствия между печатным изданием данной книги и текущей версией обсуждаемой спецификации.

Весь приводимый код на Java основан на платформе Java 1.2. Если вы пока еще не пользуетесь Java 1.2, постарайтесь это сделать – одни только классы коллекций стоят этого. Синтаксический анализатор Apache Xerces, процессор Apache Xalan, библиотека Apache SOAP и библиотеки Apache FOP применялись в виде самых свежих доступных стабильных версий на июнь 2000 года, а система веб-публикации Apache Cocoon – в версии 1.8.2. Версия использованных Java-библиотек XML-RPC – 1.0 beta 4. Все обсуждаемое программное обеспечение распространяется свободно, и его можно получить в сети по адресам *http://java.sun.com*, *http://xml.apache.org* и *http://www.xml-rpc.com*.

Исходные тексты примеров из этой книги полностью содержатся на ее страницах. Как исходные тексты, так и скомпилированные версии всех примеров (включая объемную документацию Javadoc, не обязательно включенную в текст книги) доступны в сети по адресам *http://www.oreilly.com/catalog/javaxml2* и *http://www.newInstance.com*. Все примеры, которые могут выполняться в качестве сервлетов либо могут быть преобразованы для выполнения в таком качестве, можно просмотреть и использовать, посетив страницу *http://www.newInstance.com*.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсивом выделяются:

- Имена путей Unix, имена файлов и имена программ
- Адреса Интернета, включая доменные имена и URL
- Определения новых терминов

Полужирным шрифтом выделены:

- Имена элементов GUI: заголовки окон, кнопки, пункты меню и т. д.

Моноширинным шрифтом выделены:

- Командные строки и параметры, которые необходимо ввести дословно
- Имена и ключевые слова в программах Java, включая названия методов, имена переменных и имена классов
- Имена элементов и тегов XML, имена атрибутов и других конструкций XML, приводимых в том виде, в каком они присутствуют в XML-документе

Комментарии и вопросы

Пожалуйста, направляйте замечания и вопросы относительно этой книги издательству:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (в США или Канаде)
(707) 829-0515 (международный/местный)
(707) 829-0104 (факс)

Также можно отправить сообщение по электронной почте. Чтобы попасть в наш список рассылки или заказать каталог, воспользуйтесь адресом

info@oreilly.com

Чтобы задать технический вопрос или прокомментировать содержание книги, напишите по адресу

bookquestions@oreilly.com

Мы поддерживаем веб-сайт, посвященный этой книге, на котором будут приведены примеры, замечания о найденных ошибках и планы переизданий. Эта страница расположена по адресу

http://www.oreilly.com/catalog/javaxml2/

Более подробную информацию об этой и других книгах можно найти на веб-сайте издательства O'Reilly:

http://www.oreilly.com

Благодарности

Вот и снова я пишу благодарности. Сейчас вспомнить каждого ничуть не проще, чем в первый раз. Мой редактор Майк Лукидес (Mike Loukides) даже по ночам беспокоится о том, как сделать все правильно, и именно таким должен быть хороший редактор! Кайл Харт (Kyle Hart) – суперженщина из отдела маркетинга – ведет всю работу и напоминает мне о том, что в конце туннеля виден свет. Тим О'Рейли (Tim O'Reilly) и Фрэнк Уиллисон (Frank Willison) терпеливы, но настойчивы, и именно такими должны быть хорошие руководители. А Боб Экштейн (Bob Eckstein) и Марк Лой (Marc Loy) помогали мне разрешать досадные проблемы с графическим интерфейсом Swing. (Кроме того, Боб просто забавный парень. Посмотрим правде в глаза.) O'Reilly – настолько хорошее издательство, насколько это возможно, во всех отношениях. Я горжусь этим сотрудничеством.

Также хочу поблагодарить бесподобную команду моих рецензентов. Много раз они просматривали главу меньше чем за 24 часа, при этом выдавая добросовестный отчет по техническим деталям. Именно эти люди внесли весомый вклад в поддержание технического уровня книги. Роберт Сиз (Robert Sese), Филип Нельсон (Philip Nelson) и Виктор Брилон (Victor Brilon) – все они просто изумительны. Конечно же, я всегда готов благодарить своего «соучастника» Джейсона Хантера (Jason Hunter) за то, что он предан JDOM и другим техническим вопросам (старина, устрой себе выходной на одну ночь!). Наконец, компания, в которой я работаю – Lutris Technologies, – это, пожалуй, лучшее место, в котором можно пожелать работать. Они позволили мне много часов работать над книгой и не жаловались ни разу. В особенности я хочу отметить Янси Линд (Yancy Lind), Пола Моргана (Paul Morgan), Дэвида Юнга (David Young) и Кейт Бигелоу (Keith Bigelow) – они просто мастера своего дела. Спасибо!

Снова спасибо моим родителям – Ларри (Larry) и Джуди Мак-Лахлин (Judy McLaughlin). Я люблю вас за то, что вы вырастили в меру честолюбивого и целеустремленного сына (вы, конечно, понимаете, что эти характеристики также относятся и к невыносимому ребенку!). Сара Джейн (Sarah Jane) – моя тетя, и мои бабушка и дедушка Дин (Dean) и Глэдис Мак-Лахлин (Gladys McLaughlin) никогда не допускали даже и мысли о том, что я о них не думаю только из-за того, что вижу их редко. Дедушка, ты даже не догадываешься, насколько я благодарен тебе за то, что ты собираешься посмотреть на второе издание. Я люблю вас всех.

Спасибо моим вторым родителям (родителям жены), Гари (Gary) и Ширли Грейтхаус (Shirley Greathouse) – вы просто лучше всех. Когда-нибудь я соберусь и объясню, что вы значите для меня, но это может потребовать отдельной книги. Я люблю вас обоих за ваш юмор и мудрость. Спасибо Куину (Quinn) и Джони (Joni) за ту легкость, которую они привносят в наши воскресные обеды. Спасибо Лонни и Лауре, не могу дождаться, когда увижу Беби Джей (Baby J). Спасибо Биллу и Терри за то, что они были очень мудрыми друзьями, а также спасибо Биллу за то, что он такой пастор, какого еще нужно поискать.

Источник смеха в моей жизни – это несколько веселых людей, и я просто не могу не упомянуть их тут: Кендра (Kendra), Британи (Brittany), Лизетт (Lisette), Дженей (Janay), Роки (Rocky), Дастин (Dustin), Тони (Tony), Стефани (Stephanie), Робби (Robbie), Эрин (Erin), Анджела (Angela), Майк (Mike), Мэтт (Matt), Карлос (Carlos) и Джон (John). Я увижусь с вами в воскресенье, и, пожалуйста, давайте не пойдем больше в Mazzio's. Также спасибо моим собакам: Сету, Чарли, Джейку, Мозесу, Молли и Дэйзи. Тот не жил по-настоящему, кого не будил по утрам холодный язык бассет-хаунда.

Наконец, спасибо двум людям, которые значат для меня гораздо больше, чем остальные – моему дедушке Роберту Эрлу Бердону, с которым я вновь встречусь однажды. Я думаю о тебе каждый день, и мои дети тоже вскоре о тебе услышат. Но больше всего я хочу поблагодарить мою жену Ли. Словами этого не выразить. Однажды все песни и слезы, связанные с тобой, выплеснутся наружу, и ты наконец-то поймешь, как много для меня значишь.

Наконец, спасибо Господу, который дал мне достичь столь многого.

- *XML имеет значение*
- *Что важно?*
- *Основы*
- *Что дальше?*

1

Введение

Обычно писать вводные главы очень просто. Практически в каждой книге автор приводит обзор рассматриваемой технологии, разъясняет некоторые основы и пытается заинтересовать читателя. Но в случае со вторым изданием «Java и XML» все немного сложнее. Когда шла работа над первым изданием, многие люди только начинали знакомиться с XML, а многие скептики желали проверить, действительно ли этот новый тип разметки так хорош, как обещает рекламная шумиха. Но прошло чуть больше года, и уже все используют XML сотнями различных способов. В известной степени введение, вероятно, вам не нужно. Но я все же вкратце опишу, о чем пойдет речь, почему тема важна и что понадобится, чтобы начать работу.

XML имеет значение

Во-первых, я расскажу о том, что имеет значение. Да, это звучит как вводная фраза на занятии по аутотренингу, но с этого имеет смысл начать. По-прежнему многие разработчики, менеджеры и руководители боятся XML. Их пугает восприятие XML как передовой технологии и скорость, с которой эта технология обновляется. (Это второе издание, вышедшее через год, верно? Многое ли изменилось?) Они боятся затрат, связанных с наймом людей вроде нас с вами, людей, которые будут работать с XML. Но более всего они боятся дальнейшего усложнения своих и без того головоломных приложений.

Для того чтобы устранить эти страхи, кратко перечислим основные причины, по которым следует начать работать с XML уже сегодня. Во-первых, XML переносим. Во-вторых, он предоставляет невероятный потенциал для взаимодействий. И наконец, XML имеет значение... потому что не имеет значения! Если это окончательно вас запутало, читайте дальше, и очень скоро все станет на свои места.

Переносимость

Во-первых, XML переносим. Если вы давно работаете с Java или когда-либо участвовали в конференции JavaOne[1], то слышали заклинание адептов Java: «переносимый код». Скомпилируйте код на Java, перенесите файлы *.class* или *.jar* в любую операционную систему, и код заработает. Все что нужно – окружение Java Runtime Environment (JRE) или виртуальная машина Java (JVM). Переносимость всегда была одной из наиболее привлекательных особенностей Java. Разработчики могут создавать и тестировать код на рабочих станциях под управлением Linux или Windows, и созданный код будет выполняться на Sparc, E4000, HP-UX – в любой операционной среде, которую только можно представить.[2]

Так что технология XML заслуживает больше, чем мимолетного взгляда. XML – это просто текст, и очевидно, что его можно переносить на различные платформы. Что еще более важно, XML должен соответствовать спецификации, определенной консорциумом World Wide Web (W3C), веб-сайт которого расположен по адресу *http://www.w3.org*. Таким образом, XML – это стандарт. Когда вы посылаете XML-текст, он соответствует этому стандарту; когда некоторое приложение его получает, XML по-прежнему соответствует этому стандарту. Приложение, использующее XML-данные, вправе на это рассчитывать. Так работает Java: любая виртуальная машина Java знает, чего ожидать, и коль скоро код соответствует этим ожиданиям, он будет выполняться. Используя XML, вы получаете переносимые данные. В последнее время можно услышать слова «переносимый код», «переносимые данные», относящиеся к комбинации Java и XML. Это хорошие слова, потому что они правдивы (что не всегда справедливо для рекламных лозунгов).

Способность к взаимодействию

Во-вторых, XML позволяет достичь такого уровня взаимодействия, который никогда раньше не был доступен корпоративным приложениям. Некоторые из читателей, вероятно, думают, что речь идет лишь о еще одной форме переносимости. Это не так. Вспомним, что XML – это *расширяемый язык разметки*. Именно расширяемость так важна во взаимодействии. Рассмотрим для примера HTML – язык гипертекстовой разметки. HTML – это стандарт. Он полностью текстовый. В этом отношении он так же переносим, как и XML. И действительно,

---

[1] Конференция JavaOne (в конвент-центре им. Москони, г. Сан-Франциско) ежегодно собирает десятки тысяч разработчиков и ключевых фигур индустрии Java. Более подробную информацию о конференции можно найти по адресу *http://servlet.java.sun.com/javaone/*. – *Примеч. науч. ред.*

[2] Наличие на целевой платформе соответствующей версии JRE или JVM является обязательным условием. – *Примеч. науч. ред.*

клиенты, использующие различные броузеры и различные операционные системы, могут увидеть в той или иной степени одно и то же. Однако HTML специально разработан в целях визуализации информации. HTML нельзя использовать для представления описи имущества или выписанного счета.[1] Дело в том, что стандарт HTML жестко определяет разрешенные теги, формат документа и прочие особенности. Таким образом сохраняется ориентация на визуализацию, или представление данных, что одновременно является и преимуществом и недостатком.

XML, напротив, практически не затрагивает элементы и содержимое документа. Вместо этого речь идет о структуре документа: элементы должны открываться и закрываться, у атрибута должно быть только одно значение и т. д. Содержимое документа, конкретные элементы и атрибуты оставлены на усмотрение разработчика. Существует возможность создать собственный формат документа, определить его содержимое и правила представления данных, повысив таким образом эффективность взаимодействия. Различные мебельные компании (furniture chains) могут согласовать некоторый набор ограничений для XML, а затем обмениваться данными в этих форматах; они получают все преимущества XML (в частности, переносимость), а также возможность применять свои бизнес-знания к данным, чтобы придать им смысл. В биллинговой системе может быть реализован собственный формат, подходящий для генерации счетов, который будет распространяться в качестве формата для обмена данными, для экспорта и импорта счетов из других биллинговых систем. Расширяемость XML делает эту технологию идеальной для реализации совместной работы приложений.

Еще больше интригует большое количество разрабатываемых вертикальных стандартов[2]. Взгляните на проект ebXML по адресу *http://www.ebxml.org*. Компании ведут совместную работу по созданию основанных на XML стандартов, которые сделают возможной глобальную электронную коммерцию. Аналогичное движение имеет место в индустрии телекоммуникаций. В ближайшем будущем вертикальные

---

[1] На самом деле можно. В основе языка HTML – традиционные DTD-определения; в последних версиях существует атрибут `class`, позволяющий классифицировать произвольные элементы с целью применения стилей. Таким образом, теоретически HTML позволяет передавать определенную логическую нагрузку для элементов. Но, само собой, подобный способ менее практичен, чем применение XML. – *Примеч. науч. ред.*

[2] *Вертикальный стандарт*, или *вертикальный рынок*, – это стандарт или рынок, имеющий своей целью определенную отрасль бизнеса. Вместо движения по горизонтали (когда предпочтительна общая функциональность) акцент делается на вертикальном движении, когда функциональность предназначена определенной аудитории, скажем, производителям обуви или гитарным мастерам.

рынки всего мира договорятся о стандартах обмена данными, построенных на XML.

Потеря значимости

Итак, XML имеет значение потому, что он не имеет значения. Я хочу повторить это снова, поскольку речь идет о причинах, делающих XML важной технологией. Фирменные решения, связанные с хранением данных, двоичные форматы, для декодирования которых существуют конкретные алгоритмы, и другие решения этой серии – вот что в конечном счете значимо и учитывается при анализе. Разбор данных подразумевает взаимодействие с другими компаниями, обширную документацию, написание кода и повторное изобретение инструментов для передачи. Привлекательность XML заключается в том, что эта технология не требует никакой специальной подготовки, она позволяет тратить время на другие задачи. В главе 2 «Основы технологии» примерно на 30 страницах описана бо́льшая часть того, что когда-либо может понадобиться авторам XML-данных. Эти данные требуют документирования, поскольку вся необходимая документация уже существует. Они не требуют применения специальных кодировщиков и декодировщиков, поскольку уже существуют интерфейсы прикладного программирования и анализаторы, выполняющие всю обработку. Эта технология не подвергает пользователя риску, т. к. она уже испытана, и многие миллионы разработчиков ежедневно трудятся над ней, улучшая и дополняя новыми возможностями.

XML очень важен, поскольку становится надежной, не привлекающей лишнего внимания частью приложений. Создайте ограничения, закодируйте данные в XML и забудьте об этой задаче. Переходите к важным вещам – сложной бизнес-логике и представлению, которые требуют недель и месяцев размышлений и тяжелого труда. Между тем, XML будет довольно пыхтеть, без плача и жалоб управляясь с вашими данными (да, я несколько драматизирую, но вы поняли, что я хотел сказать).

Так что если вы боялись XML или скептически к нему относились, присоединяйтесь сейчас. Это может быть самым важным решением, которое вы когда-либо принимали, да еще и с минимальным числом побочных эффектов. В оставшейся части книги будут изучены интерфейсы прикладного программирования, транспортные протоколы и прочие вопросы, с которыми вы можете столкнуться.

Что важно?

Если вы согласны, что XML способен вам помочь, встает следующий вопрос – какая его *часть* вам нужна. Как я говорил ранее, существуют сотни приложений XML, и найти нужное – задача довольно сложная.

Я собрал двенадцать или тринадцать ключевых тем из этих сотен и постарался их для вас подготовить – тоже задача не из простых! К счастью, у меня был год на то, чтобы собрать отзывы по первому изданию этой книги, и я работаю с XML над созданием приложений уже два с лишним года. Это означает, что я по крайней мере имею понятие, что интересно и полезно. Если «выпарить» все различные механизмы XML, то все будет сведено лишь к нескольким категориям.

Низкоуровневые API

API – это интерфейс прикладного программирования, а низкоуровневые API – это те интерфейсы, которые позволяют работать непосредственно с содержимым XML-документа. Другими словами, не происходит практически никакой предварительной обработки, и мы получаем «нетронутые» XML-данные, которыми и будем манипулировать. Это самый эффективный, а также и самый мощный способ работы с XML. Однако он требует обширных знаний об XML и обычно связан с колоссальной работой по превращению содержимого документа во что-то полезное.

В настоящее время два наиболее распространенных интерфейса – это SAX (Simple API for XML, простой API для XML) и DOM (Document Object Model, объектная модель документа). Кроме того, в последнее время все большую силу набирает JDOM (это не аббревиатура и не расширение DOM). Все три технологии находятся в той или иной стадии стандартизации (SAX – стандарт де-факто, DOM стандартизирован W3C, а JDOM – Sun) и имеют хорошие шансы на то, чтобы оказаться долгоживущими технологиями. Все три предоставляют доступ к XML-документу в различных формах и позволяют делать с документом практически все что угодно. Довольно большая часть моей книги посвящена этим интерфейсам, поскольку они представляют собой основу всего остального, что вы будете делать в XML. Отдельная глава отведена интерфейсу JAXP (Sun's Java API for XML Processing), который является тонким уровнем абстракции над SAX и DOM.

Высокоуровневые API

Высокоуровневые API – это следующий шаг вверх по лестнице. Они не предоставляют непосредственного доступа к документу, полагаясь в этом вопросе на API низкоуровневые. Кроме того, высокоуровневые интерфейсы представляют документ в иной форме – либо более дружественной к пользователю, либо каким-либо образом моделированной, либо в форме, отличающейся от основной структуры XML-документа. Хотя эти интерфейсы часто проще использовать и с их помощью проще разрабатывать приложения, преобразование данных в иной формат может сказаться на производительности. Кроме того, понадобится потратить некоторое время на изучение интерфейса, а некоторые низкоуровневые интерфейсы придется изучить в любом случае.

В этой книге основной пример высокоуровневого интерфейса – это связывание данных XML. Связывание данных обеспечивает преобразование документа XML в объект Java, который имеет структуру, отличную от древовидной. Если у вас были элементы под названием «person» и «firstName», вы получите объект с методами, подобными `getPerson()` и `setFirstName()`. Очевидно, что это простой способ быстро разобраться с XML; вряд ли для этого требуются какие-либо глубокие знания! Однако нельзя просто изменить структуру документа (например, сделав элемент «person» элементом «employee»), поэтому применение связывания данных ограничено определенным видом приложений. Все о связывании данных можно найти в главе 15 «Связывание данных».

Приложения на основе XML

Помимо интерфейсов прикладного программирования, специально созданных для работы с документом или его содержимым, также существуют приложения, построенные на XML. Эти приложения непосредственно или косвенно используют XML, но фокусируются на решении определенной задачи, например на отображении оформленных веб-данных или на связи между приложениями. В основе работы всех описанных приложений лежит XML, что и определяет базовые особенности их поведения. Некоторые приложения требуют обширных знаний XML, некоторые не требуют ничего, но все они оказываются к месту при обсуждении Java и XML. Из их числа я выделил наиболее популярные и полезные, о которых и буду говорить здесь.

Сначала рассмотрим системы публикации, предназначенные для форматирования XML в виде данных HTML или WML (Wireless Markup Language) либо в таких двоичных форматах, как PDF (Portable Document Format). Такие системы обычно используются для предоставления клиентам сложных, специализированных веб-приложений. Затем перейдем к XML-RPC, являющемуся XML-вариантом удаленных вызовов процедур. Это лишь некоторые из полного комплекта инструментов, обеспечивающих взаимодействие приложений. Основываясь на XML-RPC, я опишу SOAP, простой протокол доступа к объектам (Simple Object Access Protocol), а также его потенциал в сочетании с XML-RPC. Затем, изучая UDDI (Universal Discovery, Description and Integration) и WSDL (Web Services Descriptor Language) в главе, посвященной межкорпоративным (business-to-business) приложениям, вы увидите новых игроков на поле веб-служб. Обогатив свой арсенал этими средствами, вы будете подкованы не только в XML, но и в любой среде корпоративных приложений.

И наконец, в последней главе, всмотревшись в свой хрустальный шар, я попробую предсказать, какие тенденции будут набирать силу в ближайшие месяцы и годы, а также попытаюсь указать те из них, что заслуживают внимания. Это позволит вам вырваться вперед – туда, где и должен находиться любой хороший разработчик.

Основы

Теперь вы готовы к тому, чтобы узнать, как наилучшим образом применить Java и XML. Что вам нужно? Я коснусь этой темы, покажу основные направления, и дальше такими вопросами вы будете заниматься самостоятельно.

Операционная система и Java

В эти слова я вкладываю иронию; если вы считаете, что эту книгу можно прочесть, не имея установленной операционной системы и Java-системы, она может оказаться выше вашего понимания. Поэтому стоит сказать, чего я ожидаю. Я написал первую половину этой книги и примеры для этих глав на машине с Windows 2000, на которой были установлены и JDK 1.2 и JDK 1.3 (а также версия 1.3.1). Большая часть была скомпилирована мной под Cygwin (среда от Cygnus), поэтому обычно я работал в Unix-подобной среде. Вторая половина книги была написана на моем новом Macintosh G4 с операционной системой X. Эта система поставляется с JDK 1.3, и она просто чудо, если хотите знать.

В любом случае все примеры должны работать с Java 1.2 и выше – я не использовал никаких возможностей из JDK 1.3. Правда, я не старался написать код так, чтобы он компилировался в Java 1.1, поскольку мне кажется важным применение классов коллекций Java 2. Кроме того, тем, кто работает с XML и по-прежнему пользуется версией JDK 1.1, потребуется серьезно заняться обновлением JDK (я знаю, не у всех есть выбор). Если вы находитесь в жесткой зависимости от версии 1.1, то можете получить набор классов от Sun (*http://java.sun.com*), внести небольшие изменения, после чего все замечательно заработает.

Анализатор

Вам понадобится XML-анализатор. Одним из наиболее важных компонентов любого XML-ориентированного приложения является анализатор XML. Этот компонент выполняет важную задачу – он получает на входе необработанный XML-документ и придает ему смысл. Анализатор проверяет корректность исходного документа, а если документ ссылается на DTD или схему, может проверить и действительность документа. Результатом анализа XML-документа обычно является структура данных, с которой можно работать при помощи других XML-инструментов или Java API. Подробное обсуждение этих интерфейсов приводится в последующих главах. Пока же достаточно понимать, что анализатор – это один из ключевых элементов в применении XML-данных.

Выбор анализатора – непростая задача. Не существует жестких и простых правил, но обычно учитываются два основных критерия. Первый – быстродействие анализатора. Поскольку XML-документы используются все чаще, а их сложность постоянно растет, быстродействие анализатора становится важным фактором и может сильно повлиять на производительность всего приложения. Второй критерий – соответствие спецификации XML. Поскольку производительность зачастую имеет более высокий приоритет, чем отдельные несущественные возможности XML, некоторые анализаторы могут в чем-то пренебрегать требованиями спецификаций XML в целях увеличения производительности. В зависимости от требований к приложению следует выбирать разумный компромисс между этими двумя критериями. Кроме того, большинство анализаторов (но не все) позволяют проверять действительность документа. Другими словами, они могут проверить действительность документа в соответствии с указанным DTD или схемой. Убедитесь, что анализатор способен проверять действительность документов, если эта возможность необходима вашим приложениям.

Ниже приведен список наиболее распространенных анализаторов XML. В списке не отмечено, позволяет ли анализатор проверять действительность, т. к. в настоящее время ведутся работы по включению этой возможности в некоторые инструменты, в которых она отсутствует. Этот список не является рейтингом перечисленных инструментов, о каждом анализаторе можно найти массу информации на указанных веб-страницах:

- Apache Xerces: *http://xml.apache.org*
- IBM XML4J: *http://alphaworks.ibm.com/tech/xml4j*
- XP Джеймса Кларка: *http://www.jclark.com/xml/xp*
- Oracle XML Parser: *http://technet.oracle.com/tech/xml*
- Sun Microsystems Crimson: *http://xml.apache.org/crimson*
- Анализаторы Lark и Larval Тима Брэя: *http://www.textuality.com/Lark*
- Electric XML от Mind Electric: *http://www.themindelectric.com/products/xml/xml.html*
- Анализатор MSXML от Microsoft: *http://msdn.microsoft.com/xml/default.asp*

Внимание

Я включил в этот список анализатор MSXML от Microsoft из уважения к их попыткам исправить в последних версиях многочисленные проблемы с совместимостью. Однако их анализатор по-прежнему стремится «делать то, что ему вздумается», и поэтому нет гарантий, что он будет работать с примерами из этой книги. Обратитесь к нему при необходимости, но будьте готовы проделать дополнительную работу, если примете это решение.

Работая над этой книгой, я старался использовать свободно доступный анализатор Apache Xerces. Открытость кода для меня является большим плюсом, поэтому если вы еще не выбрали анализатор, рекомендую попробовать Xerces.

Интерфейсы прикладного программирования

После того как проблема выбора анализатора будет решена, вам понадобятся различные интерфейсы прикладного программирования, обсуждаемые ниже (высокоуровневые и низкоуровневые). Некоторые из них войдут в состав дистрибутива загружаемого вами анализатора, другие же придется загрузить самостоятельно. Будем считать, что либо вы ими уже располагаете, либо сможете получить их, для чего необходимо убедиться в наличии доступа к Интернету, прежде чем погружаться в чтение последующих глав.

Сначала перечислим низкоуровневые интерфейсы: SAX, DOM, JDOM и JAXP. Первые два (SAX и DOM) должны входить в состав любого загружаемого анализатора, т. к. они основаны на интерфейсе и будут реализованы в анализаторе. С большинством анализаторов вы также получите JAXP, хотя у вас может оказаться старая версия; можно надеяться, что к моменту выхода этой книги большинство анализаторов будут полностью поддерживать JAXP 1.1 (последняя вышедшая версия). В настоящее время JDOM загружается отдельно, и его можно получить с веб-сайта *http://www.jdom.org*.

Что касается высокоуровневых API, то некоторые альтернативы представлены в главе о связывании данных. Вкратце рассмотрены Castor и Zeus, доступные на сайтах *http://castor.exolab.org* и *http://zeus.enhydra.org* соответственно. Также я уделю некоторое время JAXB, доступному на сайте *http://java.sun.com/xml/jaxb*. Каждый пакет полностью функционален, загрузка дополнительных пакетов не требуется.

Приложения

Последними в списке стоят мириады особых технологий, о которых пойдет речь на страницах книги. К этим технологиям относятся, в том числе, наборы инструментов SOAP, валидаторы[1] WSDL, система веб-публикации Cocoon и т. д. Вместо того чтобы пытаться рассматривать каждую из них здесь, я коснусь конкретных приложений в соответствующих главах и расскажу о том, где можно получить эти пакеты, какие версии нужны, коснусь вопросов установки и всего прочего, что вам понадобится для установки и работы. Я могу избавить вас от всех неприятных подробностей и направить занудство только на тех, кто

[1] Не то же самое, что «validating parsers». На практике вполне может быть, что «validator» проверку действительности производит, а анализировать (parse) не умеет. – *Примеч. науч. ред.*

сделал этот выбор осознанно (шучу! я попытаюсь сделать чтение возможно более увлекательным). В любом случае продолжайте читать и изучайте все, что вам нужно знать.

В некоторых случаях я буду опираться на примеры из предыдущих глав. Например, если вы начнете читать главу 6 до того, как познакомитесь с главой 5, то, вероятно, немного запутаетесь. Если это случится, просто вернитесь обратно к нужной главе, и вы увидите, где берет начало смутивший вас код. Как уже говорилось, можно пропустить главу 2 об основах XML, но я рекомендую последовательно прочесть всю книгу, т. к. материал выстроен по нарастающей.

Что дальше?

Теперь вы, вероятно, готовы продолжить. Следующая глава представляет собой ускоренный курс по XML. Тем, кто не знает XML или нетвердо владеет его основами, эта глава поможет восполнить пробелы. Если же вы имеете опыт работы с XML, я рекомендую пропустить эту главу и перейти к программам из главы 3. В любом случае готовьтесь погрузиться в мир Java и XML; начинаются интересные вещи.

- *Основы*
- *Ограничения*
- *Преобразования*
- *И еще...*
- *Что дальше?*

2

Основы технологии

Введения остались позади, и можно перейти к делу. Но прежде чем коснуться непосредственно Java, необходимо рассказать о некоторых базовых структурах. Это касается фундаментального понимания концепций XML и механизма работы расширяемого языка разметки. Другими словами, нам нужен учебник по XML для начинающих. Что касается экспертов в XML, им стоит просмотреть эту главу, чтобы убедиться, что рассматриваемые темы им знакомы. Те же, кто совершенно не знает XML, могут здесь подготовиться к пониманию остального материала, причем довольно быстро.

По ходу чтения книги эту главу можно использовать в качестве глоссария. Далее я не буду тратить время на разъяснение понятий XML, чтобы сконцентрироваться на Java и перейти к некоторым более продвинутым концепциям. Поэтому если что-то полностью вас запутает, обратитесь за информацией к этой главе. Если же вам по-прежнему будет не все ясно, то очень рекомендую при чтении этой книги держать поблизости открытым справочник Элиотта Харольда (Elliotte Rusty Harold) и Скотта Минса (W. Scott Means) «XML in a Nutshell», O'Reilly.[1] Из него вы почерпнете всю необходимую информацию об основах XML, а я смогу сосредоточиться на Java.

Наконец, я щедр на примеры и собираюсь заполнить ими оставшиеся главы как можно плотнее. Считаю, что лучше предоставить избыточную информацию, чем лишь раздразнить аппетит читателя. Для начала, в соответствии со сказанным, приведу в этой главе несколько документов XML и связанных с ними, чтобы проиллюстрировать концепции начального курса. Можно, затратив время, набрать эти примеры

[1] Гарольд Э., Минс С. «XML. Справочник». – Пер. с англ. – СПб: Символ-Плюс, 2002.

Куда делись все главы?

Читатели первого издания книги «Java & XML» могут немного смутиться. В том издании одному только XML были посвящены целых три главы (пересчитайте!). Когда больше года назад я работал над первым изданием, передо мной стояла цель написать книгу частично об XML, частично о Java, но не рассматривающую целиком ни то ни другое. Тогда не было другого надежного ресурса, к которому можно было адресовать читателей за дополнительной помощью. Сегодня же такие книги, как «Learning XML» (Eric T. Ray, O'Reilly 2001)[1] и «XML. Справочник», решили эту проблему. Именно поэтому здесь достаточно привести лишь краткий курс по XML, а все подробности по «чистому» XML можно найти в этих прекрасных книгах. В результате мне удалось собрать несколько глав в этой одной, вымостив путь для новых глав по Java, которые, я уверен, и есть именно то, что вам надо! Так что подготовьтесь к некоторым радикальным отклонениям от первого издания; по крайней мере теперь вы знаете их причину.

в текстовом редакторе, а можно загрузить их с веб-сайта этой книги (*http://www.newInstance.com*), т. к. они будут использоваться в этой и всех последующих главах. Это позволит вам сэкономить время впоследствии.

Основы

Все начинается с рекомендации консорциума W3C XML 1.0, полный текст которой можно найти по адресу *http://www.w3.org/TR/REC-xml*. В примере 2.1 приведен простой XML-документ, соответствующий этой спецификации и представляющий собой часть оглавления этой книги в формате XML (я включил лишь часть только из-за того, что оно длинное!). Полную версию файла можно найти в примерах из книги по адресу *http://www.oreilly.com/catalog/javaxml2* и *http://www.newInstance.com*. Мы будем обращаться к нему для того, чтобы проиллюстрировать несколько важных концепций.

Пример 2.1. Документ context.xml

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">

<!-"Java и XML", Оглавление -->
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
>
```

[1] Эрик Т. Рэй «Изучаем XML». – Пер. с англ. – СПб: Символ-Плюс, 2001.

```
  <title ora:series="Java">Java и XML</title>

  <!-- Список глав -->
  <contents>
    <chapter title="Введение" number="1">
      <topic name="XML имеет значение" />
      <topic name="Что важно " />
      <topic name="Основы" />
      <topic name="Что дальше?" />
    </chapter>
    <chapter title="Основы технологии" number="2">
      <topic name="Основы" />
      <topic name="Ограничения" />
      <topic name="Преобразования" />
      <topic name="И далее..." />
      <topic name="Что дальше?" />
    </chapter>
    <chapter title="SAX" number="3">
      <topic name="Подготовка" />
      <topic name="SAX-совместимые анализаторы" />
      <topic name="Обработчики содержания" />
      <topic name="Советы разработчикам" />
      <topic name="Что дальше?" />
    </chapter>
    <chapter title="Расширенный SAX" number="4">
      <topic name="Свойства и возможности" />
      <topic name="Другие обработчики" />
      <topic name="Фильтры и классы Writers" />
      <topic name="И снова обработчики" />
      <topic name="Советы разработчикам" />
      <topic name="Что дальше?" />
    </chapter>
    <chapter title="DOM" number="5">
      <topic name="Объектная модель документа" />
      <topic name="Сериализация" />
      <topic name="Подверженность изменениям (mutability)" />
      <topic name="Советы разработчикам" />
      <topic name="Что дальше?" />
    </chapter>

    <!-- и т.д... -->

  </contents>

  <ora:copyright>&OReillyCopyright;</ora:copyright>
</book>
```

XML 1.0

Значительная часть спецификации описывает то, что, как правило, понятно интуитивно. Тем, кто когда-либо занимался созданием документов HTML или SGML, уже знакомо понятие элементов (таких как

contents и chapter в этом примере) и атрибутов (таких как title и name). В XML определению правил использования этих элементов и структурирования документа уделено не так много внимания. Гораздо больше времени тратится на определение таких сложных моментов, как интерпретация пробельных символов, чем на введение понятий, с которыми вы уже хоть как-то знакомы.

XML-документ можно разбить на две основные части: пролог, предоставляющий XML-анализатору и приложениям XML информацию о том, как обрабатывать документ, и содержание, представляющее собой собственно XML-данные. И хотя такое деление довольно условно, оно помогает различать инструкции для приложений из XML-документа от непосредственно XML-содержимого, и это отличие важно понимать. Пролог – это просто объявление XML в следующем формате:

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

Пролог, кроме того, может включать информацию о кодировке и определять, является ли документ автономным или же его интерпретация требует ссылки на другие документы:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

Оставшуюся часть пролога составляют элементы, подобные объявлению DOCTYPE:

```
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">
```

В данном случае я ссылаюсь на файл *JavaXML.dtd*, находящийся на моей локальной системе в каталоге *DTD/*. Каждый раз при использовании относительного или абсолютного пути к файлу либо URL следует применять ключевое слово SYSTEM. Другая возможность – использовать ключевое слово PUBLIC, за которым следует публичный[1] идентификатор. Это означает, что W3C или иной консорциум определил стандартный DTD, а также общедоступное имя для DTD. В качестве примера рассмотрим инструкцию DTD для XHTML 1.0:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

В данном случае за публичным идентификатором (странная короткая строчка, начинающаяся с «-//») следует системный идентификатор (URL). Если публичный идентификатор не может быть преобразован в системный средствами анализатора, вместо него используется явно заданный системный идентификатор.

---

[1] Свойства идентификаторов public/system (публичный/системный) из контекста объявлений XML не следует путать с модификаторами доступа public/private/protected (открытый/закрытый/защищенный) из ООП. – *Примеч. науч. ред.*

Также в начале файла можно увидеть инструкции обработки, которые обычно считаются частью пролога документа, а не его содержимого. Выглядят они примерно так:

```
<?xml-stylesheet href="XSL\JavaXML.html.xsl" type="text/xsl"?>
<?xml-stylesheet href="XSL\JavaXML.wml.xsl" type="text/xsl"
                 media="wap"?>
<?cocoon-process type="xslt"?>
```

Каждая из них имеет *цель* (*target*), определяемую первым словом, в данном примере – `xml-stylesheet` или `cocoon-process`, и *данные* (*data*) – все остальное. Чаще всего данные представлены в виде пар имя–значение, которые заметно улучшают читаемость. Правда, это лишь необязательное правило хорошего тона, поэтому на такой формат не следует рассчитывать.

Вся остальная часть XML-документа представляет его содержимое; другими словами – это элементы, атрибуты и данные, помещаемые в документ.

Корневой элемент

Корневой элемент – это элемент высшего уровня в XML-документе. Он должен быть первым открывающим и последним закрывающим тегом в документе. Он обеспечивает точку отсчета, позволяющую XML-анализатору или XML-ориентированному приложению распознавать начало и конец XML-документа. В нашем примере корневым является элемент `book`:

```
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
>

    <!--Содержимое документа -->
</book>
```

Этот тег и парный ему закрывающий тег охватывают все остальные данные в пределах этого XML-документа. Стандарт XML гласит, что в документе может быть только один корневой элемент. Другими словами, корневой элемент должен заключать все прочие элементы документа. Помимо этого ограничения корневой элемент ничем не отличается от любого другого элемента XML. Это очень важно понимать, поскольку XML-документы могут ссылаться на другие XML-документы или включать их. В таких случаях корневой элемент включаемого документа становится вложенным элементом основного документа и должен быть обработан XML-анализатором обычным образом. Определение корневых элементов в качестве стандартных элементов XML, не имеющих особых свойств и не обладающих особым поведением, обеспечивает отсутствие проблем при включении документов.

Элементы

До сих пор я избегал определения собственно элемента. Давайте теперь подробно рассмотрим элементы, которые представляются произвольными именами и должны заключаться в угловые скобки. В приведенном ниже примере показано несколько различных видов элементов:

```
<!-- Стандартный открывающий тег элемента -->
<contents>

 <!-- Стандартный элемент с атрибутом -->
 <chapter title="Основы технологии" number="2">

 <!-- Элемент с текстовыми данными -->
 <title ora:series="Java">Java и XML</title>

 <!-- Пустой элемент -->
 <sectionBreak />

<!-- Стандартный закрывающий тег элемента -->
</contents>
```

Первое правило создания элемента: его имя должно начинаться с буквы или символа подчеркивания и может содержать любое количество букв, цифр, символов подчеркивания, дефисов или точек. Имена элементов не могут содержать пробелы:

```
<!-- Внедренные пробелы недопустимы -->
<my element name>
```

Кроме того, имена элементов XML чувствительны к регистру. Обычно применение тех же правил, которые приняты в Java для именования переменных, приводит к созданию разумных имен элементов XML. Использование элемента под названием `tcbo` для представления бизнес-объекта телекоммуникационной индустрии (Telecommunications Business Object) – не очень хорошая идея, поскольку смысл имени не очевиден, а чрезмерно многословное имя тега, вроде `beginningOfNewChapter`, лишь загромождает документ. Помните, что ваши XML-документы, вероятно, попадут в руки других разработчиков и авторов, так что документирование посредством создания разумных имен весьма важно.

Каждый открытый элемент, в свою очередь, должен быть закрыт. Из этого правила нет исключений, которые встречаются во многих других языках разметки, например в HTML. Закрывающий тег для элемента состоит из прямого слэша и имени элемента: `</content>`. Между открывающим и закрывающим тегами может присутствовать любое количество элементов или произвольный объем текстовых данных. Однако нельзя нарушать порядок следования вложенных тегов: элемент, открытый первым, должен быть закрыт последним. Если в XML-документе нарушается хотя бы одно синтаксическое правило XML, такой документ не является *корректным* (*well-formed*). Кор-

ректный документ – это документ, в котором соблюдаются все синтаксические правила XML, а все элементы и атрибуты расположены допустимым образом. Тем не менее, корректный документ не обязательно является *действительным* (*valid*), что означает соответствие документа ограничениям, устанавливаемым DTD или схемой. Между корректными и действительными документами существует значительная разница. Выполнение правил, о которых идет речь в этом разделе, гарантирует корректность документа, в то время как правила, обсуждаемые в разделе, посвященном ограничениям, определяют действительность документа.

В качестве примера неверно построенного документа рассмотрим следующий фрагмент кода XML:

```
<tag1>
 <tag2>
</tag1>
 </tag2>
```

Порядок вложенности тегов нарушен, поскольку за открывающим тегом `<tag2>` в пределах тега `<tag1>` не следует закрывающий тег `</tag2>`. Однако нет никакой гарантии, что документ станет действительным, если исправить эти синтаксические ошибки.

Хотя этот пример неверно построенного документа может показаться тривиальным, вспомните, что такая разметка была бы допустимой в HTML, и подобное часто встречается в больших таблицах HTML-документов. Другими словами, HTML и многие другие языки разметки не требуют XML-корректности.[1] Строгое соблюдение упорядоченности и правил вложенности в XML позволяет анализировать и обрабатывать данные гораздо быстрее, чем при использовании языков разметки, не имеющих подобных ограничений.

Последнее правило, которое мы рассмотрим, касается случая пустых элементов. Мы уже знаем, что теги XML всегда должны существовать в парах – открывающий тег и закрывающий тег вместе составляют полный XML-элемент. Существуют ситуации, когда элемент используется сам по себе, например в качестве признака того, что глава еще не дописана, или когда элемент имеет атрибуты, но не имеет текстовых данных, подобно ссылке на изображение в HTML.[2] Подобные элементы придется записывать следующим образом:

```
<chapterIncomplete></chapterIncomplete>
<img src="/images/xml.gif"></img>
```

---

[1] Документы XHTML, которые предполагается обрабатывать с помощью XML-анализатора, также должны быть XML-корректными. – *Примеч. науч. ред.*

[2] Речь идет о теге `<IMG>`. – *Примеч. науч. ред.*

Выглядит это несколько глупо, да еще и добавляет путаницу в XML-документ, который и без того может быть довольно объемным. Спецификация XML определяет средство обозначить оба тега – и открывающий и закрывающий – внутри одного элемента:

```
<chapterIncomplete />
<img src="/images/xml.gif" />
```

Бретт, а что это за пробел перед последним слэшем?

Что ж, я расскажу. На мою долю выпало сомнительное удовольствие работать с Java и XML с 1998 года, когда дела обстояли в лучшем случае не очень гладко. В то время (да и сейчас) некоторые веб-броузеры понимали только XHTML (корректный HTML) в специальном формате. Наиболее заметен тот факт, что такие теги, как `<br>`, которые никогда не закрываются в HTML, в XHTML должны быть закрыты, т. е. появляются теги, подобные `<br/>`. Некоторые из упомянутых броузеров игнорируют подобные теги; однако, как это ни странно, они успешно обрабатывают теги `<br  />` (обратите внимание на пробел перед закрывающим слэшем). Я привык делать XML-код не только корректным, но и понятным для таких броузеров. У меня никогда не было веской причины менять эти привычки, так что здесь вы видите их в действии.

Итак, мы видим элегантное решение проблемы замусоривания документов, позволяющее оставаться верным правилу, что каждый элемент в XML должен иметь закрывающий тег. Открывающий и закрывающий теги просто объединяются в один.

Атрибуты

Помимо текста, заключенного в теги элемента, элемент также может иметь атрибуты. Атрибуты вместе со своими значениями включаются в открывающий тег элемента (который по совместительству может быть и закрывающим). Например, в теге `chapter` название главы представляет собой часть атрибута:

```
<chapter title="Расширенный SAX" number="4">
  <topic name="Свойства и возможности" />
  <topic name="Другие обработчики" />
  <topic name="Фильтры и классы Writer" />
  <topic name="И снова обработчики" />
  <topic name="Советы разработчикам" />
  <topic name="Что дальше?" />
</chapter>
```

В данном случае `title` – это имя атрибута, значением которого является название главы, т. е. «Расширенный SAX». Имена атрибутов должны удовлетворять правилам для имен элементов XML, а значения атрибутов необходимо заключать в кавычки. Несмотря на то что допускается применение как одинарных, так и двойных кавычек, использование последних – широко распространенная практика, которая приводит к созданию XML-документов, соответствующих стилю программирования на Java. Кроме того, одинарные и двойные кавычки могут присутствовать в значениях атрибутов. Если заключить значение в двойные кавычки, одинарные кавычки могут выступать как часть значения, и наоборот, если заключить значение в одинарные кавычки, то как часть значения смогут использоваться двойные. Однако это нельзя считать хорошим стилем, поскольку анализаторы и процессоры XML часто преобразуют кавычки, ограничивающие значение атрибута, в двойные (или в одинарные), что может привести к неожиданным результатам.

Помимо вопроса о том, как использовать атрибуты, существует также вопрос о том, когда это уместно. Поскольку XML допускает множество вариантов форматирования данных, редко когда атрибут не может быть представлен элементом и редко когда элемент не может быть легко преобразован в атрибут. И хотя не существует спецификации или широко принятого стандарта, позволяющих определить, когда следует предпочесть ту или иную форму, есть хорошее практическое правило: элементы подходят для разметки сегментов данных, состоящих из нескольких значений, а атрибуты – для разметки сегментов из единственного значения. Если определенный логический сегмент данных состоит из нескольких значений либо имеет большую длину, то вероятнее всего он должен оформляться в виде элемента. Впоследствии эти данные можно будет рассматривать прежде всего как текстовые, которые легко поддаются поиску и обработке. Примеры данных такого рода: описание глав книги или URL-адреса, содержащиеся в соответствующих ссылках на сайте. Если же данные главным образом представимы в виде единственного значения, их лучше всего представить в виде атрибута. Вероятным кандидатом на оформление в виде атрибута является раздел главы; тогда как сам раздел может быть элементом и иметь собственное название, группа глав внутри раздела может быть легко представлена атрибутом `section` элемента `chapter`. Этот атрибут позволит легко группировать и индексировать главы, но он никогда не будет отображаться для пользователя. Еще один хороший пример данных, которые могут быть представлены в XML в виде атрибута, – это указание, зарезервирован ли некий товар покупателем. Такое указание позволит XML-приложению, которое используется для создания брошюры или рекламного листка, определить, какие из имеющихся на складе товаров следует включать в документ. Очевидно, что это будет либо значение «истина», либо «ложь», и единовременно атрибут может принимать только одно из них. Опять же, клиент никогда не

увидит этой информации, но данные будут использоваться при обработке XML-документа. Если подобный анализ все-таки не дал четкой картины, остановитесь на самом надежном варианте – примените для оформления элемент.

Может быть, вам уже пришли в голову альтернативные подходы к представлению рассмотренных примеров. Например, вместо того чтобы использовать атрибут `title`, возможно, имело бы смысл включить элементы `title` в элемент `chapter`. Не исключено, что пустой тег `<layaway />` был бы более подходящим способом отметить, что предмет мебельного гарнитура зарезервирован. В XML очень редко существует только один способ представить данные, и, как правило, есть несколько хороших способов решения одной и той же задачи. Чаще всего решение определяется приложением и перспективой применения данных. Вместо того чтобы рассказывать, как создавать XML-документы, что было бы сложно, я покажу, как применять XML, чтобы вы могли получить представление о том, как можно обрабатывать и использовать различные форматы данных. Это обеспечит вас знаниями, необходимыми для принятия собственных решений о форматировании XML-документов.

Константы и ссылки на сущности

Я не обсудил еще вопрос, касающийся маскировки символов, или ссылки на константы. Например, распространенным способом указания пути к каталогу установки является элемент `<path-to-Cocoon>`. Здесь пользователь должен заменить этот текст подходящим вариантом каталога установки. В данном примере в главе, рассказывающей о веб-приложениях, должны быть приведены некоторые подробности об установке и применении Apache Cocoon, и может потребоваться представить эти данные внутри элемента:

```
<topic>
 <heading>Установка Cocoon</heading>
 <content>
  Найдите файл Cocoon.properties в каталоге <path-to-Cocoon>/bin.
 </content>
</topic>
```

Неприятность заключается в том, что XML-анализаторы пытаются обработать эти данные как тег XML, а затем выдают сообщение об ошибке, поскольку закрывающий тег отсутствует. Это распространенная проблема, поскольку к такому исходу приводит любое использование угловых скобок. Справиться с этим позволяет применение *ссылок на сущности* (*entity references*). Ссылка на сущность – это специальный тип данных XML, используемый для адресации некоторой совокупности данных. Ссылка на сущность состоит из уникального имени, перед которым стоит символ &, а после – точка с запятой: `&[имя сущности];`. Когда анализатор XML встречает ссылку на сущность, ссылка заменяется определенным значением, и это значение не обрабатывается.

В XML определено пять сущностей, которые позволяют решить проблему, представленную в примере: < для знака «меньше», > для знака «больше», & для символа амперсанда (&), " для двойной кавычки, а также ' для одинарной кавычки или апострофа. Посредством этих специальных ссылок можно точно представить ссылку на каталог установки следующим образом:

```
<topic>
 <heading>Установка Cocoon</heading>
 <content>
  Найдите файл Cocoon.properties в каталоге &lt;path-to-Cocoon&gt;/bin.
 </content>
</topic>
```

Когда этот документ обрабатывается анализатором, данные интерпретируются как «<path-to-Cocoon>», и документ считается корректным.

Имейте в виду, что ссылки на сущности могут определяться пользователем. Это делает возможным применение некоего подобия сокращения разметки: в рассмотренном примере XML-кода есть ссылка на внешний, совместно используемый (shared) документами текст с информацией об авторских правах. Поскольку этот текст присутствует во многих книгах издательства O'Reilly, я не хочу включать его в XML-документ. Тем не менее, если текст с информацией о правообладании изменится, изменения должны быть отражены в XML-документе. Обратите внимание, что синтаксис, применяемый в этом случае, похож на синтаксис для ссылок на стандартные сущности XML:

```
<ora:copyright>&OReillyCopyright;</ora:copyright>
```

Мы еще не добрались до раздела о DTD, поэтому пока не знаем, каким образом объяснить XML-анализатору, на что указывает ссылка &OReillyCopyright;, но вам уже должно быть понятно, что применение ссылок на сущности не ограничивается лишь вставкой в текст сложных или необычных символов.

Неанализируемые данные

Последняя конструкция XML, которую мы рассмотрим, – это маркер секции CDATA. Секция CDATA используется в тех случаях, когда вызывающему приложению необходимо передать значительный объем данных, не подлежащих обработке XML-анализатором. Это может пригодиться, если необходимо заменить ссылками на сущности необычайно большое количество символов либо когда важно сохранить расстановку пробелов. Секция CDATA в XML-документе выглядит следующим образом:

```
<unparsed-data>
  <![CDATA[Диаграмма:
       <Step 1>Установите Cocoon в каталог "/usr/lib/cocoon"
```

```
        <Step 2>Найдите верный файл свойств.
        <Step 3>Загрузите Ant с сайта "http://jakarta.apache.org"
              -----> Для этого воспользуйтесь CVS <----
  ]]>
</unparsed-data>
```

В данном примере для информации из секции CDATA нет необходимости прибегать к ссылкам на сущности или другим механизмам, сообщающим анализатору об использовании зарезервированных символов. Вместо этого XML-анализатор передает их в неизменном виде той программе (или приложению), из которой был вызван.

Итак, мы рассмотрели основные составляющие XML-документов. И хотя каждая из них была описана лишь поверхностно, вы получили достаточно информации для того, чтобы опознать теги XML, когда вы их увидите, и понять, для чего они предназначены. При наличии такой книги, как «XML in a Nutshell» (O'Reilly)[1], вы вполне готовы к тому, чтобы взглянуть на некоторые из более «продвинутых» спецификаций XML.

Пространства имен

И хотя мы не будем очень подробно рассматривать здесь пространства имен XML, обратите внимание на использование пространства имен в корневом элементе в примере 2.1. *Пространство имен XML* – это способ связывания одного или более элементов из XML-документа с конкретным URI (Uniform Resource Identifier). На деле это означает, что элемент идентифицируется как своим именем, так и URI пространства имен. В приведенном примере XML-документа позже может понадобиться включить части других книг издательства O'Reilly. Поскольку каждая из этих книг также может иметь элементы Chapter, Heading или Topic, документ следует проектировать и создавать таким образом, чтобы избежать конфликтов пространства имен с другими документами. Спецификация пространств имен XML элегантно решает эту проблему. Поскольку наш XML-документ представляет конкретную книгу и никакой другой XML-документ ту же самую книгу представлять не должен, использование уникального URI, например *http://www.oreilly.com/javaxml2*, позволяет создать уникальное пространство имен. Спецификация пространств имен требует, чтобы с префиксом был связан уникальный URI, который позволит отличать элементы данного пространства имен от элементов других пространств имен. Рекомендуется использовать URL, как мы здесь и поступаем:

```
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
>
```

[1] Элиот Расти Гарольд, У. Скотт Минс. «XML. Справочник». – Пер. с англ. – СПб: Символ-Плюс, 2002.

На самом деле здесь определены два пространства имен. Первое считается пространством имен по умолчанию, поскольку для него не задается префикс. Любой элемент без префикса входит в это пространство имен. В результате все элементы из этого XML-документа, за исключением элемента `copyright`, предваряемого префиксом `ora`, принадлежат пространству имен по умолчанию. Второе пространство имен определяет префикс, что позволяет связать тег `<ora:copyright>` со вторым пространством имен.

И наконец, еще один интересный (и несколько сбивающий с толку) момент: спецификация схем XML, о которой подробнее рассказано в следующем разделе, требует указания схемы XML-документа таким образом, который очень похож на набор объявлений пространств имен (см. пример 2.2).

Пример 2.2. Обращение к схеме XML

```
<?xml version="1.0"?>
<addressBook xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
             xmlns="http://www.oreilly.com/catalog/javaxml"
             xsi:schemaLocation="http://www.oreilly.com/catalog/javaxml
                                 mySchema.xsd"
>
  <person>
    <name>
      <firstName>Бретт</firstName>
      <lastName>Мак-Лахлин</lastName>
    </name>
    <email>brettmclaughlin@earthlink.net</email>
  </person>
  <person>
    <name>
      <firstName>Эдди</firstName>
      <lastName>Балуччи</lastName>
    </name>
    <email>eddieb@freeworld.net</email>
  </person>
</addressBook>
```

Здесь важно понять несколько моментов. Во-первых, тут определяется и связывается с URL пространство имен экземпляра схемы XML. Это пространство имен, сокращенно `xsi`, служит для указания в XML-документах информации о схеме в точности так, как это сделано в данном случае. Таким образом, в первой строке элементы экземпляра схемы XML становятся доступными в документе. Следующая строка определяет пространство имен для самого XML-документа. Поскольку в документе пространство имен явно не используется, как это было в предыдущих примерах для пространства имен, связанного с *http://www.oreilly.com/javaxml2*, то объявляется пространство имен по умолчанию. Это означает, что все элементы без явного указания простран-

ства имен и соответствующего префикса (все элементы в этом примере) будут принадлежать пространству имен по умолчанию.

Определив таким образом пространство имен как для документа, так и для экземпляра схемы XML, можно далее делать то, что мы хотим, т. е. связать с этим документом какую-либо схему. Для этого воспользуемся атрибутом `schemaLocation`, который принадлежит пространству имен экземпляра схемы XML. Я предваряю этот атрибут идентификатором его пространства имен (`xsi`), которое было только что определено. Аргументом этого атрибута на самом деле являются *два* URI: первый определяет пространство имен, связанное со схемой, а второй – URI схемы, на которую мы ссылаемся. В результате в данном примере первый URI представляет собой только что объявленное пространство имен по умолчанию, а второй – файл *mySchema.xsd* в локальной файловой системе. Как и любой другой XML-атрибут, эта пара целиком заключается в одну пару кавычек. Вот так запросто можно указать схему для XML-документа!

На самом деле это совсем не просто и на сегодняшний день является одним из наиболее сложных для восприятия моментов в использовании пространств имен и схем XML. Чуть позже мы подробнее рассмотрим используемый здесь механизм. А пока запомните, каким образом пространства имен допускают обращение к элементам из различных наборов, сохраняя информацию о принадлежности каждого из элементов к определенной группе.

Ограничения

Следующая задача – это работа с ограничениями XML. Даже если вы ничего не вынесете из этой главы, кроме логического обоснования необходимости ограничения данных XML, то я буду счастливым автором. Поскольку XML – это расширяемый язык, позволяющий представлять данные сотнями и даже тысячами способов, ограничения, налагаемые на документ, придают этим различным форматам смысл. Без ограничений документа невозможно (в большинстве случаев) определить смысл данных в документе. В этом разделе представлены два из существующих механизмов наложения ограничений на XML-данные: DTD (включены в спецификацию XML 1.0) и XML-схемы (недавно опубликованный стандарт от W3C). Выбирайте тот механизм, который больше вам подходит.

Определение типа документа (DTD)

XML-документ не слишком полезен без сопутствующего DTD (или схемы). Точно так же, как XML может эффективно описывать данные, так DTD, определяя структуру данных, обеспечивает возможность использования этих данных самыми разными способами во множестве

разнообразных программ. В этом разделе рассмотрены наиболее распространенные конструкции, используемые в DTD. Мы вновь возьмем в качестве примера часть содержания этой книги, представленного в формате XML, и пройдем через процесс конструирования DTD для этого документа.

DTD определяет формат данных. Должны быть определены: каждый допустимый в XML-документе элемент, допустимые атрибуты и, возможно, приемлемые значения атрибутов для каждого элемента, иерархия и вхождения элементов, а также все внешние сущности. DTD может определять и многие другие вещи, касающиеся XML-документа, но мы сконцентрируем внимание на перечисленных основных вопросах. Конструкции, определяемые в DTD, можно изучить, применяя их к XML-файлу из примера 2.1 и налагая на него ограничения. Полностью DTD приведено в примере 2.3, и я буду ссылаться на него в этом разделе.

Пример 2.3. DTD для примера 2.1

```
<!ELEMENT book (title, contents, ora:copyright)>
<!ATTLIST book
          xmlns       CDATA  #REQUIRED
          xmlns:ora   CDATA  #REQUIRED
>
<!ELEMENT title (#PCDATA)>
<!ATTLIST title
          ora:series  (C | Java | Linux | Oracle |
                      Perl | Web | Windows)
                      #REQUIRED
>
<!ELEMENT contents (chapter+)>
<!ELEMENT chapter (topic+)>
<!ATTLIST chapter
          title       CDATA  #REQUIRED
          number      CDATA  #REQUIRED
>
<!ELEMENT topic EMPTY>
<!ATTLIST topic
          name        CDATA  #REQUIRED
>

<!--Информация об авторских правах -->
<!ELEMENT ora:copyright (copyright)>
<!ELEMENT copyright (year, content)>
<!ATTLIST copyright
          xmlns  CDATA  #REQUIRED
>
<!ELEMENT year EMPTY>
<!ATTLIST year
          value  CDATA  #REQUIRED
>
```

```
<!ELEMENT content (#PCDATA)>
<!ENTITY OReillyCopyright SYSTEM
   "http://www.newInstance.com/javaxml2/copyright.xml"
>
```

Элементы

Большая часть DTD состоит из определений ELEMENT (рассматриваются в этом разделе) и определений ATTRIBUTE (рассматриваются в следующем разделе). Определение элемента начинается с ключевого слова ELEMENT, следующего за стандартным открывающим тегом DTD <!. После ключевого слова указывается имя элемента. За этим именем следует *модель содержимого* элемента. Модель содержимого, как правило, заключается в скобки и определяет, какое содержимое может иметь элемент. Рассмотрим в качестве примера элемент book:

```
<!ELEMENT book (title, contents, ora:copyright)>
```

Данное определение гласит, что внутри любого элемента book могут существовать элементы title, contents и ora:copyright. Для каждого из них в тексте DTD определяется модель содержимого, и так далее для всех существующих элементов. Необходимо помнить, что порядок следования элементов в документе – для данного стандартного случая – жестко определяется моделью содержимого. Кроме того, если не используются модификаторы (я рассмотрю их немедленно), каждый элемент *должен* присутствовать один и только один раз. В данном случае любой без исключения элемент book должен содержать элемент title, элемент contents, а затем элемент ora:copyright. Если эти правила нарушаются, документ не считается действительным (хотя он может оставаться корректным).

Довольно часто появляется необходимость сделать присутствие элемента необязательным либо разрешить многократное вхождение. Это можно сделать при помощи модификаторов повторения (recurrence modifiers), перечисленных в табл. 2.1.

Таблица 2.1. Модификаторы повторения в DTD

Оператор	Описание
[По умолчанию]	Должен появиться один и только один раз (1)
?	Может появиться один раз или отсутствовать (0..1)
+	Должен появиться не менее одного раза, максимальное число вхождений не ограничено (1..N)
*	Может появиться любое число раз, в том числе и ни разу (0..N)

В качестве примера рассмотрим определение элемента contents:

```
<!ELEMENT contents (chapter+)>
```

В данном случае элемент contents должен содержать по крайней мере один элемент chapter, но может содержать сколько угодно таких элементов.

Если элемент содержит символьные данные, то его модель содержимого определяется ключевым словом #PCDATA:

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

Если элемент всегда должен быть пустым, указывается ключевое слово EMPTY:

```
<!ELEMENT topic EMPTY>
```

Атрибуты

Вслед за элементами надо определить атрибуты. Это делается с помощью ключевого слова ATTLIST. Сначала требуется указать имя элемента, а затем определить различные атрибуты для него. Определение состоит из имени атрибута, его типа, а также информации, определяющей поведение анализатора в случае отсутствия атрибута. Атрибут может быть обязательным (#REQUIRED) либо подразумеваемым, необязательным (#IMPLIED). Большинство атрибутов с текстовыми значениями будут иметь тип CDATA, как показано ниже:

```
<!ATTLIST chapter
          title       CDATA  #REQUIRED
          number      CDATA  #REQUIRED
>
```

Также можно задать набор значений, которые должен принимать атрибут, чтобы документ считался действительным:

```
<!ATTLIST title
          ora:series  (C | Java | Linux | Oracle |
                      Perl | Web | Windows)
                      #REQUIRED
>
```

Сущности

При помощи ключевого слова ENTITY в DTD можно определять значения сущностей. Во многом сущности похожи на ссылку DOCTYPE, о которой я говорил ранее, т. е. можно задать публичный и/или системный идентификатор. В примере DTD я задал системный идентификатор – URL, связанный с сущностью OReillyCopyright, следующим образом:

```
<!ENTITY OReillyCopyright SYSTEM
   "http://www.newInstance.com/javaxml2/copyright.xml"
>
```

В результате файл *copyright.xml*, расположение которого определяется URL-адресом, загружается по ссылке на сущность `OReillyCopyright`. Таким образом, в примере документа XML ссылка на сущность заменяется содержимым конкретного файла. В следующих нескольких главах вы увидите, как это работает.

Приведенную информацию трудно назвать подробным справочником по DTD, но этих базовых знаний вам должно хватить, чтобы двигаться дальше. Как я уже предлагал, если вы столкнетесь с чем-то не совсем для вас очевидным, воспользуйтесь дополнительными источниками, специально посвященными XML (например, книгой «XML. Справочник»). Считая, что у вас есть эта книга или спецификация с сайта *http://www.w3.org*, я смогу раньше перейти к темам, посвященным Java.

Схема XML

Схема XML – это новый стандарт от W3C, работа над которым недавно была завершена. Схемы по смыслу не отличаются от определений DTD, но позволяют применять более жесткую типизацию и предоставляют гораздо больше архитектурных возможностей. Помимо этого схемы, в отличие от DTD, сами имеют формат XML. Я собираюсь потратить относительно немного времени на обсуждение схем, поскольку эти подробности останутся за кулисами при работе с Java и XML. В главах, затрагивающих работу со схемами (например, в главе 14), я опишу моменты, существенные для понимания происходящего. Однако спецификация XML-схем настолько объемна, что требует отдельной книги с комментариями. В примере 2.4 показана схема XML, ограничивающая пример 2.1.

Пример 2.4. Схема XML, ограничивающая пример 2.1

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://www.oreilly.com/javaxml2"
           xmlns:ora="http://www.oreilly.com"
           targetNamespace="http://www.oreilly.com/javaxml2"
           elementFormDefault="qualified"
>
  <xs:import namespace="http://www.oreilly.com"
             schemaLocation="contents-ora.xsd" />

  <xs:element name="book">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="title" />
        <xs:element ref="contents" />
        <xs:element ref="ora:copyright" />
      </xs:sequence>
    </xs:complexType>
```

```
  </xs:element>

  <xs:element name="title">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute ref="ora:series" use="required" />
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>

  <xs:element name="contents">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="chapter" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="topic" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="name"
                                type="xs:string"
                                use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="title" type="xs:string" use="required"/>
            <xs:attribute name="number" type="xs:byte" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Кроме того, по причинам, которые скоро станут очевидными, понадобится схема из примера 2.5.

Пример 2.5. Дополнительная схема для примера 2.1

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

<xs:schema xmlns="http://www.oreilly.com"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.oreilly.com"
           attributeFormDefault="qualified"
           elementFormDefault="qualified"
>
  <xs:attribute name="series" type="xs:string"/>
  <xs:element name="copyright" type="xs:string" />
</xs:schema>
```

Перед тем как мы перейдем к специфике этих схем, обратите внимание, что были объявлены различные пространства имен. Во-первых, само пространство имен XML-схемы связано с префиксом xs, что позволяет разделять конструкции XML-схемы и ограничиваемые элементы и атрибуты. Во-вторых, пространство имен по умолчанию связано с пространством имен определяемых элементов; в примере 2.4 это пространство имен книги «Java и XML», а в примере 2.5 – пространство имен «O'Reilly». Кроме того, это же значение присвоено атрибуту targetNamespace. Этот атрибут задает пространство имен ограничиваемых элементов и атрибутов для схемы. Следует проявлять бдительность и не забывать про этот атрибут, поскольку его отсутствие может привести к катастрофическим последствиям. К этому моменту определены пространства имен для ограничиваемых элементов (пространство имен по умолчанию) и для используемых конструкций (пространство имен XML-схемы).

Наконец, для атрибутов attributeFormDefault и elementFormDefault указано значение «qualified». Это означает, что для элементов и атрибутов будут использоваться полные, а не локальные имена. Я не буду вдаваться в подробности, но настоятельно рекомендую всегда работать именно с полными именами. Попытка иметь дело одновременно с несколькими пространствами имен и локальными именами заведет нас в такие дебри, что лучше от нее заранее отказаться.

Элементы и атрибуты

Элементы определяются при помощи конструкции element. Как правило, необходимо определить пользовательский тип данных с помощью тега complexType, вложенного в элемент element, который посредством атрибута name задает имя элемента. Взгляните на фрагмент из примера 2.4:

```
<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="title" />
      <xs:element ref="contents" />
      <xs:element ref="ora:copyright" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Элемент book имеет комплексное содержимое. Он должен содержать три элемента: title, contents и ora:copyright. Конструкция sequence задает жесткий порядок следования элементов; при отсутствии модификаторов элемент должен появиться один и только один раз. Для каждого из перечисленных элементов атрибут ref является ссылкой на его определение в другой части схемы. Таким образом мы получаем прозрачные, однородные определения.

Позже в файле определяется элемент title:

```
<xs:element name="title">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute ref="ora:series" use="required" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Этот элемент имеет тип string (строковый) – базовый тип XML-схемы; но поскольку происходит создание атрибута для элемента, следует определить тип complexType. Помимо этого происходит расширение существующего типа, и для определения типа используются ключевые слова simpleContent и restriction (в качестве вложенных элементов). Ключевое слово simpleContent сообщает схеме, что это простой тип, а restriction с основанием «xs:string» говорит о том, что мы намереваемся разрешить только то, что позволено XML-схемой типу string, плюс дополнительный атрибут, определяемый здесь (ключевое слово attribute). Что касается самого атрибута, мы ссылаемся на тип, определенный в другом месте, и указываем (посредством use="required"), что атрибут является обязательным для элемента. Я понимаю, что многое из сказанного в этом абзаце не совсем прозрачно, но со временем вы все поймете.

Обратите внимание на использование атрибутов minOccurs и maxOccurs для элемента element. Эти атрибуты позволяют определить количество допустимых вхождений элемента. По умолчанию элемент может присутствовать лишь один раз. Например, если заданы значения minOccurs="0" и maxOccurs="1", элемент может появиться один раз или не появиться вообще. Чтобы позволить элементу появляться бесконечное число раз, можно указать значение "unbounded" для атрибута maxOccurs, как это сделано в примере 2.4.

Несколько пространств имен

Вас мог запутать тот факт, что были определены *две* схемы. Для каждого пространства имен документа должна быть определена одна схема. Кроме того, нельзя указывать одну и ту же внешнюю схему для обоих пространств имен. Следовательно, использование префикса и пространства имен ora требует дополнительной схемы, в примере названной contents-ora.xsd. Также понадобится воспользоваться атрибутом schemaLocation, о котором говорилось ранее, чтобы сослаться на эту схему; однако не добавляйте другой атрибут. Вместо этого можно дописать еще одну пару «пространство имен–местоположение схемы» в конец значения атрибута, как показано далее:

```
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.oreilly.com/javaxml2 XSD/contents.xsd
                            http://www.oreilly.com XSD/contents-ora.xsd"
>
```

Пространство имен *http://www.oreilly.com/javaxml2* получает инструкцию искать определения в схеме под названием *contents.xsd* в каталоге *XSD/*. Для пространства имен *http://www.oreilly.com* используется схема *contents-ora.xsd* из этого же каталога. Затем надо определить две схемы, приведенные в примерах 2.4 и 2.5. Наконец, необходимо импортировать схему «O'Reilly» в схему «Java и XML», поскольку элементы из схемы «Java и XML» ссылаются на атрибуты из схемы «O'Reilly»:

```
<xs:import namespace="http://www.oreilly.com"
           schemaLocation="contents-ora.xsd" />
```

Данная операция вполне прозрачна, поэтому не будем задерживаться на этом вопросе. Следует понимать, что работа с несколькими пространствами имен – пожалуй, самое сложное, что можно делать в схемах, и она легко может вызвать затруднения. (Так было со мной, пока положение не спас Эрик ван дер Влист[1].) Мой совет – найдите хороший редактор, ориентированный на работу со XML-схемами. И хотя я обычно не тороплюсь рекомендовать коммерческие продукты, в данном случае XMLSpy 4.0 (*http://www.xmlspy.com*) оказывается на удивление полезным.

Мы лишь поверхностно рассмотрели DTD и XML-схемы, но остались еще и другие модели ограничений, о которых мы вообще не говорили! Например, Relax (и Relax NG, включающий в себя то, что раньше называлось TREX) набирает обороты, поскольку по сравнению со схемами XML он считается облегченным и гораздо более простым. Информацию о процессе разработки можно найти на сайте *http://www.oasis-open.org/committees/relax-ng/*. Неважно, какую именно технологию вы выбрали – это должно быть что-то, что поможет ограничить XML-данные. При наличии этих ограничений проверка действительности документов становится элементарной задачей, а возможности для взаимодействия приложений расширяются. Считайте, что вы все узнали об ограничениях XML и готовы перейти к следующей теме в этом стремительном обзоре – к преобразованиям XML.

Преобразования

Несмотря на всю полезность преобразований они довольно сложны в реализации. Чтобы не было необходимости включать определение преобразований XML в спецификацию XML 1.0, были выпущены три

[1] Eric van der Vlist – веб-разработчик, участвует в проекте ODP (Open Directory Project). Кроме того, он основатель, издатель и редактор сайта *XMLfr*, популяризирующего XML. – *Примеч. ред.*

отдельные рекомендации, определяющие, как должны выполняться преобразования. И хотя одна из них (XPath) также используется в нескольких других спецификациях XML, компоненты, которые описаны здесь, чаще всего применяются для преобразования XML из одного формата в другой.

Поскольку все три спецификации тесно связаны друг с другом и практически всегда используются вместе, не всегда просто их разграничить. Зачастую это приводит к спорам, которые легко понять, но которые не обязательно являются технически корректными. Другими словами, термин XSLT, означающий конкретную вещь, «расширяемые таблицы стилей для преобразований», часто применим и к XSL, и к XPath. Подобным же образом термин XSL часто используется в качестве обобщающего термина для всех трех технологий. В данном разделе я буду различать эти три рекомендации и останусь верным букве спецификаций, рассматривая эти технологии. Однако для простоты восприятия текста в оставшейся части книги я равнозначно использую XSL и XSLT, ссылаясь на преобразования в целом. Такой подход может не соответствовать букве спецификаций, но вне всякого сомнения следует их духу, так же как и отказ от длинных определений простых понятий, когда читатель уже понимает, о чем речь.

XSL

XSL – это расширяемый язык таблиц стилей (Extensible Stylesheet Language), определяемый как язык выражения таблиц стилей. Это широкое определение состоит из двух частей:

- XSL представляет собой язык для преобразования XML-документов
- XSL является словарем XML для описания форматирования XML-документов

Эти определения похожи, но первое затрагивает переход XML-документов из одной формы в другую, а второе сфокусировано на реальном представлении содержимого каждого документа. Пожалуй, более ясное определение звучит так: XSL отвечает за способ преобразования документа из формата A в формат B. Компоненты языка управляют обработкой и идентификацией конструкций, предназначенных для решения задачи преобразования.

XSL и деревья

Самая важная концепция XSL, которую необходимо понять, – все данные на этапах XSL-обработки хранятся в древовидных структурах (рис. 2.1). Кроме того, в древовидной структуре хранятся и сами правила, определяемые при помощи XSL. Таким образом, появляется возможность простой обработки иерархических структур XML-документов. Для выбора корневого элемента обрабатываемого XML-документа используются шаблоны. Далее «дочерние» правила применяются к

«дочерним» элементам, и так вплоть до наиболее глубоко вложенных элементов. На любом уровне обработки элементы могут игнорироваться, обрабатываться, копироваться, к ним могут применяться стили и над ними может выполняться множество других операций.

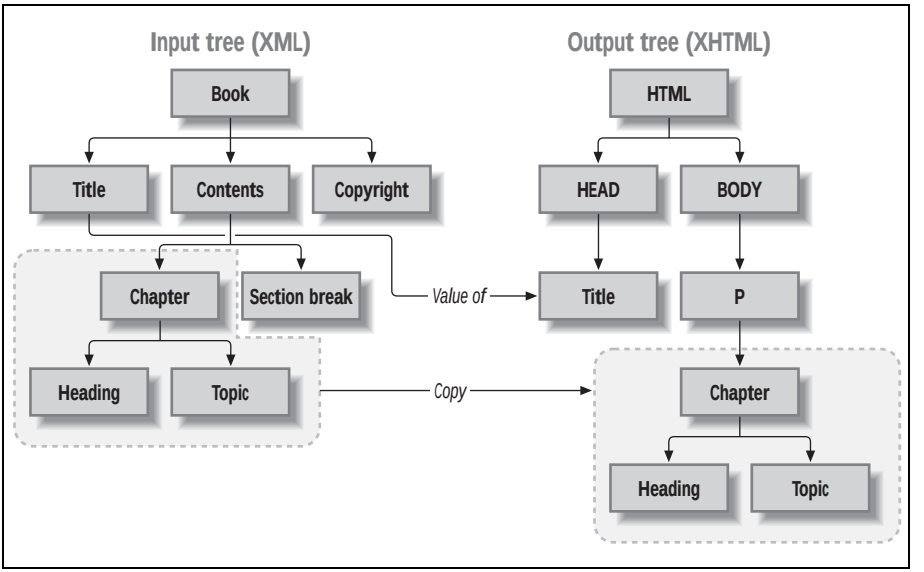

Рис. 2.1. Операции с деревьями в XSL

Замечательное преимущество подобной древовидной структуры заключается в том, что она позволяет сохранять группировку элементов в XML-документах. Если элемент A содержит элементы B и C, и элемент A перемещается или копируется, то над содержащимися в нем элементами выполняются те же самые операции.

Это ускоряет обработку больших фрагментов данных, над которыми требуется выполнять одни и те же операции, упрощает и делает более компактной запись в таблицах стилей XSL. Подробнее о том, как конструируется дерево, рассказано в следующем разделе, посвященном XSLT.

Форматирующие объекты

Почти вся спецификация XSL посвящена определению *форматирующих объектов*. Форматирующий объект основан на масштабной модели, которая, как и следовало ожидать, называется моделью форматирования. Эта модель связана с набором объектов, которые представляют собой исходные данные для системы форматирования, которая применяет объекты к исходному документу целиком или по частям, и в результате получается новый документ, полностью или частично состоящий из данных первоначального XML-документа в формате, специфичном для объектов, используемых форматирующей системой.

Поскольку это весьма неопределенная и неясная концепция, смысл спецификации XSL заключается в определении конкретной модели, которой должны соответствовать эти объекты. Другими словами, широкий набор свойств и словарь составляют совокупность характеристик, которые могут использоваться форматирующими объектами. Они включают типы областей, которые можно визуализировать с помощью объектов, свойства строк, шрифтов, графики и других видимых объектов, строковые и групповые форматирующие объекты, а также множество других синтаксических конструкций.

Особенно интенсивно форматирующие объекты используются при преобразовании текстовых XML-данных в двоичные форматы, такие как файлы PDF, изображения или фирменные форматы документов, например Microsoft Word. Для преобразования XML-данных в другой текстовый формат эти объекты редко используются явным образом. Хотя форматирующие объекты представляют собой основополагающую часть логики таблиц стилей, эти объекты редко применяются непосредственно, поскольку выходные текстовые данные чаще всего оформляются с помощью других предопределенных языков разметки, таких как HTML. Поскольку сегодня большинство корпоративных приложений по крайней мере частично основаны на веб-архитектуре и используют в качестве клиента броузер, большую часть времени мы уделим рассмотрению преобразований в HTML и XHTML. Хотя форматирующие объекты рассмотрены лишь поверхностно, эта тема достаточно широка, чтобы посвятить ее обсуждению отдельную книгу. За дополнительной информацией по этой теме обратитесь к спецификации XSL, доступной по адресу *http://www.w3.org/TR/WD-xsl*.

XSLT

Второй компонент преобразований XML – это преобразования XSL. XSLT – это язык, *описывающий* преобразование документа из одного формата в другой (где XSL определяет способы этого описания). Синтаксис XSLT обычно касается текстовых преобразований, которые не приводят к созданию двоичных данных. XSLT является основным средством генерации кода HTML или WML (Wireless Markup Language, язык разметки беспроводных устройств) из XML-документа. В действительности спецификация XSLT описывает синтаксис таблицы стилей XSL более подробно, чем сама спецификация XSL!

Так же как и в случае с XSL, XSLT-таблица всегда представляет собою корректный и действительный XML-документ. Для XSL и XSLT существует DTD-определение, описывающее допустимые конструкции. По этой причине для использования XSLT нужно лишь изучить новый синтаксис, тогда как для создания собственно DTD-определений приходится разбираться в совершенно новых конструкциях. Подобно XSL, технология XSLT основана на иерархической древовидной структуре данных, в которой вложенные элементы являются листьями или

дочерними элементами своих родителей. XSLT предоставляет механизм шаблонов для поиска в исходном XML-документе (выражения XPath, которые рассмотрены далее) и позволяет форматировать данные. В результате можно легко вывести данные, исключив ненужные имена элементов XML, или заполнить данными сложную HTML-таблицу, а затем выдать ее пользователю, применив соответствующее форматирование. XSLT содержит синтаксические конструкции многих распространенных операторов, таких как условные операторы, возможности копирования фрагментов древовидной структуры документа, расширенного поиска по шаблону, а также возможность доступа к элементам входных XML-данных по абсолютному или относительному пути. Все эти конструкции разработаны для облегчения процесса преобразования XML-документа в новый формат. Подробное описание языка XSLT можно найти в книге «Java and XSLT» (Java и XSLT), Эрик Бурк (Eric Burke), O'Reilly, в которой очень хорошо рассказано, как заставить XSLT работать с Java.

XPath

Заключительный фрагмент головоломки, связанной с преобразованием XML, – XPath – обеспечивает механизм адресации широкого многообразия имен и значений элементов и атрибутов XML-документа. Как говорилось ранее, многие спецификации XML в настоящее время используют XPath, но наше обсуждение касается его применения в контексте XSLT. Принимая во внимание сложную структуру, которую может иметь XML-документ, поиск какого-либо конкретного элемента или набора элементов может оказаться трудной задачей. Эта задача усложняется еще и тем, что изначально нельзя предполагать наличие доступа к DTD либо к другому набору ограничений, описывающих структуру документа: необходимо иметь возможность преобразовывать документы, действительность которых не подтверждена, так же как и действительные документы. Адресация элементов в XPath реализована благодаря определению синтаксиса, согласованного с древовидными структурами XML, а также с операциями и конструкциями XSLT, которые его используют.

Адресацию любого элемента или атрибута в XML-документе проще всего выполнить посредством указания пути элемента относительно текущего элемента. Другими словами, если элемент B является текущим элементом, а элементы C и D вложены в него, проще всего сослаться на них с помощью относительного пути. Здесь уместна аналогия с относительными путями, используемыми в структурах файловых систем. Одновременно XPath определяет также адресацию элементов относительно корня документа. Часто бывает необходимо обратиться к элементу, который находится за пределами области видимости текущего элемента, другими словами, к элементу, который не является вложенным в обрабатываемый элемент. Наконец, XPath

определяет синтаксис для поиска по шаблону – поиска элемента, для которого родительским элементом является элемент E и для которого имеется элемент F того же уровня. Таким образом, появляется возможность непрерывной адресации иерархии элементов документа. Все эти возможности могут применяться и для поиска по атрибутам. Несколько образцов приведено в примере 2.6.

Пример 2.6. Выражения XPath

```
<!-- Выбрать элемент с именем Book относительно текущего элемента -->
<xsl:value-of select="Book" />

<!-- Выбрать элемент Contents ,вложенный в элемент Book -->
<xsl:value-of select="Book/Contents" />

<!-- Выбрать элемент Contents по абсолютному пути -->
<xsl:value-of select="/Book/Contents" />

<!-- Выбрать атрибут name текущего элемента -->
<xsl:value-of select="@name" />

<!-- Выбрать атрибут title элемента Chapter -->
<xsl:value-of select="Chapter/@title" />
```

Поскольку входной документ не является фиксированным, вычисление выражения XPath может привести к изучению несуществующих входных данных, одного исходного элемента или атрибута либо множества исходных элементов и атрибутов. Это делает XPath очень полезным и удобным, а также требует введения некоторых дополнительных терминов. Результат вычисления выражения XPath обычно называют *набором узлов*. Такое название не должно вызывать удивления, поскольку согласуется с идеей иерархической или древовидной структуры, которая обычно рассматривается в терминах ее *листьев* (*leaves*) или *узлов* (*nodes*). Полученный в результате вычисления набор узлов можно преобразовать, скопировать, проигнорировать или выполнить над ним любую иную допустимую операцию. Помимо выражений, предназначенных для выбора узлов, XPath также определяет несколько функций для работы с этими наборами, таких как `not()` и `count()`. Эти функции получают в качестве входных данных набор узлов (обычно в форме выражения XPath) и производят дальнейший отбор результатов. Все эти выражения и функции в совокупности составляют часть спецификации и реализаций XPath. Однако понятие XPath зачастую используется для обозначения любых выражений, которые согласуются со спецификациейно тем не менее упрощает обсуждение технологий XSL и XPath.

С учетом сказанного вы уже готовы к тому, чтобы взглянуть на простую таблицу стилей XSL, приведенную в примере 2.7. И хотя сейчас вы можете понять еще не все, давайте быстро обсудим некоторые ключевые аспекты таблицы стилей.

Пример 2.7. Таблица стилей XSL для примера 2.1

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

<xsl:stylesheet xmlns:javaxml2="http://www.oreilly.com/javaxml2"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ora="http://www.oreilly.com"
                version="1.0"
>

  <xsl:template match="javaxml2:book">
    <html>
      <head>
        <title><xsl:value-of select="javaxml2:title" /></title>
      </head>
      <body>
        <xsl:apply-templates select="*[not(self::javaxml2:title)]" />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="javaxml2:contents">
    <center>
     <h2>Table of Contents</h2>
    </center>
    <hr />
    <ul>
     <xsl:for-each select="javaxml2:chapter">
      <b>
       Chapter <xsl:value-of select="@number" />.
       <xsl:text> </xsl:text>
       <xsl:value-of select="@title" />
      </b>
      <xsl:for-each select="javaxml2:topic">
       <ul>
        <li><xsl:value-of select="@name" /></li>
       </ul>
      </xsl:for-each>
     </xsl:for-each>
    </ul>
  </xsl:template>

  <xsl:template match="ora:copyright">
    <p align="center"><font size="-1">
     <xsl:copy-of select="*" />
    </font></p>
  </xsl:template>

</xsl:stylesheet>
```

Соответствие шаблонам

В основе работы XSL лежит соответствие шаблонам. Для элемента, над которым необходимо выполнить некое действие, обычно опреде-

ляется шаблон с критерием,[1] которому соответствует данный элемент. Шаблон обозначается ключевым словом template, а имя искомого элемента[2] указано в его атрибуте match:

```
<xsl:template match="javaxml2:book">
  <html>
    <head>
      <title><xsl:value-of select="javaxml2:title" /></title>
    </head>
    <body>
      <xsl:apply-templates select="*[not(self::javaxml2:title)]" />
    </body>
  </html>
</xsl:template>
```

В данном случае выбирается элемент book из пространства имен javaxml2. Когда XSL-процессор встречает элемент book, выполняются инструкции из этого шаблона. В данном примере выводятся несколько тегов HTML (теги html, head, title и body). Не забывайте различать элементы XSL и все остальные (скажем, элементы HTML-разметки) посредством корректного применения префиксов пространств имен.[3]

Вместо шаблона можно воспользоваться конструкцией value-of для получения значения элемента. При этом имя искомого элемента передается как значение атрибута select. В данном примере символьные данные из элемента title используются в качестве заголовка выводимой HTML-формы.

С другой стороны, если требуется применить шаблоны, которым соответствуют дочерние элементы текущего элемента, воспользуйтесь конструкцией apply-templates. Обязательно сделайте это, иначе вложенные элементы могут быть проигнорированы! При помощи атрибута select можно определить элементы, к которым будут применяться шаблоны. Значение "*" позволяет применить шаблоны ко всем вложенным элементам. В данном примере исключен элемент title (т. к. он уже использован в заголовке документа). Для решения задачи я указал ключевое слово not и задал элемент title на оси self, что, по сути, означает «все (*), кроме (not) элемента title в этом документе (self::javaxml2:title)». Это лишь краткий обзор, но я просто стараюсь предоставить вам достаточное количество информации, чтобы можно было перейти к коду на Java.

---

1 Имеется в виду выражение XPath. – *Примеч. науч. ред.*

2 Опять же, речь идет о пути к элементу в терминах XPath. – *Примеч. науч. ред.*

3 Стандартным префиксом пространства имен для таблиц стилей XSLT является xsl. – *Примеч. науч. ред.*

Циклы

При работе с XSL часто возникает необходимость в циклах. Взгляните на следующий фрагмент из примера 2.7:

```
<xsl:template match="javaxml2:contents">
  <center>
   <h2>Table of Contents</h2>
  </center>
  <hr />
  <ul>
   <xsl:for-each select="javaxml2:chapter">
    <b>
     Chapter <xsl:value-of select="@number" />.
     <xsl:text> </xsl:text>
     <xsl:value-of select="@title" />
    </b>
    <xsl:for-each select="javaxml2:topic">
     <ul>
      <li><xsl:value-of select="@name" /></li>
     </ul>
    </xsl:for-each>
   </xsl:for-each>
  </ul>
</xsl:template>
```

Тут в цикле осуществляется обход всех элементов `chapter` при помощи конструкции `for-each`. В Java это выглядело бы так:

```
for (Iterator i = chapters.iterator(); i.hasNext(); ) {
    // предпринять действия для каждой главы
}
```

Внутри цикла «текущим» элементом становится следующий найденный элемент `chapter`. Для каждого из них выводится номер главы. Я делаю это, получая значение атрибута `number` с помощью элемента `value-of`. Чтобы показать, что меня интересует атрибут (а не элемент, как установлено по умолчанию), я предваряю имя атрибута символом «@». То же самое делается для получения значения атрибута `title`, а затем во вложенном цикле просматриваются темы всех глав.

Обратите внимание на довольно странный фрагмент кода `<xsl:text> </xsl:text>`. Конструкция `text` предоставляет возможность непосредственно выводить символы в конечное дерево. Эта конструкция генерирует пробел между словом «Chapter» и номером главы (между открывающим и закрывающим тегами `text` всего один пробел).

Копирование

Кроме того, вы столкнетесь с ситуацией, когда всевозможные варианты соответствия шаблону не столь полезны, как простая передача не-

измененного содержимого в получаемое дерево. Именно это и происходит в случае с элементом `copyright`:

```
<xsl:template match="ora:copyright">
  <p align="center"><font size="-1">
   <xsl:copy-of select="*" />
  </font></p>
</xsl:template>
```

Помимо некоторого форматирования HTML этот шаблон при помощи конструкции `copy-of` указывает на то, что все содержимое элемента `copyright` необходимо скопировать в получаемое дерево. Довольно просто.

В главе 10 рассказано о том, как использовать систему публикации (скажем, Cocoon) для вывода результатов этого преобразования в HTML, PDF или ином формате. Не буду утомлять читателей ожиданием: на рис. 2.2 показан преобразованный вывод для примера 2.1 и таблицы стилей из примера 2.6.

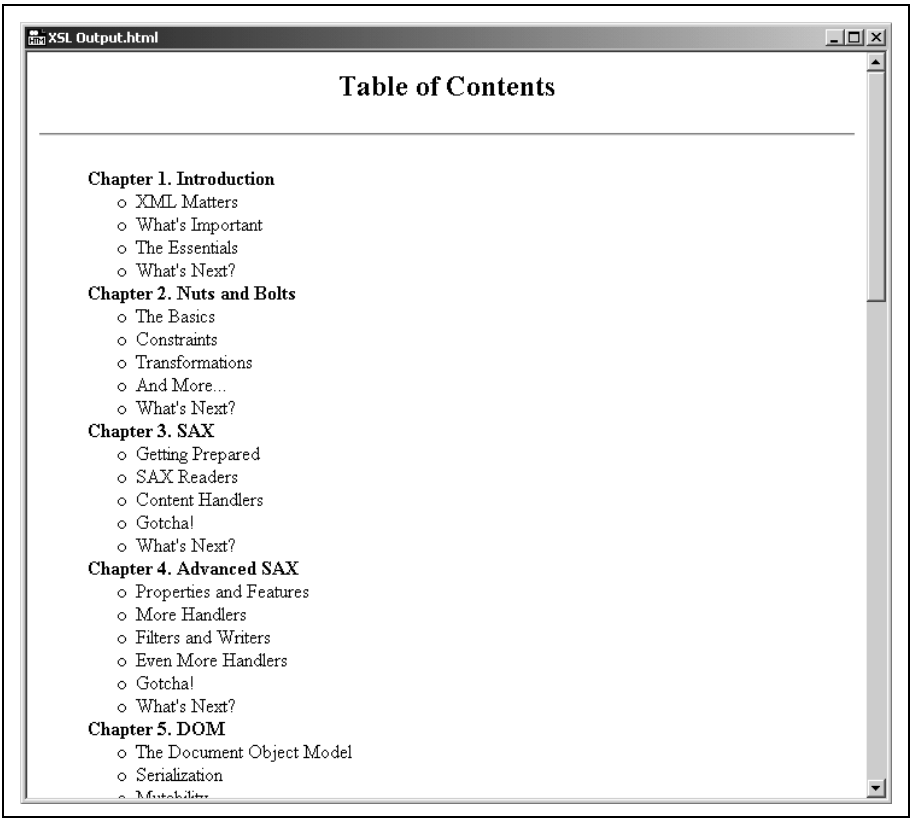

Рис. 2.2. Результат преобразования XSL

Понятно, что я практически «пролетел» через весь этот материал, но, опять же, я просто пытаюсь провести вас от основ к более полезным вещам – Java и XML. Держите под рукой справочник и не особо тревожьтесь.

И еще...

Чтобы не ввести вас в заблуждение, будто все, связанное с XML, уже описано, я скажу, что существует множество других технологий, имеющих отношение к XML. Мы не можем рассмотреть их все. Те, кто работает в области веб-дизайна, должны получить представление о таких вещах, как CSS (каскадные таблицы стилей) и XHTML. Создатели документов захотят больше узнать об XLink и XPointer (они рассмотрены в главе 16). XQL (язык запросов XML) будет интересен программистам баз данных. Другими словами, сейчас XML применяется практически во всех технологиях. Взгляните на страницу W3C, посвященную XML, по адресу *http://www.w3.org/XML* и вы найдете то, что покажется вам интересным.

Что дальше?

Имея некоторые базовые познания в XML, можно углубиться в вопросы, касающиеся Java. В следующей главе рассказывается о SAX – простом API для XML. Это ядро для понимания Java и интерфейсов прикладного XML-программирования. Здесь вы узнаете, как использовать XML в собственных приложениях на Java, как читать документы, устанавливать различные параметры для проверки действительности по DTD или схеме, работать с пространствами имен и многое другое, а также поймете, в каких случаях SAX является подходящим инструментом для решения задачи. Запускайте свой редактор и переворачивайте страницу.

- *Подготовительные операции*
- *SAX-совместимые анализаторы*
- *Обработчики содержимого*
- *Обработчики ошибок*
- *Советы разработчикам*
- *Что дальше?*

3

SAX

Приступая к работе с XML-ориентированными программными средствами, следует, в числе прочего, перво-наперво получить XML-документ и проанализировать его. Как только документ проанализирован, его данные становятся доступными для приложения, использующего анализатор XML, и, таким образом, мы получаем XML-ориентированное приложение! Хотя все это звучит слишком просто, чтобы быть правдой, это почти так и есть. В данной главе рассказано, как происходит анализ XML-документа. Особое внимание мы уделим событиям, происходящим в ходе этого процесса. Эти события очень важны, т. к. они представляют собой те точки, где можно вставлять специфический для приложения код и где могут происходить манипуляции с данными.

Кроме того, в качестве движущей силы в этой главе я обираюсь представить простой интерфейс прикладного программирования для XML (Simple API for XML, SAX). Именно SAX делает возможным привязку специфичного для приложения кода к событиям. Интерфейсы, входящие в состав SAX, для каждого программиста станут важной составляющей набора средств для обработки XML. Хотя классы SAX невелики и немногочисленны, они формируют важную систему, позволяющую работать с XML в Java. Основательное понимание того, как они помогают получить доступ к XML-данным, является решающим для эффективного внедрения XML в программы на Java. В последующих главах мы добавим к набору инструментов другие интерфейсы прикладного программирования для Java и XML, такие как DOM, JDOM, JAXP, а также рассмотрим связывание данных. Но довольно, пришло время перейти от пустых разговоров к изучению SAX.

Подготовительные операции

Прежде чем приступать к программированию, необходимо иметь несколько вещей. Вот они:

- Анализатор XML
- Классы SAX
- XML-документ

Для начала нужно найти анализатор XML. Создание синтаксического анализатора для XML – серьезная задача, и сейчас предпринимаются усилия для создания хороших анализаторов, особенно в области разработки программного обеспечения с открытыми исходными кодами. Я не собираюсь здесь подробно расписывать процесс написания анализатора. Вместо этого я рассмотрю приложения, которые реализуют их функциональность, обращая внимание на применение существующих инструментов для манипулирования XML-данными. Такой подход приведет к созданию более качественных и более быстрых программ, поскольку ни вы, ни я не будем тратить время на попытку изобрести велосипед. Выбрав анализатор, необходимо убедиться в наличии классов SAX. Найти их нетрудно, и они представляют собой ключ к способности нашего кода на Java обрабатывать XML. Наконец, нам нужен сам XML-документ. После этого можно переходить к написанию кода!

Получение анализатора

Первым шагом при написании кода на Java, который использует XML, является поиск и получение анализатора, который вы хотите применять. Вкратце об этом процессе сказано в главе 1, где были перечислены различные анализаторы XML. Ваш анализатор будет работать со всеми примерами из этой книги, если он совместим со спецификацией XML. В связи с тем, что в настоящее время доступно множество анализаторов, а также в связи с быстрыми темпами изменений в мире технологий XML, все подробности о том, какой уровень совместимости обеспечивает каждый из анализаторов, выходят за рамки данной книги. Для получения этой информации следует обратиться к производителю анализатора и/или посетить указанные ранее страницы в Интернете.

В духе сообщества ПО с открытыми исходными кодами во всех примерах из этой книги используется анализатор Apache Xerces. Доступный свободно как в скомпилированном виде, так и в виде исходного кода на сервере *http://xml.apache.org* этот анализатор, существующий в виде C- и Java-версий, уже сейчас является одним из наиболее широко используемых среди доступных анализаторов (таких упрямых программистов на Java, как мы, ведь не интересует C, верно?). К тому же применение анализатора с открытым исходным кодом, такого как

Xerces, позволяет посылать вопросы и сообщения об ошибках его разработчикам, что приводит к повышению качества продукта, а также помогает работать с ним быстро и правильно. Чтобы продписаться на список рассылки, посвященный анализатору Xerces, пошлите пустое сообщение по адресу *xerces-j-dev-subscribe@xml.apache.org*. Если у вас имеются вопросы или проблемы, не рассмотренные в настоящей книге, его участники смогут оказать вам помощь. Разумеется, все примеры из этой книги нормально работают с любым анализатором XML, использующим рассмотренную здесь реализацию SAX.

Выбрав и загрузив анализатор XML, убедитесь, что пути к классам анализатора XML прописаны в переменной окружения `CLASSPATH` вашей среды Java, будь то интегрированная среда разработки (IDE) или командная строка. Это будет основным требованием для всех следующих примеров.

Примечание

Если вы не знаете, как быть с переменной `CLASSPATH`, то окажетесь в затруднительной ситуации. Однако будем считать, что вы легко разбираетесь с переменной `CLASSPATH` в своей системе, поэтому добавьте к другим путям файл с расширением *jar* вашего анализатора, как показано здесь:

```
c: set CLASSPATH=.;c:\javaxml2\lib\xerces.jar;%CLASSPATH%

c: echo %CLASSPATH%
.;c:\javaxml2\lib\xerces.jar;c:\java\jdk1.3\lib\tools.jar
```

Конечно, на других машинах путь будет отличаться от этого, но суть должна быть ясна.

Получение классов и интерфейсов SAX

Итак, анализатор у вас есть, теперь надо найти классы SAX. Эти классы почти всегда поставляются вместе с анализатором, и Xerces не исключение. Если это так и в вашем случае, то вам не нужно скачивать классы SAX отдельно, поскольку ваш анализатор, вероятнее всего, снабжен самой последней версией SAX, которую он поддерживает. На момент написания этих строк версия SAX 2.0 применяется уже давно, поэтому можете рассчитывать, что приведенные здесь примеры (все они ориентированы на SAX 2) будут работать без изменений.

Если вы не уверены, что у вас есть классы SAX, посмотрите на *jar*-файл или структуру классов, используемую анализатором. Классы SAX собраны в пакет `org.xml.sax`. Убедитесь, что вы видите, как минимум, класс `org.xml.sax.XMLReader`. Это означает, что вы (почти наверняка) используете анализатор с поддержкой SAX 2, т. к. класс `XMLReader` является ключевым для SAX 2.

Наконец, вы, возможно, захотите загрузить документацию Javadoc по SAX API или создать закладку для ее адреса в Интернете. Эта доку-

ментация исключительно полезна при работе с классами SAX, а формат Javadoc обеспечивает простой и стандартный способ поиска дополнительной информации о классах и о том, что они делают. Эту документацию можно найти по адресу *http://www.saxproject.org/*. При желании также можно сгенерировать документацию в формате Javadoc либо из исходного кода SAX, прилагаемого к вашему анализатору, либо скачав полный комплект исходных текстов со страницы *http://www.saxproject.org/*. Наконец, в дистрибутивы многих анализаторов уже включена документация, и в ее состав может входить документация по SAX API (именно так обстоят дела с Xerces).

Подготовка XML-документа

Также необходимо убедиться, что у вас есть подходящий для синтаксического анализа XML-документ. Результаты, приводимые в примерах, получены в результате синтаксического анализа XML-документа, который мы обсуждали в главе 2. Сохраните этот файл под именем *contents.xml* на локальном жестком диске. Настоятельно рекомендую следить за тем, как происходит работа с этим документом; для наглядности он содержит различные конструкции XML. Этот файл можно просто набрать вручную либо загрузить с веб-сайта этой книги *http://www.newInstance.com*.

SAX-совместимые анализаторы

Но перейдем к написанию кода и не будем тратить время на предварительные замечания. В качестве примера, позволяющего ознакомиться с SAX, в этой главе мы подробно рассмотрим класс `SAXTreeViewer`. Этот класс использует SAX для анализа XML-документа, имя которого передается в командной строке, и отображает документ в компоненте `JTree` библиотеки Swing. Если вы ничего не знаете о Swing, не волнуйтесь – она используется здесь для наглядности, и мы не станем останавливаться на ней подробно. Все внимание будет уделено SAX и тому, как использовать события, происходящие в процессе анализа, для выполнения необходимых действий. На самом деле происходит следующее: используется компонент `Jtree`, отображающий входной XML-документ в виде простой древовидной модели. Основой для этого дерева является класс `DefaultMutableTreeNode`, к которому вы привыкнете в процессе рассмотрения этого примера, а также класс `DefaultTreeModel`, отвечающий за компоновку.

Первое, что нужно сделать в любом приложении, основанном на SAX, – получить экземпляр класса, реализующего интерфейс SAX `org.xml.sax.XMLReader`. Этот интерфейс определяет методику анализа документа и позволяет устанавливать свойства и возможности (они рассмотрены далее в этой главе). Для тех из вас, кто знаком с SAX 1.0, скажу, что этот интерфейс заменяет интерфейс `org.xml.sax.Parser`.

Внимание

Сейчас самое время обратить внимание на то, что в этой книге не рассматривается SAX 1.0. И хотя в конце этой главы присутствует очень маленький раздел о том, как преобразовать код для SAX 1.0 в код для SAX 2.0, тем, кто работает с SAX 1.0, не позавидуешь. И если первое издание этой книги вышло вслед за SAX 2.0, то сейчас уже прошло больше года с тех пор, как версия 2.0 интерфейса была выпущена в своем окончательном виде. Я настоятельно советую перейти к версии 2, если вы еще этого не сделали.

Создание экземпляра класса Reader

SAX определяет интерфейс, который должны реализовывать все SAX-совместимые анализаторы XML. Благодаря этому SAX точно знает, какие методы доступны для обратного вызова и использования в приложении. Например, основной класс SAX-анализатора Xerces, `org.apache.xerces.parsers.SAXParser`, реализует интерфейс `org.xml.sax.XMLReader`. Посмотрев на исходный код своего анализатора, вы должны увидеть тот же самый интерфейс, реализованный в основном классе SAX-анализатора. Каждый анализатор XML должен иметь как минимум один класс (а иногда и более), реализующий этот интерфейс, и для того, чтобы иметь возможность производить разбор XML, необходимо создать экземпляр именно этого класса:

```
// Создание экземпляра класса Reader
XMLReader reader =
  new org.xml.sax.SAXParser();

// Делаем что-либо с помощью анализатора
reader.parse(uri);
```

Имея это в виду, перейдем к более реалистичному примеру. Пример 3.1 – это основа, «скелет» класса `SAXTreeViewer`, только что упомянутого и позволяющего просматривать XML-документ в виде дерева. Также мы сможем взглянуть на все события SAX и связанные с ними методы обратного вызова, применяемые для выполнения действий при анализе XML-документа.

Пример 3.1. «Скелет» класса SAXTreeViewer

```
package javaxml2;

import java.io.IOException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.ErrorHandler;
import org.xml.sax.InputSource;
```

```
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

// Это книга про XML - нет необходимости в явном
// импортировании Swing
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;

public class SAXTreeViewer extends JFrame {

    /** Анализатор по умолчанию */
    private String vendorParserClass =
        "org.apache.xerces.parsers.SAXParser";

    /** Основное отображаемое дерево */
    private JTree jTree;

    /** Используемая модель дерева */
    DefaultTreeModel defaultTreeModel;

    public SAXTreeViewer() {
        // Обработка установки Swing
        super("SAX Tree Viewer");
        setSize(600, 450);
    }

    public void init(String xmlURI) throws IOException, SAXException {
        DefaultMutableTreeNode base =
            new DefaultMutableTreeNode("XML Document: " +
                xmlURI);

        // Строим модель дерева
        defaultTreeModel = new DefaultTreeModel(base);
        jTree = new JTree(defaultTreeModel);

        // Создаем иерархию дерева
        buildTree(defaultTreeModel, base, xmlURI);

        // Отображаем результаты
        getContentPane().add(new JScrollPane(jTree),
            BorderLayout.CENTER);
    }

    public void buildTree(DefaultTreeModel treeModel,
                          DefaultMutableTreeNode base, String xmlURI)
        throws IOException, SAXException {

        // Создаем экземпляры, необходимые для анализа
        XMLReader reader =

      XMLReaderFactory.createXMLReader(vendorParserClass);

        // Регистрируем обработчик содержимого
```

```
        // Регистрируем обработчик ошибок

        // Анализируем
    }

    public static void main(String[] args) {
        try {
            if (args.length != 1) {
                        System.out.println(
                           "Usage: java javaxml2.SAXTreeViewer " +
                           "[XML Document URI]");
                        System.exit(0);
            }
            SAXTreeViewer viewer = new SAXTreeViewer();
            viewer.init(args[0]);
            viewer.setVisible(true);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Происходящее здесь должно быть вполне прозрачно.[1] Помимо установки свойств Swing, этот код принимает URI XML-документа (файл *contents.xml* из предыдущей главы). В методе `init()` создается дерево `JTree` для отображения данных, на которые указывает URI. Затем объекты дерева и URI-идентификатора передаются методу `buildTree()`, на который стоит обратить внимание. Именно здесь будет происходить анализ и будет создаваться визуальное представление XML-документа. Кроме того, этот «каркас» создает основной узел графического дерева с путем к указанному XML-документу в качестве текста этого узла.

Если описанные ранее подготовительные операции выполнены и анализатор XML и пути к классам SAX упомянуты в переменной `CLASSPATH`, то можно успешно скомпилировать и запустить эту программу. Если применяется анализатор, отличный от Apache Xerces, то можно изменить значение переменной `vendorParserClass` на класс реализации `XMLReader` анализатора, а весь остальной код оставить без изменений. Эта простая программа пока что не делает ничего особенного: если запустить ее и передать ей в качестве аргумента корректное имя файла, она успешно завершится и выдаст пустое дерево с именем файла документа в качестве основного узла. Это происходит потому, что мы всего лишь создали экземпляр класса `reader`, но не требовали, чтобы документ был проанализирован.

---

[1] Не волнуйтесь, если вы не знакомы с концепциями Swing, используемыми здесь. Если честно, мне самому пришлось разбираться в большинстве из них! Очень хороший справочник по Swing – книга «Java Swing» Роберта Экштейна, Марка Лоя и Дэйва Вуда (Robert Eckstein, Marc Loy and Dave Wood) издательства O'Reilly.

U-R-Что?

В этой и предыдущей главах я промчался мимо определения URI. В двух словах, URI – это *унифицированный идентификатор ресурса* (*uniform resource identifier*). Как следует из названия, он обеспечивает стандартный способ идентификации (а значит и поиска, в большинстве случаев) конкретного ресурса. В соответствии с целями, преследуемыми этой книгой, данный ресурс практически всегда является некоторым XML-документом. URI родственны URL – *унифицированным указателями ресурсов* (*uniform resource locators*). На самом деле URL – это всегда URI (хотя обратное не верно). Поэтому в примерах из этой и других глав можно задавать имя файла или URL примерно так: *http://www.newInstance.com/javaxml2/copyright.xml*; допустимы оба варианта.

Внимание

Если вы столкнулись с проблемами при компиляции этого исходного файла, вероятнее всего, проблема в настройке системной или IDE-переменной `CLASSPATH`. Первым делом убедитесь, что у вас имеется анализатор Apache Xerces (или анализатор другого производителя). Чтобы получить анализатор Xerces, требуется скачать сжатый (в формате zip или gzip) файл. Затем этот архив можно распаковать – в нем содержится файл *xerces.jar*; именно этот *jar*-файл содержит скомпилированные файлы классов для этой программы. Добавьте этот архив в пути к классам (в переменную `CLASSPATH`). После этого можно скомпилировать приведенный исходный код примера.

Анализ документа

Как только класс `reader` загружен и готов к использованию, можно предписать ему проанализировать документ. Это удобно делать при помощи метода `parse()` класса `org.xml.sax.XMLReader`. В качестве параметра метода может выступать экземпляр класса `org.xml.sax.InputSource` либо строка, содержащая URI. Оптимально использовать класс SAX `InputSource`, поскольку он предоставляет больше информации, чем простой указатель ресурса. Подробнее об этом мы поговорим позже, а пока достаточно сказать, что `InputSource` можно создать из классов ввода/вывода `InputStream` и `Reader`, а также на основе строки, содержащей URI.

Теперь можно добавить в пример создание экземпляра класса `InputSource` на основе полученного программой URI, а также вызов метода `parse()`. Поскольку документ должен быть загружен (либо локально, либо удаленным образом), может быть сгенерировано исключение `java.io.IOException`, которое необходимо перехватить. Кроме того, если возникнут проблемы при анализе документа, будет сгенерировано

исключение `org.xml.sax.SAXException`. Обратите внимание, что метод `buildTree` может генерировать оба этих исключения:

```
public void buildTree(DefaultTreeModel treeModel,
                      DefaultMutableTreeNode base, String xmlURI)
    throws IOException, SAXException {

    // Создаем экземпляры, необходимые для анализа
    XMLReader reader =
        XMLReaderFactory.createXMLReader(vendorParserClass);

    // Регистрируем обработчик содержимого

    // Регистрируем обработчик ошибок

    // Анализируем
    InputSource inputSource =
        new InputSource(xmlURI);
    reader.parse(inputSource);
}
```

Скомпилируйте эти изменения – и можете запускать данный пример. В качестве первого аргумента командной строки программы нужно указать полный путь к вашему файлу:

```
c:\javaxml2\build>java javaxml2.SAXTreeViewer ..\Ch03\xml\contents.xml
```

Внимание

Передача URI может оказаться довольно необычной задачей. В версиях Xerces до 1.1 можно было задавать обычное имя файла (например, в Windows – *..\xml\contents.xml*). В Xerces версий 1.1 и 1.2 URI должен быть представлен в виде *file:///c:/javaxml2/xml/contents.xml*. Однако в самых последних версиях Xerces (начиная от 1.3, а также и в 2.0) все вернулось к старому формату, т. е. принимаются обычные имена файлов. Имейте это в виду, если вы работаете с Xerces версий 1.1 – 1.2.

Довольно непримечательный результат, показанный на рис. 3.1, может вызывать сомнения в том, произошло ли что-либо вообще. Но если

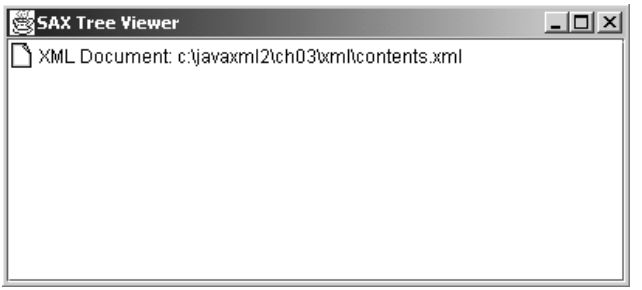

Рис. 3.1. Непримечательное дерево JTree

прислушаться, можно услышать, как коротко прошуршал диск (или просто уверовать в байт-код). В действительности XML-документ проанализирован. Однако в отсутствие реализованных обратных вызовов SAX не предпринимает никаких действий в процессе анализа документа, т. е. анализ документа производится без выдачи каких-либо сообщений и без вмешательства нашего приложения. Разумеется, мы хотим вмешаться в этот процесс, поэтому пришло время рассмотреть создание нескольких *методов обратного вызова* (*callback methods*). Методы обратного вызова не вызываются в коде приложения непосредственно, но они вызываются анализатором SAX[1] при возникновении конкретных событий без всякого вмешательства. Другими словами, ваш код не работает с анализатором, но анализатор работает с вашим код (отсюда и название – обратные вызовы). Мы получаем возможность программным образом изменять поведение анализатора в процессе работы. Такое вмешательство является самой важной частью использования SAX. Методы обратного вызова анализатора позволяют вставлять определенные действия в процесс выполнения программы и превращают весьма скучный, немногословный анализ XML-документа в приложение, которое реагирует на данные, элементы, атрибуты и структуру анализируемого документа, а также взаимодействует при этом с другими программами и клиентами.

Использование InputSource

Я уже говорил, что еще коснусь вопроса использования `InputSource`, пусть даже очень поверхностно. Преимущество применения `InputSource` вместо непосредственной передачи URI очевидно: это позволяет предоставить анализатору больше информации. `InputSource` инкапсулирует информацию об одном объекте – документе, который необходимо проанализировать. В ситуациях, когда системный идентификатор, публичный идентификатор или поток могут быть все связаны с одним URI, использование `InputSource` для инкапсуляции может стать очень удобным. Этот класс предоставляет методы для чтения и модификации системного и публичного идентификаторов, кодировки символов, байтового потока (`java.io.InputStream`) и символьного потока (`java.io.Reader`). Также SAX гарантирует, что анализатор никогда не изменит объект `InputSource`, передаваемый в качестве аргумента методу `parse()`. Значит, первоначальные входные данные, переданные анализатору, по-прежнему будут доступны в неизменном виде после того, как они будут использованы анализатором или приложением, поддерживающим XML. В нашем примере это важно, т. к. в XML-документе указан относительный путь к DTD в элементе:

```
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">
```

[1] Анализаторы других типов не обязательно предоставляют возможность привязывать события к пользовательским методам. – *Примеч. науч. ред.*

Используя `InputSource` и заключая в него переданный URI, мы определяем системный идентификатор документа. По сути дела, устанавливаем путь к документу для анализатора, что и позволяет разрешать все относительные пути внутри этого документа, как, например, путь для файла *JavaXML.dtd*. Если вместо установки этого идентификатора проанализировать поток ввода/вывода, DTD не будет найдено (поскольку отсутствует система координат). Можно сымитировать такую ситуацию, изменив код из метода `buildTree()` следующим образом:

```
// Анализируем
InputSource inputSource =
    new InputSource(new java.io.FileInputStream(
        new java.io.File(xmlURI)));
reader.parse(inputSource);
```

В результате при запуске программы просмотра вы получите следующее исключение:

```
C:\javaxml2\build>java javaxml2.SAXTreeViewer ..\ch03\xml\contents.xml
org.xml.sax.SAXParseException: File
  "file:///C:/javaxml2/build/DTD/JavaXML.dtd" not found.
```

И хотя заключение URI в файл и поток ввода/вывода кажется глупым, разработчики довольно часто пытаются использовать потоки ввода/вывода в качестве входных данных для анализатора. Просто убедитесь, что вы не ссылаетесь на другие файлы из XML и что для потока XML установлен системный идентификатор (при помощи метода `setSystemId()` объекта `InputSource`). Так что приведенный выше фрагмент кода можно «исправить», изменив его следующим образом:

```
// Анализируем
InputSource inputSource =
    new InputSource(new java.io.FileInputStream(
        new java.io.File(xmlURI)));
inputSource.setSystemId(xmlURI);
reader.parse(inputSource);
```

Всегда устанавливайте системный идентификатор. Простите за чрезмерные подробности. Теперь и вы можете досаждать коллегам рассказами о классе SAX `InputSource`.

Обработчики содержимого

Чтобы дать приложению возможность делать что-нибудь полезное с XML-данными во время их анализа, нужно зарегистрировать в SAX-совместимом анализаторе *обработчики* (*handlers*). Обработчик – это всего лишь набор методов обратного вызова, с помощью которых SAX позволяет программистам связывать код приложения с важными событиями, возникающими во время синтаксического разбора документа.

Эти события происходят в процессе анализа документа, но не после его завершения. Это одна из причин, делающих SAX столь мощным интерфейсом: документ обрабатывается последовательно, нет необходимости загружать его в память целиком. В дальнейшем мы рассмотрим объектную модель документа (Document Object Model, DOM), которая имеет данное ограничение.

В SAX 2.0 определены четыре основных интерфейса-обработчика: `org.xml.sax.ContentHandler`, `org.xml.sax.ErrorHandler`, `org.xml.sax.DTDHandler` и `org.xml.sax.EntityResolver`. В этой главе речь пойдет об интерфейсах `ContentHandler` и `ErrorHandler`. Рассказ о `DTDHandler` и `EntityResolver` отложим до следующей главы; пока же достаточно знать, что `EntityResolver` работает так же, как и остальные обработчики, и создан специально для интерпретации внешних сущностей, на которые ссылается XML-документ. Классы пользовательских приложений, выполняющие специфические действия в процессе анализа документа, могут реализовывать каждый из этих интерфейсов. Классы, реализующие эти интерфейсы, можно зарегистрировать в анализаторе с помощью методов `setContentHandler()`, `setErrorHandler()`, `setDTDHandler()` и `setEntityResolver()`. После этого в процессе синтаксического разбора анализатор будет вызывать методы соотвествующих обработчиков.

Для примера `SAXTreeViewer` было бы хорошо начать с реализации интерфейса `ContentHandler`. Этот интерфейс определяет несколько важных событий процесса синтаксического анализа, на которые может реагировать приложение. Поскольку все необходимые операторы `import` уже существуют (я схитрил и вписал их заранее), остается лишь создать код для реализации интерфейса `ContentHandler`. Для упрощения я создам закрытый класс в исходном файле *SAXTreeViewer.java*. Добавьте класс `JTreeContentHandler`, как показано далее:

```
class JTreeContentHandler implements ContentHandler {

    /** Древовидная модель, в которую добавляем узлы */
    private DefaultTreeModel treeModel;

    /** Текущий узел, к которому добавляем подузлы */
    private DefaultMutableTreeNode current;

    public JTreeContentHandler(DefaultTreeModel treeModel,
                               DefaultMutableTreeNode base) {
        this.treeModel = treeModel;
        this.current = base;
    }

    // Реализация методов ContentHandler
}
```

Не тратьте время на попытки скомпилировать этот исходный файл в таком виде – вы получите массу сообщений о том, что методы, определенные в `ContentHandler`, не реализованы. В оставшейся части этого

раздела мы рассмотрим и добавим в текст приложения каждый из методов. В этом основном классе достаточно передать реализацию TreeModel, используемую для добавления новых узлов к дереву JTree, и основной узел (созданный ранее в методе buildTree()). Основной узел устанавливается в значение переменной-члена с именем current. Эта переменная всегда указывает на узел, с которым происходит работа, и код должен перемещать указатель вниз по иерархии дерева (когда найдены вложенные элементы), а также вверх по дереву (когда вложенные элементы заканчиваются и родительский элемент вновь становится текущим). А теперь пора рассмотреть обратные вызовы ContentHandler и реализовать каждый из них. Для начала взглянем на интерфейс ContentHandler, перечисляющий обратные вызовы, которые должны быть реализованы:

```
public interface ContentHandler {
    public void setDocumentLocator(Locator locator);
    public void startDocument() throws SAXException;
    public void endDocument() throws SAXException;
    public void startPrefixMapping(String prefix, String uri)
throws SAXException;
    public void endPrefixMapping(String prefix)
throws SAXException;
    public void startElement(String namespaceURI, String localName,
                 String qName, Attributes atts)
throws SAXException;
    public void endElement(String namespaceURI, String localName,
          String qName)
throws SAXException;
    public void characters(char ch[], int start, int length)
throws SAXException;
    public void ignorableWhitespace(char ch[], int start, int length)
throws SAXException;
    public void processingInstruction(String target, String data)
throws SAXException;
    public void skippedEntity(String name)
throws SAXException;
}
```

Указатель позиции в документе

Первый метод, который нужно определить, позволяет получать значение указателя позиции в документе для каждого события SAX. Когда происходит вызов пользовательской функции (обратный вызов), классу, реализующему обработчик, часто требуется доступ к текущей позиции SAX-анализатора в XML-файле. Приложение получает возможность обрабатывать события на основе их типов и позиции внутри XML-документа. Скажем, приложение оказывается в состоянии определить строку, с которой связана ошибка. Класс Locator имеет несколько полезных методов, таких как getLineNumber() и getColumnNumber(),

которые позволяют определить текущую позицию в XML-файле. Поскольку эта позиция действительна только для текущего цикла анализа, локатор следует использовать только в области видимости реализации интерфейса ContentHandler. Поскольку это может пригодиться и в дальнейшем, в приводимом ниже фрагменте кода сохраним полученный экземпляр класса Locator в переменной-члене:

```
class JTreeContentHandler implements ContentHandler {

    /** Переменная для копии указателя позиции в документе */
    private Locator locator;

    // Конструктор

    public void setDocumentLocator(Locator locator) {
        // Сохраняем для дальнейшего использования
        this.locator = locator;
    }
}
```

Начало и конец документа

В любом процессе всегда должно быть начало и конец. Каждое из этих важных событий должно случиться лишь один раз: первое из них – до всех остальных событий, а второе – после. Этот очевидный факт исключительно важен для приложений, поскольку дает им возможность точно знать, когда начинается и когда заканчивается анализ документа. В SAX существуют методы обратного вызова для каждого из этих событий: startDocument() и endDocument().

Первый метод, startDocument(), вызывается до всех остальных событий, включая вызов методов других обработчиков SAX, таких как DTDHandler. Другими словами, метод startDocument() вызывается первым не только в ContentHandler, но и вообще во всем процессе анализа документа, за исключением метода setDocumentLocator(), который мы только что обсудили. Это гарантирует вполне точное определение начала анализа документа и позволяет приложению выполнять любые операции, которые оно должно выполнить прежде, чем произойдет анализ.

Второй метод, endDocument(), всегда вызывается последним, опять же, последним среди методов всех остальных обработчиков. Это касается и тех ситуаций, когда возникают ошибки, приводящие к прекращению анализа документа. Об ошибках поговорим позже, а пока отметим, что существуют как критические, так и некритические ошибки. Если имеет место критическая ошибка, будет вызван метод обработчика ErrorHandler, и далее заключительный вызов метода endDocument() завершит попытку синтаксического анализа документа.

В коде примера никаких действий, затрагивающих эти методы, мы не увидим. Но поскольку речь идет о реализации интерфейса, методы должны присутствовать:

```
public void startDocument() throws SAXException {
    // Тут видимых действий не происходит
}

public void endDocument() throws SAXException {
    // Тут видимых действий не происходит
}
```

Оба эти метода могут генерировать исключения SAXException. Это единственный тип исключений, которые могут генерироваться при возникновении событий SAX, он обеспечивает еще один стандартный интерфейс к функциональности, связанной с анализом документа. В то же время, эти исключения обычно являются оболочками для других исключений, которые несут информацию о том, какие именно проблемы имели место. Например, если XML-файл анализируется через сеть по URL-адресу и связь внезапно прерывается, в результате возникает исключение java.net.SocketException. Однако приложение, использующее классы SAX, не должно перехватывать данное исключение, потому что оно не обязано знать, где расположен ресурс XML (это может быть локальный файл, а не сетевой ресурс). Вместо этого приложение может перехватить только одно исключение SAXException. В SAX-совместимом анализаторе первоначальное исключение перехватывается и затем генерируется заново как исключение SAXException, при этом первоначальное исключение помещается внутрь нового. Это позволяет приложениям обрабатывать всего одно исключение, предоставляя при этом возможность собирать подробную информацию о том, какие ошибки возникли в процессе анализа, и делать ее доступной вызывающей программе посредством стандартного исключения. В классе SAXException имеется метод getException(), возвращающий первоначальный объект Exception (если таковой существует).

Инструкции обработки

Мы уже говорили об инструкциях обработки (PI, processing instructions) в XML как о некотором частном случае. Они не считались элементами XML и обрабатывались иным образом – посредством передачи их вызывающему приложению. Как следствие SAX определяет специальный метод обратного вызова для обработки PI-инструкций. Этот метод получает цель инструкции обработки и любые данные, передаваемые ей. В примере из этой главы инструкция обработки может быть преобразована в новый узел и отображена при визуализации дерева:

```
public void processingInstruction(String target, String data)
    throws SAXException {

    DefaultMutableTreeNode pi =
        new DefaultMutableTreeNode("PI (target = '" + target +
                                   "', data = '" + data + "')");
    current.add(pi);
}
```

Реальное приложение, использующее XML-данные, именно здесь могло бы получать инструкции и устанавливать значания переменных либо вызывать методы для выполнения специфической для приложения обработки. Например, система публикации Apache Cocoon может получать флаги, предписывающие выполнение преобразований над данными после того, как они будут проанализированы, или вывод XML в специфическом формате. Данный метод, как и остальные методы обратного вызова в SAX, при возникновении ошибок генерирует исключение `SAXException`.

Примечание

Стоит заметить, что этот метод не получит уведомления об объявлении XML:

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

На самом деле в SAX не существует способа, позволяющего получить эту информацию (и вы обнаружите, что в настоящее время такого способа нет ни в DOM, ни в JDOM!). Причиной этому служит следующий принцип: информация объявления предназначена для анализатора XML, но не для «потребителя» данных документа. Именно поэтому информация не предоставляется разработчику.

Обратные вызовы для работы с пространством имен

Вспоминая разговор о пространстве имен из главы 2, вы должны начать понимать значимость пространств имен и их влияние на анализ и обработку XML. Наряду со схемами XML пространства имен XML являются наиболее значительным понятием, введенным в XML с момента выхода спецификации XML 1.0. В SAX 2.0 поддержка пространств имен осуществляется на уровне элементов. Это позволяет различать пространство имен элемента, представленное префиксом элемента и связанным с этим префиксом URI, и локальное имя элемента. В данном случае под *локальным именем* понимается имя элемента без префикса. Например, локальным именем элемента `ora:copyright` является `copyright`. Префикс пространства имен – `ora`, а URI пространства имен объявлен в виде *http://www.oreilly.com*.

В SAX существует два метода, предназначенных специально для работы с пространствами имен. Эти методы вызываются, когда анализатор достигает начала и конца *отображения префикса* (*prefix mapping*). Хотя термин новый, концепция нам уже знакома. Область отображения префикса – это просто элемент с атрибутом `xmlns`, объявляющим пространство имен. Часто это корневой элемент (который может иметь несколько отображений), но это может быть любой элемент в XML-документе, в котором непосредственно объявлено пространство имен. Например:

```
<catalog>
  <books>
```

```
    <book title="XML in a Nutshell"
          xmlns:xlink="http://www.w3.org/1999/xlink">
      <cover xlink:type="simple" xlink:show="onLoad"
             xlink:href="xmlnutCover.jpg" ALT="XML in a Nutshell"
             width="125" height="350" />
    </book>
  </books>
</catalog>
```

В данном случае явное пространство имен объявлено для нескольких вложенных элементов в документе. Этот префикс и связанный с ним URI (в данном случае `xlink` и *http://www.w3.org/1999/xlink*, соответственно) становятся доступными для элементов и атрибутов, содержащихся внутри элемента, для которого объявлено пространство имен.

Методу обратного вызова `startPrefixMapping()` передается префикс пространства имен, а также URI, связанный с эти префиксом. Считается, что область отображения «закрывается» или «заканчивается», когда элемент, в котором это отображение объявлено, закрывается. Закрытие области отображения приводит к вызову метода `endPrefixMapping()`. Единственная особенность этих методов заключается в том, что они ведут себя не в полном согласии с последовательным принципом, в соответствии с которым разработан SAX. Метод, связанный с областью отображения префикса, вызывается непосредственно *перед* методом, связанным с элементом, в котором пространство имен объявлено, а закрытие области отображения происходит *после* закрытия этого элемента. Однако в этом достаточно смысла: для того чтобы элемент, в котором объявлено пространство имен, мог использовать отображение объявленного пространства имен, отображение должно быть доступно до обратного вызова элемента. Для закрытия области отображения все работает с точностью до наоборот: элемент должен быть закрыт (т. к. он может использовать пространство имен), а затем можно удалить отображение пространства имен из списка доступных отображений.

В классе `JTreeContentHandler` не существует видимых действий, которые должны быть связаны с этими двумя событиями. Тем не меенее, распространенной практикой является сохранение отображения префикса и URI в структуре данных. Вскоре вы увидите, что обратные вызовы элемента сообщают об URI пространства имен, а не о префиксе. Если не сохранить эти префиксы (полученные при помощи метода `startPrefixMapping()`), они не будут доступны в коде обратного вызова элемента. Самый простой способ произвести сохранение – использовать объект `Map`, добавить к нему полученный префикс и URI в методе `startPrefixMapping()`, а затем удалить их в методе `endPrefixMapping()`. Сделать это можно при помощи следующего кода:

```
class JTreeContentHandler implements ContentHandler {

    /** Переменная для копии указателя позиции в документе */
    private Locator locator;
```

```
/** Сохраняем соответствие URI и префикса */
private Map namespaceMappings;

/** Дерево, к которому добавляем узлы */
private DefaultTreeModel treeModel;

/** Текущий узел, к которому добавляем подузлы */
private DefaultMutableTreeNode current;

public JTreeContentHandler(DefaultTreeModel treeModel,
                           DefaultMutableTreeNode base) {
    this.treeModel = treeModel;
    this.current = base;
    this.namespaceMappings = new HashMap();
}

// Существующие методы

public void startPrefixMapping(String prefix, String uri) {
    // Здесь видимых действий не происходит.
    namespaceMappings.put(uri, prefix);
}

public void endPrefixMapping(String prefix) {
    // Здесь видимых действий не происходит.
    for (Iterator i = namespaceMappings.keySet().iterator();
         i.hasNext(); ) {

        String uri = (String)i.next();
        String thisPrefix = (String)namespaceMappings.get(uri);
        if (prefix.equals(thisPrefix)) {
            namespaceMappings.remove(uri);
            break;
        }
    }
}
    }
```

Заметьте, что в качестве ключа для отображения использован URI, а не префикс. Как только что говорилось, обратный вызов `startElement()` предоставляет информацию об URI пространства имен для элемента, а не о префиксе. Поэтому можно ускорить поиск, если сделать ключами URI. Но как можно видеть, по этой причине в `endPrefixMapping()` для удаления отображения, когда оно становится недоступным, требуется выполнить чуть больше работы. В любом случае сохранение отображения пространства имен подобным образом довольно типично для SAX-приложений, поэтому отложите этот прием в набор инструментов для программирования с XML.

Внимание

Рассмотренное здесь решение далеко не идеально, если говорить о более сложных ситуациях, связанных с пространствами имен. Вполне допустимо повторно присваивать префиксы новым URI для области действия элемента либо присваивать одному и тому же URI несколько различных префиксов. Если связать один и тот же URI с различными префиксами, то в нашем примере это приведет к тому, что отображение пространства имен с более широкой областью действия перезапишется отображением с более узкой областью действия. В более надежном приложениии следовало бы хранить префиксы и URI отдельно, а также создать метод, ставящий их в соответствие друг другу без перезаписи. Тем не менее, благодаря этому примеру вы уже получили представление, как следует в общем случае работать с пространствами имен.

Методы, связанные с элементами

Теперь вы, вероятно, готовы перейти к данным в XML-документе. В действительности более половины методов обратного вызова SAX никак не касаются элементов XML, их атрибутов и данных. Дело в том, что анализ XML имеет целью нечто большее, чем простое предоставление приложению доступа к XML-данным. Он должен передавать приложению команды из инструкций обработки XML таким образом, чтобы оно имело информацию о том, какие действия следует предпринять, он должен сообщать приложению, когда анализ документа начинается и когда он заканчивается, и он даже должен указывать приложению на последовательности пробелов, которые можно игнорировать! Если какие-либо из этих методов кажутся вам бессмысленными, читайте дальше.

Конечно же, существуют методы SAX, предназначенные для того, чтобы предоставить пользователю доступ к XML-данным в его документах. Три основных события, связанных с получением данных, отмечают начало и конец каждого элемента, а также обратный вызов метода `characters()`. Они сообщают о начале анализа элемента, о данных внутри элемента и о достижении закрывающего тега элемента. Первый из методов, вызываемых при этом, `startElement()`, предоставляет приложению информацию об элементе XML и любых его атрибутах. Параметры этого метода – имя элемента (в разных формах), а также экземпляр интерфейса `org.xml.sax.Attributes`. Этот вспомогательный интерфейс предоставляет методы для доступа ко всем атрибутам элемента. Он допускает простой перебор атрибутов элемента в цикле (аналогично перебору для класса `Vector`). Помимо возможности ссылаться на атрибут по его индексу (используемому при организации цикла по всем атрибутам) существует возможность ссылаться на атрибут по его имени. Разумеется, теперь необходимо быть немного осторожнее, слово «имя» употребляется в отношении элемента или атрибута XML, поскольку оно может означать разные вещи. В данном случае может

использоваться либо полное имя атрибута (с префиксом пространства имен, если он есть), называемое уточненным именем (*QName*), либо комбинация его локального имени и URI пространства имен, если пространство имен используется. Также существуют вспомогательные методы, такие как `getURI(int index)` и `getLocalName(int index)`, которые позволяют получить дополнительную информацию о пространствах имен, связанную с атрибутом. Применяемый как единое целое, интерфейс `Attributes` предоставляет исчерпывающий набор данных об атрибутах элемента.

Помимо атрибутов элемента мы получаем несколько форм имени элемента. И вновь благодаря пространствам имен в XML. Первым передается URI пространства имен элемента. Это позволяет правильно определить контекст элемента в полном наборе пространств имен документа. Затем передается локальное имя элемента, которое является именем элемента без префикса. Кроме того, в целях сохранения обратной совместимости передается уточненное имя элемента, т. е неизмененное имя элемента, включающее префикс пространства имен, если таковой имеется. Другими словами, как раз то имя, которое было использовано в XML-документе: `ora:copyright` для элемента с информацией о правообладании. Получив всю эту информацию, можно описать элемент с учетом или без учета пространства имен.

В данном примере эта возможность проиллюстрирована рядом действий. Во-первых, создается и добавляется к дереву новый узел с локальным именем элемента. Затем этот узел становится текущим, поэтому все вложенные элементы и атрибуты добавляются к дереву в качестве листьев. Затем при помощи полученного URI пространства имен и объекта `namespaceMappings` (реализующего получение префикса), который только что был добавлен в код из последнего раздела, определяется пространство имен. Оно тоже добавляется в виде узла дерева. Наконец, в цикле обрабатываются атрибуты, полученные через интерфейс `Attributes`, при этом каждый атрибут (с локальным именем и информацией о пространстве имен) добавляется в дерево в виде дочернего узла. Код, выполняющий эти действия:

```
public void startElement(String namespaceURI, String localName,
                         String qName, Attributes atts)
        throws SAXException {

        DefaultMutableTreeNode element =
            new DefaultMutableTreeNode("Element: " + localName);
        current.add(element);
        current = element;

        // Определяем пространство имен
        if (namespaceURI.length() > 0) {
            String prefix =
                (String)namespaceMappings.get(namespaceURI);
```

```
            if (prefix.equals("")) {
                prefix = "[None]";
            }
            DefaultMutableTreeNode namespace =
                new DefaultMutableTreeNode("Пространство имен: префикс = '" +
                    prefix + "', URI = '" + namespaceURI + "'");
            current.add(namespace);
        }

        // Обрабатываем атрибуты
        for (int i=0; i<atts.getLength(); i++) {
            DefaultMutableTreeNode attribute =
                new DefaultMutableTreeNode("Атрибут (имя = '" +
                                           atts.getLocalName(i) +
                                           "', значение = '" +
                                           atts.getValue(i) + "')");
            String attURI = atts.getURI(i);
            if (attURI.length() > 0) {
                String attPrefix =
                    (String)namespaceMappings.get(attURI);
                if (attPrefix.equals("")) {
                    attPrefix = "[None]";
                }
                DefaultMutableTreeNode attNamespace =
                    new DefaultMutableTreeNode("Пространство имен: префикс =
'" +
                        attPrefix + "', URI = '" + attURI + "'");
                attribute.add(attNamespace);
            }
            current.add(attribute);
        }
    }
```

Написать код для события окончания элемента гораздо проще. Поскольку нет необходимости выдавать какую-либо визуальную информацию, нужно лишь вернуться на один узел вверх по дереву, делая родительский элемент новым текущим узлом:

```
    public void endElement(String namespaceURI, String localName,
                           String qName)
        throws SAXException {

        // Поднимаемся вверх по дереву
        current = (DefaultMutableTreeNode)current.getParent();
    }
```

И еще одно последнее замечание, перед тем как перейти к данным элемента: вы, должно быть, заметили, что, имея URI пространства имен и уточненное имя элемента, можно вычислить префикс. Кроме того, на основе информации, переданной методу `startElement()`, можно узнать URI, не используя для этого карту отображения пространств имен. Это

совершенно справедливо, и очень сильно пригодится в коде примера. Однако в большинстве приложений эти обратные вызовы состоят из сотен и даже тысяч строк кода (или, что более верно, не сами обратные вызовы, а методы, вызываемые в коде обратных вызовов). В этих случаях анализ уточненного имени элемента не столь надежен, как хранение данных в собственной структуре. Другими словами, разбиение уточненного имени по двоеточию работает в простых приложениях, но в сложных (а значит и в более реалистичных) все обстоит не столь радужно.

Данные элемента

Когда идентифицированы начало и конец блока элемента и определен набор атрибутов элемента, следующим фрагментом важной информации являются данные, содержащиеся в самом элементе. Обычно это дополнительные элементы, текстовые данные либо комбинация того и другого. Для дополнительных элементов также выполняются соответствующие методы обратного вызова, и имеет место псевдорекурсия: элементы, вложенные в другие элементы, приводят к вызовам методов, «вложенных» в другие методы. В конце концов будут обнаружены текстовые данные. Как правило, именно эти данные имеют наибольшее значение для клиента XML: обычно происходит либо их визуализация, либо обработка с целью получения ответа от клиента.

В XML текстовые данные элементов передаются вызывающему приложению через метод обратного вызова `characters()`. Данный метод передает вызывающему приложению массив символов, а также индекс первого символа в массиве и количество символов для чтения. Создать на основе этого массива объект типа `String` и использовать данные очень просто:

```
public void characters(char[] ch, int start, int length)
    throws SAXException {

    String s = new String(ch, start, length);
    DefaultMutableTreeNode data =
        new DefaultMutableTreeNode("Символьные данные : '" + s + "'");
    current.add(data);
}
```

Этот, казалось бы, простой вызов часто приводит к значительному количеству недоразумений, поскольку стандарты интерфейса SAX строго не определяют, каким образом данный метод должен применяться в случае длинных последовательностей символьных данных. Другими словами, синтаксический анализатор может предпочесть вернуть весь фрагмент символьных данных за один вызов либо передать тот же фрагмент данных приложению за несколько вызовов метода. Для произвольного элемента этот метод может вообще не вызываться (если в элементе отсутствуют текстовые данные) либо может быть вызван

один или несколько раз. Разные анализаторы реализуют эту функциональность по-разному, зачастую используя алгоритмы, разработанные для повышения скорости синтаксического разбора. Ни в коем случае нельзя рассчитывать на то, что все текстовые данные элемента будут получены за один вызов метода; и наоборот, не следует считать, что непрерывная последовательность данных одного элемента непременно приведет к многократному вызову метода.

При написании обработчиков событий SAX следует помнить об иерархической структуре документа. Иначе говоря, не следует привыкать к тому, что всякий элемент *владеет* своими данными и *дочерними элементами*, – на деле он всего лишь является родительским элементом. Помните также, что синтаксический анализатор обрабатывает элементы, атрибуты и данные последовательно, в порядке их появления в тексте документа. Это может привести к несколько неожиданным результатам. Рассмотрим следующий фрагмент XML-документа:

```
<parent>Этот элемент содержит <child>вложенный текст </child>.</parent>
```

Забыв о том, что SAX выполняет анализ последовательно, вызывая методы по мере нахождения элементов и данных, как и о том, что XML-данные обрабатываются иерархически, можно предположить, что результат анализа фрагмента выглядит так (рис. 3.2).

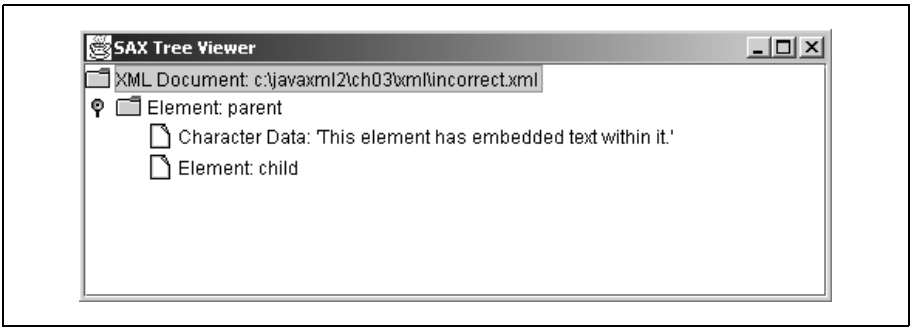

Рис. 3.2. Ожидаемое (и неверное) графическое дерево

Казалось бы, дерево выглядит логично, поскольку родительский элемент `parent` полностью «владеет» дочерним элементом `child`. На самом деле вызов пользовательского метода происходит для каждого события SAX, что приводит к созданию дерева, показанного на рис. 3.3.

SAX не забегает вперед при чтении, поэтому показанный здесь результат именно такой, какой можно было бы ожидать, если рассматривать XML-документ как последовательно обрабатываемые данные, без каких-либо дополнительных предположений, которые мы, люди, склонны делать. Об этом обстоятельстве очень важно помнить.

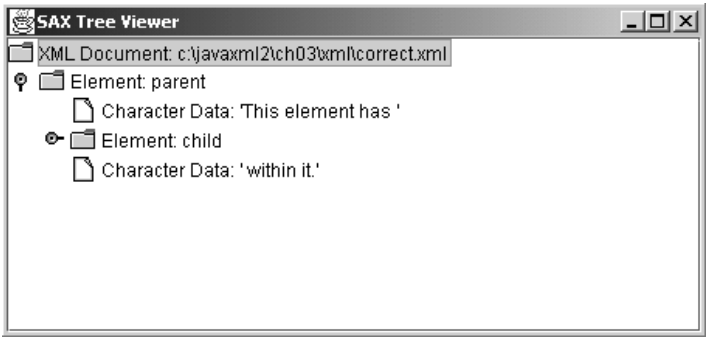

Рис. 3.3. Дерево, создаваемое в действительности

Внимание

В настоящее время ни Apache Xerces, ни какой-либо другой существующий анализатор по умолчанию не выполняет проверку действительности документа. В примере программы проверки действительности не происходит, т. к. ничего не было сделано, чтобы ее включить. Однако практически во всех случаях это не означает, что DTD или схема не обрабатываются. Заметьте, что даже без проверки действительности возникает исключительная ситуация, если не может быть найден системный идентификатор и не может быть разрешена ссылка на DTD (пример в разделе, посвященном `InputSource`). Поэтому очень важно видеть разницу между *проверкой действительности* и *обработкой DTD или схемы*. Вызов метода `ignorableWhitespace()` требует лишь *обработки* DTD или схемы, а не проверки действительности.

Наконец, довольно часто вызовы метода `characters()` используются и для обработки пробельных символов. Это вносит дополнительную путаницу, поскольку еще один метод SAX, `ignorableWhitespace()`, также сообщает о присутствии пробельных символов. К сожалению, во многих книгах (включая, я вынужден это признать, и первое издание моей книги «Java и XML») информация о пробельных символах либо неполна, либо полностью неверна. Поэтому предоставьте мне возможность исправить это. Самое главное, если нет ссылки на DTD или схему XML, метод `ignorableWhitespace()` никогда не должен вызываться. Точка.

Дело в том, что DTD (или схема) описывает модель содержимого элемента. Другими словами, в файле *JavaXML.dtd* элемент `contents` может иметь внутри себя только элементы `chapter`. Любое пустое пространство между началом элемента `contents` и началом элемента `chapter` (по логике вещей) игнорируется. Оно не значимо, т. к. поскольку в DTD не упоминается наличие каких-либо символьных данных (пробельных или иных). То же верно для пробельных символов между кон-

цом элемента chapter и началом следующего элемента chapter либо между концом элемента chapter и концом элемента contents. Поскольку ограничения (налагаемые DTD или схемой) говорят о том, что символьные данные недопустимы, эти пробельные символы и не могут быть значимыми. Однако *без* ограничения, задающего эту информацию анализатору, эти пробельные символы *не могут* интерпретироваться как не имеющие значения. Поэтому если удалить ссылку на DTD, пробельные символы, наличие которых ранее приводило к вызову метода ignorableWhitespace(), станут причиной вызова только метода characters(). Таким образом, пробельные символы никогда не могут быть однозначно проигнорированы либо однозначно учтены, – все зависит от ограничений, накладываемых (либо не накладываемых) на документ. Изменяя ограничения, можно изменить и значение пробельных символов.

Но давайте нырнем еще глубже. Если элемент может содержать только другие элементы, все довольно очевидно. Пробелы между элементами игнорируются. Однако рассмотрим смешанную модель содержимого:

```
<!ELEMENT p (#PCDATA | b* | i* | a* )>
```

Если вам кажется, что здесь написана ерунда, то вспомните об HTML. Эта модель представляет собой (частично) ограничения для элемента p или тега абзаца. Конечно же, внутри этого тега может существовать текст, а также элементы b (жирный шрифт), i (курсив) и a (ссылки). В этой модели между открывающим и закрывающим тегами p нет пробелов, которые могут игнорироваться (со ссылкой на DTD или без нее). Дело в том, что невозможно отличить пробелы, повышающие удобочитаемость текста от пробелов, которые обязательно должны присутствовать в документе. Например:

```
<p>
  Второе издание книги <i>Java и XML</i> теперь доступно как в магазинах,
    так и на сайте издательства O'Reilly
    <a href="http://www.oreilly.com">http://www.oreilly.com</a>.
</p>
```

В этом фрагменте XHTML пробелы между открывающим элементом p и открывающим элементом i не игнорируются, а значит, для них происходит вызов метода characters(). Если вы еще не полностью запутались (а я думаю, что это так), будьте готовы внимательно рассмотреть методы, связанные с символами. Это позволит моментально воспринять сведения о последнем из вызовов SAX, связанных с этим вопросом.

Игнорируемые пробельные символы

После завершения разговора о пробельных символах добавить реализацию метода ignorableWhitespace() совсем не сложно. Раз уж пробель-

ные символы заявлены как игнорируемые, код именно это и делает – он их игнорирует:

```
public void ignorableWhitespace(char[] ch, int start, int length)
    throws SAXException {

    // Пробельные символы игнорируются, поэтому их не отображаем
}
```

Пробельные символы передаются так же, как и символьные данные. Они могут быть получены в одном вызове, а могут быть переданы анализатором SAX за несколько вызовов. В любом случае тщательно придерживайтесь принципа не делать предположений и не считать пробельные символы текстовыми данными, чтобы избежать ошибок в приложениях.

Сущности

Как вы помните, в документе *contents.xml* есть лишь одна ссылка на сущность – `OReillyCopyright`. Ее анализ и интерпретация приводят к загрузке другого файла либо из локальной файловой системы, либо по адресу URI. Однако в используемой реализации класса `Reader` проверка действительности не включена. Часто из вида упускается тот аспект, что анализаторы без проверки действительности не обязаны интерпретировать ссылки на сущности и потому могут их пропускать. Раньше это вызывало немало головной боли, поскольку результаты работы анализатора могли просто не содержать ссылки на сущности, против ожидания. В SAX 2.0 эта задача изящно решается с помощью метода обратного вызова, который выполняется, если анализатор без проверки действительности пропускает сущность. Этот метод получает имя сущности, которое можно включить в вывод программы визуализации:

```
public void skippedEntity(String name) throws SAXException {
    DefaultMutableTreeNode skipped =
        new DefaultMutableTreeNode("Пропущена сущность: '" + name + "'");
    current.add(skipped);
}
```

Прежде чем обратиться к дереву в поисках узла `OReillyCopyright`, заметим, что наиболее авторитетные анализаторы не пропускают сущности, даже если они не проверяют действительность. Например, Apache Xerces никогда не вызовет этот метод; напротив, ссылка на сущность будет обработана и результат будет включен в данные, полученные после анализа. Другими словами, у анализаторов есть возможность использовать данный метод, но трудно найти ситуацию, в которой это происходит! Если у вас есть анализатор, реализующий такую функциональность, не забудьте, что передаваемый методу параметр не содер-

жит амперсанд и точку с запятой в ссылке на сущность. Для ссылки &OReillyCopyright; методу skippedEntity() передается только имя сущности OReillyCopyright.

Результаты

Наконец, следует зарегистрировать реализацию обработчика содержимого в созданном экземпляре XMLReader. Это делается с помощью метода setContentHandler(). Добавьте следующие строки к методу buildTree():

```
public void buildTree(DefaultTreeModel treeModel,
                      DefaultMutableTreeNode base, String xmlURI)
    throws IOException, SAXException {

    // Создаем экземпляры, необходимые для анализа
    XMLReader reader =
        XMLReaderFactory.createXMLReader(vendorParserClass);
    ContentHandler jTreeContentHandler =
        new JTreeContentHandler(treeModel, base);

    // Регистрируем обработчик содержимого
    reader.setContentHandler(jTreeContentHandler);

    // Регистрируем обработчик ошибок

    // Анализируем
    InputSource inputSource =
        new InputSource(xmlURI);
    reader.parse(inputSource);
}
```

Если вы набрали все эти методы обратного вызова, то сможете скомпилировать исходный файл SAXTreeViewer. После этого можно запустить демонстрацию работы SAX-совместимого визуализатора на примере созданного ранее XML-файла. Кроме того, убедитесь, что вы добавили путь к своему рабочему каталогу в переменную CLASSPATH. Полная команда Java должна выглядеть следующим образом :

```
C:\javaxml2\build>java javaxml2.SAXTreeViewer ..\ch03\xml\contents.xml
```

В результате будет создано окно Swing, в которое загружено содержимое XML-документа. Небольшая пауза при запуске, вероятно, означает, что ваша машина соединяется с Интернетом, чтобы интерпретировать ссылку на сущность OReillyCopyright. Если вы не подключены к Интернету, вернитесь к инструкциям из главы 2 и замените ссылку в DTD ссылкой на локальный файл с информацией о правообладании. В любом случае будет выведено дерево, подобное приведенному на рис. 3.4.

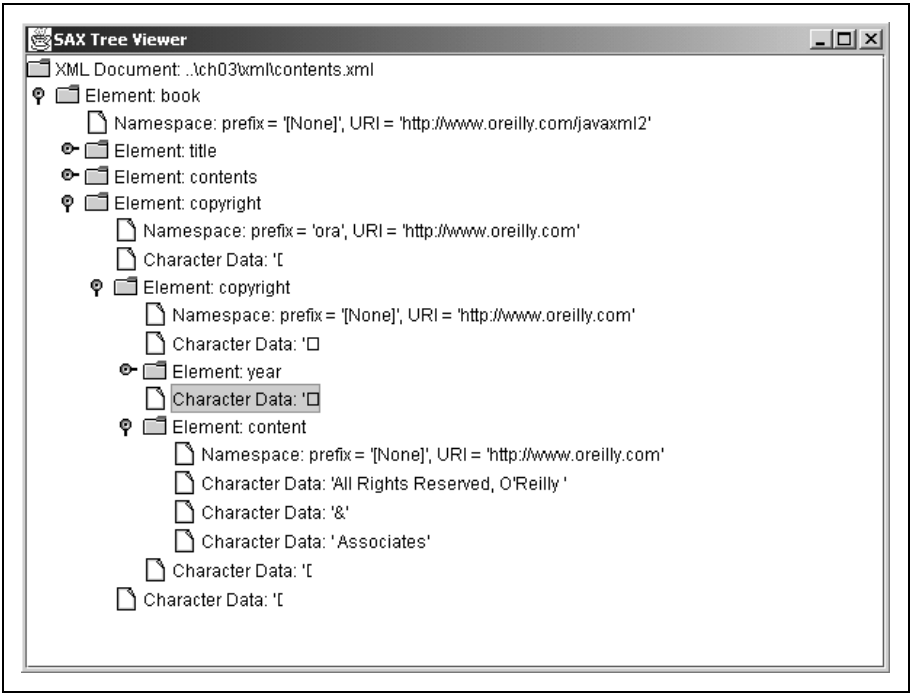

Рис. 3.4. SAXTreeViewer в действии

Необходимо обратить внимание на пару моментов: во-первых, отсутствуют окружающие элементы пробельные символы, поскольку наличие DTD и строгой модели содержимого требует, чтобы пробелы игнорировались (что и происходит при вызове метода `ignorableWhitespace()`). Во-вторых, интерпретируется ссылка на сущность, и можно увидеть содержимое файла *copyright.xml*, вложенное в древовидную структуру. Кроме того, поскольку сам этот файл не имеет DTD, пробельные символы, которые могли быть проигнорированы, были восприняты как символьные данные благодаря обратному вызову `characters()`. Это привело к появлению нескольких странных управляющих символов в значениях текстовых узлов дерева (как правило, причиной их появления служат символы возврата каретки, содержащиеся в исходном документе). Наконец, обратите внимание на то, что текст «O'Reilly & Associates» из файла *copyright.xml* получен посредством трех вызовов метода `characters()`. Отличная иллюстрация того, что единый блок текстовых данных может передаваться приложению по частям. В данном случае анализатор использовал в качестве разделителя символьную сущность (`&`), что является распространенной практикой. В любом случае попробуйте запустить программу просмотра для различных XML-документов и вы увидите, как меняются выводимые данные.

Только что вы видели, как SAX-совместимый анализатор обрабатывает корректный XML-документ. Также вы должны были получить представление об обратных вызовах, которые происходят в процессе анализа документа, и о том, как приложение может использовать эти обратные вызовы для получения информации об XML-документе в процессе его анализа. В следующей главе рассмотрена проверка действительности XML-документа с помощью дополнительных классов SAX, созданных для обработки DTD. Однако перед этим обратимся к вопросу о том, что происходит, когда XML-документ не является действительным, и к каким ошибкам может привести это обстоятельство.

Обработчики ошибок

В дополнение к интерфейсу `ContentHandler` для обработки событий, возникающих при анализе документа, SAX предоставляет интерфейс `ErrorHandler`, который может быть реализован для обработки различных ошибочных ситуаций, которые могут возникать в процессе анализа. Этот класс работает таким же образом, как и только что созданный обработчик документа, но он определяет только три метода обратного вызова. С помощью этих трех методов SAX-совместимые анализаторы перехватывают все возможные ошибочные ситуации и выдают сообщения об их возникновении. Посмотрим на интерфейс `ErrorHandler`:

```
public interface ErrorHandler {
    public abstract void warning (SAXParseException exception)
throws SAXException;
    public abstract void error (SAXParseException exception)
throws SAXException;
    public abstract void fatalError (SAXParseException exception)
throws SAXException;
}
```

Каждый метод получает информацию о возникшей ошибке или предупреждении через объект класса `SAXParseException`. Этот объект хранит номер строки, в которой была обнаружена ошибка, URI обрабатываемого документа (которым может быть как анализируемый документ, так и внешняя ссылка в этом документе), а также обычную информацию по исключительной ситуации – сообщение об ошибке и отображаемый стек вызовов. Кроме того, каждый метод может генерировать исключение `SAXException`. На первый взгляд это может показаться странным – обработчик исключения, который генерирует исключение? Имейте в виду, что каждый обработчик получает исключение, возникающее при анализе документа. Это может быть предупреждение, которое не должно вызывать прекращение процесса анализа, или ошибка, которая должна быть обработана, чтобы анализ документа мог продолжаться. Однако методу обратного вызова может потребоваться выполнять операцию системного ввода/вывода или другую операцию,

которая может сгенерировать исключение, и необходимо иметь возможность передать информацию о проблемах, возникших в результате этих действий, по цепочке вызовов приложению. Это достигается с помощью исключения SAXException, которое метод обработчика ошибок может сгенерировать.

В качестве примера рассмотрим обработчик ошибок, получающий сообщения об ошибках и записывающий их в специальный журнал. Необходимо, чтобы данный метод имел возможность либо добавлять записи в существующий журнал, либо создавать новый журнал ошибок в локальной файловой системе. Если в процессе анализа XML-документа генерируется предупреждение, то данный метод получает соответствующее уведомление. Смысл передачи предупреждения в том, чтобы предоставить информацию методу обратного вызова, а затем продолжить анализ документа. Однако если обработчик ошибок не может записать сообщение в журнал регистрации ошибок, ему может потребоваться сообщить анализатору и приложению о том, что процесс анализа следует прервать. Такого поведения можно добиться путем реализации перехвата исключений системы ввода/вывода и передачи их вызывающему приложению. В результате срабатывания этого механизма процесс анализа документа будет остановлен. Этот общий сценарий объясняет, почему обработчики ошибок должны иметь возможность генерировать исключения (пример 3.2).

Пример 3.2. Обработчик ошибок, который может генерировать исключение SAXException

```
public void warning(SAXParseException exception)
    throws SAXException {

    try {
        FileWriter fw = new FileWriter("error.log");
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write("Предупреждение: " + exception.getMessage() + "\n");
        bw.flush();
        bw.close();
        fw.close();
    } catch (IOException e) {
        throw new SAXException("Невозможно произвести запись в журнал
регистрации ошибок", e);
   }
}
```

Учтя это обстоятельство, теперь можно определить каркас реализации обработчика ошибок ErrorHandler и зарегистрировать его в экземпляре анализатора уже известным нам способом. Чтобы эта книга не стала учебником по Swing, методы будут просто останавливать анализ и отображать предупреждения и сообщения об ошибках в окне терминала. Во-первых, добавим еще один закрытый класс в конец исходного файла *SAXTreeViewer.java*:

```
class JTreeErrorHandler implements ErrorHandler {

    // Реализация метода

}
```

Чтобы пользовательский обработчик ошибок был в действительности задействован, необходимо зарегистрировать его в экземпляре SAX-класса Reader. Делается это с помощью метода `setErrorHandler()` интерфейса `XMLReader`, вызываемого в методе `buildTree()` нашего примера:

```
public void buildTree(DefaultTreeModel treeModel,
                      DefaultMutableTreeNode base, String xmlURI)
    throws IOException, SAXException {

    // Создаем экземпляры, необходимые для анализа
    XMLReader reader =
        XMLReaderFactory.createXMLReader(vendorParserClass);
    ContentHandler jTreeContentHandler =
        new JTreeContentHandler(treeModel, base);
    ErrorHandler jTreeErrorHandler = new JTreeErrorHandler();

    // Регистрируем обработчик содержимого
    reader.setContentHandler(jTreeContentHandler);

    // Регистрируем обработчик ошибок
    reader.setErrorHandler(jTreeErrorHandler);

    // Анализируем
    InputSource inputSource =
        new InputSource(xmlURI);
    reader.parse(inputSource);
}
```

Наконец, взглянем на код трех методов интерфейса `ErrorHandler`.

Предупреждения

Всякий раз, когда возникает предупреждение (согласно определению спецификации XML 1.0), вызывается соответствующий метод зарегистрированного обработчика ошибок. Существует целый ряд ситуаций, в которых могут возникать предупреждения. Все они, однако, связаны с DTD и действительностью документа, и о них рассказывается в следующей главе. Пока же необходимо определить простой метод, который при возникновении предупреждения отображает номер строки, URI и соответствующее сообщение. Поскольку (в целях демонстрации) я хочу, чтобы любое предупреждение приводило к остановке процесса анализа документа, в этом фрагменте кода генерируется исключение `SAXException`, и вызывающее приложение может элегантно завершить работу, освободив все используемые ресурсы:

```
public void warning(SAXParseException exception)
    throws SAXException {
```

```
    System.out.println("**Предупреждение при анализе документа**\n" +
                       "  Строка:    " +
                          exception.getLineNumber() + "\n" +
                       "  URI:     " +
                          exception.getSystemId() + "\n" +
                       "  Сообщение: " +
                          exception.getMessage());
    throw new SAXException("Получено предупреждение");
}
```

Некритические ошибки

Ошибки, которые возникают в процессе анализа документа и могут быть обработаны, но представляют собой нарушение какого-либо положения спецификации XML, считаются некритическими. Обработчик ошибок должен, по крайней мере, всегда заносить их в журнал регистрации ошибок, поскольку они, если и не являются основанием для прекращения анализа документа, обычно достаточно серьезны, чтобы проинформировать об этом пользователя или администратора приложения. Как и предупреждения, большинство некритических ошибок связано с проверкой действительности, поэтому подробно они будут рассмотрены в следующей главе. Опять же, как и в случае с предупреждениями, обработчик ошибок в этом примере просто выдает информацию, полученную методом обратного вызова, и завершает процесс анализа:

```
public void error(SAXParseException exception)
    throws SAXException {

    System.out.println("**Ошибка при анализе**\n" +
                       "  Строка:    " +
                          exception.getLineNumber() + "\n" +
                       "  URI:     " +
                          exception.getSystemId() + "\n" +
                       "  Сообщение: " +
                          exception.getMessage());
    throw new SAXException("Возникла ошибка");
}
```

Критические ошибки

Критическими ошибками являются такие ошибки, которые неизбежно вызывают остановку анализатора. Обычно они связаны с документами, которые не являются корректными, и делают дальнейший анализ документа либо бессмысленной тратой времени, либо технически невозможным. Когда возникает критическая ошибка, обработчик ошибок практически всегда должен сообщать об этом пользователю или администратору приложения. Без постороннего вмешательства это может привести к внештатной остановке приложения. В нашем

примере обработчик будет просто эмулировать поведение двух других методов, останавливая процесс анализа и выдавая сообщение об ошибке на экран в том случае, если возникает критическая ошибка:

```
public void fatalError(SAXParseException exception)
    throws SAXException {

    System.out.println("**Анализируем критическую ошибку**\n" +
                       "  Строка:    " +
                          exception.getLineNumber() + "\n" +
                       "  URI:     " +
                          exception.getSystemId() + "\n" +
                       "  Сообщение: " +
                          exception.getMessage());
    throw new SAXException("Возникла критическая ошибка");
}
```

Написав этот третий обработчик ошибок, вы сможете скомпилировать исходный код примера и вновь запустить его для анализа XML-документа. Полученный результат ни в чем не должен отличаться от предыдущего, поскольку в этом XML-документе нет ошибок, о которых можно было бы сообщить. Далее я покажу, как добиться того, чтобы возникли некоторые из этих ошибок (разумеется, в целях тестирования!).

Внесение ошибок в данные

Теперь, когда у нас имеется несколько обработчиков ошибок, мы искусственно создадим несколько проблем и посмотрим на эти обработчики в действии. Большинство предупреждений и некритических ошибок связано с вопросами действительности документа, о которых речь пойдет в следующей главе (где подробно рассматривается проверка действительности). Тем не менее, существует одна некритическая ошибка, которая может возникнуть в документе, не проверявшемся на действительность. Она связана с версией XML, которая указана в документе. Чтобы увидеть эту ошибку, внесите следующее изменение в первую строку примера содержания книги в формате XML:

```
<?xml version="1.2"?>
```

Теперь нужно запустить программу, выполняющую синтаксический анализ измененного XML-документа. Результат должен быть подобен этому:

```
C:\javaxml2\build>java javaxml2.SAXTreeViewer ..\ch03\xml\contents.xml
**Ошибка при анализе **
  Строка:    1
  URI:     file:///C:/javaxml2/ch03/xml/contents.xml
  Сообщение: XML version "1.2" is not supported.
org.xml.sax.SAXException: Возникла ошибка
```

Для случаев, когда в документе, обрабатываемом анализатором, заявлена версия XML выше той, которая поддерживается системой, спецификация XML 1.0 предписывает выдачу сообщения о некритической ошибке. Таким образом приложение получает возможность узнать, что более новые возможности, которые, как предполагается, используются в документе, могут быть недоступными в анализаторе и той версии XML, которую он поддерживает. Поскольку анализ документа продолжается, эта ошибка не является критической. Тем не менее, поскольку она имеет важное значение для работы с документом (например, использование другой версии может быть связано с более новым синтаксисом, который способен привести к ошибкам на более поздних стадиях анализа), она считается более серьезной, чем простое предупреждение. Вот почему вызывается метод `error()` и выдается сообщение об ошибке, и процесс анализа в примере останавливается.

Все остальные предупреждения и некритические ошибки, имеющие значение, будут обсуждаться в следующей главе; кроме того, существует множество критических ошибок, которые может содержать XML-документ, не проверявшийся на действительность. Они связаны с тем, что XML-документ не является корректным. В анализаторах XML не предусмотрена функциональность, позволяющяя решать такие проблемы либо предлагать способы их решения, поэтому ошибка в синтаксисе приводит к остановке анализа документа. Простейший способ продемонстрировать одну из таких ошибок – это ввести ошибки в XML-документ. Приведите объявление XML к прежнему виду, чтобы оно указывало на XML версии 1.0, и измените XML-документ следующим образом:

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">

<!-- "Java и XML", Содержание -->
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
>
  <!-- Обратите внимание на пропущенный слэш у закрывающего элемента title -->
  <title ora:series="Java">Java and XML<title>

  <!-- Остальные данные -->
</book>
```

Теперь этот документ не является корректным. Чтобы увидеть критическую ошибку, которая генерируется в процессе анализа данного документа, запустите программу `SAXTreeViewer` для этого измененного файла и вы увидите следующие результаты:

```
C:\javaxml2\build>java javaxml2.SAXTreeViewer ..\ch03\xml\contents.xml
**Анализируем критические ошибки**
  Строка:    23
  URI:     file:///C:/javaxml2/ch03/xml/contents.xml
```

```
    Сообщение: The element type "title" must be terminated by the matching
            end-tag "</title>".
org.xml.sax.SAXException: Возникла критическая ошибка
```

Анализатор сообщает о неверном завершении элемента title. Данная критическая ошибка – в точности та самая, которую мы и ожидали, после возникновения этой ошибки анализ документа не может продолжаться. Применяя этот обработчик ошибок, мы понимаем, что идет не так в процессе анализа и как эти события можно обрабатывать. В главе 4 мы вновь обратимся к обработчику ошибок и его методам и рассмотрим проблемы, о которых может сообщить анализатор, проверяющий действительность (документа).

Советы разработчикам

Прежде чем закончить введение в анализ XML-документов, предостерегу читателей от некоторых ошибок. Эти предостережения помогут избежать часто встречающихся ловушек, связанных с использованием SAX. Данная тема получит свое развитие для других API в соответствующих разделах.

Мой анализатор не поддерживает SAX версии 2.0

Пусть не отчаиваются те, кому приходится работать с анализатором, поддерживающим SAX 1.0. Во-первых, всегда есть возможность поменять анализатор; поддержка соответствия текущим стандартам SAX является важной характеристикой анализатора XML, и если производитель этого соответствия не обеспечивает, то могут возникнуть и другие проблемы с данным продуктом. Тем не менее, конечно, иногда мы вынуждены использовать анализатор по причине наличия разработанного ранее кода или приложений, но и из этих ситуаций существует выход.

SAX 2.0 содержит вспомогательный класс org.xml.sax.helpers.ParserAdapter, который может заставить анализатор, реализующий SAX 1.0, вести себя подобно реализации интерфейса XMLReader в SAX 2.0. Этот удобный класс предоставляет метод, принимающий в качестве аргумента экземпляр интерфейса Parser версии SAX 1.0. Инициализированный таким образом класс может использоваться приложением в обычной манере. Это позволяет создавать обработчик ContentHandler (являющийся конструкцией SAX 2.0), а также обрабатывать должным образом все события, связанные с пространствами имен (это тоже возможность из SAX 2.0). Единственная возможность, которая не будет поддерживаться, связана с уведомлениями о сущностях, пропущенных анализатором, поскольку эта возможность не была доступна в SAX 1.0 ни в какой форме и не может эмулироваться классом адаптера версии 2.0. В примере 3.3 показано, как действует этот класс.

Пример 3.3. Использование SAX 1.0 с конструкциями из SAX 2.0

```
try {
    // Регистрируем анализатор с SAX
    Parser parser =
         ParserFactory.makeParser(
            "org.apache.xerces.parsers.SAXParser");

    ParserAdapter myParser = new ParserAdapter(parser);

    // Регистрируем обработчик документа
    myParser.setContentHandler(contentHandler);

    // Регистрируем обработчик ошибок
    myParser.setErrorHandler(errHandler);

    // Анализируем документ
    myParser.parse(uri);

    } catch (ClassNotFoundException e) {
          System.out.println(
                 "Невозможно найти класс анализатора.");
    } catch (IllegalAccessException e) {
          System.out.println(
                "Недостаточно привилегий для загрузки класса анализатора.");
    } catch (InstantiationException e) {
          System.out.println(
               "Невозможно создать экземпляр класса анализатора.");
    } catch (ClassCastException e) {
          System.out.println(
                 "Отсутствует реализация класса org.xml.sax.Parser
                  в анализаторе");
    } catch (IOException e) {
          System.out.println("Ошибка при чтении URI: " + e.getMessage());
    } catch (SAXException e) {
          System.out.println("Ошибка при анализе: " + e.getMessage());
    }
```

Если вы новичок в SAX и этот пример не особо понятен – не беспокойтесь. Вы работаете с новейшей и наилучшей версией SAX (2.0), и вам, вероятнее всего, никогда не придется писать подобный код. Этот код полезен только в тех случаях, когда необходимо использовать синтаксический анализатор, реализующий интерфейс SAX 1.0.

XMLReader: повторное использование и число точек входа

Одной из самых приятных особенностей языка Java является легкость повторного использования объектов и оптимизация памяти, с этим связанная. SAX-совместимые анализаторы не являются исключением. Когда создан экземпляр объекта `XMLReader`, этот экземпляр анализатора может быть использован для анализа десятков и даже сотен

XML-документов. Анализатору можно непрерывно передавать разнообразные документы и объекты InputSource, что позволяет решать с его помощью множество разнообразных задач. Однако классы чтения имеют лишь одну *точку входа*. Это означает, что анализатор нельзя использовать до тех пор, пока анализ требуемого документа или исходных данных не завершится. Другими словами, в процесс анализа невозможно зайти заново. Для тех из вас, кто склонен к применению рекурсивных методов, это определенно может стать ловушкой! При первой же попытке использовать анализатор, который находится в процессе анализа другого документа, возникнет довольно неприятное исключение SAXException, и анализ будет остановлен. Какой же урок стоит извлечь из этого? Единовременно выполняйте синтаксический разбор только одного документа, в противном случае для инициализации нескольких экземпляров анализатора будут расходоваться дополнительные ресурсы.

Недействительный указатель позиции в документе

Другая, на первый взгляд безобидная, но опасная особенность, связанная с событиями SAX, относится к реализации интерфейса Locator, передаваемой анализатору посредством метода обратного вызова setDocumentLocator(). Этот объект указывает приложению на точку события SAX и полезен при принятии решений о продолжении анализа и о том, как реагировать на события. Однако эта точка действительна только на время существования объекта, реализующего интерфейс ContentHandler: когда анализ завершен, указатель позции перестает быть действительным (включая и случай, когда начинается новый процесс анализа). Ошибка, которую совершают многие новички в XML, заключается в хранении ссылки на указатель в переменной-члене класса *за пределами* метода обратного вызова:

```
public void setDocumentLocator(Locator locator) {
    // Сохранение указателя позиции в классе за пределами ContentHandler
    myOtherClass.setLocator(locator);
}
...

public myOtherClassMethod() {
    // Попытка использовать указатель позиции за пределами ContentHandler
    System.out.println(locator.getLineNumber());
}
```

Это чрезвычайно плохая идея, потому что данный указатель позиции теряет смысл, как только случается выход за пределы видимости экземпляра реализации ContentHandler. Зачастую использование переменной-члена, получаемой в результате этой операции, приводит не только к передаче ошибочной информации приложению, но и к возникновению исключительной ситуации в выполняемом коде. Другими

словами, используйте этот объект локально, а не глобально. В классе реализации JTreeContentHandler указатель сохраняется в переменной-члене. Затем его можно корректно использовать, например, для выдачи номера строки каждого элемента:

```
public void startElement(String namespaceURI, String localName,
                         String rawName, Attributes atts)
    throws SAXException {

    DefaultMutableTreeNode element =
        new DefaultMutableTreeNode("Элемент: " + localName +
            " в строке " + locator.getLineNumber());
    current.add(element);
    // Остальной код...
}
```

Чтение за пределами данных

Метод characters() получает массив символов, а также параметры start и length, указывающие, с какого индекса следует начинать чтение данных из массива и какова длина сегмента данных. Это может привести к недоразумению: обычной ошибкой является написание кода для чтения данных из массива символов, подобного приведенному ниже:

```
public void characters(char[] ch, int start, int length)
    throws SAXException {

    for (int i=0; i<ch.length; i++)
        System.out.print(ch[i]);
}
```

Ошибка здесь заключается в том, что чтение производится от начала до конца массива символов. Эта естественная ошибка является следствием многолетней практики написания циклов перебора элементов массивов на Java, C или других языках программирования. Однако в случае события SAX эта ошибка весьма серьезна. SAX-совместимые анализаторы должны указывать начальную позицию и длину сегмента данных, которые любая циклическая конструкция должна использовать при чтении данных из массива. Это позволяет выполнять низкоуровневые манипуляции с текстовыми данными в целях оптимизации производительности синтаксического анализатора, такие как опережающее чтение данных в документе, а также повторное использование массивов. Все это является нормальным явлением в SAX, поскольку предполагается, что вызывающее приложение не будет «забегать вперед» относительно конечного индекса, передаваемого методу обратного вызова.

Ошибки, подобные показанной в примере, могут привести к выдаче невнятных данных на экран или к использованию таких данных в вызывающей программе, и почти всегда вызывают проблемы в приложе-

ниях. Конструкция цикла выглядит вполне нормально и компилируется «без сучка и задоринки», поэтому отыскать такую ловушку может оказаться весьма трудно. Чтобы не иметь подобных проблем, можно просто преобразовать данные в строку `String`, использовать ее и никогда не волноваться:

```
public void characters(char[] ch, int start, int length)
    throws SAXException {

    String data = new String(ch, start, length);
    // Используем эту строку
}
```

Что дальше?

Теперь, ознакомившись с интерфейсом SAX, мы готовы перейти к изучению дополнительных возможностей. В их число входит установка свойств и возможностей, проверка действительности и обработка пространств имен, а также интерфейсы `EntityResolver` и `DTDHandler`. Кроме того, вы увидите многие используемые реже (но, тем не менее, ценные) возможности простого интерфейса прикладного программирования для XML, а также необязательные дополнения к SAX, такие как фильтры и пакет `org.xml.sax.ext`. Тем из вас, кто использует SAX в своих приложениях, это позволит несколько обойти коллег-разработчиков. Не закрывайте свой редактор и переворачивайте страницу.

4

- *Свойства и возможности*
- *И опять обработчики*
- *Фильтры и объекты Writer*
- *И снова обработчики*
- *Советы разработчикам*
- *Что дальше?*

Расширенный SAX

Предыдущая глава была хорошим введением в SAX. Однако существует еще несколько тем, которые дополнят ваши знания о SAX. Хотя я и назвал эту главу «Расширенный SAX», не пугайтесь. С таким же успехом ее можно было бы назвать «Реже используемые возможности SAX, которые, тем не менее, очень важны». При написании этих двух глав я следовал принципу 80/20. Вероятно, 80% читателей, никогда не воспользуются приведенным ниже материалом – для их нужд хватит главы 3. Однако продвинутые пользователи, изо дня в день работающие с XML, найдут в этой главе некоторые тонкости SAX.

Начнем с настройки – покажем, как с ее помощью заставить SAX выполнять то, что нам нужно. Затем перейдем к другим обработчикам: `EntityResolver` и `DTDHandler`, оставшимся с предыдущей главы. К этому моменту вы будете иметь полное представление о стандарте SAX 2.0. Однако мы пойдем дальше и рассмотрим некоторые расширения SAX, включая классы записи, которые можно использовать совместно с SAX, а также механизмы фильтрации. Наконец, я познакомлю вас с некоторыми новыми обработчиками: `LexicalHandler` и `DeclHandler`, и покажу вам, как они используются. Когда все будет сказано и сделано (включая еще один раздел «Советы разработчикам»), вы будете готовы покорить мир с помощью анализатора и классов SAX. Так что облачайтесь в сверкающий космический скафандр и хватайтесь за штурвал... гхм. Да, что-то я увлекся завоеванием мира. В любом случае перейдем к работе.

Свойства и возможности

В связи с обилием связанных с XML спецификаций и технологий, исходящих от консорциума World Wide Web (W3C), введение поддержки какой-либо новой возможности или свойства анализатора XML стало затруднительным. Во многих реализациях синтаксических анализаторов добавлены фирменные расширения или методы – ценой потери переносимости пользовательского кода. В то время как эти программные продукты могут реализовывать интерфейс SAX `XMLReader`, методы, отвечающие за включение проверки действительности документа и действительности по схеме, поддержку пространств имен, а также другие основные возможности не являются стандартными для всех анализаторов. С целью решения этой проблемы в SAX 2.0 определен стандартный механизм для установки свойств и возможностей анализатора, что позволяет добавлять новые свойства и возможности, если они утверждены консорциумом W3C, без использования фирменных расширений или методов.

Установка свойств и возможностей

К счастью для нас, интерфейс `XMLReader` в SAX 2.0 предоставляет методы, необходимые для установки свойств и возможностей. Таким образом, путем внесения незначительных изменений в существующий код можно включать проверку действительности документа, задавать разделитель для пространств имен, а также устанавливать значения прочих свойств и возможностей. Методы, применяемые для этих целей, приведены в табл. 4.1.

Таблица 4.1. Методы для работы со свойствами и возможностями

Метод	Возвращает	Параметры	Синтаксис
`setProperty()`	`void`	`String propertyID, Object value`	`parser.setProperty("[Property URI]", propertyValue);`
`setFeature()`	`void`	`String featureID, boolean state`	`parser.setFeature("[Feature URI]", featureState);`
`getProperty()`	`Object`	`String propertyID`	`Object propertyValue = parser.getProperty("[Property URI]");`
`getFeature()`	`boolean`	`String featureID`	`boolean featureState = parser.getFeature("[Feature URI]");`

Для каждого из этих методов идентификатор конкретного свойства или возможности представлен URI. Полный перечень возможностей и свойств приведен в приложении B «Возможности и свойства SAX 2.0». Также должна быть доступна дополнительная документация по свойствам и возможностям, реализованным разработчиками в конкретном

анализаторе XML. В данном случае URI сходны с URI пространств имен: они используются только в качестве связей для конкретных возможностей. Хорошие анализаторы гарантируют, что для интерпретации этих возможностей и свойств вам не потребуется доступ в сеть Интернет; поэтому считайте, что это обычные константы, имеющие форму URI. Эти методы вызываются обычным образом, а URI интерпретируется локально, обычно как константа, предписывающая анализатору совершение определенного действия.

Примечание

Не стоит «проверять существование» этих URI свойств и возможностей, набирая их в броузере. Очень часто это приводит к появлению ошибки 404 *Not Found*. Очень многие броузеры сообщали мне об этом, настаивая, что введенные URI недействительны. Это совсем не так. В данном случае URI – просто идентификатор, и, как я уже говорил, он, как правило, интерпретируется локально. Поверьте мне: просто используйте эти URI и доверьте анализатору выполнение того, что необходимо сделать.

В контексте конфигурации анализатора *свойство* (*property*) требует использования некоторого объекта. Например, для обработки лексических конструкций в качестве значения соответствующего свойства может быть передан экземпляр объекта DOM `Node`. *Возможность* (*feature*), напротив, – это флаг, указывающий анализатору, должен ли производиться определенный тип обработки. Распространенными возможностями являются проверка действительности, поддержка пространств имен и включение внешних параметрических сущностей.

Наиболее удобный аспект указанных методов состоит в том, что они предоставляют возможность простого добавления или изменения настроек. Хотя новые или обновленные возможности потребуют добавления соответствующего кода для их поддержки в реализацию анализатора, методы, с помощью которых осуществляется доступ к возможностям и свойствам, остаются простыми и стандартными; изменениям могут подвергаться только URI. Вне зависимости от сложности (или неясности) новых идей, связанных с XML, четырех методов этого надежного набора должно быть достаточно, чтобы синтаксические анализаторы могли претворять новые идеи в жизнь.

Возможности и свойства SAX

В большинстве случаев мы имеем дело со стандартными возможностями и свойствами, определенными в SAX. Это те возможности и свойства, которые должны быть доступны в любом дистрибутиве SAX и которые должен поддерживать любой анализатор, совместимый с SAX. Кроме того, это позволяет создавать независимый от платформы код, поэтому я рекомендую везде, где только возможно, использовать свойства и функции, стандартные для SAX.

Проверка действительности

Чаще всего вы будете прибегать к возможности проверки действительности. URI для нее – *http://xml.org/sax/features/validation* и, как можно догадаться, с его помощью можно включать или выключать проверку действительности в анализаторе. Например, если необходимо включить проверку действительности в анализируемом примере из предыдущей главы (вспомните визуализацию в Swing), измените исходный файл *SAXTreeViewer.java* следующим образом:

```
public void buildTree(DefaultTreeModel treeModel,
                      DefaultMutableTreeNode base, String xmlURI)
    throws IOException, SAXException {

    // Создаем экземпляры, необходимые для анализа
    XMLReader reader =
        XMLReaderFactory.createXMLReader(vendorParserClass);
    ContentHandler jTreeContentHandler =
        new JTreeContentHandler(treeModel, base, reader);
    ErrorHandler jTreeErrorHandler = new JTreeErrorHandler();

    // Регистрируем обработчик содержимого
    reader.setContentHandler(jTreeContentHandler);

    // Регистрируем обработчик ошибок
    reader.setErrorHandler(jTreeErrorHandler);

    // Запрашиваем проверку действительности
    reader.setFeature("http://xml.org/sax/features/validation", true);

    // Анализируем документ
    InputSource inputSource =
        new InputSource(xmlURI);
    reader.parse(inputSource);
}
```

Внеся изменения, скомпилируйте и запустите программу. Ничего не произошло, верно? Нет ничего удивительного – XML-файл, который мы использовали, действителен, он полностью соответствует указанному DTD-определению. Однако исправить это довольно просто. Внесите в XML-файл следующие изменения (обратите внимание, что элемент из объявления `DOCTYPE` больше не совпадает с настоящим корневым элементом, т. к. XML чувствителен к регистру символов):

```
<?xml version="1.0"?>
<!DOCTYPE Book SYSTEM "DTD/JavaXML.dtd">

<!-- “Java и XML”, Содержание -->
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
>
```

Теперь запустите программу для этого измененного документа. Поскольку проверка действительности включена, вы получите распечатку стека, свидетельствующую об ошибке. А раз методы нашего обработчика ошибок умеют только это, мы больше ничего и не ожидали увидеть:

```
C:\javaxml2\build>java javaxml2.SAXTreeViewer
    c:\javaxml2\ch04\xml\contents.xml
**Ошибка при анализе**
  Строка:    7
  URI:     file:///c:/javaxml2/ch04/xml/contents.xml
  Сообщение: Document root element "book", must match DOCTYPE root "Book".
org.xml.sax.SAXException: Error encountered
        at javaxml2.JTreeErrorHandler.error(SAXTreeViewer.java:445)
[Продолжение распечатки стека...]
```

Запомните, что включение и выключение проверки действительности не влияет на процесс обработки DTD; я уже говорил об этом в предыдущей главе и хочу еще раз напомнить об этой особенности. Чтобы лучше это понять, отключите проверку действительности (закомментируйте строку, в которой происходит установка значения для возможности, либо установите значение «false») и запустите программу вновь. И хотя DTD-определение обработано, как можно видеть по интерпретированной сущности `OReillyCopyright`, ошибок при анализе документа не возникло. Таково различие между обработкой DTD и проверкой действительности XML-документа относительно этого DTD. Запомните, поймите и повторяйте это время от времени; в будущем вы сбережете уйму времени.

Пространства имен

Помимо проверки действительности чаще всего вы будете иметь дело с пространствами имен. С пространствами имен связаны две возможности: одна включает и отключает обработку пространств имен, а другая указывает, нужно ли сообщать о префиксах, связанных с объявлением пространств имен. Обе возможности тесно связаны друг с другом, и их всегда нужно «переключать» в соответствии с табл. 4.2.

Таблица 4.2. Значения для возможностей, связанных с пространствами имен

Значение для обработки пространств имен	Значение для отслеживания префиксов пространств имен
Истина	Ложь
Ложь	Истина

Это правило вполне осмысленно: если включена обработка пространств имен, объявления в `xmlns`-стиле не должны передаваться приложению в качестве атрибутов, т. к. они полезны только для обработки

пространств имен. Предположим, вы не хотите, чтобы происходила обработка пространств имен (или хотите самостоятельно производить эту обработку); есть смысл получать объявления xmlns в виде атрибутов и использовать точно так же, как и другие атрибуты. Однако если эти две возможности не будут синхронизированы (либо обе будут включены, либо обе выключены), у вас могут начаться неприятности!

Имеет смысл создать небольшой вспомогательный метод, гарантирующий, что эти две возможности синхронизированы. Я часто пользуюсь приведенным ниже методом именно в таких целях:

```
private void setNamespaceProcessing(XMLReader reader, boolean state)
    throws SAXNotSupportedException, SAXNotRecognizedException {

    reader.setFeature(
        "http://xml.org/sax/features/namespaces", state);
    reader.setFeature(
        "http://xml.org/sax/features/namespace-prefixes", !state);
}
```

Метод поддерживает верные настройки для обеих возможностей и в коде приложения он может заменить два вызова метода setFeature(). Лично я обращался к этой возможностью менее десяти раз за два года; практически всегда меня устраивали значения по умолчанию (производится обработка пространств имен, но не отслеживание префиксов). Если речь не идет о создании низкоуровневых приложений, в которых не требуется обработка пространств имен, либо требуется увеличение скорости, достигаемое за счет отключения обработки, и если нет необходимости обрабатывать пространства имен самостоятельно, не стоит особенно заострять внимание на этих возможностях.

Тем не менее, этот фрагмент кода открывает довольно важный аспект возможностей и свойств: вызов методов, связанных со свойствами и возможностями, может привести к возникновению исключений SAXNotSupportedException и SAXNotRecognizedException. Оба они принадлежат пакету org.xml.sax и должны быть импортированы в любой код SAX, в котором они используются. Возникновение первого исключения говорит о том, что анализатор знает о возможности (или свойстве), но не поддерживает ее. Это исключение не часто встречается даже в анализаторах среднего качества, но оно часто возникает, когда для стандартного свойства или возможности еще не написан код. Поэтому вызов метода setFeature() для возможности, связанной с обработкой пространств имен, иногда приводит к исключению SAXNotSupportedException, если используется анализатор, находящийся в стадии разработки. Анализатор распознает возможность, но не в состоянии выполнить запрошенную обработку. Второе исключение чаще всего возникает при использовании фирменных свойств и возможностей, присущих лишь одной платформе (о них речь пойдет в следующем разделе), при последующей смене анализатора. Одна реализация анализатора ничего

не знает о фирменных свойствах и возможностях другой реализации, и это приводит к возникновению исключения SAXNotRecognizedException.

Эти исключения всегда следует перехватывать явным образом с целью обработки. В противном случае теряется ценная информация о событиях, произошедших в процессе выполнения кода. Ниже приводится измененная версия кода из предыдущей главы. Поскольку код содержит вызовы, связанные с установкой возможностей, архитектура обработки исключений должна измениться соответственно:

```
public void buildTree(DefaultTreeModel treeModel,
                      DefaultMutableTreeNode base, String xmlURI)
    throws IOException, SAXException {

    String featureURI = "";

    try {
        // Создаем экземпляры, необходимые для выполнения анализа
        XMLReader reader =
            XMLReaderFactory.createXMLReader(vendorParserClass);
        ContentHandler jTreeContentHandler =
            new JTreeContentHandler(treeModel, base, reader);
        ErrorHandler jTreeErrorHandler = new JTreeErrorHandler();

        // Регистрируем обработчик содержимого
        reader.setContentHandler(jTreeContentHandler);

        // Регистрируем обработчик ошибок
        reader.setErrorHandler(jTreeErrorHandler);

        /** Работаем с возможностями **/
        featureURI = "http://xml.org/sax/features/validation";

        // Запрашиваем проверку действительности
        reader.setFeature(featureURI, true);

        // Включаем обработку пространств имен
        featureURI = "http://xml.org/sax/features/namespaces";
        setNamespaceProcessing(reader, true);

        // Включаем канонизацию строк
        featureURI = "http://xml.org/sax/features/string-interning";
        reader.setFeature(featureURI, true);

        // Отключаем обработку схем
        featureURI =
            "http://apache.org/xml/features/validation/schema";
        reader.setFeature(featureURI, false);

        // Анализ документа
        InputSource inputSource =
            new InputSource(xmlURI);
```

```
            reader.parse(inputSource);
        } catch (SAXNotRecognizedException e) {
            System.out.println("Класс анализатора " + vendorParserClass +
                " не опознает возможность с указанным URI: " + featureURI);
            System.exit(0);
        } catch (SAXNotSupportedException e) {
            System.out.println("Класс анализатора " + vendorParserClass +
                " не поддерживает возможность с указанным URI: " +
                  featureURI);
            System.exit(0);
        }
    }
```

Обрабатывая эти исключения, а также другие особые случаи, вы предоставляете пользователю более полную информацию и повышаете качество своего кода.

Канонизация и сущности

Оставшиеся три возможности SAX довольно запутанны. Первая, *http://xml.org/sax/features/string-interning*, включает и отключает канонизацию строк. По умолчанию в большинстве анализаторов эта возможность имеет значение `false` (отключена). Если установить ее в значение `true`, то для всех имен элементов, имен атрибутов, URI и префиксов пространств имен, а также для всех остальных строк будет вызываться метод `java.lang.String.intern()`. Полагаю, здесь не стоит углубляться в подробности канонизации строк. Если вы не знаете, что это такое, обратитесь к документации Javadoc от Sun, доступной по адресу *http://java.sun.com/j2se/1.3/docs/api*. Если коротко, то каждый раз, когда встречается строка, Java пытается по возможности вернуть ссылку на строку, уже существующую во внутреннем пуле строк, вместо того чтобы создавать новый объект `String`. Звучит неплохо, правда? Что ж, причина, по которой эта возможность по умолчанию отключена, в том, что в большинстве анализаторов реализована оптимизация, которая может быть более эффективной, чем канонизация строк. Мой совет – оставить эту возможность в покое; разработчики потратили много недель, настраивая подобные вещи, так что не стоит с ними самостоятельно возиться.

Две другие возможности определяют, будут ли интерпретироваться текстовые сущности (*http://xml.org/sax/features/external-general-entities*), и будут ли при анализе рассматриваться параметрические сущности (*http://xml.org/sax/features/external-parameter-entities*). Эти возможности включены в большинство анализаторов, охватывая все типы сущностей, определенные в XML. Опять же, рекомендую оставить эти настройки нетронутыми, если только у вас нет особой причины запретить обработку сущностей.

Узлы DOM и строковые константы

Способ применения этих стандартных свойств SAX несколько менее очевиден. В обоих случаях эти свойства удобны в основном для получения значений, тогда как возможности чаще применяются для установки значений. Кроме того, оба свойства более полезны при обработке ошибок, чем в штатных ситуациях. И наконец, оба свойства обеспечивают доступ сегменту данных, который анализируется в конкретный момент времени. Первое свойство, определяемое URI *http://xml.org/sax/properties/dom-node*, возвращает узел DOM, который обрабатывается в данный момент, либо корневой узел DOM, если анализ еще не начался. Мы еще не затрагивали DOM, но все прояснится в следующих двух главах. Второе свойство, определяемое URI *http://xml.org/sax/properties/xml-string*, возвращает строку символов, которая обрабатывается в данный момент. В различных анализаторах эти свойства поддерживаются по-разному, из чего можно сделать вывод, что разработчики анализаторов также считают, что полезность этих свойств сомнительна. Например, анализатор Xerces не поддерживает свойство `xml-string`, чтобы избежать буферизации исходного документа (по крайней мере, этим конкретным способом). С другой стороны, он поддерживает свойство `dom-node`, так что можно, по сути дела, превратить анализатор SAX в итератор дерева DOM.

Фирменные свойства и возможности

Помимо стандартных свойств и возможностей, определяемых в SAX, большинство анализаторов определяют несколько фирменных свойств и возможностей. Например, на странице *http://xml.apache.org/xerces-j/features.html* перечислены все возможности, поддерживаемые анализатором Apache Xerces, а на странице *http://xml.apache.org/xerces-j/properties.html* – все поддерживаемые свойства. Я не стану рассматривать их подробно, а вы старайтесь избегать их применения, поскольку в противном случае создаваемый код оказывается привязанным к конкретной платформе. Разумеется, существуют ситуации, когда использование специальных возможностей платформы позволяет сократить объем работы. В таких случаях проявляйте осторожность, но не делайте глупостей – воспользуйтесь тем, что предоставляет анализатор!

В качестве примера рассмотрим возможность Xerces, разрешающую и запрещающую обработку схемы XML: *http://apache.org/xml/features/validation/schema*. Поскольку поддержка схем XML в анализаторах и в SAX не стандартизирована, используйте эту специфическую возможность (по умолчанию она включена), чтобы сэкономить время на анализе схем XML, на которые ссылается документ. Мы экономим время, если обработка не производится, а для ее отключения требуется возможность, специфичная для платформы. Чтобы выяснить, какие возможности доступны помимо стандартных возможностей SAX, изучите документацию, предоставляемую разработчиком.

И опять обработчики

В предыдущей главе мы изучили интерфейсы ContentHandler и ErrorHandler, а также говорили о существовании интерфейсов EntityResolver и DTDHandler. Теперь, когда вы хорошо понимаете основы SAX, вы готовы к рассмотрению этих обработчиков.[1] Вы обнаружите, что обработчику EntityResolver периодически находится применение (чаще в тех случаях, когда речь идет о коммерческих разработках), а обработчик DTDHandler практически не используется.

Использование EntityResolver

Первый из этих новых обработчиков – org.xml.sax.EntityResolver. Этот интерфейс делает именно то, о чем говорит его название: он интерпретирует сущности (либо, по меньшей мере, объявляет метод, интерпретирующий сущности; думаю, вы улавливаете мысль). Интерфейс определяет единственный метод, который выглядит так:

```
public InputSource resolveEntity(String publicID, String systemID)
    throws SAXException, IOException;
```

Можно создать реализацию этого интерфейса и зарегистрировать ее в экземпляре XMLReader (с помощью метода setEntityResolver(), как можно догадаться). Впоследствии, встретив в документе сущность, анализатор передает ее публичный[2] и системный идентификаторы методу resolveEntity(), реализованному разработчиком. Теперь можно изменить традиционный процесс разрешения сущностей.

Обычно сущность по указанному публичному или системному идентификатору, будь то имя файла, URL или иной указатель ресурса, интерпретируется анализатором XML. И если метод resolveEntity() возвращает значение null, этот процесс протекает в традиционном варианте. В результате следует заботиться о том, чтобы метод resolveEntity() всегда возвращал null по умолчанию, вне зависимости от его реализации. Иначе говоря, в качестве основы можно взять реализацию, представленную в примере 4.1.

Пример 4.1. Простая реализация EntityResolver

```
package javaxml2;

import java.io.IOException;

import org.xml.sax.EntityResolver;
import org.xml.sax.InputSource;
```

[1] Да, я знаю, что формально EntityResolver не является «обработчиком». Я мог бы легко доказать, что интерфейс должен называться EntityHandler, так что для меня выбранная терминология вполне точна.

[2] Если таковой существует. – *Примеч. науч. ред.*

```
import org.xml.sax.SAXException;

public class SimpleEntityResolver implements EntityResolver {

    public InputSource resolveEntity(String publicID, String systemID)
        throws IOException, SAXException {

        // По умолчанию возвращаем null
        return null;
    }
}
```

Можно без проблем скомпилировать этот класс и зарегистрировать его в реализации XMLReader, используемой в классе SAXTreeViewer в методе buildTree():

```
        // Создаем экземпляры, необходимые для анализа
        XMLReader reader =
            XMLReaderFactory.createXMLReader(vendorParserClass);
        ContentHandler jTreeContentHandler =
            new JTreeContentHandler(treeModel, base, reader);
        ErrorHandler jTreeErrorHandler = new JTreeErrorHandler();

        // Регистрируем обработчик содержимого
        reader.setContentHandler(jTreeContentHandler);

        // Регистрируем обработчик ошибок
        reader.setErrorHandler(jTreeErrorHandler);

        // Регистрируем интерпретатор сущностей
        reader.setEntityResolver(new SimpleEntityResolver());

        // Прочие инструкции и анализ...
```

Повторная компиляция и запуск примера ничего не изменят. Конечно, именно это и прогнозировалось, так что не очень удивляйтесь. По той причине, что все время возвращается значение null, процесс интерпретации сущностей протекает как обычно. Если вы считаете, что ничего не происходит, можно внести небольшое изменение и выводить на экран информацию о событиях:

```
    public InputSource resolveEntity(String publicID, String systemID)
        throws IOException, SAXException {

        System.out.println("Найдена сущность с публичным идентификатором " +
        publicID +
            " и системным идентификатором " + systemID);

        // По умолчанию возвращаем null
        return null;
    }
```

Снова скомпилируйте этот класс и запустите программу визуализации дерева. Подвиньте появившееся окно Swing и взгляните на вывод в окне терминала. Он должен выглядеть примерно так, как показано в примере 4.2.

Пример 4.2. Подробный вывод SAXTreeViewer

```
C:\javaxml2\build>java javaxml2.SAXTreeViewer
    c:\javaxml2\ch04\xml\contents.xml
Найдена сущность с публичным идентификатором null и системным идентификатором
  file:///c:/javaxml2/ch04/xml/DTD/JavaXML.dtd
Найдена сущность с публичным идентификатором null и системным идентификатором
  http://www.newInstance.com/javaxml2/copyright.xml
```

Как всегда, разрывы строк приведены только для наглядности. В любом случае мы видим, что обе ссылки в XML-документе – на DTD и на сущность `OReillyCopyright` – переданы методу `resolveEntity()`.

Сейчас вы, возможно, озадачены: разве DTD – сущность? Термин «сущность» в контексте обработчика `EntityResolver` несколько расплывчат. Вероятно, лучшим именем для интерфейса было бы `ExternalReferenceResolver`,[1] но его не очень-то удобно набирать. В любом случае запомните, что любая внешняя ссылка в вашем XML-документе будет передаваться этому методу. Так в чем же смысл происходящего? Помните ссылку для `OReillyCopyright` и ее связь с Интернет-адресом (*http://www.newInstance.com/javaxml2/copyright.xml*)? Что если у вас нет доступа в Интернет? Что если уже существует локальная копия, и вы хотите сэкономить время, использовав ее? Что если вы просто хотите включить в документ свою собственную информацию о правообладании? Это вполне резонные вопросы, и речь идет о реальных задачах, которые могут возникать при разработке приложений. Ответом, конечно же, является метод `resolveEntity()`, о котором я говорил.

Если вернуть из метода корректный объект `InputSource` (вместо значения `null`), то вместо указанного публичного или системного идентификатора в качестве значения ссылки на сущности будет использоваться этот объект. Другими словами, можно указать на произвольный ресурс, предупредив таким образом самостоятельную обработку сущностей анализатором. В качестве примера создайте на локальной машине файл *copyright.xml*, как показано в примере 4.3.

Пример 4.3. Локальная копия файла copyright.xml

```
<copyright xmlns="http://www.oreilly.com">
  <year value="2001" />
  <content>Локальная версия информации о правообладании.</content>
</copyright>
```

[1] Интерпретатор внешних ссылок. – *Примеч. науч. ред.*

Сохраните этот файл в каталоге, который доступен вам из кода на Java (я использую тот же каталог, в котором хранится файл *contents.xml*), и внесите следующие изменения в метод `resolveEntity()`:

```
public InputSource resolveEntity(String publicID, String systemID)
    throws IOException, SAXException {

    // Обрабатываем ссылки на online-версию файла copyright.xml
    if (systemID.equals(
        "http://www.newInstance.com/javaxml2/copyright.xml")) {
        return new InputSource(
            "file:///c:/javaxml2/ch04/xml/copyright.xml");
    }

    // По умолчанию возвращаем null
    return null;
}
```

Как видите, интерпретация ссылки на ресурс, доступный в сети, заменяется созданием объекта `InputSource`, который обеспечивает доступ к локальной версии файла *copyright.xml*. Если повторно скомпилировать исходный файл и запустить программу визуализации, можно убедиться, что используется именно эта локальная версия. На рис. 4.1 показан раскрытый элемент `ora:copyright`, включающий содержимое локального документа с информацией о правообладании.

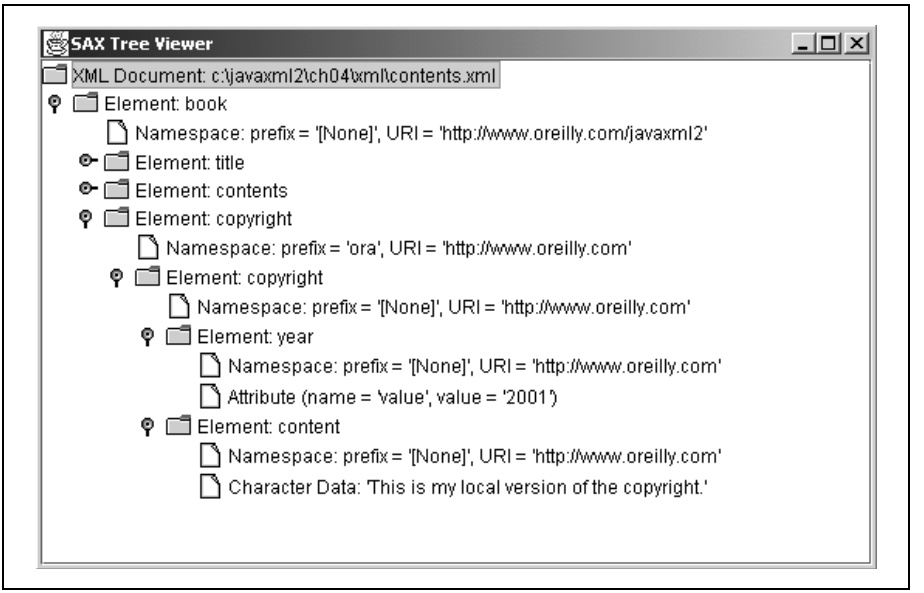

Рис. 4.1. Программа SAXTreeViewer, запущенная с локальным файлом copyright.xml

В реальных приложениях этот метод стремится стать длинной последовательностью блоков `if/then/else`, каждый из которых обрабатывает определенный системный или публичный идентификатор. Тут есть один важный момент: старайтесь не делать из этого класса и метода мусорное ведро для идентификаторов. Если определенный способ интерпретации ссылки больше не нужен, удалите соответствующий ему оператор `if`. Кроме того, для различных приложений старайтесь использовать различные реализации `EntityResolver`, а не одну общую реализацию для всех приложений. Благодаря этому код не разрастается и, что более важно, увеличивается скорость интерпретации сущностей. Если каждый раз анализатор выполняет пятьдесят или сто сравнений `String.equals()`, это очень сильно замедляет приложение. Обязательно помещайте в начале последовательности блоков `if/else` те ссылки, которые чаще используются, поскольку это ускоряет работу метода и интерпретацию сущностей в целом.

Наконец, еще одна рекомендация, касающаяся реализаций `EntityResolver`. Вы заметите, что я поместил свою реализацию в отдельный файл класса, тогда как реализации обработчиков `ErrorHandler`, `ContentHandler` и (как мы увидим в следующем разделе) `DTDHandler` содержатся в том же исходном файле, в котором происходит анализ. Это не случайно! Вы увидите, что способы работы с содержимым, ошибками и DTD довольно статичны. Мы пишем программу, вот и все. Когда мы вносим исправления, то переписываем многое, так что в любом случае изменения бывают заметными. То же относится и к способу, которым приложение интерпретирует сущности. В зависимости от компьютера, на котором мы работаем, от типа клиента, использующего приложение, и от того, какие документы доступны и где, нам понадобятся различные версии реализации `EntityResolver`. Чтобы предусмотреть возможность частых изменений реализации без необходимости редактировать или перекомпилировать основной код, выполняющий анализ, я советую отвести отдельный исходный файл для реализаций `EntityResolver`. Теперь вы знаете все, что следует знать об интерпретации сущностей в приложениях, использующих SAX.

Использование DTDHandler

После довольно пространного описания `EntityResolver` мы очень коротко остановимся на интерфейсе `DTDHandler`. За два года интенсивного программирования XML я обращался к этому интерфейсу лишь один раз – при написании JDOM, и то это был довольно запутанный случай. Если в ваших XML-документах не очень много неанализируемых сущностей, то вы, скорее всего, не будете работать с этим интерфейсом.

Интерфейс `DTDHandler` позволяет получать уведомление, когда анализатор встречает неанализируемую сущность или объявление нотации. Конечно же, оба этих события происходят в DTD, а не в XML-документах, вот потому интерфейс и называется `DTDHandler`. Имеет смысл

без лишних слов взглянуть, как выглядит этот интерфейс. Он приведен в примере 4.4.

Пример 4.4. Интерфейс DTDHandler

```
package org.xml.sax;

public interface DTDHandler {

    public void notationDecl(String name, String publicID,
                             String systemID)
        throws SAXException;

    public void unparsedEntityDecl(String name, String publicId,
                                   String systemId, String notationName)
         throws SAXException;
}
```

Никаких сюрпризов здесь нет. Первый метод получает сообщения, связанные с объявлениями нотаций; сообщение содержит имя нотации, публичный и системный идентификаторы. Помните структуру `NOTATION` в DTD?

```
<!NOTATION jpeg SYSTEM "images/jpeg">
```

Второй метод получает информацию, связанную с объявлением неанализируемой сущности, которое выглядит следующим образом:

```
<!ENTITY stars_logo SYSTEM "http://www.nhl.com/img/team/dal38.gif"
                    NDATA jpeg>
```

Если создать реализацию `DTDHandler` и зарегистрировать ее в анализаторе с помощью метода `setDTDHandler()`, то в обоих случаях можно предпринимать произвольные действия по наступлении этих событий. Это особенно полезно при написании низкоуровневых приложений, которые должны воссоздавать XML-данные (например, редактор XML), либо когда необходимо создать Java-представление ограничений, накладываемых DTD (скажем, с целью связывания данных, которое мы рассмотрим в главе 15). В большинстве других ситуаций этот интерфейс будет нужен вам не часто.

Класс DefaultHandler

Прежде чем закончить с обработчиками (по крайней мере, сейчас), необходимо рассказать об одном классе, также имеющем к ним отношение. Это класс `org.xml.sax.helpers.DefaultHandler`, который может очень здорово помочь разработчикам, использующим SAX. Вспомните, что до сих пор реализация различных интерфейсов-обработчиков требовала реализации разных обработчиков `ContentHandler` — одного для реализации `ErrorHandler`, одного для `EntityResolver` (это нормально по тем причинам, из-за которых лучше хранить реализацию этого ин-

терфейса в отдельном исходном файле) и одного для реализации DTDHandler, если он нужен. Кроме того, вы познаете «радость» реализации многочисленных методов ContentHandler, даже если они в действительности ничего не делают.

Тут на помощь приходит DefaultHandler. Сам по себе этот класс не определяет никакого поведения, он реализует интерфейсы ContentHandler, ErrorHandler, EntityResolver и DTDHandler и предоставляет пустые реализации для каждого метода каждого интерфейса. Поэтому можно иметь единственный класс (назовем его, например, MyHandlerClass), расширяющий DefaultHandler. Этот класс должен переопредлять только те методы, в которых должны выполняться какие-либо действия. Например, можно реализовать методы startElement(), characters(), endElement() и fatalError(). В любом случае жизнь становится проще: не приходится писать множество строк кода для методов, которые ничего не делают; кроме того, код становится более прозрачным. К тому же, в качестве аргументов методов setErrorHandler(), setContentHandler() и setDTDHandler() будет выступать один и тот же экземпляр класса MyHandlerClass. Теоретически можно было бы передать этот экземпляр и методу setEntityResolver(), хотя (говорю уже, наверное, в четвертый раз!) я не советую смешивать метод resolveEntity() с методами из других интерфейсов.

Фильтры и объекты Writer

Пора сойти с проторенного пути. До сих пор мы подробно рассматривали все, что находится в «стандартном» приложении SAX, начиная от анализатора и заканчивая обратными вызовами методов обработчиков. Однако в SAX существует масса дополнительных возможностей, которые могут сделать разработчика настоящим волшебником и вывести его за границы «стандартного» SAX. В этом разделе представлены две такие возможности: фильтры SAX и объекты writer. Используя классы из стандартного дистрибутива SAX, а также классы, отдельно доступные на веб-сайте SAX (*http://www.saxproject.org*), можно сделать приложения SAX более функциональными. Вы научитесь использовать SAX в качестве конвейера событий, а не одного уровня обработки. Мы рассмотрим эту концепцию более подробно, а пока достаточно сказать, что это действительно ключ к написанию эффективного и модульного SAX-кода.

Фильтры XMLFilter

Первым в списке стоит класс, входящий в базовую версию SAX, доступную для загрузки с сайта проекта SAX (*http://www.saxproject.org*), он должен входить в состав любого дистрибутива анализатора, поддерживающего SAX 2.0. Речь идет о классе org.xml.sax.XMLFilter. Этот

класс расширяет интерфейс XMLReader и добавляет к нему два новых метода :

```
public void setParent(XMLReader parent);

public XMLReader getParent();
```

На первый взгляд, обсуждать здесь особенно нечего. С помощью иерархии объектов XMLReader данный мезханизм фильтрации позволяет создать цепь обработки, или *конвейер* событий. Чтобы понять, что я понимаю под конвейером, покажу, как выглядит нормальный ход процесса анализа SAX:

- События из XML-документа передаются анализатору SAX.
- Анализатор SAX и зарегистрированные обработчики передают события и данные приложению.

Но разработчики пришли к мысли, что очень просто добавить в эту цепь одно или несколько дополнительных звеньев:

- События из XML-документа передаются анализатору SAX.
- Анализатор SAX выполняет некоторую обработку и передает информацию другому анализатору SAX.
- Процедура повторяется до тех пор, пока не будет выполнена вся обработка SAX.
- Наконец, анализатор SAX и зарегистрированные обработчики передают события и данные приложению.

Конвейер представлен промежуточными шагами: один анализатор, выполняющий определенный вид работы, несколько раз передает информацию другому анализатору. Это позволяет распределить код между анализаторами XMLReader, а не втискивать весь код в один класс. Конвейер из нескольких анализаторов позволяет создавать модульные, эффективные программы. Именно для этого и предназначен класс XMLFilter: для создания цепей реализаций XMLReader посредством фильтрации. Расширение более высокого уровня — класс org.xml.sax.helpers.XMLFilterImpl, обеспечивающий вспомогательную реализацию класса XMLFilter. Это объединение класса XMLFilter и класса DefaultHandler, о котором говорилось в предыдущем разделе. Класс XMLFilterImpl реализует XMLFilter, ContentHandler, ErrorHandler, EntityResolver и DTDHandler, предоставляя разделяемые методы каждого из обработчиков. Другими словами, он позволяет создать конвейер для всех событий SAX, при этом переопределив любые методы, которые необходимо переопределить для добавления в конвейер поведения, определяемого разработчиком приложения.

Воспользуемся одним из этих фильтров. В примере 4.5 приведен работающий, готовый к использованию фильтр. Вы уже изучили основы, поэтому мы рассмотрим его в темпе.

Пример 4.5. Класс NamespaceFilter

```
package javaxml2;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLFilterImpl;

public class NamespaceFilter extends XMLFilterImpl {

    /** Старый URI для замены */
    private String oldURI;

    /** Новый URI, заменяющий старый */
    private String newURI;

    public NamespaceFilter(XMLReader reader,
                           String oldURI, String newURI) {
        super(reader);
        this.oldURI = oldURI;
        this.newURI = newURI;
    }

    public void startPrefixMapping(String prefix, String uri)
        throws SAXException {

        // При необходимости изменяем URI
        if (uri.equals(oldURI)) {
            super.startPrefixMapping(prefix, newURI);
        } else {
            super.startPrefixMapping(prefix, uri);
        }
    }

    public void startElement(String uri, String localName,
                             String qName, Attributes attributes)
        throws SAXException {

        // При необходимости изменяем URI
        if (uri.equals(oldURI)) {
            super.startElement(newURI, localName, qName, attributes);
        } else {
            super.startElement(uri, localName, qName, attributes);
        }
    }

    public void endElement(String uri, String localName, String qName)
        throws SAXException {

        // При необходимости изменяем URI
        if (uri.equals(oldURI)) {
            super.endElement(newURI, localName, qName);
```

```
            } else {
                super.endElement(uri, localName, qName);
            }
        }
    }
```

Наш класс расширяет XMLFilterImpl, поэтому можно не беспокоиться о событиях, которые не требуют специальных действий, – о них позаботится класс XMLFilterImpl, передавая в неизменном виде события, для которых не были переопределены методы. Соответственно, мы можем сразу уделить внимание тому, что именно должен делать фильтр. В данном случае он изменяет URI пространств имен. Задача может показаться тривиальной, но не стоит недооценивать полезность такой реализации. Много раз за последние несколько лет менялся URI пространства имен для спецификации (например, XML Schema или XSLT). Вместо того чтобы вручную редактировать все свои XML-документы или писать специальный код для чужих XML-документов, я решаю проблему с помощью класса NamespaceFilter.

Передавая экземпляр XMLReader конструктору, мы делаем анализатор родительским объектом для экземпляра нашего класса. Соответственно, родительский анализатор получает все события, переданные фильтром (т. е. все события, в силу наличия класса XMLFilterImpl, если только класс NamespaceFilter не переопределяет это поведение). Фильтр инициализируется парой URI – первоначальным и URI, его заменяющим. Три переопределенных метода реализуют необходимые операции над URI. После того как подобный фильтр создан, ему передается анализатор, и впоследствии все операции производятся над объектом *filter*, а не *reader*. Возвращаясь назад к файлу *contents.xml* и программе визуализации SAXTreeViewer, предположим, что мне сообщили из O'Reilly, что URL моей книги изменился с *http://www.oreilly.com/javaxml2* на *http://www.oreilly.com/catalog/javaxml2*. Вместо того чтобы редактировать все мои примеры и обновлять сайт, я могу просто воспользоваться классом NamespaceFilter:

```
    public void buildTree(DefaultTreeModel treeModel,
                          DefaultMutableTreeNode base, String xmlURI)
        throws IOException, SAXException {

        // Создаем экземпляры, необходимые для анализа
        XMLReader reader =
            XMLReaderFactory.createXMLReader(vendorParserClass);
        NamespaceFilter filter =
            new NamespaceFilter(reader,
                "http://www.oreilly.com/javaxml2",
                "http://www.oreilly.com/catalog/javaxml2");
        ContentHandler jTreeContentHandler =
            new JTreeContentHandler(treeModel, base, reader);
        ErrorHandler jTreeErrorHandler = new JTreeErrorHandler();
```

```
        // Регистрируем обработчик содержимого
        filter.setContentHandler(jTreeContentHandler);

        // Регистрируем обработчик ошибок
        filter.setErrorHandler(jTreeErrorHandler);

        // Регистрируем интерпретатор сущностей
        filter.setEntityResolver(new SimpleEntityResolver());

        // Анализируем
        InputSource inputSource =
            new InputSource(xmlURI);
        filter.parse(inputSource);
    }
```

Повторюсь, все операции выполняются для фильтра, а не экземпляра анализатора. Создав такой фильтр, можно скомпилировать оба исходных файла (*NamespaceFilter.java* и *SAXTreeViewer.java*) и запустить программу визуализации для файла *contents.xml*. Вы увидите, что URI пространства имен O'Reilly для моей книги изменился в каждом случае (рис. 4.2).

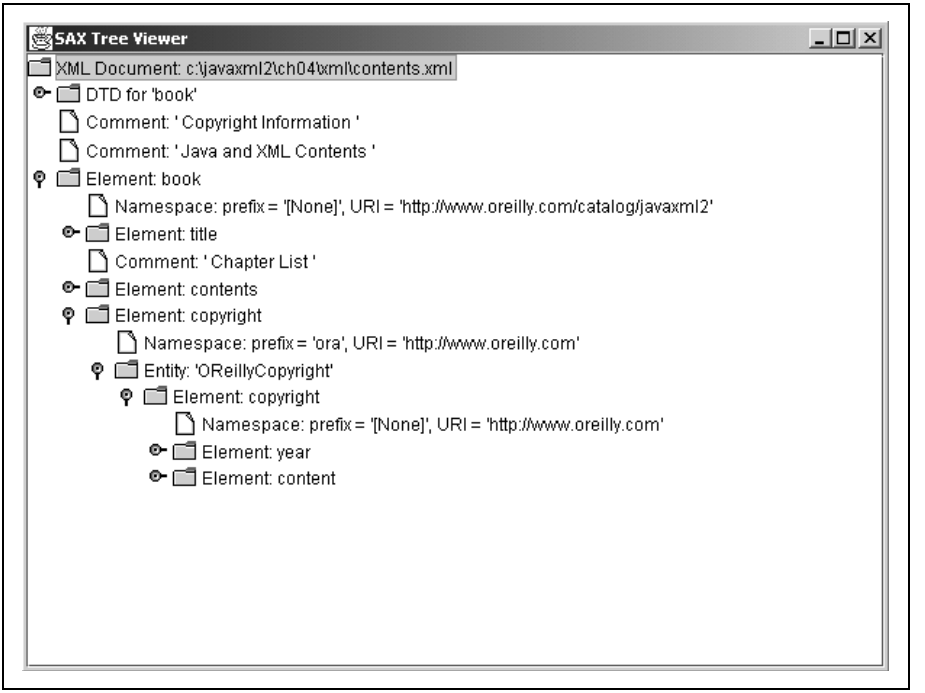

Рис. 4.2. SAXTreeViewer для файла contents.xml с использованием NamespaceFilter

Конечно же, можно объединить эти фильтры в цепь и использовать их в качестве стандартных библиотек. Работая с более старыми XML-до-

кументами, я часто создаю и использую несколько фильтров со старыми URI для XSL и схемы XML, поэтому мне не приходится волноваться о неверных URI:

```
XMLReader reader =
        XMLReaderFactory.createXMLReader(vendorParserClass);
    NamespaceFilter xslFilter =
        new NamespaceFilter(reader,
            "http://www.w3.org/TR/XSL",
            "http://www.w3.org/1999/XSL/Transform");
    NamespaceFilter xsdFilter =
        new NamespaceFilter(xslFilter,
            "http://www.w3.org/TR/XMLSchema",
            "http://www.w3.org/2001/XMLSchema");
```

В этом случае я создаю более длинный конвейер, чтобы гарантировать, что не осталось ни одного старого URI для пространств имен, который мог бы привести к проблемам в приложении. Но будьте осторожны и не создавайте слишком длинные конвейеры. Каждое новое звено в цепи несколько замедляет обработку. Тем не менее, это отличный способ создавать повторно используемые компоненты для SAX.

XMLWriter

Теперь, когда вы понимаете принципы работы фильтров в SAX, я хочу показать вам особый фильтр – `XMLWriter`. Этот класс, а также и его подкласс `DataWriter` можно загрузить с веб-сайта проекта SAX, *http://www.saxproject.org*. Класс `XMLWriter` расширяет `XMLFilterImpl`, а `DataWriter` расширяет `XMLWriter`. Оба класса применяются для вывода данных XML, что несколько выбивается из общего ряда вопросов SAX, которые мы рассмотрели к этому моменту. Но раз мы могли добавлять в обратные вызовы SAX операторы, выводящие данные в объекты записи, то и данный класс может делать то же самое. Не станем уделять этому классу много времени, поскольку это не тот способ, которым в общем случае удобно выводить XML. Для решения этой задачи, если вам нужна изменчивость, лучше пользоваться DOM, JDOM или иным XML API. При этом класс `XMLWriter` предоставляет удобный способ выяснить, что происходит на конвейере SAX. Будучи добавленным в конвейер между другими фильтрами и анализаторами, он может применяться для получения «моментальной фотографии» данных в точке его присутствия в цепи обработки. Например, при изменении URI пространства имен может возникнть необходимость сохранения XML-документа с новым URI пространства имен (будь то измененный URI O'Reilly, обновленный URI для XSL или для схем XML) для последующего использования. В случае применения класса `XMLWriter` это становится пустяковой задачей. Проскольку у нас уже есть `SAXTreeViewer`, использующий реализацию `NamespaceFilter`, рассмотрим его в качестве примера. Во-первых, добавьте операторы импортирования для клас-

сов java.io.FileWriter (для вывода) и com.megginson.sax.XMLWriter. Во-вторых, необходимо добавить экземпляр класса XMLWriter между экземплярами NamespaceFilter и XMLReader. Это означает, что данные будут выведены после замены пространства имен, но до возникновения видимых событий. Внесем в наш код следующие изменения:

```
public void buildTree(DefaultTreeModel treeModel,
                      DefaultMutableTreeNode base, String xmlURI)
    throws IOException, SAXException {

    // Создаем экземпляры, необходимые для анализа
    XMLReader reader =
        XMLReaderFactory.createXMLReader(vendorParserClass);
    XMLWriter writer =
       new XMLWriter(reader, new FileWriter("snapshot.xml"));
    NamespaceFilter filter =
        new NamespaceFilter(writer,
            "http://www.oreilly.com/javaxml2",
            "http://www.oreilly.com/catalog/javaxml2");
    ContentHandler jTreeContentHandler =
        new JTreeContentHandler(treeModel, base, reader);
    ErrorHandler jTreeErrorHandler = new JTreeErrorHandler();

    // Регистрируем обработчик содержимого
    filter.setContentHandler(jTreeContentHandler);

    // Регистрируем обработчик ошибок
    filter.setErrorHandler(jTreeErrorHandler);

    // Регистрируем интерпретатор сущностей
    filter.setEntityResolver(new SimpleEntityResolver());

    // Анализируем
    InputSource inputSource =
        new InputSource(xmlURI);
    filter.parse(inputSource);
}
```

Убедитесь, что родительским классом экземпляра NamespaceFilter является XMLWriter, а не XMLReader. В противном случае ничего не будет выведено. Скомпилировав эти изменения, запустите пример. В результате в рабочем каталоге будет создан файл snapshot.xml, фрагмент которого приведен далее:

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

<book xmlns="http://www.oreilly.com/catalog/javaxml2">
  <title ora:series="Java"
         xmlns:ora="http://www.oreilly.com">Java и XML</title>
```

```
  <contents>
    <chapter title="Введение" number="1">
      <topic name="XML имеет значение"></topic>
      <topic name="Что важно?"></topic>
      <topic name="Основы"></topic>
      <topic name="Что дальше?"></topic>
    </chapter>
    <chapter title="Основы технологии" number="2">
      <topic name="Основы"></topic>
      <topic name="Ограничения"></topic>
      <topic name="Преобразования"></topic>
      <topic name="И далее..."></topic>
      <topic name="Что дальше?"></topic>
    </chapter>
    <!-- Остальное содержимое... -->

  </contents>
</book>
```

Обратите внимание, что пространство имен, изменяемое фильтром `NamespaceFilter`, здесь модифицировано. Подобные «снимки», создаваемые экземплярами `XMLWriter`, могут быть отличным подспорьем для отладки и ведения журнала событий SAX.

Классы `XMLWriter` и `DataWriter` предоставляют гораздо больше методов для вывода XML как в целом, так и по частям, поэтому следует обращаться к документации Javadoc, включаемой в состав используемого пакета. Я не поощряю применение этих классов для рутинного создания данных XML. По моему опыту, они наиболее полезны в случаях, подобных рассмотренному выше.

И снова обработчики

Теперь рассмотрим еще два класса, которые предлагает нам SAX. Оба этих интерфейса не принадлежат основному дистрибутиву SAX и находятся в пакете `org.xml.sax.ext`, т. е. являются расширениями SAX. Однако большинство анализаторов (например, Apache Xerces) содержат оба этих класса. Просмотрите документацию от разработчика анализатора и если этих классов у вас нет, можете загрузить с веб-сайта SAX. Предупреждаю, что не все драйверы SAX поддерживают эти расширения, поэтому, если они не включены в состав анализатора производителем, следует поинтересоваться, по какой причине, а также выяснить, планируется ли поддержка расширений SAX в новых версиях системы от этого разработчика.

Обработчик LexicalHandler

Первый из этих двух обработчиков — `org.xml.sax.ext.LexicalHandler` — наиболее полезен. Этот обработчик предоставляет методы, которые

могут получать уведомления о возникновении определенных лексических событий, таких как комментарии, объявления сущностей, объявления DTD и секции CDATA. В обработчике ContentHandler эти лексические события попросту игнорируются, и приложение получает только данные и объявления без уведомления о том, когда или каким образом они были получены.

Как правило, данный обработчик не применяется, т. к. большинству приложений безразлично, где находился текст – в секции CDATA или нет. Однако если речь идет о редакторе XML, о сохранении данных в виде потока либо о произвольном компоненте, который работает на основе точного формата исходного документа, а не только его содержимого, на помощь приходит обработчик LexicalHandler. Чтобы увидеть его в действии, следует импортировать org.xml.sax.ext.LexicalHandler в исходном файле *SAXTreeViewer.java*. После этого можно добавить *LexicalHandler* в конструкцию implements в закрытом классе JTreeContentHandler этого исходного файла:

```
class JTreeContentHandler implements ContentHandler, LexicalHandler {
    // Реализации методов обратного вызова
}
```

Повторно используя обработчик содержимого, уже реализованный в этом классе, обратные вызовы лексического обработчика могут взаимодействовать с JTree с целью визуализации лексических событий. Осталось реализовать все методы, определенные в LexicalHandler. Вот они:

```
public void startDTD(String name, String publicID, String systemID)
        throws SAXException;
public void endDTD() throws SAXException;
public void startEntity(String name) throws SAXException;
public void endEntity(String name) throws SAXException;
public void startCDATA() throws SAXException;
public void endCDATA() throws SAXException;
public void comment(char[] ch, int start, int length)
        throws SAXException;
```

Для начала рассмотрим первое лексическое событие, которое может произойти при обработке XML-документа: начало и конец объявления или ссылки на DTD. Оно приводит к вызову методов startDTD() и endDTD(), показанных ниже:

```
public void startDTD(String name, String publicID,
                     String systemID)
    throws SAXException {

    DefaultMutableTreeNode dtdReference =
        new DefaultMutableTreeNode("DTD для '" + name + "'");
    if (publicID != null) {
        DefaultMutableTreeNode publicIDNode =
```

```
                new DefaultMutableTreeNode("Публичный идентификатор: '" +
                    publicID + "'");
            dtdReference.add(publicIDNode);
        }
        if (systemID != null) {
            DefaultMutableTreeNode systemIDNode =
                new DefaultMutableTreeNode("Системный ID: '" +
                    systemID + "'");
            dtdReference.add(systemIDNode);
        }
        current.add(dtdReference);
    }

    public void endDTD() throws SAXException {
        // Здесь никаких действий предпринимать не нужно
    }
```

В результате при нахождении DTD и системного и публичного идентификаторов, если они существуют, происходит визуализация этого события. Далее следует пара похожих методов для сущностей: `startEntity()` и `endEntity()`. Они вызываются до и после (соответственно) обработки ссылки на сущность. При помощи приведенного ниже кода можно реализовать визуализацию и для этого события:

```
    public void startEntity(String name) throws SAXException {
        DefaultMutableTreeNode entity =
            new DefaultMutableTreeNode("Сущность: '" + name + "'");
        current.add(entity);
        current = entity;
    }

    public void endEntity(String name) throws SAXException {
        // Отход вверх по дереву
        current = (DefaultMutableTreeNode)current.getParent();
    }
```

Код гарантирует, что данные, извлекаемые по ссылке на сущность, например `OreillyCopyright`, содержатся в узле дерева «Сущность». Довольно просто.

Следующее лексическое событие – секция `CDATA`, но таких секций нет в документе *contents.xml*. Необходимо изменить этот документ следующим образом (`CDATA` допускает наличие амперсанда в содержимом элемента `title`):

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">

<!-- «Java и XML», Оглавление -->
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
```

```
>
  <title ora:series="Java"><![CDATA[Java & XML]]></title>

  <!-- Прочие данные -->
</book>
```

Теперь можно добавить код для обратных вызовов CDATA. Добавьте к классу JTreeContentHandler следующие методы:

```
public void startCDATA() throws SAXException {
    DefaultMutableTreeNode cdata =
        new DefaultMutableTreeNode("Секция CDATA");
    current.add(cdata);
    current = cdata;
}

public void endCDATA() throws SAXException {
    // Отход вверх по дереву
    current = (DefaultMutableTreeNode)current.getParent();
}
```

Здесь нам уже все знакомо; содержимое элемента title теперь появляется в виде дочернего узла для узла «Секция CDATA». Так что нам осталось рассмотреть лишь один метод – тот, который получает уведомления о комментариях:

```
public void comment(char[] ch, int start, int length)
    throws SAXException {

    String comment = new String(ch, start, length);
    DefaultMutableTreeNode commentNode =
        new DefaultMutableTreeNode("Комментарий: '" + comment + "'");
    current.add(commentNode);
}
```

Этот метод ведет себя так же, как методы characters() и ignorableWhitespace(). Помните, что этому методу передается только текст комментария, без окружающих его разделителей <!-- и -->. После внесения всех этих изменений можно скомпилировать программу и запустить ее. Будет получен вывод, подобный показанному на рис. 4.3.

Вы заметите одну странность: сущность под названием [dtd]. Это событие происходит каждый раз, когда встречается объявление DOCTYPE, и от него можно избавиться (вероятно, вы не захотите видеть эту сущность в результатах) при помощи простой конструкции в методах startEntity() и endEntity():

```
public void startEntity(String name) throws SAXException {
     if (!name.equals("[dtd]")) {
        DefaultMutableTreeNode entity =
            new DefaultMutableTreeNode("Сущность: '" + name + "'");
        current.add(entity);
```

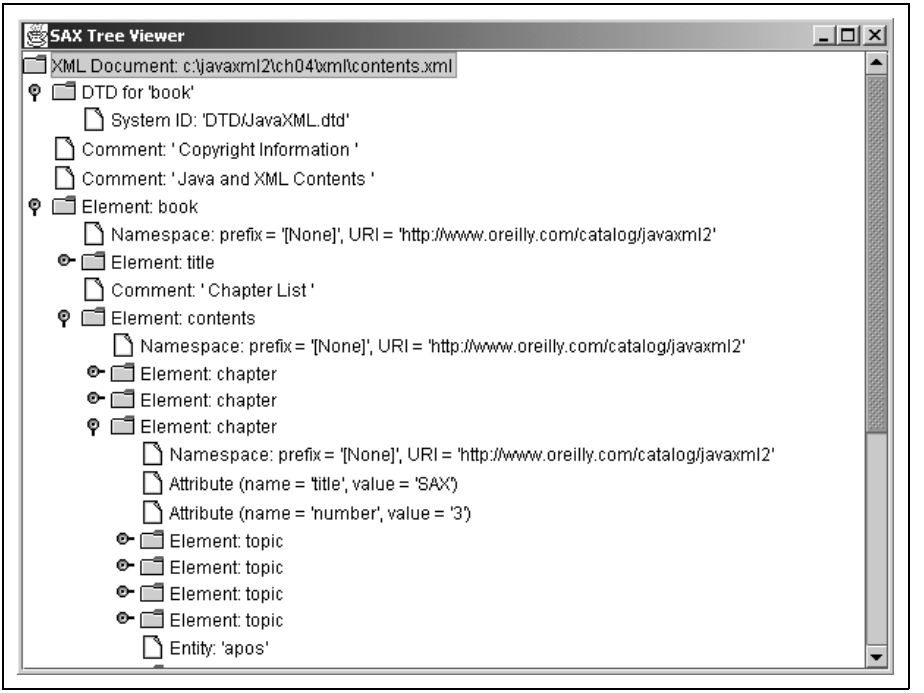

Рис. 4.3. Вывод при использовании реализации LexicalHandler

```
current = entity;
      }
  }

  public void endEntity(String name) throws SAXException {
       if (!name.equals("[dtd]")) {
          // Отход вверх по дереву
          current = (DefaultMutableTreeNode)current.getParent();
       }
  }
```

Данная конструкция удаляет сущность-раздражитель. Вот, пожалуй, и все, что можно тут сказать о `LexicalHandler`. И хотя он рассмотрен в разделе, посвященном более сложным возможностям SAX, работа с ним вполне прозрачна.

DeclHandler

Последний обработчик, с которым мы будем иметь дело, – это `DeclHandler`. Этот интерфейс определяет методы, получающие уведомления об особых событиях, происходящих в DTD, таких как объявления элементов и атрибутов. Про него также можно сказать, что он нужен только в очень специальных случаях: в тех же редакторах XML и ком-

понентах, которые должны знать точную лексическую структуру документа и его DTD. Я не собираюсь приводить пример использования обработчика DeclHandler; сейчас вы уже знаете об обратных вызовах обработчиков больше, чем вам когда-либо может понадобиться. Вместо этого я просто представлю интерфейс (пример 4.6).

Пример 4.6. Интерфейс DeclHandler

```
package org.xml.sax.ext;

import org.xml.sax.SAXException;

public interface DeclHandler {

    public void attributeDecl(String eltName, String attName,
                              String type, String defaultValue,
                              String value)
        throws SAXException;

    public void elementDecl(String name, String model)
        throws SAXException;

    public void externalEntityDecl(String name, String publicID,
                                   String systemID)
        throws SAXException;

    public void internalEntityDecl(String name, String value)
        throws SAXException;
}
```

Этот код достаточно прозрачен. Первые два метода обрабатывают конструкции <!ELEMENT> и <!ATTLIST>. Третий метод, externalEntityDecl(), получает уведомления об объявлениях сущностей (при помощи <!ENTITY>), ссылающихся на внешние ресурсы. Последний метод, internalEntityDecl(), сообщает о сущностях, определяемых в самом документе. Вот, собственно, и все, о чем тут можно рассказать.

Итак, я предоставил вам всю информацию по SAX. Ну, может быть, это и некоторое преувеличение; несомненно, однако, что вы приобрели достаточные навыки, чтобы начать действовать. Осталось лишь приступить к разработке кода, постепенно создавая собственный набор инструментов и приемов. Прежде чем закрыть тему SAX, рассмотрим несколько распространенных ошибок, допускаемых при работе с этим API.

Советы разработчикам

Применение дополнительных возможностей SAX, разумеется, не способствует сокращению количества потенциальных проблем. Но зачастую новые проблемы менее очевидны и приводят к ошибкам, которые очень трудно обнаружить. Сейчас мы рассмотрим некоторые из распространенных проблем.

Значения, возвращаемые EntityResolver

Как было сказано в разделе, посвященном EntityResolver, следует помнить, что реализация метода resolveEntity() должна возвращать как минимум значение null. К счастью, Java гарантирует, что метод возвращает какое-либо значение, но мне часто приходилось видеть нечто, подобное нижеследующему:

```
public InputSource resolveEntity(String publicID, String systemID)
    throws IOException, SAXException {

    InputSource inputSource = new InputSource();

    // Обрабатываем ссылки на online-версию файла copyright.xml
    if (systemID.equals(
        "http://www.newInstance.com/javaxml2/copyright.xml")) {
        inputSource.setSystemId(
            "file:///c:/javaxml2/ch04/xml/copyright.xml");
    }

    // По умолчанию возвращаем null
    return inputSource;
}
```

Как видите, сначала создается экземпляр InputSource, а затем для него устанавливается системный идентификатор. Проблема тут заключается в том, что если программа не входит в блок if, в качестве результата работы метода возвращается InputSource, не инициализированный системным или публичным идентификатором, а также не связанный с объектом типа Reader или InputStream. Это может привести к непредсказуемым результатам, но в некоторых анализаторах обработка продолжается без проблем. В то время как в других анализаторах возвращение пустого InputSource приводит к тому, что игнорируются сущности либо генерируются исключения. Другими словами, возвращаейте null в конце каждого метода resolveEntity(), и вам не придется беспокоиться об этих проблемах.

DTDHandler и проверка действительности

В этой главе описаны установка свойств и возможностей, их влияние на проверку действительности, а также интерфейс DTDHandler. В ходе моего рассказа о DTD и проверке действительности вы могли не так понять некоторые вещи. Поэтому я уточню, что интерфейс DTDHandler никоим образом не связан с проверкой действительности. Многие разработчики регистрируют реализацию DTDHandler и удивляются, почему не выполняется проверка действительности. DTDHandler всего лишь предоставляет уведомления о нотациях и объявлениях неанализируемых сущностей! Вполне вероятно, что разработчик ожидает чего-то дру-

гого. Запомните, что проверку действительности устанавливает свойство, а не экземпляр обработчика:

```
reader.setFeature("http://xml.org/sax/features/validation", true);
```

Никак иначе (исключая вариант анализатора, выполняющего проверку действительности по умолчанию) задать проверку действительности нельзя. Помните об этом и вы избежите многих разочарований.

Анализ объекта reader вместо объекта filter

В этой главе мы говорили о конвейерах в SAX, и, надеюсь, вы поняли, насколько полезными они могут быть. У начинающих разработчиков, применяющих фильтры, я вновь и вновь встречаю одну и ту же ошибку, которая служит источником больших неприятностей. Проблема заключается в неверной инициализации конвейера: это случается, когда фильтр не устанавливает предыдущий фильтр в качестве родительского, включая и экземпляр `XMLReader`. Посмотрите на этот фрагмент кода:

```
public void buildTree(DefaultTreeModel treeModel,
                      DefaultMutableTreeNode base, String xmlURI)
    throws IOException, SAXException {

    // Создаем экземпляры, необходимые для анализа
    XMLReader reader =
        XMLReaderFactory.createXMLReader(vendorParserClass);
    XMLWriter writer =
       new XMLWriter(reader, new FileWriter("snapshot.xml"));
    NamespaceFilter filter =
        new NamespaceFilter(reader,
            "http://www.oreilly.com/javaxml2",
            "http://www.oreilly.com/catalog/javaxml2");
    ContentHandler jTreeContentHandler =
        new JTreeContentHandler(treeModel, base, reader);
    ErrorHandler jTreeErrorHandler = new JTreeErrorHandler();

    // Регистрируем обработчик содержимого
    reader.setContentHandler(jTreeContentHandler);

    // Регистрируем обработчик ошибок
    reader.setErrorHandler(jTreeErrorHandler);

    // Регистрируем интерпретатор сущностей
    reader.setEntityResolver(new SimpleEntityResolver());

    // Анализируем
    InputSource inputSource =
        new InputSource(xmlURI);
    reader.parse(inputSource);
}
```

Заметили ошибку? Анализ выполняется для экземпляра `XMLReader`, а не для последнего звена цепи. Кроме того, экземпляр `NamespaceFilter` устанавливает в качестве родительского класс `XMLReader`, а не экземпляр класса `XMLWriter`, который должен предшествовать ему в цепи. Эти ошибки не очевидны, но вместо запланированного конвейера получается полный беспорядок. В этом примере фильтрации не будет вообще, поскольку анализ происходит для объекта `reader`, а не для фильтра. Если эту ошибку исправить, вывода все равно не будет, т. к. `writer` – по причине неверной установки родительского экземпляра для `NamespaceFilter` – не включен в конвейер вовсе. Установив корректно родительские связи, вы, наконец, получите желаемый результат. При создании конвейеров SAX следует проявлять внимание в отношении родительских экземпляров классов и анализа.

Что дальше?

Я привел достаточно информации по простому API для XML. И хотя сказать можно еще очень много, того, что есть в этой и предыдущей главах, должно хватить, чтобы подготовить читателя практически ко всему, с чем он может столкнуться. Конечно же, SAX – это не единственный API для работы с XML; чтобы стать настоящим экспертом в XML, нужно владеть технологиями DOM, JDOM, JAXP и другими. В следующей главе я познакомлю вас со следующим API из списка – объектной моделью документа (Document Object Model, DOM).

Наше знакомство с DOM мы начнем с основ, аналогично тому, как в предыдущей главе мы это сделали для SAX. Вы узнаете об API для деревьев и о том, что DOM значительно отличается от SAX, а также изучите основные классы DOM. В качестве примера приводится приложение, сериализующее деревья DOM, и скоро вы начнете создавать собственный код для работы с DOM.

- *Объектная модель документа*
- *Сериализация*
- *Изменяемость*
- *Советы разработчикам*
- *Что дальше?*

5

Объектная модель документа

В предыдущих главах мы говорили о Java и XML в общем смысле, но подробно рассмотрели только SAX. Возможно, вы знаете, что SAX – это всего лишь один из нескольких интерфейсов прикладного программирования, позволяющий выполнять работу, связанную с XML, в Java. Эта и следующая глава расширят ваши знания об API, т. к. я расскажу об объектной модели документа, или DOM (Document Object Model). Этот API довольно сильно отличается от SAX и во многих отношениях дополняет его. Чтобы стать разработчиком, компетентным в XML, вам понадобятся оба этих интерфейса, а также и другие API и инструменты, рассмотренные в этой книге.

Поскольку DOM фундаментально отличается от SAX, я потрачу довольно много времени на обсуждение концепций, лежащих в основе DOM, и расскажу, почему для некоторых приложений DOM можно использовать вместо SAX. Выбор API для XML – это компромисс, и выбор между DOM и SAX, разумеется, не исключение. Мы перейдем к наиболее важной теме – коду. И рассмотрим вспомогательный класс, сериализующий деревья DOM (в DOM API эта возможность в настоящее время не поддерживается). Это позволит довольно подробно изучить структуру DOM и сопутствующих классов и подготовит вас к более сложным аспектам работы с DOM. Наконец, я раскрою несколько проблемных областей и важных аспектов DOM в разделе «Советы разработчикам».

Объектная модель документа

В отличие от SAX, объектная модель документа (DOM) берет свое начало в консорциуме World Wide Web (W3C). В то время как SAX является открытым программным продуктом, созданным в результате длительных дискуссий среди участников списка рассылки *XML-dev*,

DOM представляет собой стандарт, точно так же, как и сама спецификация XML. Кроме того, модель DOM не привязана к Java – она создавалась для представления содержимого и модели документов в самых разнообразных языках программирования и средствах разработки. Существуют частные реализации DOM для JavaScript, Java, CORBA и других инструментов программирования, что позволяет DOM быть не только межплатформенной, но и межъязыковой спецификацией.

Помимо отличий от SAX в плане стандартизации и связи с языками программирования, DOM организован в виде уровней (levels), а не версий. DOM Level One (первого уровня) является утвержденной рекомендацией W3C, и законченная спецификация доступна по адресу *http://www.w3.org/TR/REC-DOM-Level-1*. Первый уровень описывает функциональность и навигацию по содержимому документа. Документы в DOM не ограничиваются документами XML, они включают документы HTML, а также документы с другими моделями содержимого. Спецификация DOM Level Two (второго уровня), завершенная в ноябре 2000 года, дополняет первый уровень введением модулей и факультативных средств, предназначенных для специфических моделей содержимого документа, таких как XML, HTML и CSS (каскадные таблицы стилей, Cascading Style Sheets). Таким образом, более специализированные модули начинают заполнять пробелы, оставленные общим инструментарием DOM Level One. Текущий вариант рекомендации можно найти на странице *http://www.w3.org/TR/DOM-Level-2/*. DOM Level 3 находится в процессе разработки, и предполагается, что в нем появятся дополнительные модули для конкретных типов документов, таких как обработчики проверки действительности для XML, а также другие возможности, о которых рассказывается в главе 6.

Частные реализации DOM

Применение DOM для определенного языка программирования требует присутствия набора интерфейсов и классов, определяющих и реализующих собственно объектную модель документа. Поскольку спецификация DOM фокусирует внимание на модели документа, а конкретные методы в ней не описаны, для представления концептуальной структуры DOM и ее использования в Java или любом другом языке программирования необходимо разрабатывать частные реализации (*language bindings*). Эти реализации выступают в качестве API для манипулирования документами в соответствии со способами, описанными в спецификации DOM.

Нас, разумеется, интересует реализация DOM для Java. Последнюю версию DOM Level 2 для Java можно загрузить со страницы *http://www.w3.org/TR/DOM-Level-2/java-binding.html*. Классы, которые следует добавить в переменную `classpath`, находятся в пакете `org.w3c.dom` и его подпакетах. Но прежде чем начать установку этих классов, следует изучить анализатор XML и процессор XSLT, которые вы приобрели

или загрузили через Интернет. Подобно пакетам SAX, пакеты DOM часто включаются в состав этих продуктов. Это гарантирует корректную совместную работу анализатора, процессора и реализованного уровня DOM.

Большинство процессоров XSLT не берут на себя решение задачи генерирования исходного дерева DOM, и в этом вопросе полагаются на анализаторы XML. Так образуется слабое зацепление между анализатором и процессором, что позволяет заменять тот или другой сравнимыми продуктами. Поскольку Apache Xalan по умолчанию использует для анализа XML и создания дерева DOM Apache Xerces, нас будет интересовать уровень поддержки DOM, который обеспечивает Xerces. Это же справедливо, если применяется процессор XSLT и анализатор XML от Oracle.[1]

Основы

Рассмотрев основные принципы спецификации DOM, поговорим немного о программной структуре самой объектной модели документа (DOM). В основе DOM лежит модель дерева. Вы помните, что SAX представляет XML-документ фрагмент за фрагментом, сообщая о событиях по мере их возникновения в цикле анализа. Во многих отношениях модель DOM является противоположностью модели SAX, поскольку хранит полное представление документа в памяти. Документ представляется в виде дерева, целиком построенного на интерфейсе `org.w3c.dom.Node` DOM. DOM предоставляет и несколько производных интерфейсов, специфичных для XML, таких как `Element`, `Document`, `Attr` и `Text`. Так что в случае обычного документа можно получить структуру, подобную представленной на рис. 5.1.

Модель дерева соблюдается во всех отношениях. Особенно это заметно в случае с узлами `Element`, имеющими текстовые значения (возьмем для примера элемент `Title`). Текстовое значение узла не доступно через узел `Element` при помощи метода вроде `getText()`, вместо этого существует дочерний узел типа `Text`. Соответственно, значение и дочерний элемент (или элементы) можно получить из самого узла `Text`. И хотя это может показаться несколько странным, речь идет и о сохранении очень строгой модели дерева в DOM, которая позволяет решать такие задачи, как обход дерева, с помощью простых алгоритмов, избавляя нас от необходимости обрабатывать большое количество особых случаев. Благодаря этой модели все структуры DOM можно считать имеющими либо родовой тип `Node`, либо конкретный тип (`Element`, `Attr` и т. д.). Многие из методов навигации, вроде `getParent()` и `getChild-`

[1] Я не хочу сказать, что нельзя работать с анализатором и процессором от разных производителей. В большинстве случаев можно применять иной анализатор. Однако «по умолчанию» всегда лучше работать с программным обеспечением от одного производителя.

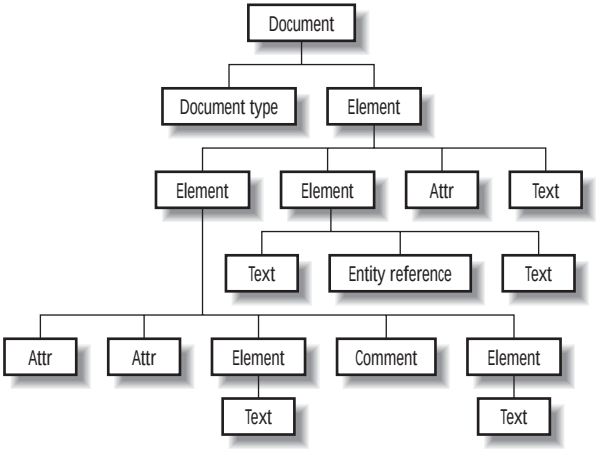

Рис. 5.1. Структура DOM, представляющая XML

`ren()`, принадлежат основному интерфейсу `Node`, так что можно перемещаться вверх и вниз по дереву, не задумываясь о конкретных типах структур.

Еще один важный аспект DOM заключается в том, что DOM, как и SAX, определяет собственные списочные структуры. При работе с DOM придется использовать классы `NodeList` и `NamedNodeMap`, а не коллекции Java. Независимо от вашей точки зрения, это ни положительный момент, ни отрицательный, а всего лишь жизненный факт. На рис. 5.2 приведена простая модель базовых интерфейсов и классов DOM в стиле UML, к которой можно обращаться на протяжении всей этой главы.

Почему не SAX?

И последнее замечание, прежде чем перейти к коду. Новички в XML могут удивиться, почему нельзя просто воспользоваться SAX для работы с XML. Но иногда применение SAX подобно стрельбе из пушки по воробьям – это просто неподходящий инструмент для такой работы. Вот некоторые моменты, связанные с SAX и делающие этот интерфейс не столь идеальным в некоторых ситуациях.

SAX последователен

Последовательная модель, которую обеспечивает SAX, не позволяет организовать прямой доступ к фрагментам документа XML. Другими словами, при работе с SAX мы получаем ту информацию о XML-документе, которую получает анализатор, и теряем те данные, которые теряет анализатор. Когда текущим элементом является второй элемент,

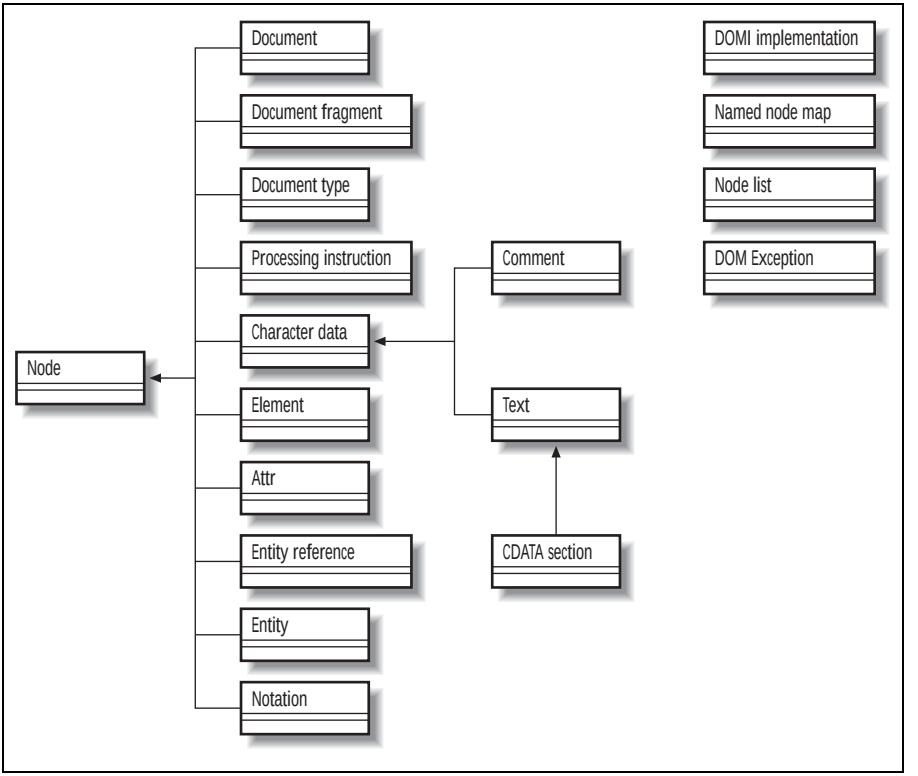

Рис. 5.2. UML-модель основных классов и интерфейсов DOM

невозможно получить доступ к данным из четвертого элемента, потому что четвертый элемент еще не был проанализирован. Когда текущим *становится* четвертый элемент, нет возможности вернуться ко второму элементу. Разумеется, мы имеем полное право сохранять полученную информацию, получаемую в процессе анализа, однако создание кода для всех специальных случаев может оказаться весьма сложным делом. Другое, более радикальное решение связано с созданием в памяти представления XML-документа. Через минуту мы увидим, что анализатор DOM делает именно это, а значит, решение той же самой задачи с помощью SAX было бы бессмысленным и, вероятно, более медленным и трудоемким.

Одноуровневые элементы в SAX

Другой задачей, которую трудно решить в модели SAX, является перемещение между одноуровневыми элементами. Доступ к данным в SAX не только последовательный, но и по большей части иерархический. Сначала мы перемещаемся к дочерним узлам (листьям) первого элемента, затем возвращаемся по дереву обратно, затем снова вниз

к дочерним узлам второго элемента и т. д. Ни в какой момент времени нет ясных указаний на то, на каком уровне иерархии мы находимся. Хотя задачу можно решить с помощью каких-нибудь «умных» счетчиков, это отнюдь не то, для чего спроектирован SAX. В нем не существует понятия одноуровневых элементов, не существует понятия следующего элемента того же уровня или представлений о связях вложенных элементов.

Такой недостаток информации служит источником проблем. Скажем, процессор XSLT (см. главу 2) должен иметь возможность определять элементы одного уровня и, что еще важнее, дочерние элементы данного элемента. Рассмотрим следующий фрагмент кода шаблона XSL:

```
<xsl:template match="parentElement">
  <!-- Добавляем содержимое в конечное дерево -->
  <xsl:apply-templates select="childElementOne|childElementTwo" />
</xsl:template>
```

В данном случае шаблоны применяются с помощью конструкции `xsl:apply-templates`, но они применяются к определенному набору узлов, соответствующих указанному выражению XPath. Шаблон должен применяться только к элементам `ChildElementOne` и `ChildElementTwo` (разделенных XPath-оператором «ИЛИ», символом «|»). К тому же, поскольку используется относительный путь, это должны быть непосредственные потомки элемента `ParentElement`. Определение и поиск этих узлов в контексте XML-структур SAX – задача, которую исключительно сложно решить. В иерархическом, хранимом в памяти представлении XML-документа, поиск этих узлов – тривиальная задача, что является основной причиной, по которой модель DOM интенсивно используется для передачи данных в процессоры XSLT.

Зачем вообще использовать SAX?

Все эти споры о «недостатках» SAX, возможно, вызвали у вас вопрос, зачем вообще использовать SAX. Но все эти недостатки проявляются в специфическом применении XML-документов, в данном случае – при обработке их с помощью XSL или при необходимости получить прямой доступ для любых других целей. В действительности же все эти «проблемы» с применением SAX являются и основаниями для того, чтобы остановить на нем свой выбор.

Представьте себе анализ оглавления номера журнала «National Geographic», представленного в формате XML. Этот документ вполне может содержать даже более 500 строк , если в данном выпуске достаточно много материала. Представьте себе предметный указатель в формате XML какой-либо книги издательства O'Reilly. Сотни слов с номерами страниц, перекрестные ссылки и т. д. Речь идет о приложениях XML небольшого масштаба. По мере роста размера XML-документа растет и его DOM-представление в памяти. Представьте (да, продол-

жайте представлять) себе настолько большой XML-документ и с таким большим количеством вложенных элементов, что его хранение в дереве DOM начинает влиять на производительность вашего приложения. А теперь представьте себе, что задачи, ради решения которых создается приложение, могут быть решены путем последовательного анализа того же самого исходного документа с помощью SAX, и для этого потребуется всего лишь одна десятая или одна сотая часть ресурсов вашей системы.

Так же как в Java существует множество способов решения одной и той же задачи, так при обработке XML существует много способов получения данных из документов. В некоторых ситуациях предпочтителен SAX – для реализации быстрого, нетребовательного к ресурсам процесса анализа и обработки. В других случаях DOM обеспечивает простой в применении, прозрачный интерфейс для доступа к данным в желаемом формате. Именно разработчик должен проанализировать нужды приложения и позаботиться о принятии правильного решения: использовать DOM или SAX, или даже сочетание моделей. Как всегда, способность принимать хорошее или плохое решение зиждется на знании альтернатив. Имея это в виду, рассмотрим DOM в действии.

Сериализация

Одним из наиболее распространенных вопросов по DOM является следующий: «Есть дерево DOM, как записать его в файл?» Этот вопрос задается столь часто из-за того, что DOM уровня 1 и 2 не предоставляет стандартного способа сериализации деревьев DOM. И хотя это в некоторой степени недостаток интерфейса, он обеспечивает отличный пример использования DOM (и как вы увидите в следующей главе, в DOM Level 3 эта проблема будет исправлена). Продолжая знакомство с DOM, в этом разделе мы рассмотрим класс, исходными данными для которого служит дерево DOM и который сериализует это дерево на выходе.

Где взять анализатор DOM

Перед тем как начать разговор о выводе дерева DOM, давайте выясним, где взять дерево DOM. Ради примера код, приведенный в этой главе, лишь читает файл, создает дерево DOM, а затем записывает это дерево в другой файл. Однако это неплохое введение в работу с DOM, которое подготовит вас к некоторым более сложным темам из следующей главы.

Итак, в этой главе нас интересуют два исходных файла на Java. Первый – это собственно код сериализатора под названием (не удивительно) *DOMSerializer.java*. Второй, с которого мы и начнем, – *SerializerTest.java*. Этот класс принимает имя файла исходного XML-документа и имя файла, в который будет выводиться сериализованный документ. Кроме того, этот класс демонстрирует, как получить файл,

проанализировать его и получить конечный объект в виде дерева DOM, представленный классом org.w3c.dom.Document. Загрузите этот класс с веб-сайта книги либо введите код класса SerializerTest из примера 5.1.

Пример 5.1. Класс SerializerTest

```
package javaxml2;

import java.io.File;
import org.w3c.dom.Document;

// Импортируем анализатор
import org.apache.xerces.parsers.DOMParser;

public class SerializerTest {

    public void test(String xmlDocument, String outputFilename)
        throws Exception {

        File outputFile = new File(outputFilename);
        DOMParser parser = new DOMParser();

        // Получаем дерево DOM в виде объекта Document

        // Сериализация
    }

    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println(
                "Использование: java javaxml2.SerializerTest " +
                "[исходный XML-документ] " +
                "[имя файла для вывода]");
            System.exit(0);
        }

        try {
            SerializerTest tester = new SerializerTest();
            tester.test(args[0], args[1]);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Очевидно, что в этом примере пропущена пара фрагментов, представленных двумя комментариями в методе test(). Я приведу их в следующих двух разделах. Сначала поговорим о том, как получить дерево DOM, а затем подробно рассмотрим и сам класс DOMSerializer.

Результат работы анализатора DOM

Как вы помните, в SAX наибольший интерес в работе анализатора представлял сам цикл процесса синтаксического анализа, поскольку

методы обратного вызова обеспечивали нам доступ к анализируемым данным. В DOM внимание сосредоточено на результате процесса анализа. До тех пор пока весь документ не проанализирован и не загружен в конечную древовидную структуру, данные документа использовать невозможно. Результатом анализа, предназначенным для использования с интерфейсами DOM, является объект `org.w3c.dom.Document`. Он используется в качестве «дескриптора» для доступа к древовидной структуре, в которой хранятся XML-данные, и в терминах иерархии элементов, которую мы здесь обсуждаем, он эквивалентен уровню, находящемуся непосредственно над корневым элементом XML-документа. Другими словами, он является «владельцем» всех элементов исходного документа XML.

Поскольку стандарт DOM сосредоточен на манипуляции данными, существует множество механизмов, применяемых для получения объекта `Document` после выполнения анализа. Во многих реализациях, таких как ранние версии анализатора IBM XML4J, метод `parse()` возвращал объект `Document`. Код, использующий такую реализацию DOM-совместимого анализатора, мог бы выглядеть следующим образом:

```
File outputFile = new File(outputFilename);
DOMParser parser = new DOMParser();
Document doc = parser.parse(xmlDocument);
```

Более новые анализаторы, такие как Apache Xerces, не следуют этой методологии. Чтобы сохранить непротиворечивость интерфейсов для анализаторов, поддерживающих SAX и DOM, метод `parse()` в этих анализаторах возвращает значение типа `void`, как это делал метод `parse()` в примере использования SAX. Это изменение позволяет приложению использовать класс анализатора DOM и класс анализатора SAX взаимозаменяемым образом, однако в таком случае необходим дополнительный метод для получения документа, являющегося результатом синтаксического анализа XML. В Apache Xerces этот метод называется `getDocument()`. Работая с анализатором данного типа (как я делаю в примере), можно добавить следующий код в метод `test()` с целью получения конечного дерева DOM в результате анализа заданного входного файла:

```
public void test(String xmlDocument, String outputFilename)
    throws Exception {

    File outputFile = new File(outputFilename);
    DOMParser parser = new DOMParser();

    // Получаем дерево DOM в виде объекта Document
    parser.parse(xmlDocument);
    Document doc = parser.getDocument();
    // Сериализация
}
```

Конечно же, тут подразумевается, что вы используете анализатор Xerces, о чем говорит оператор `import` в начале исходного файла:

```
import org.apache.xerces.parsers.DOMParser;
```

Если вы пользуетесь иным анализатором, то понадобится изменить этот оператор соответствующим образом. Также следует обратиться к документации от разработчика системы, чтобы определить, какие из механизмов метода `parse()` следует применять для получения результата анализа. В главе 7 мы рассмотрим JAXP API, разработанный фирмой Sun, а также другие способы стандартизации средств доступа к дереву DOM с помощью любой реализации анализатора. Итак, что касается дерева DOM, существуют некоторые различия в методах его получения, однако все случаи его использования, которые мы будем рассматривать, являются стандартными по отношению к спецификации DOM, поэтому на протяжении этой главы вам не следует беспокоиться о прочих особенностях реализаций анализаторов.

DOMSerializer

Я время от времени употребляю термин *сериализация* и мне, вероятно, следует убедиться, что вы понимаете, что я имею в виду. Когда я говорю «сериализация», я просто имею в виду вывод данных XML. Это может быть файл (объект `File` Java), объект `OutputStream` или `Writer`. Разумеется, в Java существуют и другие формы вывода, но эти три охватывают большинство из них (на самом деле две последние, поскольку `File` можно легко преобразовать к объекту `Writer`, но объект `File` просто удобно использовать). В данном случае речь идет о сериализации в формате XML: дерево DOM преобразуется обратно в корректный XML-документ в текстовом формате. Очень важно заметить, что используется формат XML, хотя можно было бы легко написать сериализатор, выводящий данные в HTML, WML, XHTML или ином формате. Кстати, Apache Xerces предоставляет подобные классы, и я кратко коснусь их в конце этой главы.

Начинаем

Чтобы закончить с предварительными замечаниями, взглянем на пример 5.2, в котором приводится каркас класса `DOMSerializer`. Он импортирует все необходимые классы и определяет различные точки входа (для объектов типа `File`, `OutputStream` и `Writer`) для класса. Два из этих трех методов просто передают управление третьему (при помощи определенной магии ввода/вывода). Помимо этого устанавливаются значения переменных, связанных с отступами и разделением строк, а также создаются методы для изменения этих свойств.

Пример 5.2. Каркас класса DOMSerializer

```
package javaxml2;
import java.io.File;
```

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class DOMSerializer {

    /** Отступ */
    private String indent;

    /** Разделитель строк */
    private String lineSeparator;

    public DOMSerializer() {
        indent = "";
        lineSeparator = "\n";
    }

    public void setLineSeparator(String lineSeparator) {
        this.lineSeparator = lineSeparator;
    }

    public void serialize(Document doc, OutputStream out)
        throws IOException {

        Writer writer = new OutputStreamWriter(out);
        serialize(doc, writer);
    }

    public void serialize(Document doc, File file)
        throws IOException {

        Writer writer = new FileWriter(file);
        serialize(doc, writer);
    }

    public void serialize(Document doc, Writer writer)
        throws IOException {

        // Сериализация документа
    }
}
```

Сохранив код в исходном файле *DOMSerializer.java*, мы получим версию метода `serialize()`, работающую с объектом `Writer`. Аккуратно и просто.

Запускаем сериализацию

Закончив предварительную подготовку к сериализации, мы должны определить порядок работы с деревом DOM. Один приятный аспект DOM, о котором я уже говорил, заключается в том, что все XML-структуры DOM (включая и объект Document) расширяют интерфейс Node. Это позволяет создать единственный метод, отвечающий за сериализацию узлов DOM всех типов. В пределах этого метода можно различать типы узлов, но на входе принимать Node, охватывая таким образом все существующие типы DOM. Дополнительно такой подход позволяет использовать рекурсию, а рекурсия, как известно, – лучший друг любого программиста. Добавьте приведенный здесь метод serializeNode(), а также первоначальный вызов этого метода из метода serialize() (распространенный момент, о котором только что шла речь):

```
public void serialize(Document doc, Writer writer)
    throws IOException {

    // Начало рекурсивной сериализации без отступов
    serializeNode(doc, writer, "");
    writer.flush();
}

public void serializeNode(Node node, Writer writer,
                          String indentLevel)
    throws IOException {
}
```

Кроме того, обратите внимание на параметр indentLevel. Метод serializeNode() может получать информацию о том, какой отступ следует использовать для текущего узла, а когда происходит рекурсивный вызов, может добавить дополнительный уровень отступа (при помощи переменной-члена indent). В самом начале (в методе serialize()) для создания отступов используется пустая строка. На следующем уровне по умолчанию используются два пробела, на следующем уровне четыре и т. д. Возврат из рекурсивных вызовов приводит к сокращению ширины отступов. Теперь нам осталось лишь обработать различные типы узлов.

Работаем с узлами

Первое, что следует сделать в методе serializeNode(), – определить, какого типа узел получен. И хотя можно было бы решить задачу в стиле языка Java, при помощи ключевого слова instanceof и Reflection API,[1] частная реализация DOM для Java позволяет сделать это гораздо проще. Интерфейс Node определяет вспомогательный метод getNodeType(), возвращающий целое значение. При помощи набора констант (также

[1] Пакет java.lang.reflect. – *Примеч. науч. ред.*

определяемых в интерфейсе Node) можно быстро и просто определить тип изучаемого узла. Такой метод хорошо сочетается с конструкцией switch, которую можно использовать для разбиения метода сериализации на логические разделы. Приводимый ниже код охватывает практически все типы узлов DOM. Хотя существует несколько дополнительных типов узлов (см. рис. 5.2), эти наиболее распространены, и приведенные здесь концепции можно применить к узлам менее распространенных типов:

```
public void serializeNode(Node node, Writer writer,
                          String indentLevel)
    throws IOException {

    // Определяем действие исходя из типа узла
    switch (node.getNodeType()) {
        case Node.DOCUMENT_NODE:
            break;

        case Node.ELEMENT_NODE:
            break;

        case Node.TEXT_NODE:
            break;

        case Node.CDATA_SECTION_NODE:
            break;

        case Node.COMMENT_NODE:
            break;

        case Node.PROCESSING_INSTRUCTION_NODE:
            break;

        case Node.ENTITY_REFERENCE_NODE:
            break;

        case Node.DOCUMENT_TYPE_NODE:
            break;
    }
}
```

Этот код практически бесполезен, но он позволяет увидеть типы узлов DOM вне контекста кода, выполняющего собственно сериализацию. О коде мы поговорим прямо сейчас, и начнем с первого узла, передаваемого методу, – экземпляра интерфейса Document.

Поскольку интерфейс Document – это расширение интерфейса Node, его можно использовать равнозначно с другими типами узлов. Однако это особый случай, т. к. он содержит корневой элемент, а также DTD XML-документа и некоторую другую особую информацию, не принадлежащую иерархии элементов XML. Следовательно, надо выделить корневой элемент и передать его методу, выполняющему сериализа-

цию (начало рекурсии). Кроме того, мы выводим собственно объявление XML:

```
case Node.DOCUMENT_NODE:
    writer.write("<?xml version=\"1.0\"?>");
    writer.write(lineSeparator);

    Document doc = (Document)node;
    serializeNode(doc.getDocumentElement(), writer, "");
    break;
```

Внимание

DOM Level 2 (как и SAX 2.0) не предоставляет доступа к объявлению XML. Может показаться, что это не важно. Но вспомним, что именно это объявление содержит информацию о кодировке документа. Ожидается, что в DOM Level 3 этот недостаток будет исправлен, но об этом мы поговорим в следующей главе. Старайтесь не писать DOM-приложений, зависящих от этой информации, до тех пор, пока эта возможность не будет реализована.

Наш код должен иметь доступ к методам интерфейса `Document` (а не родового интерфейса `Node`), отсюда и приведение реализации `Node` к интерфейсу `Document`. С помощью метода `getDocumentElement()` производится извлечение корневого элемента исходного XML-документа, и этот элемент передается методу `serializeNode()`. Таким образом, начинается цепочка рекурсивных вызовов и обход дерева DOM.

Конечно же, наиболее распространенной задачей в сериализации является обработка элемента DOM: вывод его имени, атрибутов, содержимого, а также всех дочерних элементов. Как можно ожидать, все это легко сделать при помощи методов DOM. Прежде всего, следует получить имя элемента XML при помощи метода `getNodeName()` интерфейса `Node`. Затем надо получить дочерние элементы и также подвергнуть их сериализации. Получить доступ к дочерним элементам узла `Node` можно при помощи метода `getChildNodes()`, возвращающего экземпляр `NodeList`. Можно элементарно узнать длину этого списка и затем обойти в цикле все дочерние элементы, выполняя для каждого рекурсивную сериализацию. Кроме того, в коде присутствует логика, гарантирующая наличие корректных отступов и разделителей строк. Вопросов форматирования мы не будем касаться более подробно. Наконец, завершающий этап, связанный непосредственно с выводом элемента:

```
case Node.ELEMENT_NODE:
    String name = node.getNodeName();
    writer.write(indentLevel + "<" + name);
    writer.write(">");

    // Рекурсивный вызов для каждого дочернего элемента
    NodeList children = node.getChildNodes();
    if (children != null) {
        if ((children.item(0) != null) &&
```

```
                (children.item(0).getNodeType() ==
                Node.ELEMENT_NODE)) {

                writer.write(lineSeparator);
            }
            for (int i=0; i<children.getLength(); i++) {
                serializeNode(children.item(i), writer,
                    indentLevel + indent);
            }
            if ((children.item(0) != null) &&
                (children.item(children.getLength()-1)
                        .getNodeType() ==
                Node.ELEMENT_NODE)) {

                writer.write(indentLevel);
            }
        }

        writer.write("</" + name + ">");
        writer.write(lineSeparator);
        break;
```

Конечно же, внимательные читатели (или эксперты в DOM) заметили, что я упустил нечто очень важное: атрибуты элемента! Атрибуты – единственное подобие исключения из правил в строгом дереве модели DOM. Вообще говоря, они и должны быть исключением, поскольку атрибут в действительности не является дочерним элементом для элемента, он (в некотором роде) просто связан с элементом. По существу, это родство несколько туманно. В любом случае атрибуты элемента могут быть получены с помощью метода `getAttributes()` интерфейса `Node`. Этот метод возвращает объект `NamedNodeMap`, который также может быть обработан в цикле. У каждого узла `Node` из данного списка можно запросить его имя и значение, и вот – атрибуты уже обработаны! Добавьте код, приведенный ниже:

```
case Node.ELEMENT_NODE:
    String name = node.getNodeName();
    writer.write(indentLevel + "<" + name);
    NamedNodeMap attributes = node.getAttributes();
    for (int i=0; i<attributes.getLength(); i++) {
        Node current = attributes.item(i);
        writer.write(" " + current.getNodeName() +
                     "=\"" + current.getNodeValue() +
                     "\"");
    }
    writer.write(">");

    // Рекурсивный вызов для каждого дочернего элемента
    NodeList children = node.getChildNodes();
    if (children != null) {
        if ((children.item(0) != null) &&
            (children.item(0).getNodeType() ==
```

```
                Node.ELEMENT_NODE)) {

                writer.write(lineSeparator);
            }
            for (int i=0; i<children.getLength(); i++) {
                serializeNode(children.item(i), writer,
                    indentLevel + indent);
            }
            if ((children.item(0) != null) &&
                (children.item(children.getLength()-1)
                        .getNodeType() ==
                Node.ELEMENT_NODE)) {

                writer.write(indentLevel);
            }
        }

        writer.write("</" + name + ">");
        writer.write(lineSeparator);
        break;
```

Далее по списку типов – узлы `Text`. Выводить их достаточно просто, нужно лишь воспользоваться уже знакомым методом `getNodeValue()` интерфейса `Node` для получения и вывода текстовых данных. То же справедливо и для узлов `CDATA` с одним исключением: данные в секции `CDATA` должны быть выделены при помощи семантики XML, используемой для элемента `CDATA` (т. е. окружены маркерами `<![CDATA[` и `]]>`). Теперь можно добавить код для этих двух случаев:

```
case Node.TEXT_NODE:
    writer.write(node.getNodeValue());
    break;

case Node.CDATA_SECTION_NODE:
    writer.write("<![CDATA[" +
                 node.getNodeValue() + "]]>");
    break;
```

Работать с комментариями в DOM настолько просто, что дальше некуда. Метод `getNodeValue()` возвращает текст, выделенный XML-лексемами `<!--` и `-->`. Это, пожалуй, и все, что можно сказать. А вот собственно код:

```
case Node.COMMENT_NODE:
    writer.write(indentLevel + "<!-- " +
                 node.getNodeValue() + " -->");
    writer.write(lineSeparator);
    break;
```

Перейдем к следующему типу узлов в DOM. Реализация DOM для Java определяет интерфейс для инструкций обработки, которые встречаются в исходном документе. Этот интерфейс имеет вполне очевидное

имя: ProcessingInstruction. Это полезно, поскольку инструкции обработки не следуют той же модели разметки, которой следуют элементы и атрибуты XML, но, тем не менее, они могут представлять ценность для приложения. В нашем XML-документе, который содержит оглавление книги, инструкций обработки нет (хотя их можно легко добавить для тестирования).

Узел инструкции обработки (PI) в DOM несколько отличается от того, что мы видели до сих пор: в целях соблюдения синтаксиса модели интерфейса Node, метод getNodeValue() возвращает всю связанную с инструкцией информацию в одной строке. Это позволяет быстро вывести инструкцию на экран. Тем не менее для получения имени инструкции обработки все же нужно использовать метод getNodeName(). Если бы мы писали приложение, получающее инструкции обработки из XML-документа, можно было бы предпочесть интерфейс ProcessingInstruction — он выдает те же данные, но имена методов (getTarget() и getData()) больше соответствуют формату инструкции обработки. Приняв это во внимание, можно добавить следующий код для печати инструкций обработки, встреченных в исходном XML-документе:

```
case Node.PROCESSING_INSTRUCTION_NODE:
    writer.write("<?" + node.getNodeName() +
                 " " + node.getNodeValue() +
                 "?>");
    writer.write(lineSeparator);
    break;
```

Приведенный код вполне работоспособен, и все же существует проблема. При обработке узлов документа сериализатор всего лишь запрашивал элемент документа и производил рекурсивные вызовы. Проблема заключается в том, что такой подход не позволяет обработать прочие дочерние узлы объекта Document, в частности — инструкции обработки верхнего уровня и любые объявления DOCTYPE. Узлы этих типов в действительности находятся на том же уровне, что и корневой элемент, а следовательно, игнорируются. Поэтому следующий код сериализует *все* дочерние узлы полученного объекта Document, а не только те, которые являются потомками корневого элемента документа:

```
case Node.DOCUMENT_NODE:
    writer.write("<xml version=\"1.0\">");
    writer.write(lineSeparator);

    // Рекурсивный вызов для каждого дочернего элемента
    NodeList nodes = node.getChildNodes();
    if (nodes != null) {
        for (int i=0; i<nodes.getLength(); i++) {
            serializeNode(nodes.item(i), writer, "");
        }
    }
    /*
```

```
            Document doc = (Document)node;
            serializeNode(doc.getDocumentElement(), writer, "");
            */
            break;
```

Теперь программа может обрабатывать и узлы DocumentType, с помощью которых представляются объявления DOCTYPE. Подобно инструкциям обработки, объявления DTD могут быть очень полезными – они позволяют получить внешнюю информацию, которая может пригодиться при обработке XML-документа. Однако поскольку могут присутствовать публичный и системный идентификаторы, а также другие данные, специфичные для DTD, чтобы получить доступ к этим дополнительным данным, следует выполнить приведение интерфейса Node к интерфейсу DocumentType. После этого мы можем воспользоваться вспомогательными методами для получения имени узла, которое является именем корневого элемента ограничиваемого документа, публичного идентификатора (если он существует) и системного идентификатора для DTD-определения, на которое ссылается документ. Используя эту информацию, можно сериализовать DTD:

```
        case Node.DOCUMENT_TYPE_NODE:
            DocumentType docType = (DocumentType)node;
            writer.write("<!DOCTYPE " + docType.getName());
            if (docType.getPublicId() != null)  {
                System.out.print(" PUBLIC \"" +
                    docType.getPublicId() + "\" ");
            } else {
                writer.write(" SYSTEM ");
            }
    writer.write("\"" + docType.getSystemId() + "\">");
            writer.write(lineSeparator);
            break;
```

Теперь осталось лишь обработать сущности и ссылки на сущности. В этой главе мы пропустим сущности и сосредоточимся на ссылках на сущности; подробности о сущностях и нотациях приведены в главе 6. Что касается ссылок, то их можно предварять символом & и завершать символом ; :

```
        case Node.ENTITY_REFERENCE_NODE:
            writer.write("&" + node.getNodeName() + ";");
            break;
```

Когда дело дойдет до вывода данных таких узлов, вы можете столкнуться с неожиданностями. Определение порядка обработки ссылок на сущности в DOM допускает значительную свободу и в значительной степени связано с функциональностью анализатора. В действительности большинство анализаторов XML интерпретируют и обрабатывают ссылки на сущности еще до того, как данные из XML-документа попадают в дерево DOM. Поэтому зачастую, ожидая увидеть ссылку на

сущность в структуре DOM, можно найти готовый текст или подставленные значения, а вовсе не ссылку на сущность. Чтобы проверить, как поведет себя в этих обстоятельствах анализатор, запустите класс `SerializerTest` для документа *contents.xml* (который рассмотрен в следующем разделе) и посмотрите, что произойдет со ссылкой на сущность `OreillyCopyright`. Между прочим, в Apache она останется ссылкой.

Вот и все! Как я говорил, существуют и другие типы узлов, но не стоит тратить время на их рассмотрение сейчас; главное – вы теперь понимаете, как работает DOM. В следующей главе мы «копнем» глубже, чем вам, вероятно, когда-либо хотелось. А пока соберем все кусочки вместе и посмотрим на результаты.

Результаты

Теперь, когда класс `DOMSerializer` закончен, осталось лишь вызвать метод `serialize()`. Для этого добавьте следующие строки к классу `SerializerTest`:

```
public void test(String xmlDocument, String outputFilename)
    throws Exception {

    File outputFile = new File(outputFilename);
    DOMParser parser = new DOMParser();

    // Получаем дерево DOM в виде объекта Document
    parser.parse(xmlDocument);
    Document doc = parser.getDocument();

    // Сериализация
    DOMSerializer serializer = new DOMSerializer();
    serializer.serialize(doc, new File(outputFilename));
}
```

Это довольно простое дополнение завершает класс и теперь можно запустить программу для документа *contents.xml* из главы 2, как показано далее:

```
C:\javaxml2\build>java javaxml2.SerializerTest
    c:\javaxml2\ch05\xml\contents.xml
    output.xml
```

И хотя сейчас мы не получим интересных результатов, можно открыть вновь созданный файл *output.xml* и проверить его правильность. Он должен содержать всю информацию из первоначального XML-документа, с учетом моментов, о которых говорилось в предыдущих разделах. Фрагмент файла *output.xml* приведен в примере 5.3.

Пример 5.3. Фрагмент файла output.xml

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">
```

```
<!--  "Java и XML", Оглавление  -->
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com">
  <title ora:series="Java">Java и XML</title>


  <!--  Список глав  -->

  <contents>
    <chapter number="2" title="Основы технологии">
      <topic name="Основы"></topic>

      <topic name="Ограничения"></topic>

      <topic name="Преобразования"></topic>

      <topic name="И далее..."></topic>

      <topic name="Что дальше?"></topic>

    </chapter>
```

Вы могли заметить, что в полученном выводе немало лишних пробелов. Дело в том, что сериализатор добавляет разделитель строк каждый раз, когда встречает в коде вызов `writer.write(lineSeparator)`. Разумеется, и само дерево DOM также содержит разделители строк, которые представляются узлами типа `Text`. В результате во многих случаях появляются двойные пустые строки, как можно видеть на примере вывода.

Внимание

Следует учитывать, что класс `DOMSerializer` показан в этой главе в качестве примера и не очень годится для использования в реальных приложениях. Разумеется, он может применяться, но следует понимать, что в данной реализации упущены некоторые важные моменты, в частности, связанные с кодировкой и более точной настройкой отступов, разделения и разбиения строк. Кроме того, обработка сущностей минимальна (полная их обработка заняла бы две таких главы!). Вероятно, ваш анализатор реализует собственный (а то и несколько) класс `serializer`, решающий поставленную задачу, по крайней мере, не хуже, чем пример из этой главы. Однако вы теперь понимаете, что происходит внутри этих классов. Если вы работаете с Apache Xerces, взгляните на классы `org.apache.xml.serialize`. Особенно полезны такие классы, как `XMLSerializer`, `XHTMLSerializer` и `HTMLSerializer`. До тех пор пока не появится DOM Level 3 со стандартным решением, используйте эти классы.

Изменяемость

В этой главе мы не рассмотрели тему, касающуюся изменения дерева DOM, и это бросается в глаза. Это не случайно – работать с DOM гораздо сложнее, чем с SAX. Вместо того чтобы заваливать вас информа-

цией, я хотел дать ясную картину различных типов узлов и структур, применяемых в DOM. В следующей главе помимо рассмотрения тонкостей, связанных с DOM Level 2 и 3, мы коснемся вопросов изменяемости дерева DOM, и в частности – вопросов создания деревьев DOM. Не поддавайтесь панике – помощь уже идет!

Советы разработчикам

Как и в предыдущих главах, я хочу повторно рассмотреть некоторые из «скользких мест», с которыми чаще всего сталкиваются начинающие разработчики на Java для XML. В этой главе речь шла о модели DOM, и в данном разделе это будет отражено. Некоторые из мыслей, приводимых здесь, скорее следует принять к сведению, а не считать руководством к действию при создании кода, но они помогут в принятии конструктивных решений о том, когда следует использовать DOM, а также будут способствовать пониманию того, что происходит «за кулисами» в ваших XML-приложениях.

Память, производительность и DOM с отсрочкой

Ранее я уже говорил о причинах использования DOM или SAX. Хотя я обращал внимание на то, что применение DOM требует, чтобы весь XML-документ был загружен в память и хранился в древовидной структуре, на эту тему можно говорить бесконечно. Слишком часто случается, что разработчик загружает свой обширный набор сложных XML-документов в процессор XSLT и запускает последовательность преобразований в автономном режиме, а сам идет перекусить. Вернувшись, он обнаруживает, что его компьютер под управлением Windows показывает ужасный синий экран, а Linux сообщает о проблемах с памятью. Этому разработчику и сотням таких, как он, мой совет: остерегайтесь применения DOM на больших объемах данных!

Использование DOM требует объема памяти, пропорционального размеру и сложности XML-документа. Однако стоит внимательнее изучить документацию по вашему анализатору. В современных анализаторах часто реализуется возможность отложенной загрузки (*deferred DOM*). DOM с отсрочкой загрузки пытается сократить потребление памяти за счет того, что считывание и загрузка информации, связанной с узлом DOM, происходит только в тот момент, когда информация запрошена. До этого существующие, но не используемые узлы просто «обнуляются». Это позволяет сократить потребление памяти в случае больших документов, которые не требуется обрабатывать целиком. Однако необходимо понимать, что сокращение потребления памяти увеличивает затраты времени на обработку. Раз узлы не хранятся в памяти постоянно и должны инициализироваться данными по запросу, увеличивается задержка, связанная с получением информации для узлов, к которым до этого не обращались. Таков компромисс. Однако

DOM с отложенной загрузкой зачастую позволяет спасти положение, если речь идет о больших документах.

Полиморфизм и интерфейс Node

Ранее в этой главе я уже подчеркивал, что DOM основан на древовидной модели. Также я говорил, что ключевым является родовой интерфейс org.w3c.dom.Node. Этот класс обеспечивает общую функциональность для всех классов DOM, а в некоторых случаях – и дополнительную. В частности, этот класс определяет метод под названием getNodeValue(), возвращающий значение типа String. Кажется неплохой идеей, не так ли? Можно быстро получить значение узла, не приводя при этом узел Node к нужному типу. Однако это уже не так хорошо, когда речь заходит о таких типах, как Element. Вы помните, что Element не имеет текстовых данных, а вместо этого имеет дочерний элемент типа Text? Поэтому узел Element в DOM не имеет осмысленного значения. В результате мы получаем что-то вроде #ELEMENT#. Точное значение варьируется в зависимости от анализатора, но мысль, думаю, ясна.

То же справедливо и для других методов интерфейса Node, таких как getNodeName(). Для узлов Text мы получим значение #TEXT#, которое вряд ли является полезным. Так в чем же тут ловушка? Просто при работе с различными типами DOM через интерфейс Node следует проявлять осторожность. В качестве нагрузки к удобству родового интерфейса можно получить неожиданные результаты.

Анализаторы DOM, генерирующие исключения SAX

В примере использования DOM из этой главы не были явно перечислены исключения, которые могут возникнуть в результате анализа документа. Вместо этого перехватывалось исключение более высокого уровня. Дело в том, что процесс формирования дерева DOM, как уже говорилось, возложен на конкретную реализацию анализатора и не всегда одинаков. Тем не менее, обычно считается хорошей практикой обрабатывать характерные исключения, которые могут возникнуть, и реагировать на них по-разному, поскольку тип исключения дает информацию о том, какая проблема имела место. Переработка в таком ключе основного метода SerializerTest, выполняющего анализ документа, может выявить весьма неожиданную сторону этого процесса. Для Apache Xerces это можно сделать следующим образом:

```
public void test(String xmlDocument, String outputFilename)
    throws Exception {

    try {
        File outputFile = new File(outputFilename);
        DOMParser parser = new DOMParser();
        parser.parse(xmlDocument);
        Document doc = parser.getDocument();
```

```
        } catch (IOException e) {
            System.out.println("Ошибка при чтении URI: " + e.getMessage());
        } catch (SAXException e) {
            System.out.println("Ошибка при анализе: " + e.getMessage());
        }

        // Сериализация
        DOMSerializer serializer = new DOMSerializer();
        serializer.serialize(doc, new File(outputFilename));
    }
```

Встречающееся здесь исключение `IOException` не должно вызывать удивления, поскольку оно означает ошибку при поиске указанного имени файла, как это было в приведенных ранее примерах использования SAX. Еще один фрагмент секции `catch` может вызвать мысль о том, что что-то не так – вы заметили, что может быть сгенерировано исключение `SAXException`? Анализатор DOM генерирует исключение SAX? Должно быть, мы импортировали не тот набор классов! Отнюдь; классы именно те, которые нужны. Вспомните – раньше мы говорили о том, что можно было бы создать древовидную структуру данных XML-документа при помощи SAX, но DOM предоставляет нам альтернативу такому подходу. Однако это не мешает *использовать* при этом SAX. На самом деле SAX обеспечивает быстрый и нетребовательный к ресурсам способ анализа документа. В этом случае проанализированные данные помещаются в дерево DOM. Поскольку никакого стандарта для создания дерева DOM не существует, такой подход является приемлемым и встречается довольно часто. Поэтому не удивляйтесь и не теряйтесь, заметив, что в своих приложениях DOM вы импортируете и перехватываете исключение `org.xml.sax.SAXException`.

Что дальше?

В главе 6 я по-прежнему буду вашим проводником в мире DOM, и мы рассмотрим некоторые дополнительные (и менее известные) возможности объектной модели документа. Для начала мы узнаем, как изменять деревья DOM и как их создавать. Затем перейдем к менее распространенным возможностям DOM. Начинающим адресовано рассмотрение дополнительных возможностей, включенных в DOM Level 2 (с какими-то вы уже знакомы, а с какими-то нет). Затем мы рассмотрим использование модуля HTML для DOM, который поможет при работе с DOM и веб-страницами. Наконец, приводится информация о нововведениях, ожидаемых в спецификации DOM Level 3. Такого количества «боеприпасов» вполне достаточно, чтобы завоевать мир при помощи DOM!

6

- *Изменения*
- *Пространства имен*
- *Модули DOM Level 2*
- *DOM Level 3*
- *Советы разработчикам*
- *Что дальше?*

Расширенный DOM

Так же как и в главе 4, нет ничего мистического в том, что мы будем рассматривать здесь. Все темы построены на информации по DOM, которая была дана в предыдущей главе. Однако за исключением первого раздела, посвященного изменениям, большая часть приводимой информации редко бывает нужна разработчикам. В то время как практически все, что можно найти в SAX (исключая, вероятно, `DTDHandler` и `DeclHandler`), оказывается полезным, я обнаружил, что многие из дополнительных возможностей DOM находят применение только при разработке специфических приложений. Так, если вы не занимаетесь логикой визуализации, вам, скорее всего, никогда не понадобится модуль HTML для DOM. То же самое касается и многих возможностей DOM Level 2 – если они нужны, то нужны очень сильно, а если нет, значит, не нужны вообще.

В этой главе представлены некоторые специальные темы, связанные с DOM; эта информация будет полезна при создании приложений DOM. Данная глава в большей степени, чем остальные, тяготеет к формату справочника. Чтобы получить сведения о модуле Traversal из DOM Level 2, просто обратитесь к соответствующему разделу. Тем не менее, код примеров главы усложняется постепенно, и тем, кто хочет получить полное представление о модели DOM, я рекомендую последовательное прочтение. Это позволяет привести более полезные примеры кода, а не бесполезные, надуманные и ничего не дающие. Так что пристегиваемся крепче и еще глубже погружаемся в мир DOM.

Изменения

В первую очередь поговорим об изменяемости деревьев DOM. Самое большое ограничение в случае применения SAX для работы с XML заключается в том, что нельзя изменить структуру XML, с которой вы работаете, по крайней мере, без помощи фильтров и объектов записи. Но и они не предназначены для масштабного изменения документа, так что придется воспользоваться другим API, если необходимо изменить XML. DOM отвечает этим требованиям, поскольку предоставляет возможность создавать и изменять данные XML.

При работе с DOM процесс создания XML-документа немного отличается от изменения уже существующего документа, поэтому рассмотрим оба случая. В этом разделе приведен довольно реалистичный пример, который следует изучить. Если вы когда-либо посещали онлайн-аукционы, подобные eBay,[1] то знаете, что самые важные аспекты аукциона – возможность найти товары и информацию о них. Реализация этой возможности зависит как от пользователя, вводящего описание товара, так и от аукциона, передающего эту информацию. На хороших сайтах пользователи могут ввести некоторую информацию, а также HTML-описания. Это означает, что сообразительный пользователь может выделить что-то жирным шрифтом, курсивом, добавить ссылки или иное форматирование в описание товара. Вот и удобный случай применить DOM.

Создание нового дерева DOM

Для начала нам понадобится фундамент. В примере 6.1 приведена простая HTML-форма, принимающая основную информацию о товаре, которая становится доступной на сайте аукциона. Очевидно, что для реального сайта ее нужно было бы несколько «украсить», но смысл, полагаю, ясен.

Пример 6.1. HTML-форма для ввода информации о товаре

```
<html>
 <head><title>Ввод/обновление списка товаров</title></head>
 <body>
  <h1 align="center"> Ввод/обновление списка товаров </h1>
  <p align="center">
   <form method="POST" action="/javaxml2/servlet/
javaxml2.UpdateItemServlet">
    ID товара (Уникальный идентификатор): <br />
    <input name="id" type="text" maxLength="10" /><br /><br />
    Название товара: <br />
```

[1] Русскоязычным читателям, скорее, знаком онлайн-аукцион «Молоток» – *http://www.molotok.ru*. – *Примеч. науч. ред.*

```
    <input name="name" type="text" maxLength="50" /><br /><br />
    Описание товара: <br />
    <textarea name="description" rows="10" cols="30" wrap="wrap" ></textarea>
    <br /><br />
    <input type="reset" value="Сбросить" />  
    <input type="submit" value="Добавить/обновить" />
   </form>
  </p>
 </body>
</html>
```

Обратите внимание на то, что целью формы является сервлет. Код сервлета приведен в примере 6.2. Метод `doPost()` считывает исходные параметры и сохраняет их значения во временных переменных. Затем сервлет ищет в файловой системе конкретный файл, содержащий указанную информацию.

Внимание

В интересах ясности изложения в этом сервлете я работаю с файловой системой напрямую. Как правило, это не лучший способ. Рассмотрите применение интерфейса `ServletContext` для получения доступа к локальным ресурсам; такой вариант позволит легко распространять и изменять сервлет в зависимости от сервера и среды исполнения сервлетов. Такого рода детали не добавляют ясности примерам, потому я не буду ничего усложнять.

Если файл не существует (а для нового товара так и будет), будет создано новое дерево DOM и построена древовидная структура с использованием переданных значений. После этого сервлет применяет класс `DOMSerializer` (из главы 5) для вывода дерева DOM в файл, делая его доступным при следующем вызове сервлета. Кроме того, добавлен код для метода `doGet()`. Этот метод просто отображает HTML-код из примера 6.1. Это пригодится позже, при реализации возможности изменения. Пока же не обращайте на него внимания.

Пример 6.2. Класс UpdateItemServlet

```
package javaxml2;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// Импортируем DOM
import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Element;
```

```
import org.w3c.dom.Text;

// Импортируем анализатор
import org.apache.xerces.dom.DOMImplementationImpl;

public class UpdateItemServlet extends HttpServlet {

    private static final String ITEMS_DIRECTORY = "/javaxml2/ch06/xml/";

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Получаем вывод
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");

        // Выводим HTML
        out.println("<html>");
        out.println(" <head><title>Ввод/обновление списка товаров
                    </title></head>");
        out.println(" <body>");
        out.println("  <h1 align='center'> Ввод/обновление списка товаров
                     </h1>");
        out.println("  <p align='center'>");
        out.println("   <form method='POST' " +
            "action='/javaxml2/servlet/javaxml2.UpdateItemServlet'>");
        out.println("    ID товара (Уникальный идентификатор): <br />");
        out.println("    <input name='id' type='text' maxLength='10' />" +
            "<br /><br />");
        out.println("    Имя товара: <br />");
        out.println("    <input name='name' type='text' maxLength='50' />" +
            "<br /><br />");
        out.println("    Описание товара: <br />");
        out.println("    <textarea name='description' rows='10' cols='30' " +
            "wrap='wrap' ></textarea><br /><br />");
        out.println("    <input type='reset' value='Reset Form'
                     />  ");
        out.println("    <input type='submit' value='Add/Update Item' />");
        out.println("   </form>");
        out.println("  </p>");
        out.println(" </body>");
        out.println("</html>");

        out.close();
    }

    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Получаем значения параметров
        String id = req.getParameterValues("id")[0];
        String name = req.getParameterValues("name")[0];
        String description = req.getParameterValues("description")[0];
        // Создаем новое дерево DOM
        DOMImplementation domImpl = new DOMImplementationImpl();
```

```
        Document doc = domImpl.createDocument(null, "item", null);
        Element root = doc.getDocumentElement();

        // ID товара (в виде атрибута)
        root.setAttribute("id", id);

        // Название товара
        Element nameElement = doc.createElement("name");
        Text nameText = doc.createTextNode(name);
        nameElement.appendChild(nameText);
        root.appendChild(nameElement);

        // Описание товара
        Element descriptionElement = doc.createElement("description");
        Text descriptionText = doc.createTextNode(description);
        descriptionElement.appendChild(descriptionText);
        root.appendChild(descriptionElement);

        // Сериализуем дерево DOM
        DOMSerializer serializer = new DOMSerializer();
        serializer.serialize(doc, new File(ITEMS_DIRECTORY + "item-" + id +
           ".xml"));

        // Печать подтверждения
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");
        out.println("<HTML><BODY>Спасибо за информацию. " +
            "Ваш товар обработан.</BODY></HTML>");
        out.close();
    }

}
```

Теперь можно скомпилировать этот класс. Через минуту я рассмотрю его подробно, а пока что проверьте, правильно ли настроены параметры среды и включены ли требуемые классы.

Примечание

При компиляции класса `UpdateItemServlet` убедитесь, что класс `DOMSerializer` из предыдущей главы доступен в путях к классам. Кроме того, следует добавить его к классам в контексте среды исполнения сервлетов. Мой контекст – я пользуюсь средой Tomcat – называется *javaxml2* и находится в подкаталоге *javaxml2* каталога *webapps*. В каталоге *WEB-INF/classes* находится каталог *javaxml2* (для пакета), а в нем файлы *DOMSerializer.class* и *UpdateItemServlet.class*. Также следует убедиться, что копия jar-файла анализатора (в моем случае *xerces.jar*) находится в путях к классам среды. В Tomcat можно просто перенести копию в подкаталог *lib/*. Наконец, необходимо убедиться, что Xerces и реализация DOM Level 2 в среде исполнения сервлетов загружаются до реализации DOM Level 1, которая содержится в архиве Tomcat *parser.jar*. Сделать это можно, переименовав файл *parser.jar* в *z_parser.jar*. Более подробно об этом рассказано в главе 10, а пока доверьтесь мне и внесите это изменение. Затем перезапустите Tomcat, и все заработает.

После того как сервлет установлен, а среда исполнения сервлетов запущена, вызовите сервлет, чтобы запрос GET, сгенерированный броузером, загрузил HTML-форму для ввода данных. Заполните эту форму, как показано на рис. 6.1.

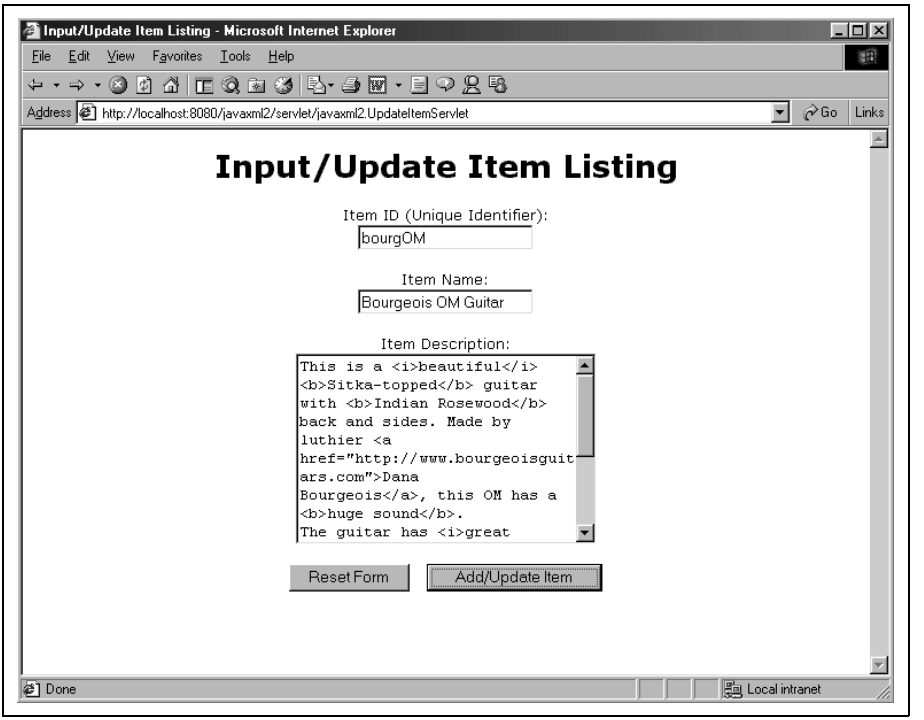

Рис. 6.1. Заполнение формы

Через какое-то время мы будем подробно говорить о поле описания, так что здесь я привожу полный текст, который набрал в этом поле. Да, в нем много разметки (я просто помешался на жирном шрифте и курсиве!), но позже это нам очень пригодится:

```
This is a <i>beautiful</i> <b>Sitka-topped</b> guitar with <b>Indian
Rosewood</b>
back and sides. Made by luthier <a href="http://
www.bourgeoisguitars.com">Dana
Bourgeois</a>, this OM has a <b>huge sound</b>.
The guitar has <i>great action</i>, a 1 3/4" nut, and all
<i>fossilized ivory</i> nut and saddle, with <i>ebony</i> end pins.
New condition, this is a <b>great guitar</b>!
[Перевод]
<i>Великолепная</i> гитара: верхняя дека из ели, спинка и боковины из
<b>индийского палисандра</b>.
Эта оркестровая гитара сделана мастером <a href="http://
www.bourgeoisguitars.com">Дейной Буржуа</a>, и звучит просто
<b>превосходно</b>.
```

```
Механика инструмента <i>замечательная</i>: верхний порожек 1 и s дюйма, оба
порожка сделаны из <i>ископаемой слоновой кости</i>, а крепления для струн из
<i>черного дерева</i>.
<b>Отличная</b> гитара, состояние идеальное.
```

При отправлении формы данные из нее передаются (способом POST) сервлету, и за дело принимается метод `doPost()`. Что касается создания данных в DOM, то тут все оказывается очень просто. Во-первых, следует создать экземпляр класса `org.w3c.dom.DOMImplementation`. Это основа всех действий по созданию дерева DOM. Разумеется, можно и напрямую реализовать экземпляр интерфейса DOM `Document`, но тогда нельзя будет создать класс `DocType`, как это можно сделать в случае `DOMImplementation`. Так что лучше остановиться на нем. Кроме того, класс `DOMImplementation` предоставляет и еще один полезный метод, `hasFeature()`. Позже мы подробно рассмотрим этот метод, поэтому сейчас не обращайте на него внимание. В коде примера используется реализация Xerces `org.apache.xerces.dom.DOMImplementationImpl` (правда ведь, загадочное имя?). В настоящее время не существует не зависящего от платформы способа достигнуть желаемого, хотя в будущем такая возможность может появиться в DOM Level 3 (который рассмотрен в конце главы). JAXP, подробно описанный в главе 9, предлагает другие решения, но к ним мы перейдем позже.

Когда создан экземпляр класса `DOMImplementation`, все уже очень просто. Давайте снова посмотрим на соответствующий код:

```
// Создаем новое дерево DOM
DOMImplementation domImpl = new DOMImplementationImpl();
Document doc = domImpl.createDocument(null, "item", null);
Element root = doc.getDocumentElement();

// ID товара (в виде атрибута)
root.setAttribute("id", id);

// Имя товара
Element nameElement = doc.createElement("name");
Text nameText = doc.createTextNode(name);
nameElement.appendChild(nameText);
root.appendChild(nameElement);

// Описание товара
Element descriptionElement = doc.createElement("description");
Text descriptionText = doc.createTextNode(description);
descriptionElement.appendChild(descriptionText);
root.appendChild(descriptionElement);

// Сериализуем дерево DOM
DOMSerializer serializer = new DOMSerializer();
serializer.serialize(doc, new File(ITEMS_DIRECTORY + "item-" + name +
    ".xml"));
```

Во-первых, для получения нового экземпляра Document используется метод createDocument(). Первый аргумент этого метода – пространство имен корневого элемента документа. Мы еще не получили пространство имен, потому опускаем его и передаем значение null. Второй аргумент – имя собственно корневого элемента, в данном случае просто "item". Последний аргумент – экземпляр класса DocType, и мы опять передаем значение null, т. к. его у этого документа нет. Если бы нам понадобился экземпляр DocType, то его можно было бы создать при помощи метода createDocType()все того же класса DOMImplementation. Те, кто заинтересовался этим методом, могут найти полное описание DOM API в приложении А.

Имея дерево DOM, с которым будем работать, можно получить корневой элемент (при помощи метода getDocumentElement(), рассмотренного в предыдущей главе). Сделав это, добавляем атрибут с идентификатором товара при помощи метода setAttribute(). Методу передается имя и значение атрибута. И вот корневой элемент готов. Дальше все еще проще: конструкцию DOM любого типа можно создать при помощи объекта Document, действующего в качестве фабрики. Для создания элементов «name» и «description» мы прибегаем к методу createElement(), аргументом в каждом случае является имя элемента. Аналогичный подход используется и при создании текстового содержимого для каждого из них. Поскольку элемент хранит содержимое не напрямую, а в дочерних узлах Text (вспомните, что об этом говорилось в предыдущей главе), то правильно будет применить метод createTextNode(). Методу передается текст для узла, который служит описанием и именем товара. Можно было бы воспользоваться методом createCDATASection() и заключить этот текст в теги CDATA, поскольку внутри элемента присутствует HTML-код. Но тогда мы не сможем прочитать содержимое как набор элементов, и получим его в виде обычного текста. Поскольку позже у нас появится необходимость работать с элементами этого текста, сделаем его обычным узлом Text, само собой, при помощи метода createTextNode(). Узлы созданы, и осталось только связать их в иерархию. Лучше всего обратиться к методу appendChild() для каждого из них, добавляя элементы к корню, а текстовое содержимое элементов к соответствующему «родителю». Здесь все вполне прозрачно. И, наконец, весь документ передается классу DOMSerializer, рассмотренному в предыдущей главе, и выводится в XML-файл на диске.

Внимание

Я предполагаю, что пользователь вводит корректный HTML-код, другими словами, XHTML. В реальном приложении, вероятно, имеет смысл обработать полученные от пользователя данные с помощью программы JTidy (*http://www.sourceforge.net/projects/jtidy*). В приводимом примере мы просто предполагаем, что введенные данные – это XHTML.

В сервлете определена константа ITEMS_DIRECTORY, позволяющая задать рабочий каталог. В коде примера используется каталог в системе Windows, и обратите внимание, что все обратные слэши экранированы. Не забывайте об этой детали! Измените значение константы, присвоив ей имя каталога, который должен выступать в качестве рабочего в вашей системе. Сгенерированный XML-код можно просмотреть, перейдя в каталог, заданный этой константой, и открыв XML-файл, который будет находиться там. Мой выглядел так, как показано в примере 6.3.

Пример 6.3. XML-файл, сгенерированный сервлетом UpdateItemServlet

```
<?xml version="1.0"?>
<item id="bourgOM">
<name>Bourgeois OM Guitar</name>
<description>This is a <i>beautiful</i> <b>Sitka-topped</b> guitar with
<b>Indian Rosewood</b> back and sides. Made by luthier
<a href="http://www.bourgeoisguitars.com">Dana Bourgeois</a>, this OM has a
<b>huge sound</b>.
The guitar has <i>great action</i>, a 1 3/4" nut, and all
<i>fossilized ivory</i> nut and saddle, with <i>ebony</i> end pins.
New condition, this is a <b>great guitar</b>!</description>
</item>[1]
```

Мы довольно быстро рассмотрели этот пример, но вы должны начать разбираться в DOM. Далее мы поговорим об изменении уже существующего дерева DOM.

Изменение дерева DOM

Процесс изменения существующего дерева DOM несколько отличается от процесса его создания. Как правило, он включает загрузку DOM из некоторого источника, обход дерева и внесение изменений. Эти изменения обычно касаются либо структуры, либо содержимого. Если изменение касается структуры, то задача сводится к созданию дерева:

```
// Добавляем элемент copyright в корень
Element root = doc.getDocumentElement();
Element copyright = doc.createElement("copyright");
copyright.appendChild(doc.createTextNode("Copyright O'Reilly 2001"));
root.appendChild(copyright);
```

Именно об этом я только что говорил. Процесс изменения существующего содержимого несколько отличается, хотя он не чрезмерно сложен. В качестве примера рассмотрим измененную версию UpdateItemServlet. Эта версия сервлета считывает переданный идентификатор и пытается загрузить существующий файл (если он существует). Если попытка успешна, то новое дерево DOM не создается, вместо этого

[1] Перевод см. выше.

изменяется существующее. Поскольку изменений много, ниже вновь приводится весь класс целиком, а изменения выделяются:

```
package javaxml2;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.xml.sax.SAXException;

// Импортируем DOM
import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;

// Импортируем анализатор
import org.apache.xerces.dom.DOMImplementationImpl;
import org.apache.xerces.parsers.DOMParser;

public class UpdateItemServlet extends HttpServlet {

    private static final String ITEMS_DIRECTORY = "/javaxml2/ch06/xml/";

    // Метод doGet() не изменился

    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        // Получаем значения параметров
        String id = req.getParameterValues("id")[0];
        String name = req.getParameterValues("name")[0];
        String description = req.getParameterValues("description")[0];

        // Проверяем, существует ли файл
        Document doc = null;
        File xmlFile = new File(ITEMS_DIRECTORY + "item-" + id + ".xml");
        String docNS = "http://www.oreilly.com/javaxml2";

        if (!xmlFile.exists()) {
            // Создаем новое дерево DOM
            DOMImplementation domImpl = new DOMImplementationImpl();
            doc = domImpl.createDocument(null, "item", null);
            Element root = doc.getDocumentElement();

            // ID товара (в виде атрибута)
            root.setAttribute("id", id);

            // Наименование товара
            Element nameElement = doc.createElement("name");
```

```
        Text nameText = doc.createTextNode(name);
        nameElement.appendChild(nameText);
        root.appendChild(nameElement);

        // Описание товара
        Element descriptionElement = doc.createElement("description");
        Text descriptionText = doc.createText(description);
        descriptionElement.appendChild(descriptionText);
        root.appendChild(descriptionElement);
    } else {
        // Загружаем документ
        try {
            DOMParser parser = new DOMParser();
            parser.parse(xmlFile.toURL().toString());
            doc = parser.getDocument();

            Element root = doc.getDocumentElement();

            // Название товара
            NodeList nameElements =
                root.getElementsByTagNameNS(docNS, "name");
            Element nameElement = (Element)nameElements.item(0);
            Text nameText = (Text)nameElement.getFirstChild();
            nameText.setData(name);

            // Описание товра
            NodeList descriptionElements =
                root.getElementsByTagNameNS(docNS, "description");
            Element descriptionElement =
            (Element)descriptionElements.item(0);

            // Удаляем и вновь создаем описание
            root.removeChild(descriptionElement);
            descriptionElement = doc.createElement("description");
            Text descriptionText = doc.createTextNode(description);
            descriptionElement.appendChild(descriptionText);
            root.appendChild(descriptionElement);
        } catch (SAXException e) {
            // Выводим сообщение об ошибке
            PrintWriter out = res.getWriter();
            res.setContentType("text/html");
            out.println("<HTML><BODY>Ошибка при чтении XML: " +
                e.getMessage() + ".</BODY></HTML>");
            out.close();
            return;
        }
    }

    // Сериализуем дерево DOM
    DOMSerializer serializer = new DOMSerializer();
    serializer.serialize(doc, xmlFile);

    // Печать подтверждения
    PrintWriter out = res.getWriter();
```

```
            res.setContentType("text/html");
            out.println("<HTML><BODY> Спасибо за информацию. " +
                "Ваш товар обработан.</BODY></HTML>");
            out.close();
        }
    }
```

Опять же, никаких сюрпризов здесь нет. Мы создали экземпляр File для указанного файла (при помощи переданного идентификатора) и проверяем, существует ли файл. Так сервлет узнает, существует ли уже XML-файл, соответствующий указанному товару. Если файл не существует, то сервлет делает все то, о чем говорилось в последнем разделе, без изменений. Если XML-файл уже существует (какая-то информация о товаре уже поступала), он загружается в дерево DOM при помощи механизмов, рассмотренных в предыдущей главе. С этого момента начинается собственно обход дерева.

Код получает корневой элемент и затем использует метод getElementsByTagName() для поиска всех элементов с именами «name» и «description». В каждом случае известно, что в списке NodeList присутствует ровно один элемент. Доступ к нему можно получить при помощи метода item() класса NodeList и посредством передачи «0» в качестве аргумента (все индексы начинаются с нуля). Это позволяет получить искомый элемент. Можно было бы просто извлечь все дочерние элементы корневого элемента при помощи метода getChildren(), а затем отбросить первые два. Однако применение имен узлов проще документировать и оно более очевидно. Вызов метода getFirstChild() позволяет получить текстовое содержимое элемента «name». Известно, что элемент «name» имеет единственный узел Text и его можно напрямую привести к нужному типу. Наконец, метод setData() позволяет изменить существующее значение на новое, т. е. поменять старую информацию на новую, которая передана пользователем через форму.

Обратите внимание, что при обработке описания товара мы выбрали несколько иной подход. Элемент описания вполне может содержать целые фрагменты документа (вспомним, что пользователь может ввести HTML-код, т. е. допустимы вложенные элементы «b», «a» и «img»), и в данном случае проще удалить существующий элемент «description» и заменить его новым. Это избавляет нас от необходимости рекурсивно обходить дерево и удалять каждый дочерний элемент, и экономит время. Узел, удаленный с помощью метода removeChild(), можно элементарно создать заново и повторно дописать к корневому элементу документа.

Код не случайно жестко связан с форматом вывода XML. На деле, в большинстве случаев код, реализующий изменение DOM-дерева, основан на понимании логики содержимого. Для случаев, когда структура или формат неизвестны, больше подходит модель обхода (Traversal) DOM Level 2. Ее мы рассмотрим несколько позже в этой главе.

Пока же согласитесь, что знание структуры формата XML является огромным преимуществом. Такое преимущество у нас есть, ранее мы сами же и создавали структуру в сервлете. Можно использовать методы, подобные `getFirstChild()`, и приводить результат к определенному типу, вместо того чтобы долго сравнивать типы и писать длинные блоки switch.

После завершения процесса создания или изменения дерева DOM оно сериализуется в XML, и процесс можно повторить. Мне пришлось добавить обработку некоторых SAX-ошибок, связанных с возникающими в процессе DOM-анализа проблемами, но в этом для читателей нет ничего нового. В качестве упражнения обновите метод `doGet()` так, чтобы он считывал параметр из URL и загружал данные XML, разрешая пользователю изменять их в форме. Например, URL *http://localhost:8080/javaxml2/servlet/javaxml2.UpdateItemServlet?id = bourgOM* говорит о том, что должен быть загружен для редактирования товар с идентификатором «bourgOM». Изменение несложное, и к этому времени вы уже должны быть готовы внести его самостоятельно.

Пространства имен

Важное нововведение DOM Level 2, о котором еще не говорилось, – это поддержка пространств имен XML в DOM. Вы помните из глав 3 и 4, что поддержка пространств имен добавлена в SAX 2.0 и то же справедливо и для второй версии DOM. Ключевыми тут являются два новых метода интерфейса `Node`: `getPrefix()` и `getNamespaceURI()`. Кроме того, все методы создания узлов имеют аналоги, поддерживающие пространства имен. Например, вызов метода `createElement()` следует заменить вызовом `createElementNS()`.

Перегружен?

Всем программистам на Java метод `createElementNS()` покажется подозрительным. Почему бы просто не перегрузить метод `createElement()` так, чтобы он принимал дополнительные параметры? Что ж, это можно было бы сделать, если бы модель DOM использовалась только в Java или в языках, поддерживающих перегрузку. Однако это не так. Спецификация не должна зависеть от возможностей языка, и в результате, когда дело доходит до изменения сигнатур существующих методов, возникают ограничения на имена методов и обратную совместимость. Так что для реализации поддержки пространств имен DOM определяет новые методы с суффиксом `NS`. Это плохо для Java, но хорошо для DOM как стандарта, не зависящего от языка.

В каждом из этих новых методов, поддерживающих пространства имен, первым аргументом является URI пространства имен, а вторым – *уточненное* (*qualified*) имя элемента, атрибута и т. д. Заметьте, я сказал «уточненное», это означает, что если требуется использовать пространство имен с URI «http://www.oreilly.com» и префикс «ora» для элемента «copyright», то следует вызвать метод `createElementNS("http://www.oreilly.com", "ora:copyright")`. Важно помнить о необходимости указывать префикс, это поможет сохранить много времени впоследствии. Вызов `getPrefix()` для этого нового элемента вернет значение «ora», как и должен. Если же требуется создать элемент в пространстве имен по умолчанию (без префикса), просто передайте имя элемента (в данном случае локальное имя), вот и все. Вызов метода `getPrefix()` для элемента из пространства имен по умолчанию возвращает значение `null`, впрочем, как и для элемента, вообще не находящегося в каком-либо пространстве имен.

Внимание

Префикс говорит очень мало относительно того, принадлежит ли элемент пространству имен. Для элементов из пространства имен по умолчанию (без префикса) значение, возвращаемое методом `getPrefix()`, совпадает со значением, возвращаемым для элементов, не принадлежащих *никакому* пространству имен. Есть надежда, что в следующей версии спецификации для элементов из пространства имен по умолчанию будет возвращаться пустая строка.

Вместо того чтобы просто перечислять все новые методы, поддерживающие пространства имен (этот список можно найти в приложении А), я лучше приведу реальный код. В качестве примера применения пространств имен замечательно подойдет метод `doPost()` из сервлета `UpdateItemServlet`:

```
public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

    // Получаем значения параметров
    String id = req.getParameterValues("id")[0];
    String name = req.getParameterValues("name")[0];
    String description = req.getParameterValues("description")[0];

    // Проверяем, существует ли файл
    Document doc = null;
    File xmlFile = new File(ITEMS_DIRECTORY + "item-" + id + ".xml");
    String docNS = "http://www.oreilly.com/javaxml2";

    if (!xmlFile.exists()) {
        // Создаем новое дерево DOM
        DOMImplementation domImpl = new DOMImplementationImpl();
        doc = domImpl.createDocument(docNS, "item", null);
        Element root = doc.getDocumentElement();
```

```
        // ID товара (в виде атрибута)
        root.setAttribute("id", id);

        // Название товара
        Element nameElement = doc.createElementNS(docNS, "name");
        Text nameText = doc.createTextNode(name);
        nameElement.appendChild(nameText);
        root.appendChild(nameElement);

        // Описание товара
        Element descriptionElement =
                 doc.createElementNS(docNS, "description");
        Text descriptionText = doc.createText(description);
        descriptionElement.appendChild(descriptionText);
        root.appendChild(descriptionElement);
    } else {
        // Загружаем документ
        try {
            DOMParser parser = new DOMParser();
            parser.parse(xmlFile.toURL().toString());
            doc = parser.getDocument();

            Element root = doc.getDocumentElement();

            // Название товара
            NodeList nameElements =
                root.getElementsByTagNameNS(docNS, "name");
            Element nameElement = (Element)nameElements.item(0);
            Text nameText = (Text)nameElement.getFirstChild();
            nameText.setData(name);

            // Описание товара
            NodeList descriptionElements =
                root.getElementsByTagNameNS(docNS, "description");
            Element descriptionElement =
            (Element)descriptionElements.item(0);

            // Удаляем и вновь создаем описание
            root.removeChild(descriptionElement);
            descriptionElement = doc.createElementNS(docNS,
            "description");
            Text descriptionText = doc.createTextNode(description);
            descriptionElement.appendChild(descriptionText);
            root.appendChild(descriptionElement);
        } catch (SAXException e) {
            // Выводим сообщение об ошибке
            PrintWriter out = res.getWriter();
            res.setContentType("text/html");
            out.println("<HTML><BODY>Ошибка при чтении XML: " +
                e.getMessage() + ".</BODY></HTML>");
            out.close();
            return;
        }
    }
```

```
        // Сериализуем дерево DOM
        DOMSerializer serializer = new DOMSerializer();
        serializer.serialize(doc, xmlFile);

        // Печать подтверждения
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");
        out.println("<HTML><BODY>Спасибо за информацию. " +
            "Ваш товар обработан.</BODY></HTML>");
        out.close();
    }
```

Применение метода `createElementNS()` для создания элементов в пространстве имен и метода `getElementsByTagNameNS()` для их поиска кажется отличной идеей. Метод `createDocument()` даже позволяет указать URI пространства имен для корневого элемента. Элементы помещаются в пространство имен по умолчанию, и все выглядит замечательно. Однако тут есть большая проблема. Посмотрите на вывод, полученный при запуске этого сервлета для несуществующего XML-файла (это сгенерированный, а не измененный XML-код):

```
<?xml version="1.0"?>
<item id="bourgOM">
<name>Bourgeois OM Guitar</name>
<description>This is a <i>beautiful</i> <b>Sitka-topped</b> guitar with
<b>Indian Rosewood</b> back and sides. Made by luthier
<a href="http://www.bourgeoisguitars.com">Dana Bourgeois</a>, this OM has a
<b>huge sound</b>.
The guitar has <i>great action</i>, a 1 3/4" nut, and all
<i>fossilized ivory</i> nut and saddle, with <i>ebony</i> end pins.
New condition, this is a <b>great guitar</b>!</description>
</item>[1]
```

Выглядит знакомо? Это XML-код, который мы уже видели, *без каких-либо изменений*! Единственное, чего не делает DOM – так это не добавляет объявления пространств имен. Следует вручную добавлять атрибуты `xmlns` к дереву DOM. В противном случае при чтении документа элементы не будут помещены в пространство имен, и у вас возникнут некоторые проблемы. Но следующее небольшое изменение позволяет этого избежать:

```
// Создание нового дерева DOM
DOMImplementation domImpl = new DOMImplementationImpl();
doc = domImpl.createDocument(docNS, "item", null);
Element root = doc.getDocumentElement();
root.setAttribute("xmlns", docNS);
```

Теперь вы получите объявление пространства имен, которое, вероятно, ожидали увидеть еще в первый раз. Можно скомпилировать изме-

[1] Перевод см. выше.

ненный код и посмотреть, что получится. Разница незаметна: изменения сделаны так, как будто они были раньше. Однако документ теперь будет иметь пространства имен как в разделе чтения, так и в разделе записи сервлета.

И последнее, что следует сказать о пространствах имен: помните, что можно было изменить класс `DOMSerializer` таким образом, чтобы он искал пространства имен элементов и выводил соответствующие объявления `xmlns` при обходе дерева. Это вполне допустимое изменение и оно даже представляет некоторую ценность. На самом деле это то, что уже сделано во многих других решениях, подобных тем, которые можно найти в Xerces. В любом случае тот, кто знает о таком поведении, уже не станет его жертвой.

Модули DOM Level 2

Теперь, разобравшись с базовыми возможностями DOM Level 2, мы перейдем к некоторым дополнениям DOM Level 2. Речь идет о различных модулях, дополняющих функциональность ядра. В определенных приложениях DOM они время от времени оказываются полезны.

Но, во-первых, само собой, нужно иметь анализатор, поддерживающий DOM Level 2. У того, кто работает с анализатором, который он самостоятельно загрузил или купил, неприятностей быть не должно. Например, можно пойти на веб-сайт Apache XML *http://xml.apache.org*, загрузить самую последнюю версию Xerces и получить поддержку DOM Level 2. Однако если воспользоваться анализатором, входящим в состав иной платформы, все может усложниться. Например, если вы пользуетесь средой исполнения сервлетов Jakarta Tomcat, и у вас есть *xml.jar* и *parser.jar* в каталоге *lib/* и в путях к классам Tomcat. Они не очень полезны, поскольку предоставляют только DOM Level 1 и не поддерживают многие возможности, о которых пойдет речь в этом разделе; в таком случае вручную загрузите анализатор, поддерживающий DOM Level 2, и убедитесь, что он загружается *раньше* анализаторов, поддерживающих DOM Level 1.

Внимание

Остерегайтесь новых версий Tomcat. При запуске они загружают все файлы *jar* из каталога *lib/*, что якобы удобно. К сожалению, загрузка производится в алфавитном порядке, а значит, если поместить в каталог *lib/* файл *xerces.jar*, то *parser.jar*, анализатор, поддерживающий DOM Level 1, по-прежнему будет загружаться первым, и мы не получим поддержки DOM Level 2. Распространенное решение этой проблемы заключается в следующем: нужно переименовать файл *parser.jar* в *z_parser.jar*, а *xml.jar* в *z_xml.jar*. В результате они загружаются после Xerces, а мы получаем поддержку DOM Level 2. Эта проблема уже упоминалась ранее в примере сервлета.

Имея анализатор, соответствующий задаче, мы готовы приступить к работе. Но прежде чем перейти к новым модулям, посмотрим, для чего же они нужны.

Ветвление

Когда вышла спецификация DOM Level 1, она была единственной спецификацией по DOM. За некоторыми исключениями она практически полностью соответствовала тому, что сказано в главе 5. Когда началась работа над DOM Level 2, появилось множество спецификаций, каждая из которых называлась *модулем*. Если посмотреть на полный набор спецификаций DOM Level 2, можно увидеть шесть различных модулей. Кажется, что этого много, не так ли? Я не собираюсь рассматривать здесь все эти модули – о DOM вы будете читать на протяжении следующих четырех или пяти глав. Однако в табл. 6.1 коротко рассказывается о каждом модуле. Приведены спецификация модуля, имя и назначение – параметры, которые очень скоро вам понадобятся.

Таблица 6.1. Спецификации DOM и их назначение

Спецификация	Имя модуля	Назначение
DOM Level 2 Core	XML	Расширяет спецификацию DOM Level 1; посвящена основным структурам DOM, таким как `Element`, `Attr`, `Document` и т. д.
DOM Level 2 Views	Views	Обеспечивает модель динамического обновления структуры DOM сценариями
DOM Level 2 Events	Events	Определяет модель событий для программ и сценариев, используемых при работе с DOM
DOM Level 2 Style	CSS	Модель для CSS (каскадных таблиц стилей), основанная на спецификациях DOM Core и DOM Views
DOM Level 2 Traversal and Range	Traversal/ Range	Определяет DOM-расширения для обхода документа и идентификации диапазона содержимого документа
DOM Level 2 HTML	HTML	Расширяет DOM определениями интерфейсов для работы с HTML-структурами в формате DOM

Если бы модули views, events, CSS, HTML и traversal описывались в одной спецификации, то W3C не удалось бы сделать вообще ничего! Чтобы разработка спецификаций могла продолжаться, не сдерживая при этом применение модели DOM, самостоятельные концепции были выделены в самостоятельные спецификации.

Если решение о том, какие спецификации использовать, принято, то вы почти готовы к работе. Анализатор, поддерживающий DOM Level 2, не обязан поддерживать все эти спецификации; поэтому необходимо

убедиться, что желаемые возможности присутствуют в вашем анализаторе XML. К счастью, сделать это довольно просто. Помните метод hasFeature(), который мы обсуждали при рассмотрении класса DOMImplementation? Если передать ему имя модуля и версию, он сообщит, поддерживается ли запрошенная возможность или модуль. В примере 6.4. приведена маленькая программа, определяющая, поддерживает ли анализатор XML модули DOM, перечисленные в табл. 6.1. Имя класса реализации DOMImplementation может варьироваться в зависимости от системы, но в остальном эта программа будет работать для любого анализатора.

Пример 6.4. Проверка возможностей реализации DOM

```
package javaxml2;

import org.w3c.dom.DOMImplementation;

public class DOMModuleChecker {

    /** Класс частной реализации DOMImplementation */
    private String vendorImplementationClass =
        "org.apache.xerces.dom.DOMImplementationImpl";

    /** Модули, для которых производится проверка */
    private String[] moduleNames =
        {"XML", "Views", "Events", "CSS", "Traversal", "Range", "HTML"};

    public DOMModuleChecker() {
    }

    public DOMModuleChecker(String vendorImplementationClass) {
        this.vendorImplementationClass = vendorImplementationClass;
    }

    public void check() throws Exception {
        DOMImplementation impl =
            (DOMImplementation)Class.forName(vendorImplementationClass)
                                    .newInstance();
        for (int i=0; i<moduleNames.length; i++) {
            if (impl.hasFeature(moduleNames[i], "2.0")) {
                System.out.println("Поддержка модуля " + moduleNames[i] +
                    " присутствует в этой реализации DOM.");
            } else {
                System.out.println("Поддержка модуля " + moduleNames[i] +
                    " отсутствует в этой реализации DOM.");
            }
        }
    }

    public static void main(String[] args) {
        if ((args.length != 0) && (args.length != 1)) {
        System.out.println("Использование: java javaxml2.DOMModuleChecker " +
                "[Испытуемый класс реализации DOMImplementation]");
```

```
                System.exit(-1);
            }

            try {
                DOMModuleChecker checker = null;
                if (args.length == 1) {
                    checker = new DOMModuleChecker(args[1]);
                } else {
                    checker = new DOMModuleChecker();
                }
                checker.check();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Запустив эту программу (файл *xerces.jar* доступен в путях к классам), мы получим следующий вывод:

```
C:\javaxml2\build>java javaxml2.DOMModuleChecker
Поддержка модуля XML присутствует в этой реализации DOM.
Поддержка модуля Views отсутствует в этой реализации DOM.
Поддержка модуля Events присутствует в этой реализации DOM.
Поддержка модуля CSS отсутствует в этой реализации DOM.
Поддержка модуля Traversal присутствует в этой реализации DOM.
Поддержка модуля Range отсутствует в этой реализации DOM.
Поддержка модуля HTML отсутствует в этой реализации DOM.
```

Указав класс частной реализации `DOMImplementation` используемой системы, можно выяснить, какие из модулей поддерживает этот DOM-анализатор. В нескольких следующих подразделах я коснусь некоторых из этих модулей, которые показались мне полезными, и о которых вам тоже будет небезынтересно узнать.

Модуль Traversal

Первым в списке стоит модуль Traversal. Его назначение – обеспечить возможность обхода дерева, позволяя при этом изменять природу этой функциональности. В разделе, посвященном изменению деревьев DOM, говорилось, что большая часть кода, работающего с DOM, будет что-то знать о структуре дерева, которое подвергается обработке; это позволяет быстро обходить и изменять как структуру, так и содержимое. Но если структура документа заранее неизвестна, на помощь приходит модуль Traversal.

Вернемся к сайту аукциона и товарам, описания которых вводятся пользователем. Наиболее важны название и описание товара. Поскольку большинство популярных сайтов аукционов предлагают возможность поиска, нужно предоставить ее и в этом фиктивном примере.

Простой поиск по заголовкам товаров не подходит в реальной жизни. Вместо этого нужно выделить набор ключевых слов из описания товара. Я говорю «ключевые слова», потому что нам не надо, чтобы поиск по шаблону «adirondack top» (что для ценителя гитар, очевидно, обозначает дерево, из которого сделана гитара) возвращал волчки («top») с горного массива («Adirondack»). Самый лучший способ сделать это для обсуждаемого в данный момент формата – выделить слова, отформатированные определенным способом. Поэтому наиболее подходящими кандидатами становятся те слова, которые в описании выделены жирным шрифтом или курсивом. Конечно, можно было бы собрать все нетекстовые дочерние элементы элемента `description`. Но тогда бы пришлось столкнуться с изображениями (элемент `img`), ссылками (`a`) и прочими элементами. На самом деле нам требуется обход дерева в определенной манере. Хорошие новости: вы попали в нужное место.

Полностью модуль traversal содержится в пакете `org.w3c.dom.traversal`. Все в ядре DOM начинается с интерфейса `Document`, а в модуле DOM Traversal все начинается с интерфейса `org.w3c.dom.traversal.DocumentTraversal`. Этот интерфейс предоставляет два метода:

```
NodeIterator createNodeIterator(Node root, int whatToShow, NodeFilter
                                filter, boolean expandEntityReferences);
TreeWalker createTreeWalker(Node root, int whatToShow, NodeFilter filter,
                            boolean expandEntityReferences);
```

Большинство реализаций DOM, поддерживающих модуль traversal, имеют свой класс реализации `org.w3c.dom.Document`, также реализующий и интерфейс `DocumentTraversal`. Именно так это работает в Xerces. Если говорить вкратце, то использование `NodeIterator` обеспечивает списочное представление элементов, которые он обходит. Ближайшая аналогия – стандартный класс Java `List` (из пакета `java.util`). `TreeWalker` обеспечивает представление дерева, к которому вы уже могли привыкнуть, работая с XML.

Интерфейс NodeIterator

Оставим теоретические понятия и перейдем к примеру кода, на который я ссылался ранее. Нам необходим доступ ко всему содержимому из описания товара, которое отмечено определенными тегами форматирования. Для этого сначала нужно получить доступ к самому дереву DOM. Поскольку это не укладывается в подход с использованием сервлетов (вероятно, составлять фразы для поиска будет не сервлет, а некоторый автономный класс), нам понадобится новый класс `ItemSearcher` (пример 6.5). Этот класс позволяет задать любое количество файлов, в которых производится поиск.

Пример 6.5. Класс ItemSearcher

```
package javaxml2;

import java.io.File;
```

```
// Импортируем DOM
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.traversal.DocumentTraversal;
import org.w3c.dom.traversal.NodeFilter;
import org.w3c.dom.traversal.NodeIterator;

// Реализация анализатора
import org.apache.xerces.parsers.DOMParser;

public class ItemSearcher {

    private String docNS = "http://www.oreilly.com/javaxml2";

    public void search(String filename) throws Exception {
        // Анализ данных и создание дерева DOM
        File file = new File(filename);
        DOMParser parser = new DOMParser();
        parser.parse(file.toURL().toString());
        Document doc = parser.getDocument();

        // Получаем узел, с которого начинается итерация
        Element root = doc.getDocumentElement();
        NodeList descriptionElements =
            root.getElementsByTagNameNS(docNS, "description");
        Element description = (Element)descriptionElements.item(0);

        // Получаем NodeIterator
        NodeIterator i = ((DocumentTraversal)doc)
          .createNodeIterator(description, NodeFilter.SHOW_ALL, null, true);

        Node n;
        while ((n = i.nextNode()) != null) {
            if (n.getNodeType() == Node.ELEMENT_NODE) {
                System.out.println("Найден элемент: '" +
                    n.getNodeName() + "'");
            } else if (n.getNodeType() == Node.TEXT_NODE) {
                System.out.println("Найден текст: '" +
                    n.getNodeValue() + "'");
            }
        }
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Файлы для поиска не заданы.");
            return;
        }

        try {
            ItemSearcher searcher = new ItemSearcher();
            for (int i=0; i<args.length; i++) {
```

```
                System.out.println("Обрабатываю файл: " + args[i]);
                searcher.search(args[i]);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Как видите, мы создали объект NodeIterator и передали ему элемент description, с которого начинается итерация. Константа, переданная в качестве фильтра, предписывает показывать все узлы. Также можно было бы задать значения Node.SHOW_ELEMENT и Node.SHOW_TEXT, показывающие только элементы или текстовые узлы, соответственно. Мы пока не рассмотрели реализацию NodeFilter (это будет сделано на следующем этапе), и я разрешил расширение ссылок на сущности. Замечательно то, что, будучи один раз созданным, итератор обходит не только дочерние узлы элемента description. На самом деле он обходит все узлы из элемента description, даже если они находятся на нескольких уровнях вложенности. Это чрезвычайно удобно при работе с данными XML, структура которых заранее не определена.

Сейчас мы по-прежнему выбираем все узлы, а это не совсем то, чего нам хотелось. Добавим немного кода (последний цикл while), чтобы показать, как печатать текстовые узлы и узлы элементов. Выполнение кода в таком виде не принесет особой пользы. Имеет смысл организовать фильтр, чтобы программа выбирала только элементы с требуемым форматом: текст из блоков i или b. Такое настраиваемое поведение можно обеспечить с помощью реализации интерфейса NodeFilter, в котором определен лишь один метод:

```
public short acceptNode(Node n);
```

Этот метод должен возвращать одну из констант: NodeFilter.FILTER_SKIP, NodeFilter.FILTER_REJECT или NodeFilter.FILTER_ACCEPT. Первая предписывает пропустить узел, но продолжает обходить его дочерние элементы. Вторая предписывает исключить из рассмотрения узел и его дочерние элементы (доступно только в TreeWalker), а третья – принять узел и передать его далее. Такое поведение очень похоже на SAX, когда можно было перехватывать узлы и решать, нужно ли передавать их вызывающему методу. Добавьте в исходный файл ItemSearcher.java следующий закрытый класс:

```
class FormattingNodeFilter implements NodeFilter {

    public short acceptNode(Node n) {
        if (n.getNodeType() == Node.TEXT_NODE) {
            Node parent = n.getParentNode();
            if ((parent.getNodeName().equalsIgnoreCase("b")) ||
                (parent.getNodeName().equalsIgnoreCase("i"))) {
```

```
                return FILTER_ACCEPT;
            }
        }
        // Если мы оказались здесь, узел не представляет интереса
        return FILTER_SKIP;
    }
}
```

Этот старый добрый код DOM не должен показаться вам сложным. Во-первых, данный код выбирает только текстовые узлы; вернее, текст из отформатированных элементов, а не сами элементы. Затем определяется родительский элемент и, поскольку вполне можно предположить, что родительскими для узлов `Text` являются узлы `Element`, то код сразу же вызывает метод `getNodeName()`. Если имя элемента – «b» или «i», значит, найден текст, и код возвращает `FILTER_ACCEPT`. В противном случае возвращается `FILTER_SKIP`.

Теперь осталось только изменить вызов, создающий итератор, чтобы сообщить ему о необходимости использовать новую реализацию фильтра, а также код вывода. Оба фрагмента принадлежат уже существующему методу `search()` класса `ItemSearcher`:

```
// Получаем NodeIterator
NodeIterator i = ((DocumentTraversal)doc)
    .createNodeIterator(description, NodeFilter.SHOW_ALL,
        new FormattingNodeFilter(), true);

Node n;
while ((n = i.nextNode()) != null) {
    System.out.println("Найдена фраза: '" + n.getNodeValue() + "'");
}
```

Примечание

Внимательные читатели, возможно, заинтересуются, что произойдет, если реализация `NodeFilter` конфликтует с константой, переданной методу `createNodeIterator()` (в нашем случае `NodeFilter.SHOW_ALL`). На самом деле сначала применяется фильтр, заданный константой, а затем полученный список узлов передается реализации фильтра. Если бы я задал константу `NodeFilter.SHOW_ELEMENT`, я бы не получил никаких фраз, поскольку мой фильтр не получил бы ни одного узла `Text`, только узлы `Element`. Будьте внимательны и используйте фильтры осмысленным образом. В этом примере можно было бы также использовать `NodeFilter.SHOW_TEXT`.

Теперь класс несет полезную нагрузку и готов к применению. Запустив его для файла *bourgOM.xml*, о котором говорилось в первом разделе, мы получим следующие результаты:

```
bmclaugh@GANDALF ~/javaxml2/build
$ java javaxml2.ItemSearcher ../ch06/xml/item-bourgOM.xml
Обрабатываю файл: ../ch06/xml/item-bourgOM.xml
Найдена фраза: 'beautiful'
```

```
Найдена фраза: 'Sitka-topped'
Найдена фраза: 'Indian Rosewood'
Найдена фраза: 'huge sound'
Найдена фраза: 'great action'
Найдена фраза: 'fossilized ivory'
Найдена фраза: 'ebony'
Найдена фраза: 'great guitar'
```

Все отлично: фразы, выделенные жирным шрифтом и курсивом, теперь можно проиндексировать в поисковой системе. (Извините, но ее вам придется писать самостоятельно!)

Интерфейс TreeWalker

Интерфейс `TreeWalker` практически полностью совпадает с интерфейсом `NodeIterator` за одним исключением – результаты получаются в виде дерева, а не списка. Это особенно полезно, если хочется иметь дело лишь с узлами определенного типа; например, с деревом, включающим только элементы или не имеющим комментариев. Используя константу для фильтра (такую как `NodeFilter.SHOW_ELEMENT`) и реализацию фильтра (скажем, такую, которая передает для всех комментариев значение `FILTER_SKIP`), можно получить представление дерева DOM, исключив из него постороннюю информацию. Интерфейс `TreeWalker` предоставляет все основные операции для работы с узлами, такие как методы `firstChild()`, `parentNode()`, `nextSibling()` и, конечно, `getCurrentNode()`, позволяющий определить, на каком узле вы находитесь в данный момент.

Не буду приводить здесь примеры. Думаю, всем понятно, что работа с этим интерфейсом мало чем отличается от работы со стандартным деревом DOM; за тем исключением, что можно отфильтровывать нежелательные элементы при помощи констант `NodeFilter`. Это отличный, простой способ ограничить представление XML-документов только интересующей вас информацией. Пользуйтесь этой возможностью – это настоящая находка, как и `NodeIterator`! Также можно обратиться к полной спецификации на сайте *http://www.w3.org/TR/DOM-Level-2-Traversal-Range/*.

Модуль Range

Модуль Range из DOM Level 2 используется реже других, и причина этого кроется скорее в недостатке понимания модуля, а не в невозможности его практического применения. Этот модуль позволяет работать с наборами фрагментов содержимого документа. Когда диапазон содержимого определен, его можно копировать, изменять и манипулировать другими доступными способами. Для начала очень важно понять, что «диапазон» (range) в данном случае означает некоторое количество фрагментов дерева DOM, сгруппированных по определенному признаку. Здесь нет связи со множеством допустимых значений,

когда определены границы диапазона. Поэтому модуль DOM Range не имеет ничего общего с проверкой корректности значений данных. Если вы понимаете это, то уже опережаете других.

При работе с Range, как и при работе с Traversal, требуется новый пакет DOM: org.w3c.dom.ranges. Данный класс определяет лишь два интерфейса и одно исключение, так что вам не потребуется много времени, чтобы сориентироваться. Первый интерфейс — org.w3c.dom.ranges.DocumentRange представляет собой аналог интерфейса Document (и DocumentTraversal): подобно DocumentTraversal, класс Document в Xerces реализует Range. И так же, как и DocumentTraversal, он имеет совсем немного интересных методов, по сути, всего один:

```
public Range createRange();
```

Все остальные операции с диапазонами производятся над классом Range (вернее, над реализацией интерфейса). Создав экземпляр интерфейса Range, можно установить начальную и конечную точку и вычеркнуть все остальное. Вернемся к сервлету UpdateItemServlet. Я уже говорил, что довольно сложно удалить все дочерние элементы элемента description, а затем установить новое описание. Дело в том, что не существует способа определить, сколько и каких элементов и текстовых узлов, а также вложенных узлов содержится в описании. Было показано, как удалить старый элемент description и создать новый. Однако с модулем DOM Range это уже не нужно. Давайте посмотрим на измененный метод doPost() этого сервлета:

```
// Загружаем документ
try {
    DOMParser parser = new DOMParser();
    parser.parse(xmlFile.toURL().toString());
    doc = parser.getDocument();

    Element root = doc.getDocumentElement();

    // Название товара
    NodeList nameElements =
        root.getElementsByTagNameNS(docNS, "name");
    Element nameElement = (Element)nameElements.item(0);
    Text nameText = (Text)nameElement.getFirstChild();
    nameText.setData(name);

    // Описание товара
    NodeList descriptionElements =
        root.getElementsByTagNameNS(docNS, "description");
    Element descriptionElement =
    (Element)descriptionElements.item(0);

    // Удаляем и вновь создаем описание
    Range range = ((DocumentRange)doc).createRange();
    range.setStartBefore(descriptionElement.getFirstChild());
    range.setEndAfter(descriptionElement.getLastChild());
```

```
        range.deleteContents();
        Text descriptionText = doc.createTextNode(description);
        descriptionElement.appendChild(descriptionText);

        range.detach();
    } catch (SAXException e) {
        // Выводим сообщение об ошибке
        PrintWriter out = res.getWriter();
        res.setContentType("text/html");
        out.println("<HTML><BODY>Ошибка при чтении XML: " +
            e.getMessage() + ".</BODY></HTML>");
        out.close();
        return;
    }
```

Чтобы удалить все содержимое, мы сначала создали новый диапазон `Range` при помощи приведения `DocumentRange`. Также потребуется импортировать в сервлет классы `DocumentRange` и `Range` (оба принадлежат пакету `org.w3c.dom.ranges`).

Примечание

В первой части раздела, посвященного модулям DOM Level 2, было показано, как проверить, какие модули поддерживаются реализацией анализатора. При этом Xerces сообщил, что не поддерживает Range. Однако в версиях Xerces 1.3.0, 1.3.1 и 1.4 этот код работал безукоризненно. Странно, не правда ли?

Подготовив диапазон, необходимо установить начальную и конечную точки. Поскольку нам нужно все содержимое элемента `description`, то начинаем мы с первого дочернего элемента этого узла `Element` (при помощи метода `setStartBefore()`), а заканчиваем за его последним дочерним элементом (при помощи метода `setEndAfter()`). Существуют также и другие, подобные методы для выполнения этой задачи: `setStartAfter()` и `setEndBefore()`. Сделав это, можно очень просто вызвать метод `deleteContents()`. При этом не остается никакого содержимого. Затем сервлет создает новое текстовое описание и добавляет его. Наконец, мы даем знать JVM, что можно освободить все ресурсы, связанные с `Range`, вызвав метод `detach()`. Хотя об этом шаге часто забывают, он может здорово помочь при работе с длинными фрагментами кода, занимающими лишние ресурсы.

Другая возможность – воспользоваться `extractContents()` вместо `deleteContents()`. Этот метод удаляет содержимое, а затем возвращает удаленное содержимое. Например, можно добавить это в качестве архивного элемента:

```
// Удаляем и вновь создаем описание
Range range = ((DocumentRange)doc).createRange();
range.setStartBefore(descriptionElement.getFirstChild());
range.setEndAfter(descriptionElement.getLastChild());
```

```
Node oldContents = range.extractContents();
Text descriptionText = doc.createTextNode(description);
descriptionElement.appendChild(descriptionText);

// Устанавливаем в качестве содержимого для некоторого другого, архивного
   элемента
archivalElement.appendChild(oldContents);
```

Не пытайтесь воспроизвести это в своем сервлете: в коде нет никакого элемента `archivalElement`, и пример приведен лишь в демонстрационных целях. Однако вы должны начать понимать, что модуль Range из DOM Level 2 действительно в состоянии помочь при редактировании содержимого документа. Также он предоставляет дополнительный способ получить дескриптор для содержимого, когда его структура заранее не определена.

О диапазонах в DOM можно сказать еще много; и заинтересованные читатели могут самостоятельно исследовать эту тему, как и другие модули DOM, рассмотренные в этой главе. Но даже сейчас у вас должно быть достаточно знаний, чтобы двигаться дальше. Важнее всего понять, что в любой точке активного диапазона `Range` можно просто вызвать метод `range.insertNode(Node newNode)` и добавить новое содержимое, находясь при этом где угодно внутри документа! Привлекательными диапазоны делает именно такая возможность надежного редактирования. Когда в следующий раз вам понадобится удалить, скопировать, выделить или добавить содержимое к структуре, о которой известно не много, подумайте о диапазонах. На сайте *http://www.w3.org/TR/DOM-Level-2-Traversal-Range/* доступна спецификация, в которой можно найти информацию о том, что уже сказано, и о многом другом.

Модули Events, Views и Style

Помимо модуля HTML, о котором я расскажу позже, существуют еще три модуля DOM Level 2: Events, Views и Style. Мы не будем подробно рассматривать их в этой книге – по большей части из-за того, что они более полезны при создании клиентских приложений. До сих пор мы уделяли основное внимание программированию на стороне сервера, и в оставшейся части книги в этом смысле ничего не изменится. Эти три модуля чаще всего используются в программном обеспечении клиента, таком как IDE, веб-страницы и прочее. Но все же мы кратко коснемся каждого из них, чтобы вы оказались в числе первых на следующей вечеринке фанатов DOM.

Модуль Events

Модуль Events обеспечивает именно то, что от него, вероятно, ожидают: способ «прослушивания» документа DOM. Все связанные с ним классы находятся в пакете `org.w3c.dom.events`, а всю работу обеспечи-

вает интерфейс DocumentEvent. Нет ничего удивительного в том, что совместимые анализаторы (как Xerces) реализуют этот интерфейс в том же классе, который реализует интерфейс org.w3c.dom.Document. Данный интерфейс определяет только один метод:

```
public Event createEvent(String eventType);
```

Ему передается строка, соответствующая типу события. Допустимые значения в DOM Level 2 – «UIEvent», «MutationEvent» и «MouseEvent». Каждое из них имеет соответствующий класс: UIEvent, MutationEvent и MouseEvent. Как видно из Javadoc-документации Xerces, этот анализатор предоставляет лишь интерфейс MutationEvent, являющийся единственным типом события, поддерживаемым Xerces. Когда событие возникает, его может обработать (или «перехватить») EventListener.

Именно тут на помощь приходит ядро DOM – анализатор, поддерживающий события DOM, должен иметь интерфейс org.w3c.dom.Node, реализующий интерфейс org.w3c.dom.events.EventTarget. Так что каждый узел может быть целью события. Это означает, что мы имеем следующий метод для этих узлов:

```
public void addEventListener(String type, EventListener listener,
                             boolean capture);
```

Вот как выглядит весь процесс. Вы создаете новую реализацию EventListener (это пользовательский класс, который вы напишете). И реализовать необходимо лишь один метод:

```
public void handleEvent(Event event);
```

Зарегистрируйте его для всех узлов, с которыми хотите работать. Обычно код метода выполняет полезную работу: например, уведомляет пользователя по электронной почте о том, что его данные в XML-файле изменились, повторно проверяет действительность XML (в случае XML-редакторов) либо уточняет, уверен ли пользователь, что хочет выполнить данное действие.

Вместе с тем, вы наверняка захотите, чтобы код инициировал новое событие Event при выполнении некоторых действий, например, когда пользователь выбирает узел в IDE и вводит новый текст либо удаляет выбранный элемент. Когда событие инициировано, оно передается доступным экземплярам EventListener, начиная с активного узла и далее. Именно здесь и выполняется написанный вами код, *если совпадают типы событий*. Кроме того, в этот момент можно остановить дальнейшее распространение информации о событии (после того как оно обработано) либо начать цепь событий и, вероятно, обработать их другими регистраторами (registered listeners).

Ну, вот и все, все события рассмотрены на одной странице! А вы думали, что спецификацию трудно читать. Если серьезно, то это очень полезные сведения, а если создается код, работающий на стороне кли-

ента, либо программное обеспечение, которое будет работать автономно на компьютере пользователя (как, например, XML-редактор, о котором я не перестаю вспоминать), то этот модуль должен стать частью вашего набора инструментов для работы с DOM. Полностью спецификацию можно найти на сайте *http://www.w3.org/TR/DOM-Level-2-Events/*.

Модуль Views

Следующий по списку – модуль Views. Причина, по которой я не рассматриваю этот модуль в подробностях, заключается в том, что на самом деле о нем можно сказать очень мало. После прочтения спецификации (занимающей одну страницу!) я решил, что это просто основа для дальнейшей работы, возможно, на вертикальных рынках. Спецификация определяет лишь два интерфейса, оба в пакете `org.w3c.dom.views`. Вот первый из них:

```
package org.w3c.dom.views;

public interface AbstractView {
    public DocumentView getDocument();

}
```

А вот второй:

```
package org.w3c.dom.views;

public interface DocumentView {
    public AbstractView getDefaultView();

}
```

Не правда ли, выглядит как зацикливание? С одним исходным документом (деревом DOM) может быть связано несколько представлений (views). В данном случае это может быть документ со стилями (после применения XSL или CSS) либо версия документа с содержимым Shockwave, а другая без. Реализуя интерфейс `AbstractView`, можно определить собственные варианты отображения дерева DOM. Например, представьте себе следующий интерфейс:

```
package javaxml2;

import org.w3c.dom.views.AbstractView;

public interface StyledView extends AbstractView {

    public void setStylesheet(String stylesheetURI);

    public String getStylesheetURI();
}
```

Реализации методов опущены, но и без них можно понять, как воспользоваться этим интерфейсом, чтобы получить представление

дерева, к которому применены стили. Кроме того, в совместимом анализаторе должна быть реализация org.w3c.dom.Document, предоставляющая интерфейс DocumentView и позволяющая получать от документа информацию о стандартном для него представлении. Ожидается, что в следующей версии спецификации можно будет регистрировать несколько представлений для документа и более тесно привязывать представления к документу.

Считайте, что это будет сделано во многом подобно тому, как броузеры Netscape, Mozilla и Internet Explorer реализуют такие представления для XML. Кроме того, короткая спецификация приведена на странице *http://www.w3.org/TR/DOM-Level-2-Views/*. Прочитав ее, вы будете знать все то, что знаю я.

Модуль Style

Наконец, существует модуль Style, также называемый просто CSS (Cascading Style Sheets, каскадные таблицы стилей). Эта спецификация доступна на сайте *http://www.w3.org/TR/DOM-Level-2-Style/*. Она описывает представление каскадных таблиц стилей в виде конструкций DOM. Все, что представляет интерес, находится в пакетах org.w3c.dom.stylesheets и org.w3c.dom.css. Первый из них содержит общие основные классы, а второй предоставляет специфические приложения для каскадных таблиц стилей. Оба используются в основном для того, чтобы отобразить для клиента документ с примененными стилями.

Этот модуль применяется точно так же, как и базовые интерфейсы DOM: мы получаем анализатор, совместимый со Style, анализируем таблицу стилей и применяем реализацию для CSS. Особенно это полезно, если требуется проанализировать таблицу стилей CSS и применить ее к документу DOM. Набор базовых концепций не изменился, если вам это о чем-либо говорит (должно говорить – когда можно сделать две вещи при помощи API, а не одну, обычно это хорошо!). Опять же, мы лишь кратко коснулись модуля Style, т. к. полная информация доступна в документации Javadoc. Классы названы удачно (CSSValueList, Rect, CSSDOMImplementation) и довольно близки к своим аналогам из XML, при работе с которыми, я уверен, у вас не будет проблем.

HTML

DOM предоставляет набор интерфейсов для HTML, моделирующих различные HTML-элементы. Например, можно использовать классы HTMLDocument, HTMLAnchorElement и HTMLSelectElement (все из пакета org.w3c.dom.html) для представления их аналогов в HTML (<HTML>, <A> и <SELECT> в данном случае). Все они предоставляют удобные методы, например, setTitle() (для HTMLDocument), setHref() (для HTMLAnchorElement) и getOptions() (для HTMLSelectElement). Все они расширяют базовые

структуры DOM, такие как Document и Element, и поэтому их можно применять наравне с другими узлами Node.

Однако оказывается, что частные реализации модуля HTML применяются редко (по крайней мере, непосредственно). Дело не в том, что они бесполезны, нет, просто уже написано множество инструментов, предоставляющих подобный тип доступа при помощи еще более дружественных к пользователю интерфейсов. XMLC, созданный в рамках сервера приложений Enhydra, – один из таких примеров (можно найти на сайте *http://xmlc.enhydra.org*), а Cocoon, рассматриваемый в главе 10, – второй. Они позволяют разработчику иметь дело с языком HTML и веб-страницами таким образом, который не требует даже базовых познаний в DOM, делая его более доступным для веб-дизайнеров и новичков, программирующих на Java. Как следствие, модуль HTML в DOM редко бывает нужен (но все-таки бывает). Кроме того, возможно использование стандартных средств DOM для корректных HTML-документов (XHTML), если считать элементы узлами Element, а атрибуты – узлами Attr. Даже без модуля HTML можно применять DOM для работы с HTML. Очень просто.

Всякая всячина

Что еще осталось в DOM Level 2 помимо этих модулей и поддержки пространств имен? Осталось очень мало, и вы, вероятно, уже имели дело с большей частью нерассмотренных возможностей. Методы createDocument() и createDocumentType(), новые для класса DOMImplementation, мы уже применяли. Кроме того, методы getSystemId() и getPublicId(), используемые в классе DOMSerializer интерфейса DocumentType, также являются дополнениями из DOM Level 2. Помимо этого осталось не много: несколько новых кодов ошибок DOMException, вот, пожалуй, и все. Полный список изменений можно найти на сайте *http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/changes.html*. Все остальные изменения – это дополнительные модули, один из которых рассмотрен далее.

DOM Level 3

Перед тем как завершить разговор о DOM и посмотреть на распространенные ловушки, задержимся немного и посмотрим, что ожидается в DOM Level 3, который находится в стадии разработки. На самом деле ожидается, что эта спецификация выйдет в начале 2002 года,[1] совсем вскоре после того, как вы, вероятно, прочтете эту книгу. Поэтому здесь рассмотрены не все изменения и дополнения из DOM Level 3, но я считаю, что они представляют особый интерес для большинства разработ-

[1] Обновление DOM Level 3 датируется 14 января 2002 года. – *Примеч. ред.*

чиков DOM (теперь и вы в их числе, если хотите знать). Многие изменения – это то, что в течение нескольких лет просили добавить программисты DOM.

Объявление XML

Первое изменение в DOM, на которое я хочу обратить внимание, на первый взгляд кажется тривиальным: вид объявления XML. Помните, как оно выглядит? Вот пример:

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

В настоящее время в DOM не доступны три важных фрагмента информации, которая содержится в объявлении: версия, состояние атрибута `standalone` и указанная кодировка. Кроме того, само дерево DOM имеет кодировку, которая может совпадать, а может и не совпадать со значением атрибута `encoding`. Например, кодировка «UTF-8» в Java оказывается кодировкой «UTF8», и должен существовать способ различать их. Все эти проблемы в DOM Level 3 решаются добавлением четырех атрибутов к интерфейсу `Document`. Вот эти атрибуты: `version` (тип `String`), `standalone` (тип `boolean`), `encoding` (`String`) и `actualEncoding` (опять же, `String`). Методы для работы с этими атрибутами довольно прозрачны:

```
public String getVersion();
public void setVersion(String version);

public boolean getStandalone();
public void setStandalone(boolean standalone);

public String getEncoding();
public void setEncoding(String encoding);

public String getActualEncoding();
public void setActualEncoding(String actualEncoding);
```

Важнее всего то, что наконец-то можно получить доступ к информации из объявления XML. Это настоящий подарок для тех, кто пишет XML-редакторы и нечто подобное, для чего нужна эта информация. Также это помогает разработчикам, занимающимся языковой поддержкой в XML, поскольку они могут определить кодировку документа (`encoding`), создать дерево DOM в этой кодировке (`actualEncoding`), а затем перевести его соответственно потребностям.

Сравнение узлов

В DOM Level 1 и Level 2 сравнить два узла можно было только вручную. В итоге разработчики писали вспомогательные классы, использующие `instanceof` для определения типа узла, а затем сравнивали значения, возвращаемые всеми доступными методами. Другими словами, это были мучения. DOM Level 3 предлагает несколько методов для

сравнения, облегчающих эти страдания. Я представлю здесь предложенные сигнатуры и расскажу о каждой. Все они – дополнения к интерфейсу org.w3c.dom.Node и выглядят следующим образом:

```
// Проверяем, является ли входной узел тем же объектом, что и этот узел
public boolean isSameNode(Node input);

// Проверка равенства структуры (не объектов)
public boolean equalsNode(Node input, boolean deep);

/** Константы порядка документов */
public static final int DOCUMENT_ORDER_PRECEDING = 1;
public static final int DOCUMENT_ORDER_FOLLOWING = 2;
public static final int DOCUMENT_ORDER_SAME      = 3;
public static final int DOCUMENT_ORDER_UNORDERED = 4;

// Определяем порядок входного документа относительно указанного узла
public int compareDocumentOrder(Node input) throws DOMException;

/** Константы для указателей в дереве */
public static final int TREE_POSITION_PRECEDING  = 1;
public static final int TREE_POSITION_FOLLOWING  = 2;
public static final int TREE_POSITION_ANCESTOR   = 3;
public static final int TREE_POSITION_DESCENDANT = 4;
public static final int TREE_POSITION_SAME       = 5;
public static final int TREE_POSITION_UNORDERED  = 6;

// Определяем положение в дереве входного узла относительно указанного узла
public int compareTreePosition(Node input) throws DOMException;
```

Первый метод, isSameNode(), позволяет сравнивать объекты. Он не определяет, имеют ли два узла одинаковую структуру или данные, но проверяет, являются ли они одним и тем же объектом в JVM. Второй метод, equalsNode(), вероятно, будет чаще применяться в ваших приложениях. Он проверяет равенство узлов на основе данных и типа (очевидно, что Attr никогда не будет равен DocumentType). Он имеет параметр deep, позволяющий сравнивать только сам узел или все его дочерние узлы.

Следующие два метода, compareDocumentOrder() и compareTreePosition(), учитывают относительное положение текущего и входного узлов. Для обоих методов определено несколько констант, используемых в качестве возвращаемых значений. Узел может находиться перед текущим, после него, в той же позиции или может быть неупорядоченным. Значение «неупорядоченный» возникает при сравнении атрибута с элементом или в любом другом случае, когда понятие «порядок в документе» не имеет контекстного смысла. И наконец, возникает исключение DOMException, если два узла, подвергаемых сравнению, принадлежат разным объектам DOM Document. Последний новый метод, compareTreePosition(), обеспечивает сравнение подобного рода, но добавляет возможность определения «предков». Для этого служат две дополнительные константы – TREE_POSITION_ANCESTOR и TREE_POSITION_DES-

CENDANT. Первая указывает на то, что входной узел расположен в иерархии выше контекстного узла (узла, метод которого вызывается); вторая – что входной узел находится ниже контекстного.

Имея эти четыре метода, можно выделить любую структуру DOM и определить, как она соотносится с другой. Это дополнение из DOM Level 3 призвано сослужить разработчикам добрую службу, и они могут рассчитывать на использование этих методов сравнения в своих программах. Следите за именами и значениями констант, т. к. они могут измениться в процессе работы над спецификацией.

Самозагрузка

Последнее дополнение в DOM Level 3, которое мы рассмотрим, пожалуй, самое важное – это возможность самозагрузки. Ранее уже говорилось, что при создании структур DOM приходится использовать код, специфичный для платформы (если только мы не пользуемся JAXP, о котором речь пойдет в главе 9). Разумеется, это плохо, поскольку исключает независимость от платформы. В интересах обсуждения повторим фрагмент кода, создающий объект DOM Document при помощи DOMImplementation:

```
import org.w3c.dom.Document;
import org.w3c.dom.DOMImplementation;

import org.apache.xerces.dom.DOMImplementationImpl;

// Объявление класса и другие конструкции Java

DOMImplementation domImpl = DOMImplementationImpl.getDOMImplementation();
Document doc = domImpl.createDocument();
// И т. д. ...
```

Проблема заключается в том, что не существует способа получить DOMImplementation без импортирования и применения класса реализации из конкретной системы. Решение – применить фабрику, создающую экземпляры DOMImplementation. Разумеется, фабрика фактически предоставляет реализацию DOMImplementation конкретной платформы (знаю, знаю, это несколько запутанно). Производители систем могут устанавливать системные свойства или предоставлять собственные версии этой фабрики, чтобы она возвращала нужный класс реализации. В результате код для создания дерева DOM будет выглядеть так:

```
import org.w3c.dom.Document;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.DOMImplementationFactory;

// Объявление класса и другие конструкции Java

DOMImplementation domImpl =
    DOMImplementationFactory.getDOMImplementation();
Document doc = domImpl.createDocument();
// И т.д. ...
```

Тут добавлен класс DOMImplementationFactory, и если он применяется, он должен решить большинство проблем, связанных с независимостью от платформы. Считайте его флагманом DOM Level 3, поскольку это одна из наиболее полезных возможностей для текущих уровней DOM.

Советы разработчикам

Так же как и SAX, да и так же, как и другие API, которые мы рассмотрим в следующих нескольких главах, DOM имеет собственный набор сложностей. Некоторые из них представлены ниже и, надеюсь, благодаря этому вам удастся спасти несколько часов, которые были бы потрачены на отладку. Пользуйтесь, это те проблемы, с которыми я сталкивался и с которыми боролся некоторое время, прежде чем понять, что же происходит.

Страшное исключение WRONG DOCUMENT

Известную мне напасть номер один для разработчиков DOM я называю «страшным исключением WRONG DOCUMENT». Это исключение возникает при попытке смешать узлы из различных документов. Чаще всего это случается при переносе узла из одного документа в другой, что в действительности представляет собой распространенную задачу.

«Неувязка» эта возникает из-за «фабричного» подхода, о котором говорилось выше. Поскольку каждый элемент, атрибут, инструкция обработки и прочее создается из экземпляра Document, небезопасно думать, что эти узлы совместимы с другими экземплярами Document. Два экземпляра Document могут иметь различные платформы и поддерживать различные возможности, так что попытка смешать и сопоставить узлы из одного с узлами из другого может приводить к осложнениям, зависящим от применяемых реализаций. В итоге, для того чтобы использовать узел из другого документа, нужно передать этот узел методу insertNode() целевого документа. В результате выполнения метода будет создан новый узел, совместимый с целевым документом. Другими словами, приводимый ниже код станет источником неприятностей:

```
Element otherDocElement = otherDoc.getDocumentElement();
Element thisDocElement = thisDoc.getDocumentElement();

// Тут возникает проблема - смешение узлов из различных документов
thisDocElement.appendChild(otherDocElement);
```

В результате возникнет исключение:

```
org.apache.xerces.dom.DOMExceptionImpl: DOM005 Wrong document
 at org.apache.xerces.dom.ChildAndParentNode.internalInsertBefore(
     ChildAndParentNode.java:314)
 at org.apache.xerces.dom.ChildAndParentNode.insertBefore(
     ChildAndParentNode.java:296)
 at org.apache.xerces.dom.NodeImpl.appendChild(NodeImpl.java:213)
 at MoveNode.main(MoveNode.java:30)
```

Чтобы избежать этого, сначала надо импортировать желаемый узел в новый документ:

```
Element otherDocElement = otherDoc.getDocumentElement();
Element thisDocElement = thisDoc.getDocumentElement();

// Импортируем узел в нужный документ
Element readyToUseElement = (Element)thisDoc.importNode(otherDocElement);

// Теперь это работает
thisDocElement.appendChild(readyToUseElement);
```

Обратите внимание, что результатом работы метода `importNode()` является узел, так что его нужно привести к нужному интерфейсу (в данном случае это `Element`). Сэкономьте время и усилия, зафиксируйте этот факт в памяти; напишите это на листочке и спрячьте под подушкой. Поверьте мне, это практически самое досадное исключение, известное людям!

Создание, добавление и вставка

Исправление только что описанной ошибки зачастую приводит к другой. Распространенный промах, известный мне, заключается в том, что разработчик помнит о необходимости импортировать узел, но затем забывает добавить его! Другими словами, код выглядит так:

```
Element otherDocElement = otherDoc.getDocumentElement();
Element thisDocElement = thisDoc.getDocumentElement();

// Импортируем узел в нужный документ
Element readyToUseElement = (Element)thisDoc.importNode(otherDocElement);

// Узел не добавляется!
```

В данном случае мы имеем элемент, принадлежащий целевому документу, но не добавленный в этот документ. В итоге появляется ошибка – документ владеет элементом, но элемент содержится в дереве DOM; причем, чтобы обнаружить эту ошибку, придется потрудиться. После всего этого в конечном дереве нет и следов импортированного узла, что может разочаровать довольно ощутимо. Будьте внимательны!

Что дальше?

Что ж, читатель должен начинать чувствовать, что уже освоился с этой штукой. В следующей главе курс по API будет продолжен, мы рассмотрим JDOM – еще один API для доступа к XML из Java. JDOM похож на DOM (но это не DOM) тем, что XML-данные моделируются в виде дерева. Мы увидим, как работает этот интерфейс прикладного программировать, узнаем, когда его использовать, и рассмотрим различия между разными API XML, которые мы уже изучили. Не стоит слишком гордиться достигнутым – предстоит еще многое узнать!

- *Основы*
- *Класс PropsToXml*
- *Класс XMLProperties*
- *Является ли JDOM стандартом?*
- *Советы разработчикам*
- *Что дальше?*

7

JDOM

Объектная модель документа в Java (JDOM) обеспечивает способ доступа к XML-документу из Java через древовидную структуру, и в этом отношении JDOM представляет собой нечто похожее на DOM. Однако она создавалась специально для Java (помните обсуждение частных реализаций для различных языков в случае DOM?), и поэтому во многих отношениях более наглядна для разработчиков на Java, чем DOM. Данная глава посвящена этим аспектам JDOM, а также причинам, по которым надо применять SAX, DOM или JDOM. Подробную информацию о JDOM можно найти на сайте *http://www.jdom.org*.

Кроме того, и это важно, JDOM – это API с открытым исходным кодом. И поскольку работа над версией 1.0 API все еще ведется, он также остается гибким.[1] Вы можете сами предлагать или реализовывать изменения. Если JDOM вам понравится, за исключением какой-нибудь досадной мелочи, то вы можете оказать нам содействие в поиске решений, позволяющих от нее избавиться. В этой главе рассказывается о текущем состоянии JDOM (при этом особое внимание уделяется стандартизации) и об основах использования API, а также приведено несколько работающих примеров.

Основы

В главах 5 и 6 содержится достаточно информации о том, как работать с древовидным представлением данных XML. Поэтому, говоря, что

[1] Поскольку версия 1.0 JDOM не является окончательной, кое-что из того, что описано в книге, может измениться к тому времени, когда вы загрузите JDOM. Я пытаюсь поддерживать список изменений на веб-сайте JDOM (*http://www.jdom.org*) и работаю с O'Reilly, чтобы как можно быстрее сделать эти изменения и обновления доступными.

JDOM также обеспечивает древовидное представление XML-документа, я даю как бы точку отсчета для понимания особенностей поведения JDOM. Чтобы вам было проще сопоставлять классы JDOM и структуры XML, взгляните на рис. 7.1, представляющий UML-модель базовых классов JDOM.

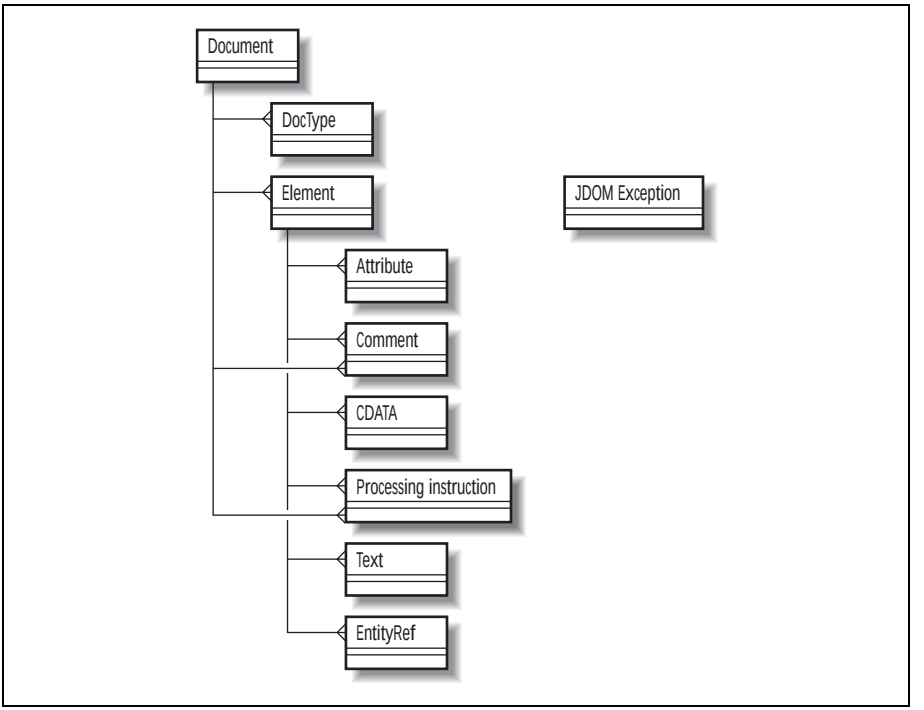

Рис. 7.1. UML-модель основных классов JDOM

Как видите, имена классов о многом говорят. Объект `Document` – ядро структуры JDOM, он является как представлением XML-документа, так и контейнером для всех остальных структур JDOM. `Element` соответствует элементу XML, `Attribute` – атрибуту, и так далее по списку. Если вы внимательно изучили DOM, то можете подумать, что в JDOM кое-что пропущено. Например, где класс `Text`? Если вы помните, DOM следует очень строгой древовидной модели, и содержимое элемента представлено дочерним узлом (или узлами) элемента. В JDOM это показалось неудобным во многих случаях, и в API поэтому существует метод `getText()` класса `Element`. Это позволяет получить содержимое элемента из самого элемента, и класс `Text` оказывается ненужным. Нам казалось, что это будет в большей степени привычно для тех разработчиков на Java, которые не успели познакомиться с XML, DOM или некоторыми причудливыми особенностями деревьев.

Чистосердечное признание

Должен признаться, что являюсь одним из создателей JDOM; мой «сообщник» в этом деле – Джейсон Хантер (Jason Hunter), знаменитый автор книги «Java Servlet Programming» (Программирование сервлетов Java), O'Reilly. У нас с Джейсоном были некоторые разногласия относительно DOM, и после долгой дискуссии на конференции O'Reilly Enterprise Java Conference в 2000 году мы придумали JDOM. Также я многим обязан Джеймсу Дэвидсону (James Davidson) (работает в Sun Microsystems, является создателем спецификации сервлетов версии 2.2, автором Ant и т. д.) и Пьеру Фумагалли (Pier Fumagalli) (супергерой проектов Apache/Jakarta/Cocoon). А также сотням корреспондентов из списков рассылки JDOM.

Все это нужно для того, чтобы предупредить, что я неравнодушен к JDOM. Так что если в этой главе вы почувствуете налет фаворитизма, то приношу свои извинения. Я часто пользуюсь SAX, DOM и JDOM, но так получилось, что JDOM мне нравится больше остальных, поскольку это мое собственное детище, и он очень помог мне во многих случаях. В любом случае, я вас предупредил!

Поддержка Java Collections

Еще один важный момент, на который нужно обратить внимание, заключается в отсутствии классов списков, таких как классы SAX `Attributes` или `NodeList` и `NamedNodeMap`. Это поклон в сторону разработчиков на Java. Мы сочли, что применение коллекций Java (`java.util.List`, `java.util.Map` и т. д.) позволит реализовать более удобный и простой API для работы с XML. Модель DOM должна применяться в различных языках программирования (вспомните про реализации для Java, о которых шла речь в главе 5) и не может пользоваться преимуществами особенностей языка, такими как коллекции Java. Например, при вызове метода `getAttributes()` класса `Element` мы получаем список `List`; с ним можно работать так же, как и с любым другим списком в Java, не изучая для этого новые методы или синтаксис.

Конкретные классы и фабрики

Еще один, менее заметный догмат JDOM, в котором проявляется отличие от DOM, заключается в том, что JDOM – это API конкретных классов. Другими словами, `Element`, `Attribute`, `ProcessingInstruction`, `Comment` и остальные – это все классы, экземпляры которых можно создавать непосредственно при помощи ключевого слова `new`. Преимущество тут

заключается в том, что не нужны фабрики, т. к. они зачастую неудобны в коде. Создать новый документ JDOM можно так:

```
Element rootElement = new Element("root");
Document document = new Document(rootElement);
```

Проще некуда. Но с другой стороны, отсутствие фабрик можно рассматривать как недостаток. В то время как можно создавать подклассы классов JDOM, в коде их придется использовать явно:

```
element.addContent(new FooterElement("Copyright 2001"));
```

В данном случае `FooterElement` — это подкласс `org.jdom.Element`, и он выполняет некоторую обработку (например, он мог бы создавать несколько элементов, отражающих нижний колонтитул страницы). Поскольку наследует класс `Element`, он может быть добавлен в элемент обычным способом при помощи метода `addContent()`. Однако не существует способа определить подкласс элемента и указать, что он всегда должен применяться для создания экземпляра элемента:

```
// Этот код не работает!!
JDOMFactory factory = new JDOMFactory();
factory.setDocumentClass("javaxml2.BrettsDocumentClass");
factory.setElementClass("javaxml2.BrettsElementClass");

Element rootElement = JDOMFactory.createElement("root");
Document document = JDOMFactory.createDocument(rootElement);
```

Идея заключается в том, что когда фабрика создана, можно указать, что определенные подклассы структур JDOM следует использовать в качестве основных классов для этих структур. Затем, каждый раз при создании экземпляра `Element` (например) с помощью фабрики, будет использоваться класс `javaxml2.BrettsElementClass` вместо класса `org.jdom.Element` (по умолчанию).

Поддержка этой возможности растет, хотя она и не является стандартным способом работы в JDOM. В мире открытого программного обеспечения эта возможность вполне может быть добавлена к тому моменту, когда вы будете читать книгу, или к тому времени, когда выйдет окончательный вариант версии 1.0 JDOM. Следите за информацией о разработке на сайте *http://www.jdom.org*.

Ввод и вывод

Последний важный аспект JDOM — это его модель ввода и вывода. Во-первых, следует понимать, что JDOM не является анализатором, а лишь реализует представление документов XML в Java. Другими словами, подобно DOM и SAX, это просто набор классов, которые можно использовать для работы с данными, предоставляемыми анализатором. Таким образом, в чтении необработанных XML-данных JDOM

полагается на анализатор.[1] Также он может принимать в качестве ввода события SAX или дерево DOM, а также экземпляры JDBC `ResultSet` и многое другое. JDOM предоставляет пакет `org.jdom.input`, предназначенный специально для организации ввода. Этот пакет предоставляет классы создания (builder classes); чаще всего вы будете пользоваться двумя из них – `SAXBuilder` и `DOMBuilder`. Данные классы создают основную структуру JDOM – документ JDOM `Document` на основе набора событий SAX или дерева DOM. В соответствии со стандартом JDOM (см. раздел «Является ли JDOM стандартом?» в конце этой главы) ожидается, что прямая поддержка JDOM появится в анализаторах Apache Xerces и Sun Crimson.

Для работы с потоками ввода, файлами или документами на диске, а также для создания документа JDOM из существующего XML-кода, не хранимого в дереве DOM, больше всего подходит класс `SAXBuilder`. Он быстр и эффективен – так же, как и SAX. Использовать его очень просто:

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(new FileInputStream("contents.xml"));
```

Подробнее об этом рассказывается позже, при рассмотрении кода примеров, но и сейчас видно, что получение доступа к XML-коду не составляет труда. Если же документ уже представлен в виде структуры DOM, то нужно воспользоваться классом `DOMBuilder`, реализующим быстрый переход из одного API в другой:

```
DOMBuilder builder = new DOMBuilder();
Document doc = builder.build(myDomDocumentObject);
```

Все довольно очевидно. По существу, тут происходит преобразование из `org.w3c.dom.Document` в `org.jdom.Document`. Обратное преобразование (документа JDOM в одну из таких структур) практически такое же. Для выполнения этой задачи служит пакет `org.jdom.output`. Для перехода от структуры JDOM к DOM применяется класс `DOMOutputter`:

```
DOMOutputter outputter = new DOMOutputter();
org.w3c.dom.Document domDoc = outputter.output(myJDOMDocumentObject);
```

Создание событий SAX на основе JDOM `Document` работает таким же образом:

```
SAXOutputter outputter = new SAXOutputter();
outputter.setContentHandler(myContentHandler);
outputter.setErrorHandler(myErrorHandler);
outputter.output(myJDOMDocumentObject);
```

[1] По умолчанию это анализатор Xerces, поставляемый вместе с JDOM. Тем не менее, с JDOM можно использовать любой другой XML-анализатор.

Здесь все аналогично работе с обычными событиями SAX после регистрации обработчиков содержимого, ошибок и других. Для этих обработчиков генерируются события SAX на основе объекта JDOM `Document`, переданного методу `output()`.

Последний класс, предназначенный для вывода, – это класс `org.jdom.output.XMLOutputter`, с которым вы, пожалуй, будете работать больше, чем с остальными. Он выводит XML-код в поток или иной класс записи, скрывающий сетевое соединение, файл или иную структуру, в которую необходимо записать XML-код. Данный класс представляет собой готовую к широкому применению версию класса `DOMSerializer` (который мы рассматривали в главе 5). Различие лишь в том, что эта версия (разумеется) работает с JDOM, а не DOM. Класс `XMLOutputter` используется следующим образом:

```
XMLOutputter outputter = new XMLOutputter();
outputter.output(jdomDocumentObject, new FileOutputStream("results.xml"));
```

Ну вот мы и рассмотрели ввод и вывод JDOM всего в нескольких абзацах. И последнее, что хочется отметить: как видно из рис. 7.2, очень просто все «зациклить», поскольку ввод и вывод в JDOM входят в API. Другими словами, можно использовать в качестве источника данных файл, работать с ним в JDOM, вывести его в SAX, DOM или файл, а затем начать цикл сначала и опять использовать файл в качестве исходных данных. Особенно это удобно в приложениях, реализующих обмен сообщениями, либо в случаях, когда JDOM играет роль промежуточного звена между другими компонентами, поставляющими или использующими XML-код.

Конечно же, это не полный обзор JDOM, но этой информации должно быть достаточно для начала, и в любом случае я хотел бы показать все в контексте работающего кода! Так что давайте рассмотрим полезную программу, которая может преобразовывать файлы свойств Java в XML.

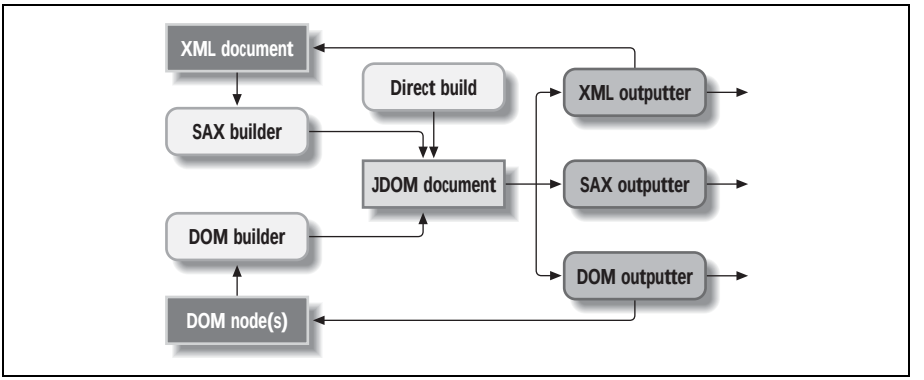

Рис. 7.2. Циклы ввода и вывода в JDOM

Класс PropsToXml

В целях изучения JDOM перейдем к реальному коду. Позвольте представить класс PropsToXML. Этот класс – утилита, преобразующая стандартный файл свойств Java в его эквивалент в формате XML. Многим разработчикам был необходим способ решения этой задачи – зачастую это позволяет избежать ручного преобразования файлов настройки при модернизации старых приложений для работы с XML.

Файлы свойств Java

Для тех, кто никогда не работал с файлами свойств Java, скажем, что это файлы, содержащие пары имя–значение, которые можно легко читать при помощи некоторых классов Java (например, java.util.Properties). Часто эти файлы выглядят подобно файлу из примера 7.1. Мы будем пользоваться этим примером на протяжении оставшейся части главы. Кстати, это файл сервера приложений Enhydra.

Пример 7.1. Типичный файл свойств Java

```
#
# Свойства, добавляемые к системным
#

# Класс реализации анализатора sax
org.xml.sax.parser="org.apache.xerces.parsers.SAXParser"

#
# Свойства, используемые для запуска сервера
#

# Класс, используемый для запуска сервера
org.enhydra.initialclass=org.enhydra.multiServer.bootstrap.Bootstrap

# Первоначальные аргументы, передаваемые серверу (заменяют аргументы
командной строки)
org.enhydra.initialargs="./bootstrap.conf"

# Classpath для загрузчика классов enhydra
org.enhydra.classpath="."

# Разделитель для classpath
org.enhydra.classpath.separator=":"
```

Ничего особенного, верно? Что ж, применяя класс Properties, можно загрузить эти свойства в объект (при помощи метода load(InputStream inputStream)), а затем работать с ними как с хеш-таблицей Hashtable. На самом деле класс Properties расширяет класс Hashtable в Java; здорово? Проблема заключается в том, что многие разделяют имена точками, чтобы сформировать некое подобие иерархической структуры. В данном примере мы будем иметь верхний уровень (сам файл свойств),

затем узел org, в нем узлы xml и enhydra, а в узле enhydra несколько узлов, некоторые из них со значениями. Другими словами, мы ожидаем увидеть структуру, подобную представленной на рис. 7.3.

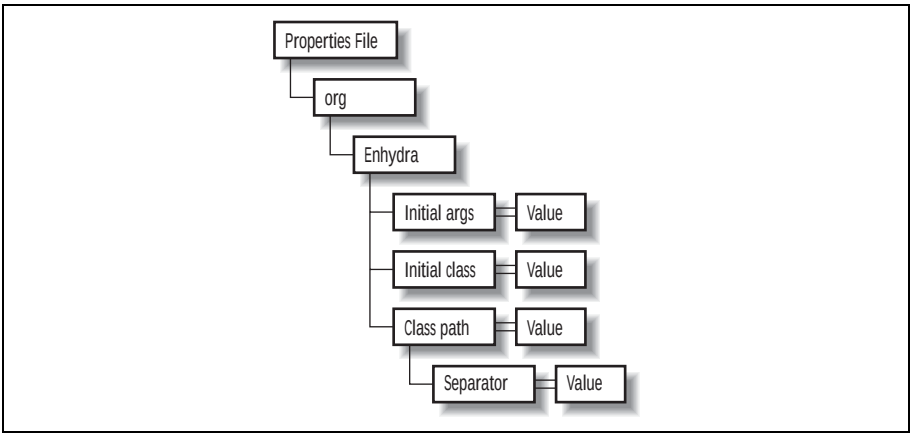

Рис. 7.3. Предполагаемая структура файла свойств из примера 7.1

Звучит здорово, однако Java не предоставляет способа для доступа к парам имя–значение подобным образом – Java не придает точке особого значения, а просто считает ее еще одним символом. Так что если нижеследующее сделать можно:

```
String classpathValue = Properties.getProperty("org.enhydra.classpath");
```

то вот это уже нельзя:

```
List enhydraProperties = Properties.getProperties("org.enhydra");
```

Можно ожидать (по крайней мере, я бы ожидал!), что второй пример работает и предоставляет все подсвойства структуры org.enhydra (org.enhydra.classpath, org.enhydra.initialargs и т. д.). К сожалению, такое поведение в классе Properties не реализовано. Поэтому многие разработчики вынуждены писать собственные дополнительные методы, работающие с этим объектом, что, разумеется, не является стандартом и доставляет неудобства. Не было бы лучше, если бы эту информацию можно было моделировать в XML, где операции, подобные второму примеру, легко выполнимы? Именно для решения этой задачи я и хочу написать код, и я воспользуюсь JDOM, чтобы продемонстрировать работу этого API.

Преобразование в XML

Как и в предыдущих главах, проще всего начать с каркаса класса, а потом добавлять в него код. В классе PropsToXML я хочу разрешить передавать в качестве исходных данных файл свойств, а также имя

файла для вывода данных XML. Класс считывает файл свойств, преобразует его в XML-документ при помощи JDOM и выводит документ в файл с указанным именем. Что ж, начнем с примера 7.2.

Пример 7.2. Каркас класса PropsToXml

```
package javaxml2;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Properties;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.XMLOutputter;

public class PropsToXML {

    /**
     * <p> Класс будет принимать переданный файл свойств и
     *   преобразовывать его в XML-документ, который затем
     *   выводится в указанный XML-файл. </p>
     *
     * @param propertiesFilename читаемый файл свойств.
     * @param xmlFilename файл, в который выводится XML-представление.
     * @throws <code>IOException</code> - при возникновении ошибки.
     */
    public void convert(String propertiesFilename, String xmlFilename)
        throws IOException {

        // Получаем объект Java Properties
        FileInputStream input = new FileInputStream(propertiesFilename);
        Properties props = new Properties();
        props.load(input);

        // Преобразуем в XML
        convertToXML(props, xmlFilename);
    }

    /**
     * <p> Позаботится о деталях преобразования из объекта Java
     *  <code>Properties</code> в XML-документ. </p>
     *
     * @param props <code>Properties</code> исходный объект.
     * @param xmlFilename файл, в который выводится XML.
     * @throws <code>IOException</code> - при возникновении ошибки.
     */
    private void convertToXML(Properties props, String xmlFilename)
        throws IOException {

        // Код для преобразования JDOM
    }
```

```
    /**
     * <p> Обеспечивает статическую точку входа для запуска. </p>
     */
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Использование: java javaxml2.PropsToXML " +
                "[файл свойств] [XML-файл для вывода]");
            System.exit(0);
        }

        try {
            PropsToXML propsToXML = new PropsToXML();
            propsToXML.convert(args[0], args[1]);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Единственная новая часть кода – это объект `Properties`, о котором уже вскользь упоминалось. Переданный файл свойств используется в методе `load()`, и этот объект делегируется методу, который будет использовать JDOM и который мы рассмотрим позже.

Создание XML при помощи JDOM

Когда свойства становятся доступны в коде в более удобной форме, приходит время обратиться к JDOM. Первым делом надо создать JDOM `Document`. Для этого следует создать корневой элемент документа при помощи класса JDOM `Element`. Поскольку XML-документ не может существовать без корневого элемента, то в качестве ввода для конструктора класса `Document` требуется экземпляр класса `Element`.

Для создания элемента `Element` требуется лишь передать его имя. Существуют и альтернативные версии, принимающие информацию о пространстве имен; о них мы поговорим чуть позже. Пока же проще всего воспользоваться именем коневого элемента, а т. к. это должно быть произвольное имя (содержащее все свойства) верхнего уровня, в коде мы остановимся на «properties». Будучи создан, этот элемент применяется для создания нового документа JDOM.

Затем он применяется для работы со свойствами из переданного файла. Список имен свойств в виде перечисления `Enumeration` мы получаем при помощи метода `propertyNames()` объекта `Properties`. Когда имя свойства доступно, по нему можно получить значение свойства посредством метода `getProperty()`. К этому моменту мы уже имеем корневой элемент нового XML-документа, имя добавляемого свойства и значение этого свойства. Затем, как и в любой другой хорошей программе, следует перебрать в цикле все остальные свойства. На каждом шаге информация передается новому методу `createXMLRepresentation()`. Он

выполняет преобразование одного свойства в набор элементов XML. Добавьте приведенный ниже код в свой исходный файл:

```
private void convertToXML(Properties props, String xmlFilename)
    throws IOException {

    // Создаем новый документ JDOM с корневым элементом «properties»
    Element root = new Element("properties");
    Document doc = new Document(root);

    // Получаем имена свойств
    Enumeration propertyNames = props.propertyNames();
    while (propertyNames.hasMoreElements()) {
        String propertyName = (String)propertyNames.nextElement();
        String propertyValue = props.getProperty(propertyName);
        createXMLRepresentation(root, propertyName, propertyValue);
    }

    // Выводим документ в указанный файл
    XMLOutputter outputter = new XMLOutputter("  ", true);
    FileOutputStream output = new FileOutputStream(xmlFilename);
    outputter.output(doc, output);
}
```

Пока рано волноваться из-за последних нескольких строк, отвечающих за вывод документа JDOM. С ними мы будем иметь дело в следующем разделе, а пока что рассмотрим метод `createXMLRepresentation()`, содержащий логику для работы со свойством и создания XML-представления.

Самый простой (и логически первый) шаг при переходе от свойства к XML – создание элемента (`Element`) с именем свойства. Мы уже знаем, как это сделать: нужно просто передать имя элемента конструктору. После создания элемента необходимо сделать значение свойства текстовым содержимым элемента. Это можно сделать довольно просто при помощи метода `setText()`, принимающего, конечно же, строку. Когда элемент готов к использованию, его можно добавить как дочерний элемент к корневому элементу при помощи метода `addContent()`. На самом деле любая допустимая конструкция JDOM может быть передана методу `addContent()` элемента, т. к. он перегружен и может принимать различные типы, в том числе `Entity`, `Comment`, `ProcessingInstruction` и многие другие. Но к ним мы перейдем позже, пока же добавим в наш исходный файл следующий метод:

```
/**
 * <p> Этот метод преобразует свойство и его значение
 * в элемент XML и текстовые данные. </p>
 *
 * @param root корневой элемент JDOM <code>Element</code>,
 * к которому добавляются дочерние элементы
 * @param propertyName имя, на котором строится создание элемента.
```

```
     * @param propertyValue значение  свойства.
     */
    private void createXMLRepresentation(Element root,
                                         String propertyName,
                                         String propertyValue) {

        Element element = new Element(propertyName);
        element.setText(propertyValue);
        root.addContent(element);
    }
```

Сейчас уже можно скомпилировать исходный файл, а затем использовать полученный класс `PropsToXML`. Укажите файл свойств (его можно набрать вручную либо загрузить файл *enhydra.properties*, приведенный ранее в этой главе) и имя выходного файла, как показано ниже:[1]

```
/javaxml2/build $ java javaxml2.PropsToXML \
                  /javaxml2/ch07/properties/enhydra.properties \
                  enhydraProps.xml
```

Проработав долю секунды, программа создаст файл *enhydraProps.xml*. Откройте его, он должен выглядеть примерно так, как в примере 7.3.[2]

Пример 7.3. Первая версия документа enhydraProps.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<properties>
  <org.enhydra.classpath.separator>":"</org.enhydra.classpath.separator>
  <org.enhydra.initialargs>"./bootstrap.conf"</org.enhydra.initialargs>
  <org.enhydra.initialclass>org.enhydra.multiServer.bootstrap.Bootstrap
</org.enhydra.initialclass>
  <org.enhydra.classpath>"."</org.enhydra.classpath>
  <org.xml.sax.parser>"org.apache.xerces.parsers.SAXParser"
</org.xml.sax.parser>
</properties>
```

Примерно в 50 строках кода мы совершили переход от свойств Java к XML. Однако этот XML-документ не намного лучше, чем файл свойств – по-прежнему нет способа связать свойство `org.enhydra.initialArgs` со свойством `org.enhydra.classpath`. Значит, наша работа еще не окончена.

---

[1] Для тех, кто не знаком с операционными системами *NIX, сообщаем, что обратный слэш в конце каждой строки просто позволяет продолжать команду на следующей строке. Пользователи Windows должны набирать всю команду целиком на одной строке.

[2] Учтите, что строки в примере продолжаются на следующей строке только в издательских целях. В документе каждое свойство с открывающим тегом, текстом и закрывающим тегом будет находиться на одной строке.

Вместо того чтобы использовать в качестве элемента имя свойства, код должен получить имя свойства и разбить его по точкам. Для каждого из этих подэлементов должны создаваться элементы и добавляться к стеку. Затем процесс может повториться. Для имени свойства org.xml.sax должна получиться следующая структура XML:

```
<org>
  <xml>
    <sax>[Значение свойства]</sax>
  </xml>
</org>
```

Применение на каждом шаге конструктора Element и метода addContent() решает эту задачу. А когда имя будет «разобрано» полностью, настанет черед метода setText(). С его помощь мы установим текстовое значение последнего элемента иерархии. Лучше всего создать новый элемент под названием current и использовать его в качестве «указателя» (в Java нет никаких указателей – это просто термин). Он всегда будет указывать на элемент, к которому должно быть добавлено содержимое. Кроме того, на каждом шаге необходимо проверять, существует ли добавляемый элемент. Например, первое свойство org.xml.sax создает элемент org. Когда добавляется новое свойство (org.enhydra.classpath), элемент org уже не нужно создавать.

Метод getChild() упрощает нам жизнь. Этот метод принимает имя искомого дочернего элемента и доступен всем экземплярам класса Element. Если указанный дочерний элемент существует, то этот элемент возвращается. Однако если дочерний элемент не существует, то возвращается значение null, которое и использует наш код. Другими словами, если возвращаемое значение является элементом, то он становится элементом current и новый элемент не добавляется (он уже существует). Но если метод getChild() возвращает значение null, то должен быть создан новый элемент из контекстной составляющей имени, он должен быть добавлен в качестве содержимого в элемент current, а указатель current в этом случае переместится вниз по дереву. Наконец, когда итерация завершится, можно будет добавить текстовое значение свойства в качестве листа, который оказывается (удачно) элементом, на который ссылается указатель current. Добавим в наш исходный файл следующий код:

```
private void createXMLRepresentation(Element root,
                                     String propertyName,
                                     String propertyValue) {

    /*
    Element element = new Element(propertyName);
    element.setText(propertyValue);
    root.addContent(element);
    */
```

```
        int split;
        String name = propertyName;
        Element current = root;
        Element test = null;

        while ((split = name.indexOf(".")) != -1) {
            String subName = name.substring(0, split);
            name = name.substring(split+1);

            // Проверка существования элемента
            if ((test = current.getChild(subName)) == null) {
                Element subElement = new Element(subName);
                current.addContent(subElement);
                current = subElement;
            } else {
                current = test;
            }
        }

        // Мы за пределами цикла, значит, осталось имя последнего элемента
        Element last = new Element(name);
        last.setText(propertyValue);
        current.addContent(last);
    }
```

Внеся эти изменения, перекомпилируйте программу и снова ее запустите. На этот раз вывод будет выглядеть гораздо аккуратнее, как это видно из примера 7.4.

Пример 7.4. Обновленный вывод PropsToXml

```
<?xml version="1.0" encoding="UTF-8"?>
<properties>
  <org>
    <enhydra>
      <classpath>
        <separator>":"</separator>
      </classpath>
      <initialargs>"./bootstrap.conf"</initialargs>
      <initialclass>org.enhydra.multiServer.bootstrap.Bootstrap</
initialclass>
      <classpath>"."</classpath>
    </enhydra>
    <xml>
      <sax>
        <parser>"org.apache.xerces.parsers.SAXParser"</parser>
      </sax>
    </xml>
  </org>
</properties>
```

Мы едва начали работать с моделью JDOM, но уже с ней освоились. Однако следует обратить внимание, что XML-документ нарушает одно из правил архитектуры документов, представленных в главе 2 (в разделе, рассказывающем об использовании элементов вместо атрибутов). Мы видим, что каждое свойство имеет единственное текстовое значение. Следовательно, можно утверждать, что значения свойств подобает сделать атрибутами последнего элемента в иерархии, а не его содержимым. Правила созданы для того, чтобы их нарушать, и я предпочитаю, чтобы в данном случае значения были представлены содержимым; но это совсем некстати.

Исключительно в наглядных целях рассмотрим преобразование значений свойств в атрибуты, а не в текстовое содержимое. Это оказывается довольно простой задачей, и ее можно решить двумя способами. Первый способ – создание экземпляра класса JDOM `Attribute`. Конструктор этого класса принимает имя атрибута и его значение. Затем полученный экземпляр можно добавить к элементу (листу дерева) при помощи метода `setAttribute()` этого элемента. Этот подход представлен далее:

```
// Мы за пределами цикла, значит, осталось имя последнего элемента
    Element last = new Element(name);
    /* last.setText(propertyValue);  */
    Attribute attribute = new Attribute("value", propertyValue);
    current.setAttribute(attribute);
    current.addContent(last);
```

Внимание

Собираясь компилировать файл с этими изменениями, не забудьте импортировать класс `Attribute`:

```
import org.jdom.Attribute;
```

Более простой способ – воспользоваться одним из удобных методов, предлагаемых JDOM. Поскольку добавление атрибутов – очень распространенная задача, класс `Element` предоставляет перегруженную версию метода `setAttribute()`, принимающую имя и значение и создающую объект `Attribute`. В данном случае такой подход несколько понятнее:

```
// Мы за пределами цикла, значит, осталось имя последнего элемента
 Element last = new Element(name);
    /* last.setText(propertyValue); */
    last.setAttribute("value", propertyValue);
    current.addContent(last);
```

Данный вариант работает так же, как и предыдущий, и позволяет избежать необходимости использовать лишний оператор импортирования. Эти изменения можно скомпилировать и запустить программу. Новый вывод должен соответствовать представленному в примере 7.5.

Пример 7.5. Вывод PropsToXml с использованием атрибутов

```
<?xml version="1.0" encoding="UTF-8"?>
<properties>
  <org>
    <enhydra>
      <classpath>
        <separator value="&quot;:&quot;" />
      </classpath>
      <initialargs value="&quot;./bootstrap.conf&quot;" />
      <initialclass value="org.enhydra.multiServer.bootstrap.Bootstrap" />
      <classpath value="&quot;.&quot;" />
    </enhydra>
    <xml>
      <sax>
        <parser value="&quot;org.apache.xerces.parsers.SAXParser&quot;" />
      </sax>
    </xml>
  </org>
</properties>
```

Значение каждого свойства теперь является атрибутом последнего из вложенных элементов. Обратите внимание, что JDOM преобразовывает кавычки, недопустимые внутри значений атрибутов, в ссылки на сущности, так что выводимый документ является корректным. Однако вывод из-за этого становится менее аккуратным, в связи с чем некоторые предпочитают размещать текстовые данные внутри элементов, а не атрибутов.

Вывод XML с помощью JDOM

Перед тем как продолжить, немного поговорим о той части кода, которая выводит XML и которую раньше мы пропустили. Здесь она снова выделена:

```
private void convertToXML(Properties props, String xmlFilename)
    throws IOException {

    // Создаем новый документ JDOM с корневым элементом "properties"
    Element root = new Element("properties");
    Document doc = new Document(root);

    // Получаем имена свойств
    Enumeration propertyNames = props.propertyNames();
    while (propertyNames.hasMoreElements()) {
        String propertyName = (String)propertyNames.nextElement();
        String propertyValue = props.getProperty(propertyName);
        createXMLRepresentation(root, propertyName, propertyValue);
    }

    // Выводим документ в указанный файл
    XMLOutputter outputter = new XMLOutputter("  ", true);
```

```
        FileOutputStream output = new FileOutputStream(xmlFilename);
        outputter.output(doc, output);
    }
```

Мы уже знаем, что XMLOutputter — это класс, предназначенный для вывода в файл, поток или иной статический ресурс. Тем не менее, в примере кода мы передали несколько аргументов конструктору; класс без аргументов выводил бы текст без форматирования. XML-код никак не изменялся бы. В большинстве случаев это приводит к отсутствию разрывов строк и отступов в выводе. В полученном документе все данные, за исключением объявления XML, будут находиться в одной строке. Я бы показал вам это, но на одной странице это не поместится и может вас запутать. Класс XMLoutputter имеет несколько конструкторов:

```
public XMLOutputter();

public XMLOutputter(String indent);

public XMLOutputter(String indent, boolean newlines);

public XMLOutputter(String indent, boolean newlines, String encoding);

public XMLOutputter(XMLOutputter that);
```

Большинство из них говорят сами за себя. Параметр indent позволяет задать размер одиночного отступа. В нашем примере кода это два пробела (" "). Логическое значение для newlines определяет, будет ли использоваться символ новой строки (как в нашем примере). При необходимости можно задать параметр кодировки, который становится значением атрибута encoding в объявлении XML:

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

Кроме того, в этом классе существуют методы-модификаторы для всех свойств (setIndent(), setEncoding( ) и т. д.). Также существуют версии метода output() (одна из них используется в примере), принимающие либо объект OutputStream, либо Writer. Также существуют версии, работающие с различными конструкциями JDOM, что позволяет выводить либо документ целиком, либо отдельные элементы, комментарии, инструкции обработки (Element, Comment, ProcessingInstruction), в общем, произвольные конструкции:

```
// Создаем outputter с 4 пробелами для отступов и с разбиением строк
XMLOutputter outputter = new XMLOutputter("    ", true);

// Выводим различные конструкции JDOM
outputter.output(myDocument, myOutputStream);
outputter.output(myElement, myWriter);
outputter.output(myComment, myOutputStream);
// и т. д...
```

Другими словами, XMLOutputter служит для выполнения всех задач, связанных с выводом XML. Конечно, также можно использовать DOMOutputter и SAXOutputter, которые подробно рассмотрены в следующей главе.

Класс XMLProperties

Перейдем к следующему логическому шагу и поговорим о чтении данных XML. Продолжая работать с примером преобразования файла свойств в XML, вы, вероятно, интересуетесь, как получить доступ к информации из XML-файла. К счастью, для этого тоже существует решение! Чтобы объяснить, как JDOM читает XML, я хочу в этом разделе представить новый вспомогательный класс XMLProperties. Данный класс – это просто совместимая с XML версия класса Properties. По сути дела, он расширяет этот класс. Этот класс организует доступ к XML-документу через обычные методы для доступа к свойствам, наподобие getProperty() и properties(). Иначе говоря, доступ в стиле Java (при помощи класса Properties) к источнику данных в формате XML. По моему мнению, это лучшая комбинация, которую можно получить.

Решение задачи можно начать с создания класса XMLProperties, расширяющего класс java.util.Properties. Используя такой подход, для работы достаточно переопределить методы load(), save() и store(). Первый из них – load() – считывает XML-документ и загружает свойства из этого документа в объект суперкласса.

Внимание

Не следует путать этот класс с многоцелевой реализацией класса, отвечающего за переход от XML к файлам свойств. Он читает XML-код только в формате, подробно рассмотренном ранее в этой главе. Иначе говоря, свойства – это элементы либо с текстовыми значениями, либо с атрибутами, но не с теми и другими вместе. Мы рассмотрим оба подхода, но вам придется выбирать один из них. Не думайте, что можно взять произвольный XML-документ, обработать его и получить ожидаемые результаты!

Второй метод, save(), в Java 2 устарел, т. к. он не позволяет получать информацию об ошибках; тем не менее, пользователи Java 1.1 должны его переопределять. С этой целью в реализации метода из XMLProperties просто вызывается метод store(). А метод store() записывает информацию из свойств в XML-документ. Неплохой основой может послужить код из примера 7.6, с которым мы и будем работать.

Пример 7.6. Каркас класса XMLProperties

```
package javaxml2;

import java.io.File;
import java.io.FileReader;
```

```
import java.io.FileWriter;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Reader;
import java.io.Writer;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;
import java.util.Properties;

import org.jdom.Attribute;
import org.jdom.Comment;
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.output.XMLOutputter;

public class XMLProperties extends Properties {

    public void load(Reader reader)
        throws IOException {

        // Считываем XML-документ в объект Properties
    }

    public void load(InputStream inputStream)
        throws IOException {

        load(new InputStreamReader(inputStream));
    }

    public void load(File xmlDocument)
        throws IOException {

        load(new FileReader(xmlDocument));
    }

    public void save(OutputStream out, String header) {
        try {
            store(out, header);
        } catch (IOException ignored) {
            // Устаревшая версия, не передает ошибку
        }
    }

    public void store(Writer writer, String header)
        throws IOException {

        // Преобразуем свойства в XML и выводим их
    }
```

```
    public void store(OutputStream out, String header)
        throws IOException {

        store(new OutputStreamWriter(out), header);
    }

    public void store(File xmlDocument, String header)
        throws IOException {

        store(new FileWriter(xmlDocument), header);
    }
}
```

Обратите внимание, что мы перегрузили методы `load()` и `store()`. Хотя в классе `Properties` имеются лишь те версии, которые принимают `InputStream` и `OutputStream` (соответственно), я твердо верю в необходимость предоставления альтернатив пользователю. Дополнительные версии, работающие с `File` и `Reader/Writer`, упрощают пользователям жизнь и лишь незначительно увеличивают объем кода класса. Кроме того, эти перегруженные методы могут делегировать результаты работы существующим методам, так что остается лишь создать код для загрузки и сохранения данных.

Сохранение XML

Начнем с сохранения XML в основном из-за того, что этот код уже написан. Запись объекта `Properties` в XML-формате – это назначение класса `PropsToXML`, и мы просто повторно используем фрагмент кода:

```
    public void store(Writer writer, String header)
        throws IOException {

        // Создаем новый документ JDOM с корневым элементом "properties"
        Element root = new Element("properties");
        Document doc = new Document(root);

        // Получаем имена свойств
        Enumeration propertyNames = propertyNames();
        while (propertyNames.hasMoreElements()) {
            String propertyName = (String)propertyNames.nextElement();
            String propertyValue = getProperty(propertyName);
            createXMLRepresentation(root, propertyName, propertyValue);
        }

        // Выводим документ в указанный файл
        XMLOutputter outputter = new XMLOutputter("  ", true);
        outputter.output(doc, writer);
    }

    private void createXMLRepresentation(Element root,
                                         String propertyName,
                                         String propertyValue) {
```

```
        int split;
        String name = propertyName;
        Element current = root;
        Element test = null;

        while ((split = name.indexOf("."")) != -1) {
            String subName = name.substring(0, split);
            name = name.substring(split+1);

            // Проверка существования элемента
            if ((test = current.getChild(subName)) == null) {
                Element subElement = new Element(subName);
                current.addContent(subElement);
                current = subElement;
            } else {
                current = test;
            }
        }

    // Мы за пределами цикла, значит, осталось имя последнего элемента
        Element last = new Element(name);
        last.setText(propertyValue);
        /** Отмените комментарий, чтобы использовать атрибуты */
        /*
        last.setAttribute("value", propertyValue);
        */
        current.addContent(last);
    }
```

Не многое тут нуждается в комментариях. Несколько строк в тексте программы выделены, и отражают некоторые изменения. Первые два изменения гарантируют, что для получения имен и значений свойств используется суперкласс, а не объект `Properties`, переданный в версию этого метода в `PropsToXML`. Третье изменение позволяет перейти от строкового имени файла к объекту `Writer`, применяемому для вывода. Внеся эти незначительные изменения, можно скомпилировать исходный файл `XMLProperties`.

Тем не менее, мы упустили одну деталь. Обратите внимание, что метод `store()` позволяет указать параметр `header`; в стандартном файле свойств Java она (ее значение) добавляется в качестве комментария в заголовок файла (head). В соответствии с этим можно изменить класс `XMLProperties` так, чтобы он выполнял то же самое. Для этого придется воспользоваться классом `Comment`. Следующее изменение кода делает именно это:

```
    public void store(Writer writer, String header)
        throws IOException {

        // Создаем новый документ JDOM с корневым элементом "properties"
        Element root = new Element("properties");
        Document doc = new Document(root);
```

```
        // Добавляем информацию из заголовка
        Comment comment = new Comment(header);
        doc.addContent(comment);

        // Получаем имена свойств
        Enumeration propertyNames = propertyNames();
        while (propertyNames.hasMoreElements()) {
            String propertyName = (String)propertyNames.nextElement();
            String propertyValue = getProperty(propertyName);
            createXMLRepresentation(root, propertyName, propertyValue);
        }

        // Выводим документ в указанный файл
        XMLOutputter outputter = new XMLOutputter("  ", true);
        outputter.output(doc, writer);
    }
```

Метод `addContent()` объекта `Document` перегружен и принимает как объект `Comment`, так и объект `ProcessingInstruction` и дописывает содержимое в файл. Тут он применяется для добавления параметра `header` в качестве комментария в XML-документ, в который производится запись.

Загрузка XML

Осталось сделать не много. По существу, класс записывает данные в XML-формате, предоставляет доступ к XML (при помощи методов, уже существующих в классе `Properties`), и остается реализовать чтение XML. Это достаточно простая задача и сводится она к дополнительной рекурсии. Посмотрим, какие нужно внести изменения в код, а затем прокомментируем их. Введите представленный ниже код в исходный файл *XMLProperties.java*:

```
    public void load(Reader reader)
        throws IOException {

        try {
            // Загружаем XML в документ JDOM
            SAXBuilder builder = new SAXBuilder();
            Document doc = builder.build(reader);

            // Превращаем данные в объекты свойств
            loadFromElements(doc.getRootElement().getChildren(),
                new StringBuffer(""));

        } catch (JDOMException e) {
            throw new IOException(e.getMessage());
        }
    }

    private void loadFromElements(List elements, StringBuffer baseName) {
        // Перебираем элементы
        for (Iterator i = elements.iterator(); i.hasNext(); ) {
```

```
            Element current = (Element)i.next();
            String name = current.getName();
            String text = current.getTextTrim();

            // Если нет имени (baseName), то "." не дописываем
            if (baseName.length() > 0) {
                baseName.append(".");
            }
            baseName.append(name);

            // Проверяем, присутствует ли значение элемента
            if ((text == null) || (text.equals(""))) {
                // Если текста нет, рекурсивно обходим дочерние элементы
                loadFromElements(current.getChildren(),
                                 baseName);
            } else {
                // Если текст есть, то это свойство
                setProperty(baseName.toString(),
                            text);
            }

            // При выходе из рекурсии удаляем последнее имя
            if (baseName.length() == name.length()) {
                baseName.setLength(0);
            } else {
                baseName.setLength(baseName.length() -
                    (name.length() + 1));
            }
        }
    }
```

Реализация метода `load()` (который является делегатом всех перегруженных версий) для чтения переданного XML-документа использует `SAXBuilder`. Мы уже говорили об этом и рассмотрим это в еще больших подробностях в следующей главе. Пока же достаточно понять, что он просто считывает XML в объект JDOM `Document`.

Имя свойства состоит из имен всех элементов, предшествующих значению свойства и разделенных точками. Вот пример свойства в XML-формате:

```
<properties>
  <org>
    <enhydra>
      <classpath>"."</classpath>
    </enhydra>
  </org>
</properties>
```

Имя свойства можно получить, собрав имена элементов, предшествующих его значению (исключая элемент `properties`, который использовался в качестве контейнера верхнего уровня): `org`, `enhydra` и `class-`

path. Поставьте между ними точки и получите org.enhydra.classpath, то есть искомое имя свойства. Для выполнения этой задачи предназначен метод loadFromElements(). Он принимает список элементов, перебирает их в цикле и обрабатывает каждый элемент в отдельности. Если элемент имеет текстовое значение, то это значение добавляется к свойствам объекта суперкласса. Если же он имеет дочерние элементы, то он получает их, и рекурсия начинается заново для нового списка дочерних элементов. На каждом шаге рекурсии к переменной baseName, отслеживающей имена свойств, приписывается имя обрабатываемого элемента. В начале рекурсии переменная baseName будет иметь значение org, затем org.enhydra, а затем org.enhydra.classpath. На выходе же из рекурсии переменная baseName укорачивается, и из нее удаляется имя последнего элемента. Рассмотрим вызовы методов JDOM, позволяющие осуществить это.

Во-первых, вы должны были заметить несколько вызовов метода getChildren() экземпляров класса Element. Этот метод возвращает все дочерние элементы текущего элемента в виде списка Java List. Существуют версии этого метода, которые дополнительно принимают имя искомого элемента и возвращают либо все элементы с этим именем (getChildren(String name)), либо только первый дочерний элемент с этим именем (getChild(String name)). Также существуют версии этого метода, поддерживающие пространства имен, но их мы рассмотрим в следующей главе. До начала процесса рекурсии из объекта JDOM Document при помощи метода getRootElement() мы получаем корневой элемент, а затем его дочерние элементы, используемые для инициализации рекурсии. Внутри метода loadFromElements() для перемещения по списку элементов применяются стандартные классы Java (такие как java.util.Iterator). Чтобы проверить, является ли содержимое текстом, применяется метод getTextTrim(). Этот метод возвращает текстовое содержимое элемента и возвращает элемент без окружающих пробелов.[1] Таким образом, содержимое « текстовое содержимое » (обратите внимание на окружающие пробелы) будет возвращено в виде «текстовое содержимое». Да, это кажется чем-то тривиальным, но представьте себе более реалистичный пример XML:

```
<chapter>
  <title>
    Расширенный SAX
  </title>
</chapter>
```

Действительное текстовое содержимое элемента title, оказывается, состоит из нескольких пробелов, за которыми следует символ новой

[1] Также он удаляет повторные пробелы *между* словами. Текстовое содержимое « тут много пробелов » будет возвращено методом getTextTrim() в виде «тут много пробелов».

строки, затем опять пробелы, затем символы «Расширенный SAX», потом пробел, затем еще один символ новой строки, а затем еще пробелы. Другими словами, это не совсем то, что мы ожидали увидеть. Строка, возвращаемая методом `getTextTrim()`, – это просто «Расширенный SAX», т. е. то, что мы хотели бы получить в большинстве случаев. Однако если нам необходимо все содержимое полностью (часто это требуется для восстановления исходного документа точно в том виде, в каком он был получен), тогда можно воспользоваться методом `getText()`, возвращающим неизмененное содержимое элемента. Если содержимого нет, то метод возвращает пустую строку (`""`), что упрощает сравнение, как показано в примере кода. Вот и все, что касается этого вопроса: несколько простых методов – и код читает XML при помощи JDOM. Давайте посмотрим на этот класс в действии.

Пробная прогулка

Теперь, когда в классе `XMLProperties` все находится на своих местах, скомпилируйте его. Для тестирования его работы можно либо набрать, либо загрузить пример 7.7 – класс, который использует `XMLProperties` для загрузки XML-документа, выводит о нем некоторую информацию, а затем записывает свойства в формате XML.

Пример 7.7. Тестирование класса XMLProperties

```
package javaxml2;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Enumeration;

public class TestXMLProperties {

    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Использование: java
javaxml2.TestXMLProperties " +
                "[исходный XML-документ] [конечный XML-документ]");
            System.exit(0);
        }

        try {
            // Создаем и загружаем свойства
            System.out.println("Читаю свойства XML из " + args[0]);
            XMLProperties props = new XMLProperties();
            props.load(new FileInputStream(args[0]));

            // Выводим свойства и значения
            System.out.println("\n\n---- Значения свойств ----");
            Enumeration names = props.propertyNames();
            while (names.hasMoreElements()) {
                String name = (String)names.nextElement();
```

```
                String value = props.getProperty(name);
                System.out.println("Имя свойства: " + name +
                                   " имеет значение " + value);
            }

            // Сохраняем свойства
            System.out.println("\n\nЗаписываю свойства XML в " + args[1]);
            props.store(new FileOutputStream(args[1]),
                "Тестирование класса XMLProperties");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Этот класс делает не многое: он считывает свойства, использует их для вывода имен и значений всех свойств, а затем записывает в виде XML. Можно запустить эту программу для XML-файла, сгенерированного классом PropsToXML, который я приводил ранее в этой главе.

Внимание

Версия класса XMLProperties, используемая здесь, обрабатывает значения свойств, представленные в виде текстового содержимого элементов (первая версия класса PropsToXML), а не в виде значений атрибутов (вторая версия PropsToXML). Если потребуется применить класс PropsToXML для создания XML-данных, являющихся исходными для класса TestXMLProperties, то надо будет вернуться к этой ранней версии класса либо отменить все изменения. В противном случае мы не получим никаких значений свойств.

Передайте тестовой программе исходный XML-файл и имя конечного файла:

```
C:\javaxml2\build>java javaxml2.TestXMLProperties enhydraProps.xml
output.xml
Читаю свойства XML из enhydraProps.xml


---- Значения свойств ----
Имя свойства: org.enhydra.classpath.separator имеет значение ":"
Имя свойства: org.enhydra.initialargs имеет значение "./bootstrap.conf"
Имя свойства: org.enhydra.initialclass имеет значение
  org.enhydra.multiServer.bootstrap.Bootstrap
Имя свойства: org.enhydra.classpath имеет значение "."
Имя свойства: org.xml.sax.parser имеет значение
  "org.apache.xerces.parsers.SAXParser"


Записываю свойства XML в output.xml
```

Теперь мы имеем форматирование данных в формате XML.

Поиск с возвратом

Перед тем как завершить работу с кодом, обсудим некоторые моменты. Во-первых, давайте посмотрим на XML-файл, сгенерированный классом `TestXMLProperties`, т. е. на результат вызова метода `store()` для свойств. Он должен выглядеть подобно примеру 7.8, если вы используете XML-версию *enhydra.properties*, рассмотренную ранее в этой главе.

Пример 7.8. Вывод TextXMLProperties

```
<?xml version="1.0" encoding="UTF-8"?>
<properties>
  <org>
    <enhydra>
      <classpath>
        <separator>":"</separator>
      </classpath>
      <initialargs>"./bootstrap.conf"</initialargs>
      <initialclass>org.enhydra.multiServer.bootstrap.Bootstrap</
initialclass>
      <classpath>"."</classpath>
    </enhydra>
    <xml>
      <sax>
        <parser>"org.apache.xerces.parsers.SAXParser"</parser>
      </sax>
    </xml>
  </org>
</properties>
<!-- Тестирование класса XMLProperties -->
```

Не замечаете ничего странного? Комментарий для заголовка находится не там, где нужно. Давайте снова посмотрим на код, добавляющий этот комментарий, из метода `store()`:

```
// Создаем новый документ JDOM с корневым элементом "properties"
Element root = new Element("properties");
Document doc = new Document(root);

// Добавляем информацию в заголовок
Comment comment = new Comment(header);
doc.addContent(comment);
```

Корневой элемент появился перед комментарием из-за того, что он раньше был добавлен к объекту `Document`. Однако объект `Document` не может быть создан без корневого элемента – в некотором роде проблема курицы и яйца. Чтобы решить проблему, придется воспользоваться новым методом `getContent()`. Этот метод возвращает список, но этот список содержит все содержимое документа, включая комментарии, корневой элемент и инструкции обработки. Затем нужно предварить,

как показано ниже, элементы списка комментарием с помощью методов класса List:

```
        // Добавляем информацию в заголовок
        Comment comment = new Comment(header);
        doc.getContent().add(0, comment);
```

После внесения этих изменений вывод должен выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Тестирование класса XMLProperties -->
<properties>
  <org>
    <enhydra>
      <classpath>
        <separator>":"</separator>
      </classpath>
      <initialargs>"./bootstrap.conf"</initialargs>
      <initialclass>org.enhydra.multiServer.bootstrap.Bootstrap</
initialclass>
      <classpath>"."</classpath>
    </enhydra>
    <xml>
      <sax>
        <parser>"org.apache.xerces.parsers.SAXParser"</parser>
      </sax>
    </xml>
  </org>
</properties>
```

Метод getContent() также доступен для класса Element, и он возвращает все содержимое элемента независимо от модели (элементы, инструкции обработки, комментарии, сущности и строки текстовых данных).

Также важны изменения, которые необходимо внести в XMLProperties, чтобы для записи значений свойств применялись атрибуты, а не элементы. Эти изменения, которые необходимо внести в код для сохранения свойств (они закомментированы в исходном коде, так что ничего нового писать не придется), мы уже видели. Что касается загрузки, то изменения касаются проверки атрибута, а не текстового содержимого элемента. Это можно сделать при помощи метода getAttributeValue(String name), возвращающего значение указанного атрибута, либо null, если значение не существует. Изменение показано ниже:

```
    private void loadFromElements(List elements, StringBuffer baseName) {
        // Перебор всех элементов
        for (Iterator i = elements.iterator(); i.hasNext(); ) {
            Element current = (Element)i.next();
            String name = current.getName();
            // String text = current.getTextTrim();
            String text = current.getAttributeValue("value");
```

```
            // Если нет имени (baseName), то "." не дописываем
            if (baseName.length() > 0) {
                baseName.append(".");
            }
            baseName.append(name);

            // Проверяем, присутствует ли значение атрибута
            if ((text == null) || (text.equals(""))) {
                // Если текста нет, рекурсивно обходим дочерние элементы
                loadFromElements(current.getChildren(),
                                 baseName);
            } else {
                // Если текст есть, то это свойство
                setProperty(baseName.toString(),
                            text);
            }

            // При выходе из рекурсии удаляем последнее имя
            if (baseName.length() == name.length()) {
                baseName.setLength(0);
            } else {
                baseName.setLength(baseName.length() -
                    (name.length() + 1));
            }
        }
    }
```

Скомпилируйте эти изменения, и вы сможете работать со значениями атрибутов, а не с содержимым элементов. Оставьте код в том состоянии, которое вы предпочитаете (как я говорил раньше, я предпочитаю содержимое элементов), но если вы хотите использовать содержимое текстового элемента, обязательно отмените все внесенные изменения. Что бы вы ни предпочли, надеюсь, что вы уже выбрали свой путь в JDOM. И так же, как в случае с SAX и DOM, я настоятельно рекомендую Javadoc (либо локальную версию, либо версию в Интернете) в качестве краткого руководства по тем методам, которые не можете вспомнить. В любом случае, перед тем как закончить, поговорим немного на актуальную тему, имеющую отношение к JDOM, – о стандартизации.

Является ли JDOM стандартом?

По поводу JDOM мне чаще всего задают такой вопрос: является ли JDOM стандартом? Это распространенный вопрос, особенно среди тех, кто хочет пользоваться JDOM (и кому нужно обоснование), и среди тех, кому JDOM не нравится (и им тоже нужно обоснование). Мы поговорим здесь об этом, так что, к какому лагерю вы не принадлежали бы, вы будете знать подробности о JDOM и его процессе стандартизации.

JDOM – это JSR

Во-первых, JDOM официально входит в JSR, т. е. запрос на разработку или изменение спецификации Java-технологии (Java Specification Request). Другими словами, JDOM сейчас проходит формальный процесс стандартизации, спонсируемый Sun и управляемый JCP (Java Community Process). Все о процессах JSR и JCP можно прочитать на сайте *http://jcp.org*. Что касается JDOM, то сейчас он официально зарегистрирован как JSR-102, а соответствующая информация доступна по адресу *http://jcp.org/jsr/detail/102.jsp*.

Когда JDOM пройдет стадию JCP, произойдет несколько вещей. Во-первых, он получит более высокий статус в терминах стандартов; хотя JCP и Sun не идеальны, они внушают достаточно доверия. JCP имеет поддержку и своих членов в IBM, BEA, Compaq, HP, Apache и прочих компаниях. Во-вторых, упростится интеграция JDOM с другими стандартами Java. Например, в Sun заинтересованы в том, чтобы сделать JDOM частью следующей версии JAXP либо 1.2, либо 2.0 (о JAXP мы подробно поговорим в главе 9). И, в-третьих, планируется, что будущие версии JDK будут иметь XML в составе ядра. Со временем JDOM может оказаться в каждом дистрибутиве Java.

SAX и DOM как стандарты

Помните, что JDOM не будет особенным стандартом; DOM и SAX уже являются частью JAXP, и потому в этом отношении находятся впереди JDOM. Однако стоит сказать пару слов о «стандартизации» DOM и SAX. Во-первых, SAX является всеобщим достоянием, и сегодня остается стандартом де-факто. Первоначально разработанный в рамках списка рассылки XML-dev, SAX не имел утвержденного стандарта до тех пор, пока не начал широко применяться. И хотя я ни в коей мере не критикую SAX, я настороженно отношусь к тем, кто утверждает, что нельзя использовать JDOM, поскольку он не разрабатывался в рамках стандарта.

С другой стороны, DOM был разработан концерном W3C и является формальным стандартом. По этой причине он имеет стойких приверженцев. DOM – это отличное решение для многих приложений. Опять же, W3C – это просто одна из компаний, создающих стандарты, JCP – другая, IETF – еще одна и т. д. Я не обсуждаю достоинства какой-то определенной группы, я просто предостерегаю вас от признания стандарта (JDOM или иного), если он не соответствует потребностям вашего приложения. Споры о «стандартизации» отодвигают на задний план понятие usability (удобства и простоты использования). Если вам нравится DOM и он соответствует вашим потребностям, тогда пользуйтесь им. То же самое относится и к SAX и к JDOM. Мне бы очень хотелось, чтобы все перестали принимать решения за других (да, я защищаю свой API, но я постоянно сталкиваюсь с подобным!). К счастью,

эта книга достаточно подробно знакомит со всеми тремя API, так что вы можете принять обоснованное решение.

Советы разработчикам

Хочу предупредить о некоторых распространенных ловушках в JDOM, и делаю это не для того, чтобы вас разочаровать. Надеюсь, это сэкономит вам немного времени при программировании с JDOM.

JDOM – это не DOM

Во-первых, необходимо понимать, что JDOM – это не DOM. Он не включает в себя DOM и не предоставляет к нему расширений. Другими словами, эти модели не связаны в части реализации. Понимание этой простой истины сохранит вам много времени и усилий. Сейчас существует много статей, в которых говорится о настройке интерфейсов DOM для работы с JDOM, или о том, что необходимо избегать JDOM, поскольку он скрывает некоторые методы DOM. Такие утверждения смущают больше, чем что-либо другое. Вам не нужны интерфейсы DOM, а вызовы DOM (такие как `appendChild()` или `createDocument()`) просто не будут работать в JDOM. Извините, вы ошиблись API!

Возвращаемые значения Null

Еще один интересный аспект JDOM, вызывающий споры, – это значения, возвращаемые методами, извлекающими содержимое элементов. Например, различные методы `getChild()` класса `Element` могут возвращать значение `null`. Это было показано в примере кода класса `PropsToXML`. Ловушка подстерегает того, кто без проверки (как сделано в коде примера) предполагает, что элемент существует. Чаще всего это происходит, когда другое приложение или компонент посылает XML-код вашему приложению, а оно ожидает, что этот код соответствует определенному формату (будь то DTD, схема XML или просто некий оговоренный стандарт). Например:

```
Document doc = otherComponent.getDocument();
String price = doc.getRootElement().getChild("item")
                                   .getChild("price")
                                   .getTextTrim();
```

Проблема тут заключается в том, что если внутри корневого элемента отсутствует элемент `item` или элемент `price`, то в результате вызова метода `getChild()` возвращается значение `null`. Внезапно этот кажущийся безобидным код начинает генерировать исключение `NullPointerException`, причину которого отследить бывает довольно сложно. Эту ситуацию можно обработать двумя способами. Первый – на каждом шаге проверять, получено ли значение `null`:

```
Document doc = otherComponent.getDocument();
Element root = doc.getRootElement();
Element item = root.getChild("item");
if (item != null) {
    Element price = item.getChild("price");
    if (price != null) {
        String price = price.getTextTrim();
    } else {
        // Обработка исключительной ситуации
    }
} else {
    // Обработка исключительной ситуации
}
```

Второй вариант – заключить весь фрагмент кода в блок `try/catch`:

```
Document doc = otherComponent.getDocument();
try {
    String price = doc.getRootElement().getChild("item")
                                        .getChild("price")
                                        .getTextTrim();
} catch (NullPointerException e) {
    // Обработка исключительной ситуации
}
```

И хотя оба подхода работают, я рекомендую первый; он допускает тонкую подстройку обработки ошибок и позволяет точно определить, какая именно проверка сработала, а значит, точно выяснить, какая проблема имеет место. Второй вариант кода лишь информирует о том, что где-то возникла проблема. В любом случае тщательная проверка возвращаемых значений может избавить вас от довольно назойливых исключений `NullPointerException`.

Класс DOMBuilder

Последнее, но не менее важное – нужно быть очень осторожным при работе с классом `DOMBuilder`. Дело не в том, как используется этот класс, а когда. Как я уже говорил, этот класс работает аналогично классу `SAXBuilder` для ввода. И подобно классу SAX он имеет методы `build()`, принимающие такие формы ввода, как `File` или `InputStream`. Однако создание документа JDOM на основе файла, URL или потока ввода/вывода всегда происходит медленнее, чем при использовании `SAXBuilder`. Дело в том, что в `DOMBuilder` для создания дерева DOM применяется SAX, а затем это дерево DOM преобразуется в JDOM. Конечно же, это гораздо медленнее, чем в случае отсутствия промежуточного шага (создания дерева DOM) и простого перехода от SAX к JDOM.

Поэтому, увидев такой код:

```
DOMBuilder builder = new DOMBuilder();

// Создание на основе файла
```

```
Document doc = builder.build(new File("input.xml"));

// Создание по URL
Document doc = builder.build(
    new URL("http://newInstance.com/javaxml2/copyright.xml"));

// Создание на основе потока ввода/вывода
Document doc = builder.build(new FileInputStream("input.xml"));
```

надо убегать с криком! Если серьезно, то `DOMBuilder` имеет свою нишу: он отлично подходит для перехода от существующих структур DOM к JDOM. В остальных случаях это просто худший выбор в терминах производительности. Зафиксируйте этот факт в памяти и избавьте себя тем самым от головной боли!

Что дальше?

Далее следует глава о расширенном JDOM. В ней рассмотрены некоторые тонкости работы с API, вроде пространств имен и адаптеров DOM; рассмотрена внутренняя организация списков в JDOM, а также все то, что может заинтересовать тех из вас, кто действительно хочет освоиться с этим интерфейсом. Этих знаний вам должно быть достаточно, чтобы использовать JDOM наравне с DOM и SAX в своих приложениях.

8

- *Полезная информация по внутренней организации JDOM*
- *JDOM и фабрики*
- *Классы Wrapper и Decorator*
- *Советы разработчикам*
- *Что дальше?*

Расширенный JDOM

Продолжаем работать с JDOM и переходим к более сложным понятиям. В предыдущей главе мы узнали, как читать и записывать данные в формате XML при помощи JDOM, а также получили представление о том, какие классы доступны в дистрибутиве JDOM. Копнем немного глубже, чтобы выяснить, что происходит на самом деле. Вы увидите некоторые из классов, которые применяются в JDOM, но незаметны при выполнении обычных действий, а также начнете понимать, как построена модель JDOM. После этого мы рассмотрим применение в JDOM фабрик и пользовательских классов реализаций, причем способом, совершенно непохожим на тот, который нам уже знаком (помните разговор о DOM?). И наконец, нам останется рассмотреть довольно сложный пример применения оберток (wrappers) и декораторов (decorators), которые предоставляют дополнительный способ изменения базовой функциональности JDOM без необходимости применять интерфейсный API.

Полезная информация по внутренней организации JDOM

Первая тема, которую мы рассмотрим, – это архитектура JDOM. В главе 7 была приведена простая UML-модель основных классов JDOM. Однако если присмотреться внимательнее, то можно увидеть, что в классах, вероятно, есть некоторые вещи, с которыми вы не работали или о которых не знали. Их мы и рассмотрим в этом разделе.

Примечание

Версия JDOM beta 7 вышла буквально за несколько дней до того, как была написана эта глава. В этой версии класс Text был лишь обозначен, но не был интегрирован в архитектуру JDOM. Однако все развивается очень быстро, и, вероятно, этот класс станет полноправным элементом модели еще до того, как данная книга попадет в руки читателя. Даже если это и не так, он все равно будет интегрирован очень скоро, и все моменты, обсуждаемые здесь, будут применимы. Если у вас возникнут проблемы при использовании фрагментов кода из этого раздела, проверьте версию JDOM, с которой вы работаете, и попытайтесь получить самую свежую из доступных.

Класс Text

Увидев в JDOM класс Text, вы, должно быть, немного удивились. Тот, кто читал предыдущую главу, вероятно, понял, что основное различие между DOM и JDOM заключается в том, что JDOM (по крайней мере, так кажется) непосредственно раскрывает текстовое содержимое элемента, тогда как в DOM необходимо получить дочерний узел Text и затем выделить его значение. На самом же деле происходит следующее: JDOM моделирует символьное содержимое во многом подобно тому, как DOM делает это архитектурно: каждый фрагмент символьного содержимого хранится в экземпляре JDOM Text. Однако когда вызывается метод getText() (или getTextTrim(), или getTextNormalize()) экземпляра JDOM Element, автоматически возвращается значение(я) из его дочерних узлов Text:

```
// Получаем текстовое содержимое
String textualContent = element.getText();

// Получаем текстовое содержимое, отбрасывая оконечные пробелы
String trimmedContent = element.getText().trim();
// или...
String trimmedContent = element.getTextTrim();

// Получаем нормализованное текстовое содержимое (каждая внедренная
последовательность пробелов
// сжимается до одного). Например, из фразы "   это   будет "
// получится "это будет"
String normalizedContent = element.getTextNormalize();
```

В результате обычно кажется, что на самом деле класс Text не используется. То же применимо и к вызову метода setText() элемента; новый текст сохраняется в новом экземпляре Text, и этот экземпляр добавляется в качестве дочернего элемента к элементу (element). Опять же, логическое обоснование этого заключается в том, что процесс чтения и записи текстового содержимого XML-элемента – настолько распространенная задача, что она должна решаться максимально просто и эффективно.

В то же время, как говорилось в предшествующих главах, строгая древовидная модель упрощает навигацию; instanceof и рекурсия позволяют реализовывать простые решения для исследования дерева. Следовательно, явным образом существующий класс Text в роли дочернего элемента экземпляра Element лишь дополнительно упрощает задачу. Кроме того, класс Text допускает расширение, тогда как классы java.lang.String не расширяемы. По всем этим причинам (и другим, которые можно найти в списках рассылки jdom-interest) класс Text был добавлен в JDOM. И хотя это не так очевидно, как в других API, он все же доступен в случаях, когда применяются итерации. Так, вызвав метод getContent() для экземпляра Element, мы получим все содержимое этого элемента. В него могут входить комментарии, инструкции обработки, ссылки на сущности, секции CDATA и текстовое содержимое. В данном случае текстовое содержимое возвращается в виде одного или нескольких экземпляров класса Text, а не в виде строк, что позволяет выполнить следующую обработку:

```
public void processElement(Element element) {
    List mixedContent = element.getContent();
    for (Iterator i = mixedContent.iterator(); i.hasNext(); ) {
        Object o = i.next();
        if (o instanceof Text) {
            processText((Text)o);
        } else if (o instanceof CDATA) {
            processCDATA((CDATA)o);
        } else if (o instanceof Comment) {
            processComment((Comment)o);
        } else if (o instanceof ProcessingInstruction) {
            processProcessingInstruction((ProcessingInstruction)o);
        } else if (o instanceof EntityRef) {
            processEntityRef((EntityRef)o);
        } else if (o instanceof Element) {
            processElement((Element)o);
        }
    }
}

public void processComment(Comment comment) {
    // Обработка комментариев
}

public void processProcessingInstruction(ProcessingInstruction pi) {
    // Обработка PI
}

public void processEntityRef(EntityRef entityRef) {
    // Обработка ссылок на сущности
}

public void processText(Text text) {
    // Обработка текста
```

```
}

public void processCDATA(CDATA cdata) {
    // Обработка CDATA
}
```

Такова подготовка к довольно простой рекурсивной обработке дерева JDOM. А вот так ее можно инициировать:

```
// Получаем документ JDOM через builder
Document doc = builder.build(input);

// Начало рекурсии
processElement(doc.getRootElement());
```

Комментарии и инструкции обработки придется обрабатывать на уровне документа, но саму идею вы поняли. Есть возможность использовать класс `Text` только тогда, когда это имеет смысл, и не беспокоиться о нем в противном случае.

Класс EntityRef

Следующим в списке составляющих JDOM стоит класс `EntityRef`. Это еще один класс, который, вероятно, вам не часто будет нужен, но о котором полезно знать для программирования в конкретных случаях. В JDOM данный класс реализует работу с сущностями XML. Если помните, в примерах мы пользовались ссылкой на сущность `OReilly-Copyright`:

```
<ora:copyright>&OReillyCopyright;</ora:copyright>
```

Этот класс позволяет устанавливать и получать имя, публичный и системный идентификаторы точно так же, как это можно делать при определении сущности в XML DTD или в схеме. Подобно узлам `Elements` и `Text` он может появляться в любом месте дерева JDOM. Однако, как и узлы `Text`, класс `EntityRef` зачастую неудобен в обычных условиях. Например, в документе *contents.xml*, смоделированном в JDOM, нас скорее заинтересует текстовое значение ссылки (т. е. уже раскрытое содержимое), а не собственно ссылка. Другими словами, при вызове метода `getContent()` для элемента с информацией о правообладании в дереве JDOM вы хотите получить «Copyright O'Reilly, 2000» или иное текстовое значение, на которое указывает ссылка на сущность. Это гораздо полезнее (опять же, в общем случае), чем получение указания на отсутствие содержимого (пустой строки) с последующей проверкой существования `EntityRef`. По этой причине в случае применения `SAXBuilder` и `DOMBuilder` для создания деревьев JDOM на основе существующих данных XML все ссылки на сущности по умолчанию интерпретируются. По умолчанию элементы `EntityRef` практически не встречаются, да и нет интереса с ними возиться. Однако если необходимо получить ссылки на сущности в исходном виде (в виде элементов

EntityRef), можно обратиться к методу setExpandEntities() builder-классов:

```
// Создаем новый builder
SAXBuilder builder = new SAXBuilder();

// Не интерпретировать ссылки на сущности (по умолчанию интерпретируются)
builder.setExpandEnitites(false);

// Создаем дерево с объектами EntityRef (если это нужно, конечно же)
Document doc = builder.build(inputStream);
```

В данном случае в дереве могут присутствовать элементы EntityRef (например, если мы работаем с документом *contents.xml*). Всегда можно непосредственно создать элементы EntityRef и поместить их в дерево JDOM:

```
// Создаем новую ссылку на сущность
EntityRef ref = new EntityRef("TrueNorthGuitarsTagline");
ref.setSystemID("tngTagline.xml");

// Добавляем в дерево
tagLineElement.addContent(ref);
```

При сериализации этого дерева мы получим подобный XML-код:

```
<guitar>
  <tagLine>&TrueNorthGuitarsTagline;</tagLine>
</guitar>
```

А при чтении документа при помощи builder-класса JDOM результат будет зависеть от флага expandEntities. Если он установлен в значение «ложь», то мы получим первоначальную ссылку на сущность с верным именем и системным идентификатором. Если же значение – «истина» (по умолчанию), то будет получено раскрытое содержимое. Вторая сериализация может привести к следующим результатам:

```
<guitar>
  <tagLine>two hands, one heart</tagLine>
</guitar>
```

И хотя может показаться, что я искусственно создаю шум вокруг этого вопроса, очень важно понимать, что в зависимости от того, интерпретируются или нет ссылки на сущности, может изменяться исходный и получаемый XML-код, с которым вы работаете. Всегда учитывайте, в какие значения установлены флаги builder-класса и какими вы хотите видеть дерево JDOM и получаемый XML-код.

Класс Namespace

Хочу вкратце рассмотреть еще один класс JDOM – Namespace. В архитектуре JDOM этот класс применяется и для создания экземпляров

и в качестве фабрики. Для создания нового пространства имен для элемента или для поиска следует воспользоваться статическим методом getNamespace() из этого класса:

```
// Создаем пространство имен с префиксом
Namespace schemaNamespace =
    Namespace.getNamespace("xsd", "http://www.w3.org/XMLSchema/2001");

// Создаем пространство имен без префикса
Namespace javaxml2Namespace =
    Namespace.getNamespace("http://www.oreilly.com/javaxml2");
```

Как видите, существуют две версии: одна для создания пространств имен с префиксом, а другая – без префикса (пространство имен по умолчанию). Можно использовать любую из них, а затем передавать полученный элемент различным методам JDOM:

```
// Создание элемента с пространством имен
Element schema = new Element("schema", schemaNamespace);

// Поиск дочерних элементов в указанном пространстве имен
List chapterElements = contentElement.getChildren("chapter",
javaxml2Namespace);

// Объявление нового пространства имен для этого элемента
catalogElement.addNamespaceDeclaration(
    Namespace.getNamespace("tng", "http://www.truenorthguitars.com"));
```

Здесь все вполне очевидно. Кроме того, когда выполняется сериализация XML при помощи SAXOutputter, DOMOutputter либо XMLOutputter, объявления пространств имен обрабатываются автоматически и добавляются в конечный XML-код.

И последнее замечание: в JDOM сравнение пространств имен основано исключительно на URI. Другими словами, два объекта Namespace идентичны, если совпадают их URI, независимо от префикса. Это соответствует букве и духу спецификации пространств имен XML, в которой указано, что два элемента находятся в одном и том же пространстве имен, если идентичны их URI, независимо от префикса. Взгляните на этот фрагмент XML-документа:

```
<guitar xmlns="http://www.truenorthguitars.com">
  <ni:owner xmlns:ni="http://www.newInstance.com">
    <ni:name>Brett McLaughlin</ni:name>
    <tng:model xmlns:tng="http://www.truenorthguitars.com>Model
    1</tng:model>
    <backWood>Madagascar Rosewood</backWood>
  </ni:owner>
</guitar>
```

Элементы guitar, model и backWood принадлежат одному пространству имен несмотря на то, что имеют различные префиксы. То же справед-

ливо и для класса JDOM `Namespace`. На самом деле метод `equals()` класса `Namespace` будет определять равенство исходя лишь из URI, не принимая во внимание префикс.

Мы коснулись лишь трех классов JDOM, но именно о них чаще всего задают вопросы и именно они представляют сложность. Остальная часть API была рассмотрена в предыдущей главе и будет повторена в следующих разделах этой главы. Теперь вы сможете легко работать с текстовым содержимым, ссылками на сущности и пространствами имен в JDOM, переходя от строк к текстовым узлам, от раскрытого содержимого к ссылкам на сущности, а также с пространствами имен с несколькими префиксами. Теперь перейдем к более сложным примерам.

JDOM и фабрики

Двигаясь далее, вспомните обсуждение JDOM и фабрик в предыдущей главе. Я уже говорил, что вы никогда не увидите подобного кода (по крайней мере, по состоянию версий JDOM на сегодня) в приложениях JDOM:

```
// Такой код не работает!!
JDOMFactory factory = new JDOMFactory();
factory.setDocumentClass("javaxml2.BrettsDocumentClass");
factory.setElementClass("javaxml2.BrettsElementClass");

Element rootElement = JDOMFactory.createElement("root");
Document document = JDOMFactory.createDocument(rootElement);
```

Да, это по-прежнему справедливо. Однако в том разговоре ничего не было сказано о некоторых довольно важных аспектах, и я хочу снова поднять этот вопрос. Как было сказано в главе 7, возможность иметь некое подобие фабрики допускает большую гибкость в отношении того, как XML моделируется в Java. Взгляните на простой подкласс класса JDOM `Element`, показанный в примере 8.1.

Пример 8.1. Подкласс класса Element JDOM

```
package javaxml2;

import org.jdom.Element;
import org.jdom.Namespace;

public class ORAElement extends Element {

    private static final Namespace ORA_NAMESPACE =
        Namespace.getNamespace("ora", "http://www.oreilly.com");

    public ORAElement(String name) {
        super(name, ORA_NAMESPACE);
    }

    public ORAElement(String name, Namespace ns) {
```

```
        super(name, ORA_NAMESPACE);
    }

    public ORAElement(String name, String uri) {
        super(name, ORA_NAMESPACE);
    }

    public ORAElement(String name, String prefix, String uri) {
        super(name, ORA_NAMESPACE);
    }
}
```

Данный подкласс прост до безобразия. В какой-то степени он похож на класс `NamespaceFilter` из главы 4. Он не придает значения указанному пространству имен элемента (даже если пространство имен не задано!) и устанавливает пространство имен элемента, определяемое URI *http://www.oreilly.com* с префиксом `ora`.[1] Это простой случай, но он дает представление о доступных возможностях и служит хорошим примером для этого раздела.

Создание фабрики

Создав пользовательский подкласс, на следующем шаге необходимо приступить к работе с ним. Как уже говорилось, JDOM считает создание всех объектов при помощи фабрик чрезмерным. Простое создание элемента в JDOM работает так:

```
// Создание нового элемента
Element element = new Element("guitar");
```

Настолько же простым все остается в ситуации с пользовательскими подклассами:

```
// Создание нового элемента ORAElement
Element oraElement = new ORAElement("guitar");
```

Элемент попадает в пространство имен O'Reilly из-за особенностей реализации пользовательского подкласса. Кроме того, такой подход более нагляден, чем применение фабрик. В любой момент понятно, какие конкретно классы предназначены для создания объектов. Сравните это со следующим фрагментом кода:

```
// Создание элемента: какого он типа?
Element someElement = doc.createElement("guitar");
```

Совершенно неясно, является ли создаваемый элемент экземпляром класса `Element`, `ORAElement` или какого-то совершенно иного класса.

[1] Он несколько отличается от класса `NamespaceFilter`, поскольку изменяет значение пространства имен для всех элементов, а не только для тех, которые принадлежат определенному пространству имен.

По этим причинам пользовательские классы весьма удобны в JDOM. Для создания объекта можно просто создать экземпляр пользовательского класса напрямую. Однако при создании документа необходимость в фабриках все же возникает:

```
// Создание на основе источника данных
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(someInputStream);
```

Очевидно, что тут невозможно указать пользовательские классы в процессе создания (построения). Полагаю, кто-то мог бы отважиться изменить класс SAXBuilder (и связанный с ним класс org.jdom.input.SAXHandler), но это довольно глупо. Чтобы решить вопрос, был создан интерфейс JDOMFactory из пакета org.jdom.input. Этот интерфейс определяет методы для создания объектов всех типов (полный список можно найти в приложении А). Так, существует четыре метода для создания элемента, что соответствует четырем конструкторам класса Element:

```
public Element element(String name);
public Element element(String name, Namespace ns);
public Element element(String name, String uri);
public Element element(String name, String prefix, String uri);
```

Подобные методы можно найти для Document, Attribute, CDATA и всего остального. По умолчанию JDOM использует фабрику org.jdom.input.DefaultJDOMFactory, которая просто возвращает все основные классы JDOM из этих методов. Однако можно очень просто создать подкласс для этой реализации и предоставить пользовательские методы фабрик. Посмотрите на пример 8.2, определяющий собственную фабрику.

Пример 8.2. Собственная реализация JDOMFactory

```
package javaxml2;

import org.jdom.Element;
import org.jdom.Namespace;
import org.jdom.input.DefaultJDOMFactory;

class CustomJDOMFactory extends DefaultJDOMFactory {

    public Element element(String name) {
        return new ORAElement(name);
    }

    public Element element(String name, Namespace ns) {
        return new ORAElement(name, ns);
    }

    public Element element(String name, String uri) {
        return new ORAElement(name, uri);
    }

    public Element element(String name, String prefix, String uri) {
```

```
        return new ORAElement(name, prefix, uri);
    }
}
```

Это простая реализация, она не должна быть очень сложной. Она переопределяет все методы element() и возвращает экземпляр пользовательского подкласса ORAElement вместо класса JDOM Element (по умолчанию). Теперь любой builder-класс, использующий эту фабрику, получит экземпляры ORAElement в созданном объекте JDOM Document, а не экземпляры Element, которые там можно увидеть по умолчанию. Теперь осталось лишь указать в процессе создания на эту пользовательскую фабрику.

Построение дерева с применением пользовательских классов

Теперь у нас есть рабочая реализация JDOMFactory, и следует сообщить builder-классу о необходимости ее использования при помощи вызова setFactory(). В качестве аргумента передается экземпляр фабрики. Этот метод доступен для обоих классов, SAXBuilder и DOMBuilder. Чтобы увидеть это в действии, посмотрите на пример 8.3. Этот простой класс принимает XML-документ и создает его представление при помощи классов ORAElement и CustomJDOMFactory из примеров 8.1 и 8.2. Затем он записывает документ обратно в указанный файл, так что можно увидеть результат работы пользовательских классов.

Пример 8.3. Создание документа при помощи пользовательских классов и фабрики

```
package javaxml2;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

import org.jdom.Document;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;
import org.jdom.input.JDOMFactory;
import org.jdom.output.XMLOutputter;

public class ElementChanger {

    public void change(String inputFilename, String outputFilename)
        throws IOException, JDOMException {

        // Создание builder-класса и указание фабрики
        SAXBuilder builder = new SAXBuilder();
        JDOMFactory factory = new CustomJDOMFactory();
        builder.setFactory(factory);

        // Создание документа
```

```
        Document doc = builder.build(inputFilename);

        // Конечный документ
        XMLOutputter outputter = new XMLOutputter();
        outputter.output(doc, new FileWriter(new File(outputFilename)));
    }

    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Использование: javaxml2.ElementChanger " +
                "[Имя исходного XML-файла] [Имя конечного XML-файла]");
            return;
        }

        try {
            ElementChanger changer = new ElementChanger();
            changer.change(args[0], args[1]);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

После запуска этой программы для файла *contents.xml*, который мы использовали в первых главах книги:

```
bmclaugh@GANDALF
$ java javaxml2.ElementChanger contents.xml newContents.xml
```

компьютер задумался на секунду, а затем создал новый документ (*newContents.xml*). Фрагмент этого нового документа приводится в примере 8.4.

Пример 8.4. Фрагмент файла, получаемого из contents.xml после завершения работы ElementChanger

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">
<!-- "Java and XML", Содержание -->
<ora:book xmlns:ora="http://www.oreilly.com">
  <ora:title ora:series="Java">Java и XML</ora:title>

  <!-- Список глав -->
  <ora:contents>
    <ora:chapter title="Введение" number="1">
      <ora:topic name="XML имеет значение" />
      <ora:topic name="Что важно" />
      <ora:topic name="Основы" />
      <ora:topic name="Что дальше?" />
    </ora:chapter>
    <ora:chapter title="Основы технологии" number="2">
      <ora:topic name="Основы" />
      <ora:topic name="Ограничения" />
```

```
        <ora:topic name="Преобразования" />
        <ora:topic name="И далее..." />
        <ora:topic name="Что дальше?" />
      </ora:chapter>
      <ora:chapter title="SAX" number="3">
        <ora:topic name="Подготовительные операции" />
        <ora:topic name="SAX-совместимые анализаторы" />
        <ora:topic name="Обработчики содержимого" />
        <ora:topic name="Советы разработчикам" />
        <ora:topic name="Что дальше?" />
      </ora:chapter>
      <ora:chapter title="Расширенный SAX" number="4">
        <ora:topic name="Свойства и возможности" />
        <ora:topic name="И опять обработчики" />
        <ora:topic name="Фильтры и объекты Writer" />
        <ora:topic name="И снова обработчики" />
        <ora:topic name="Советы разработчикам" />
        <ora:topic name="Что дальше?" />
      </ora:chapter>
      <!-- Другие главы -->
  </ora:book>
```

Все элементы теперь принадлежат пространству имен O'Reilly с префиксом и URI, заданными классом `ORAElement`.

Очевидно, что можно перенести это наследование классов на более высокий уровень сложности. Среди распространенных примеров – добавление определенных атрибутов или даже дочерних элементов каждому обрабатываемому элементу. Многие разработчики уже имеют существующие бизнес-интерфейсы и определяют пользовательские подклассы JDOM, расширяющие основные классы JDOM и реализующие эти специфические интерфейсы. Другие разработчики создают «облегченные» подклассы, которые отвергают информацию о пространствах имен и поддерживают лишь основные элементы, уменьшая при этом размер документов (которые при этом не всегда сохраняют совместимость с XML). Единственные ограничения – это ваше собственное представление о наследовании. Просто помните о том, что необходимо указать пользовательскую фабрику перед созданием документа, чтобы применить созданные вами новые возможности.

Классы Wrapper и Decorator

Наиболее распространенные просьбы, возникающие в связи с JDOM, связаны с интерфейсами. Многие, многие пользователи просили реализовать интерфейсы в JDOM, но эти просьбы постоянно отвергались. Причина проста: невозможно создать набор общих методов для всех конструкций JDOM. Желания же воспользоваться подходом DOM, предоставляющим набор общих методов для большинства конструкций, не возникало. Например, метод `getChildren()` находится в общем

интерфейсе DOM org.w3c.dom.Node; однако когда он не применим, он возвращает значение null, как в случае с узлом Text. Подход в стиле JDOM состоит в том, чтобы включать в базовый интерфейс лишь методы, применимые для всех классов JDOM, но классы, удовлетворяющие этим требованиям, найдены не были. Кроме того, на каждую просьбу о добавлении интерфейсов приходит просьба оставить интерфейс таким, какой он есть.

Однако существуют способы использовать интерфейсную функциональность в JDOM, при этом не меняя кардинально API (на самом деле не меняя его вовсе!). В данном разделе мы поговорим о наиболее эффективных способах, связанных с применением *оберток* (*wrappers*) или *декораторов* (*decorators*). В этой книге мы не будем вдаваться в подробности образцов проектирования, но достаточно сказать, что «wrapper» и «decorator» (в данной главе они используются взаимозаменяемым образом) представляют собой внешние сущности по отношению к существующим классам, хотя базовый интерфейс JDOM был частью этих классов. Другими словами, имеет место инкапсуляция существующей функциональности. Здесь будет показано, как образцы позволяют настроить JDOM (или другой API) любым желаемым способом.

Примечание

К этому времени читатель уже должен довольно хорошо разбираться в Java и XML. Поэтому код примеров в этом разделе приводится с минимальным количеством комментариев. Вы должны довольно легко сообразить, что происходит, и мне лучше приводить больше кода, чем говорить.

Интерфейс JDOMNode

Для начала в примере 8.5 определяется интерфейс JDOMNode. Этот интерфейс описывает очень простые возможности, которые я хочу сделать доступными для всех узлов JDOM без необходимости выполнять приведение типов.

Пример 8.5. Интерфейс decorator для узлов

```
package javaxml2;

import java.util.List;
import java.util.Iterator;

// Импортируем JDOM
import org.jdom.Document;

public interface JDOMNode {

    public Object getNode();

    public String getNodeName();

    public JDOMNode getParentNode();
```

```
    public String getQName();

    public Iterator iterator();

    public String toString();
}
```

Единственный метод, который может показаться странным, – это `iterator()`; он будет возвращать либо Java-объект `Iterator` вместо дочернего узла, либо пустой список `Iterator`, если у узла нет дочерних элементов (как в случае атрибутов или текстовых узлов). Стоит заметить, что так же легко можно было бы выбрать интерфейс DOM `org.w3c.dom.Node` (если бы нам необходима была возможность взаимодействия DOM и JDOM на уровне классов) или иной интерфейс, удовлетворяющий нашим потребностям. У этого базового интерфейса нет ограничений.

Реализация классов

Следующий, более интересный шаг заключается в обеспечении реализации интерфейса, декорирующего существующие конструкции JDOM. Они являются обертками для конкретных классов, уже существующих в JDOM, и большинство методов интерфейса `JDOMNode` просто передаются базовому (декорированному) объекту. Прежде всего, рассмотрим пример 8.6, декорирующий JDOM `Element`.

Пример 8.6. Декоратор для JDOM Element

```
package javaxml2;

import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

// Импортируем JDOM
import org.jdom.Element;

public class ElementNode implements JDOMNode {

    /** Декорированный Element */
    protected Element decorated;

    public ElementNode(Element element) {
        this.decorated = element;
    }

    public Object getNode() {
        return decorated;
    }

    public String getNodeName() {
        if (decorated != null) {
            return decorated.getName();
        }
        return "";
```

```
    }

    public JDOMNode getParentNode() {
        if (decorated.getParent() != null) {
            return new ElementNode(decorated.getParent());
        }
        return null;
    }

    public String getQName() {
        if (decorated.getNamespacePrefix().equals("")) {
            return decorated.getName();
        } else {
            return new StringBuffer(decorated.getNamespacePrefix())
                            .append(":")
                            .append(decorated.getName()).toString();
        }
    }

    public Iterator iterator() {
        List list = decorated.getAttributes();
        ArrayList content = new ArrayList(list);

        // помещаем содержимое элемента в список
        Iterator i = decorated.getMixedContent().iterator();
        while (i.hasNext()) {
            content.add(i.next());
        }
        return content.iterator();
    }

    public String toString() {
        return decorated.toString();
    }
}
```

Тут нет ничего, что заслуживало бы внимания, поэтому двинемся дальше. В примере 8.7 определен сходный класс `AttributeNode`, декорирующий JDOM `Attribute` и реализующий мой основной класс `JDOMNode`. Обратите внимание на несколько методов, предназначенных для получения дочерних элементов атрибута и не выполняющих действий; здесь скрупулезно моделируется подход, применяемый в DOM. Опять же, помните, что эти классы могли бы точно так же реализовывать любой другой интерфейс (в данном случае вспомните об `org.w3c.dom.Attr`), и для этого не придется вносить изменения в базовый JDOM API.

Пример 8.7. Декоратор для JDOM Attribute

```
package javaxml2;

import java.util.Iterator;
import java.util.Collections;

// Импортируем JDOM
```

```
import org.jdom.Attribute;

public class AttributeNode implements JDOMNode {

    /** Декорированный атрибут */
    protected Attribute decorated;

    public AttributeNode(Attribute attribute) {
        this.decorated = attribute;
    }

    public Object getNode() {
        return decorated;
    }

    public String getNodeName() {
        if (decorated != null) {
            return decorated.getName();
        }
        return "";
    }

    public JDOMNode getParentNode() {
        if (decorated.getParent() != null) {
            return new ElementNode(decorated.getParent());
        }
        return null;
    }

    public String getQName() {
        if (decorated.getNamespacePrefix().equals("")) {
            return decorated.getName();
        } else {
            return new StringBuffer(decorated.getNamespacePrefix())
                            .append(":")
                            .append(decorated.getName()).toString();
        }
    }

    public Iterator iterator() {
        return Collections.EMPTY_LIST.iterator();
    }

    public String toString() {
        return decorated.toString();
    }
}
```

Наконец, декорируется текстовое содержимое JDOM (пример 8.8). Во время написания этой книги класс JDOM `Text`, о котором говорилось в начале этой главы, еще не был полностью интегрирован в окончательном виде в дерево исходных текстов JDOM. В результате фактически происходит оборачивание (wrapping) класса Java `String` в класс `TextNode`. Когда появится узел `Text`, код необходимо будет обновить таким образом, чтобы он работал с данным типом, и это будет несложно сделать.

Пример 8.8. Декоратор для текстового содержимого JDOM

```
package javaxml2;

import java.util.Collections;
import java.util.Iterator;

// Импортируем JDOM
import org.jdom.Element;

public class TextNode implements JDOMNode {

    /** Декорированная строка */
    protected String decorated;

    /** Вручную устанавливаем родительский узел для текстового содержимого */
    private Element parent = null;

    public TextNode(String string) {
        decorated = string;
    }

    public Object getNode() {
        return decorated;
    }

    public String getNodeName() {
        return "";
    }

    public JDOMNode getParentNode() {
        if (parent == null) {
            throw new RuntimeException(
                "Родительский узел для данного строкового содержимого
                 не установлен!");
        }
        return new ElementNode(parent);
    }

    public String getQName() {
        // Текстовые узлы не имеют имени
        return "";
    }

    public Iterator iterator() {
        return Collections.EMPTY_LIST.iterator();
    }

    public TextNode setParent(Element parent) {
        this.parent = parent;
        return this;
    }

    public String toString() {
        return decorated;
    }
}
```

Я не собираюсь приводить декораторы для всех остальных типов JDOM, потому что читатели уже получили об этом представление. Обратите внимание, что так же можно было создать реализацию JDOMNode с именем вроде ConcreteNode, которая интегрировала бы в одном классе различные типы JDOM. Однако для этого потребовалось бы написать особый код с проверкой условий, который здесь неуместен. Вместо этого я обеспечил соответствие один к одному между основными классами JDOM и реализациями JDOMNode.

Обеспечение поддержки для Xpath

Теперь, когда у нас есть несколько интерфейсных узлов JDOM, продвинемся еще немного дальше. Распространенный сценарий – необходимо обеспечить особую функциональность на основе существующего API. В качестве практического примера рассмотрим XPath. Для любой реализации JDOMNode было бы хорошо иметь возможность получить выражение Xpath, представляющее этот узел. Для реализации этой функциональности я написал еще один класс-обертку, показанный в примере 8.9. Этот класс – XpathDisplayNode – инкапсулирует существующий узел (любого типа, благодаря логике, основанной на интерфейсах) и обеспечивает единственный открытый метод XPath, getXPath(). Этот метод возвращает выражение XPath для узла в виде строки символов.

Пример 8.9. Обложка для поддержки Xpath

```
package javaxml2;

import java.util.Vector;
import java.util.List;
import java.util.Iterator;
import java.util.Stack;

// Импортируем JDOM
import org.jdom.Attribute;
import org.jdom.Element;
import org.jdom.Namespace;

public class XPathDisplayNode {

    /** Узел JDOMNode, на котором основано это выражение xpath */
    JDOMNode node;

    public XPathDisplayNode(JDOMNode node) {
        this.node = node;
    }

    private String getElementXPath(JDOMNode currentNode) {
        StringBuffer buf = new StringBuffer("/")
            .append(currentNode.getQName());
        Element current = (Element)currentNode.getNode();
```

```
        Element parent = current.getParent();
        // Проверяем, находимся ли мы в корневом элементе
        if (parent == null ) {
            return buf.toString();
        }

        // Проверяем другие одноуровневые элементы с тем же именем
        // и пространством имен
        Namespace ns = current.getNamespace();
        List siblings = parent.getChildren(current.getName(), ns);

        int total = 0;
        Iterator i = siblings.iterator();
        while (i.hasNext()) {
            total++;
            if (current == i.next()) {
                break;
            }
        }

        // Селектор не нужен, если это единственный элемент
        if ((total == 1) && (!i.hasNext())) {
            return buf.toString();
        }

        return buf.append("[")
                  .append(String.valueOf(total))
                  .append("]").toString();
    }

    public String getXPath() {
        // Обработка элементов
        if (node.getNode() instanceof Element) {
            JDOMNode parent = node.getParentNode();

            // Если значение - null, значит, это корень
            if (parent == null) {
                return "/" + node.getQName();
            }

            // В противном случае создаем путь к корню
            Stack stack = new Stack();
            stack.add(node);
            do {
                stack.add(parent);
                parent = parent.getParentNode();
            } while (parent != null);

            // Создание пути
            StringBuffer xpath = new StringBuffer();
            while (!stack.isEmpty()) {
                xpath.append(getElementXPath((JDOMNode)stack.pop()));
            }
```

```
            return xpath.toString();
        }
        // Обработка атрибутов
        if (node.getNode() instanceof Attribute) {
            Attribute attribute = (Attribute)node.getNode();
            JDOMNode parent = node.getParentNode();
            StringBuffer xpath = new StringBuffer("//")
                .append(parent.getQName())
                .append("[@")
                .append(node.getQName())
                .append("='")
                .append(attribute.getValue())
                .append("']");

            return xpath.toString();
        }

        // Обработка текста
        if (node.getNode() instanceof String) {
            StringBuffer xpath = new StringBuffer(
                new XPathDisplayNode(node.getParentNode()).getXPath())
                    .append("[child::text()]");
            return xpath.toString();
        }

        // Другие типы узлов попадут сюда
        return "Поддержка типа этого узла не реализована.";
    }
}
```

В этом классе обеспечено особое условное ветвление для всех типов узлов. Другими словами, удалось избежать реализации набора классов `XPathElementNode`, `XPathAttributeNode` и т. д. Дело в том, что преимущества от использования похожести действий, приводящих к созданию операторов XPath для узлов различных типов, гораздо больше, чем от разбиения кода на реализации по типам. Конечно же, это полная противоположность предоставлению декоратора узла для каждого типа JDOM. Необходимо всегда пытаться оценивать различия в своих приложениях, это позволяет создавать более прозрачный и более лаконичный код.

Подробности процесса, рассмотренного в этом коде, оставлены читателям для самостоятельного изучения. Выражение XPath для любого узла вычисляется и собирается вручную, и вы должны без труда понять всю логику. Затем это выражение возвращается вызывающей программе, которую мы рассмотрим далее.

Эндшпиль

Теперь, когда у нас есть все типы узлов, а также обертка XPath, пришло время сделать что-нибудь полезное. В данном случае это будет

программа просмотра документа, подобная классу SAXTreeViewer из главы 3, для дерева JDOM. Однако также будет представлено выражение XPath для каждого элемента из этого дерева в строке состояния. В примере 8.10 показано, как это сделать при помощи узлов и оберток, рассмотренных в этом разделе.

Пример 8.10. Класс SimpleXPathViewer

```
package javaxml2;

import java.awt.*;
import java.io.File;
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.util.Iterator;

// Импортируем JDOM
import org.jdom.*;
import org.jdom.input.SAXBuilder;

public class SimpleXPathViewer extends JFrame {

    /** Базовый класс обработчика событий */
    EventHandler eventHandler = new EventHandler();

    /** Текстовое поле для отображения пути XPath выбранного узла */
    private JTextField statusText;

    /** Дерево Jtree, используемое для отображения узлов документа XML */
    private JTree jdomTree;

    /** Модель выбора, используемая для определения того, какой узел был
    выбран */
    private DefaultTreeSelectionModel selectionModel;

    /** Имя XML-файла для просмотра */
    private String filename;

    /** Временная хитрость, предназначенная для того, чтобы обойти отсутствие
    текстовых узлов */
    private static Element lastElement;

    class EventHandler implements TreeSelectionListener {

        public void valueChanged(TreeSelectionEvent e) {
            TreePath path= selectionModel.getLeadSelectionPath();

            // Если вы только сворачиваете дерево, то вы можете не иметь
            // нового пути
            if (path != null) {
              JDOMNode selection=
             (JDOMNode)((DefaultMutableTreeNode)path.getLastPathComponent())
                 .getUserObject();
```

```
            buildXPath(selection);
        }
    };
};
public SimpleXPathViewer(String fileName) throws Exception {
    super();
    this.filename = fileName;
    setSize(600, 450);
    initialize();
}

private void initialize() throws Exception {
    setTitle("Простая программа просмотра XPath");

    // Настраиваем пользовательский интерфейс
    initConnections();

    // Загружаем документ JDOM
    Document doc = loadDocument(filename);

    // Создаем и инициализируем JDOMNode методом фабрики
    JDOMNode root = createNode(doc.getRootElement());

    // Создаем корневой узел дерева JTree и создаем дерево на основе
    // документа JDOM
    DefaultMutableTreeNode treeNode =
        new DefaultMutableTreeNode("Документ: " + filename);
    buildTree(root, treeNode);

    // Добавляем узел в модель дерева
    ((DefaultTreeModel)jdomTree.getModel()).setRoot(treeNode);
}

private void initConnections() {
 setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);

    // Настройка дерева JTree и панели для его отображения
    jdomTree = new JTree();
    jdomTree.setName("Дерево JDOM");
    jdomTree.addTreeSelectionListener(eventHandler);
    selectionModel =
           (DefaultTreeSelectionModel)jdomTree.getSelectionModel();
   getContentPane().add(new JScrollPane(jdomTree), BorderLayout.CENTER);

    // Настройка текстовой области, используемой в строке состояния
    statusText = new JTextField("Щелкните на элементе для просмотра пути
                                XPath");
    JPanel statusBarPane= new JPanel();
    statusBarPane.setLayout(new BorderLayout());
    statusBarPane.add(statusText, BorderLayout.CENTER );
    getContentPane().add(statusBarPane, BorderLayout.SOUTH);
}
```

```
private Document loadDocument(String filename) throws JDOMException {
    SAXBuilder builder = new SAXBuilder();
    builder.setIgnoringElementContentWhitespace(true);
    return builder.build(new File(filename));
}

private JDOMNode createNode(Object node) {
    if (node instanceof Element) {
        lastElement = (Element)node;
        return new ElementNode((Element)node);
    }

    if (node instanceof Attribute) {
        return new AttributeNode((Attribute)node);
    }

    if (node instanceof String) {
        return new TextNode((String)node).setParent(lastElement);
    }

    // Поддержка прочих узлов  не реализована
    return null;
}

private void buildTree(JDOMNode node, DefaultMutableTreeNode treeNode) {
    // Если это пустой узел или узел, который не может быть обработан,
    // мы его игнорируем
    if ((node == null) || (node.toString().trim().equals(""))) {
        return;
    }

    DefaultMutableTreeNode newTreeNode = new
    DefaultMutableTreeNode(node);

    // Обходим дочерние элементы узла
    Iterator i = node.iterator();
    while (i.hasNext()) {
        // Создаем узлы JDOMNodes для дочерних элементов и добавляем
        // их в дерево
        JDOMNode newNode = createNode(i.next());
        buildTree(newNode, newTreeNode);
    }

    // После того как все дочерние узлы добавлены, соединяем элемент
    // с деревом
    treeNode.add(newTreeNode);
}

private void buildXPath(JDOMNode node) {
    statusText.setText(new XPathDisplayNode(node).getXPath());
}

public static void main(java.lang.String[] args) {
```

```
        try {
            if (args.length != 1) {
                System.out.println("Использование: java
                          javaxml2.SimpleXPathViewer " + "[имя файла
                          с XML-документом]");
                return;
            }

            /* Создание рамки */
            SimpleXPathViewer viewer= new SimpleXPathViewer(args[0]);

            /* Добавляем windowListener для windowClosedEvent */
            viewer.addWindowListener(new java.awt.event.WindowAdapter() {
                    public void windowClosed(java.awt.event.WindowEvent e) {
                        System.exit(0);
                    };
                });
            viewer.setVisible(true);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Как обычно, пропущены подробности, касающиеся Swing. Обратите внимание, что когда документ загружается при помощи SAXBuilder, мы получаем корневой элемент этого документа (в методе initialize()). Данный элемент используется для создания экземпляра JDOMNode посредством вспомогательной функции createNode(). Последняя просто выполняет преобразование между типами JDOM и реализациями JDOMNode, и на ее написание тратится около 15 с. Воспользуйтесь подобным методом в собственных программах, применяющих декораторы и обертки.

Итак, у нас есть реализации JDOMNode и можно очень просто обойти дерево, создавая для каждого встреченного узла графические объекты. Кроме того, для каждого узла в окне состояния отражается соответствующее выражение XPath. Скомпилируйте все эти примеры и протестируйте их с помощью команды:

```
C:\javaxml2\build>java javaxml2.SimpleXPathViewer
                       c:\javaxml2\ch08\xml\contents.xml
```

Убедитесь, что JDOM и анализатор XML находятся в путях к классам. Результатом выполнения этого примера будет графический интерфейс (UI) Swing, показанный на рис. 8.1. Обратите внимание, что в строке состояния отображается выражение XPath для узла, выбранного в данный момент. Проверьте все сами, четыре или пять снимков экрана в книге не принесут вам такой пользы, как собственные исследования этого инструмента.

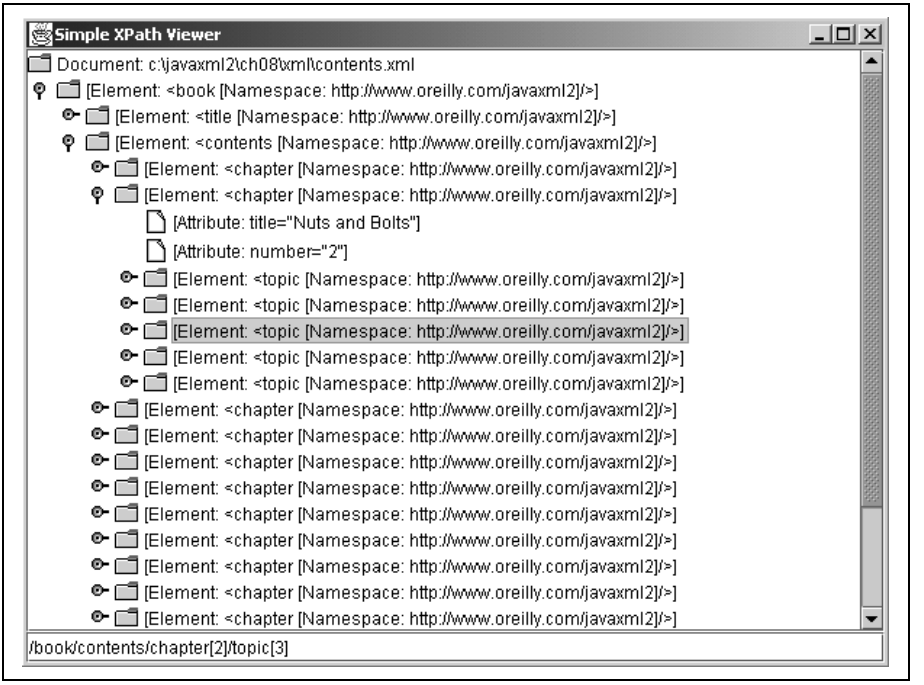

Рис. 8.1. Просмотр файла contents.xml и XPath

Ну вот и все! Да, мы очень быстро «прошлись» по этому материалу, но рассмотренные понятия просты. Подумайте о том, как декораторы и обертки способны помочь в приложениях, для которых требуются функциональные возможности интерфейсов. Также посетите веб-сайт JDOM по адресу *http://www.jdom.org*, чтобы узнать, не появились ли новые наборы оберток (подобных этой или набору декораторов для DOM).

Наконец, я хотел бы поблагодарить Филипа Нельсона (Philip Nelson), проделавшего львиную долю работы по созданию кода представленного здесь декоратора. Именно он внес ясность в применение декораторов в JDOM и оказал неоценимую помощь в написании этого раздела.

Советы разработчикам

Как и в других главах об API, мы коснемся некоторых наиболее сложных вопросов, имеющих отношение к теме этой главы. Все это распространенные проблемы, которые могут заставить вас буквально биться головой об стену, так что постарайтесь избежать их.

Еще немного о подклассах

Поскольку в этой главе говорилось о фабриках и пользовательских классах, то стоит указать на несколько важных моментов в создании

производных классов, которые могут стать ловушками. При расширении класса, а в особенности классов JDOM, необходимо убедиться, что написанная вами функциональность будет применяться так, как вы задумали. Другими словами, убедитесь, что для приложения не существует способа использовать суперкласс способом, который вы не одобряете. Практически во всех случаях это означает переопределение всех конструкторов суперкласса. Можно заметить, что в примере 8.1 – в классе `ORAElement` – переопределены все четыре конструктора класса `Element`. Это гарантирует, что любое приложение, использующее `ORAElement`, должно будет создавать объект при помощи одного из этих конструкторов. И хотя это может показаться тривиальной деталью, представьте себе, что пропущен конструктор, принимающий имя и URI элемента. По существу, этот шаг уменьшает на единицу количество способов, которыми можно создать объект. Это может показаться тривиальным, но это не так!

Продолжая этот гипотетический пример, представим, что вы реализуете класс `CustomJDOMFactory` подобно классу из примера 8.2 и переопределяете различные методы `element()`. Однако можно забыть переопределить метод `element(String name, String uri)` так же, как мы забыли переопределить конструктор в подклассе. Вот вам и неожиданная неприятность. Каждый раз, когда элемент будет запрашиваться по имени и URI (что, кстати, довольно часто происходит в `SAXBuilder`), мы ожидаем, что получим простой, обычный экземпляр элемента. Но все остальные методы создания элементов возвращают экземпляры `ORAElement`. Из-за одного пропущенного конструктора документ будет иметь две реализации элементов, что почти наверняка не то, чего вам хотелось. Очень важно проверять все способы создания объектов в подклассах и (обычно) гарантировать переопределение всех открытых конструкторов в суперклассе.

Создание недействительного XML-кода

Еще один сложный случай, который необходимо отслеживать при создании подклассов, – это непреднамеренное создание недействительного XML-кода. При использовании JDOM практически невозможно создать XML-код, который не будет корректным, но снова представьте подкласс `ORAElement`. Этот подкласс добавлял префикс `ora` к каждому элементу, и это могло привести к созданию недействительного кода. Вероятно, это не так уж и важно, но чтобы избежать проблем при чтении документа, необходимо либо закомментировать, либо удалить объявление `DOCTYPE`.

Еще более важно то, что можно получить неожиданные результаты, если не быть внимательным. Взгляните на фрагмент XML-кода, сгенерированного при помощи подкласса `ORAElement`, в котором показана лишь последняя небольшая часть сериализованного документа:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book SYSTEM "DTD/JavaXML.dtd">
<!-- "Java и XML", Содержание -->
<ora:book xmlns:ora="http://www.oreilly.com">
  <ora:title ora:series="Java">Java и XML</ora:title>

  <!-- Остальное содержимое -->

  <ora:copyright>

<ora:copyright>
  <ora:year value="2001" />
  <ora:content>All Rights Reserved, O'Reilly &amp; Associates</ora:content>
</ora:copyright>
</ora:copyright>
</ora:book>
```

Обратите внимание, что теперь элементов `ora:copyright` два! Произошло следующее: существующий элемент находился в пространстве имен O'Reilly (первый элемент `ora:copyright`). Однако элементу `copyright`, вложенному в него и не имеющему пространства имен, также был присвоен префикс `ora` пространства имен O'Reilly через класс `ORAElement`. В результате появились два элемента с одинаковым именем и пространством имен, но с разными моделями содержимого. Это очень сильно усложняет проверку действительности и, вероятно, является не тем, чего вы добивались. Это были простые примеры, а в более сложных документах с более сложными подклассами придется тщательно следить за генерируемыми результатами, особенно если это касается DTD, схемы XML или иной формы ограничений документа.

Что дальше?

Мы заканчиваем с низкоуровневыми API и в следующей главе завершим этот разговор, рассмотрев JAXP – Java API для обработки XML (от Sun). JAXP требует использования SAX и DOM, и потому логично, что он похож на них. Следующая глава разделяет книгу на две части, и, вероятно, после того как вы ее осилите, стоит немного передохнуть! Я рассмотрю обе широко используемые сейчас версии JAXP: 1.0 и 1.1; объясню, как они работают с API, которые мы уже изучили; а также расскажу, какую помощь JAXP реально может оказать при создании приложений. Так что приготовьтесь добавить еще один инструмент в свою коллекцию и переворачивайте страницу.

- *API или абстракция*
- *JAXP 1.0*
- *JAXP 1.1*
- *Советы разработчикам*
- *Что дальше?*

9 JAXP

Когда компания Sun представила Java API для обработки XML (JAXP), она ухитрилась вызвать целый ряд противоречий в мире Java. Одним махом компания выпустила самый важный интерфейс, не являющийся интерфейсом для разработчиков на Java, и вызвала беспорядок с самым простым API. Люди переходили на новый анализатор, не зная об этом. JAXP окружает много неразберихи, связанной не только со способами применения технологии, но и с ее сутью.

В этой главе сначала мы коснемся вопроса о том, чем является, а чем не является JAXP.[1] Затем рассмотрим JAXP 1.0, который по-прежнему активно используется. После этого перейдем к JAXP 1.1, последней на момент написания книги версии API.[2] Это поможет вам освоиться с новыми возможностями из последней версии, в особенности с TrAX API, включенным в JAXP 1.1. Пристегнитесь и подготовьтесь до конца понять тайну JAXP.

API или абстракция

Перед тем как переходить к коду, очень важно изучить базовые понятия. Собственно говоря, JAXP – это API, но более точно было бы назвать его *уровнем абстракции* (*abstraction layer*). Он не предоставляет новых способов анализа XML, добавленных к SAX, DOM или JDOM, и не предоставляет новых возможностей для обработки XML в Java.

---

[1] Если эта глава вызывает у вас ощущение dïja vu, то вы должно быть читали более ранние версии этого текста в «IBM DeveloperWorks». Первоначально это были две статьи, опубликованные на сайте *http://www.ibm.com/developer*, посвященном JAXP. Данная глава представляет собой обновленную и несколько измененную версию этих статей.

[2] Уже вышла версия JAXP 1.2. – *Примеч. науч. ред.*

Вместо этого он упрощает работу с некоторыми сложными задачами в DOM и SAX. Также он позволяет решать специфичные для различных платформ задачи, возникающие при использовании интерфейсов DOM и SAX, что, в свою очередь, позволяет использовать эти API не зависящим от платформы способом.

И хотя мы рассмотрим все эти возможности по отдельности, очень важно понять, что JAXP не обеспечивает функциональности для анализа! Без SAX, DOM или иного API для анализа XML, анализировать *XML невозможно*. Я видел массу попыток сравнения DOM, SAX или JDOM с JAXP. Подобное сравнение просто невозможно, поскольку первые три интерфейса служат совершенно иной цели, нежели JAXP. SAX, DOM и JDOM анализируют XML. JAXP предоставляет способ доступа к этим API и работы с результатами анализа документа. Сам по себе он не дает нового способа анализировать документ. Это очень важное отличие, которое необходимо понять, для того чтобы корректно применять JAXP. Также это, вероятнее всего, позволит вам сильно опередить других разработчиков.

Те, кто по-прежнему сомневается, могут загрузить дистрибутив JAXP 1.0 с веб-сайта Sun *http://java.sun.com/xml* и получить представление о том, насколько прост JAXP. В архиве *jaxp.jar* вы обнаружите лишь шесть классов! Насколько сложным может быть этот интерфейс? Все классы (часть пакета `javax.xml.parsers`) строятся на существующем анализаторе, а два из них предназначены для обработки ошибок. JAXP проще, чем можно подумать.

JAXP и анализатор от Sun

Часть неприятностей возникает из-за того, что в дистрибутив JAXP входит анализатор Sun. Классы анализатора находятся в архиве *parser.jar* как часть пакета `com.sun.xml.parser` и связанных с ним подпакетов. Этот анализатор (с кодовым названием Crimson) не является частью JAXP. Он представляет собой часть дистрибутива JAXP, но не является частью JAXP *API*. Странно? Отчасти. Подумайте об этом так: в дистрибутив JDOM включен анализатор Apache Xerces. Этот анализатор не является частью JDOM, но он включен для того, чтобы с JDOM можно было работать. Те же принципы применимы и для JAXP, но об этом не настолько хорошо известно: JAXP поставляется с анализатором от Sun, и потому им можно пользоваться сразу же. Однако многие считают классы, включенные в анализатор от Sun, частью самого JAXP API. Например, в группах новостей часто встречается такой вопрос: «Как я могу применять класс `XMLDocument`, поставляемый с JAXP? Каково его назначение?» Ответить на этот вопрос непросто.

Во-первых, класс `com.sun.xml.tree.XMLDocument` *не является частью JAXP*. Это часть анализатора от Sun. Поэтому вопрос изначально по-

ставлен некорректно. Во-вторых, основная цель JAXP – предоставить независимость от производителя при работе с анализаторами. Один и тот же код, использующий JAXP, будет работать с XML-анализаторами Sun, Apache Xerces и Oracle. Поэтому применение класса, специфичного для Sun, является не лучшей идеей. Оно противоречит самой идее применения JAXP. Видите, как все сплетается: анализатор и API из дистрибутива JAXP (по крайней мере, в дистрибутиве от Sun) объединены, и разработчики ошибочно принимают возможности и классы из одной части за возможности и классы из другой и наоборот.

Старое и новое

С JAXP связан еще один сбивающий с толку момент. JAXP 1.0 поддерживает только SAX 1.0 и DOM Level 1. Это обычная политика Sun – не выпускать API или продукты, основанные на черновых, бета- или иных неокончательных версиях API. Когда вышла окончательная версия JAXP 1.0, Sun все так же остановились на SAX 1.0, поскольку SAX 2.0 все еще находился в состоянии бета-версии, и DOM Level 1, поскольку версия Level 2 еще не была принята в окончательном виде. Многие пользователи применяли JAXP с существующими анализаторами (такими как Apache Xerces, например), поддерживающими SAX 2.0 и DOM Level 2, и их программы внезапно начинали работать не так, как задумано. Разумеется, это служило источником вопросов о том, какой толк от возможностей, которые просто нельзя использовать с JAXP. Примерно в то же время SAX 2.0 был принят в окончательном виде, и это окончательно все запутало. Однако это не остановило многих, кому не нужны были последние версии DOM и SAX, и JAXP 1.0 применяется по-прежнему, так что я был бы недобросовестным, если бы не рассмотрел обе версии: и старую (1.0), и новую (1.1), поддерживающую SAX 2.0 и DOM Level 2. Оставшаяся часть главы разделена на две части: первая посвящена JAXP 1.0, а вторая – 1.1. Поскольку версия 1.1 в том, что касается функциональности, строится на версии 1.0, то необходимо прочитать оба раздела, независимо от того, с какой версией API вы работаете.

JAXP 1.0

Все начинается (и началось) с JAXP 1.0. Эта первая версия API по сути предоставляла тонкий уровень абстракции для существующих API и позволяла анализировать код способом, не зависящим от платформы. Для SAX это не такое уж и преимущество; теперь читатели уже стали экспертами в SAX и понимают, что можно воспользоваться классом `XMLReaderFactory` вместо того, чтобы непосредственно создавать экземпляр класса анализатора конкретной платформы. Читатели являются экспертами еще и в DOM и, само собой, понимают, что работать с DOM способом, не зависящим от платформы, достаточно мучительно, так

что в этом отношении JAXP помогает достаточно хорошо. Кроме того, JAXP предоставляет несколько методов для работы с проверкой действительности и пространствами имен, а это еще одна задача, которая решается обычно в терминах конкретных платформ. Теперь ее (в большинстве случаев) можно решать более эффективно.

Начинаем работать с SAX

Перед тем как рассмотреть комбинацию JAXP/SAX, рассмотрим некоторые подробности, связанные с SAX 1.0. Помните класс `org.xml.sax.helpers.DefaultHandler`, представленный в главе 4, который реализовывал все основные обработчики из SAX 2.0? В SAX 1.0 был подобный класс под названием `org.xml.sax.HandlerBase`; этот класс реализовывал обработчики SAX 1.0 (которые немного отличались в этой версии). Те, кто это понимают, готовы к работе с JAXP 1.0.

При использовании JAXP с SAX-совместимым анализатором наша единственная задача заключается в том, чтобы расширить класс `HandlerBase` и реализовать методы обратных вызовов, необходимые для работы приложения. Точно то же делалось для `DefaultHandler` в SAX 2.0. Экземпляр вашего класса расширения затем становится основным аргументом для большинства методов JAXP, имеющих дело с SAX.

Вот краткое описание процесса для SAX:

- Создаем экземпляр `SAXParser` на основе конкретной реализации анализатора.
- Регистрируем реализации методов обратных вызовов (используя класс, расширяющий `HandlerBase`).
- Начинаем процесс анализа и отдыхаем, пока выполняются реализации методов обратных вызовов.

Компонент SAX в JAXP предоставляет простой способ, позволяющий сделать все это. Без JAXP необходимо либо непосредственно создать экземпляр анализатора SAX из класса производителя (например, `org.apache.xerces.parsers.SAXParser`), либо при помощи вспомогательного класса SAX под названием `ParserFactory` (версия SAX 1.0 класса `XMLReaderFactory` из SAX 2.0).

JAXP предусматривает лучшую альтернативу. Он позволяет воспользоваться конкретным классом платформы для создания анализатора посредством системного свойства Java. Конечно же, при загрузке дистрибутива от Sun вы получаете реализацию JAXP, использующую по умолчанию анализатор Sun. Те же интерфейсы JAXP, но реализованные на основе Apache Xerces, можно загрузить с веб-сайта Apache *http://xml.apache.org*; они по умолчанию используют Apache Xerces. Следовательно (в обоих случаях), смена анализатора требует изменения настроек путей к классам либо изменения системного значения,

но не требует перекомпиляции кода. Именно в этом и заключается волшебство абстракции JAXP.

Внимание

Очень важно, откуда вы взяли классы JAXP. И пусть даже можно устанавливать системные свойства для изменения класса анализатора, анализатор по умолчанию (когда системных свойств нет) зависит от реализации, которая, в свою очередь, зависит от места, откуда взят JAXP. Версия от Apache XML по умолчанию пользуется Xerces, а версия от Sun – Crimson. Если их перепутать, можно получить неверный анализатор в путях к классам и исключение `ClassNotFound`.

Взгляд на класс SAXParserFactory

Класс JAXP `SAXParserFactory` (из класса `javax.xml.parsers`, как и все классы JAXP) является ключом к легкой смене реализаций анализатора. Необходимо создать новый экземпляр этого класса (как это сделать, мы скоро узнаем). После того как фабрика создана, она предоставляет метод для получения SAX-совместимого анализатора. За кулисами реализация JAXP работает с кодом, специфичным для платформы, оставляя ваш код чистым. Эта фабрика также предоставляет и другие полезные возможности.

Помимо создания экземпляров анализаторов SAX, фабрика позволяет устанавливать параметры настройки, которые влияют на все экземпляры анализаторов, получаемые посредством этой фабрики. В JAXP 1.0 доступны два параметра: поддержка пространств имен (`setNamespaceAware (boolean awareness)`) и проверка действительности (`setValidating (boolean validating)`). Запомните, что после того как эти параметры установлены, они влияют на все экземпляры, получаемые впоследствии от фабрики.

Когда фабрика настроена, вызов метода `newSAXParser()` возвращает готовый к применению экземпляр класса JAXP `SAXParser`. Этот класс представляет собой обертку для применяемого анализатора SAX (экземпляра класса SAX `org.xml.sax.Parser`). Также он защищает от использования любых расширений класса анализатора, специфичных для платформы. (Помните наш разговор о классе `xmlDocument`?) Этот класс позволяет инициировать собственно процесс анализа. В примере 9.1 показано, как создать, настроить и использовать фабрику SAX.

Пример 9.1. Использование класса SAXParserFactory

```
package javaxml2;

import java.io.File;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
```

```
// JAXP
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;

// SAX
import org.xml.sax.AttributeList;
import org.xml.sax.HandlerBase;
import org.xml.sax.SAXException;

public class TestSAXParsing {

    public static void main(String[] args) {
        try {
            if (args.length != 1) {
                System.err.println(
                    "Использование: java TestSAXParsing [имя файла
                     XML-документа]");
                System.exit(1);
            }

            // Создаем фабрику SAX анализатора
            SAXParserFactory factory = SAXParserFactory.newInstance();

            // Включение проверки действительности и отключение пространств
            // имен
            factory.setValidating(true);
            factory.setNamespaceAware(false);

            SAXParser parser = factory.newSAXParser();
            parser.parse(new File(args[0]), new MyHandler());

        } catch (ParserConfigurationException e) {
            System.out.println("Анализатор не поддерживает " +
                               "запрашиваемые возможности.");
        } catch (FactoryConfigurationError e) {
            System.out.println(
                "При настройке фабрики SAX-анализатора произошла ошибка.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class MyHandler extends HandlerBase {
    // Реализации методов обратных вызовов SAX из DocumentHandler,
    // ErrorHandler, DTDHandler и EntityResolver
}
```

Обратите внимание, что при работе с фабрикой могут возникнуть две проблемы, специфичные для JAXP: ошибки при создании или на-

стройке фабрики SAX и ошибки при настройке анализатора SAX. Первая из этих проблем представлена исключением FactoryConfigurationError и обычно возникает тогда, когда невозможно загрузить анализатор, указанный в реализации JAXP или системном свойстве. Второе исключение – ParserConfigurationException – возникает тогда, когда запрошенная возможность недоступна в применяемом анализаторе. Обе ситуации легко обработать, и они не будут представлять никаких сложностей.

Объект SAXParser мы создаем после того, как создана фабрика, отключены пространства имен и включена проверка действительности; затем начинается процесс анализа. Обратите внимание, что метод parse() анализатора SAX принимает экземпляр класса SAX HandlerBase, о котором мы говорили раньше (реализация класса не приведена в коде примера, но можно взять полный исходный файл TestSAXParsing.java с веб-сайта книги). Очевидно, что методу также передается файл (объект Java File) для анализа. Однако класс SAXParser не ограничивается одним методом.

Работа с классом SAXParser

Созданный экземпляр SAXParser можно использовать не только для анализа файла. Компоненты крупных приложений взаимодействуют таким образом, что не всегда безопасно считать создателя объекта его пользователем. Другими словами, один компонент может создать экземпляр SAXParser, тогда как другому компоненту (вероятно, написанному другим разработчиком) может потребоваться использовать тот же экземпляр. Поэтому существуют методы для определения настроек экземпляра анализатора. В частности, это метод isValidating(), который позволяет определить, будет ли анализатор проверять действительность, и isNamespaceAware(), который позволяет определить, будет ли анализатор обрабатывать пространства имен в XML-документе. И хотя методы могут предоставить информацию о возможностях анализатора, нет способа изменить эти возможности. Делать это следует на уровне фабрики анализатора.

Кроме того, существует несколько способов запросить анализ документа. Метод parse() класса SAXParser может работать не только с файлом и экземпляром SAX HandlerBase, но также с экземплярами SAX InputSource, Java InputStream либо URL в виде строки; при этом вторым аргументом все так же является экземпляр HandlerBase. Различные типы входных документов можно анализировать различными способами.

Наконец, применяемый анализатор SAX (экземпляр org.xml.sax.Parser) можно получить и использовать непосредственно с помощью метода SAXParser getParser(). Для этого экземпляра доступны стандартные методы SAX. В примере 9.2 показаны различные способы использования класса SAXParser, основного класса JAXP для SAX-анализа.

Пример 9.2. Использование класса SAXParser

```
// Получаем экземпляр анализатора SAX
SAXParser saxParser = saxFactory.newSAXParser();

// Проверяем, поддерживается ли проверка действительности
boolean isValidating = saxParser.isValidating();

// Проверяем, поддерживаются ли пространства имен
boolean isNamespaceAware = saxParser.isNamespaceAware();

// ------- Анализируем различными способами ----------------- //

// Используем файл и экземпляр SAX HandlerBase
saxParser.parse(new File(args[0]), myHandlerBaseInstance);

// Используем экземпляры SAX InputSource и SAX HandlerBase
saxParser.parse(mySaxInputSource, myHandlerBaseInstance);

// Используем InputStream и экземпляр SAX HandlerBase
saxParser.parse(myInputStream, myHandlerBaseInstance);

// Используем URI и экземпляр SAX HandlerBase
saxParser.parse("http://www.newInstance.com/xml/doc.xml",
                myHandlerBaseInstance);

// Получаем применяемый анализатор SAX
org.xml.sax.Parser parser = saxParser.getParser();

// Используем его
parser.setContentHandler(myContentHandlerInstance);
parser.setErrorHandler(myErrorHandlerInstance);
parser.parse(new org.xml.sax.InputSource(args[0]));
```

Я уже долго говорю о SAX, но пока не сказал ничего выдающегося или даже удивительного. Дело в том, что возможности JAXP очень ограничены, особенно если речь идет о SAX. Для меня это хорошо (и должно быть хорошо для вас), поскольку сокращение функциональности повышает переносимость кода и дает другим разработчикам возможность использовать его либо бесплатно (в случае свободно распространяемого ПО), либо на коммерческой основе, с любым SAX-совместимым XML-анализатором. Вот и все. Больше нечего сказать о применении SAX в JAXP. Если вы уже знаете SAX, то проделали большую часть пути. Осталось лишь изучить два новых класса и пару исключений Java, чтобы подготовиться к работе. Если вы никогда не работали с SAX, то начать сейчас будет достаточно просто.

Работаем с DOM

Процесс использования JAXP с DOM практически идентичен применению JAXP с SAX; необходимо лишь изменить имена двух классов и тип возвращаемого значения одного из методов. У тех, кто понимает,

как работает SAX и что собой представляет DOM, проблем не будет. (Конечно, всегда можно обратиться к главам 5 и 6.) JAXP не должен выполнять обратные вызовы SAX при работе с DOM, поэтому он отвечает лишь за предоставление объекта DOM `Document` после завершения анализа.

Фабрика DOM-анализатора

Если понимание основ DOM и различий между DOM и SAX достигнуто, то остается сказать не много. Код примера 9.3 выглядит во многом подобно коду для SAX из примера 9.1. Первым делом мы получаем экземпляр `DocumentBuilderFactory` (таким же образом, как и экземпляр `SAXParserFactory` в SAX). Затем фабрика настраивается на проверку действительности и обработку пространств имен (таким же способом, как это было сделано для SAX). Затем из этой фабрики мы получаем `DocumentBuilder`, DOM-аналог `SAXParser`. После этого можно выполнить анализ; полученный объект DOM `Document` передается экземпляру класса `DOMSerializer` (из главы 5).

Пример 9.3. Использование класса DocumentBuilderFactory

```
package javaxml2;

import java.io.File;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;

// JAXP
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;

// DOM
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class TestDOMParsing {

    public static void main(String[] args) {
        try {
            if (args.length != 1) {
                System.err.println (
                    "Использование: java TestDOMParsing [имя файла]");
                System.exit(1);
            }

            // Получаем фабрику Document Builder
```

```
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();

            // Включаем проверку действительности и отключаем пространства
            // имен
            factory.setValidating(true);
            factory.setNamespaceAware(false);

            DocumentBuilder builder = factory.newDocumentBuilder();
            Document doc = builder.parse(new File(args[0]));

            // Сериализация дерева DOM
            DOMSerializer serializer = new DOMSerializer();
            serializer.serialize(doc, System.out);

        } catch (ParserConfigurationException e) {
            System.out.println("Анализатор не поддерживает " +
                "запрошенные возможности.");
        } catch (FactoryConfigurationError e) {
            System.out.println("При получении фабрики Document " +
                "Builder произошла ошибка.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

При работе этого кода могут возникать исключения `FactoryConfigurationError` и `ParserConfigurationException`. Причины их возникновения те же, что и в случае SAX. Либо проблема связана с классом реализации (`FactoryConfigurationError`), либо анализатор не поддерживает запрошенные возможности (`ParserConfigurationException`). Единственное отличие между DOM и SAX заключается в том, что в DOM необходимо заменить `SAXParserFactory` на `DocumentBuilderFactory` и `SAXParser` на `DocumentBuilder`.

Работаем с DOM-анализатором

После того как фабрика DOM получена, можно создать с ее помощью экземпляр `DocumentBuilder`. Методы экземпляра `DocumentBuilder` сходны с методами его SAX-аналога. Основное различие заключается в том, что метод `parse()` не умеет работать с экземплярами класса SAX `HandlerBase`. Вместо этого он возвращает экземпляр DOM `Document`, представляющий собой проанализированный XML-документ. И еще одно отличие связано с парой методов, обеспечивающих функциональность, подобную существующей в SAX. Это `setErrorHandler()`, посредством которого можно указать реализацию SAX `ErrorHandler` для обработки ошибок, которые могут возникнуть при анализе, и `setEntityResolver()`, позволяющий указать реализацию SAX `EntityResolver` для интерпретации сущностей. В примере 9.4 иллюстрируется практическое использование этих методов.

Пример 9.4. Использование JAXP DocumentBuilder

```
// Получаем экземпляр DocumentBuilder
DocumentBuilder builder = builderFactory.newDocumentBuilder();

// Проверяем, поддерживается ли проверка действительности
boolean isValidating = builder.isValidating();

// Проверяем, поддерживаются ли пространства имен
boolean isNamespaceAware = builder.isNamespaceAware();

// Устанавливаем обработчик SAX ErrorHandler
builder.setErrorHandler(myErrorHandlerImpl);

// Устанавливаем SAX EntityResolver
builder.setEntityResolver(myEntityResolverImpl);

// ------------ Анализируем различными способами ------------------- //

// Используем файл
Document doc = builder.parse(new File(args[0]));

// Используем SAX InputSource
Document doc = builder.parse(mySaxInputSource);

// Используем InputStream
Document doc = builder.parse(myInputStream, myHandlerBaseInstance);

// Используем URI
Document doc = builder.parse("http://www.newInstance.com/xml/doc.xml");
```

Да, применить к DOM то, что вы узнали о SAX, действительно настолько просто. Так что можете заключать пари с друзьями и коллегами, утверждая, что использовать JAXP очень просто. Будете постоянно выигрывать.

Смена анализатора

Последняя тема, которой необходимо коснуться, говоря о JAXP, связана с возможностью легко менять анализатор, используемый классами фабрик. Смена анализатора, используемого JAXP, на самом деле означает смену фабрики анализатора, поскольку экземпляры `SAXParser` и `DocumentBuilder` создаются посредством фабрик. Какой именно анализатор будет загружен, определяется фабриками, следовательно, и менять нужно именно их. Применяемую реализацию класса `SAXParserFactory` можно изменить, установив системное свойство Java `javax.xml.parsers.SAXParserFactory`. Если это свойство не определено, то возвращается реализация по умолчанию (анализатор, указанный производителем системы). Такой же принцип применим и к реализации `DocumentBuilderFactory`. В данном случае запрашивается системное свойство `javax.xml.parsers.DocumentBuilderFactory`. Вот так просто мы и рассмотрели все! И это все, что касается JAXP 1.0: создайте связи с SAX, создайте связи с DOM и меняйте анализаторы как перчатки.

JAXP 1.1

В конце 2000 года был создан комитет для работы над JAXP 1.1, и началась работа по переходу от JAXP 1.0 к лучшему, более эффективному решению для работы с XML-документами. Во время написания этой главы версия JAXP 1.1 была выложена в окончательном виде на веб-сайте Sun *http://java.sun.com/xml*. Многие изменения в API связаны с анализом документов, что вполне осмысленно, раз буква «P» в JAXP означает именно анализ (parsing). Но наиболее значимые изменения в JAXP 1.1 касаются преобразований данных XML, которые мы рассмотрим в последней части этой главы. Если говорить о разнице по сравнению с версией 1.0, то изменения довольно несущественны. Самое заметное добавление – это поддержка SAX 2.0, вышедшего в мае 2000 года, и DOM Level 2, вышедшего в ноябре 2000. Помните, что JAXP 1.0 поддерживал лишь SAX 1.0 и DOM Level 1. Отсутствие поддержки обновленных стандартов являлось основной причиной критических отзывов в адрес JAXP 1.0, и, вероятно, именно поэтому версия 1.1 появилась так быстро.

Кроме поддержки последних версий SAX и DOM, в список возможностей API был внесен ряд легких изменений. Практически все они связаны с отзывами, полученными от различных компаний и отдельных членов комитета JAXP. В частности, изменения касаются настройки анализаторов, создаваемых двумя фабриками JAXP: `SAXParserFactory` и `DocumentBuilderFactory`. Сейчас мы рассмотрим их, а также обновления в поддержке стандартов SAX и DOM, а затем перейдем к новому интерфейсу TrAX API, представляющему собой часть JAXP 1.1.

Обновление стандартов

Наиболее желанное изменение в JAXP 1.1 – это поддержка обновленных стандартов SAX и DOM. Очень важно отметить, что в SAX 2.0 появилась возможность работы с пространствами имен (в SAX 1.0 такой возможности нет).[1] Поддержка пространств имен делает возможным использование таких словарей XML, как схемы XML, XLink и XPointer. И хотя в SAX 1.0 работа с ними также была возможна, разработчику приходилось самостоятельно разделять локальное (уточненное) имя элемента и пространство имен и следить за примененим пространств имен в конструкциях документов. SAX 2.0 предоставляет эту инфор-

[1] Внимательные читатели обратят внимание, что JAXP 1.0 обеспечивал обработку пространств имен посредством методов `setNamespaceAware()` фабрик `SAXParserFactory` и `DocumentBuilderFactory`. Код JAXP вынужден был выполнять эту работу «вручную» вместо того, чтобы полагаться на интерфейсы SAX и DOM. Появление SAX 2.0 и DOM Level 2 позволило стандартизировать этот процесс, а значит, и сделать его более надежным и ясным, чем он был в реализации JAXP 1.0. А это хорошая новость.

мацию разработчику, существенно упрощая программирование. То же самое относится и к DOM Level 2: появилась поддержка пространств имен, а также множество других методов в классах DOM.

Очень приятно то, что эти изменения остаются прозрачными для разработчика, применяющего JAXP. Другими словами, обновления стандартов происходят «автоматически», без участия пользователя. Чтобы обратиться к новым возможностям, достаточно указать совместимый с SAX 2.0 анализатор в `SAXParserFactory` и анализатор, совместимый с DOM Level 2 в `DocumentBuilderFactory`.

Дорога к SAX 2.0

С обновлениями стандартов связано несколько значительных изменений, особенно они касаются SAX. В SAX 1.0 в качестве интерфейса анализатора, реализуемого разработчиками систем и анализаторов XML, применялся интерфейс `org.xml.sax.Parser`. Класс JAXP `SAXParser` предоставлял метод `getParser()` для получения данного класса реализации. Сигнатура этого метода выглядит так:

```
public interface SAXParser {

    public org.xml.sax.Parser getParser();

    // Другие методы
}
```

Однако при переходе от SAX 1.0 к версии 2.0 интерфейс `Parser` устарел и был заменен новым интерфейсом `org.xml.sax.XMLReader` (с ним вы знакомы по предшествующим главам). В результате метод `getParser()` стал бесполезным в плане получения экземпляра класса SAX 2.0 `XMLReader`. Для поддержки этого нового интерфейса к классу JAXP `SAXParser` был добавлен новый метод. Не удивительно, что этот метод называется `getXMLReader()`, и выглядит он следующим образом:

```
public interface SAXParser {

    public org.xml.sax.XMLReader getXMLReader();

    public org.xml.sax.Parser getParser();

    // Другие методы
}
```

Подобным же образом в JAXP 1.0 применялся метод `parse()`, который получал экземпляр класса `HandlerBase` (или подкласса, если быть более точным). Конечно же, в SAX 2.0 класс `HandlerBase` был заменен классом `DefaultHandler`. Для того чтобы привести все методы `parse()` класса `SAXParser` в соответствие с этими изменениями, их дополнили версиями этого же метода, работающими с `DefaultHandler`. Чтобы прочувствовать эту разницу, взгляните на пример 9.5, в котором приведен довольно большой фрагмент интерфейса `SAXParser`.

Пример 9.5. Методы parse( ) интерфейса SAXParser

```
public interface SAXParser {

    // Методы анализа из SAX 1.0
    public void parse(File file, HandlerBase handlerBase);
    public void parse(InputSource inputSource, HandlerBase handlerBase);
    public void parse(InputStream inputStream, HandlerBase handlerBase);
    public void parse(InputStream inputStream, HandlerBase handlerBase,
                      String systemID);
    public void parse(String uri, HandlerBase handlerBase);

    // Методы анализа из SAX 2.0
    public void parse(File file, DefaultHandler defaultHandler);
    public void parse(InputSource inputSource,
                      DefaultHandler defaultHandler);
    public void parse(InputStream inputStream,
                      DefaultHandler defaultHandler);
    public void parse(InputStream inputStream,
                      DefaultHandler defaultHandler,
                      String systemID);
    public void parse(String uri, DefaultHandler defaultHandler);

    // Другие методы

}
```

Может показаться, что обилие методов анализа несколько сбивает с толку, но сложности возникают только при работе с обеими версиями SAX. В случае применения SAX 1.0 мы будем работать с интерфейсом `Parser` и классом `HandlerBase`, и будет очевидно, какие методы нам нужны. Точно так же, при использовании SAX 2.0 будет очевидно, что необходимы методы, получающие экземпляры класса `DefaultHandler` и возвращающие экземпляры класса `XMLReader`. Так что используйте приведенный интерфейс в качестве справочника и не очень переживайте по этому поводу! Существуют и другие изменения API, касающиеся SAX.

Изменения в классах SAX

Чтобы завершить разговор об изменениях возможностей JAXP, необходимо рассмотреть несколько новых методов, доступных пользователям SAX в JAXP. Во-первых, класс `SAXParserFactory` имеет новый метод `setFeature()`. Если вы помните, в JAXP 1.0 класс `SAXParserFactory` допускал настройку экземпляров `SAXParser`, возвращаемых фабрикой. Дополняя методы, уже существующие в версии 1.0 (`setValidating()` и `setNamespaceAware()`), этот новый метод позволяет запрашивать для новых экземпляров анализатора возможности из SAX 2.0. Например, пользователь может запросить возможность *http://apache.org/xml/features/validation/schema*, позволяющую включать и отключать про-

верку действительности по схеме XML. Теперь это можно сделать непосредственно с SAXParserFactory, как показано ниже:

```
SAXParserFactory myFactory = SAXParserFactory.newInstance();

// Включение проверки действительности по схеме XML
myFactory.setFeature(
    "http://apache.org/xml/features/validation/schema", true);

// Теперь получаем экземпляр анализатора с разрешенной проверкой
действительности по схеме
SAXParser parser = myFactory.newSAXParser();
```

Метод getFeature() представляет собой дополнение к методу setFeature() и позволяет запрашивать конкретные возможности. Он возвращает простое логическое значение.

Помимо установки возможностей SAX (при помощи значений «истина» и «ложь») JAXP 1.1 поддерживает установку свойств SAX (при помощи объектных значений). Например, для экземпляра анализатора SAX можно установить свойство *http://xml.org/sax/properties/lexical-handler*, связав его с реализацией интерфейса SAX LexicalHandler. Поскольку свойства, подобные данному лексическому свойству, зависят от анализатора, а не от фабрики (как возможности), метод setProperty() принадлежит классу JAXP SAXParser, а не SAXParserFactory. И так же как в случае с возможностями, в классе SAXParser существует дополняющий метод getProperty(), возвращающий значение, связанное с определенным свойством.

Обновления в DOM

Несколько новых методов появилось и в части JAXP, касающейся DOM. Эти методы были добавлены в существующие классы JAXP для поддержки обоих вариантов DOM Level 2, а также конфигураций, которые применялись за последний год. Не будем рассматривать здесь все эти возможности и соответствующие методы, поскольку многие из них используются лишь в очень специфических ситуациях и не понадобятся в большинстве приложений. За информацией о них обратитесь к последней спецификации JAXP. Теперь, рассмотрев обновления стандартов, изменения в SAX и дополнительные методы DOM, перейдем к наиболее значимому изменению в JAXP 1.1: интерфейсу прикладного программирования TrAX.

TrAX API

До сих пор мы рассматривали изменения в JAXP, касающиеся анализа XML. А сейчас перейдем к преобразованиям XML в JAXP 1.1. Наиболее захватывающее нововведение в последней версии API от Sun связано с преобразованиями XML-документов, не зависящими от платформы. И хотя такая независимость может потеснить определение

JAXP как API для анализа, это очень важная возможность, поскольку в процессорах XSL в настоящее время применяются самые различные методы и способы взаимодействия пользователя и разработчика. На самом деле разница между процессорами XSL, как правило, больше, чем между анализаторами.

Первоначально комитет JAXP предложил создать простой класс `Transform` с несколькими методами, чтобы разрешить задание таблицы стилей и выполнение преобразований. Эта первая попытка оказалась довольно сомнительной, но я рад сообщить, что мы (комитет JAXP) продолжаем работу и непрерывно идем вперед. Скотт Боаг (Scott Boag) и Майкл Кей (Michael Kay), два гуру в процессорах XSL (они работают над Apache Xalan и SAXON, соответственно), в числе многих других разработчиков участвовали в создании TrAX, который поддерживает гораздо более широкий набор параметров и возможностей и предоставляет полную поддержку практически всех видов преобразований XML – и все это под крышей JAXP. Результатом является добавление пакета `javax.xml.transform` и некоторых подпакетов в JAXP API.

Как и часть JAXP, отвечающая за анализ документов, преобразования XML требуют трех основных шагов:

- Создания фабрики `Transformer`
- Создания объекта `Transformer`
- Выполнения преобразований

Работа с фабрикой

Работа с преобразованиями в JAXP связана с применением фабрики, которая представлена классом `javax.xml.transform.TransformerFactory`. Этот класс аналогичен классам `SAXParserFactory` и `DocumentBuilderFactory`, которые были рассмотрены в разделах, посвященных JAXP 1.0 и 1.1. Конечно же, создание экземпляра фабрики – это очень простое дело:

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

Тут нет ничего особенного, просто базовые принципы проектирования фабрик в действии, в сочетании с единственным образцом (singleton pattern).

Когда фабрика создана, для нее можно устанавливать различные параметры. Они повлияют на все экземпляры класса `Transformer` (который рассматривается чуть позже), созданные этой фабрикой. Также при помощи фабрики `TransformerFactory` можно получить экземпляры класса `javax.xml.transform.Templates`. Шаблоны – это более сложное понятие JAXP/TrAX, которое рассматривается в конце главы.

Во-первых, рассмотрим *атрибуты*. Это не атрибуты XML, они скорее похожи на свойства, используемые в SAX. Атрибуты позволяют пере-

давать параметры процессору XSL, которым может быть процессор Apache Xalan, SAXON или Oracle XSL (или, теоретически, любой TrAX-совместимый процессор). Правда, параметры очень сильно зависят от производителя. Как и для анализа в JAXP, существует метод `setAttribute()`, а также метод `getAttribute()`. Так же как и метод-модификатор `setProperty()`, метод `setAttribute()` принимает имя атрибута и объектное значение `Object`. И так же, как и метод `getProperty()`, метод доступа `getAttribute()` принимает имя атрибута и возвращает связанное с ним объектное значение `Object`.

Установка `ErrorListener` – это вторая доступная возможность. `ErrorListener` определен в интерфейсе `javax.xml.transform.ErrorListener` и позволяет перехватывать и программным способом обрабатывать проблемы, возникающие при преобразованиях. Это звучит похоже на `org.xml.sax.ErrorHandler`, и они на самом деле похожи. Интерфейс представлен в примере 9.6.

Пример 9.6. Интерфейс ErrorListener

```
package javax.xml.transform;

public interface ErrorListener {
    public void warning(TransformerException exception)
        throws TransformerException;
    public void error(TransformerException exception)
        throws TransformerException;
    public void fatalError(TransformerException exception)
        throws TransformerException;
}
```

Создав реализацию этого интерфейса, а также три метода обратного вызова и вызвав метод `setErrorListener()` экземпляра `TransformerFactory`, с которым мы работаем, мы будем готовы обработать любые ошибки, возникающие во время преобразования.

Наконец, существует метод для установки и получения интерпретатора URI для экземпляров, созданных фабрикой. Интерфейс, определенный в `javax.xml.transform.URIResolver`, также ведет себя подобно своему аналогу из SAX – `org.xml.sax.EntityResolver`. Интерфейс имеет один метод, представленный в примере 9.7.

Пример 9.7. Интерфейс URIResolver

```
package javax.xml.transform;

public interface URIResolver {
    public Source resolve(String href, String base)
        throws TransformerException;
}
```

Будучи реализованным, этот интерфейс позволяет обрабатывать URI, обнаруженные в таких конструкциях XSL, как `xsl:import` и `xsl:include`.

Возвращая объект Source (который мы рассмотрим очень скоро), при нахождении определенного URI можно сообщать экземпляру, выполняющему преобразование, о том, что необходимо искать указанный документ в различных местах. Например, если найдено включение документа по URI *http://www.oreilly.com/oreilly.xsl*, можно вернуть локальный документ *alternateOreilly.xsl* и избежать необходимости обращаться к сети. Реализации интерфейса URIResolver можно указывать посредством метода setURIResolver() класса TransformerFactory и получать их при помощи метода getURIResolver().

Наконец, когда установлены все параметры, можно создать экземпляр или экземпляры интерфейса Transformer при помощи метода newTransformer() фабрики, как показано ниже:

```
// Создание фабрики
TransformerFactory factory = TransformerFactory.newInstance();

// Настройка фабрики
factory.setErrorResolver(myErrorResolver);
factory.setURIResolver(myURIResolver);

// Создание экземпляра Transformer с заданными возможностями
Transformer transformer =
    factory.newTransformer(new StreamSource("foundation.xsl"));
```

Как видите, методу передается таблица стилей, которая применяется во всех преобразованиях для этого экземпляра Transformer. Другими словами, если необходимо преобразовать документ при помощи таблицы стилей A и таблицы стилей B, то понадобятся два экземпляра Transformer – по одному для каждой таблицы стилей. Однако если необходимо преобразовать несколько документов при помощи одной и той же таблицы стилей (назовем ее таблицей стилей C), то понадобится лишь один экземпляр Transformer, связанный с таблицей стилей C. Не беспокойтесь о классе StreamSource; он рассматривается далее.

Преобразование XML

Если экземпляр Transformer получен, можно приступить к выполнению преобразований XML. Этот процесс состоит из двух основных шагов:

- Указания таблицы стилей XSL
- Выполнения преобразования, после указания XML-документа и конечной цели

Как мы уже видели, первый шаг действительно простейший. Таблицу стилей можно указать при создании экземпляра Transformer посредством фабрики. Местоположение этой таблицы стилей должно быть задано при помощи экземпляра javax.xml.transform.Source (на самом деле экземпляра реализации интерфейса Source). Интерфейс Source, который мы видели в коде примеров, – это средство поиска источника

данных, будь то таблица стилей, документ или иной набор данных. TrAX предоставляет интерфейс Source и три конкретные реализации:

- javax.xml.transform.stream.StreamSource
- javax.xml.transform.dom.DOMSource
- javax.xml.transform.sax.SAXSource

Первая из них, StreamSource, считывает исходные данные с какого-либо устройства ввода/вывода. Существуют конструкторы, позволяющие использовать в качестве ввода InputStream, Reader или строковое представление системного идентификатора. После создания экземпляр StreamSource можно передать Transformer. Вероятно, именно эту реализацию Source вы будете чаще всего применять в своих программах. Она отлично подходит для чтения документа из сети, потока ввода, пользовательского ввода или иного статического представления таблицы стилей XSL.

Следующая реализация Source — DOMSource — предназначена для чтения из существующего дерева DOM. Она предоставляет конструктор, принимающий узел DOM org.w3c.dom.Node, и читает данные из этого узла. Она отлично справляется с передачей существующего дерева DOM для преобразования, если анализ уже произведен и XML-документ хранится в памяти в виде структуры DOM либо если дерево DOM построено программным образом.

SAXSource предназначен для чтения данных от генераторов событий SAX. Эта реализация Source работает либо с экземпляром SAX org.xml.sax.InputSource, либо с экземпляром org.xml.sax.XMLReader и использует события из этих источников. Данная реализация идеальна в ситуациях, когда уже есть обработчик содержимого SAX, а методы для обратных вызовов настроены и должны автоматически вызываться до преобразования.

Когда экземпляр Transformer создан (используемая таблица стилей предоставлена через соответствующий экземпляр Source), можно выполнять преобразование. Метод transform() используется следующим образом:

```
// Создание фабрики
TransformerFactory factory = TransformerFactory.newInstance();

// Настройка фабрики
factory.setErrorResolver(myErrorResolver);
factory.setURIResolver(myURIResolver);

// Создание экземпляра Transformer с заданными возможностями
Transformer transformer =
    factory.newTransformer(new StreamSource("foundation.xsl"));

// Выполнение преобразования для myDocument и печать результатов
transfomer.transform(new StreamSource("asimov.xml"),
                     new StreamResult("results.xml"));
```

Метод transform() принимает два аргумента: реализацию Source и реализацию javax.xml.transform.Result. Вы уже должны видеть симметрию в работе этого механизма и иметь представление о функциональности интерфейса Result. Экземпляр Source предоставляет XML-документ, который должен быть преобразован, а экземпляр Result – конечную цель преобразования. Как и в случае Source, есть три реализации интерфейса Result в TrAX и JAXP:

- javax.xml.transform.stream.StreamResult
- javax.xml.transform.dom.DOMResult
- javax.xml.transform.sax.SAXResult

Класс StreamResult позволяет использовать OutputStream (подобно System.out для упрощения отладки!), файл File, строковый системный идентификатор либо Writer. DOMResult работает с узлом DOM Node, в который выводятся результаты преобразования (вероятно, в виде документа DOM org.w3c.dom.Document), а SAXResult принимает экземпляр SAX ContentHandler, которому передаются обратные вызовы, полученные при обработке преобразованных XML-данных. Все это аналогично реализации интерфейса Source.

Хотя в предыдущем примере показано преобразование из потока в поток, возможна любая комбинация источника и результата. Вот несколько примеров:

```
// Выполняем преобразование для jordan.xml и печатаем результат
transformer.transform(new StreamSource("jordan.xml"),
                      new StreamResult(System.out));

// Преобразуем из SAX и выводим результаты в DOM Node
transformer.transform(new SAXSource(
                          new InputSource(
                              "http://www.oreilly.com/catalog.xml")),
                       new DOMResult(DocumentBuilder.newDocument()));

// Преобразуем из DOM и выводим в файл
transformer.transform(new DOMSource(domTree),
                      new StreamResult(
                          new FileOutputStream("results.xml")));

// Используем собственный источник и цель (JDOM)
transformer.transform(new org.jdom.trax.JDOMSource(myJdomDocument),
                      new org.jdom.trax.JDOMResult(
                          new org.jdom.Document()));
```

TrAX обеспечивает потрясающую гибкость при переходе от различных типов ввода к различным типам вывода и использовании таблиц стилей XSL в различных форматах, таких как файлы, деревья DOM, хранимые в памяти, классы чтения SAX и т. д.

Всякая всячина

Перед тем как закончить разговор о JAXP, следует сказать еще о некоторых составляющих TrAX. Не будем рассматривать их полностью, поскольку они применяются реже, а коснемся их лишь кратко. Во-первых, в TrAX существует интерфейс SourceLocator, входящий в пакет javax.xml.transform. Этот класс в контексте преобразований несет ту же нагрузку, что и класс Locator в контексте анализа SAX: он передает информацию о том, где производились действия. Этот интерфейс чаще всего используется для диагностирования ошибок и выглядит следующим образом:

```
package javax.xml.transform;

public interface SourceLocator {
    public int getColumnNumber();
    public int getLineNumber();
    public String getPublicId();
    public String getSystemId();
}
```

Не буду подробно комментировать этот интерфейс, поскольку он достаточно прозрачен. Однако необходимо знать, что в пакете javax.xml.transform.dom находится интерфейс DOMLocator. Этот интерфейс добавляет метод getOriginatingNode(), возвращающий обрабатываемый узел DOM. Это позволяет достаточно просто обрабатывать ошибки при работе с DOMSource и может пригодиться в приложениях, работающих с деревьями DOM.

Также TrAX предоставляет конкретный класс javax.xml.transform.OutputKeys, определяющий набор констант, используемых в свойствах вывода преобразований. Эти постоянные можно использовать для установки свойств объектов Transformer и Templates. А это приводит нас к последней теме, имеющей отношений к TrAX.

Интерфейс Templates в TrAX применяется тогда, когда в различных преобразованиях желателен один и тот же набор свойств вывода либо когда набор инструкций преобразования можно использовать повторно. Передав экземпляр Source методу newTemplates() фабрики TransformerFactory, мы получаем экземпляр объекта Templates:

```
// Создаем фабрику
TransformerFactory factory = TransformerFactory.newInstance();

// Получаем объект Templates
Templates template = factory.newTemplates(new StreamSource("html.xsl"));
```

Теперь объект template представляет собой скомпилированное представление преобразования, подробно описанное в файле *html.xsl* (в нашем примере это таблица стилей, преобразующая XML в HTML). При-

меняя объект Templates, можно выполнять преобразования по этому шаблону в нескольких потоках приложения, кроме того, достигается некоторая оптимизация – за счет того, что инструкции предварительно скомпилированы. Теперь необходимо сгенерировать экземпляр Transformer, но из объекта Templates, а не из фабрики:

```
// Получаем объект преобразования
Transformer transformer = template.newTransformer();

// Выполняем преобразование
transformer.transform(new DOMSource(orderForm),
                      new StreamResult(res.getOutputStream()));
```

В данном случае нет необходимости передавать методу newTransformer() экземпляр Source, поскольку объект преобразования – это просто набор (уже) скомпилированных инструкций. Начиная с этого момента он ведет себя стандартным образом. В данном примере дерево DOM, представляющее форму заказа, передается для преобразования, обрабатывается при помощи таблицы стилей *html.xsl*, а затем посылается в поток вывода сервлета для отображения. Ловко, не правда ли? Примите за правило использовать объект Templates, если собираетесь применить таблицу стилей более двух раз. Это окупится повышением производительности. Кроме того, каждый раз, когда вы имеете дело с многопоточными приложениями, единственный допустимый вариант – это применение объекта Templates.

Советы разработчикам

Главы про API будут неполными, если не рассказать о некоторых проблемах, с которыми я часто сталкивался либо о которых меня спрашивали. Надеюсь, они помогут сохранить вам время и, может быть, сделают ваш код более устойчивым. Читайте дальше и вы увидите, где люди попадаются в ловушки JAXP.

Анализаторы по умолчанию и реализации JAXP

Этот момент заслуживает повторения: реализация JAXP определяет анализатор, запускаемый по умолчанию. Смена реализации JAXP часто приводит к смене используемого анализатора, если не были установлены системные свойства для JAXP. Как правило, следует изменить при этом пути к классам, иначе будут генерироваться всевозможные исключения ClassNotFoundException.

Чтобы полностью избавить себя от этой проблемы, можно просто установить соответствующее системное свойство JAXP для фабрики анализатора, которую предполагается использовать. И независимо от выбранной реализации мы получим то, что ожидаем. Или, что еще луч-

ше, поместите файл *jaxp.properties* в подкаталог *lib* каталога, в котором установлена система Java.[1] Этот файл может быть предельно простым:

```
javax.xml.parsers.SAXParserFactoryorg.apache.xerces.XercesFactory
```

Изменяя реализацию фабрики, вы изменяете обертку анализатора, возвращаемую вызовом `newSAXParser()`. Не пытайтесь проверить приведенный пример – класс `org.apache.xerces.XercesFactory` не существует; он приведен лишь в качестве примера. Он просто уместился в одну строку блока с кодом!

Возможности фабрик и свойства анализаторов

Разработчики, использующие JAXP, часто путают возможности и свойства. Запомните, что свойства настраивают анализаторы, а возможности – фабрики. Вы бы удивились, узнав, сколько я получил писем, в которых утверждалось, что у отправителя письма «испорченная» версия JAXP, т. к. следующий код не компилируется:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setProperty(
    "http://apache.org/xml/properties/dom/document-class-name",
    "org.apache.xerces.dom.DocumentImpl");
```

Конечно же, это свойство, а значит, его нужно устанавливать для экземпляра `SAXParser`, но не для экземпляра `SAXParserFactory`. Обратное, разумеется, справедливо для установки возможностей для анализаторов:

```
SAXParser parser = factory.newSAXParser();
parser.setFeature("http://xml.org/sax/features/namespaces", true);
```

В любом случае это ошибка пользователя, которую не стоит списывать на загрузку дистрибутива с сайта. Людей, утверждающих, что все методы, кроме некоторых, были получены в отличном состоянии, я, как правило, отсылаю к хорошим книжкам по вводу/выводу. Также это отличный пример ненадлежащего использования документации Javadocs. Я истинно верю в ценность Javadoc.

---

[1] В этом варианте предполагается, что переменная окружения `JAVA_HOME` установлена в значение, равное каталогу, в котором установлен JDK. Так предполагается из-за того, что это хорошая, если не обязательная привычка, которая поможет вам в дальнейшем. В действительности JAXP ищет файл `%JAVA_HOME%/lib/jaxp.properties`.

Что дальше?

Поскольку JAXP – это уровень абстракции над уже рассмотренными в предыдущих главах API, то нет необходимости переходить к «Расширенному JAXP». Кроме того, концепции JAXP не так сложны, чтобы оправдать наличие дополнительной главы. Теперь, когда мы рассмотрели различные «низкоуровневые» Java и XML API, у вас должно быть достаточно инструментов, чтобы начать программировать в XML.

Однако в настоящее время с XML связано многое другое, помимо низкоуровневых интерфейсов прикладного программирования. В дополнение к вертикальным приложениям XML существует ряд высокоуровневых API, построенных на концепциях (и API) из первой половины этой книги, предоставляющих удобства разработчику. Эти более специфические концепции и инструменты разработки являются основой второй половины книги. Главу 10 мы начнем с разговора о системах публикации, обеспечивающих множество украшений для XML. Читайте дальше: в следующей главе вам предстоит стать оформителями.

- *Выбор системы публикации*
- *Установка*
- *Применение системы публикации*
- *XSP*
- *Cocoon 2.0 и выше*
- *Что дальше?*

10

Системы веб-публикации

С этой главы начинается обзор специальных тем, посвященных Java и XML. Мы уже рассмотрели основы применения XML в Java, уделили внимание использованию API SAX, DOM, JDOM и JAXP для работы с XML, а также вопросам использования и создания XML-документов. Теперь, когда читатели понимают, как использовать XML в собственном коде, можно перейти к обсуждению конкретных приложений. В следующих шести главах (главы 10–15) представлены важнейшие приложения XML и, в частности, их реализации в мире технологий Java. Хотя уже существуют тысячи важнейших приложений XML, темы, рассматриваемые в этих главах, постоянно находятся в центре внимания, и в них заложен значительный потенциал для изменения традиционных процессов разработки приложений.

Первая актуальная тема, которую мы рассмотрим, связана с приложением XML, которое вызвало наибольший резонанс среди разработчиков на XML и Java. Речь идет о системе веб-публикации. Хотя я постоянно говорю, что визуализацию содержимого документа, возможно, переоценивают по сравнению со значимостью переносимости данных, которую обеспечивает XML, применение XML для формирования представления данных все же является очень важным. Потребность в визуализации возрастает, когда речь заходит о веб-ориентированных приложениях.

Практически каждое серьезное приложение, которое я видел, либо полностью основано на веб-технологиях либо, как минимум, имеет веб-интерфейс. В то же время, пользователи требуют больше функциональности, а отделы маркетинга – больше гибкости в пользовательском интерфейсе. В результате растет количество веб-оформителей.

Чем больше все меняется, тем больше остается неизменным

Те, кто читал первое издание книги, обнаружат, что большая часть материала по Cocoon в этой главе осталась прежней. Хоть я и обещал, что к этому времени выйдет Cocoon 2, и собирался написать целую главу, посвященную ему, все развивалось не так быстро, как я того ожидал. Стефано Маццокки (Stefano Mazzocchi) – главная движущая сила Cocoon – решил, наконец, закончить учебу (так держать, Стефано!), и в результате разработка Cocoon 2 замедлилась. Основная разработка по-прежнему сосредоточена на Cocoon 1.x, так что ее и следует использовать. Раздел, посвященный Cocoon 2, обновлен и отражает нововведения. Следите за новыми книгами от O'Reilly, посвященными вопросам, связанным с системой Cocoon.

Эта новая профессия отличается от веб-мастера только тем, что в профессиональные обязанности художника не входит знание Perl, ASP, JavaScript и других языков создания сценариев. Весь рабочий день веб-оформителя посвящен созданию, изменению и разработке документов HTML и WML.[1] Быстрые изменения в деловой активности и рыночной стратегии могут требовать полной переработки приложения или сайта с частотой до раза в неделю, и зачастую веб-оформитель вынужден тратить целые дни на переделку сотен HTML-страниц. Хотя каскадные таблицы стилей (Cascading Style Sheets, CSS) помогают в этом деле, поддержание согласованности всех этих страниц требует огромного количества времени. Но даже если считать эту далекую от совершенства ситуацию приемлемой, ни один разработчик захочет тратить свою жизнь на внесение бесконечных изменений в разметку веб-страниц.

С приходом серверных технологий Java эта проблема лишь усугубилась. Оказалось, что разработчикам сервлетов приходится тратить многие часы на переделку операторов `out.println()`, которые отвечают за вывод конструкций HTML, и они обычно исподлобья смотрят в сторону отдела маркетинга, когда выясняется, что изменения в оформлении сайта требуют внесения изменений в код приложения. Есть основания считать, что следствием этой ситуации явилась целая спецификация Java Server Pages (JSP). Однако технология JSP не решает проблемы, она лишь перекладывает головную боль на автора напол-

[1] Языки разметки HTML и WML используются совместно с сопутствующими технологиями, такими как Shockwave Flash. Грамотное применение подобных технологий далеко не тривиально, так что я никоим образом не преуменьшаю роль этих людей.

нения страниц, которому постоянно приходится проявлять осторожность, чтобы по ошибке не внести изменения во внедренный код Java. К тому же, технология JSP не обеспечивает четкого разделения содержимого и представления, т. е. не решает поставленной задачи. Требовалось получить средство создания содержимого, свободного от посторонних данных, а также средство согласованного оформления содержимого – либо в в заданные моменты времени (*статическое создание наполнения*), либо динамически во время выполнения запроса (*динамическое создание наполнения*).

Те, кто участвовал в разработке веб-приложений, наверняка кивнули, узнав знакомую проблему, и есть надежда, что ваши мысли обращены к технологиям XSL и XSLT. Проблема вот в чем: должна существовать система для управления созданием наполнения документов, в особенности, если речь идет о динамическом создании наполнения. Бессмысленно иметь сайт, состоящий из сотен XML-документов, если отсутствует механизм, обеспечивающий преобразование, когда запрашивается тот или иной документ. Добавьте к этому потребность сервлетов и других серверных компонентов в выводе XML-данных, которые должны быть единообразно оформлены, и вы получите минимальный набор требований, предъявляемых к системе веб-публикации. В этой главе мы рассмотрим такую систему, посмотрим, как она позволяет экономить многие часы возни с HTML-разметкой. Эта система позволит веб-оформителям стать знатоками XML и XSL, сделав пользовательский интерфейс приложений максимально гибким и легко изменяемым.

Системы веб-публикации стремятся решить эти сложные задачи. Подобно тому как веб-сервер несет ответственность за передачу файла в ответ на запрос для URL этого файла, система веб-публикации ответственна за реагирование на схожие запросы. Однако вместо того чтобы возвращать сам файл, система веб-публикации обычно возвращает *публикуемую* версию файла. Публикуемый файл отличается от исходного тем, что мог подвергнуться преобразованиям с помощью XSLT, мог быть изменен приложением либо преобразован в иной формат, такой как PDF. Отправитель запроса не видит «необработанные» данные, которые могут стоять за публикуемым результатом, но и не должен явным образом запрашивать выполнение преобразования для публикации. Как правило, базовый URI (скажем, *http://yourHost.com/publish*) указывает на то, что запросы должна обрабатывать система публикации, установленная поверх веб-сервера. Как вы, возможно, догадываетесь, эта концепция существенно проще, чем фактическая реализация такой системы публикации, и поиск системы, подходящей для конкретных задач, является отнюдь не простой задачей.

Выбор системы публикации

Возможно, читатель надеется обнаружить список, содержащий информацию о сотнях возможных решений. Как вы могли видеть, язык

Java и различные API для XML предоставляют простые средства для создания приложений XML. Помимо этого, простым решением обработки запросов и создания ответов в веб-пространстве могут оказаться сервлеты Java. Реальность же такова, что перечень систем веб-публикации невелик, а если оставить лишь качественные и стабильные, он станет еще меньше. Один из лучших источников информации о доступных в настоящее время продуктах – это список, публикуемый на сайте XML Software, *http://xmlsoftware.com/publishing/*. Этот список обновляется достаточно часто, и поэтому нет смысла приводить его в этой книге. Тем не менее, следует упомянуть некоторые важные критерии выбора системы, оптимальной для конкретных задач.

Устойчивость

Не удивляйтесь, если вам (по-прежнему!) трудно найти продукт, номер версии которого выше 2.x. В действительности, наверное, надо как следует поискать, чтобы найти систему хотя бы второго поколения. И хотя более высокий номер версии отнюдь не гарантирует стабильной работы, обычно он отражает время и усилия, затраченные на производство продукта, а также количество переработок кода, которые претерпела система. Системы публикации XML являются настолько новым явлением, что рынок заполнен продуктами версий 1.0 и 1.1, которые просто недостаточно устойчивы для практического применения.

Зачастую можно убедиться в устойчивости продукта, изучив другие разработки того же производителя. Обычно производитель выпускает целый набор средств, и если некоторые из продуктов не обеспечивают поддержки SAX 2.0 и DOM Level 2 либо все имют версии 1.0 и 1.1, было бы разумно воздержаться от применения такой системы до тех пор, пока она не станет более совершенной и не будет соответствовать новейшим стандартам XML. Также старайтесь держаться подальше от технологий, ориентированных на какую-либо конкретную платформу. Если система веб-публикации жестко привязана к операционной системе (например, Windows или даже к определенному варианту Unix), это значит, что вы имеете дело с решением, которое построено не только на Java. Помните, что системы веб-публикации предназначены для клиентов, работающих на любой платформе. Зачем же пользоваться продуктом, который сам не может работать на любой платформе?

Интеграция с другими XML-продуктами и API

Убедившись, что ваша система публикации достаточно устойчива для решения ваших задач, удостоверьтесь, что она поддерживает широкий диапазон XML-анализаторов и процессоров. Если система жестко привязана к определенному анализатору или процессору, то вы буде-

те ограничены какой-то одной реализацией технологии. А это плохо. Хотя системы публикации обычно хорошо интегрированы с анализатором вполне определенного производителя, выясните, являются ли анализаторы взаимозаменяемыми. Проверьте, можно ли будет использовать ваш любимый (или оставшийся от предыдущих проектов) процессор.

Поддержка SAX и DOM является обязательной, а многие системы веб-публикации в настоящее время также поддерживают JDOM и JAXP. Даже если у вас есть любимый API, чем больше возможностей вы имеете, тем лучше! Попробуйте также найти систему, разработчики которой пристально следят за спецификациями схем XML, XLink, XPointer и других словарей XML. По этому признаку вы поймете, можно ли ожидать появление поддержки этих технологий в новых версиях системы (а это важный признак ее долговечности). Не стесняйтесь спрашивать, как скоро следует ожидать включения новых спецификаций в продукт, и настаивайте на четком ответе.

Факт применения

Последний и, вероятно, самый важный вопрос, на который надо ответить при выборе системы веб-публикации, – применяется ли она в реальных приложениях. Если вам не могут указать хотя бы несколько приложений или ссылок на сайты, которые используют данную систему публикации, не удивляйтесь, если их вообще не существует. Производители (или разработчики, если говорить о мире приложений с открытым кодом) должны с радостью и чувством гордости сообщать, где можно увидеть их систему в действии. Колебания в этой ситуации – признак того, что при освоении этого продукта вы можете стать большим первопроходцем, чем вам того хотелось бы. Например, Apache Cocoon предоставляет такой список по адресу *http://xml.apache.org/cocoon/livesites.html*.

Принятие решения

Оценив продукты по перечисленным критериям, скорее всего, можно получить ясный результат. Очень немногие системы публикации могут дать положительные ответы на все вопросы, поставленные здесь, не говоря уже о требованиях, предъявляемых вашим приложением. На самом деле, к июлю 2001 года существовало меньше десяти систем публикации, поддерживающих самые последние версии SAX (версия 2.0), DOM (Level 2) и JAXP (версия 1.1), используемых при этом хотя бы на одном сайте и претерпевших по меньшей мере три существенные переработки кода за время своего существования. Их список здесь не приводится по той причине, что, откровенно говоря, в течение полугода они могут исчезнуть или измениться радикальным образом.

Мир систем веб-публикаций развивается столь динамично, что я скорее введу вас в заблуждение, если рекомендую четыре или пять возможных вариантов, полагая, что они будут доступны через несколько месяцев.

Но есть одна система публикации, которая пользуется неизменным успехом в сообществе разработчиков Java и XML. Если говорить о разработках с открытым кодом, разработчики на Java часто обращаются именно к этой системе. Проект Apache Cocoon, начатый Стефано Мацокки, привел к созданию надежной системы с самого начала. Система Cocoon создавалась уже в то время, когда большинство из нас лишь пытались разобраться в том, что же такое XML, и сегодня разрабатывается второе поколение этой системы, основанной полностью на Java. Эта система является также частью проекта Apache XML и по умолчанию поддерживает Apache Xerces и Apache Xalan. Она позволяет использовать любой анализатор XML, соответствующий стандарту, и основана на чрезвычайно популярной архитектуре сервлетов Java. Кроме того, существует несколько сайтов, построенных на Apache Cocoon (в версии 1.x), которые расширяют границы традиционной разработки веб-приложений и при этом показывают исключительно хорошую производительность. По этой причине, и, как всегда, следуя принципам открытого исходного кода, я остановил свой выбор в этой главе на Apache Cocoon.

В предыдущих главах выбор анализатора и процессора XML был достаточно произвольным. Иначе говоря, с очень незначительными изменениями в коде примеры должны были работать при использовании разработок других производителей. Тем не менее, системы веб-публикации еще не стандартизованы, и каждая из них реализует самые разные возможности и следует разным соглашениям. Приводимые в этой главе примеры работают только для Apache Cocoon и не являются переносимыми. Тем не менее, популярность концепций и архитектурных решений, использованных в Cocoon, заслуживает отдельной главы. Если ваш выбор пал на другую систему публикации, следует, по крайней мере, взглянуть на приводимые здесь примеры, потому что концепции применимы для всех реализаций, хотя код таковым и не является.

Установка

В других главах инструкции по установке сводились к адресу сайта в Интернете, где можно получить дистрибутив, и добавлению *jar-файла* в переменную `CLASSPATH`. Установка системы публикации вроде Cocoon – задача, вовсе не столь простая, поэтому рассмотрим процедуру в подробностях. Кроме того, на сайте Cocoon приведены инструкции по установке, касающиеся различных сред исполнения

сервлетов; их можно найти по адресу *http://xml.apache.org/cocoon1/install.html*.[1]

Исходный код или скомпилированные версии

Прежде всего, следует определить, в каком виде нужна система Cocoon: в исходных кодах или в скомпилированном виде. Решение можно свести к следующему вопросу: вам нужны самые последние возможности или максимально надежная версия? Если вы активный разработчик и хотите разобраться с Cocoon, установите инструмент CVS, а затем извлеките последнюю версию исходного кода Cocoon из CVS-репозитория на *xml.apache.org*. Чтобы не расписывать подробно собственно процесс, который будет интересен далеко не всем читателям, сошлюсь на книгу Грегора Парди (Gregor Purdy) «*CVS Pocket Reference*» (O'Reilly). Этой книги, а также инструкций, приведенных на странице *http://xml.apache.org/cvs.html*, будет вполне достаточно.

Если стоит цель оценить работу системы Cocoon или использовать ее в реальных приложениях, следует загрузить скомпилированную версию с сайта *http://xml.apache.org/cocoon/dist/cocoon1*.[2] Как уже было сказано, версия 1.8.2 доступна для Windows (*Cocoon-1.8.2.zip*) и для Linux/Unix (*Cocoon-1.8.2.tar.gz*). Получив архив, распакуйте его во временный каталог, в котором можно работать. Отметим одну важную деталь – при этом создается каталог *lib/*. Этот каталог содержит все библиотеки, необходимые для запуска Cocoon в среде исполнения сервлетов.

Примечание

Если каталог *lib/* отсутствует либо не содержит нескольких *jar*-файлов, то у вас, возможно, более старая версия Cocoon. Только новые версии (начиная с 1.8) содержат эти библиотеки (которые, кстати, существенно облегчают жизнь!).

Настройка среды исполнения сервлетов

После сборки системы Cocoon следует настроить среду исполнения сервлетов для совместной работы с Cocoon и указать, какие запросы должны передаваться Cocoon. Рассмотрим настройку работы системы Cocoon со средой сервлетов Jakarta Tomcat.[3] Поскольку она является

[1] Основной сайт Cocoon расположен по адресу *http://xml.apache.org/cocoon*, а последней версией считается 2.0. С корневой страницы основного сайта есть ссылка на сайт Cocoon 1.x, который, соответственно, находится в каталоге *cocoon1*. – *Примеч. науч. ред.*

[2] В *.../dist/* доступна лишь ветка 2.0, ветка 1.x хранится в подкаталоге *cocoon1*. – *Примеч. науч. ред.*

[3] Страница проекта Tomcat доступна по адресу *http://jakarta.apache.org/tomcat/*. – *Примеч науч. ред.*

эталонной реализацией для Java Servlet API версии 2.2, проблем с повторением этих шагов для произвольной среды исполнения сервлетов быть не должно.

Прежде всего, следует скопировать все библиотеки, необходимые для работы Cocoon, в каталог библиотек Tomcat, *TOMCAT_HOME/lib*, где *TOMCAT_HOME* – это каталог, в который установлен Tomcat. На моей системе Windows это каталог *c:\java\jakarta-tomcat*, а в Linux – */usr/local/jakarta-tomcat*. Не весь каталог *lib/* Cocoon подлежит копированию; для выполнения Cocoon требуются следующие *jar*-файлы:

- *bsfengines.jar* (система сценариев Bean)
- *bsf.jar* (система сценариев Bean)
- *fop_0_15_0.jar* (FOP)
- *sax-bugfix.jar* (исправления SAX в части обработки ошибок)
- *turbine-pool.jar* (Turbine)
- *w3c.jar* (W3C)
- *xalan_1_2_D02.jar* (Xalan)
- *xerces_1_2.jar* (Xerces)

Кроме того, скопируйте в тот же каталог (*TOMCAT_ HOME/lib*) файл *bin/cocoon.jar*. Теперь у вас есть все библиотеки, необходимые для запуска Cocoon.

Последние версии Tomcat (я пользуюсь версией 3.2.1) автоматически загружают все библиотеки из каталога Tomcat *lib/*, то есть вам не придется мучиться с путями к классам. Если среда не поддерживает автоматическую загрузку, добавьте все *jar*-файлы в пути к классам среды сервлетов.

После добавления необходимых библиотек остается указать среде исполнения сервлетов *контекст* для работы Cocoon, то есть объяснить, где искать файлы, запрашиваемые через Cocoon. Делается это путем изменения файла *server.xml* из каталога Tomcat *conf/*. Добавьте следующую инструкцию в конец этого файла, в элемент `ContextManager`:

```
<Server>
  <!-- Прочие элементы Server -->

  <ContextManager>
    <!-- Прочие инструкции Context -->

    <Context path="/cocoon"
             docBase="webapps/cocoon"
             debug="0"
             reloadable="true" >
    </Context>
  </ContextManager>
</Server>
```

Другими словами, запросы, основанные на URI */cocoon* (такие как */cocoon/index.xml*), должны быть отображены в контекст указанного каталога (*webapps/cocoon*). Разумеется, следует создать каталоги только что определенного контекста. Создайте каталоги *cocoon* и *cocoon/WEB-INF* в каталоге Tomcat *webapps*. Должна получиться структура каталогов, подобная представленной на рис. 10.1.

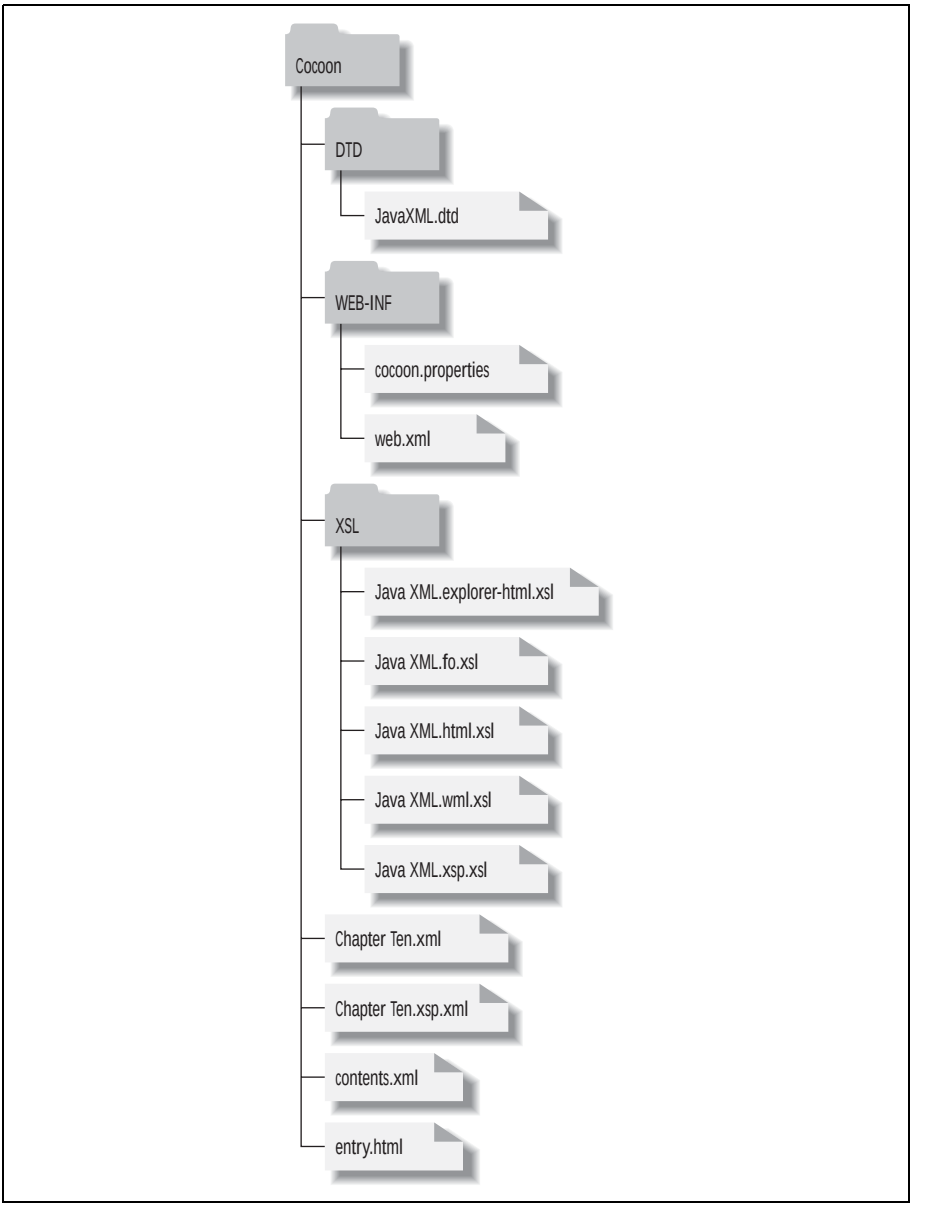

Рис. 10.1. Структура каталогов контекста Cocoon

После этого необходимо скопировать несколько файлов из дистрибутива Cocoon в контекст. Скопируйте файлы *conf/cocoon.properties* и *src/WEB-INF/web.xml* в каталог *TOMCAT_HOME/webapps/cocoon/WEB-INF/*. Теперь осталось лишь изменить новую копию файла *web.xml*. Он должен ссылаться на новую копию файла *cocoon.properties*:

```
<web-app>
 <servlet>
  <servlet-name>org.apache.cocoon.Cocoon</servlet-name>
  <servlet-class>org.apache.cocoon.Cocoon</servlet-class>
  <init-param>
   <param-name>properties</param-name>
   <param-value>WEB-INF/cocoon.properties</param-value>
  </init-param>
 </servlet>

 <servlet-mapping>
  <servlet-name>org.apache.cocoon.Cocoon</servlet-name>
  <url-pattern>*.xml</url-pattern>
 </servlet-mapping>
</web-app>
```

Остался последний, довольно неприятный шаг. Tomcat автоматически загружает все *jar*-файлы из каталога *TOMCAT_HOME/lib/*, причем последовательность загрузки определяется алфавитным порядком следования имен *jar*-файлов. Проблема заключается в том, что для работы Cocoon требуется реализация DOM Level 2 (из анализатора Xerces, который входит в состав Cocoon в виде файла *xerces_1_2.jar*); однако Tomcat использует реализацию DOM Level 1, включенную в файл *parser.jar*. Разумеется, из-за загрузки в алфавитном порядке файл *parser.jar* загружается до файла *xerces_1_2.jar*, и Cocoon остается не у дел. От этой неприятности можно избавиться. Для этого переименуйте архив *parser.jar* таким образом, чтобы он загружался после Xerces (я назвал его *z_parser.jar*). Этот шаг гарантирует, что классы DOM Level 1 по-прежнему доступны Tomcat, но классы DOM Level 2 загружаются первыми и используются системой Cocoon.

Выполнив все эти шаги, проверьте работу Cocoon, загрузив информационный URI Cocoon, который сообщает подробности об установке Cocoon: *http://[имя узла:порт]/cocoon/Cocoon.xml*. По умолчанию это будет адрес *http://localhost:8080/cocoon/Cocoon.xml*. Ваш броузер отобразит информацию, подобную приведенной на рис. 10.2.[1]

Сделав это, можно размещать реальное содержимое. В описанном варианте установки все запросы документов с URI, оканчивающимися на *.xml* и принадлежащими указанному контексту, будут обрабатываться сервлетом Cocoon.

---

[1] Разумеется, чтобы система заработала, среда исполнения сервлетов должна быть уже настроена и интегрирована с веб-сервером. – *Примеч. науч. ред.*

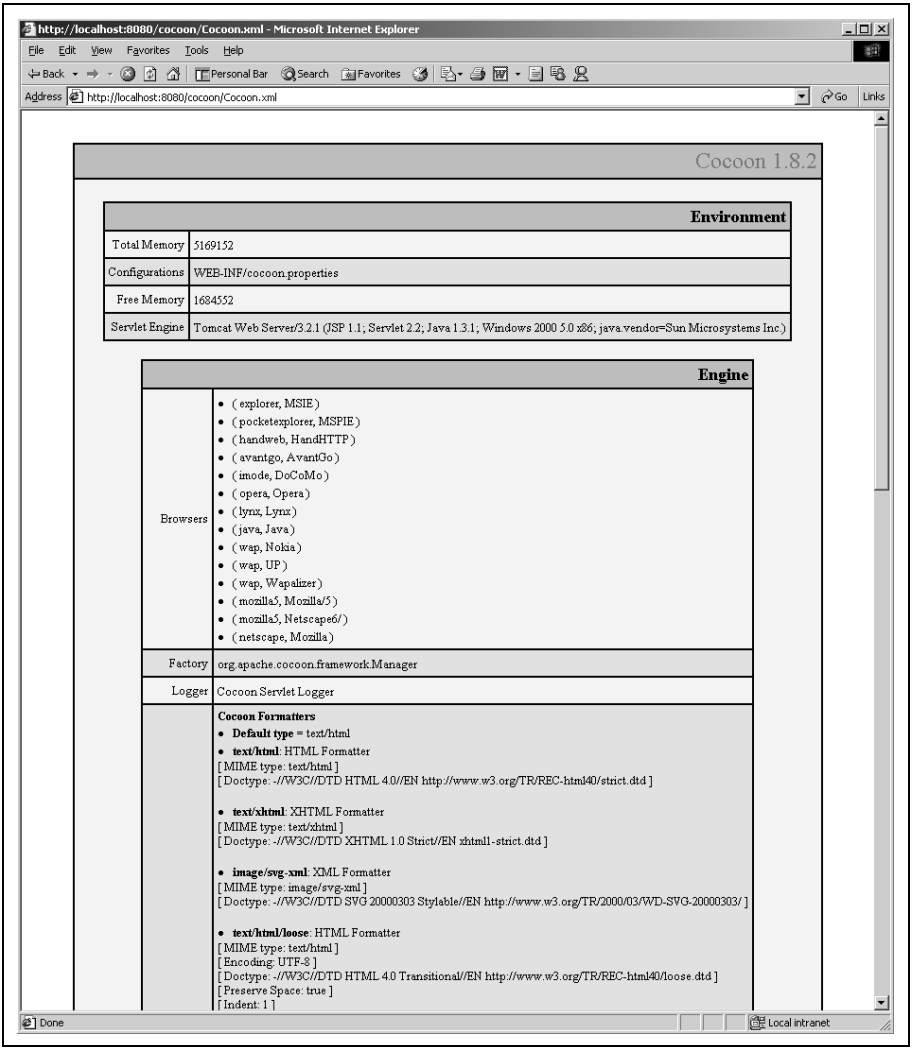

Рис. 10.2. Проверка установки Cocoon

Применение системы публикации

Использование хорошей системы публикации, такой как Cocoon, не требует каких-либо специальных знаний. Это не настолько сложное приложение, чтобы пользователям приходилось к нему привыкать. На самом деле все применения Cocoon основаны на простых URL, которые вводятся в стандартном веб-броузере. Создание динамического HTML из XML, просмотр XML-данных, преобразованных в формат PDF, и даже создание приложений VRML из XML осуществляется

простым набором в веб-броузере URL-адреса нужного XML-файла. Остальное делает система Cocoon и мощь технологии XML.

Преобразование XML в HTML

Наконец, наша система публикации установлена и корректно обрабатывает запросы для адресов, оканчивающихся на *.xml*, и мы можем увидеть, как она публикует наши XML-файлы. В состав Cocoon входит набор примеров XML-файлов и связанных с ними таблиц стилей XSL, который хранится в подкаталоге *samples/*. Но у нас имеются собственный XML-документ и таблица стилей XSL из предыдущих глав, поэтому попробуем преобразовать содержание книги в формате XML (*contents.xml*) с помощью таблицы стилей XSL (*JavaXML.html.xsl*) из главы 2. Найдите сохраненный XML-файл и скопируйте его в корневой каталог документов Cocoon *webapps/cocoon/*. Наш документ связан с таблицей стилей *XSL/JavaXML.html.xsl*. Создайте каталог *XSL/* в корневом каталоге документов веб-сервера и скопируйте в этот каталог таблицу стилей. XML-документ также ссылается и на DTD; ссылку следует либо закомментировать, либо скопировать файл *JavaXML.dtd* (см. главу 2) в каталог *DTD/*, созданный в корневом каталоге веб-сервера.

Разместив XML-документ и таблицу стилей, можно обратиться к документу с помощью веб-броузера, указав URL *http://<имя узла>:<порт>/cocoon/contents.xml*.[1] Если все инструкции, связанные с установкой Cocoon, выполнены, то преобразованный XML-документ должен выглядеть примерно так, как показано на рис. 10.3.

Это должно быть практически тривиальным. Когда Cocoon установлен и настроен, выдача динамического содержимого становится простейшей задачей! Соответствие между расширениями XML и системой Cocoon работает для всех запросов в контексте Cocoon.

Преобразование XML в формат PDF

Говоря о визуализации данных XML, я концентрировался на преобразовании XML в HTML. Но это лишь начало разговора о форматах, в которые может быть преобразован XML. В качестве формата конечных документов может выступать любой из многочисленных языков разметки; но, кроме того, Java предоставляет библиотеки для преобразования XML в форматы, не имеющие отношения к разметке. Наиболее популярной и стабильной библиотекой такого рода является FOP, процессор форматирования, разработанный в рамках проекта Apache XML.

---

[1] Здесь есть некоторое противоречие. Cocoon, фактически, срабатывает только при указании контекста (каталог *cocoon* в данном случае). Запрос *http://localhost/contents.xml* будет обработан только веб-сервером, и сервер, скорее всего, сообщит, что документ не найден (404). – *Примеч. науч .ред*.

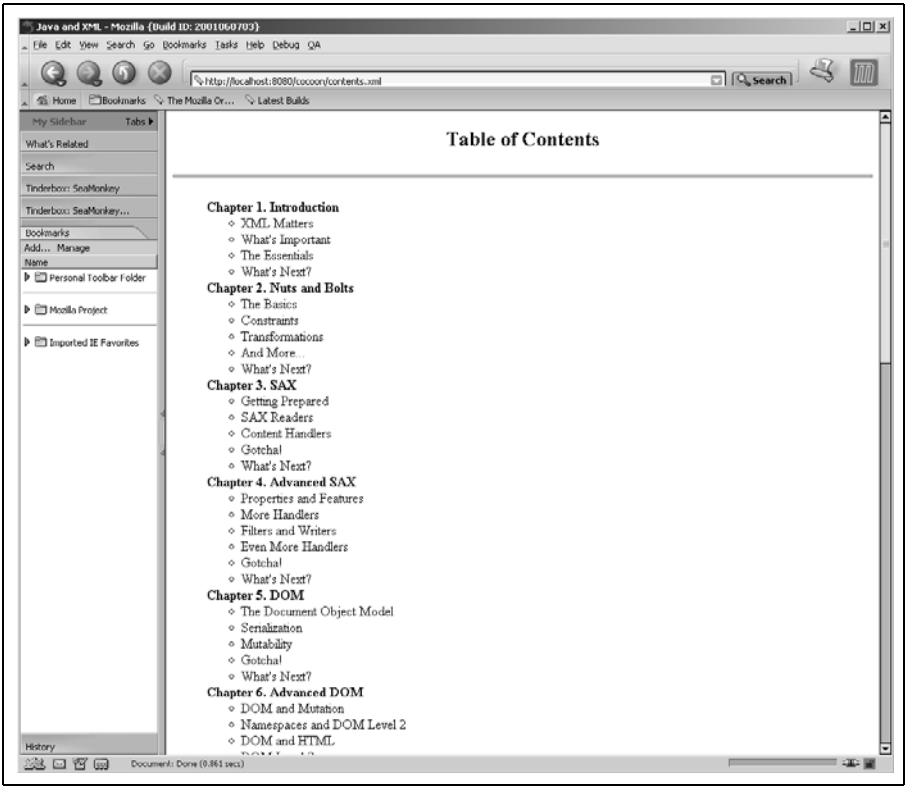

Рис. 10.3. Cocoon в действии для файла contents.xml

С помощью этого пакета любая система публикации, включая Cocoon, получает возможность преобразовывать документы XML в формат PDF, для просмотра которого обычно используют Adobe Acrobat (*http://www.adobe.com*).

Важность преобразования документа из XML-формата в формат PDF невозможно переоценить, и, если говорить о сайтах, ориентированных на публикацию отдельных документов, таких как сайты печатных изданий или издательских компаний, подобная возможность способна полностью преобразить доставку данных в Сети. Рассмотрим следующий XML-документ – отрывок из этой главы, представленный в формате XML (пример 10.1).

Пример 10.1. XML-версия «Java и XML»

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

<?cocoon-process type="xslt"?>

<?xml-stylesheet href="XSL/JavaXML.fo.xsl" type="text/xsl"?>

<book>
 <cover>
```

```
  <title>Java и XML</title>
   <author>Бретт Маклафлин </author>
 </cover>

 <contents>
  <chapter title="Системы веб-публикации" number="10">

   <paragraph> С этой главы начинается обзор специальных тем, посвященных
Java и XML. Мы уже рассмотрели основы применения XML в Java, уделили внимание
использованию API SAX, DOM, JDOM и JAXP для работы с XML, а также вопросам
использования и создания XML-документов. Теперь, когда вы понимаете, как
использовать XML в вашем коде, можно перейти к обсуждению конкретных
приложений. В следующих шести главах представлены важнейшие приложения XML и,
в частности, их реализации в мире технологий Java. Хотя уже существуют тысячи
важнейших приложений XML, темы, рассматриваемые в этих главах, постоянно
находятся в центре внимания и несут в себе значительный потенциал для
изменения традиционных процессов разработки приложений.
   </paragraph>

    <sidebar title="Чем больше все меняется, тем больше остается неизменным">
Те, кто читал первое издание книги, обнаружат, что большая часть разговора о
Cocoon в этой главе осталась прежней. Хоть я и обещал, что к этому времени
выйдет Cocoon 2, и собирался написать целую главу, посвященную ему, все
развивалось не так быстро, как я того ожидал. Стефано Маццокки (Stefano
Mazzochi) - автор Cocoon - решил наконец закончить учебу (так держать,
Стефано!), и в результате разработка Cocoon 2 замедлилась. Основная
разработка по-прежнему сосредоточена на Cocoon 1.x,  так что ее и следует
использовать. Раздел, посвященный Cocoon 2, обновлен и отражает ожидаемые
нововведения. В ближайшие месяцы следует ожидать новых книг от O'Reilly,
посвященных вопросам, связанным с  системой Cocoon. - -</sidebar>

   <paragraph> Первая актуальная тема, которую я рассмотрю, связана с
приложением XML, которое вызвало наибольший резонанс среди разработчиков на
XML и Java. Речь идет о системе веб-публикации. Хотя я постоянно говорю, что
визуализацию содержимого документа, возможно, переоценивают по сравнению со
значимостью переносимости данных, которую обеспечивает XML, применение XML
для формирования представления данных все же является очень важным.
Потребность в визуализации возрастает, когда речь заходит о веб-
ориентированных приложениях. -</paragraph>
  </chapter>

 </contents>
</book>
```

Вы уже видели, как таблицы стилей XSL позволяют преобразовать документ в HTML. Однако преобразование целой главы книги в формат HTML может привести к созданию гигантского HTML-документа и, конечно, совершенно не читаемого; возможные читатели, желающие получить книгу в режиме реального времени, в общем случае предпочитают PDF-документ. С другой стороны, статическая генерация PDF-файла из текста главы означает, что изменения в тексте главы должны быть согласованы с последующей генерацией PDF-файла. Использование XML в качестве единого формата документов означает, что главу можно легко обновить (с помощью любого редактора XML), перефор-

матировать в SGML для печати бумажной копии, передать другим компаниям и приложениям и включить в другие книги и сборники. А теперь добавьте к этому мощному набору характеристик возможность для пользователей набрать URL и получить доступ к книге в формате PDF – и вы получаете законченную систему публикации.

Здесь не рассмотрены объекты форматирования и FOP для Java в подробностях, но исчерпывающую информацию по объектам форматирования в спецификации XSL можно найти на сайте *http://www.w3.org/TR/xsl/*. Пример 10.2 представляет собой таблицу стилей XSL с применением объектов форматирования, определяющих преобразование документа XML, соответствующего данной главе, в PDF-документ.

Пример 10.2. Таблица стилей XSL для преобразования в PDF

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <xsl:template match="book">
    <xsl:processing-instruction name="cocoon-format">
      type="text/xslfo"
    </xsl:processing-instruction>
    <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
      <fo:layout-master-set>
      <fo:simple-page-master
        master-name="right"
        margin-top="75pt"
        margin-bottom="25pt"
        margin-left="100pt"
        margin-right="50pt">
        <fo:region-body margin-bottom="50pt"/>
        <fo:region-after extent="25pt"/>
      </fo:simple-page-master>
      <fo:simple-page-master
        master-name="left"
        margin-top="75pt"
        margin-bottom="25pt"
        margin-left="50pt"
        margin-right="100pt">
        <fo:region-body margin-bottom="50pt"/>
        <fo:region-after extent="25pt"/>
      </fo:simple-page-master>
      <fo:page-sequence-master master-name="psmOddEven">
        <fo:repeatable-page-master-alternatives>
          <fo:conditional-page-master-reference
              master-name="right"
              page-position="first"/>
          <fo:conditional-page-master-reference
              master-name="right"
              odd-or-even="even"/>
          <fo:conditional-page-master-reference
```

```
            master-name="left"
            odd-or-even="odd"/>
        <!-- recommended fallback procedure -->
        <fo:conditional-page-master-reference
            master-name="right"/>
      </fo:repeatable-page-master-alternatives>
    </fo:page-sequence-master>
    </fo:layout-master-set>

    <fo:page-sequence master-name="psmOddEven">

      <fo:static-content flow-name="xsl-region-after">
        <fo:block text-align-last="center" font-size="10pt">
          <fo:page-number/>
        </fo:block>
      </fo:static-content>

      <fo:flow flow-name="xsl-region-body">
        <xsl:apply-templates/>
      </fo:flow>
    </fo:page-sequence>

  </fo:root>
</xsl:template>

<xsl:template match="cover">
  <fo:block font-size="10pt"
            space-before.optimum="10pt">
    <xsl:value-of select="title"/>
    (<xsl:value-of select="author"/>)
  </fo:block>
</xsl:template>

<xsl:template match="contents">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="chapter">
  <fo:block font-size="24pt"
            text-align-last="center"
            space-before.optimum="24pt">
    <xsl:value-of select="@number" />.
    <xsl:value-of select="@title" />
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="paragraph">
  <fo:block font-size="12pt"
            space-before.optimum="12pt"
            text-align="justify">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="sidebar">
  <fo:block font-size="14pt"
```

```
                font-style="italic"
                color="blue"
                space-before.optimum="16pt"
                text-align="center">
        <xsl:value-of select="@title" />
      </fo:block>
      <fo:block font-size="12pt"
                color="blue"
                space-before.optimum="16pt"
                text-align="justify">
        <xsl:apply-templates/>
      </fo:block>
    </xsl:template>
  </xsl:stylesheet>
```

Если создать оба этих файла, сохранить главу в файле *chapterTen.xml*, а таблицу стилей XSL в файле *JavaXML.fo.xsl* в подкаталоге *XSL/*, то результат преобразования можно получить с помощью веб-броузера. Убедитесь, что располагаете программой Adobe Acrobat Reader и соответствующим модулем (plug-in) для веб-броузера, а затем обратитесь к только что созданному XML-документу. Результаты показаны на рис. 10.4.

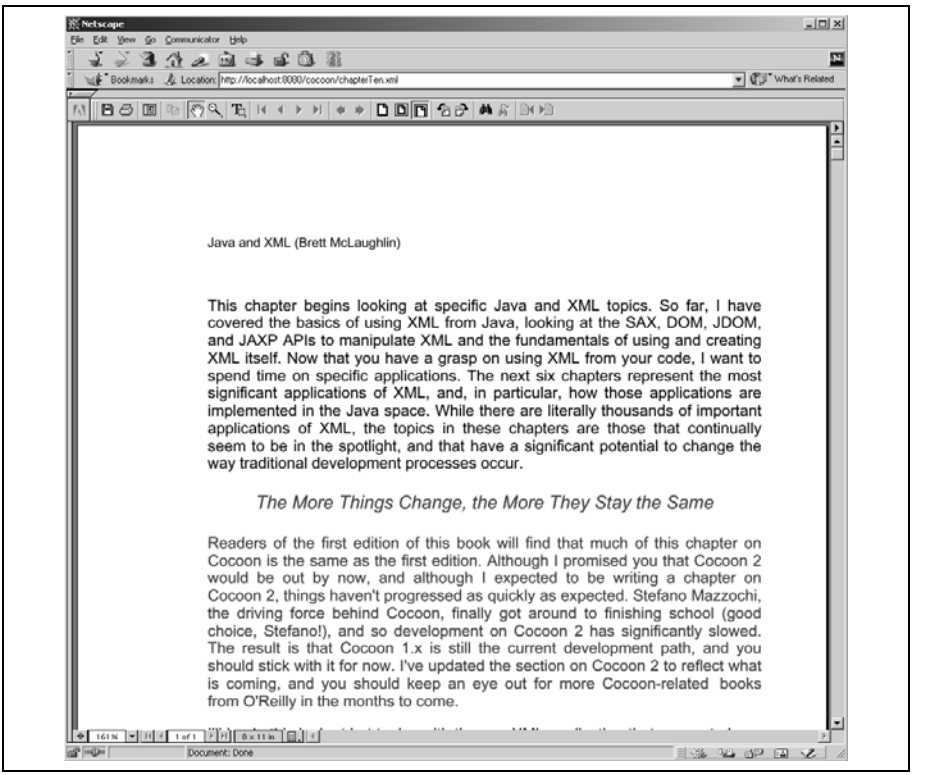

Рис. 10.4. Результат преобразования в PDF из chapterTen.xml

Форматирование в зависимости от броузера

Помимо специальных типов преобразований, таких как конвертирование в PDF, система Cocoon предусматривает динамическую обработку, основанную на способе доставки информации. Распространенным примером такой обработки является использование различного форматирования для разных типов клиентов. В традиционном веб-окружении это позволяет различными способами визуализировать XML-документы исходя из категории и версии броузера клиента. Для клиента, работающего с Internet Explorer, может использоваться иное представление данных, чем для клиента, работающего с Netscape. Принимая во внимание недавние «войны» версий HTML, DHTML и JavaScript, в которых активно участвовали компании Netscape и Microsoft, эта мощная возможность стоит того, чтобы иметь ее в своем распоряжении. Система Cocoon обеспечивает встроенную поддержку для множества распространенных типов броузеров. Найдите файл *cocoon.properties*, о котором мы говорили ранее, откройте его и перейдите в конец файла. Вы увидите следующую секцию (в более новых версиях она может несколько отличаться):

```
##########################################
# User Agents (Browsers)                 #
##########################################


# NOTE: numbers indicate the search order. This is very important since
# some words may be found in more than one browser description. (MSIE is
# presented as "Mozilla/4.0 (Compatible; MSIE 4.01; ...")
#
# for example, the "explorer=MSIE" tag indicates that the XSL stylesheet
# associated to the media type "explorer" should be mapped to those
# browsers that have the string "MSIE" in their "user-Agent" HTTP header.
[Перевод:
##########################################
# Пользовательские агенты (броузеры)     #
##########################################

# ПРИМЕЧАНИЕ: числа определяют порядок поиска. Это ОЧЕНЬ ВАЖНО,
# поскольку некоторые слова могут быть найдены в описании различных
# броузеров.
# (MSIE заявляет о себе как "Mozilla/4.0 (Compatible; MSIE 4.01; ...")
#
# например, метка "explorer=MSIE" указывает на то, что таблица стилей XSL,
# связанная с типом клиента "explorer", должна быть привязана к тем
броузерам, которые
# имеют строку "MSIE" в HTTP-заголовке "user-Agent".]

browser.0 = explorer=MSIE
browser.1 = pocketexplorer=MSPIE
```

```
browser.2 = handweb=HandHTTP
browser.3 = avantgo=AvantGo
browser.4 = imode=DoCoMo
browser.5 = opera=Opera
browser.6 = lynx=Lynx
browser.7 = java=Java
browser.8 = wap=Nokia
browser.9 = wap=UP
browser.10 = wap=Wapalizer
browser.11 = mozilla5=Mozilla/5
browser.12 = mozilla5=Netscape6/
browser.13 = netscape=Mozilla
```

Здесь нас интересуют ключевые слова после первого знака равенства. Например, `explorer`, `lynx`, `java` и `mozilla5` позволяют различать агенты пользователей и являются «кодами», которые посылают броузеры при выполнении запросов по URL-адресам. В качестве примера применения таблиц стилей, основанных на этом свойстве, создадим таблицу стилей XSL, применяемую в том случае, когда клиент обращается к оглавлению книги в формате XML (*contents.xml*) с помощью броузера Internet Explorer. Скопируйте первоначальную таблицу стилей *JavaXML.html.xsl* в файл *JavaXML.explorer-html.xsl*, а затем внесите изменения, как показано в примере 10.3.

Пример 10.3. Измененная таблица стилей XSL для Internet Explorer

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

<xsl:stylesheet xmlns:javaxml2="http://www.oreilly.com/javaxml2"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ora="http://www.oreilly.com"
                version="1.0"
>

  <xsl:template match="javaxml2:book">
  <xsl:processing-instruction name="cocoon-format">
    type="text/html"
  </xsl:processing-instruction>
    <html>
      <head>
        <title>
          <xsl:value-of select="javaxml2:title" /> (Версия для броузера
          Explorer)
        </title>
      </head>
      <body>
        <xsl:apply-templates select="*[not(self::javaxml2:title)]" />
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:template match="javaxml2:contents">
    <center>
     <h2>Оглавление (Версия для броузера Explorer)</h2>
     <small>
       Попробуйте броузер <a href="http://www.mozilla.org">Mozilla</a>!
     </small>
    </center>
    <!-- Прочие инструкции XSL -->
  </xsl:template>

  <!-- Прочие шаблоны/сопоставления XSL -->

</xsl:stylesheet>
```

И хотя данный пример тривиален, у нас есть потенциальная возможность включить динамический HTML для броузера Internet Explorer 5.5, а для Netscape Navigator или Mozilla, которые реализуют менее полную поддержку DHTML, использовать стандартный HTML. При этом следует сообщить XML-документу о том, что если способ доставки (или идентификатор пользовательского агента) совпадает с определенным в файле свойств типом `explorer`, необходимо использовать соответствующую таблицу стилей XSL. Эту задачу решает дополнительная инструкция обработки, показанная в примере 10.4, и ее можно добавить в файл *contents.xml*.

Пример 10.4. Измененный документ contents.xml, распознавание способа доставки

```
<?xml version="1.0"?>
<!DOCTYPE Book SYSTEM "DTD/JavaXML.dtd">
<?xml-stylesheet href="XSL/JavaXML.html.xsl" type="text/xsl"?>
<?xml-stylesheet href="XSL/JavaXML.explorer-html.xsl" type="text/xsl"
                 media="explorer"?>

<?cocoon-process type="xslt"?>

<!-- "Java и XML", Оглавление -->
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
>
  <!-- XML-данные -->
</book>
```

Обращение к XML-документу при помощи броузера Netscape дает те же результаты, что и раньше. Однако если обратиться к этой странице с помощью броузера Internet Explorer, можно увидеть, что документ был преобразован с применением другой таблицы стилей и выглядит так, как показано на рис. 10.5.

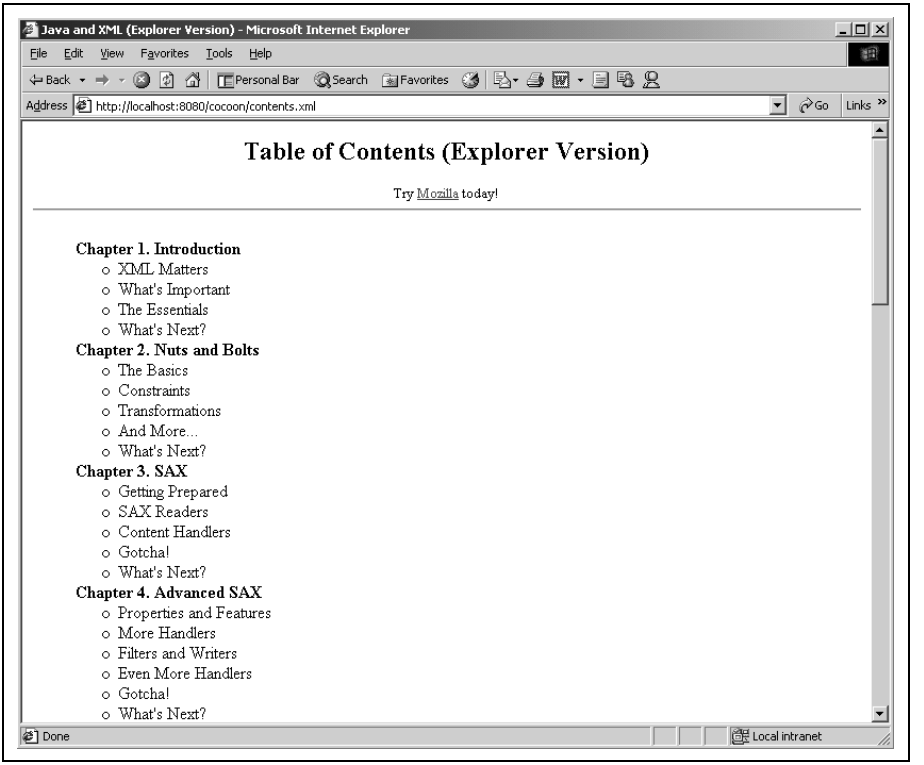

Рис. 10.5. Файл contents.xml, полученный с помощью Internet Explorer

WAP и XML

Одна из самых сильных сторон динамического применения таблиц стилей связана с использованием беспроводных устройств. Помните файл свойств?

```
##########################################
# User Agents (Browsers)                 #
##########################################

# NOTE: numbers indicate the search order. This is very important since
# some words may be found in more than one browser description. (MSIE is
# presented as "Mozilla/4.0 (Compatible; MSIE 4.01; ...")
#
# for example, the "explorer=MSIE" tag indicates that the XSL stylesheet
# associated to the media type "explorer" should be mapped to those
# browsers that have the string "MSIE" in their "user-Agent" HTTP header.

browser.0 = explorer=MSIE
browser.1 = pocketexplorer=MSPIE
browser.2 = handweb=HandHTTP
```

```
browser.3 = avantgo=AvantGo
browser.4 = imode=DoCoMo
browser.5 = opera=Opera
browser.6 = lynx=Lynx
browser.7 = java=Java
browser.8 = wap=Nokia
browser.9 = wap=UP
browser.10 = wap=Wapalizer
browser.11 = mozilla5=Mozilla/5
browser.12 = mozilla5=Netscape6/
browser.13 = netscape=Mozilla
```

Выделенные записи указывают, что для доступа к содержимому документа используется беспроводное устройство, такое как телефон, имеющий выход в Интернет. Система Cocoon в состоянии определить тип броузера, запросившего документ (Internet Explorer или Netscape), и вернуть результат, отформатированный с применением соответствующей таблицы стилей; и одна из этих таблиц стилей может предназначаться для WAP-устройств. Добавьте ссылку на еще одну таблицу стилей в документ *contents.xml*:

```
<?xml version="1.0"?>
<!DOCTYPE Book SYSTEM "DTD/JavaXML.dtd">
<?xml-stylesheet href="XSL/JavaXML.html.xsl" type="text/xsl"?>
<?xml-stylesheet href="XSL/JavaXML.explorer-html.xsl" type="text/xsl"
                 media="explorer"?>
<?xml-stylesheet href="XSL/JavaXML.wml.xsl" type="text/xsl"
                 media="wap"?>

<?cocoon-process type="xslt"?>

<!-- "Java и XML", Оглавление -->
<book xmlns="http://www.oreilly.com/javaxml2"
      xmlns:ora="http://www.oreilly.com"
>
  <!-- Оглавление в формате XML -->
</book>
```

Теперь необходимо создать эту новую таблицу стилей для WAP-устройств. Как правило, при создании таблицы стилей для устройства WAP используется язык разметки для беспроводных устройств (Wireless Markup Language, WML). Он является вариантом HTML, но имеет несколько иной метод представления различных страниц. Когда беспроводное устройство запрашивает документ по URL, возвращаемый ответ должен размещаться внутри элемента `wml`. Корневой элемент может содержать несколько *карт* (*cards*), каждая из которых определяется при помощи элемента WML `card`. Устройство загружает сразу несколько карт (обычно этот набор называется *колодой* (*deck*)) таким образом, что ему не требуется обращаться к серверу за дополнительными страницами. В примере 10.5 показана простая WML-страница, использующая упомянутые конструкции.

Пример 10.5. Простая WML-страница

```
<wml>
 <card id="index" title="Домашняя страница">
  <p align="left">
   <i>Основное меню</i><br />
   <a href="#title">Титульная страница</a><br />
   <a href="#myPage">Моя страница</a><br />
  <p>
 </card>

 <card id="title" title="Моя титульная страница ">
  Добро пожаловать на мою титульную страницу!<br />
  Я очень рад вас видеть.
 </card>

 <card id="myPage" title="Здравствуй, мир ">
  <p align="center">
   Здравствуй, мир!
  </p>
 </card>
</wml>
```

Запрос этого документа приводит к получению меню и пары страниц, доступ к которым можно получить с помощью пунктов меню. Полная спецификация WML 1.1, а также иные спецификации, связанные с WAP, доступны в Интернете на странице *http://www.wapforum.org/what/technical_1_1.htm*. Также рекомендую книгу Мартина Фроста (Martin Frost) «Learning WML and WMLScript», O'Reilly (Изучаем WML и WMLScript). Кроме того, можно загрузить WAP-версию продукта Openwave™ SDK с сайта *http://www.openwave.com/products/developer_products/sdk/*. Этот программный продукт предназначен для разработки WAP-страниц и в числе прочих инструментов включает эмуляцию беспроводного устройства.[1] С его помощью можно разработать таблицу стилей XSL для выдачи WML-кода устройствам WAP и проверить результаты, указав в эмуляторе SDK адрес *http://<имя узла>:<порт>/cocoon/contents.xml*.

Поскольку телефонные дисплеи гораздо меньше, чем экраны компьютеров, мы хотим отобразить лишь некоторую часть информации из оглавления книги в формате XML. В примере 10.6 приведена таблица стилей XSL, которая выдает три карты WML. Первая из них представляет собой меню, имеющее ссылки на две другие карты. Вторая карта генерирует оглавление из документа *contents.xml*. Третья карта – это просто страничка с информацией о правообладании. Данную таблицу

[1] *phone.com* была поглощена компанией *openwave.com*, и обнаружить в числе продуктов Openwave UP.SDK не удалось. Зато нашелся Openwave SDK, WAP Edition, который выполняет точно те же функции. – *Примеч. науч. ред.*

стилей можно сохранить под именем *JavaXML.wml.xsl* в подкаталоге *XSL/* корневого каталога документов веб-сервера.

Пример 10.6. Таблица стилей для вывода WML

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

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:javaxml2="http://www.oreilly.com/javaxml2"
                xmlns:ora="http://www.oreilly.com"
                exclude-result-prefixes="javaxml2 ora"
>

 <xsl:template match="javaxml2:book">
  <xsl:processing-instruction name="cocoon-format">
    type="text/wml"
  </xsl:processing-instruction>

  <wml>
   <card id="index" title="{javaxml2:title}">
    <p align="center">
     <i><xsl:value-of select="javaxml2:title"/></i><br />
     <a href="#contents">Contents</a><br/>
     <a href="#copyright">Copyright</a><br/>
    </p>
   </card>

   <xsl:apply-templates select="javaxml2:contents" />

   <card id="copyright" title="Copyright">
    <p align="center">
     Copyright 2000, O&apos;Reilly &amp; Associates
    </p>
   </card>
  </wml>
 </xsl:template>

 <xsl:template match="javaxml2:contents">
  <card id="contents" title="Contents">
   <p align="center">
    <i>Contents</i><br />
    <xsl:for-each select="javaxml2:chapter">
     <xsl:value-of select="@number" />.
     <xsl:value-of select="@title" /><br />
    </xsl:for-each>
   </p>
  </card>
 </xsl:template>

</xsl:stylesheet>
```

Если не считать тегов WML, большая часть этого примера должна выглядеть привычно. Здесь также присутствует инструкция обработки для Cocoon, целевое приложение которой – cocoon-format. Передаваемые инструкцией данные (type="text/wml") предписывают системе Cocoon осуществлять вывод этой таблицы стилей с заголовком содержимого text/wml (вместо обычного text/html или text/plain). Очень важен атрибут, добавленный к корневому элементу таблицы стилей:

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

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:javaxml2="http://www.oreilly.com/javaxml2"
                xmlns:ora="http://www.oreilly.com"
                exclude-result-prefixes="javaxml2 ora"
>
```

По умолчанию любые объявления пространств имен XML, за исключением объявления пространства имен XSL, добавляются в корневой элемент результата преобразований. В данном примере к корневому элементу результата преобразования wml были бы добавлены объявления пространств имен, связанных с префиксами javaxml2 и ora:

```
<wml xmlns:javaxml2="http://www.oreilly.com/javaxml2"
     xmlns:ora="http://www.oreilly.com"
>
  <!-- Данные WML -->
</wml>
```

Но такое добавление вызвало бы сообщение об ошибке в WAP-броузере, поскольку xmlns:javaxml2 и xmlns:ora не являются допустимыми атрибутами элемента wml. WAP-броузеры не столь снисходительны, как HTML-броузеры, и оставшаяся часть WML-данных не будет показана. Тем не менее это пространство должно быть объявлено, чтобы таблица стилей XSL могла выполнять сопоставление шаблонов для исходного документа, который использует пространство имен, связанное с префиксом javaxml. Для решения этой проблемы в XSL предусмотрен атрибут exclude-result-prefixes, который следует добавить к элементу xsl:stylesheet. Префикс пространства имен, указанный в этом атрибуте, не будет добавляться к результату преобразований, а именно это нам и нужно. Теперь результат будет выглядеть следующим образом:

```
<wml>
  <!-- Данные WML -->
</wml>
```

Этот код отлично воспринимается WAP-броузером. Если у вас эмулятор WAP-устройства (Openwave SDK) загружен, можно обратиться с его помощью к оглавлению в формате XML и увидеть результаты.

На рис. 10.6 показано главное меню, полученное в результате преобразования с использованием таблицы стилей для WML. Преобразование выполняется в момент, когда WAP-устройство запрашивает файл *contents.xml* через систему публикации Cocoon.

Рис. 10.6. Основное меню для оглавления «Java и XML»

Внимание

В тестируемых автором версиях броузера UP.SDK ссылка на сущность `OreillyCopyright` не разрешалась. Чтобы примеры работали, пришлось закомментировать эту строку в XML-коде. Вероятно, вам потребуется делать то же самое до тех пор, пока эта ошибка в эмуляторе не будет исправлена.

На рис. 10.7 показано сгенерированное оглавление книги, доступ к которому был осуществлен посредством нажатия кнопки `Link`, когда на дисплее указана ссылка *Contents*.

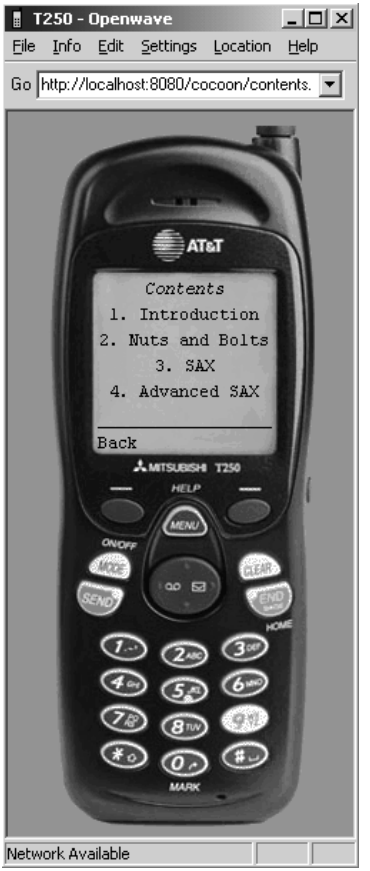

Рис. 10.7. Оглавление в формате WML

За более подробной информацией о WML и WAP обратитесь к страницам в Интернете: *http://www.openwave.com*[1] и *http://www.wapforum.org*. На каждом из этих сайтов представлен богатый выбор ресурсов, связанных с разработкой приложений для мобильных устройств.

К этому моменту читатели должны уже хорошо представлять все разнообразие конечных документов, которые можно создать с помощью системы Cocoon. С минимальным количеством затраченных усилий и дополнительных таблиц стилей один и тот же XML-документ может быть представлен во множестве форматов для самых разнообразных

[1] Внимательные читатели заметят отсутствие ссылок на phone.com. Дело в том, что phone.com теперь стал частью OpenWave (*http://www.openwave.com*).

типов клиентов. Это одна из возможностей, делающих системы веб-публикации столь мощным средством. Без XML и подобных систем для каждого типа клиента пришлось бы создавать отдельный сайт. Теперь, когда вы убедились в гибкости системы Cocoon в вопросах создания выходных данных, перейдем к обзору технологии Cocoon, которая позволяет динамически создавать и модифицировать исходные данные для этих преобразований.

XSP

Аббревиатура XSP означает *расширяемые серверные страницы* (Extensible Server Pages), и эта технология представляет собой, возможно, наиболее важную разработку, вышедшую из недр проекта Cocoon. Серверные страницы Java (JSP) позволяют внедрять метки и код Java в обычные HTML-страницы. Когда поступает запрос на получение такой страницы JSP, внедренный код выполняется, а результаты вставляются в передаваемый клиенту HTML-код[1]. Эта технология стала широко применяться разработчиками на Java и ASP, упрощая, якобы, серверное программирование на Java и позволяя разделять логику и выводимые данные. Однако здесь все еще существует несколько серьезных проблем. Во-первых, JSP в действительности не обеспечивает разделения содержания и представления. Проблема все та же, и мы о ней уже говорили: изменение баннера, цвета или размера шрифта требует внесения изменений в JSP (включая внедренный код на Java и ссылки на компоненты JavaBean). JSP привязывает содержимое (чистые данные) к представлению точно так же, как это делает статический HTML. Во-вторых, JSP-страницы не могут быть преобразованы в другие форматы и не могут использоваться в нескольких приложениях, поскольку спецификация JSP разработана главным образом «вокруг» выдачи данных.

XSP решает эти проблемы. По сути дела, XSP – это просто XML. Рассмотрим XSP-страницу, приведенную в примере 10.7.

Пример 10.7. Простая XSP-страница

```
<?xml version="1.0"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="myStylesheet.xsl" type="text/xsl"?>

<xsp:page language="java"
```

[1] Это чересчур сильное упрощение. В действительности JSP прекомпилируется в сервлет, и выводом данных управляет класс `PrintWriter`. Дополнительную информацию о JSP можно найти в книге Ханса Бергстена (Hans Bergsten) «JavaServer Pages», вышедшей в издательстве O'Reilly & Associates.

```
          xmlns:xsp="http://www.apache.org/1999/XSP/Core"
>

 <xsp:logic>
  private static int numHits = 0;

  private synchronized int getNumHits() {
   return ++numHits;
  }
 </xsp:logic>

 <page>
  <title>Счетчик посещений</title>

  <p>Эта страница была запрошена <xsp:expr>getNumHits()</xsp:expr> раз.</p>
 </page>
</xsp:page>
```

Этот пример следует всем соглашениям XML. Считайте пока, что доступ к содержимому элемента `xsp:logic` для анализатора XML «запрещен», – позже мы к этому вернемся. В остальном документ представляет собой обычный XML-документ с несколькими новыми элементами. Он ссылается на таблицу стилей XSL, в которой нет ничего особенного, как видно из примера 10.8.

Пример 10.8. Таблица стилей XSL для XSP-страницы

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

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>

  <xsl:template match="page">
    <xsl:processing-instruction name="cocoon-format">
      type="text/html"
    </xsl:processing-instruction>
    <html>
      <head>
        <title><xsl:value-of select="title"/></title>
      </head>
      <body>
        <xsl:apply-templates select="*[not(self::title)]" />
      </body>
    </html>
  </xsl:template>

  <xsl:template match="p">
    <p align="center">
      <xsl:apply-templates />
    </p>
  </xsl:template>

</xsl:stylesheet>
```

Таким образом, XSP легко решает первую главную проблему JSP, обеспечивая четкое разделение содержимого и представления. Такое разделение дает возможность разработчикам управлять созданием содержимого документов (XSP-страницу можно сгенерировать с помощью сервлета или другого кода на Java), в то время как авторы XML- и XSL-документов могут управлять представлением и форматированием данных, изменяя таблицы стилей XSL. Так же легко XSP восполняет другой существенный недостаток JSP: поскольку обработка XSP выполняется до применения каких-либо таблиц стилей, получаемый XML-документ может быть преобразован в любой другой формат. XSP сохраняет все преимущества XML. XSP-страницу можно передавать между приложениями или использовать для представления данных.

Создание XSP-страницы

Теперь, когда вы познакомились с технологией XSP, можно создать собственную XSP-страницу. Обратимся для этого к созданному ранее XML-документу, представляющему собой часть этой главы и ранее уже преобразованному в формат PDF. Вместо того чтобы просто использовать этот документ для вывода информации, предположим, что автор хочет предоставить возможность редактору просматривать документ в процессе написания. Однако в дополнение к тексту книги редактор должен иметь возможность видеть примечания автора, которые все остальные видеть не должны: например, замечания о стиле и форматировании. Прежде всего, добавим примечание к созданному ранее файлу *chapterTen.xml*:

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

<?cocoon-process type="xslt"?>
<?xml-stylesheet href="XSL/JavaXML.fo.xsl" type="text/xsl"?>

<book>
 <cover>
  <title>Java и XML</title>
  <author>Бретт Мак-Лахлин </author>
 </cover>

 <contents>
  <chapter title="Системы веб-публикации" number="10">

   <paragraph>  С этой главы начинается обзор специальных тем, посвященных
Java и XML. Мы уже рассмотрели основы применения XML в Java, уделили внимание
использованию API SAX, DOM, JDOM и JAXP  для работы с XML, а также вопросам
использования и создания XML-документов. Теперь, когда вы понимаете, как
обеспечить взаимодействие XML со своим кодом, можно перейти к обсуждению
конкретных приложений. В следующих шести главах представлены важнейшие
приложения XML и, в частности, их реализации в мире технологий Java. Хотя уже
существуют тысячи важнейших приложений XML, темы, рассматриваемые в этих
главах, постоянно находятся в центре внимания и несут в себе значительный
потенциал для изменения традиционных процессов разработки приложений.
```

```
   </paragraph>

   <authorComment>Майк, ты не считаешь, что следующая врезка несколько
велика? Я вполне могу ее выбросить, если и без нее все понятно.
   </authorComment>

    <sidebar title="Чем больше все меняется, тем больше остается неизменным">
Те, кто читал первое издание книги, обнаружат, что большая часть информации о
Cocoon в этой главе осталась прежней. Хоть я и обещал, что к этому времени
выйдет Cocoon 2, и собирался написать целую главу, посвященную ему, все
развивалось не так быстро, как я того ожидал. Стефано Маццокки (Stefano
Mazzochi) - автор Cocoon - решил наконец закончить учебу (так держать,
Стефано!), и в результате разработка Cocoon 2 замедлилась. Основная
разработка по-прежнему сосредоточена на Cocoon 1.x, так что ее и следует
использовать. Раздел, посвященный Cocoon 2, обновлен и отражает ожидаемые
нововведения. В ближайшие месяцы следует ожидать новых книг от O'Reilly,
посвященных системе Cocoon. - -</sidebar>

   <paragraph> Первая актуальная тема, которую я рассмотрю, связана с
приложением XML, которое вызвало наибольший резонанс среди разработчиков на
XML и Java. Речь идет о системе веб-публикации. Хотя я постоянно говорю, что
визуализацию содержимого документа, возможно, переоценивают по сравнению со
значимостью переносимости данных, которую обеспечивает XML, применение XML
для формирования представления данных все же очень важно. Потребность в
визуализации возрастает, когда речь заходит о веб-ориентированных
приложениях. -</paragraph>
  </chapter>

 </contents>
</book>
```

Добавив примечание в XML-документ, вставим соответствующий элемент в таблицу стилей XSL *JavaXML.fo.xsl*:

```
<xsl:template match="sidebar">
  <fo:block font-size="14pt"
            font-style="italic"
            color="blue"
            space-before.optimum="16pt"
            text-align="center">
    <xsl:value-of select="@title" />
  </fo:block>
  <fo:block font-size="12pt"
            color="blue"
            space-before.optimum="16pt"
            text-align="justify">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="authorComment">
 <fo:block font-size="10pt"
           font-style="italic"
```

```
            color="red"
            space-before.optimum="12pt"
            text-align="justify">
      <xsl:apply-templates/>
    </fo:block>
  </xsl:template>
```

Примечания будут отображены чуть более мелким курсивным шрифтом красного цвета. Теперь можно преобразовать XML-документ в XSP-страницу (как показано в примере 10.9), добавив инструкции обработки для Cocoon и заключив элементы в новый корневой элемент, `xsp:page`.

Пример 10.9. Преобразование документа chapterTen.xml в XSP-страницу

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

<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="XSL/JavaXML.fo.xsl" type="text/xsl"?>

<xsp:page language="java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core"
>
<book>
 <cover>
  <title>Java и XML</title>
  <author>Бретт Мак-Лахлин</author>
 </cover>

 <contents>
  <chapter title="Системы веб-публикации" number="10">
  <!-- Текст главы -->
  </chapter>
 </contents>
</book>
</xsp:page>
```

Прежде чем добавить XSP-логику, позволяющую определять, надо ли показывать примечание, создадим простую HTML-страницу, увидев которую, пользователь сможет понять, является ли он редактором книги. В реальном приложении это могла бы быть страница, осуществляющая аутентификацию и определяющая полномочия пользователя. В нашем примере она позволяет выбрать, является ли пользователь автором, редактором или только любопытствующим читателем, а также ввести пароль для проверки. Выполняющая эту задачу HTML-страница показана в примере 10.10. Сохраните этот файл под именем *entry.html* в корневом каталоге документов контекста.

Пример 10.10. Начальная страница для XSP-страницы chapterTen.xml

```
<html>
 <head>
```

```
  <title>Welcome to the Java and XML Book in Progress</title>
 </head>

 <body>
  <h1 align="center">Процесс написания книги <i>Java и XML</i></h1>
  <center>
   <form action="/cocoon/chapterTen.xml" method="POST">
    Выберите свою роль:
    <select name="userRole">
     <option value="author">Я - автор </option>
     <option value="editor">Я - редактор</option>
     <option value="reader">Я - читатель</option>
    </select>
    <br />
    Введите ваш пароль:
    <input type="password" name="password" size="8" />
    <br /><br />
    <input type="submit" value="Go!" />
   </form>
  </center>
 </body>
</html>
```

Заметьте также, что данные из HTML-формы отправляются непосредственно XSP-странице. В этом примере XSP-страница работает как сервлет. Она считывает параметры запроса, определяет, какую роль выбрал пользователь, производит проверку подлинности, используя указанный пароль, и в заключение определяет, нужно ли показывать комментарии. Для начала определим логическую переменную – она будет содержать результат сравнения параметров запроса и позволит определить, является ли пользователь автором или редактором и правильный ли пароль введен. Если значение переменной истинно, мы отображаем элемент `authorComment`. Заключим `authorComment` в инструкции XSP, как показано ниже:

```
<xsp:logic>
 boolean authorOrEditor = false;

 // Выясняем, является  пользователь автором или редактором

 if (authorOrEditor) {
   <xsp:content>
     <authorComment>Mike - Do you think the following sidebar is
     a little much? I could easily leave it out if it's still
     clear without it.</authorComment>
   </xsp:content>
 }
</xsp:logic>
```

Этот фрагмент должен казаться вам вполне естественным: если не принимать в расчет теги, специфичные для XSP, мы всего лишь определяем переменную и проверяем ее значение. Если значение перемен-

ной истинно, элемент authorComment добавляется в результат, выдаваемый XSP-страницей, в противном случае этот элемент не включается. Интересно отметить, что собственно конечный XML-документ помещается внутрь блока xsp:logic, в элемент xsp:content (который, в свою очередь, содержится во внешнем элементе xsp:page). Это гарантирует, что процессор XSP не будет пытаться интерпретировать никакие элементы или текст внутри блока как структуры XSP. Тот же самый код в JSP мог бы выглядеть примерно так:

```
<%
 if (authorOrEditor) {
%>
    <authorComment> Майк, ты не считаешь, что следующая врезка несколько
      велика? Я вполне могу ее выбросить, если и без нее все понятно.
      </authorComment>
<%
 }
%>
```

Этот код не является достаточно структурированным, поскольку блок JSP заканчивается перед тем, как начинается элемент authorComment. После элемента добавляется новый блок, который закрывает скобку, открытую в предыдущем блоке JSP. Здесь очень легко перепутать структуры кода или забыть добавить соответствующие блоки JSP. Принципы XSP требуют, чтобы каждый открытый элемент был закрыт (стандартная корректность данных XML) и чтобы один блок кода соответствовал одному элементу.

После того как логические структуры подготовлены, XSP-странице остается лишь интерпретировать параметры запроса. Можно использовать встроенную переменную XSP request, имитирующую объект класса javax.servlet.http.HttpServletRequest. Следующие дополнения к коду читают значения параметров запроса userRole и password (если они существуют). Затем значение параметра userRole сравнивается с типами пользователей, которые имеют право просматривать комментарии («author» и «editor»). Если соответствие имеет место, проверяется также и пароль. Если пароль совпадает с ключом для указанной роли, логическая переменная устанавливается в значение «истина», и элемент authorComments становится частью конечного XML-документа:

```
  <xsp:logic>
   boolean authorOrEditor = false;

   // Проверяем, является пользователь автором или редактором
<![CDATA[
String[] roleValues = request.getParameterValues("userRole");
String[] passwordValues = request.getParameterValues("password");
if ((roleValues != null) && (passwordValues != null)) {
  String userRole = roleValues[0];
  String password = passwordValues[0];
```

```
    if (userRole.equals("author") && password.equals("brett")) {
      authorOrEditor = true;
    } else
      if (userRole.equals("editor") && password.equals("mike")) {
      authorOrEditor = true;
    }
  }
  ]]>

     if (authorOrEditor) {
...
```

Обратите внимание, что приличный по размеру фрагмент кода содержится внутри секции CDATA. Помните, что страницы XSP обрабатываются как документы XML, и должны следовать всем правилам данных XML. Но двойные кавычки и амперсанды, используемые во фрагментах кода Java, недопустимы в XML-документах. Вместо того чтобы экранировать все эти символы и получить в результате весьма странный фрагмент XSP, можно воспользоваться тегом CDATA и писать стандартный код Java. Не имея такой возможности, мы создали бы следующий код:

```
<xsp:logic>
 boolean authorOrEditor = false;

 String[] roleValues =
   request.getParameterValues(&quot;userRole&quot;);
 String[] passwordValues =
   request.getParameterValues(&quot;password&quot;);
 if ((roleValues != null) &amp;&amp;
     (passwordValues != null)) {
   String userRole = roleValues[0];
   String password = passwordValues[0];
   if (userRole.equals("author") &amp;&amp;
       password.equals("brett")) {
     authorOrEditor = true;
   } else
   if (userRole.equals("editor") &amp;&amp;
       password.equals("mike")) {
     authorOrEditor = true;
   }
 }
...
</xsp:logic>
```

Теперь можно проверить стартовую страницу и получаемый PDF-документ, сгенерированный из данных XML. Если указать в броузере адрес *http://<hostname>:<port>/cocoon/entry.html*, то должен получиться результат, сходный с тем, что показан на рис. 10.8.

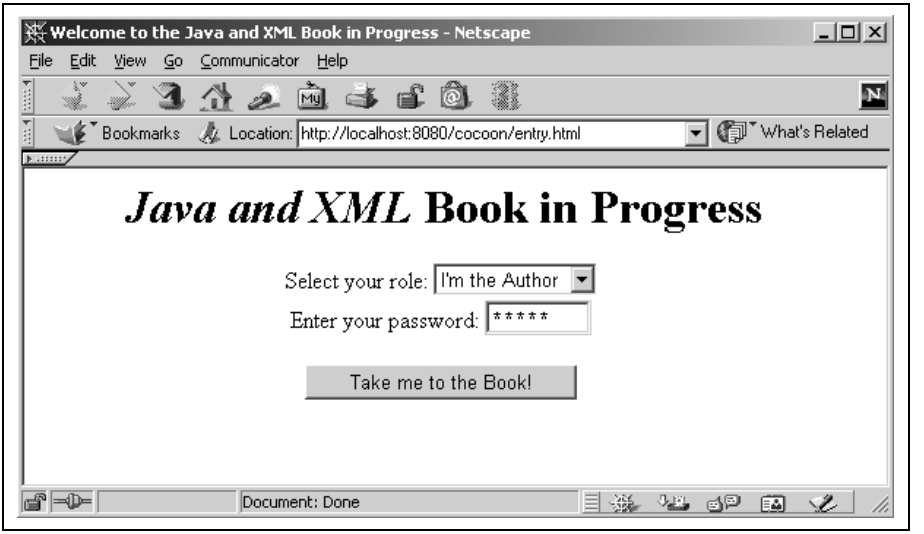

Рис. 10.8. Стартовая страница для XSP-страницы chapterTen.xml

Выберите роль автора и укажите пароль «brett» либо выберите роль редактора с паролем «mike». В обоих случаях будет получен PDF-документ, показанный на рис. 10.9.

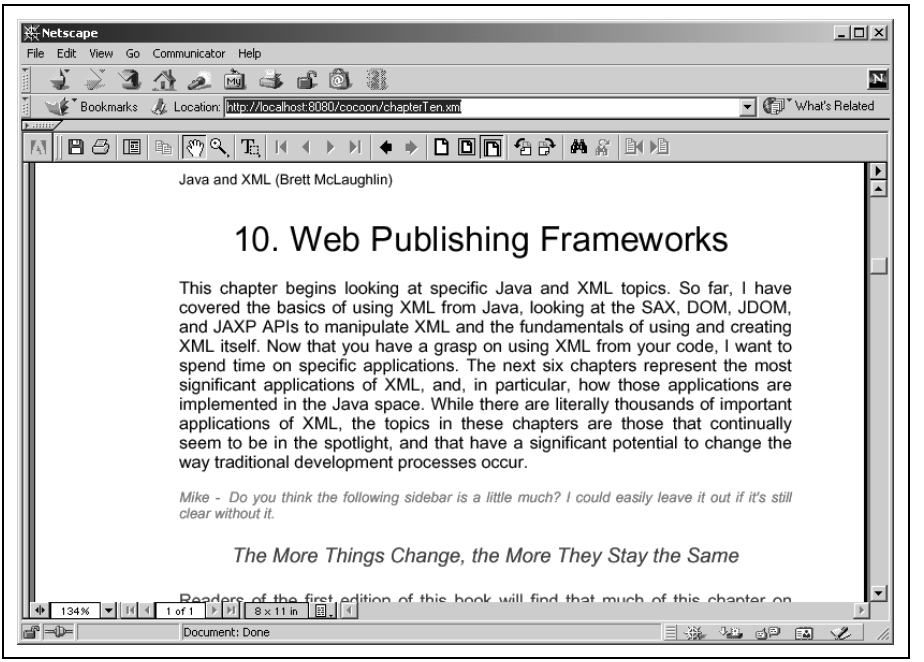

Рис. 10.9. Документ, отображаемый для автора или редактора

Здесь осталось сделать лишь одно – отделить логику страницы от ее информационного наполнения. Как страницы JSP допускают использование JavaBeans для изоляции содержимого и представления от логики компонента приложения, так страницы XSP, в свою очередь, допускают применение *библиотек тегов* (*tag libraries*). Тег библиотеки может быть связан с фрагментом логики приложения, выполнение которого происходит, если тег встречается в документе.

Использование библиотек тегов XSP

Помимо отображения комментариев в зависимости от пользователя, XSP-страница должна указывать, что глава находится в черновом состоянии. Можно вывести текущую дату, чтобы показать дату черновика (когда глава завершена, дата фиксируется). Вместо применения внедренного кода на Java для загрузки текущей даты можно просто создать для этой цели пользовательскую библиотеку тегов. Также заслуживает внимания процесс создания элемента XSP, который получает номер и название главы, а затем формирует полный заголовок. Эта функция также будет управлять отображением даты для черновика. Прежде всего, необходимо создать библиотеку тегов, доступную из страницы XSP. Библиотека тегов во многом основана на таблице стилей XSL. Можно начать с каркаса, показанного в примере 10.11, который в качестве результата выдает все, что получает. Сохраните этот каркас в файле *JavaXML.xsp.xsl* в подкаталоге *XSL/*. Не забудьте включить объявление пространства имен `javaxml2`, поскольку оно будет использовано для отбора элементов, применяемых в страницах XSP и принадлежащих данному пространству имен.

Пример 10.11. Каркас библиотеки тегов XSP

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

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsp="http://www.apache.org/1999/XSP/Core"
  xmlns:javaxml2="http://www.oreilly.com/javaxml2"
>
  <xsl:template match="xsp:page">
    <xsp:page>
      <xsl:copy>
        <xsl:apply-templates select="@*"/>
      </xsl:copy>

      <xsl:apply-templates/>
    </xsp:page>
  </xsl:template>

  <xsl:template match="@*|*|text()|processing-instruction()">
    <xsl:copy>
      <xsl:apply-templates
```

```
            select="@*|*|text()|processing-instruction()"/>
      </xsl:copy>
    </xsl:template>

  </xsl:stylesheet>
```

Создав шаблон для элемента `xsp:page`, можно гарантировать, что в этой таблице стилей – или, в терминах XSP, *таблице логики* (*logicsheet*) – осуществляется выборка и обработка всех таких элементов. Теперь можно добавить методы Java, вызываемые для шаблонов из этой таблицы логики:

```
<xsl:template match="xsp:page">
  <xsp:page>
    <xsl:copy>
      <xsl:apply-templates select="@*"/>
    </xsl:copy>

    <xsp:structure>
      <xsp:include>java.util.Date</xsp:include>
      <xsp:include>java.text.SimpleDateFormat</xsp:include>
    </xsp:structure>

    <xsp:logic>
      private static String getDraftDate() {
        return (new SimpleDateFormat("MM/dd/yyyy"))
          .format(new Date());
      }

      private static String getTitle(int chapterNum,
                                     String chapterTitle) {
        return chapterNum + ". " + chapterTitle;
      }
    </xsp:logic>

    <xsl:apply-templates/>
  </xsp:page>
</xsl:template>
```

Здесь представлено несколько новых элементов XSP. Во-первых, структура `xsp:structure`, которая содержит несколько инструкций `xsp:include`. Они работают точно так же, как их аналог в Java – оператор `import`, обеспечивая доступ к указанным классам Java по краткому имени (а не по полному, содержащему название пакета). Затем таблица логики определяет и реализует два метода: один из них формирует заголовок на основе номера главы и ее названия, а другой возвращает текущую дату в виде отформатированной строки (`String`). Эти методы доступны для всех элементов из данной таблицы логики.

Теперь создадим элемент, который определяет, когда результат выполнения XSP должен заменять элемент XML. Пространство имен, связанное с префиксом `javaxml2`, уже определено в корневом элементе документа, и его можно использовать в качестве пространства имен

для элементов библиотеки тегов. Добавьте следующий шаблон в таблицу логики:

```
<!-- Создание отформатированного заголовка -->
<xsl:template match="javaxml2:draftTitle">
  <xsp:expr>getTitle(<xsl:value-of select="@chapterNum" />,
                     "<xsl:value-of select="@chapterTitle" />")
  </xsp:expr> (<xsp:expr>getDraftDate()</xsp:expr>)
</xsl:template>

<xsl:template match="@*|*|text()|processing-instruction()">
  <xsl:copy>
    <xsl:apply-templates
         select="@*|*|text()|processing-instruction()"/>
  </xsl:copy>
</xsl:template>
```

Когда в документе, из которого доступна эта библиотека тегов, встречается элемент `javaxml2:draftTitle` (или просто `draftTitle`, если пространство имен по умолчанию связано с *http://www.oreilly.com/javaxml2*), то значение, возвращаемое методом `getTitle()`, предваряет значение, возвращаемое методом `getDraftDate()`. Элемент `javaxml2:draftTitle` требует также наличия двух атрибутов: номера главы и ее названия. Помещая вызов метода внутрь пары тегов `<xsp:expr>`, мы указываем процессору XSP, что происходит вызов определенного метода. Чтобы указать, что второй аргумент (название главы) является строкой, мы заключаем его в кавычки. Номер главы должен интерпретироваться как целочисленное значение, так что он остается без кавычек.

Закончив составление таблицы логики XSP (которая также доступна на сайте этой книги), необходимо сделать ее доступной для системы Cocoon. Это можно сделать одним из двух способов. Первый – указать местоположение файла в виде URI, который позволяет среде исполнения сервлетов (а следовательно, системе Cocoon) найти таблицу логики. Например, чтобы добавить таблицу логики XSP к набору ресурсов системы Cocoon с помощью URI соответствующего файла в Unix-системе, можно добавить следующую строку в файл *cocoon.properties*:

```
# Установка библиотек, связанных с данным пространством имен.
# Используйте синтаксис:
#   processor.xsp.logicsheet.<namespace-tag>.<language> = URL для файла
# где "URL для файла" обычно начинается с file://, если пользовательская
# библиотека находится в вашей файловой системе
processor.xsp.logicsheet.context.java  = resource://org/apache/cocoon/
processor/xsp/library/java/context.xsl
processor.xsp.logicsheet.cookie.java   = resource://org/apache/cocoon/
processor/xsp/library/java/cookie.xsl
processor.xsp.logicsheet.global.java   = resource://org/apache/cocoon/
processor/xsp/library/java/global.xsl
```

```
processor.xsp.logicsheet.request.java  = resource://org/apache/cocoon/
processor/xsp/library/java/request.xsl
processor.xsp.logicsheet.response.java = resource://org/apache/cocoon/
processor/xsp/library/java/response.xsl
processor.xsp.logicsheet.session.java  = resource://org/apache/cocoon/
processor/xsp/library/java/session.xsl
processor.xsp.logicsheet.util.java     =
  resource://org/apache/cocoon/processor/xsp/library/java/util.xsl
processor.xsp.logicsheet.sql.java      =
  resource://org/apache/cocoon/processor/xsp/library/sql/sql.xsl
processor.xsp.logicsheet.esql.java      =
  resource://org/apache/cocoon/processor/xsp/library/sql/esql.xsl
processor.xsp.logicsheet.fp.java       =
  resource://org/apache/cocoon/processor/xsp/library/fp/fp.xsl

processor.xsp.library.JavaXML.java =
  file:///usr/local/jakarta-tomcat/webapps/cocoon/XSL/JavaXML.xsp.xsl
```

Для систем Windows это может выглядеть следующим образом:

```
# Установка библиотек, связанных с данным пространством имен.
# Используйте синтаксис:
#   processor.xsp.logicsheet.<namespace-tag>.<language> = URL для файла
# где "URL для файла" обычно начинается с file://, если пользовательская
# библиотека находится в вашей файловой системе
processor.xsp.logicsheet.context.java  = resource://org/apache/cocoon/
processor/xsp/library/java/context.xsl
processor.xsp.logicsheet.cookie.java   = resource://org/apache/cocoon/
processor/xsp/library/java/cookie.xsl
processor.xsp.logicsheet.global.java   = resource://org/apache/cocoon/
processor/xsp/library/java/global.xsl
processor.xsp.logicsheet.request.java  = resource://org/apache/cocoon/
processor/xsp/library/java/request.xsl
processor.xsp.logicsheet.response.java = resource://org/apache/cocoon/
processor/xsp/library/java/response.xsl
processor.xsp.logicsheet.session.java  = resource://org/apache/cocoon/
processor/xsp/library/java/session.xsl
processor.xsp.logicsheet.util.java     =
  resource://org/apache/cocoon/processor/xsp/library/java/util.xsl
processor.xsp.logicsheet.sql.java      =
  resource://org/apache/cocoon/processor/xsp/library/sql/sql.xsl
processor.xsp.logicsheet.esql.java     =
  resource://org/apache/cocoon/processor/xsp/library/sql/esql.xsl
processor.xsp.logicsheet.fp.java       =
  resource://org/apache/cocoon/processor/xsp/library/fp/fp.xsl

processor.xsp.library.javaxml2.java =
  file:///C:/java/jakarta-tomcat/webapps/cocoon/XSL/JavaXML.xsp.xsl
```

Хотя такое решение вполне подходит для тестирования, оно не слишком эффективно в смысле уменьшения зацепления таблиц логики

и среды исполнения сервлетов. Кроме того, оно влечет за собой значительные накладные расходы, связанные с сопровождением: чтобы сделать доступной новую таблицу логики, придется добавить новые строки в файл свойств *cocoon.properties*. Альтернативный метод загрузки таблиц логики связан с описанием ресурса в `classpath`-переменной среды исполнения сервлетов. Мы получаем возможность поместить все пользовательские таблицы логики в *jar*-файл и сослаться на этот файл в пути к классам (в Tomcat это делается простым добавлением архива в каталог *lib*/!) среды исполнения сервлетов. Кроме того, в этот *jar*-файл можно помещать новые таблицы логики, а значит, есть возможность централизованного хранения таблиц логики XSP. Для создания jar-файла, содержащего нашу таблицу логики, выполните следующую команду в подкаталоге *XSL*/ корневого каталога документов своего веб-сервера:

```
jar cvf logicsheets.jar JavaXML.xsp.xsl
```

Перенесите созданный архив *logicsheets.jar* в каталог *<TOMCAT_HOME>/lib*/ (в нем уже хранятся остальные библиотеки Cocoon). Это гарантирует, что Tomcat загрузит библиотеку при запуске. Обеспечив доступ к таблице логики, можно теперь указать системе Cocoon, где следует искать ссылки на пространство имен *javaxml2* в страницах XSP. Отредактируйте файл *cocoon.properties*; найдите раздел, в котором перечислены различные XSP-ресурсы системы Cocoon, и добавьте новую ссылку на таблицу стилей:

```
# Установка библиотек, связанных с заданным пространством имен.
# Используйте синтаксис:
#   processor.xsp.logicsheet.<namespace-tag>.<language> = URL для файла
# где "URL для файла" обычно начинается с file://, если пользовательская
# библиотека находится в вашей файловой системе
processor.xsp.logicsheet.context.java  = resource://org/apache/cocoon/
processor/xsp/library/java/context.xsl
processor.xsp.logicsheet.cookie.java   = resource://org/apache/cocoon/
processor/xsp/library/java/cookie.xsl
processor.xsp.logicsheet.global.java   = resource://org/apache/cocoon/
processor/xsp/library/java/global.xsl
processor.xsp.logicsheet.request.java  = resource://org/apache/cocoon/
processor/xsp/library/java/request.xsl
processor.xsp.logicsheet.response.java = resource://org/apache/cocoon/
processor/xsp/library/java/response.xsl
processor.xsp.logicsheet.session.java  = resource://org/apache/cocoon/
processor/xsp/library/java/session.xsl
processor.xsp.logicsheet.util.java     =
  resource://org/apache/cocoon/processor/xsp/library/java/util.xsl
processor.xsp.logicsheet.sql.java      =
  resource://org/apache/cocoon/processor/xsp/library/sql/sql.xsl
processor.xsp.logicsheet.esql.java      =
  resource://org/apache/cocoon/processor/xsp/library/sql/esql.xsl
processor.xsp.logicsheet.fp.java       =
```

```
    resource://org/apache/cocoon/processor/xsp/library/fp/fp.xsl
processor.xsp.logicsheet.javaxml2.java = resource://JavaXML.xsp.xsl
```

Данная таблица логики не располагается ни в одном из подкаталогов архива *logicsheets.jar*, поэтому можно просто использовать ее имя в качестве идентификатора ресурса. Наконец, перезапустите среду исполнения сервлетов (это также гарантирует, что Tomcat автоматически загрузит новую библиотеку). При этом файл *cocoon.properties* будет загружен заново, и таблица логики станет доступной для использования. Поскольку для обработки запросов применяется система Cocoon, любая XSP-страница, в которой объявлено, что она использует пространство имен javaxml2, получит доступ к таблице логики, описанной как библиотека javaxml2. Таким образом, в XSP-страницу следует добавить объявление пространства имен javaxml2:

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

<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="XSL/JavaXML.fo.xsl" type="text/xsl"?>

<xsp:page language="java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core"
          xmlns:javaxml2="http://www.oreilly.com/javaxml2"
>
<book>
  <!-- Содержимое книги -->
</book>
</xsp:page>
```

Теперь, когда библиотека тегов доступна для использования, можно, наконец, добавить элемент javaxml2:draftTitle в XML-документ *chapterTen.xml*:

```
<contents>
  <chapter title="Web Publishing Frameworks" number="10">
    <javaxml2:draftTitle chapterNum="10"
                         chapterTitle="Web Publishing Framework" />
...
```

Замените жестко закодированный заголовок главы элементом, определенным в библиотеке тегов XSP, изменив следующим образом таблицу стилей *JavaXML.fo.xsl*:

```
  <xsl:template match="chapter">
    <fo:block font-size="24pt"
              text-align-last="center"
              space-before.optimum="24pt">
<!--
      <xsl:value-of select="@number" />.
      <xsl:value-of select="@title" />
-->
```

```
        <xsl:apply-templates/>
      </fo:block>
    </xsl:template>
```

В результате должен быть сгенерирован заголовок с номером главы, ее названием и датой создания черновика. При обращении к новой версии XSP-страницы выдается результат, показанный на рис. 10.10.

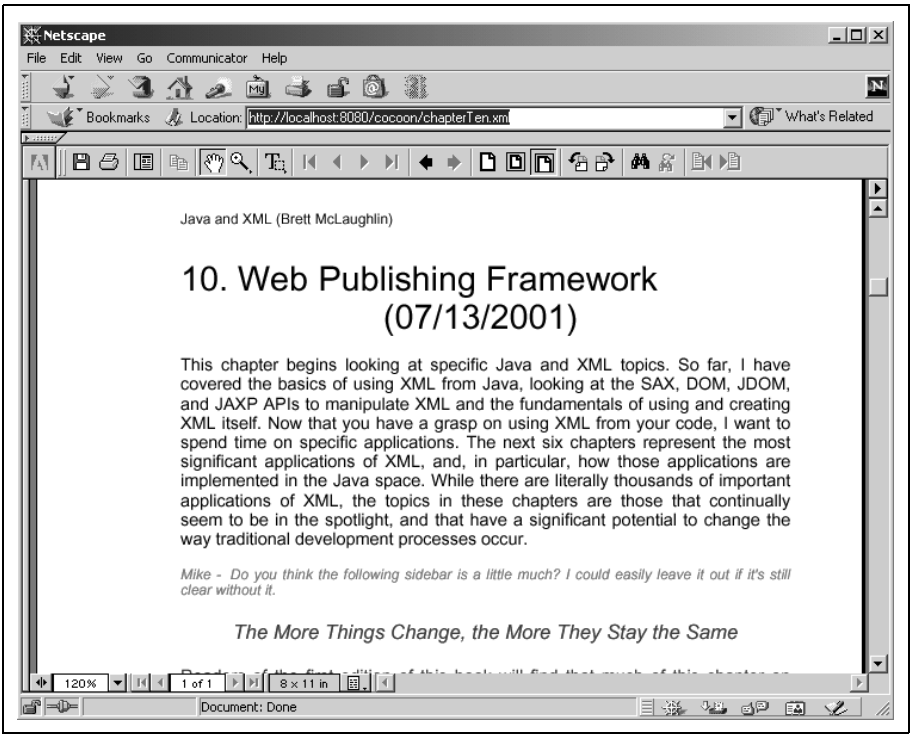

Рис. 10.10. Вывод, получаемый при использовании библиотеки тегов XSP

Мы лишь коснулись азов технологии XSP. Но даже этот простой пример позволяет преобразовать заголовок в другой формат, когда глава будет завершена, не меняя ни содержания, ни представления страницы, а внеся изменение лишь в таблицу логики XSP. Аналогичным образом XSP позволяет создавать очень строгие границы, разделяющие представление, информационное наполнение и логику приложения. Серверные компоненты Java, такие как Enterprise JavaBeans, могут добавить в это «уравнение» еще и бизнес-логику. Лучше избегать менее гибких решений, подобных JSP, которые довольно жестко связаны с HTML и представлением данных. Применение XSP позволяет сократить зацепление компонентов и, следовательно, представляет собой более предпочтительное решение в смысле разработки приложений. XSP также обещает стать ключевой технологией в системе Cocoon 2.0, которую мы сейчас рассмотрим.

Cocoon 2.0 и выше

Следующее поколение Cocoon, система Cocoon 2.0, должна стать гигантским шагом вперед в области систем веб-публикации. Система Cocoon 1.x, основанная главным образом на связке XML+XSL, попрежнему имеет ряд серьезных ограничений. Прежде всего, она не столь значительно сокращает расходы на сопровождение крупных сайтов. И хотя один XML-документ может быть преобразован в различные клиентские форматы, количество документов от этого не сокращается. В общем случае результатом являются либо длинные URI (скажем, */content/publishing/books/javaxml/contents.xml*), либо большое количество виртуальных отображений путей (*/javaxml* соответствующий */content/publishing/books/javaxml*), либо комбинация обоих случаев. Кроме того, достаточно сложно достичь – и еще сложнее поддерживать – строгое разделение информационного наполнения, представления и логики.

Cocoon 2 концентрирует внимание на совершенствовании разграничения между этими различными уровнями и, следовательно, сокращении расходов на сопровождение. Основой архитектуры Cocoon 2.0 является технология XSP. Кроме того, карта сайта позволяет скрыть различия между XSP, XML и статическими страницами HTML от любопытствующего пользователя. В системе Cocoon 2 появится более совершенный вариант предварительной компиляции и более эффективное управление памятью, что даст ей еще больше преимуществ перед системой Cocoon 1.x, чем Cocoon 1.x имела по сравнению со стандартным веб-сервером.

Среда исполнения сервлетов: отображения

Одно из значительных изменений в Cocoon 2 состоит в том, что больше нет необходимости использовать примитивный механизм отображений для XML-документов. Хотя эта методика очень хорошо работает в модели 1.x, она полностью оставляет управление документами, отличными от XML, на совести веб-мастера, которым может оказаться совсем не тот человек, который отвечает за XML-документы. Система Cocoon 2 стремится принять на себя управление целым веб-сайтом, поэтому основной сервлет Cocoon (которым в версии 2.0 является `org.apache.cocoon.servlet.CocoonServlet`) обычно ставится в соответствие URI вроде */Cocoon*. В целях полного контроля за сайтом основной сервлет может быть связан с корневым каталогом веб-сервера (т. е. «/»). После этого URL запрашиваемых документов должны учитывать отображение: например *http://myHost.com/Cocoon/myPage.xml* или *http://myHost.com/Cocoon/myDynamicPage.xsp*.

При наличии такого отображения можно совместно хранить документы XML и статические документы HTML. Это делает возможным управление всеми файлами сервера централизованно одним человеком

или группой. Если HTML-, WML- и XML-документы необходимо хранить в одном и том же каталоге, никакой путаницы не происходит, и можно использовать единообразные URI. Cocoon 2 будет с таким же успехом выдавать HTML-документ, как и документ любого другого типа. В результате привязки корневого каталога сервера к системе публикации Cocoon система веб-публикации становится фактически невидимой для клиента.

Карта сайта

Еще одно важное нововведение в Cocoon 2 – это *карта сайта* (*sitemap*). Карта сайта обеспечивает в системе Cocoon централизованное администрирование веб-сайта. Cocoon использует эту карту сайта для принятия решений о том, как обрабатывать получаемые запросы к URI. Например, когда Cocoon получает запрос вида *http://myCocoonSite.com/Cocoon/javaxml/chapterOne.html*, сервлет системы Cocoon анализирует запрос и определяет, что реальным запрашиваемым URI является */javaxml/chapterOne.html*. Предположим, однако, что файл *chapterOne.html* должен соответствовать не статическому HTML-файлу, а преобразованию XML-документа (как в наших предыдущих примерах). Карта сайта может достаточно просто учесть такую ситуацию! Взгляните на пример 10.12.

Пример 10.12. Простая карта сайта Cocoon 2

```
<sitemap>
 <process match="/javaxml/*.html">
  <generator type="file" src="/docs/javaxml/*.xml"
  <filter type="xslt">
   <parameter name="stylesheet" value="/styles/JavaXML.html.xsl"/>
  </filter>
  <serializer type="html"/>
 </process>

 <process match="/javaxml/*.pdf">
  <generator type="file" src="/docs/javaxml/*.xml"
  <filter type="xslt">
   <parameter name="stylesheet" value="/styles/JavaXML.pdf.xsl"/>
  </filter>
  <serializer type="fop"/>
 </process>
</sitemap>
```

В этом примере система Cocoon сопоставляет URI */javaxml/chapterOne.html* с инструкцией карты сайта `/javaxml/*.html`. Она определяет, что указанный URI соответствует реальному файлу и что файл, содержащий исходные данные, следует определить, используя отображение `/docs/javaxml/*.xml`, что дает */docs/javaxml/chapterOne.xml* (имя файла, который мы хотим преобразовать). Затем применяется фильтр XSLT – применяемая таблица стилей *JavaXML.html.xsl* также задана

в карте сайта. Далее преобразованные данные выдаются пользователю. Кроме того, XML-файл может быть файлом XSP, который обрабатывается до того, как преобразуется в XML, и затем форматируется.

Таким же образом можно вывести данные в формате PDF по запросу *http://myCocoonSite.com/Cocoon/javaxml/chapterOne.pdf*, добавив всего несколько строк в приведенную выше карту сайта. Это означает, что можно полностью удалить инструкции обработки из индивидуальных XML-документов, что представляет собой значительное изменение по сравнению с Cocoon 1.x. В первую очередь, в зависимости от каталога, в котором размещены файлы, можно осуществлять единообразное применение таблиц стилей и единообразную обработку. Одно лишь создание XML-документа и помещение его в каталог */docs/javaxml/* в данном примере означает, что доступ к документу возможен как в формате HTML, так и в PDF. Так же легко можно будет изменить таблицу стилей, применяемую для набора документов, что представляет собой трудную и утомительную задачу в Cocoon 1.x. Вместо того чтобы вносить изменения в каждый XML-документ, необходимо изменить всего лишь одну строку в карте сайта.

Карта сайта Cocoon еще находится в стадии разработки, и ко времени выхода окончательной версии Cocoon 2.0 в ее формат и структуру, вероятно, будут внесены некоторые дополнительные изменения и усовершенствования. Чтобы быть в курсе последних событий, подпишитесь на списки рассылок *cocoon-users@xml.apache.org* и *cocoon-dev@xml.apache.org*. Подробности об участии в этих списках и в проекте Cocoon можно найти на сервере проекта Apache XML *http://xml.apache.org*.

Генераторы и процессоры

И последнее усовершенствование, которое должен включать Cocoon 2, – это прекомпилированные и событийно-ориентированные *генераторы* (*producers*) и *процессоры* (*processors*). В системе Cocoon генератор управляет преобразованием URI запроса в поток, связанный с XML-документом. Далее процессор получает входной поток (в настоящее время XML-документ, представленный в виде дерева DOM) и преобразует его в результат, пригодный для чтения клиентом. Мы не рассматриваем генераторы и процессоры для модели Cocoon 1.x, поскольку в Cocoon 2.0 они будут серьезно переработаны – любые генераторы и процессоры, используемые в настоящее время, вероятнее всего станут неприменимыми в системе Cocoon 2.0.

В системе Cocoon 2 делается переход от применения для этих структур модели DOM к SAX, которая более тесно связана с событиями, оболочкой для которой, как и прежде, будет модель DOM. Поскольку в версиях 1.x генератор должен был хранить XML-документ в памяти, соответствующая структура DOM могла становиться крайне большой. В конце концов, это приводило к истощению системных ресурсов, осо-

бенно при выполнении таких сложных задач, как крупномасштабные преобразования или работа с объектами форматирования (создание документов в формате PDF). По этим причинам в системе Cocoon 2 модель DOM будет играть роль простой оболочки для событий SAX, сообщая генераторам и процессорам легкость и эффективность.

Кроме того, в виде генераторов и процессоров будут реализованы прекомпилированные версии других форматов. Например, таблицы стилей XSL могут быть прекомпилированы в процессоры, а страницы XSP – в генераторы. Это еще больше увеличивает производительность и в то же время снимает нагрузку с клиента. Эти и другие изменения происходят по правилами компонентной модели, что делает систему Cocoon очень гибкой и легко внедряемой. Чтобы быть в курсе последних изменений, следите за информацией, публикуемой на сайте проекта Cocoon.[1]

Что дальше?

В следующей главе рассматривается технология, обеспечивающая использование XML как формата данных в важной модели запроса и ответа – XML-RPC. Вызов удаленных процедур с применением XML позволяет клиентам распределенной системы выполнять задачи на сервере или серверах, расположенных в другом сегменте сети. До последнего времени популярность RPC была низкой, главным образом, из-за волны RMI-технологий в мире Java (в первую очередь, это EJB). Однако, используя XML как формат данных, XML-RPC стал новым решением для многих проблем, которые нельзя решить ясно и эффективно другими методами. В следующей главе мы рассмотрим технологию XML-RPC и, в частности, ее реализацию на Java.

[1] *http://xml.apache.org/cocoon. – Примеч. науч. ред.*

11

XML-RPC

- *RPC и RMI: за и против*
- *Простейшее приложение*
- *Переложение нагрузки на сервер*
- *Реальные ситуации*
- *Что дальше?*

XML-RPC представляет собой специальную разновидность RPC, т. е. механизма *удаленного вызова процедур* (Remote Procedure Calls). Для начинающих разработчиков или тех, кто знаком с языком Java лишь поверхностно, удаленный вызов процедур может оказаться новой темой. Опытные специалисты-разработчики уже могли успеть позабыть о ней, поскольку RPC в последние годы вышел из моды. В этой главе я расскажу, каким образом три буковки, предваряющие аббревиатуру RPC, способны вернуть к жизни это компьютерное ископаемое, и как применять XML-RPC в мире технологий Java. В конце главы мы рассмотрим реальные приложения XML-RPC и постараемся пролить свет не только на то, как применять эту технологию, но и на то, когда это следует делать.

Участников движения объектно-ориентированного программирования, которое возникло в последние три-пять лет, от одного слова «процедура» бросает в дрожь. Процедурные языки, такие как PL/SQL и ANSI C, стали непопулярными по целому ряду весьма веских причин. Возможно, вас когда-нибудь ругали за то, что вы называли метод Java функцией или процедурой, и вы почти наверняка знаете, что лучше не писать код, похожий на «спагетти», в котором методы сцепляются один за другим в длинную строку. RPC потерял популярность в той же мере, в какой эти языки и способы написания программ. Сейчас появились новые объектно-ориентированные способы достижения тех же самых результатов, которые позволяют в большинстве случаев достичь более элегантной архитектуры и лучшей производительности. Тем более занятно, что рост популярности XML вызвал рост популярности и известности API, созданных для работы XML-RPC, и тенденцию к применению XML-RPC в определенных ситуациях, несмотря на коннотации названия.

Прежде чем пытаться применять эти API, полезно уделить некоторое время изучению RPC и сравнению этой технологии с аналогичными

решениями мира Java, в первую очередь, с *удаленным вызовом методов* (Remote Method Invocation, RMI). Решив использовать XML-RPC в приложениях (а вы, вероятнее всего, когда-нибудь захотите это сделать), имейте в виду, что вам, скорее всего, придется обосновать свой выбор другим разработчикам, особенно тем, кто, может быть, только что прочитал книги об EJB (Enterprise JavaBeans) или RMI. Разумеется, каждая из этих технологий имеет свою область применения. Понимание того, когда нужно применять каждую из них, является залогом вашего успеха не только как разработчика, но и как члена коллектива и руководителя. Учитывая важность понятий, стоящих за этими методологиями удаленного взаимодействия, рассмотрим два наиболее распространенных способа работы с объектами через сеть: RPC и RMI.

RPC и RMI: за и против

Те, кто следил за событиями последних лет, должны знать, что технологии EJB и RMI стремительно распространяются в мире Java. Вся спецификация EJB основана на принципах RMI, и написать трехзвенное приложение без использования (пусть даже косвенного) RMI весьма непросто. Другими словами, те, кто еще не знают, как применять RMI, возможно, захотят приобрести книгу «Java Enterprise in a Nutshell» (Java Enterprise. Справочник) Дэвида Флэнагана (David Flanagan), Джима Фарли (Jim Farley), Вильяма Кроуфорда (William Crawford) и Криса Магнуссона (Kris Magnusson) либо книгу Джима Фарли «Java Distributed Computing» (Java и распределенные вычисления) (обе выпущены издательством O'Reilly & Associates) и уделить некоторое время изучению этой полезной технологии.

Что такое RMI?

Вкратце RMI – это удаленный вызов методов. RMI позволяет программе вызывать методы объекта, расположенного не на той машине, на которой запущена программа. Этот принцип лежит в основе архитектуры распределенных вычислений в Java и является основой как технологии Enterprise JavaBeans (EJB), так и многих других корпоративных технологий и приложений. Если не слишком углубляться в подробности, в RMI применяются «заглушки» на стороне клиента для описания вызываемых методов удаленного объекта. Клиент работает с заглушками, которые являются интерфейсами Java, а RMI берет на себя «магию» преобразования обращений к заглушкам в сетевые вызовы. Сетевые вызовы приводят к выполнению методов на той машине, где находится реальный объект, а результаты возвращаются также по сети. В итоге заглушка возвращает результат клиенту, изначально вызвавшему метод, и клиент продолжает работу. Основная идея заключается в том, что клиенту не приходится беспокоиться о деталях работы RMI и сетевых вызовов. Благодаря заглушкам он получает возможность работать с удаленным объектом как с существующим локально.

RMI (используя протокол JRMP™ – Java's Remote Method Protocol) оставляет все сетевое взаимодействие за кадром, позволяя клиенту обрабатывать базовое исключение (`java.rmi.RemoteException`), уделяя больше времени бизнес-правилам и логике приложения. RMI позволяет использовать разные протоколы (такие как Internet Inter-ORB Protocol, IIOP), предоставляя возможность взаимодействия между объектами Java и объектами CORBA, зачастую созданными на других языках, таких как C или C++.

Однако за преимущества RMI приходится платить. Во-первых, применение RMI является ресурсоемким. Протокол JRMP дает очень низкую производительность, а написание нового протокола удаленного взаимодействия представляет собой отнюдь не простую задачу. RMI-вызовы клиентов приводят к открытию и последующему сопровождению сокетов, а количество сокетов может повлиять на производительность системы, особенно когда доступ к системе осуществляется через сеть (что требует открытия дополнительных сокетов для организации доступа). RMI также требует наличия сервера или провайдера, к которым привязываются объекты. Чтобы объект стал доступен приложениям, он должен быть связан с именем на одном из таких серверов. Эта система требует применения реестра RMI, сервера каталогов LDAP или различных служб JNDI (Java Naming and Directory Interface). Наконец, применение RMI может быть связано с созданием больших объемов кода, даже при наличии вспомогательных серверных классов RMI, входящих в JDK. Необходимо создать удаленный интерфейс, описывающий доступные для вызова методы (а также некоторое количество других интерфейсов, если используется EJB). Как следствие, введение дополнительного метода в класс сервера приводит к изменению интерфейса и перекомпиляции клиентских заглушек, что зачастую нежелательно, а иногда и невозможно.

Что такое RPC?

RPC – это *удаленный вызов процедур*. В то время как RMI позволяет взаимодействовать непосредственно с объектом Java, технология RPC разработана в несколько ином стиле. Вместо работы с объектами RPC позволяет вызывать через сеть самостоятельные методы. Возможности взаимодействия при этом более ограничены, но интерфейс взаимодействия для клиента несколько упрощается. Можно рассматривать RPC как средство использования «служб» на удаленных машинах, тогда как RMI позволяет использовать на удаленных машинах «серверы». Тонкое отличие заключается в том, что RMI обычно полностью управляется клиентом и все события связаны с удаленными вызовами методов. RPC обычно строится скорее как класс или набор классов, которые выполняют определенные задачи как с участием, так и без участия клиента; тем не менее, время от времени эти классы обслуживают запросы от клиентов и выполняют для них небольшие задачи. Рассмотрим несколько примеров, которые помогут разъяснить эти понятия.

Хотя среда RPC не столь интерактивна, как RMI, она предоставляет ряд значительных преимуществ. RPC позволяет системам с различной архитектурой работать вместе. В то время как RMI обеспечивает применение протокола IIOP для связи Java с серверами и клиентами CORBA, RPC делает возможным буквально любой тип взаимодействия приложений, поскольку транспортным протоколом может быть HTTP. Поскольку практически любой применяемый сегодня язык программирования имеет какие-либо механизмы HTTP-взаимодействия, технология RPC является весьма привлекательной для случаев, когда приложение должно работать с существующими (и созданными ранее) системами. Как правило, реализации RPC менее требовательны к ресурсам, чем RMI (особенно когда для кодирования применяется XML, о чем мы поговорим позже). И если RMI зачастую приходится загружать целые классы Java по сети (такие как код апплетов и вспомогательные классы для EJB), то в RPC можно передать через сеть лишь параметры запроса и получить результат, закодированный обычно в виде текстовых данных. RPC также очень хорошо сочетается с моделью API, обеспечивая системам, которые не являются частью определенного приложения, доступ к данным этого приложения. Это означает, что изменения в вашем сервере не должны приводить к изменению кода клиентских приложений. Передавая лишь текстовые данные и запросы, можно добавлять методы без перекомпиляции клиента, и для использования этих новых методов достаточно минимальных изменений.

Традиционно проблемой RPC было кодирование передаваемых данных. Вообразите себе попытку представить объекты Java типа `Hashtable` или `Vector` в весьма простом текстовом формате. Если учесть, что эти структуры могут, в свою очередь, содержать объекты других типов Java, реализовать такое представление данных становится очень сложно. В то же время, формат должен остаться применимым для самых разных языков программирования, в противном случае преимущества RPC сводятся на нет. До недавнего времени обнаруживалась обратная зависимость между качеством и практичностью кодирования данных и его простотой. Другими словами, чем проще становилось представление сложных объектов, тем более сложным становилось его применение в разнообразных языках программирования без фирменных расширений и специального кода. Тщательно продуманные текстовые форматы данных не были стандартизованы, так что для их использования требовались реализации соответствующих библиотек для всех языков программирования. Думаю, читатели уже поняли, к чему я веду.

XML-RPC

Самым главным препятствием на пути применения RPC традиционно оказывалось кодирование данных. Но вот появился XML – и это изменило все! XML обеспечил не только очень простое, текстовое представ-

ление данных, а стандарт структуры этих данных. Опасения по поводу закрытости этого технологического решения были развеяны, когда консорциум W3C опубликовал спецификацию XML 1.0, гарантирующую разработчикам, использующим RPC, что XML в ближайшее время никуда не исчезнет. К тому же SAX обеспечил нетребовательный к ресурсам, стандартный способ доступа к данным XML, значительно упрощая реализацию библиотек RPC. После этого разработчикам XML-RPC оставалось только реализовать передачу через протокол HTTP (то, что люди уже делали в течение многих лет) и специальные API для кодирования и декодирования данных. После выхода нескольких бета-версий библиотек XML-RPC стало ясно, что XML оказался очень быстрым и нетребовательным к ресурсам средством кодирования, обеспечивающим большую производительность библиотек XML-RPC, чем ожидалось. Сегодня XML-RPC представляет собой стабильное и жизнеспособное решение для удаленного вызова процедур.

XML-RPC предоставляет простой способ обеспечения каналов связи с приложением и его службами; каналов, которые могут использоваться клиентами приложения в других отделах и даже компаниях. Кроме того, данная технология отделяет API от языка Java для случаев, когда клиенты не могут пользоваться языком Java непосредственно. Наконец, XML-RPC устраняет RMI из числа технологий, которые необходимо изучать для работы с распределенными службами (по крайней мере, изначально). Эта глава посвящена созданию клиента и сервера XML-RPC; также мы рассмотрим примеры того, как сервер может работать независимо от клиентов, обеспечивая при этом интерфейсы для работы и доступа к данным через XML-RPC. Хотя в этой главе мы не будем углубляться в изучение RMI, но будем всякий раз сравнивать решения, основанные на XML-RPC и RMI, обращая внимание на то, почему для ряда задач следует предпочесть XML-RPC.

Простейшее приложение

Вероятно, читатели заинтересовались, может ли XML-RPC решить какие-то из проблем, возникающих при разработке приложений. Чтобы детально разобраться в XML-RPC, мы сейчас рассмотрим создание реально работающего кода Java с использованием технологии XML-RPC. Соблюдая традиции программирования, начнем с простейшей программы типа «Здравствуй, мир!». Посмотрим, как определить сервер XML-RPC и зарегистрировать на нем некий обработчик. Этот обработчик принимает в качестве параметра строку Java, хранящую имя пользователя, и возвращает строку «Привет», скомбинированную с именем пользователя; например, данный метод может вернуть «Привет, Ширли». Затем понадобится сделать наш обработчик доступным для клиентов XML-RPC. Наконец, будет продемонстрировано создание простого клиента, который будет подключаться к серверу и запрашивать вызов этого метода.

На практике сервер XML-RPC и обработчик располагаются на одной машине, вероятно, на весьма мощном сервере, а клиент, удаленно вызывающий процедуры, – на другой. Однако те, в чьем распоряжении нет нескольких машин, могут испытывать примеры и на одной. Хотя в этом случае они будут работать гораздо быстрее, чем реальные клиент и сервер, вы сможете увидеть, как в целом работает система, и получите представление об XML-RPC.

Библиотеки XML-RPC

В технологию RPC, а в последнее время и в XML-RPC, вложено много труда разработчиков. Применение API SAX, DOM и JDOM для обработки XML показывает, что нет смысла изобретать велосипед, если уже существуют хорошие, даже превосходные пакеты Java, позволяющие эффективно решать ваши задачи. Центром информации по XML-RPC и источником ссылок на библиотеки для Java и других языков программирования является сервер *http://www.xmlrpc.com*. Спонсируемый компанией Userland (*http://www.userland.com*), этот сайт содержит общедоступную спецификацию XML-RPC, информацию о том, какие типы данных поддерживаются, и несколько руководств по применению XML-RPC. Но важнее всего то, что там имеется ссылка на пакет XML-RPC для Java. Следуя по ссылке на главной странице, вы попадете на сайт Ханнеса Валлнофера *http://classic.helma.at/hannes/xmlrpc/*.[1]

На этом сайте вы найдете как описание классов XML-RPC, входящих в этот пакет, так и инструкции по их применению. Загрузите архивный файл и распакуйте файлы в каталог вашей среды разработки. Классы после этого можно будет скомпилировать; среди них имеется один пример сервлета Java, которому требуются классы сервлетов (*servlet.jar* для Servlet API 2.2). Эти классы можно получить вместе с Tomcat, посетив страницу *http://jakarta.apache.org*. Для программ этой главы классы сервлетов не требуются.

Основу дистрибутива (исключая примеры для апплетов и регулярных выражений из загруженного архива) составляют тринадцать классов, входящих в пакет `helma.xmlrpc`. Они содержатся в файле *lib/xmlrpc.jar* и готовы к применению. Классы из этого дистрибутива коротко рассмотрены в табл. 11.1.

Классы SAX (из предыдущих примеров) и драйвер SAX не входят в дистрибутив, но они необходимы для работы. Другими словами,

[1] Проект Helma XML-RPC теперь существует в рамках проекта Apache по адресу *http://xml.apache.org/xmlrpc/*. В случае применения этого варианта библиотек в операторах импортирования классов следует использовать имена вида «*org.apache.xmlrpc.**» вместо «*helma.xmlrpc.**». На текущий момент библиотеки *временно* доступны для загрузки по адресу *http://jakarta.apache.org/builds/xmlrpc/*. – *Примеч. науч. ред.*

должна присутствовать полная реализация синтаксического анализатора XML, поддерживающего SAX. В примерах, как и раньше, используется Apache Xerces, хотя библиотеки XML-RPC поддерживают любой драйвер, совместимый с SAX 1.0.

Таблица 11.1. Классы XML-RPC

Класс	Назначение
`XmlRpc`	Базовый класс, реализующий вызов методов обработчиков сервера XML-RPC.
`XmlRpcClient`	Клиентский класс, реализующий RPC-обмен с применением протокола HTTP; он предоставляет поддержку прокси-серверов и cookies.
`XmlRpcClientLite`	Облегченный клиентский класс, реализующий лишь часть функциональности HTTP (без поддержки прокси и cookies).
`XmlRpcServer`	Серверный класс, реализующий прием RPC-вызовов.
`XmlRpcServlet`	Реализует возможности `XmlRpcServer` в формате сервлета.
`XmlRpcProxyServlet`	Действует в качестве прокси-сервлета для XML-RPC.
`XmlRpcHandler`	Интерфейс, используемый сервером XML-RPC для вызова методов обработчиков.
`AuthenticatedXmlRpcHandler`	То же, что и `XmlRpcHandler`, но с возможностью проверки подлинности (идентификации).
`Base64`	Кодирует в base 64 и обратно.
`Benchmark`	Измеряет время, затраченное на прохождение XML-RPC-запроса, для конкретного драйвера.
`WebServer`	Облегченный HTTP-сервер для серверов XML-RPC.

Скомпилировав все исходные файлы, убедитесь, что пути к классам XML-RPC, классам SAX, а также классам анализатора упомянуты в переменной среды `CLASSPATH`. После этого можно написать собственный код и запустить простейшее приложение. Держите исходные файлы XML-RPC под рукой, т. к. наглядная картина того, что происходит за кадром, поможет вам разобраться с примерами.

Создание обработчика

Первое, что нужно сделать – это создать класс и метод, который мы хотим вызывать удаленно. Обычно он называется *обработчиком* (*handler*). Имейте в виду, однако, что механизм сервера XML-RPC, который передает запросы, тоже часто называют обработчиком – вновь двусмысленность терминологии проявляется не с самой лучшей стороны. Более четко различие может быть обрисовано следующим образом: *обработчик XML-RPC* – это метод или набор методов, которые получают запрос XML-RPC, расшифровывают его содержимое и передают запрос классу и методу. *Обработчик ответов* (*response handler*), или просто *обработчик*, – это любой метод, который может быть вызван

обработчиком XML-RPC. Применение Java-библиотек XML-RPC избавляет нас от необходимости писать обработчик XML-RPC, поскольку он включен в класс `helma.xmlrpc.XmlRpcServer`. Остается только создать класс с одним или несколькими методами и зарегистрировать его на сервере.

Может удивить тот факт, что создание обработчика ответов не требует ни наследования классов, ни других специальных действий в исходном коде. Посредством XML-RPC можно вызвать любой метод, если типы его параметров и возвращаемого значения поддерживаются (могут быть закодированы) в XML-RPC. В таблице 11.2 перечислены все реализованные в настоящее время типы Java, которые можно применять в сигнатурах методов XML-RPC.

Таблица 11.2. Типы данных Java, поддерживаемые в XML-RPC

Тип данных в XML-RPC	Тип Java
int	int
boolean	boolean
string	String
double	double
dateTime.iso8601	Date
struct	Hashtable
array	Vector
base64	byte[]
nil	null

Хотя этот список содержит лишь небольшое количество типов, с их помощью обрабатывается основная часть запросов XML-RPC, которые могут выполняться через сеть. Метод следующего примера принимает строку (имя человека, которому мы говорим «привет») и возвращает тоже строку, а потому удовлетворяет всем требованиям. Этой информации достаточно, чтобы написать простой класс-обработчик, приведенный в примере 11.1.

Пример 11.1. Класс-обработчик с одним методом

```
package javaxml2;

public class HelloHandler {
public String sayHello(String name) {
return "Привет, " + name;
}
}
```

Это в самом деле так же просто, как кажется. Метод принимает и возвращает параметры, тип которых допустим в XML-RPC, поэтому мы без проблем зарегистрируем его на сервере XML-RPC (который вскоре создадим) и будем уверены, что он доступен для вызовов через XML-RPC.

Создание сервера

Когда обработчик готов, нужно написать программу, которая будет запускать сервер XML-RPC, отслеживать поступающие запросы и передавать эти запросы обработчику. В данном примере в качестве обработчика запросов выступает класс `helma.xmlrpc.WebServer`. Хотя можно было бы использовать Java-сервлет, применение этой облегченной реализации веб-сервера позволяет обойтись без запуска среды исполнения сервлетов на сервере XML-RPC. В конце главы мы уделим время обсуждению сервлетов в контексте сервера XML-RPC. Код примера позволяет указать порт, на котором сервер запускается, и отслеживает обращения XML-RPC до тех пор, пока не будет остановлен. Наконец, нужно зарегистрировать на сервере только что созданный класс и указать серверу другие специальные параметры приложения.

Итак, создадим каркас этого класса (показанный в примере 11.2). Вам понадобится импортировать класс `WebServer` и передать номер порта в командной строке при запуске сервера.

Пример 11.2. Каркас сервера XML-RPC

```
package javaxml2;

import helma.xmlrpc.WebServer;

public class HelloServer {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println(
                "Использование: java javaxml2.HelloServer [порт]");
            System.exit(-1);
        }

        // Запуск сервера на указанном порту
    }
}
```

Прежде чем запускать сервер, следует указать драйвер SAX, используемый для анализа и кодирования XML. По умолчанию драйвером SAX для этих библиотек является разработанный Джеймсом Кларком анализатор XP, доступный на сайте *http://www.jclark.com*. Вместо него в данном примере используется анализатор Apache Xerces (мы указываем ядру XML-RPC на класс SAX-совместимого анализатора).[1] Это

[1] В настоящее время данная библиотека XML-RPC не поддерживает SAX 2.0 и не реализует интерфейс `XMLReader`. Поскольку класс `SAXParser` Apache Xerces реализует как интерфейс `Parser` SAX 1.0, так и интерфейс `XMLReader` SAX 2.0, то в примерах не потребуется изменять код, если в библиотеках XPL-RPC будет введена поддержка SAX 2.0. Однако если используется иной анализатор, то может понадобиться указать класс, реализующий SAX 2.0, если в библиотеки XML-RPC будут внесены изменения, связанные с поддержкой SAX 2.0.

делается с помощью метода setDriver() — статического метода класса XmlRpc, являющегося основой класса WebServer, но в целях изменения драйвера SAX он должен быть импортирован и использован явным образом. Указанный метод генерирует исключение ClassNotFoundException, и оно должно быть перехвачено в том случае, если при выполнении программы требуемый класс не будет найден в путях к классам. Добавьте оператор import и новый код в класс HelloServer:

```
package javaxml2;

import helma.xmlrpc.WebServer;
import helma.xmlrpc.XmlRpc;

public class HelloServer {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println(
                "Использование: java javaxml2.HelloServer [порт]");
            System.exit(-1);
        }

        try {
            // Используем драйвер SAX Apache Xerces
            XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

            // Запускаем сервер

        } catch (ClassNotFoundException e) {
            System.out.println("Драйвер SAX не найден");
        }
    }
}
```

Теперь можно добавить основную часть кода, в которой происходит создание HTTP-приемника (listener), который обслуживает запросы XML-RPC, а также регистрация отдельных классов-обработчиков, доступных для удаленного вызова процедур. Создать приемник очень просто: экземпляр вспомогательного класса WebServer, о котором я говорил, можно создать, передав конструктору требуемый номер порта. И это практически все: наш сервер уже сможет обрабатывать запросы XML-RPC. Хотя еще нет классов, методы которых доступны для вызова, уже есть работающий сервер XML-RPC. Давайте добавим в код примера операторы для создания и запуска сервера, а также для вывода строки состояния. Также необходимо добавить еще один оператор import и обработку исключения для класса java.io.IOException. Поскольку сервер должен работать через определенный порт, он может сгенерировать исключение IOException, если порт недоступен или имелись другие проблемы при запуске сервера. Измененный фрагмент кода выглядит так:

```
package javaxml2;

import java.io.IOException;
```

```
import helma.xmlrpc.WebServer;
import helma.xmlrpc.XmlRpc;

public class HelloServer {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println(
                "Использование: java javaxml2.HelloServer [порт]");
            System.exit(-1);
        }

        try {
            // Используем драйвер SAX  Apache Xerces
            XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

            // Запускаем сервер
            System.out.println("Стартует сервер XML-RPC...");
         WebServer server = new WebServer(Integer.parseInt(args[0]));

        } catch (ClassNotFoundException e) {
            System.out.println("Драйвер SAX не найден");
        } catch (IOException e) {
            System.out.println("Невозможно запустить сервер: " +
                e.getMessage());
        }
    }
}
```

Скомпилируйте этот класс и попытайтесь запустить его. Он полностью работоспособен и должен выдать строку состояния, после чего перейти в режим ожидания запросов. Теперь надо добавить к серверу класс-обработчик, который будет получать запросы.

Одно из наиболее значительных различий между RMI и RPC заключается в способе организации доступа к методам. В RMI интерфейс состоит из набора сигнатур для методов, которые могут вызываться удаленно. Если метод реализован в классе сервера, но соответствующая сигнатура не добавлена в интерфейс, этот новый метод не может быть вызван клиентом RMI. Это приводит к значительным изменениям кода и его перекомпиляции при разработке классов RMI. В RPC этот процесс существенно отличается и, в общем случае, считается проще и гибче. Когда на сервер RPC поступает запрос, этот запрос содержит набор параметров и текстовое значение, обычно в формате «имя_класса.имя_метода». Так сервер RPC узнает, в каком классе содержится искомый метод и как он называется. Далее сервер RPC пытается найти соответствующий класс и метод, объявление которого соответствует типам параметров, переданных в запросе RPC. Как только соответствующий метод найден, он вызывается, а результат кодируется и возвращается клиенту.

Таким образом, искомый метод никогда явно не определен на сервере XML-RPC, а идентифицируется на основе запроса, поступившего от

клиента. На сервере XML-RPC регистрируется лишь экземпляр класса. Можно добавить методы в этот класс, перезапустить сервер XML-RPC без какого-либо изменения кода сервера (позволяя ему зарегистрировать обновленный экземпляр класса), а затем немедленно вызвать эти новые методы в коде клиента. Новые методы становятся доступными непосредственно после создания, если у клиента есть возможность определить и послать корректные параметры серверу. В этом состоит одно из преимуществ XML-RPC перед RMI – технология XML-RPC может быть более реалистично представлена в форме API. Не существует никаких заглушек, шаблонов или интерфейсов, которые необходимо обновлять. Если метод добавлен, его сигнатуру можно опубликовать для клиентов и использовать немедленно.

Итак, вы узнали о простом способе использования обработчика RPC, а теперь зарегистрируем один такой обработчик в примере `HelloHandler`. Класс `WebServer` позволяет добавить обработчик с помощью метода `addHandler()`. Исходные данные для метода представлены именем, под которым регистрируется класс-обработчик и собственно экземпляр этого класса. Обычно новый экземпляр класса создается посредством вызова конструктора (используется ключевое слово `new`), хотя в следующем разделе рассмотрены другие методы – на тот случай, когда экземпляр класса должен быть разделяемым, а не создаваться заново для каждого клиента. В текущем примере создание нового экземпляра класса является приемлемым решением. Зарегистрируйте класс `HelloHandler` под именем «hello.» Также можно добавить несколько дополнительных строк, сообщающих о состоянии сервера, чтобы получить возможность следить за тем, что происходит на сервере при добавлении обработчика:

```
try {
    // Используем драйвер SAX Apache Xerces
    XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

    // Запускаем сервер
    System.out.println("Стартует сервер XML-RPC...");
 WebServer server = new WebServer(Integer.parseInt(args[0]));

    // Регистрируем класс-обработчик
    server.addHandler("hello", new HelloHandler());
    System.out.println(
   "Класс HelloHandler зарегистрирован под именем \"hello\"");

System.out.println("Сервер принимает запросы...");

} catch (ClassNotFoundException e) {
    System.out.println("Драйвер SAX не найден");
} catch (IOException e) {
    System.out.println("Невозможно запустить сервер: " +
        e.getMessage());
}
```

Теперь можно перекомпилировать этот исходный файл и запустить сервер. Конечный результат должен выглядеть примерно так, как показано в примере 11.3.[1]

Пример 11.3. Запуск сервера

```
$ java javaxml2.HelloServer 8585
Стартует сервер XML-RPC...
Класс HelloHandler зарегистрирован под именем "hello"
Сервер принимает запросы...
```

Все очень просто! Теперь осталось создать клиент для сервера и проверить взаимодействие по сети с помощью XML-RPC. Вот и еще одно преимущество XML-RPC: барьер, который требуется преодолеть, чтобы научиться создавать серверы и клиенты, совсем невысок, по сравнению со сложностями применения RMI. Читайте дальше и вы увидите, что создание клиента является столь же простым.

Создание клиента

Создав и запустив сервер, принимающий запросы клиентов, мы выполнили самую трудную часть написания приложения XML-RPC (хотите верьте, хотите – нет, но это была трудная часть!). Теперь дело за созданием простого клиента, который выполнит удаленный вызов метода `sayHello()`. Это легко делается с помощью класса `helma.xmlrpc.XmlRpcClient`. Этот класс отвечает за многочисленные детали на стороне клиента, за которые на сервере отвечают его аналоги `XmlRpcServer` и `WebServer`. Создание клиента потребует также использования класса `XmlRpc`. Клиент должен позаботиться о кодировании запроса, поэтому снова необходимо установить используемый класс драйвера SAX с помощью метода `setDriver()`. Начните разработку клиента с необходимых операторов `import`, проверки аргументов, передаваемых в качестве параметров методу `sayHello()` на сервере, и обработки некоторых исключительных ситуаций. Создайте исходный файл Java, показанный в примере 11.4, и сохраните его под именем *HelloClient.java*.

Пример 11.4. Клиент для сервера XML-RPC

```
package javaxml2;

import helma.xmlrpc.XmlRpc;
import helma.xmlrpc.XmlRpcClient;

public class HelloClient {
```

[1] Тем, кто работает на машине под управлением Unix, для запуска сервера на порту с номером менее 1024 необходимо войти в систему под именем root. Чтобы избежать подобных проблем, рассмотрите возможность использования порта с большим номером, как показано в примере 11.3.

```
    public static void main(String args[]) {
        if (args.length < 1) {
            System.out.println(
                "Использование: java HelloClient [ваше_имя]");
            System.exit(-1);
        }

        try {
            // Используем драйвер SAX Apache Xerces
            XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

            // Задаем сервер

            // Создаем запрос

            // Выполняем запрос и выводим результат

        } catch (ClassNotFoundException e) {
            System.out.println("Драйвер SAX не найден");
        }
    }
}
```

Как и другие фрагменты кода в этой главе, этот должен быть для вас простым и понятным. Создание клиента XML-RPC связано с созданием экземпляра класса `XmlRpcClient`, которому требуется указать имя машины сервера XML-RPC, к которому происходит подключение. Имя машины должно быть представлено полным URL, включая и префикс протокола *http://*. При создании клиента может возникнуть исключение `java.net.MalformedURLException`, если данный URL имеет недопустимый формат. Можно добавить этот класс в список импортируемых классов, создать экземпляр клиента и необходимый обработчик исключения:

```
package javaxml2;

import java.net.MalformedURLException;
import helma.xmlrpc.XmlRpc;
import helma.xmlrpc.XmlRpcClient;

public class HelloClient {

    public static void main(String args[]) {
        if (args.length < 1) {
            System.out.println(
                "Использование: java HelloClient [ваше_имя]");
            System.exit(-1);
        }

        try {
            // Используем драйвер SAX Apache Xerces
            XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

            // Задаем сервер
            XmlRpcClient client =
                new XmlRpcClient("http://localhost:8585/");
```

```
            // Создаем запрос

            // Выполняем запрос и выводим результат

        } catch (ClassNotFoundException e) {
            System.out.println("Драйвер SAX не найден");
        } catch (MalformedURLException e) {
            System.out.println(
                "Неверно задан формат URL для сервера XML-RPC: " +
                e.getMessage());
        }
    }
}
```

Хотя нет непосредственных вызовов RPC, клиентское приложение полностью функционально. Его можно скомпилировать и запустить, правда, вы не увидите никакой активности, поскольку соединение с сервером не устанавливается до тех пор, пока не инициирован запрос.

Примечание

Убедитесь, что в исходном коде используется тот номер порта, который вы планируете указать при запуске сервера. Очевидно, что это не лучший способ реализовать связь между клиентом и сервером – изменение номера порта, на котором ожидает соединения сервер, требует изменения исходного кода клиента! В своих собственных приложениях храните это значение в переменной, определяемой пользователем; я же намеренно упрощаю код.

Легкость в установлении отношений клиента и сервера впечатляет. Тем не менее, данная программа не особо полезна, если не посылает запросы и не получает результаты. Чтобы закодировать запрос, нужно вызвать метод `execute()` класса `XmlRpcClient`. Данный метод имеет два параметра: имя идентификатора класса и вызываемого метода, которые представляются в виде одного строкового параметра, а также объект `Vector`, содержащий параметры, передаваемые указанному методу. Идентификатор класса – это имя, под которым класс `HelloHandler` зарегистрирован на сервере XML-RPC. Хотя этим идентификатором может быть настоящее имя класса, обычно это нечто более удобочитаемое и имеющее смысл для клиента, в данном случае это «hello». К нему добавляется имя вызываемого метода, отделенное от идентификатора класса точкой, в следующей форме: *[идентификатор_класса].[имя_метода]*. Параметры должны передаваться в форме Java-объекта `Vector` и содержать все параметры, требуемые указанным методом. В случае простого метода `sayHello()` параметром является строка с именем пользователя, которое должно быть указано в командной строке при запуске клиента.

Запрос, закодированный клиентом XML-RPC, посылается серверу XML-RPC. Сервер ищет класс, соответствующий идентификатору класса в запросе, и метод этого класса с указанным именем. Если соответствующее имя метода найдено, типы параметров метода сравнива-

ются с параметрами, полученными в запросе. Если типы совпадают, метод выполняется. В случае когда методов с указанным именем несколько, именно параметры определяют, какой именно метод вызывается. Такой алгоритм делает возможным применение в классах-обработчиках нормальной перегрузки методов Java. Результат вызова метода кодируется сервером XML-RPC и отправляется клиенту в виде Java-объекта `Object` (который, в свою очередь, может быть вектором, содержащим объекты!). Далее этот результат можно преобразовать к соответствующему типу Java и использовать в клиенте обычным образом. Если не найдена хотя бы одна сигнатура идентификатор/метод/параметры, соответствующая параметрам запроса, клиенту возвращается исключение `XmlRpcException`. Это гарантирует, что клиент не будет пытаться вызвать метод или обработчик, который не существует, и не будет посылать некорректные параметры.

Все это достигается путем добавления нескольких строк кода Java: необходимо импортировать классы `XmlRpcException` и `java.io.IOException`. Последнее из этих двух исключений генерируется, когда при связи сервера и клиента возникают ошибки. Далее можно импортировать класс `Vector` и создать экземпляр этого класса, поместив в него единственный строковый параметр. Это позволяет вызвать метод `execute()` с именем обработчика, именем вызываемого метода и его параметрами. Результат этого вызова преобразуется к типу `String`, после чего эта строка выводится на экран. В данном примере сервер XML-RPC принимает запросы через порт 8585:

```
package javaxml2;

import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Vector;

import helma.xmlrpc.XmlRpc;
import helma.xmlrpc.XmlRpcClient;
import helma.xmlrpc.XmlRpcException;

public class HelloClient {

    public static void main(String args[]) {
        if (args.length < 1) {
            System.out.println(
                "Использование: java HelloClient [ваше_имя]");
            System.exit(-1);
        }

        try {
            // Используем драйвер SAX Apache Xerces
            XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

            // Задаем сервер
            XmlRpcClient client =
                new XmlRpcClient("http://localhost:8585/");
```

```
            // Создаем запрос
            Vector params = new Vector();
            params.addElement(args[0]);

            // Выполняем запрос и выводим результат
            String result =
               (String)client.execute("hello.sayHello", params);

         System.out.println("Ответ сервера: " + result);

      } catch (ClassNotFoundException e) {
            System.out.println("Драйвер SAX не найден");
      } catch (MalformedURLException e) {
            System.out.println(
                "Неверно задан формат URL для сервера XML-RPC: " +
                e.getMessage());
      } catch (XmlRpcException e) {
            System.out.println("Исключение XML-RPC: " + e.getMessage());
      } catch (IOException e) {
            System.out.println("Исключение IO: " + e.getMessage());
      }
   }
}
```

Вот и все, что требуется, чтобы система заработала. Теперь можно скомпилировать код и открыть командный интерпретатор для запуска этого примера.

Проверка работы

Убедитесь, что пути к классам XML-RPC и классам примера указаны в переменной окружения `CLASSPATH`. Также проверьте, доступен ли Apache Xerces или иной выбранный вами драйвер SAX, путь к нему указан в переменной `CLASSPATH`, поскольку примеры должны загружать эти классы для выполнения синтаксического анализа. А теперь запустите класс `HelloServer`, передав ему номер порта. Для запуска сервера в качестве самостоятельного процесса в Windows выполните команду *start*:

```
c:\javaxml2\build>start java javaxml2.HelloServer 8585
Стартует сервер XML-RPC...
Класс HelloHandler зарегистрирован под именем "hello"
Сервер принимает запросы...
```

В Unix используйте команду выполнения в фоновом режиме (`&`), чтобы клиент мог быть запущен в том же сеансе, либо откройте новое окно терминала и скопируйте настройки переменных окружения:

```
$ java javaxml2.HelloServer &
Стартует сервер XML-RPC...
Класс HelloHandler зарегистрирован под именем "hello"
Сервер принимает запросы...
```

Теперь можно запустить клиентское приложение, указав свое имя в качестве аргумента в командной строке. Вы должны быстро получить ответ (похожий на тот, который приведен в примере 11.5), когда класс `HelloServer` получит запрос, обработает его и вернет результат выполнения метода `sayHello()`, который затем будет отображен клиентским приложением.

Пример 11.5. Запуск класса HelloClient

```
$ java javaxml2.HelloClient Ли
Ответ сервера: Привет, Ли
```

Вы увидели технологию XML-RPC в действии. Конечно, это не слишком полезный пример, но он дает представление об основах и демонстрирует простоту разработки клиента и сервера XML-RPC на Java. С этими базовыми знаниями можно переходить к более реалистичному примеру. В следующем разделе описано создание более полезного сервера, там же вы увидите, как обычно выглядят обработчики XML-RPC. Затем мы напишем клиентское приложение (похожее на `HelloClient`) для тестирования нового кода.

Переложение нагрузки на сервер

Как бы поучителен ни был пример «hello», демонстрирующий применение XML-RPC в Java, он не очень-то реалистичен. Помимо того что этот пример тривиален, сервер не слишком гибок, и сам по себе обработчик не дает четкого представления о том, как мог бы работать реальный обработчик XML-RPC. Здесь я попытаюсь представить примеры применения XML-RPC в промышленных системах посредством совершенствования обработчика и повышения практичности сервера. И хотя приводимый код еще не столь совершенен, чтобы его стоило использовать в реальных проектах, он иллюстрирует возможные применения XML-RPC и методы создания приложений, использующих XML-RPC в числе прочих технологий.

Разделяемый обработчик

Класс `HelloHandler` устроен просто, но он бесполезен в практическом приложении. Большая часть применений XML-RPC связана с тем, чтобы дать возможность выполняться на сервере всем тем процессам, которые более соответствуют сложным задачам, одновременно предоставляя «тонкому» клиенту возможность запрашивать выполнение процедур и использовать возвращаемые результаты. К тому же, возникает возможность выполнения части или даже всех вычислений, требуемых для выдачи ответа на запрос, с опережением. Другими словами, класс обработчика может запускать задачи и гарантировать, что их результаты уже будут доступны к тому моменту, когда поступит

вызов метода. Разработчики на Java сразу же должны вспомнить о потоках и разделяемых экземплярах классов. Чтобы продемонстрировать принципы, о которых идет речь, рассмотрим пример простейшего планировщика – класс `Scheduler`.

Планировщик должен позволять клиентам добавлять и удалять события. Также клиенты должны иметь возможность запрашивать у планировщика список всех запланированных событий. Чтобы сделать его более полезным (и чтобы дать серверу задачу для выполнения в будущем), возвращаемый список запланированных текущих событий упорядочен по времени выполнения. Событие в данном примере представлено своим именем (`String`) и временем выполнения (в формате `java.util.Date`). Хотя такая реализация планировщика несколько ущербна, она позволяет продемонстрировать «закулисное» выполнение сервером некоторой работы для клиентов.

Первым делом создадим методы `addEvent()` и `removeEvent()`. Поскольку вызовы обоих методов инициируются клиентом, в них нет ничего особенного. О чем есть смысл задуматься, так это о способе хранения событий в классе `Scheduler`. Хотя сервер XML-RPC будет создавать экземпляр этого класса, и этот экземпляр будет использоваться для всех вызовов XML-RPC, поступающих на сервер, возможно, и даже весьма вероятно, что и другие классы или даже серверы XML-RPC тоже будут взаимодействовать с планировщиком. Если хранить список событий в поле класса, множественные экземпляры этого класса не смогут совместно использовать эти данные. Решением проблемы является статическое хранилище, которое подлежит совместному использованию всеми экземплярами класса `Scheduler`. Для хранения имени и времени события вполне подходит класс `Hashtable`, предоставляющий возможность использования пар «ключ-значение». Помимо этого, имена событий дополнительно хранятся в объекте типа `Vector`. Это требует дополнительных затрат памяти виртуальной машины Java, но дает возможность сортировать `Vector` и не связываться с сортировкой хеш-таблицы. При сортировке элементов объекта `Vector` на перестановку пары имен тратится одно действие, тогда как сортировка хеш-таблицы требует двух действий на каждую операцию. Располагая этой информацией, можно написать каркас класса и добавить упомянутые два метода, дающие возможность добавления и удаления событий. Кроме того, мы добавим и хранилище событий, но реализацию чтения и сортировки событий оставим на будущее. В примере 11.6 представлен код нового обработчика.

Пример 11.6. Класс Scheduler

```
package javaxml2;

import java.util.Date;
import java.util.Hashtable;
import java.util.Vector;
```

```
public class Scheduler {

    /** Список имен событий (для сортировки) */
    private static Vector events = new Vector();

    /** Информация о событиях (имя, время) */
    private static Hashtable eventDetails = new Hashtable();

    public Scheduler() {
    }

    public boolean addEvent(String eventName, Date eventTime) {
        // Добавляем событие к списку событий
        if (!events.contains(eventName)) {
            events.addElement(eventName);
            eventDetails.put(eventName, eventTime);
        }

        return true;
    }

    public synchronized boolean removeEvent(String eventName) {
        events.remove(eventName);
        eventDetails.remove(eventName);

        return true;
    }
}
```

Метод `addEvent()` добавляет имя события в оба объекта-контейнера, а в хеш-таблицу также и время. Метод `removeEvent()` производит удаление события. Оба метода возвращают логическое значение. Хотя в данном примере это значение всегда истинно, в более сложных программах это значение может быть использовано для уведомления о проблемах, происходящих при добавлении или удалении событий.

Обеспечив возможность добавлять и удалять события, необходимо добавить метод, возвращающий список событий. Этот метод должен возвращать все события, добавленные в хранилище событий, независимо от того, какой клиент или какое приложение эти события добавили. Иначе говоря, это могут быть события, добавленные другим клиентом XML-RPC, другим сервером XML-RPC, иным приложением или же самим планировщиком. Поскольку метод должен вернуть в качестве результата единственный объект типа `Object`, он может вернуть объект типа `Vector`, содержащий отформатированные строковые значения с именем и временем для каждого события. Разумеется, в более практичной реализации этот метод мог бы вернуть вектор событий или события в каком-либо ином типизированном формате (с датой, представленной объектом типа `Date` и т. п.). Наш метод полезен скорее для просмотра данных и не позволяет клиенту выполнять дальнейшие действия над ними. Для создания списка событий применяется хранилище событий и класс `java.text.SimpleDateFormat`, который обеспечивает возможность представления объектов `Date` в текстовом формате. Происхо-

дит перебор событий в цикле, для каждого создается строка с именем и установленной датой. Каждая строка добавляется в конечный список Vector, который затем возвращается клиенту. Добавьте соответствующие операторы import и код для получения списка зарегистрированных событий в класс Scheduler:

```
package javaxml2;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Hashtable;
import java.util.Vector;

public class Scheduler {

    // Реализация существующих методов

    public Vector getListOfEvents() {
        Vector list = new Vector();

        // Создание объекта для форматирования даты
        SimpleDateFormat fmt =
            new SimpleDateFormat("hh:mm a MM/dd/yyyy");

        // Добавление событий в список
        for (int i=0; i<events.size(); i++) {
            String eventName = (String)events.elementAt(i);
            list.addElement("Событие \"" + eventName +
                            "\" запланировано на " +
                             fmt.format(
                                 (Date)eventDetails.get(eventName)));
        }
        return list;
    }
}
```

Теперь можно использовать этот класс в качестве обработчика XML-RPC без особых проблем. Однако смысл этого упражнения заключается в том, чтобы продемонстрировать, как работа выполняется сервером, пока клиент занимается другими делами. Метод getListOfEvents() предполагает, что список событий (в объекте Vector) упорядочен должным образом, когда происходит вызов метода, т. е. сортировка уже выполнена. Кода для сортировки событий у нас еще нет, и, что еще важнее, не написан код, приводящий к выполнению этой сортировки. Более того, с ростом объема хранилища событий растет и время, уходящее на сортировку, а клиента нельзя заставлять ждать ее завершения. Первым делом стоит добавить метод, посредством которого класс мог бы сортировать события. Для простоты здесь используется пузырьковая сортировка (обсуждение алгоритмов сортировки выходит за пределы этой книги, поэтому данный код приводится без комментариев). В конце данного метода поле events сортируется по времени события. За информацией по данному и другим алгоритмам сортировки

обратитесь к книге Роберта Сэджвика (Robert Sedgewick) и Тима Линдхольма (Tim Lindholm) «Algorithms in Java» (Алгоритмы на Java), Addison-Wesley. Алгоритм и метод, управляющий сортировкой событий, представлены далее, и их следует добавить в код программы:

```
package javaxml2;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;

public class Scheduler {

    /** Список имен событий (для сортировки) */
    private static Vector events = new Vector();

    /** Информация о событиях (имя, время) */
    private static Hashtable eventDetails = new Hashtable();

    /** Флаг, указывающий на то, отсортированы ли события */
    private static boolean eventsSorted;

    // Реализация существующих методов

    private synchronized void sortEvents() {
        if (eventsSorted) {
            return;
        }

        // Создаем массив событий (не отсортированный)

        String[] eventNames = new String[events.size()];
        events.copyInto(eventNames);

        // Пузырьковая сортировка
        String tmpName;
        Date date1, date2;
        for (int i=0; i<eventNames.length - 1; i++) {
            for (int j=0; j<eventNames.length - i - 1; j++) {

                // Сравниваем даты для этих событий
                date1 = (Date)eventDetails.get(eventNames[j]);
                date2 = (Date)eventDetails.get(eventNames[j+1]);
                if (date1.compareTo(date2) > 0) {

                    // При необходимости переставляем местами
                    tmpName = eventNames[j];
                    eventNames[j] = eventNames[j+1];
                    eventNames[j+1] = tmpName;
            }
        }

        // Помещаем в новый Vector (отсортированный)
        Vector sortedEvents = new Vector();
        for (int i=0; i<eventNames.length; i++) {
```

```
                sortedEvents.addElement(eventNames[i]);
            }
            // Обновляем основное хранилище событий
            events = sortedEvents;
            eventsSorted = true;
        }
    }
```

В дополнение к реализации основного алгоритма мы импортируем класс `java.util.Enumeration` и добавляем логическое поле `eventsSorted`. Этот флаг позволяет пропускать выполнение сортировки, если события уже упорядочены. Код для обновления флага еще не существует, но его легко создать. Метод, выполняющий сортировку, по завершении работы указывает, что события упорядочены. Значение этого флага в конструкторе класса следует изначально установить в `true`, указав таким образом, что все события упорядочены. Список может стать неупорядоченным только тогда, когда добавляются события, поэтому нужно устанавливать этот флаг в значение `false` в методе `addEvents()`, если событие добавляется. Таким образом можно проинформировать класс `Scheduler` о событии в коде, которое должно привести к запуску процесса сортировки. Когда вызывается метод `getListOfEvents()`, события будут упорядочены и готовы к форматированию. Итак, допишем в конструктор и в метод добавления событий код, отвечающий за обновление флага сортировки:

```
package javaxml2;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;

public class Scheduler {

    public Scheduler() {
        eventsSorted = true;
    }

    public boolean addEvent(String eventName, Date eventTime) {
        // Добавляем это событие к списку событий
        if (!events.contains(eventName)) {
            events.addElement(eventName);
            eventDetails.put(eventName, eventTime);
            eventsSorted = false;
        }

        return true;
    }

  // Реализация других методов
}
```

В метод removeEvent() вносить изменения не нужно, поскольку удаление одного события не изменяет их порядка. Идеальный механизм выполнения операций на сервере, при котором клиент может продолжать выполнение дальнейших действий, – это поток, сортирующий события. Когда в виртуальной машине Java запускается такой поток, клиент может продолжать работу и не ждать завершения выполнения потока. Это особенно важно в многопоточной среде, когда применяется синхронизация и блокировка объектов. В данном примере я уклонился от рассмотрения вопросов, связанных с потоками, но достаточно нетрудно создать код, реализующий необходимую функциональность. Понадобится создать внутренний класс, расширяющий класс Thread, единственная задача которого состоит в вызове метода sortEvents(). Затем можно добавить в метод addEvents() код, отвечающий за создание и запуск этого потока при добавлении события. После этого добавление события будет инициировать пересортировку событий, что позволит клиенту продолжать независимую работу (которая может быть связана и с добавлением новых событий, и, как следствие, с запуском дополнительных потоков сортировки данных). Когда клиент запрашивает список событий, при выдаче они уже должны быть упорядочены, при этом клиент не должен ожидать выполнения этих действий и на выполнение сортировки не должны тратиться ресурсы. Добавление внутреннего класса для сортировки и кода для запуска этого класса в отдельном потоке в методе addEvents() делает класс Scheduler завершенным:

```
package javaxml2;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;

public class Scheduler {

    // Существующие переменные и методы

    public boolean addEvent(String eventName, Date eventTime) {
        // Добавляем данное событие к списку событий
        if (!events.contains(eventName)) {
            events.addElement(eventName);
            eventDetails.put(eventName, eventTime);
            eventsSorted = false;

            // Запускаем поток для сортировки на сервере
            SortEventsThread sorter = new SortEventsThread();
            sorter.start();
        }
        return true;
    }

    class SortEventsThread extends Thread {
```

```
        public void run() {
            sortEvents();
        }
    }
}
```

Компиляция нового исходного кода позволяет получить многопоточный планировщик, выполняющий ресурсоемкую задачу сортировки на сервере, что позволяет любым клиентам работать независимо, пока производится сортировка. Вот еще один простой пример разумного применения класса-обработчика, и он дает представление о распределении ресурсов и делегировании задач серверу. В дополнение к этому примеру усовершенствованного класса-обработчика далее я рассмотрю создание более надежного сервера XML-RPC.

Конфигурируемый сервер

Класс сервера XML-RPC по-прежнему требует некоторой доработки. Текущая версия требует включения классов-обработчиков в код сервера. Другими словами, добавление нового класса-обработчика требует изменения кода и перекомпиляции. Это не только нежелательно с точки зрения контроля за изменениями, но также требует времени и нервов. Извлечение наиболее актуальной версии кода из системы контроля версий, внесение изменений и тестирование ради добавления одного или двух обработчиков непрактично и не добавит вам сторонников среди руководителей. Предпочтительнее иметь надежный сервер, который может читать подобные данные из файла настройки и загружать необходимые классы в процессе выполнения. Для этой цели мы построим облегченный сервер.

Для начала создадим новый класс сервера. Можно начать с нуля или скопировать код класса `HelloServer`, приведенный ранее в этой главе. Как и предыдущем примере, начнем с создания основы, импортирования классов и инициализации сервера. Однако не будем добавлять код регистрации обработчиков, поскольку мы создадим вспомогательный метод для загрузки требуемых данных из файла. Одно из отличий от предыдущей версии заключается в том, что потребуется дополнительный параметр командной строки – имя файла. Сервер будет читать этот файл при помощи методов, которые рассмотрены ниже, и регистрировать обработчики. Можно создать класс `LightweightXmlRpcServer`, который продолжает использовать вспомогательный класс `WebServer`, при помощи кода, приводимого в примере 11.7.

Пример 11.7. Сервер XML-RPC многократного использования

```
package javaxml2;

import java.io.IOException;

import helma.xmlrpc.XmlRpc;
```

```
import helma.xmlrpc.WebServer;

public class LightweightXmlRpcServer {

    /** Вспомогательный класс сервера XML-RPC */
    private WebServer server;

    /** Номер порта для приема соединений */
    private int port;

    /** Файл настройки */
    private String configFile;

    public LightweightXmlRpcServer(int port, String configFile) {
        this.port = port;
        this.configFile = configFile;
    }

    public void start() throws IOException {
        try {
            // Используем анализатор SAX Apache Xerces

            XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

            System.out.println("Стартует сервер XML-RPC...");
            server = new WebServer(port);
            // Регистрируем обработчики

        } catch (ClassNotFoundException e) {
            throw new IOException("Ошибка при загрузке анализатора SAX: " +
                e.getMessage());
        }
    }

    public static void main(String[] args) {

        if (args.length < 2) {
            System.out.println(
                "Использование: " +
                "java com.oreilly.xml.LightweightXmlRpcServer " +
                "[порт] [файл настройки]");
            System.exit(-1);
        }

        LightweightXmlRpcServer server =
            new LightweightXmlRpcServer(Integer.parseInt(args[0]),
                                        args[1]);

        try {
            // Запускаем сервер
            server.start();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Здесь нет ничего примечательного. Мы проверяем получение необходимых параметров и запускаем сервер с учетом указанного порта. Теперь необходимо добавить методы для загрузки информации из файла и регистрации обработчиков на сервере.

Поскольку каждому обработчику требуется имя и связанный с ним класс, можно создать файл настройки, содержащий такие пары. Java позволяет элементарно загрузить и инициализировать класс по имени пакета и класса. Соответственно, обработчик может быть полностью описан парой текстовых значений. В этом файле можно описать первоначальный класс `HelloHandler`, а также новый класс `Scheduler`. Поскольку мы также создаем средство разбора файла настройки, то можем без опаски выбрать запятую в качестве разделителя и символ # в качестве знака комментария. На самом деле формат может быть произвольным, если код разбора соответствует установленному формату.

Примечание

Читатели могут удивиться, что мы не пользуемся тут форматом XML. Для этого есть ряд причин. Во-первых, в следующей главе речь пойдет о SOAP, который от начала до конца привязан к XML. Применение формата, отличного от XML, прекрасно иллюстрирует контраст между этими двумя методологиями. Во-вторых, вы уже вполне подготовлены, чтобы написать собственный код для анализа XML, и эта задача послужит хорошим упражнением. И, в-третьих, я реалист. Попробуйте выяснить, сколько «XML-систем» и «XML-приложений» используют форматы, отличные от XML, и будете удивлены. Так что привыкайте к этому сейчас, поскольку время от времени вы будете с этим сталкиваться.

Создайте файл настройки, приведенный в примере 11.8. С его помощью будет добавлен класс `HelloHandler` с идентификатором «hello» и класс `Scheduler` с идентификатором «scheduler». Сохраните его под именем *xmlrpc.conf*.

Пример 11.8. Файл настройки XML-RPC

```
# Hello Handler: sayHello()
hello,javaxml2.HelloHandler

# Scheduler: addEvent(), removeEvent(), getEvents()
scheduler,javaxml2.Scheduler
```

В целях документирования в комментариях перечислены методы, доступные в каждом из обработчиков. Эта информация будет полезна тем, кому в будущем придется сопровождать код.

Классы Java, отвечающие за ввод и вывод, позволяют легко загрузить этот файл и прочитать его содержимое. Нетрудно создать вспомогательный метод, читающий указанный файл и сохраняющий пары значений в объекте `Hashtable`. Этот объект затем можно передать другому вспомогательному классу, который загружает и регистрирует каждый

из обработчиков. Метод в примере не содержит кода для расширенной диагностики ошибок, которую мог бы выполнять производственный сервер, и просто игнорирует любую строку, в которой нет пары значений, разделенных запятой. В данный код достаточно легко добавить обработку ошибок, если вы хотите использовать его в своих приложениях. Как только строка с парой значений найдена, она разбивается, а пара «идентификатор-имя» класса сохраняется в виде записи хеш-таблицы. Добавьте операторы импортирования для вспомогательных классов и новый метод `getHandlers()` в класс `LightweightServer`:

```
package javaxml2;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Hashtable;
import helma.xmlrpc.XmlRpc;
import helma.xmlrpc.WebServer;

public class LightweightXmlRpcServer {

    // Реализация существующих методов

    private Hashtable getHandlers() throws IOException {

        Hashtable handlers = new Hashtable();

        BufferedReader reader =
            new BufferedReader(new FileReader(configFile));
        String line = null;

        while ((line = reader.readLine()) != null) {
            // Синтаксис "handlerName, handlerClass"
            int comma;

            // Пропускаем комментарии
            if (line.startsWith("#")) {
                continue;
            }

            // Пропускаем пустые или бесполезные строки
            if ((comma = line.indexOf(",")) < 2) {
                continue;
            }

            // Добавляем имя обработчика и класс обработчика
            handlers.put(line.substring(0, comma),
                         line.substring(comma+1));
        }

        return handlers;
    }
}
```

Вместо сохранения результата выполнения этого метода можно использовать его в качестве входных данных для метода, выполняющего перебор элементов хеш-таблицы и регистрацию обработчиков на сервере. Код, решающий эту задачу, довольно прост. Единственное, на что следует обратить внимание: метод addHandler() класса WebServer в качестве параметра должен получить реализованный экземпляр класса. Наш код должен извлечь имя класса, подлежащего регистрации, из хеш-таблицы, загрузить класс в JVM с помощью метода Class.forName(), а затем реализовать этот класс с помощью метода newInstance(). Такая методология применяется в загрузчиках классов и других динамических приложениях на Java, но может выглядеть непривычно для начинающих в Java, как и для тех, кому ранее не приходилось выполнять динамическую реализацию экземпляров классов по их именам. Когда класс загружен таким образом, он наряду с идентификатором класса передается методу addHandler(), и выполнение цикла продолжается. Когда содержимое хеш-таблицы загружено, сервер настроен и готов к работе. Поскольку для выполнения цикла по ключам хеш-таблицы используется класс Enumeration, необходимо добавить в файл соответствующий оператор import:

```
package javaxml2;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Hashtable;

import helma.xmlrpc.XmlRpc;
import helma.xmlrpc.WebServer;

public class LightweightXmlRpcServer {

    // Реализация существующих методов

    private void registerHandlers(Hashtable handlers) {
        Enumeration handlerNames = handlers.keys();

        // Перебор обработчиков в цикле
        while (handlerNames.hasMoreElements()) {
            String handlerName = (String)handlerNames.nextElement();
            String handlerClass = (String)handlers.get(handlerName);

            // Добавляем обработчик к серверу
            try {
                server.addHandler(handlerName,
                    Class.forName(handlerClass).newInstance());

                System.out.println("Обработчик " + handlerName +
                                   " зарегистрирован для класса " +
                                   handlerClass);
            } catch (Exception e) {
```

```
                System.out.println("Невозможно зарегистрировать обработчик " +
                                   handlerName + " для класса " +
                                   handlerClass);
            }
        }
    }
}
```

Этот код является всего лишь дополнением к методу getHandlers() и, по сути, получает результаты его выполнения в качестве входных данных. Строковые значения из хеш-таблицы используются для регистрации обработчиков. Сервер запущен и все обработчики, определенные в файле настройки, загружены, а их методы доступны для удаленных вызовов. Разумеется, мы могли бы объединить эти методы в один, но более объемный. Однако назначение этих двух методов существенно отличается: в то время как один из них, getHandlers(), отвечает за разбор файла, другой, registerHandlers(), отвечает за регистрацию обработчиков, выполняя ее, когда поступает соответствующая информация. Используя данную методологию, можно изменять способ разбора файла настройки (и даже извлекать информацию из базы данных или другого источника данных), не беспокоясь о том, каким образом регистрируются обработчики.

Добавив эти вспомогательные методы, вызовите их из метода start() класса сервера:

```
public void start() throws IOException {
    try {
        // Используем анализатор SAX Apache Xerces
        XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

        System.out.println("Стартует сервер XML-RPC...");
        server = new WebServer(port);

        // Регистрируем обработчики
        registerHandlers(getHandlers());

    } catch (ClassNotFoundException e) {
        throw new IOException("Ошибка при загрузке анализатора SAX: " +
            e.getMessage());
    }
}
```

Скомпилируйте этот код, убедитесь, что файл настройки существует, и сервер будет готов к работе.

Полезный клиент

В новом клиенте нет ни новых концепций, ни новых технологических решений. Класс SchedulerClient столь же прост, как и класс HelloClient. Он должен запускать клиент XML-RPC, вызвать методы обработ-

чиков и печатать результаты этих вызовов. Ниже приводится полный код клиента. Суть происходящего поясняется в комментариях, и поскольку все это уже рассматривалось, можно просто ввести код из примера 11.9 в текстовом редакторе и скомпилировать его.

Пример 11.9. Класс SchedulerClient

```
package javaxml2;

import java.io.IOException;
import java.net.MalformedURLException;
import java.util.Calendar;
import java.util.Date;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;

import helma.xmlrpc.XmlRpc;
import helma.xmlrpc.XmlRpcClient;
import helma.xmlrpc.XmlRpcException;

public class SchedulerClient {

    public static void addEvents(XmlRpcClient client)
        throws XmlRpcException, IOException {

        System.out.println("\Идет добавление событий...\n");

        // Параметры  событий
        Vector params = new Vector();

        // Добавляем событие на следующий  месяц
        params.addElement("Корректура окончательного черновика");

        Calendar cal = Calendar.getInstance();
        cal.add(Calendar.MONTH, 1);
        params.addElement(cal.getTime());

        // Добавляем событие
        if (((Boolean)client.execute("scheduler.addEvent", params))
                            .booleanValue()) {
            System.out.println("Событие добавлено.");
        } else {
            System.out.println("Невозможно добавить событие.");
        }

        // Добавляем событие на завтрашний день
        params.clear();
        params.addElement("Отправка окончательного черновика");

        cal = Calendar.getInstance();
        cal.add(Calendar.DAY_OF_MONTH, 1);
        params.addElement(cal.getTime());
```

```
            // Добавляем событие
            if (((Boolean)client.execute("scheduler.addEvent", params))
                                .booleanValue()) {
                System.out.println("Событие добавлено.");
            } else {
                System.out.println("Невозможно добавить сбытие.");
            }

        }

        public static void listEvents(XmlRpcClient client)
            throws XmlRpcException, IOException {

            System.out.println("\nПеречисление событий...\n");

            // Получаем события из планировщика
            Vector params = new Vector();

            Vector events =
                (Vector)client.execute("scheduler.getListOfEvents", params);
            for (int i=0; i<events.size(); i++) {
                System.out.println((String)events.elementAt(i));
            }
        }

        public static void main(String args[]) {

            try {
                // Используем реализацию SAX-анализатора Apache Xerces
                XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

                // Соединяемся с сервером
                XmlRpcClient client =
              new XmlRpcClient("http://localhost:8585/");

                // Добавляем события
                addEvents(client);

                // Получаем список событий
                listEvents(client);

            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
```

При наборе кода обратите внимание, что события добавляются в обратном порядке, если говорить о времени. Сервер должен отсортировать эти события с помощью метода `sortEvents()` и вернуть упорядоченные результаты при вызове метода `getListOfEvents()`. Далее мы увидим, как сервер выполняет сортировку.

Проверка работы (еще одна)

Написав код для обработчика, сервера и клиента, скомпилируйте все исходные файлы. Следует также создать файл настройки, в котором перечисляются регистрируемые сервером XML-RPC обработчики. Мы говорили о них в разделе «Конфигурируемый сервер». Первым делом запустите сервер XML-RPC в качестве самостоятельного процесса:

```
c:\javaxml2\build>start java javaxml2.LightweightXmlRpcServer 8585
                        c:\javaxml2\ch11\conf\xmlrpc.conf
```

Или в Unix:

```
$ java javaxml2.LightweightServer 8585 conf/xmlrpc.conf &
```

Вы должны увидеть сообщения сервера о том, что обработчики из указанного файла настройки зарегистрированы с указанными именами:

```
Стартует сервер XML-RPC...
Обработчик scheduler зарегистрирован для класса javaxml2.Scheduler
Обработчик hello зарегистрирован для класса javaxml2.HelloHandler
```

Примечание

Если вы не останавливали предыдущий сервер XML-RPC – `HelloServer`, возникнет ошибка, связанная с попыткой запустить еще один сервер на том же самом порту. Перед тем как запускать сервер `LightweightXmlRpcServer`, обязательно остановите `HelloServer`.

Наконец, запустите клиентское приложение и оцените результаты:

```
$ java javaxml2.SchedulerClient

Идет добавление событий...

Событие добавлено.
Событие добавлено.

Перечисление событий...

Событие "Отправка окончательного черновика" запланировано
          на 10:55 AM 05/09/2001

Событие "Корректура окончательного черновика" запланировано
          на 10:55 AM 06/08/2001
```

Вы не должны заметить значительной паузы, пока клиент добавляет события и выдает их список, хотя сервер при этом сортирует события в отдельном потоке виртуальной машины Java (а пузырьковая сортировка отнюдь не быстрый алгоритм!). Итак, ваше первое полезное приложение XML-RPC написано!

Реальные ситуации

Завершается эта глава кратким обзором некоторых важных аспектов применения XML-RPC в реальных приложениях. Мы сконцентрируем внимание на том, чтобы дать вам возможность использовать XML не потому, что это самая новая и самая элегантная технология, а потому, что она является наилучшим решением в целом ряде случаев. Все знания, содержащиеся в этой книге, все спецификации XML и другие книги об XML не заставят ваше приложение работать лучше, чем оно могло бы, если вы не знаете, когда и как правильно применять XML-RPC! Этот раздел освещает некоторые общие вопросы, возникающие при работе с XML-RPC.

Где же XML в XML–RPC?

Изучив эту главу, читатели, возможно, были удивлены, что нам не пришлось написать ни одной строки кода с использованием SAX, DOM или JDOM. По сути, мы практически не использовали XML напрямую. И все потому, что библиотеки XML-RPC отвечают за кодирование и декодирование информации, которой обмениваются клиенты и серверы. Отсутствие кода, работающего непосредственно с XML, может несколько разочаровать, но, тем не менее, вы определенно применяете технологию XML. Простой вызов метода `sayHello()` в действительности преобразуется в вызов HTTP, подобный показанному в примере 11.10.

Пример 11.10. XML для запроса XML-RPC

```
POST /RPC2 HTTP/1.1
User-Agent: Tomcat Web Server/3.1 Beta (Sun Solaris 2.6)
Host: newInstance.com
Content-Type: text/xml
Content-length: 234

<?xml version="1.0"?>
<methodCall>
  <methodName>hello.sayHello</methodName>
  <params>
    <param>
      <value><string>Бретт</string></value>
    </param>
  </params>
</methodCall>
```

Библиотеки XML-RPC на сервере получают этот запрос и декодируют его, подбирая к нему обработчик (если таковой имеется). Затем вызывается указанный метод Java, а сервер кодирует результат обратно в XML, как показано в примере 11.11.

Пример 11.11. Ответ XML-RPC в формате XML

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/xml
Content-Length: 149
Date: Mon, 11 Apr 2000 03:32:19 CST
Server: Tomcat Web Server/3.1 Beta-Sun Solaris 2.6

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>Привет, Бретт</string></value>
    </param>
  </params>
</methodResponse>
```

Все это взаимодействие происходит таким образом, что вам не приходится беспокоиться о деталях.

Разделяемые экземпляры

В примерах рассматривалось применение статических объектов-контейнеров, позволяющих реализовать совместное использование данных несколькими экземплярами одного и того же класса. Однако бывают случаи, когда сам экземпляр класса является разделяемым. Это может случиться не потому, что этого требует технология XML-RPC, а по причине необходимости различными способами использовать этот класс на сервере. Так, одиночный образец проектирования (singleton design pattern) предписывает создание только одного экземпляра класса, который совместно используется всеми приложениями. Обычно это достигается с помощью статического метода `getInstance()` вместо вызова конструктора для объекта:

```
Scheduler scheduler;

// Получаем единственный экземпляр класса Scheduler
scheduler = Scheduler.getInstance();

// Добавляем событие для текущего момента
scheduler.addEvent("Пикник", new Date());
```

Чтобы запретить явную реализацию класса `Scheduler`, конструктор обычно делают закрытым (private) или защищенным (protected). Это вынуждает клиентов применять приведенный выше код для получения экземпляра класса, а кроме того, может служить причиной недоразумений при попытке применения такого класса в качестве обработчика XML-RPC. Помните, что регистрация обработчика всегда осуществляется путем создания экземпляра класса-обработчика. Однако класс `WebServer` требует применения корректного экземпляра класса в качестве параметра, но не обязательно нового экземпляра. Например,

следующий код демонстрирует абсолютно допустимый способ регистрации обработчика:

```
WebServer server = new WebServer(8585);

// Создаем экземпляр класса обработчика
HelloHandler hello = new HelloHandler();
server.addHandler("hello", hello);
```

Класс сервера не различает методологии, если экземпляр класса-обработчика создается до передачи методу addHandler(). Поэтому можно внести небольшое изменение в этот код, если мы хотим добавить одиночный класс Scheduler, описанный ранее:

```
WebServer server = new WebServer(8585);

// Передаем одиночный экземпляр
server.addHandler("scheduler", Scheduler.getInstance());
```

Здесь в качестве параметра передается разделяемый экземпляр, и результат точно такой же, как в случае реализации с помощью конструктора и ключевого слова new; при этом сохраняется вся информация, связанная с одиночным классом. Многие классы, используемые в службах, подобных XML-RPC, реализованы как одиночные, чтобы избежать применения статических переменных, поскольку разделяемый экземпляр позволяет хранить данные в полях класса. В таком случае единственный экземпляр класса работает с этими полями при обработке всех клиентских запросов.

Быть сервлету или нет?

Применение сервлетов в качестве серверов XML-RPC в последнее время стало популярным решением. За более подробной информацией о сервлетах обращайтесь к книге Джейсона Хантера «Программирование Java-сервлетов», выпущенной издательством «Символ-Плюс» в 2002 году («Java Servlet Programming» by Jason Hunter, O'Reilly). В состав классов Java для работы с XML-RPC, которые вы загрузили, входит сервлет. Подобное применение сервлета, когда он не делает ничего другого, кроме обработки запросов XML-RPC, является как допустимым, так и достаточно распространенным. Тем не менее, это не всегда лучший вариант.

Если у вас есть машина, которая должна обслуживать другие HTTP-запросы для задач Java, то среда исполнения сервлетов является хорошим решением для обработки деталей, связанных с такими запросами. В таком случае применение сервлета в качестве сервера XML-RPC – неплохая идея. Но одно из преимуществ технологии XML-RPC заключено в том, что она позволяет разделять классы-обработчики, выполняющие сложные, ресурсоемкие задачи, и остальной код приложения. Класс Scheduler можно было бы расположить на сервере вместе

с классами, выполняющими сложное индексирование, алгоритмическое моделирование и, возможно, графические преобразования. Для клиента выполнение всех этих функций является чересчур ресурсоемким. Однако установка среды исполнения сервлетов и прием запросов приложений, касающихся других задач, наряду с обработкой запросов XML-RPC сильно снижает объем ресурсов системы, доступный этим классам-обработчикам. В этом случае единственными запросами, которые должны поступать на сервер, являются запросы к классам-обработчикам.

В том случае, когда принимаются только запросы XML-RPC, применение сервлета в сервере XML-RPC редко оказывается полезным. Готовый класс `WebServer` весьма мал, потребляет очень мало ресурсов и создан специально для обработки запросов XML-RPC через протокол HTTP. Среда исполнения сервлетов разработана для обработки произвольных HTTP-запросов, но не настроена специальным образом на выполнение запросов XML-RPC. Через некоторое время вы начнете замечать пониженную производительность среды исполнения сервлетов по сравнению с классом `WebServer`. Если у вас нет убедительной причины использовать сервлет для других задач, не связанных с технологией XML-RPC, было бы разумно остановить свой выбор на облегченном сервере XML-RPC, созданном специально для решения вашей основной задачи.

Что дальше?

Теперь, когда мы разобрались с RPC и XML-RPC, пришло время предпринять следующий логический шаг. Этот шаг – SOAP – простой протокол доступа к объектам (Simple Object Access Protocol). SOAP построен на XML-RPC, и в нем появилась поддержка пользовательских типов объектов, улучшенная диагностика ошибок и новые возможности. Кроме того, эта технология развивается как раз сейчас. В следующей главе читатели получат исчерпывающую информацию и будут готовы к встрече с SOAP.

- *Начинаем*
- *Установка*
- *«Пачкаемся»*
- *Идем дальше*
- *Что дальше?*

12

SOAP

SOAP – простой протокол доступа к объектам (Simple Object Access Protocol). Сегодня о нем не знают только отшельники. Это новейшая «идея фикс» в веб-программировании – неразлучная спутница фанатизма, связанного с веб-службами и охватившего последнее поколение веб-разработок. Когда вы слышите о .NET от Microsoft или о «революции» peer-to-peer, речь идет о технологиях, основанных на SOAP (даже если вы этого не знаете). Существует целых *две* реализации SOAP в рамках проекта Apache, а Microsoft посвятила этой технологии сотни страниц на своем веб-сайте MSDN (*http://msdn.microsoft.com*).

В этой главе объясняется, что такое SOAP и почему он важен для того будущего, к которому идет система понятий веб-разработки. Это поможет читателям справиться с основами и подготовиться к реальной работе с набором инструментов SOAP. Кроме того, мы вкратце рассмотрим уже существующие проекты SOAP, а затем перейдем к реализации от Apache. Данная глава не содержит полного обзора SOAP; многочисленные пробелы будут восполнены в следующей главе. Сейчас мы посмотрим первую серию двухсерийного фильма, и на многие из вопросов, возникших к концу этой главы, ответы будут даны в следующей.

Начинаем

Первым делом необходимо понять, что собой представляет SOAP. Можно прочитать достаточно длинную и полную спецификацию на сайте *http://www.w3.org/TR/SOAP*. Если вырезать всю рекламную болтовню, то окажется, что SOAP – это просто протокол. Простой (в применении, но не обязательно в реализации) протокол, основанный на идее о том, что в какой-то момент в распределенной среде появляется необходимость обмениваться информацией. Кроме того, в случае применения на загруженной системе этот протокол оказывается нетребо-

вательным к ресурсам. Наконец, он в качестве транспорта использует протокол HTTP, что позволяет справляться со сложными аспектами вроде брандмауэров и избежать создания сокетов-приемников на портах с необычными номерами. Все прочее, что связано с технологией SOAP, – просто детали.

Нет сомнений, что вы читаете книгу из интереса к этим самым деталям, так что я про них не забуду. Спецификация SOAP определяет три основных компонента: конверт SOAP, набор правил кодирования и средства доставки запросов и ответов. Можете воспринимать сообщение SOAP как обычное письмо; ну, вы знаете, такие древние штуковины в конвертах с почтовыми марками и адресами, нацарапанными от руки? Наша аналогия делает гораздо более осмысленными такие понятия SOAP, как «конверт». На рис. 12.1 изображен процесс передачи сообщения SOAP в терминах этой аналогии.

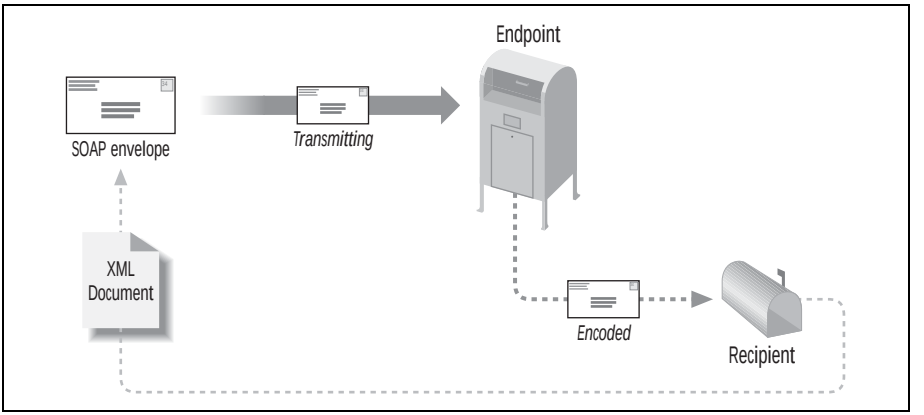

Рис. 12.1. Передача сообщения SOAP

Удерживая в голове эту картинку, рассмотрим три компонента спецификации SOAP. Поговорим коротко о каждом из них, останавливаясь на примерах, более полно раскрывающих все эти концепции. Кроме того, именно эти три ключевых компонента делают SOAP столь важным и ценным. Обработка ошибок, поддержка множества типов кодирования, сериализация произвольных параметров и тот факт, что SOAP работает поверх HTTP, во многих случаях делают его привлекательнее других реализаций распределенного протокола.[1] Кроме того, SOAP обеспечивает высокий уровень интеграции с другими приложениями, о чем более подробный разговор пойдет в следующей главе. Пока же сфокусируемся на основных составляющих SOAP.

[1] Много говорят о работе SOAP поверх других протоколов, таких как SMTP (или даже Jabber). Эти функции не являются частью стандарта SOAP, но могут быть добавлены в будущем. Не удивляйтесь, узнав, что этот вопрос обсуждается.

Конверт

Конверт SOAP аналогичен конверту обычного письма. Он передает информацию о сообщении, закодированном в SOAP, включая данные о получателе и отправителе, а также о самом сообщении. Так, заголовок конверта SOAP может определять конкретный способ обработки сообщения. Прежде чем приложение займется обработкой сообщения, оно имеет возможность получить информацию о нем и даже сделать выводы относительно самой *возможности* обработки. В отличие от ситуации со стандартными вызовами XML-RPC (помните – сообщения XML-RPC, кодирование и все остальное было заключено в один XML-фрагмент), в SOAP сообщение интерпретируется с целью получения информации о нем. Обычное сообщение SOAP также может содержать стиль кодирования, помогающий получателю интерпретировать сообщение. В примере 12.1 приведен конверт SOAP, содержащий информацию о типе кодирования.

Пример 12.1. Конверт SOAP

```
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   soap:encodingStyle="http://myHost.com/encodings/secureEncoding"
>
  <soap:Body>
    <article xmlns="http://www.ibm.com/developer">
      <name>Soapbox</name>
      <url>
        http://www-106.ibm.com/developerworks/library/x-soapbx1.html
      </url>
    </article>
  </soap:Body>
</soap:Envelope>
```

Как видите, тип кодирования задается в конверте, что позволяет приложению определить (на основе значения атрибута `encodingStyle`), способно ли оно прочитать поступившее сообщение, заключенное внутри элемента `Body`. Следует указывать корректное пространство имен конверта SOAP, в противном случае серверы SOAP, получающие сообщение, будут генерировать ошибку несовместимости версий, и вы не сможете «общаться» с ними.

Кодирование

Второй важный элемент, добавленный SOAP, – это простой способ кодирования типов данных, определенных пользователем. В RPC (и XML-RPC) кодирование возможно лишь для предопределенного набора типов данных: тех, которые поддерживаются применяемым инструментарием XML-RPC. Кодирование других типов требует модификации собственно сервера RPC и клиентов. В SOAP новые типы дан-

ных могут создаваться с помощью схем XML (о структуре `complexType` мы говорили в главе 2), и данные этих новых типов могут быть легко представлены в формате XML в информативной части сообщения SOAP. Благодаря такой интеграции с технологией XML Schema в сообщении SOAP можно закодировать любой тип данных, поддающийся описанию с помощью схемы XML.

Вызов

Самый лучший способ понять, как работают вызовы SOAP, – сравнить их с уже знакомым материалом, например с XML-RPC. Вспомним, что вызовы XML-RPC выглядят приблизительно так, как показано во фрагменте кода из примера 12.2.

Пример 12.2. Вызовы в XML-RPC

```
// Задаем используемый анализатор XML
XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

// Задаем сервер, с которым будем соединяться
XmlRpcClient client =
    new XmlRpcClient("http://rpc.middleearth.com");

// Создаем параметры
Vector params = new Vector();
params.addElement(flightNumber);
params.addElement(numSeats);
params.addElement(creditCardType);
params.addElement(creditCardNum);

// Резервирование
Boolean boughtTickets =
    (Boolean)client.execute("ticketCounter.buyTickets", params);

// Обрабатываем ответ
```

Я написал код для простого приложения, подсчитывающего количество билетов. Теперь посмотрите на пример 12.3, в котором показаны эти же вызовы в SOAP.

Пример 12.3. Вызовы в SOAP

```
// Создаем параметры
Vector params = new Vector();
params.addElement(
    new Parameter("flightNumber", Integer.class, flightNumber, null));
params.addElement(
    new Parameter("numSeats", Integer.class, numSeats, null));
params.addElement(
    new Parameter("creditCardType", String.class, creditCardType, null));
params.addElement(
    new Parameter("creditCardNumber", Long.class, creditCardNum, null));
```

```
// Создаем объект Call
Call call = new Call();
call.setTargetObjectURI("urn:xmltoday-airline-tickets");
call.setMethodName("buyTickets");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
call.setParams(params);

// Вызываем
Response res = call.invoke(new URL("http://rpc.middleearth.com"), "");

// Обрабатываем ответ
```

Как видите, собственно вызов, представленный объектом `Call`, постоянно хранится в памяти. Он позволяет устанавливать цель (абонента) вызова, вызываемый метод, стиль кодирования, параметры и многое другое, что здесь не отражено. Эта методика гибче, чем XML-RPC, и позволяет явно устанавливать различные параметры, которые в XML-RPC определялись неявным образом. В оставшейся части главы вы узнаете немного больше о процессе вызова, включая то, как SOAP предоставляет ответы с ошибками, иерархию ошибок и, конечно же, возвращаемые вызовом результаты.

В этом кратком введении, надеюсь, сказано достаточно, чтобы пробудить в вас желание посмотреть на технологию в действии. Позвольте мне продемонстрировать реализацию SOAP, которой я буду пользоваться, объяснить, почему я сделал этот выбор, и перейти к созданию кода.

Установка

Изучив основы, мы готовы перейти к более интересной части – коду. Вам потребуется проект или продукт, который вы будете применять, и найти его оказывается проще, чем можно подумать. Если есть необходимость создать Java-приложение с применением SOAP, то даже не придется искать. Существует две группы продуктов: коммерческие и свободные. В большинстве случаев я стараюсь исключать из рассмотрения коммерческие продукты. Дело не в том, что они плохи (наоборот, некоторые из них просто замечательны). Просто я хочу, чтобы любой читатель мог опробовать все примеры. А это требует доступности, которой не обеспечивают коммерческие продукты; за их использование необходимо платить, либо после их загрузки они перестают работать по истечении некоторого периода времени.

А потому мы сосредоточим свое внимание на свободно распространяемых продуктах. В этой области мне известен лишь один доступный проект: Apache SOAP. Он доступен в Интернете по адресу *http://xml.apache.org/soap* и предоставляет набор инструментов SOAP для Java. В настоящее время с веб-сайта Apache можно загрузить версию 2.2. Во всех примерах данной главы используется именно эта версия.

Другие возможности

Перед тем как перейти к установке и настройке Apache SOAP, отвечу на несколько вопросов, которые, весьма вероятно, вы хотели бы задать. Наверное, ясно, почему я не пользуюсь коммерческим продуктом. Однако может вызывать удивление, почему здесь не рассматриваются другие свободно распространяемые инструменты, которые тоже могут быть интересны многим.

Как насчет SOAP4J от IBM?

Первой в списке возможностей стоит реализация SOAP от IBM – IBM SOAP4J. Работа, проделанная в IBM, фактически является основой текущего проекта Apache SOAP аналогично тому, как IBM XML4J стал основой проекта Apache Xerces. Ожидается, что реализация IBM вновь обретет жизнь, вобрав в себя реализацию проекта Apache SOAP. Подобное произошло с IBM XML4J; в настоящее время он представляет собой лишь обертку для Xerces от IBM. Это позволяет добавить к версиям, распространяемым с исходными кодами, дополнительный уровень поддержки, хотя оба проекта (Apache и IBM) построены на одном ядре.

А Microsoft в этом участвует?

Да. Без сомнения, Microsoft и их реализация SOAP, а также и вся инициатива .NET (более подробно мы коснемся ее в следующей главе) очень важны. На самом деле, я хотел уделить время подробному рассмотрению реализации SOAP от Microsoft, но она поддерживает лишь COM-объекты и тому подобное, но не поддерживает Java. Рассмотрение этой реализации не «вписывается» в книгу про Java и XML. Однако Microsoft (несмотря на отношение разработчиков к этой компании) выполняет важную работу в области веб-служб, и списывать ее со счетов было бы ошибкой, по крайней мере, в этом отношении. Тем, кто должен обеспечивать взаимодействие с компонентами COM или Visual Basic, я настоятельно рекомендую взглянуть на набор инструментов Microsoft SOAP, который можно найти на сайте *http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000523*, как и на другие многочисленные ресурсы, посвященные SOAP.

Что такое Axis?

Те, кто наблюдает за разработками проекта Apache, могли слышать об Apache Axis. Axis – это набор инструментов SOAP следующего поколения, также разработанный под крышей проекта Apache XML. Учитывая довольно быстрое развитие и радикальные изменения в SOAP (в части спецификации, а не конкретных реализаций), уследить за ним сложно. Пытаться создать версию SOAP, которая соответствует существующим требованиям и которая идет в ногу с появлением но-

вых, невероятно сложно. В результате функциональность Apache SOAP несколько ограничена с точки зрения конструкции. Вместо того чтобы пытаться изменить архитектуру существующего набора инструментов, в Apache начали работу над другим проектом – с чистого листа. Так родился Axis. Кроме того, предполагалось, что название SOAP изменится на XP, и затем на XMLP. В результате имя нового SOAP-проекта получилось независящим от названия спецификации. Такова история «Axis». Но теперь, похоже, консорциум W3C передумал, и все идет к тому, что спецификация все-таки будет называться SOAP (версия 1.2 или 2.0), и все запутывается еще больше!

Представьте, что IBM SOAP4J – это первая архитектура набора инструментов SOAP. За ней следует Apache SOAP (рассматриваемая в этой главе) – это вторая архитектура. Наконец, Axis представляет собой архитектуру следующего поколения – архитектуру с номером три. В Axis применяется SAX, тогда как Apache SOAP построен на DOM. Кроме того, Axis реализует более удобные способы работы с заголовками, которого не хватало Apache SOAP. Принимая во внимание все эти прелести, читатели, думаю, весьма удивлены, что я не рассматриваю Axis. Просто делать это еще очень рано. Axis в настоящее время существует лишь в виде версии 0.51. Это еще не бета-версия, и даже не альфа. Так что рассматривать ее еще очень и очень рано. И хотя мне бы хотелось рассмотреть все новые возможности Axis, вряд ли ваш начальник позволит применять недоработанную версию программного обеспечения с открытыми исходными кодами в системах, от которых требуется стабильность, верно? В результате я решил рассмотреть то, чем можно пользоваться уже сегодня: Apache SOAP. Уверен, что когда выйдет завершенная версия Axis, я обновлю эту главу в переработанном варианте книги. А до тех пор сконцентрируемся на решении, которое можно использовать.

Установка

В SOAP существует две формы установки. Первая: запуск SOAP-клиента, взаимодействие которого с сервером, умеющим принимать сообщения SOAP, обеспечивается средствами SOAP API. Вторая: запуск SOAP-сервера, умеющего принимать сообщения от клиента SOAP. В этом разделе рассмотрены оба случая.

Клиент

Для использования SOAP на стороне клиента сначала надо загрузить Apache SOAP, доступный по адресу *http://xml.apache.org/dist/soap*. (Я загрузил версию 2.2 в скомпилированном виде из подкаталога *version-2.2*). Затем следует извлечь содержимое архива в каталог на локальной машине (на машине с Windows это был каталог *c:\javaxml2*, а на машине с Mac OS X – */javaxml2*). В результате я получил каталог */javaxml2/soap-2_2*. Кроме того, необходимо загрузить пакет

JavaMail, доступный на сайте Sun *http://java.sun.com/products/javamail/*. Он предназначен для поддержки протокола передачи SMTP, добавленного в Apache SOAP. Затем загрузите JavaBeans Activation Framework (JAF), также от Sun, с сайта *http://java.sun.com/products/beans/glasgow/jaf.html*. По-прежнему предполагается, что у вас есть анализатор Xerces или иной XML-анализатор.

Примечание

Убедитесь, что анализатор XML совместим с JAXP и поддерживает пространства имен. За исключением особых случаев, большинство анализаторов удовлетворяют этим требованиям. Если же вы столкнулись с проблемами, вернитесь к анализатору Xerces.

Используйте новые версии Xerces; версия 1.4 или выше должна подойти. Существует ряд проблем с SOAP и Xerces 1.3(.1), так что я бы бежал от такой комбинации, как от чумы.

Распакуйте пакеты JavaMail и JAF, а затем добавьте включенные в них *jar*-файлы в пути к классам и повторите операцию для библиотеки *soap.jar*. Каждый из этих *jar*-файлов находится либо в корневом каталоге, либо в каталоге *lib/* соответствующего установленного пакета. В конце концов переменная `CLASSPATH` должна выглядеть так:

```
$ echo $CLASSPATH
/javaxml2/soap-2_2/lib/soap.jar:/javaxml2/lib/xerces.jar:
/javaxml2/javamail-1.2/mail.jar:/javaxml2/jaf-1.0.1/activation.jar
```

Или в Windows:

```
c:\>echo %CLASSPATH%
c:\javaxml2\soap-2_2\lib\soap.jar;c:\javaxml2\lib\xerces.jar;
c:\javaxml2\javamail-1.2\mail.jar;c:\javaxml2\jaf-1.0.1\activation.jar
```

Наконец, добавьте каталог *javaxml2/soap-2_2/* в пути к классам, если хотите испытать примеры SOAP. Установка конкретных примеров будет описана в этой главе, по мере их рассмотрения.

Сервер

Организация набора серверных компонентов, работающих с SOAP, требует присутствия среды исполнения сервлетов. Как и в предыдущих главах, в примерах этой главы подразумевается работа с Apache Tomcat (доступен на сайте *http://jakarta.apache.org*). Таким образом, следует добавить все, что нужно клиенту, в пути к классам сервера. Проще всего это сделать, перетащив файлы *soap.jar*, *activation.jar* и *mail.jar*, а также анализатор в каталог библиотеки среды исполнения сервлетов. В Tomcat это просто каталог *lib/*, содержащий библиотеки, которые должны загружаться автоматически. Если необходима поддержка сценариев (в данной главе мы не будем их рассматривать, но соответствующий пример есть в Apache SOAP), в этот же каталог сле-

дует поместить файлы *bsf.jar* (доступен на сайте *http://oss.software.ibm.com/developerworks/projects/bsf*) и *js.jar* (доступен на сайте *http://www.mozilla.org/ rhino/*).

Примечание

Если используется Xerces с Tomcat, придется переименовать файлы, о чем уже говорилось в главе 10. Переименуйте файл *parser.jar* в *z_ parser.jar*, а *jaxp.jar* в *z_jaxp.jar*, чтобы гарантировать загрузку *xerces.jar* и включенной в него версии JAXP до любой другой реализации анализатора или JAXP.

Осталось перезапустить среду исполнения сервлетов, после чего можно переходить к написанию серверных компонентов SOAP.

Сервлет-маршрутизатор и клиент-администратор

В состав Apache SOAP входят сервлет-маршрутизатор (router servlet) и клиент-администратор (admin client). Даже тем, кто не собирается ими пользоваться, я советую их установить, чтобы иметь возможность проверить работу SOAP. Процесс установки варьируется в зависимости от среды исполнения сервлетов, а здесь мы рассмотрим лишь случай с Tomcat. Однако инструкции по установке для некоторых других сред исполнения сервлетов можно найти на странице *http://xml.apache.org/soap/docs*.

Установка в Tomcat не представляет сложности; просто возьмите файл *soap.war* из каталога *soap-2_2/webapps* и перенесите его в каталог *$TOMCAT_HOME/webapps*. Вот и все! Для проверки установки укажите в веб-броузере адрес *http://localhost:8080/soap/servlet/rpcrouter*. Вы должны получить ответ, подобный представленному на рис. 12.2.

Рис. 12.2. Сервлет – RPC-маршрутизатор

И хотя это похоже на ошибку, на самом деле все работает верно. Вы должны получить тот же ответ, если укажете в броузере адрес клиента-администратора *http://localhost:8080/soap/servlet/messagerouter*.

В качестве последней проверки и сервера и клиента убедитесь, что в точности выполнили все предшествующие инструкции, а затем запустите следующий класс Java, как показано ниже, передав URL сервлета для RPC-маршрутизатора:

```
C:\>java org.apache.soap.server.ServiceManagerClient
         http://localhost:8080/soap/servlet/rpcrouter list
Deployed Services:
```

Как и отражено в этом примере, вы должны получить пустой список служб. Получив любое другое сообщение, обратитесь к длинному списку возможных ошибок, доступному по адресу *http://xml.apache.org/soap/docs/trouble*. В нем представлен довольно полный перечень проблем, с которыми можно столкнуться. Если же вы получите пустой список служб, это означает, что установка завершена и можно переходить к примерам из оставшейся части главы.

«Пачкаемся»

Написание любой системы, основанной на SOAP, сводится к трем шагам, которые мы сейчас рассмотрим:

- Выбор между SOAP-RPC и SOAP-обменом сообщениями
- Написание службы SOAP или получение доступа к ней
- Написание клиента SOAP или получение доступа к нему

Первый шаг: необходимо решить, будет ли SOAP применяться для вызовов в стиле RPC, когда процедура на сервере вызывается удаленно, или для отправки сообщений, когда клиент просто отправляет серверу фрагменты информации. В следующем разделе эти процессы рассмотрены подробно. Приняв это решение, необходимо получить доступ к службе или разработать ее. И раз все мы профессионалы в Java, в этой главе будет рассказано, как написать собственную службу. Наконец, необходимо создать клиент для этой службы, после чего наблюдать, как все заработает.

RPC или обмен сообщениями?

Первая задача, на самом деле, связана не с написанием кода, а с проектированием. Необходимо определить, нужна ли вам RPC-служба или служба сообщений. С первой из них – RPC-службой – вы должны быть уже немного знакомы после прочтения предыдущей главы. Клиент вызывает процедуру на некотором сервере, а затем получает некий ответ. В таком варианте SOAP представляет собой просто более совер-

шенную систему XML-RPC, позволяя лучше обрабатывать ошибки и передавать по сети сложные типы данных. Эта концепция должна быть вам уже понятна, а поскольку написать в SOAP RPC-систему просто, начнем с нее. В этой главе показано, как написать RPC-службу, а затем и RPC-клиент, и заставить систему работать.

Второй вариант SOAP-взаимодействия основан на обмене сообщениями. Здесь акцент делается не на удаленных вызовах процедур, а на передаче информации. Как можно догадаться, это довольно мощный механизм, работа которого не зависит от познаний клиента в области методов, существующих на некотором сервере. Кроме того, обмен сообщениями более адекватен в контексте распределенных систем, позволяя передавать пакеты данных (в переносном смысле, а не в смысле сети) и организуя циркуляцию информации в многочисленных системах. Этот механизм одновременно является и более сложным, если сравнивать его применение с простым программированием в стиле RPC, и мы изучим его в следующей главе, наряду с другими аспектами разработки межкорпоративных (business-to-business) приложений. А пока что вам следует освоиться с программированием SOAP-RPC.

Как и в большинстве вопросов проектирования, принятие решения ложится на разработчика. Изучите свои приложения и определите, что именно требуется получить от SOAP. Если речь идет о сервере и наборе клиентов, которые должны лишь удаленно запрашивать выполнение задач, скорее всего, можно обойтись механизмом RPC. Но в крупных системах, сосредоточенных на обмене данными, а не предоставлении функциональности по запросу, будет более эффективен обмен сообщениями SOAP.

Служба RPC

Отбросив формальности, мы стремительно переходим к делу. Как вы помните из главы 11, в RPC необходимо иметь класс, методы которого будут вызываться удаленным образом.

Модули кода

Начну с того, что продемонстрирую несколько *модулей кода* (*code artifacts*), которые должны находиться на сервере. В данном случае модуль – это класс с методами, которые доступны RPC-клиентам.[1] Вместо того чтобы пользоваться простым классом из предыдущей главы, я привожу более сложный пример, чтобы показать возможности SOAP.

[1] Можно использовать сценарии через систему Bean Scripting Framework, но чтобы не занимать напрасно место, я не буду рассматривать здесь эту возможность. Следите за выходящими в O'Reilly книгами о SOAP, а также посетите страницу с документацией *http://xml.apache.org/soap*, где можно найти подробности о поддержке сценариев в SOAP.

Итак, класс в примере 12.4 реализует хранение каталога компакт-дисков. Такой класс был бы полезен при разработке электронного магазина музыкальных записей. Это базовая версия, которую мы будем постепенно дорабатывать.

Пример 12.4. Класс CDCatalog

```
package javaxml2;

import java.util.Hashtable;

public class CDCatalog {

    /** Компакт-диски по названию */
    private Hashtable catalog;

    public CDCatalog() {
        catalog = new Hashtable();

        // Заполняем каталог
        catalog.put("Nickel Creek", "Nickel Creek");
        catalog.put("Let it Fall", "Sean Watkins");
        catalog.put("Aerial Boundaries", "Michael Hedges");
        catalog.put("Taproot", "Michael Hedges");
    }

    public void addCD(String title, String artist) {
        if ((title == null) || (artist == null)) {
            throw new IllegalArgumentException("Ни заголовок, ни исполнитель
не могут быть пустыми.");
        }
        catalog.put(title, artist);
    }

    public String getArtist(String title) {
        if (title == null) {
            throw new IllegalArgumentException("Заголовок не может быть
                                                пустым.");
        }

        // Возвращаем информацию о диске
        return (String)catalog.get(title);
    }

    public Hashtable list() {
        return catalog;
    }
}
```

Этот класс позволяет добавлять новые компакт-диски, искать исполнителя по их названиям и получить список всех доступных дисков. Обратите внимание, что метод `list()` возвращает хеш-таблицу, чтобы все это работало, не нужны дополнительные «телодвижения» – Apache

SOAP, так же как и XML-RPC, обеспечивает автоматическое отображение для типа `Hashtable`.

Скомпилируйте этот класс и убедитесь, что все набрано (или загружено, если вы загружали примеры) верно. Обратите внимание, что класс `CDCatalog` ничего не знает о SOAP. Это означает, что существующие классы Java можно сделать доступными через SOAP-RPC с минимальными трудозатратами.

Дескрипторы развертывания

Написав код на Java, следует определить дескриптор развертывания. Он задает следующие ключевые моменты для сервера SOAP:

- URN службы SOAP, к которой имеют доступ клиенты
- Метод или методы, доступные клиентам
- Обработчики сериализации и десериализации для пользовательских классов

В первом пункте речь идет об идентификаторе, похожем на URL, который необходим для подключения клиента к серверу SOAP. Второй пункт вполне очевиден: это список методов, позволяющих клиенту узнать, какие модули доступны для SOAP-клиента. Кроме того, он указывает SOAP-серверу, который мы вскоре рассмотрим, какие запросы принимать. Третий пункт позволяет указать SOAP-серверу, как следует обрабатывать пользовательские параметры. В следующем разделе, по мере добавления функциональности к каталогу, я вернусь к этому вопросу.

Рассмотрим дескриптор развертывания и его элементы. В примере 12.5 приведен дескриптор для созданной нами службы `CDCatalog`.

Пример 12.5. Описатель развертывания CDCatalog

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:cd-catalog"
>
  <isd:provider type="java"
                scope="Application"
                methods="addCD getArtist list"
  >
    <isd:java class="javaxml2.CDCatalog" static="false" />
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</
isd:faultListener>
</isd:service>
```

Во-первых, я ссылаюсь на пространство имен развертывания Apache SOAP, а затем задаю URN службы в виде значения атрибута `id`. Это имя должно быть описательным и уникальным. Я поступил настолько же оригинально, насколько Дэйв Мэтьюс (Dave Matthews) оригиналь-

но назвал свою группу,[1] но это позволяет достичь желаемого результата. Затем с помощью элемента `java` я определил доступный для вызовов класс, включив и имя пакета (в атрибуте `class`), и указал, что предоставляемые методы – не статические (атрибут `static`).

Затем я указал применяемую реализацию регистратора ошибок (fault listener). В реализации Apache SOAP их две; я пользуюсь первой – `DOMFaultListener`. Этот регистратор возвращает все исключения и информацию об ошибках через дополнительный элемент DOM в ответе клиенту. Мы вернемся к этому вопросу при написании клиента, так что не волнуйтесь сейчас по этому поводу. Другая реализация регистратора ошибок – `org.apache.soap.server.ExceptionFaultListener`. Он передает сведения об ошибках через дополнительный параметр, возвращаемый клиенту. Поскольку во многих SOAP-приложениях уже применяются такие XML API, как DOM, а сами приложения разрабатываются на языке Java, распространенной практикой является применение в большинстве случаев `DOMFaultListener`.

Развертывание службы

Теперь у нас есть готовый дескриптор развертывания и набор модулей кода, которые должны быть предоставлены клиентам, и мы готовы к развертыванию службы. В составе Apache SOAP поставляется утилита, решающая эту задачу, при условии, что вы уже провели подготовительную работу. Во-первых, следует иметь дескриптор развертывания для службы (его мы только что рассмотрели). Во-вторых, необходимо сделать классы службы доступными SOAP-серверу. Самый лучший способ сделать это – создать *jar*-файл для класса службы из предыдущего раздела:

```
jar cvf javaxml2.jar javaxml2/CDCatalog.class
```

Перенесите этот *jar*-файл в каталог *lib/* (или в любой другой каталог библиотек, автоматически загружаемых средой исполнения сервлетов) и перезапустите Tomcat.

Внимание

Сделав это, вы «зафиксировали» состояние файла класса. Чтобы изменения были зафиксированы средой исполнения сервлетов, мало внести их в исходный файл *CDCatalog.java* и повторно скомпилировать его. Следует повторно создать *jar*-архив и заново скопировать его в каталог *lib/*, после чего следует перезапустить среду исполнения сервлетов. Эта процедура необходима для корректного обновления службы.

[1] Группа Дэйва Мэтьюса называется «Dave Matthews Band». Точно, оригинальнее не придумать. – *Примеч. перев.*

Теперь, когда класс (или классы) службы доступны SOAP-серверу, можно развернуть службу при помощи вспомогательного класса Apache SOAP `org.apache.soap.server.ServiceManager`:

```
C:\javaxml2\Ch12>java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter deploy xml\CDCatalogDD.xml
```

Первый аргумент – это SOAP-сервер или сервлет-маршрутизатор, второй – предпринимаемое действие, а третий – соответствующий дескриптор развертывания. После выполнения команды убедитесь, что служба добавлена:

```
(gandalf)/javaxml2/Ch12$ java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter list
Deployed Services:
        urn:cd-catalog
        urn:AddressFetcher
        urn:xml-soap-demo-calculator
```

Как минимум, вы должны увидеть все службы, доступные на сервере. Наконец, можно легко «свернуть» службу, если известно ее имя:

```
C:\javaxml2\Ch12>java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter undeploy urn:cd-catalog
```

При каждом обновлении исходного кода необходимо удалять и повторно устанавливать службу, чтобы гарантировать, что на SOAP-сервере выполняется самая последняя копия.

RPC-клиент

Далее следует клиент. Не хочу ничего усложнять и просто напишу пару программ, выполняемых в командной строке и вызывающих SOAP-RPC. Практически невозможно предугадать конкретный случай, поэтому я сконцентрируюсь на подробностях, связанных с SOAP, и предоставлю вам решать задачу интеграции с существующим программным обеспечением. Написав часть кода, выполняющую необходимые задачи, для каждого вызова SOAP-RPC следует предпринять несколько основных шагов:

- Создать вызов SOAP-RPC
- Установить соответствия типов для пользовательских параметров
- Задать URI применяемой SOAP-службы
- Задать вызываемый метод
- Задать используемое кодирование
- Добавить параметры для вызова
- Соединиться со службой SOAP
- Получить и интерпретировать ответ

Может показаться, что этого много, но большинство действий – это вызовы методов, занимающие одну-две строки. Другими словами, общаться со службой SOAP обычно очень просто. В примере 12.6 показан код класса CDAdder, позволяющего добавлять новый компакт-диск в каталог. Взгляните на код, а что в нем происходит, мы обсудим чуть позже.

Пример 12.6. Класс CDAdder

```
package javaxml2;

import java.net.URL;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;

public class CDAdder {

    public void add(URL url, String title, String artist)
        throws SOAPException {

        System.out.println("Добавляю компакт-диск '" + title + "'
            исполнителя'" +
            artist + "'");

        // Создаем объект Call
        Call call = new Call();
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("addCD");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // Установка параметров
        Vector params = new Vector();
        params.addElement(new Parameter("title", String.class, title, null));
        params.addElement(new Parameter("artist", String.class,
                                        artist, null));
        call.setParams(params);

        // Вызов
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault()) {
            System.out.println("Компакт-диск добавлен успешно.");
        } else {
            Fault fault = response.getFault();
            System.out.println("Произошла ошибка: " +
                               fault.getFaultString());
        }
    }
```

```
    public static void main(String[] args) {
        if (args.length != 3) {
            System.out.println("Использование: java javaxml2.CDAdder [URL
SOAP-сервера] " +
                "\"[Название диска]\" \"[Исполнитель]\"");
            return;
        }

        try {
            // URL SOAP-сервера, с которым происходит соединение
            URL url = new URL(args[0]);

            // Получаем значения для новых компакт-дисков
            String title = args[1];
            String artist = args[2];

            // Добавляем компакт-диск
            CDAdder adder = new CDAdder();
            adder.add(url, title, artist);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Эта программа получает URL SOAP-сервера, с которым должна соединиться, а также информацию, необходимую для создания и добавления нового компакт-диска в каталог. Затем в методе `add()` создается объект SOAP `Call`, с которым и происходит все взаимодействие. URI службы SOAP и вызываемый метод устанавливаются для вызова и соответствуют значениям из дескриптора развертывания службы, который приводится в примере 12.5. Затем указывается кодирование, которое всегда должно быть равным постоянной `Constants.NS_URI_SOAP_ENC`, если только речь не идет об особенном кодировании.

Программа создает новый объект `Vector`, заполненный объектами SOAP `Parameter`. Каждый из них представляет параметр указанного метода, а раз метод `addCD()` (для использования в XML и при отладке), класс для параметра и значение. Четвертый аргумент – это необязательный тип кодирования, который может указываться в случаях, когда для параметра необходимо реализовать особое кодирование. В противном случае достаточно значения `null`. Готовый объект `Vector` добавляется к объекту `Call`.

Закончив настройку вызова, вызовите его метод `invoke()`. Данный метод возвращает экземпляр `org.apache.soap.Response`, который позволяет выяснить, какие проблемы возникли при вызове. Здесь все достаточно очевидно, и вы можете самостоятельно разобраться с этим кодом. Скомпилировав клиент и выполнив приведенные ранее инструкции, касающиеся установки переменной `CLASSPATH`, запустите пример следующим образом:

```
C:\javaxml2\build>java javaxml2.CDAdder
   http://localhost:8080/soap/servlet/rpcrouter
   "Riding the Midnight Train" "Doc Watson"

Добавляю компакт-диск 'Riding the Midnight Train' исполнителя 'Doc Watson'
Компакт-диск добавлен успешно
```

В примере 12.7 приведен еще один простой класс `CDLister`**, перечисляющий все существующие в каталоге компакт-диски. Не будем рассматривать его в подробностях, поскольку он очень похож на класс из примера 12.6 и является более совершенной версией того, о чем я уже говорил.**

Пример 12.7. Класс CDLister

```
package javaxml2;

import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;

 org.apache.soap.rpc.Response;

public class CDLister {

    public void list(URL url) throws SOAPException {
        System.out.println("Перечень компакт-дисков.");

        // Создаем объект Call
        Call call = new Call();
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("list");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // Параметры не нужны

        // Вызов
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault()) {
            Parameter returnValue = response.getReturnValue();
            Hashtable catalog = (Hashtable)returnValue.getValue();
            Enumeration e = catalog.keys();
            while (e.hasMoreElements()) {
                String title = (String)e.nextElement();
                String artist = (String)catalog.get(title);
                System.out.println("  '" + title + "' by " + artist);
            }
        } else {
            Fault fault = response.getFault();
            System.out.println("Произошла ошибка: " +
fault.getFaultString());
```

```
        }
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Использование: java javaxml2.CDAdder
                                [URL сервера SOAP]");
            return;
        }

        try {
            // URL сервера SOAP, с которым мы соединяемся
            URL url = new URL(args[0]);

            // Перечисляем компакт-диски
            CDLister lister = new CDLister();
            lister.list(url);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Единственное отличие этого метода от класса `CDAdder` заключается в том, что объект `Response` имеет возвращаемое значение (хеш-таблицу из метода `list()`). Оно возвращается как объект `Parameter`, что позволяет клиенту сверить тип кодирования, а затем выделить и само значение, возвращаемое методом. Полученное возвращенное значение может использоваться как и любой другой объект Java. В данном примере выполняется обход каталога компакт-дисков и печать информации для каждого из них. Теперь это клиентское приложение можно проверить в действии:

```
C:\javaxml2\build>java javaxml2.CDLister
    http://localhost:8080/soap/servlet/rpcrouter
Перечень компакт-дисков.
  'Riding the Midnight Train' by Doc Watson
  'Taproot' by Michael Hedges
  'Nickel Creek' by Nickel Creek
  'Let it Fall' by Sean Watkins
  'Aerial Boundaries' by Michael Hedges
```

Вот, пожалуй, и все, что можно сказать об основных возможностях RPC в SOAP. Правда, я хочу пойти чуть дальше и поговорить о некоторых более сложных вещах.

Идем дальше

И хотя вы уже можете делать в SOAP все, что умели делать в XML-RPC, о SOAP еще многое можно сказать. Как уже говорилось в начале главы, SOAP обеспечивает две важные вещи – возможность задания

пользовательских параметров с минимальными усилиями и более совершенную обработку ошибок. Здесь рассмотрены обе эти темы.

Типы пользовательских параметров

Наш каталог компакт-дисков, по крайней мере на этот момент, наиболее неудобен тем, что в нем для произвольного компакт-диска хранится только название и исполнитель. Практичнее было бы иметь объект (или набор объектов), представляющий компакт-диск с названием, исполнителем, издателем, списком песен, возможно, жанром и прочей информацией такого рода. Не буду создавать всю структуру целиком, а перейду от названия и исполнителя к объекту `CD`, который будет хранить название, имя исполнителя и название компании-издателя. Этот объект будет передаваться от клиента на сервер и обратно, и с его помощью мы увидим, как SOAP может обрабатывать пользовательские типы. Этот новый класс приведен в примере 12.8.

Пример 12.8. Класс CD

```
package javaxml2;

public class CD {

    /** Название компакт-диска */
    private String title;

    /** Исполнитель */
    private String artist;

    /** Издатель компакт-диска */
    private String label;

    public CD() {
        // Конструктор по умолчанию
    }

    public CD(String title, String artist, String label) {
        this.title = title;
        this.artist = artist;
        this.label = label;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getArtist() {
        return artist;
    }
```

```
    public void setArtist(String artist) {
        this.artist = artist;
    }

    public String getLabel() {
        return label;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public String toString() {
        return "`" + title + "` by " + artist + ", on " +
            label;
    }
}
```

Это потребует внесения множества изменений и в класс CDCatalog. В примере 12.9 показана модифицированная версия этого класса. Изменения, которые связаны с применением нового класса CD, выделены.

Пример 12.9. Обновленный класс CDCatalog

```
package javaxml2;

import java.util.Hashtable;

public class CDCatalog {

    /** Компакт-диски по названию */
    private Hashtable catalog;

    public CDCatalog() {
        catalog = new Hashtable();

        // Заполняем каталог
        addCD(new CD("Nickel Creek", "Nickel Creek", "Sugar Hill"));
        addCD(new CD("Let it Fall", "Sean Watkins", "Sugar Hill"));
        addCD(new CD("Aerial Boundaries", "Michael Hedges", "Windham Hill"));
        addCD(new CD("Taproot", "Michael Hedges", "Windham Hill"));
    }

    public void addCD(CD cd) {
        if (cd == null) {
            throw new IllegalArgumentException("Объект CD не может быть
пустым.");
        }
        catalog.put(cd.getTitle(), cd);
    }

    public CD getCD(String title) {
        if (title == null) {
            throw new IllegalArgumentException("Название не может быть
пустым.");
        }
```

```
            // Возвращаем запрошенный компакт-диск
            return (CD)catalog.get(title);
        }

        public Hashtable list() {
            return catalog;
        }
    }
```

Помимо очевидных изменений, старый метод getArtist(String title) заменен новым – getCD(String title) – и возвращается объект типа CD. Это означает, что SOAP-сервер должен выполнять сериализацию и десериализацию нового класса, а клиенту требуется обновление. Сначала рассмотрим обновленный дескриптор развертывания, который подробно описывает вопросы сериализации, связанные с пользовательскими типами. Добавьте следующие строки в дескриптор развертывания для каталога компакт-дисков, а также измените названия доступных методов, проведя синхронизацию с обновленным классом CDCatalog:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:cd-catalog"
>
  <isd:provider type="java"
           scope="Application"
           methods="addCD getCD list"
  >
    <isd:java class="javaxml2.CDCatalog" static="false" />
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</
isd:faultListener>

  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:x="urn:cd-catalog-demo" qname="x:cd"
        javaType="javaxml2.CD"
        java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
       xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mappings>
</isd:service>
```

Новый элемент mappings задает способ обработки SOAP-сервером пользовательских параметров, таких как класс CD. Сначала нужно определить элемент map для каждого пользовательского типа. Для атрибута encodingStyle – по крайней мере, в Apache SOAP 2.2 – всегда нужно задавать значение *http://schemas.xmlsoap.org/soap/encoding/* – это единственный поддерживаемый в настоящее время тип кодирования. Кроме того, следует указать пространство имен для пользовательского типа, а затем название класса с префиксом данного пространства имен для этого типа. В данном случае это фиктивное пространство имен

с простым префиксом «x». Затем при помощи атрибута javaType передается имя класса Java: в данном случае javaxml2.CD. Наконец следуют волшебные атрибуты java2XMLClassName и xml2JavaClassName. Они определяют класс для преобразования из Java в XML и из XML в Java, соответственно. Я применяю невероятно удобный класс BeanSerializer, также поставляемый вместе с Apache SOAP. Если пользовательский параметр представлен в формате JavaBean, этот класс избавит вас от необходимости разработки собственного. Класс, который подвергается обработке, должен иметь конструктор по умолчанию (помните, мы определили пустой конструктор без аргументов в классе CD) и работать с данными при помощи методов setXXX и getXXX. Поскольку класс CD отвечает всем требованиям, класс BeanSerializer работает замечательно.

Примечание

Класс CD следует соглашениям JavaBean не случайно. Большинство классов данных отлично вписываются в этот формат, а я не хотел писать классы сериализации и десериализации. Писать их мучительно (это не чрезмерно сложно, но можно легко запутаться), и я советую в полной мере применять соглашения Bean в пользовательских классах-параметрах. Во многих случаях соглашения Bean требуют лишь того, чтобы в классе присутствовал конструктор по умолчанию (без аргументов).

Вновь создайте *jar*-файл службы и повторно установите ее:

```
(gandalf)/javaxml2/Ch12$ java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter deploy xml/CDCatalogDD.xml
```

Внимание

Если во время повторной установки службы среда исполнения сервлетов запущена, необходимо перезапустить ее, чтобы активизировать новые классы для службы SOAP, и вновь установить службу.

Теперь осталось лишь изменить клиентское приложение на предмет использования нового класса и методов. В примере 12.10 приведена обновленная версия класса клиента CDAdder. Изменения выделены жирным шрифтом.

Пример 12.10. Обновленный класс CDAdder

```
package javaxml2;

import java.net.URL;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
```

```
import org.apache.soap.rpc.Response;
import org.apache.soap.util.xml.QName;

public class CDAdder {

    public void add(URL url, String title, String artist, String label)
        throws SOAPException {

        System.out.println("Adding CD titled '" + title + "' by '" +
            artist + "', on the label " + label);

        CD cd = new CD(title, artist, label);

        // Создаем соответствие для типа
        SOAPMappingRegistry registry = new SOAPMappingRegistry();
        BeanSerializer serializer = new BeanSerializer();
        registry.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:cd-catalog-demo", "cd"),
            CD.class, serializer, serializer);

        // Создаем объект Call
        Call call = new Call();
        call.setSOAPMappingRegistry(registry);
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("addCD");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // Устанавливаем параметры
        Vector params = new Vector();
        params.addElement(new Parameter("cd", CD.class, cd, null));
        call.setParams(params);

        // Вызов
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault()) {
            System.out.println("Компакт-диск добавлен успешно.");
        } else {
            Fault fault = response.getFault();
            System.out.println("Произошла ошибка: " +
fault.getFaultString());
        }
    }

    public static void main(String[] args) {
        if (args.length != 4) {
            System.out.println("Использование: java javaxml2.CDAdder
          [URL SOAP-сервера] " + "\"[Название диска]\" \"[Исполнитель]\"
              \"[Издатель диска]\"");
            return;
        }

        try {
            // URL SOAP-сервера, с которым устанавливается соединение
            URL url = new URL(args[0]);
```

```
            // Получаем значения для новых компакт-дисков
            String title = args[1];
            String artist = args[2];
            String label = args[3];

            // добавляем компакт-диск
            CDAdder adder = new CDAdder();
            adder.add(url, title, artist, label);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Единственное действительно интересное изменение – это обработка соответствия (mapping) для класса CD:

```
// Создаем соответствие для типа
SOAPMappingRegistry registry = new SOAPMappingRegistry();
BeanSerializer serializer = new BeanSerializer();
registry.mapTypes(Constants.NS_URI_SOAP_ENC,
    new QName("urn:cd-catalog-demo", "cd"),
    CD.class, serializer, serializer);
```

Так можно закодировать параметр пользовательского типа и переслать его. Я уже говорил, как можно применить класс BeanSerializer для обработки параметров в формате JavaBean, как в случае с классом CD. Чтобы сообщить об этом серверу, я применил дескриптор развертывания; однако теперь мне необходимо сообщить клиенту о том, что он должен применять этот класс сериализации и десериализации. Что и позволяет сделать класс SOAPMappingRegistry. Методу mapTypes() передается тип кодирования (опять же, тут лучше всего использовать NS_URI_SOAP_ENC) и информация о типе параметра, для которого требуется особая сериализация. Сначала передается имя QName (уточненное имя XML). Именно поэтому в дескрипторе развертывания применялись необычные пространства имен; и здесь следует передать идентичный URN, а также локальное имя элемента (в данном случае CD). Затем передается объект Java Class сериализуемого класса (CD.class) и, наконец, экземпляр класса для сериализации и десериализации. В случае с BeanSerializer в обоих случаях подходит один и тот же экземпляр. После инициализации реестра необходимо сообщить о нем объекту Call при помощи метода setSOAPMapping-Registry().

Запустить этот класс можно так же, как и раньше, добавив к аргументам командной строки компанию-издателя компакт-диска. Все будет работать без ошибок:

```
C:\javaxml2\build>java javaxml2.CDAdder
    http://localhost:8080/soap/servlet/rpcrouter
    "Tony Rice" "Manzanita" "Sugar Hill"
Добавляю компакт-диск 'Tony Rice' исполнителя 'Manzanita', изданный Sugar Hill
Компакт-диск добавлен успешно.
```

Я предоставлю вам самостоятельно изменить класс CDLister подобным же образом, а в загружаемые примеры к этой книге входит уже обновленный класс.

Примечание

Можно подумать, что раз класс CDLister не работает непосредственно с объектом CD (метод list() возвращает хеш-таблицу), то не нужно вносить никаких изменений. Однако возвращаемая хеш-таблица содержит экземпляры объектов CD. Если SOAP не будет знать, как ее десериализовать, клиент сообщит об ошибке. Поэтому необходимо указать объекту Call на экземпляр SOAPMappingRegistry, чтобы все работало нормально.

Улучшенная обработка ошибок

Итак, вы создаете пользовательские объекты, выполняете RPC-вызовы и повергаете в смущение коллег. Поговорим, однако, о менее возвышенных материях: об обработке ошибок. В любой сетевой транзакции причин для ошибок может быть много – это и незапущенная служба, и ошибка на сервере, и ненайденные объекты, и несуществующие классы и целый ряд других проблем. До сих пор для диагностики ошибок мы обращались к методу fault.getString(). Но он не всегда эффективен. Чтобы проверить это утверждение в действии, закомментируйте следующую строку в конструкторе CDCatalog:

```
public CDCatalog() {
   //catalog = new Hashtable();

   // Заполняем каталог
   addCD(new CD("Nickel Creek", "Nickel Creek", "Sugar Hill"));
   addCD(new CD("Let it Fall", "Sean Watkins", "Sugar Hill"));
   addCD(new CD("Aerial Boundaries", "Michael Hedges", "Windham Hill"));
   addCD(new CD("Taproot", "Michael Hedges", "Windham Hill"));
}
```

Перекомпилируйте класс, перезапустите среду исполнения сервлетов и заново установите службу. В результате, когда конструктор класса попытается добавить компакт-диск к несуществующей хеш-таблице, возникнет исключение NullPointerException. Если запустить клиент, он сообщит, что произошла ошибка, но сообщение будет не очень полезным:

```
(gandalf)/javaxml2/build$ java javaxml2.CDLister
    http://localhost:8080/soap/servlet/rpcrouter
Перечень компакт-дисков.
Error encountered: Unable to resolve target object: null
(Произошла ошибка: Невозможно интерпретировать целевой объект: null)
```

Это не совсем та информация, которая нужна для того, чтобы выявить причину проблемы. Однако существует механизм, позволяющий более

эффективно диагностировать ошибки; помните объект DOMFaultListener, который вы указывали в качестве значения элемента faultListener? Именно сейчас он вступает в игру. Возвращаемый объект Fault в случае возникновения проблемы (как сейчас) содержит элемент DOM org.w3c.dom.Element с подробной информацией об ошибке. Прежде всего, добавьте оператор импортирования для java.util.Iterator в код клиентского приложения:

```
import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.util.xml.QName;
```

Затем внесите следующие изменения в код, отвечающий за обработку ошибок в методе list():

```
        if (!response.generatedFault()) {
            Parameter returnValue = response.getReturnValue();
            Hashtable catalog = (Hashtable)returnValue.getValue();
            Enumeration e = catalog.keys();
            while (e.hasMoreElements()) {
                String title = (String)e.nextElement();
                CD cd = (CD)catalog.get(title);
                System.out.println("  '" + cd.getTitle() + "' by " +
                   cd.getArtist( ) + " on the label " + cd.getLabel());
            }
        } else {
            Fault fault = response.getFault();
            System.out.println("Error encountered: " +
fault.getFaultString());

            Vector entries = fault.getDetailEntries();
            for (Iterator i = entries.iterator(); i.hasNext(); ) {
                org.w3c.dom.Element entry = (org.w3c.dom.Element)i.next();
                System.out.println(entry.getFirstChild().getNodeValue());
            }
        }
```

Применяя метод getDetailEntries(), мы получаем доступ к необработанным данным о сбойной ситуации, переданным службой SOAP и сервером. Код обходит их в цикле (обычно объект один, но стоит

подстраховаться) и выбирает элемент DOM `Element` из каждой записи. По существу, мы работаем со следующим XML-кодом:

```
<SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server.BadTargetObjectURI</faultcode>
  <faultstring> Невозможно интерпретировать целевой объект: null</
faultstring>
  <stacktrace>Здесь именно то, что нам нужно!</stackTrace>
</SOAP-ENV:Fault>
```

Другими словами, объект `Fault` предоставляет доступ к той части конверта SOAP, которая связана с обработкой ошибок. Кроме того, Apache SOAP предоставляет распечатку стека Java в случае возникновения ошибки, а это обеспечивает подробную информацию, необходимую для разрешения сбойной ситуации. Получив элемент `stackTrace` и отобразив значение узла `Text` для этого элемента, клиент распечатает цепочку стека, полученную с сервера. Скомпилируйте эти изменения и перезапустите клиентское приложение. Будут получены следующие результаты:

```
C:\javaxml2\build>java javaxml2.CDLister http://localhost:8080/soap/servlet/
rpcrouter
Просмотр текущего каталога компакт-дисков.
Произошла ошибка: Невозможно интерпретировать целевой объект: null
java.lang.NullPointerException
        at javaxml2.CDCatalog.addCD(CDCatalog.java:24)
        at javaxml2.CDCatalog.<init>(CDCatalog.java:14)
        at java.lang.Class.newInstance0(Native Method)
        at java.lang.Class.newInstance(Class.java:237)
```

Вывод растягивается еще на несколько строк, но уже видно, что возникла исключительная ситуация `NullPointerException` и отображаются даже номера строк в классах сервера, с которыми связаны проблемы. Результатом этого довольно незначительного изменения стал более надежный способ обработки ошибок. Это должно подготовить вас к отслеживанию ошибок в классах сервера. Да, перед тем как идти дальше, не забудьте произвести откат к старой версии класса `CDCatalog`, в которой такие ошибки не возникают!

Что дальше?

Следующая глава является непосредственным продолжением рассмотренных здесь тем. XML становится краеугольным камнем в межкорпоративных приложениях, а ключом к этим технологиям является SOAP. В следующей главе представлены две важные технологии – UDDI и WSDL. Если вы не знаете, что это такое, то вы на верном пути. Мы узнаем, как их можно объединить, чтобы сформировать стержень архитектуры веб-служб. Приготовьтесь узнать, наконец, на чем основана шумиха, связанная с веб-приложениями и взаимодействием peer-to-peer.

- *Веб-службы*
- *UDDI*
- *WSDL*
- *Собираем все вместе*
- *Что дальше?*

13

Веб-службы

Предыдущую главу я посвятил рассмотрению SOAP как самостоятельной технологии. SOAP-клиенты общались с SOAP-сервером, и при этом использовался только язык Java и среда исполнения сервлетов. Такое решение отлично подходит для случаев, когда все клиенты и службы создаются одним разработчиком, но оно довольно ограничено в плане взаимодействия с другими приложениями. Те, кто что-либо слышал о SOAP, знают, что именно возможности интеграции обеспечили ему такие хорошие отзывы в прессе. Глава 12 неполна, в ней ничего не сказано о том, как работать с другими приложениями, применяя SOAP в качестве транспортного протокола. Здесь же мы дополним картину и решим последние проблемы в области взаимодействия приложений.

Для начала дадим несколько простых определений термину «веб-службы», который страдает от злоупотребления. Это понятие в настоящее время с трудом поддается определению, поскольку всяк норовит использовать его в отношении собственного типа программного обеспечения и архитектуры, но некоторые общие принципы все же существуют. Практически во всех определениях веб-служб присутствуют слова о необходимости в обмене информацией с другими приложениями. Этот обмен требует наличия некоторых стандартов, и два самых важных из них (по крайней мере, на данный момент) – это UDDI и WSDL. Я рассмотрю оба и покажу, как они согласуются с SOAP. Наконец, мы попытаемся собрать все эти различные технологии в одном примере, завершающем данную главу.

Веб-службы

В наше время веб-службы оказались в центре внимания компьютерного мира, поэтому во втором издании этой книги нашлось место для данной главы. Однако определить это понятие довольно сложно – с точки

зрения одного человека оно может быть верным, но с точки зрения другого абсолютно неуместным. Попытаемся абстрагироваться от рекламной шумихи и различий конкретных реализаций и получить простой набор понятий, остающихся всегда и везде справедливыми.

Веб-службы – это взаимодействие приложений, необходимость в котором со временем только возрастает. При этом технологические нововведения только увеличивают число связанных с взаимодействием проблем. С появлением новых языков, более сложных структур данных, разнообразных потребностей корпоративных приложений увеличиваются и различия между системами (даже выполняющими сходные задачи!). Взаимодействие систем должно базироваться на общем языке. Но не на языке программирования, вроде Java, потому что программы, написанные на разных языках, должны общаться между собой. Речь о языке, которым, при наличии словаря, могут пользоваться все. Такой язык может применяться в качестве транспорта для систем, говорящих на любых языках.

XML решает проблему передачи данных и представляет собой одну из составляющих этого языка. Он доступен и применяется практически во всех существующих языках программирования: C, Perl, Python, LISP, Java, Pascal, C#, Visual Basic... список можно продолжать и продолжать. Но системы веб-служб направлены на дальнейшее развитие. Ключом к взаимодействию приложений являются не просто данные, но то, *какие* они. Какую информацию я могу получить от вашей системы? Какую вы можете получить от моей? Другими словами, должны существовать способы оповещения о том, какие службы предоставляет приложение. И именно в этой области до недавнего времени был зияющий провал.

Последние новости в области веб-служб начали заполнять этот пробел. Технология UDDI предоставляет возможность обнаружения существующих служб и регистрации собственных. Технология WSDL предлагает способ получить требуемую информацию по обнаруженной службе, чтобы клиент смог к ней обращаться. Здесь намеренно сейчас не говорится, что означают аббревиатуры UDDI и WSDL, поскольку сначала необходимо рассмотреть картину в целом.

На рис. 13.1 показано, как протекает процесс. Сначала поставщик службы (service provider) создает службу (подобно тому, как это делали мы в предыдущей главе). Служба может быть примитивной – просто добавлять компакт-диски в каталог, или достаточно сложной – хранить VIN (идентификационные номера транспортных средств, vehicle identification numbers) для всех автомобилей, зарегистрированных в Массачусетсе и используемых правительством. Если служба доступна через Интернет, она является кандидатом на регистрацию в *реестре служб* (*service registry*). Это все относится к UDDI. UDDI также позволяет производить поиск зарегистрированной службы по определенному

имени, такому как «cd-catalog» или «govt-bin-numbers». Реестр возвращает все найденные службы.

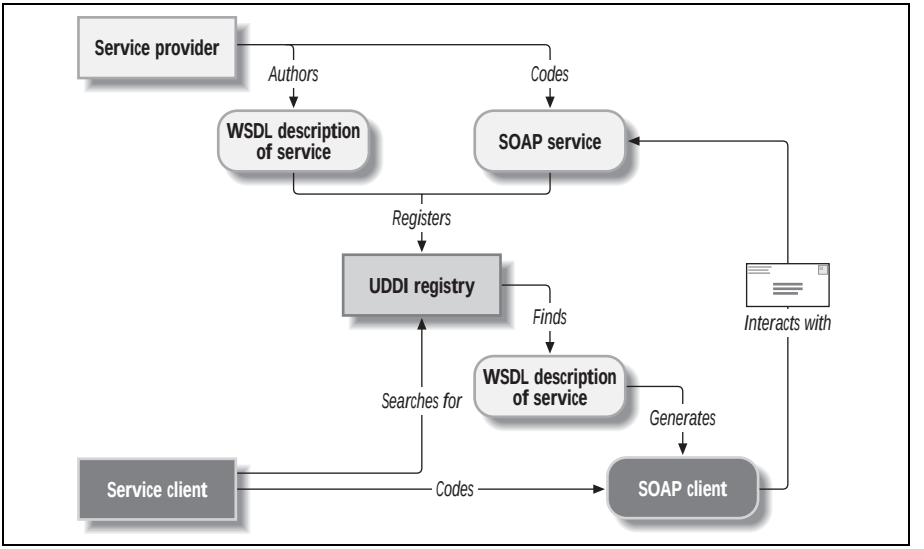

Рис. 13.1. Веб-службы

Можно надеяться, что клиент нашел желаемую службу. Но взаимодействие со службой не ограничивается именем; оно требует наличия URL, с которым мы соединяемся, обязательных параметров, а также возвращаемых значений. Здесь вновь применяется XML-формат WSDL, который мы также рассмотрим. Клиент работает с обнаруженной службой, и благодаря WSDL мы можем быть уверены, что служба используется корректно. Жизнь прекрасна, и все благодаря веб-службам. Конечно же, мы едва коснулись непростых понятий, но сейчас, когда вы имеете общее представление, рассмотрим их подробно.

UDDI

Без дальнейших проволочек я определю, что означает аббревиатура UDDI. Она расшифровывается как *Universal Discovery, Description, and Integration* и чаще всего связана со словом «реестр». Основной источник информации по UDDI – это веб-сайт *http://www.uddi.org* (рис. 13.2), также являющийся домашней страницей для реестра UDDI, столь важного для регистрации и поиска служб. На нем представлено описание проекта UDDI, который направлен на создание и определение полноценной системы для обмена данными, о которой мы будем говорить в этой главе. В поддержку этой инициативы выступают компании IBM и Microsoft, так что по всей вероятности ей обеспечено долгое будущее.

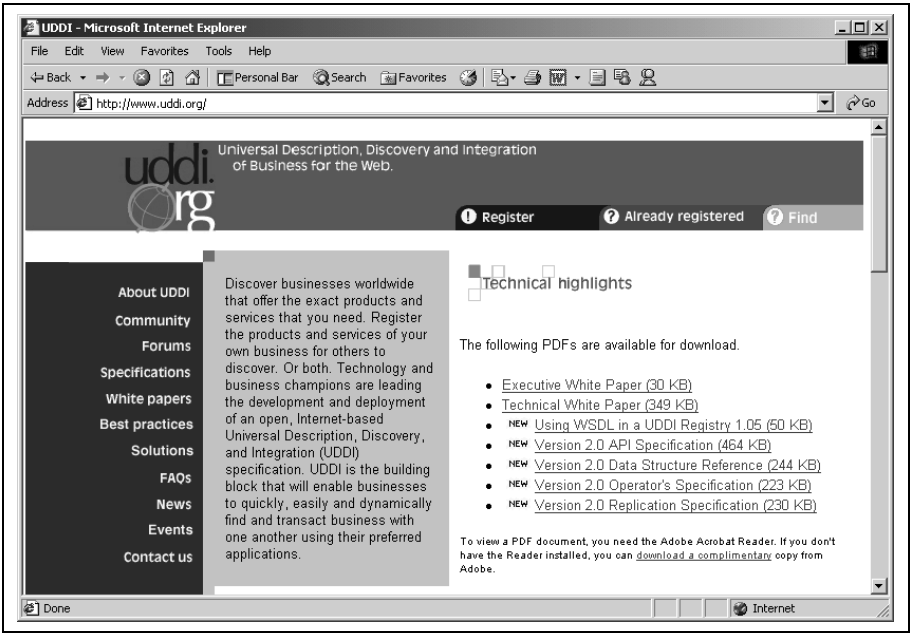

Рис. 13.2. Веб-сайт UDDI

Основой всего является сеть служб, о которых хранит информацию UDDI. Поиск и регистрация служб – это неотъемлемая возможность реестра, доступная на веб-сайте UDDI. Требуется лишь пройти простой процесс регистрации. Компании IBM и Microsoft приняли решение не взимать плату за регистрацию служб с компаний и частных лиц. В результате ежедневно регистрируется очень много служб, гораздо больше, чем в случае системы с платной регистрацией.

Такова вкратце система UDDI. Применение UDDI в большинстве случаев не сопряжено с какими-либо трудностями. Основные сложности касаются реализации системы, но мы с вами, будучи просто поставщиками служб или потребителями, этим вопросом не интересуемся. Существует несколько реализаций UDDI, доступных для загрузки и локального использования. Среди прочих я предпочитаю jUDDI. Этот Java-проект с открытыми исходными текстами доступен по адресу *http://juddi.sourceforge.net*. Кроме того, набор инструментов веб-служб IBM (рассматриваемый в разделе «WSDL» далее в этой главе) содержит ознакомительную версию коммерческого реестра UDDI. Здесь не будут рассматриваться ни jUDDI, ни реестр IBM UDDI, поскольку они помогают понять, скорее, как реализовать UDDI, а не как с ним работать. Тем, кому интересно узнать, как работает реестр UDDI, я советую обратиться к jUDDI. Если же вас интересует лишь создание веб-служб и предоставление их клиентам, то нет смысла изучать способы реализации реестра. Наконец, вопросы, связанные с регистрацией

и поиском служб, оставлены на последний раздел, в котором приводится довольно сложный пример, демонстрирующий реальное применение SOAP, UDDI и WSDL.

WSDL

WSDL – это язык описания веб-служб (Web Services Description Language). Полная спецификация по нему доступна на сайте *http://www.w3.org/TR/wsdl*, и в ней описано все, что нужно знать о службе, чтобы с ней взаимодействовать. Аналогично UDDI, WSDL – это достаточно простая технология (на деле даже не технология, а просто разметка), которая, несмотря на простоту, играет немаловажную роль в сфере веб-служб. Файлы WSDL описывают важные фрагменты информации, которая необходима клиентам служб, а именно:

- Имя службы, включая ее URN
- Адрес, по которому доступна служба (обычно это HTTP URL)
- Методы, доступные для вызова
- Типы параметров ввода и вывода для каждого метода

По отдельности эти фрагменты бесполезны, но вместе они дают клиентам полное представление о службе. Кроме того, WSDL-документ объединяет элементы схем XML, параметры в стиле XML-RPC и многое другое, о чем мы уже говорили. В примере 13.1 приведен фрагмент схемы WSDL для каталога компакт-дисков из предыдущей главы, он описывает лишь метод `getCD()` службы. Это лишь фрагмент, который призван дать представление о том, как выглядит WSDL-документ.

Пример 13.1. Фрагмент WSDL-документа

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

<definitions name="CDCatalog"
             targetNamespace="http://www.oreilly.com/javaxml2/cd-
catalog.wsdl"
             xmlns:cd="http://www.oreilly.com/javaxml2/cd-catalog.wsdl"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:cdXSD="http://www.oreilly.com/javaxml2/cd-catalog.xsd"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
>
  <types>
    <schema targetNamespace="http://www.oreilly.com/javaxml2/cd-catalog.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="Title">
        <complexType>
          <all><element name="title" type="string" /></all>
        </complexType>
```

```
      </element>
      <element name="CD">
        <complexType>
          <all>
            <element name="title" type="string" />
            <element name="artist" type="string" />
            <element name="label" type="string" />
          </all>
        </complexType>
      </element>
    </schema>
  </types>

  <message name="getCDInput">
    <part name="body" element="cdXSD:Title" />
  </message>

  <message name="getCDOutput">
    <part name="body" element="cdXSD:CD" />
  </message>

  <portType name="CDCatalogPortType">
    <operation name="getCD">
      <input message="cd:getCDInput" />
      <output message="cd:getCDOutput" />
    </operation>
  </portType>

  <binding name="CDCatalogBinding" type="cd:CDCatalogPortType">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="getCD">
      <soap:operation soapAction="urn:cd-catalog" />
      <input>
        <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:cd-catalog" />
      </input>
      <output>
        <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="urn:cd-catalog" />
      </output>
    </operation>
  </binding>

  <service name="CDCatalog">
    <documentation>CD Catalog Service from Java and XML</documentation>
    <port name="CDCatalogPort" binding="cd:CDCatalogBinding">
      <soap:address location="http://newInstance.com/soap/servlet/rpcrouter"
/>
```

```
    </port>
  </service>
</defintions>
```

Как можно видеть, формат для описания службы достаточно многословен. Но он довольно прост для понимания. В первую очередь, все типы данных, которыми необходимо обмениваться, описываются при помощи элемента `types` и синтаксиса в стиле схем XML.

Внимание

В настоящее время спецификация WSDL основана на спецификации схемы XML от 2000 года, а не на окончательном варианте, принятом в апреле 2001. До тех пор пока спецификация WSDL не будет исправлена, придется применять более старые конструкции схем.

Элемент `message` описывает взаимодействие клиента с сервером и сервера с клиентом. Описания этого типа объединены в элементе `portType` в целях определения доступных операций (в данном разделе вы также найдете дополнительные доступные методы). Элемент `binding` подробно описывает URN, по которому доступны операции, и способ получения доступа к ним, а элемент `service` объединяет определения. Чтобы не путаться, думайте об этом процессе как об иерархии.

SOAP – это все, что есть?

Не думайте, что служба SOAP – это единственный существующий тип веб-службы. Разумеется, можно создать программу (или программы), взаимодействующую с клиентами иным образом, и представить это взаимодействие через WSDL-файл. Скажем, вполне реальна служба XML-RPC. И даже если в ней нет конвертов и поддержки параметров, определяемых пользователем, она позволяет легко организовать взаимодействие с клиентами и описание параметров ввода/вывода в WSDL. При этом практически все службы, которые я видел (а видел я много!), реализованы на базе SOAP, так что SOAP является основным направлением. Но все же помните о том, что можно воспользоваться для реализации службы любым протоколом, не только SOAP.

В настоящее время реализация Apache SOAP не использует WSDL-документы напрямую. Другими словами, невозможно автоматически получить класс клиента на основе WSDL-документа. И хотя другие платформы, такие как Microsoft, ушли вперед, в рамках проекта Apache Axis работа над этими возможностями еще ведется. Пока же приходится интерпретировать WSDL-документ самостоятельно, а затем вручную писать код для клиента. И вообще говоря, это гораздо более занимательно.

Собираем все вместе

Добавив к информации о UDDI основные понятия, связанные с WSDL, можно переходить к примеру полноценной веб-службы. В этом разделе подробно описано создание службы SOAP (на этот раз службы сообщений), регистрация ее в реестре UDDI, поиск ее через UDDI и получение дескриптора WSDL, а затем – процесс взаимодействия клиента с этой службой.

Этот пример немного сложнее предыдущих. Вот его сценарий. CDs-R-Us – это новая компания, предлагающая компакт-диски распространителям по всему миру. Они оказались на рынке с заметным (прямо скажем) опозданием и хотят завоевать признание, предоставив высокотехнологичный интерфейс, разработанный в виде веб-службы, дабы упростить взаимодействие. Компания хочет предоставить возможность посылать XML-сообщения с запросами компакт-дисков через SOAP. Приложения будут заполнять эти заказы, находя компакт-диск на сервере каталога (на котором работает, конечно, сверхмощная версия службы CDCatalog из последней главы) и возвращая затем счет. Фактически речь идет о паре SOAP-транзакций: одна от клиента к CDs-R-Us, основанная на сообщениях, а вторая – внутренняя к CDs-R-Us, основанная на RPC. На рис. 13.3 показан ход всего процесса. Также компания хочет зарегистрировать свою службу сообщений в реестре UDDI, чтобы службу могли найти потенциальные клиенты.

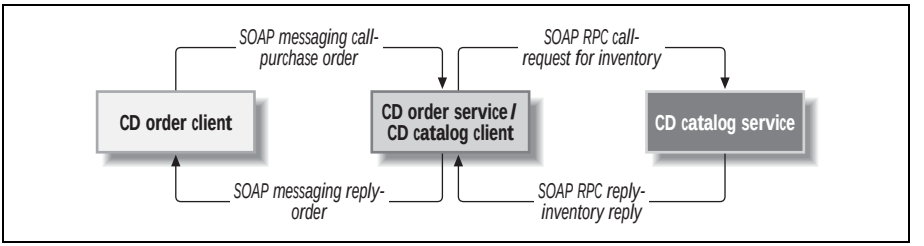

Рис. 13.3. Ход процесса для приложения из примера

Служба сообщений

Для реализации RPC-клиента мы воспользуемся программой `CDCatalog` из предыдущей главы, поэтому можно сразу перейти к новому коду – службе сообщений. Она должна получать заказ в формате XML и выполнять запрос к службе каталога, работающей на другой машине в локальной сети CDs-R-Us. Другими словами, служба сообщений по совместительству является SOAP-RPC-клиентом. Такая система вполне допустима и достаточно часто встречается в мире веб-служб. Одно приложение получает информацию от другого, и в свою очередь начинает взаимодействие с *еще одним* приложением. Если это по-прежнему кажется странным, спросите строителя дома, как много субподряд-

чиков он нанимает, а затем спросите каждого из них, сколько субподрядчиков нанимают *они*; от этого голова может пойти кругом!

Прежде всего, определим формат заказа (PO, от purchase order), требуемый компанией CDs-R-Us. Схема XML для формы заказа представлена в примере 13.2.

Пример 13.2. Схема XML po.xsd

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://www.cds-r-us.com"
           targetNamespace="http://www.cds-r-us.com">
  <xs:element name="purchaseOrder">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="recipient" />
        <xs:element ref="order" />
      </xs:sequence>
      <xs:attribute name="orderDate" type="xs:string" />
    </xs:complexType>
  </xs:element>

  <xs:element name="recipient">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name" />
        <xs:element ref="street" />
        <xs:element ref="city" />
        <xs:element ref="state" />
        <xs:element ref="postalCode" />
      </xs:sequence>
      <xs:attribute name="country" type="xs:string" />
    </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="street" type="xs:string" />
  <xs:element name="city" type="xs:string" />
  <xs:element name="state" type="xs:string" />
  <xs:element name="postalCode" type="xs:short" />

  <xs:element name="order">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="cd" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="cd">
    <xs:complexType>
      <xs:attribute name="artist" type="xs:string" />
```

```
      <xs:attribute name="title" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

После составления этой схемы обычная форма заказа будет выглядеть так, как показано в примере 13.3.

Пример 13.3. Пример формы заказа компакт-диска

```
<purcahseOrder orderDate="07.23.2001"
    xmlns="http://www.cds-r-us.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.cds-r-us.com po.xsd"
>
  <recipient country="USA">
    <name>Dennis Scannell</name>
    <street>175 Perry Lea Side Road</street>
    <city>Waterbury</city>
    <state>VT</state>
    <postalCode>05676</postalCode>
  </recipient>
  <order>
    <cd artist="Brooks Williams" title="Little Lion" />
    <cd artist="David Wilcox" title="What You Whispered" />
  </order>
</purchaseOrder>
```

Служба должна принимать XML-документы, подобные приведенному в примере 13.3, находить сопутствующую заказу информацию, а затем передавать эту информацию службе каталога компакт-дисков через RPC. Получив ответ, она составляет некий счет или подтверждение для клиента службы сообщений и отправляет сообщение. В примере я ничего не усложнял, но по мере рассмотрения кода вы легко увидите, где можно добавить дополнительную обработку.

Написание службы, принимающей сообщения в формате XML, несколько отличается от написания службы, принимающей RPC-запросы. Дело в том, что, принимая сообщения, нужно больше взаимодействовать с объектами запроса и ответа, а класс должен знать о SOAP. Вспомним, что при обработке в стиле RPC класс, получающий запросы, ничего не знает о RPC или SOAP, и потому он довольно хорошо инкапсулирован. В случае службы сообщений все методы, доступные для взаимодействия, должны удовлетворять следующему соглашению:

```
public void methodName(SOAPEnvelope env, SOAPContext req, SOAPContext res)
    throws java.io.IOException, javax.mail.MessagingException;
```

Здесь могут возникнуть ассоциации с сервлетами; вы получаете объекты запроса и ответа, с которыми взаимодействуете, а также и сам конверт SOAP для посылаемого сообщения. Исключение `IOException` мо-

жет быть сгенерировано, когда возникает ошибка, связанная с сетью, исключение MessagingException (из пакета JavaMail) возникает в случае проблем с конвертом сообщения SOAP. Кроме того, название метода должно совпадать с именем корневого элемента содержимого сообщения![1] Об этом легко забыть; в нашем случае это означает, что метод, получающий сообщение в формате XML, должен называться purchaseOrder, поскольку это имя корневого элемента из примера 13.3. Зная об этом, можно создать каркас для службы сообщений. Этот каркас приведен в примере 13.4; помимо создания системы для получения сообщений SOAP, он также включает логику для выполнения соответствующего вызова, адресованного службе CDCatalog на другой машине. Вместо кода для обработки сообщений, который мы вскоре рассмотрим, в примере присутствуют соответствующие комментарии.

Пример 13.4. Каркас службы сообщений CDs-R-Us

```
package javaxml2;

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.List;
import java.util.Vector;
import javax.mail.MessagingException;

// Импортируем SOAP
import org.apache.soap.Constants;
import org.apache.soap.Envelope;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.rpc.SOAPContext;
import org.apache.soap.util.xml.QName;

public class OrderProcessor {

    /** Соответствия для класса CD */
    private SOAPMappingRegistry registry;

    /** Сериализатор для класса CD */
    private BeanSerializer serializer;

    /** Объект RPC Call */
    private Call call;
```

[1] Это требование реализации Apache SOAP, а не самой спецификации SOAP. Однако это хорошая привычка и почти что стандарт, так что я бы стал привыкать к этому.

```
/** Параметры вызова */
private Vector params;

/** Ответ вызова RPC */
private Response rpcResponse;

/** URL для соединения */
private URL rpcServerURL;

public void initialize() {
    // Установка внутреннего URL для SOAP-RPC
    try {
        rpcServerURL =
            new URL("http://localhost:8080/soap/servlet/rpcrouter");
    } catch (MalformedURLException neverHappens) {
        // игнорируем
    }

    // Устанавливаем соответствие SOAP для трансляции объектов CD
    registry = new SOAPMappingRegistry();
    serializer = new BeanSerializer();
    registry.mapTypes(Constants.NS_URI_SOAP_ENC,
        new QName("urn:cd-catalog-demo", "cd"),
        CD.class, serializer, serializer);

    // Создаем вызов, адресованный внутренней службе SOAP
    call = new Call();
    call.setSOAPMappingRegistry(registry);
    call.setTargetObjectURI("urn:cd-catalog");
    call.setMethodName("getCD");
    call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

    // Исходные данные
    params = new Vector();
}

public void purchaseOrder(Envelope env, SOAPContext req, SOAPContext res)
    throws IOException, MessagingException {

    // Инициализация среды SOAP
    initialize();

    // Создаем список успешно заказанных компакт-дисков
    List orderedCDs = new LinkedList();

    // Создаем хеш-таблицу неудавшихся заказов
    Hashtable failedCDs = new Hashtable();

    // Анализируем входящее сообщение и получаем список заказанных
    // компакт-дисков

    // Перебираем в цикле все заказанные компакт-диски из формы заказа
        String artist = "";
        String title = "";

        // Исходные данные
        params.clear();
    params.addElement(new Parameter("title", String.class, title, null));
```

```
            call.setParams(params);

            try {
                // Выполняем вызов
                rpcResponse = call.invoke(rpcServerURL, "");

                if (!rpcResponse.generatedFault()) {
                    Parameter returnValue = rpcResponse.getReturnValue();
                    CD cd = (CD)returnValue.getValue();

                    // Проверяем, доступен ли компакт-диск
                    if (cd == null) {
                     failedCDs.put(title, "Запрашиваемый диск отсутствует.");
                      continue;
                    }

                    // Проверяем, верный ли исполнитель
                    if (cd.getArtist().equalsIgnoreCase(artist)) {
                        // Добавляем компакт-диск к списку удачных заказов
                        orderedCDs.add(cd);
                    } else {
                        // Добавляем к списку неудачных заказов
                       failedCDs.put(title, "  Исполнитель не соответствует
                                     указанному диску.");
                    }
                } else {
                    Fault fault = rpcResponse.getFault();
                    failedCDs.put(title, fault.getFaultString());
                }
            } catch (SOAPException e) {
                failedCDs.put(title, "SOAP Exception: " + e.getMessage());
            }

        // В конце возвращаем что-либо, имеющее смысл для клиента
    }
}
```

Примечание

В этом и последующих примерах данной главы используется имя узла *http://localhost:8080* для представления службы SOAP, запущенной на локальной машине. Большинство из вас будут тестировать пример локально, и это должно помочь избежать подстановки фиктивных имен узлов и неминуемого в этом случае недоумения, возникшего из-за того, что ничего не будет работать.

Предполагается, что в реальном приложении клиент будет соединяться с машиной CDs-R-Us, имеющей адрес *http://www.cds-r-us.com*, а служба сообщений будет соединяться с внутренней машиной, на которой запущен каталог компакт-дисков – *http://catalog.cds-r-us.com* – и которая, возможно, расположена за внешним брандмауэром. Но лучше, чтобы код заработал уже сейчас, поэтому я не буду подставлять в код примеров фиктивные имена узлов. Вот почему везде в качестве узла используется локальная машина.

Мы быстро рассмотрим, что тут происходит, а затем перейдем к интересному вопросу: взаимодействию посредством сообщений. Во-первых, инициализация RPC-вызова для каждого клиента выполняется посредством метода `initialize()`. Объект `Call` используется повторно, что позволяет экономить ресурсы. В то же время, каждый клиент получает собственный объект `Call`, что гарантирует отсутствие проблем, сопутствующих синхронизации и многопоточности. Кроме того, создается некоторое хранилище: список успешных заказов и хеш-таблица не прошедших. В качестве ключей хеш-таблицы используются названия заказанных компакт-дисков, а в качестве значений – информация об ошибках. Происходит чтение SOAP-сообщения от клиента (здесь пока оставлен комментарий). Заказанные диски перебираются в цикле. Из сообщения выделяются название компакт-диска и исполнитель, после чего вызывается процедура RPC для получения запрошенного объекта CD. В зависимости от результата запроса по каталогу, компакт-диск добавляется к списку либо успешных, либо неудавшихся заказов. В конце цикла будет создано сообщение, отправляемое клиенту.

Стоит заметить, что в данном контексте используется простая, неполная версия `CDCatalog`. Настоящая служба каталога компакт-дисков, вероятно, должна проверять наличие компакт-диска, удалять один компакт-диск из описи товаров, сообщать его SKU (Stock/Storage Keeping Unit) и т. д. В данном случае служба каталога лишь проверяет наличие компакт-диска в списке доступных CD. В любом случае, вы поняли смысл.

Теперь у нас есть каркас, и можно переходить к работе с сообщением пользователя. Позаботимся о некоторых дополнительных классах, которые нам понадобятся. Добавьте следующие операторы импортирования:

```
import java.io.IOException;
import java.io.StringWriter;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Vector;
import javax.mail.MessagingException;

// DOM
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
```

```
// Импортируем SOAP
import org.apache.soap.Constants;
import org.apache.soap.Envelope;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.rpc.SOAPContext;
import org.apache.soap.util.xml.QName;
```

Необходимо воспользоваться DOM, чтобы работать с XML-сообщением, отправленным клиентом. Именно на это сообщение мы и посмотрим первым делом. В примере 13.3 приведен код XML, представляющий собой содержимое сообщения, ожидаемого службой. Но перед отправкой сообщение подвергается «упаковке» в соответствии с правилами SOAP, и будет в итоге выглядеть так, как показано в примере 13.5. Дополнительная информация используется в SOAP для интерпретации сообщения.

Пример 13.5. Документ из примера 13.3 в формате SOAP

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <purchaseOrder orderDate="07.23.2001"
      xmlns="urn:cd-order-service"
  >
    <recipient country="USA">
      <name>Dennis Scannell</name>
      <street>175 Perry Lea Side Road</street>
      <city>Waterbury</city>
      <state>VT</state>
      <postalCode>05676</postalCode>
    </recipient>
    <order>
      <cd artist="Brooks Williams" title="Little Lion" />
      <cd artist="David Wilcox" title="What You Whispered" />
    </order>
  </purchaseOrder>
 </s:Body>
</s:Envelope>
```

Само сообщение заключено в тело конверта SOAP. Отображениями для этих структур в Apache SOAP являются классы `org.apache.soap.Envelope` и `org.apache.soap.Body`. Для получения записей из тела сообщения воспользуйтесь методом `envelope.getBody().getBodyEntries()`, возвращающим значение типа `Vector`. Первый (и единственный) элемент этого вектора в данном примере оказывается элементом DOM `Element` и является Java-эквивалентом XML-элемента `purchaseOrder`. Это именно

то, что нам нужно. Получив этот элемент, мы можем воспользоваться обычными методами DOM для обхода дерева и выявления всех заказанных CD. Добавьте следующий код в метод purchaseOrder(), который выявляет и обходит в цикле все заказанные клиентом компакт-диски:

```
public void purchaseOrder(Envelope env, SOAPContext req, SOAPContext res)
    throws IOException, MessagingException {

    // Инициализация среды SOAP
    initialize();

    // Создаем список успешно заказанных компакт-дисков
    List orderedCDs = new LinkedList();

    // Создаем хеш-таблицу неудавшихся заказов
    Hashtable failedCDs = new Hashtable();

    // Получаем объект purchaseOrder - всегда первый элемент в теле
       сообщения
    Vector bodyEntries = env.getBody().getBodyEntries();
    Element purchaseOrder = (Element)bodyEntries.iterator().next();

    // В настоящем приложении выполняем некие действия с информацией
       покупателя

    // Получаем заказанные CD
    Element order =
        (Element)purchaseOrder.getElementsByTagName("order").item(0);
    NodeList cds = order.getElementsByTagName("cd");

    // Перебираем в цикле все компакт-диски из формы заказа
    for (int i=0, len=cds.getLength(); i<len; i++) {
        Element cdElement = (Element)cds.item(i);
        String artist = cdElement.getAttribute("artist");
        String title = cdElement.getAttribute("title");

    // Исходные данные для процедуры SOAP-RPC, показанной в примере 13.4
    params.clear();
    params.addElement(new Parameter("title", String.class, title, null));
        call.setParams(params);

        try {
            // Существующий код RPC из примера 13.4
        } catch (SOAPException e) {
            failedCDs.put(title, "SOAP Exception: " + e.getMessage());
        }
    }

    // В конце возвращаем что-либо, имеющее смысл для клиента
}
```

После выполнения этого кода список orderedCDs будет содержать успешно заказанные компакт-диски, а хеш-таблица failedCDs — неудавшиеся заказы. Поскольку клиент уже «разговаривает» на XML (он от-

правлял сообщение в формате XML), то имеет смысл вернуть ответ в том же формате. Вместо того чтобы создавать ответ с нуля, форматировать его для SOAP и отвечать вручную, можно воспользоваться объектом Envelope, из которого только что производилось чтение. Добавьте приведенный ниже код, генерирующий ответ:

```
public void purchaseOrder(Envelope env, SOAPContext req, SOAPContext res)
    throws IOException, MessagingException {

    // Существующий код для анализа сообщений показан выше

    // Перебираем в цикле все компакт-диски из формы заказа
    for (int i=0, len=cds.getLength(); i<len; i++) {
        Element cdElement = (Element)cds.item(i);
        String artist = cdElement.getAttribute("artist");
        String title = cdElement.getAttribute("title");

    // Исходные данные
    params.clear();
    params.addElement(new Parameter("title", String.class, title, null));
        call.setParams(params);

        try {
            // Существующий код RPC из примера 13.4
        } catch (SOAPException e) {
            failedCDs.put(title, "SOAP Exception: " + e.getMessage());
        }
    }

    // В конце возвращаем что-либо, имеющее смысл для клиента
    Document doc = new org.apache.xerces.dom.DocumentImpl();
    Element response = doc.createElement("response");
    Element orderedCDsElement = doc.createElement("orderedCDs");
    Element failedCDsElement = doc.createElement("failedCDs");
    response.appendChild(orderedCDsElement);
    response.appendChild(failedCDsElement);

    // Добавляем заказанные CD
    for (Iterator i = orderedCDs.iterator(); i.hasNext(); ) {
        CD orderedCD = (CD)i.next();
        Element cdElement = doc.createElement("orderedCD");
        cdElement.setAttribute("title", orderedCD.getTitle());
        cdElement.setAttribute("artist", orderedCD.getArtist());
        cdElement.setAttribute("label", orderedCD.getLabel());
        orderedCDsElement.appendChild(cdElement);
    }

    // Добавляем CD, заказать которые не удалось
    Enumeration keys = failedCDs.keys();
    while (keys.hasMoreElements()) {
        String title = (String)keys.nextElement();
        String error = (String)failedCDs.get(title);
        Element failedElement = doc.createElement("failedCD");
```

```
            failedElement.setAttribute("title", title);
            failedElement.appendChild(doc.createTextNode(error));
            failedCDsElement.appendChild(failedElement);
        }

        // Помещаем письмо в конверт
        bodyEntries.clear();
        bodyEntries.add(response);
        StringWriter writer = new StringWriter();
        env.marshall(writer, null);

        // Возвращаем конверт клиенту
        res.setRootPart(writer.toString(), "text/xml");
    }
```

Приведенный код создает новое дерево XML, включающее информацию по успешным и не прошедшим заказам. Дерево включается в качестве содержимого тела конверта, заменяя первоначальный запрос клиента. Затем необходимо преобразовать конверт в текстовый формат, который можно отправить в качестве ответа при помощи объекта `SOAPContext res`. В SOAP для этого существует метод `marshall()`. Передача ему объекта `StringWriter` позволяет выделить из последнего значение типа `String` для последующего использования. Второй аргумент метода – экземпляр `org.apache.soap.util.xml.XMLJavaMappingRegistry`. Примером является класс `SOAPMappingRegistry` – подкласс `XMLJavaMappingRegistry`, с которым мы имели дело в предыдущей главе; раз нет необходимости в работе с соответствиями для пользовательских типов, то достаточно аргумента `null`.

Наконец, результат всех этих действий и сериализации используется в качестве содержимого ответа. Обратите внимание на метод `setRootPart()`. Второй аргумент этого метода – это медиа-тип содержимого. Поскольку речь идет о коде XML, верным значением будет «text/xml». Когда клиент получает ответ, он уже знает, что ответ представлен в формате XML, и может декодировать его. Фактически, единственное, что нужно сделать для инициации взаимодействия с клиентом – связать созданное содержимое с объектом ответа `SOAPContext`. Когда метод отработает, сервер SOAP автоматически вернет этот объект клиенту вместе с любой информацией, которая в него будет помещена. Тем, кто знаком с сервлетами, это напомнит работу с объектом `HttpServletResponse`.

Сейчас можно скомпилировать класс `OrderProcessor` и разместить его на сервере SOAP:

```
java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter deploy xml/
OrderProcessorDD.xml
```

После этого служба готова к регистрации в реестре UDDI.

Регистрация в UDDI

Прежде чем начать процесс регистрации службы, убедитесь, что она всем доступна. Невозможно зарегистрировать службу, доступную лишь на локальной машине (скажем, по адресу *http://localhost:8080*). Тестируя службу или экспериментируя с ней, прочитайте этот раздел и запомните его на будущее. А уже подготовившись к тому, чтобы действительно зарегистрировать службу где-то в сети, убедитесь, что знаете имя узла, на котором она будет доступна. После этого посетите сайт *http://www.uddi.org*.

Попав на веб-сайт UDDI, перейдите по ссылке «Register» в верхней правой части экрана (см. рис. 13.2). Затем выберите систему, в которой желаете зарегистрировать службу. В настоящее время регистрация службы в одной системе автоматически делает ее доступной через все системы, так что это решение здесь роли не играет. Я выбрал сайт IBM и нажал кнопку Go. Тут вам понадобится учетная запись для доступа к реестру IBM. Если таковой у вас нет, то ее можно получить бесплатно; нажмите кнопку Register, находящуюся слева, и следуйте приводимым инструкциям. Зарегистрировавшись, перейдите по ссылке «Login», расположенной слева (рис. 13.4). При первом входе в систему потребуется предоставить ключ активации, присланный по электронной почте после регистрации.

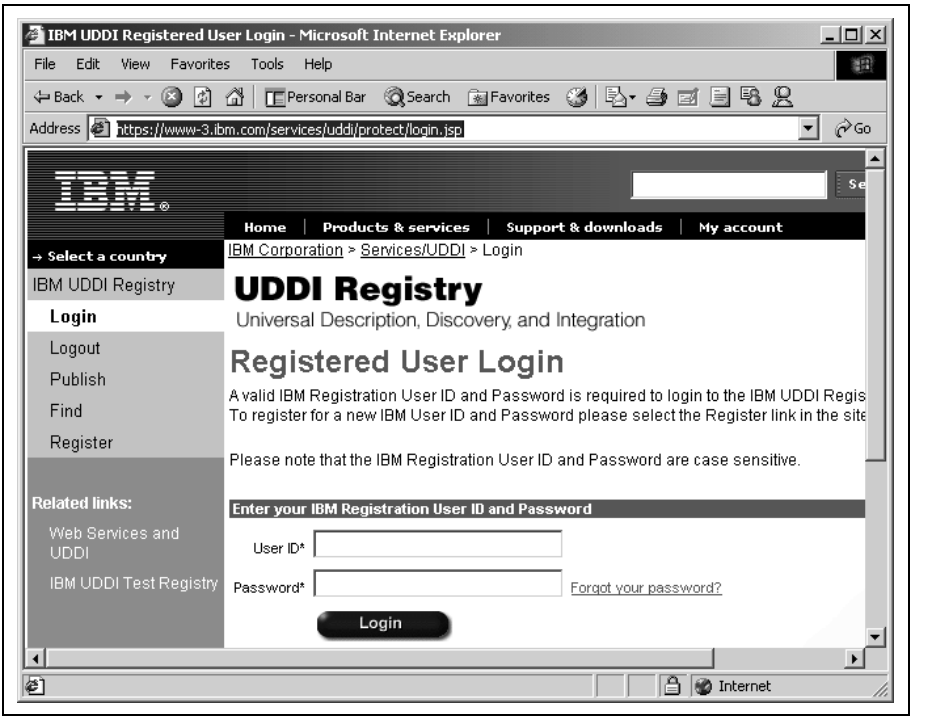

Рис. 13.4. Вход в реестр IBM UDDI

После того как регистрация выполнена и учетная запись активирована, можно публиковать службу. Сначала выберите, хотите ли вы добавить новое предприятие (business); обычно это целесообразно. Я добавил новое предприятие, как показано на рис. 13.5.

Рис. 13.5. Добавление нового предприятия в реестр UDDI

Затем можно добавлять службы и связывать их со своим предприятием, обеспечивая дополнительный уровень абстракции для поиска. Можно добавлять деловые контакты, местоположения и прочее. Покончив с этим, добавьте службу к реестру.

Выберите опцию Add a new service и введите название службы. В нашем примере это будет название `cd-order-service`. Вы получите возможность ввести описание, протокол доступа (access point) и адрес службы. Я ввел следующее описание: «Эта служба позволяет заказывать компакт-диски при помощи формы заказа». В качестве протокола доступа я выбрал «http», а затем указал «newInstance.com/soap/servlet/rpcrouter» в качестве адреса. Сделайте то же самое для вашей службы, подставляя собственное имя узла и URL. Затем можно задать локатор службы, являющийся формальным набором стандартов для определения категории службы; здесь это подробно не рассматривается, но обо всем можно прочитать на веб-сайте. Закончив вводить информацию, вы должны получить нечто подобное тому, что показано на рис. 13.6.

С этого момента процесс становится немного менее эффективным. К сожалению, не существует возможности загрузить (upload) WSDL-

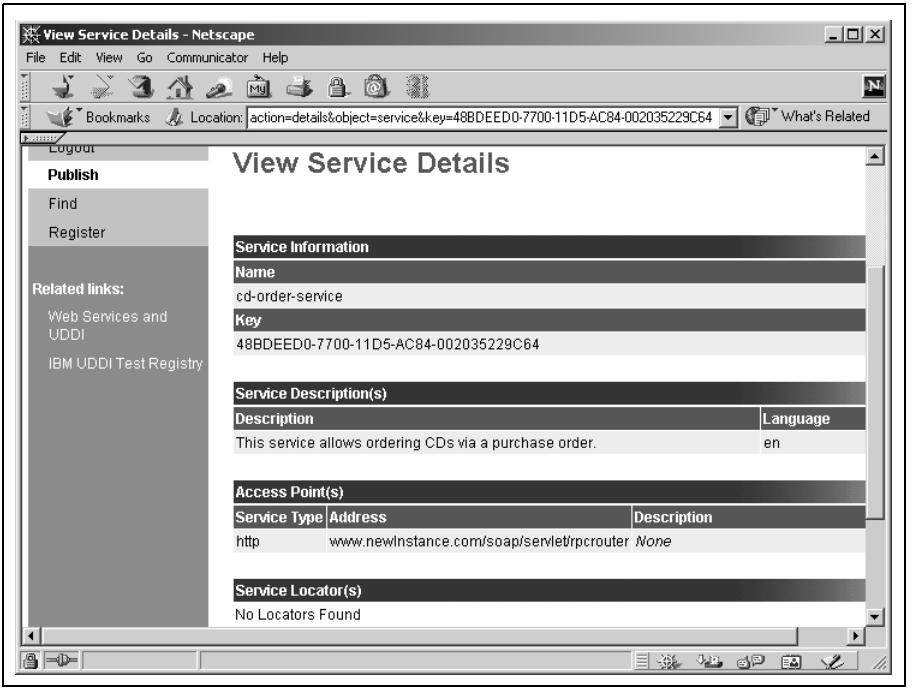

Рис. 13.6. Результат добавления службы

документ, описывающий вашу службу; это позволило бы сделать доступным для потенциальных клиентов и техническое описание. Единственная доступная информация – это имя службы и адрес доступа (или адреса, если служба доступна в нескольких местах). Однако это уже позволяет искать службу любому, кто зарегистрирован в UDDI и имеет учетную запись. Итак, служба зарегистрирована, и теперь найти ее может каждый.

Поиск в реестре UDDI

Обратная сторона монеты – это поиск службы. Теперь из поставщика службы я превращаюсь в потребителя. Тем, кто хочет использовать SOAP, WSDL и все прочее, необходимо посетить реестр на веб-сайте UDDI и зарегистрироваться, после чего будет возможен поиск служб (подобных той, которую вы только что зарегистрировали). Это очень просто: перейдите по ссылке «Find», выберите узел, на котором будете искать службы (опять же, я выбрал «IBM»), и введите имя службы. Вам потребуется зарегистрироваться (если это не было сделано раньше).

Поиск – еще одна область, в которой эволюция веб-служб еще далека от завершения. Доступен лишь примитивный поиск по имени службы.

Если служба названа «Reading Library Service», а при поиске введено слово «book», то служба никогда не будет найдена. Необходимо ввести либо «reading», либо «library». Но начало все же положено. После регистрации нашей службы мы можем ввести в поле поиска «cd». Укажите, что вы ищете службы, а не предприятия или типы служб. Затем нажмите кнопку Find в форме поиска. Будут получены результаты, подобные приведенным на рис. 13.7 (они содержат добавленную мной службу CD и могут содержать службы других читателей, когда эта книга выйдет).

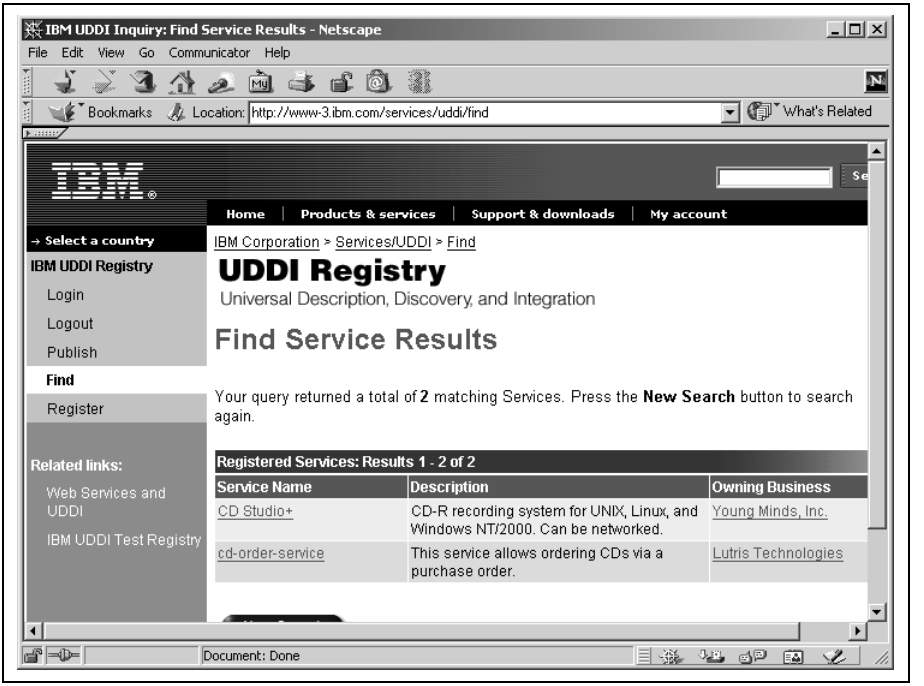

Рис. 13.7. Результаты поиска для «cd»

Можно выбрать имя службы и получить адрес доступа, определенный ранее, а также и любую другую заданную информацию (например, выбран ли тип или категория для службы). Логично было бы предположить, что вам удастся загрузить WSDL из описания этой службы, но в настоящее время это невозможно. Необходимо соединиться с предприятием, предоставляющим службу, определить доступные методы, которые будут применяться, и выяснить, как следует оплачивать использование службы.

Над реестрами UDDI еще нужно поработать. Однако уже создана мощная инфраструктура для регистрации и поиска веб-служб. После того как станет возможным загрузка на сервер (для поставщиков служб) и копирование с сервера (для потребителей служб) WSDL-документов,

применение служб уже не будет требовать личного взаимодействия. Несмотря на то что в результате сократятся контакты между людьми и усилится контакт с жидкокристаллическими мониторами, больше служб найдут своих клиентов, что будет способствовать развитию этой сферы.

WSDL

Теперь поговорим о пользе WSDL-документов. Я расскажу, как обработать такой документ простой утилитой, вроде упомянутой ранее IBM WSTK, в целях создания клиента на языке Java. Мне хотелось бы прямо сейчас рассказать, насколько полезен WSDL! Но пока что это не совсем так, и я вместо этого расскажу, что происходит с Java и WSDL, чтобы подготовить читателей к развитию событий.

Будьте готовы к появлению десятков инструментов, позволяющих создавать WSDL-документы для служб Java, таких как классы `OrderProcessor` и `CDCatalog`. На самом деле некоторые из этих инструментов уже доступны. Мы уже говорили об инструменте IBM WSTK, но также существуют и другие пакеты: от The Mind Electric (Glue – *http://www.themindelectric.com*), Silverstream (*http://www.silverstream.com*) и SOAPWiz (*http://www.bju.edu/cps/faculty/sschaub/soapwiz/*). Я пробовал применять их – с переменным успехом. В простых случаях, таких как `CDCatalog`, эти инструменты, как правило, легко генерировали документы WSDL (хотя набор инструментов от IBM «ломался» на методе, возвращающем хеш-таблицу). Дело в том, что методы ожидали получить в качестве ввода и вернуть в качестве вывода довольно примитивные типы Java, такие как `String` и `CD`, состоящий из примитивов.

Неприятности начались тогда, когда я попытался воспользоваться этими инструментами для класса `OrderProcessor`. Из-за того что последний основан на сообщениях, а не на RPC, он принимает в качестве ввода некоторые сложные типы: `Envelope` и `SOAPContext`. Поскольку эти сложные типы в свою очередь сами состоят из сложных типов, генераторы WSDL очень быстро путаются и обычно все заканчивается выводом распечатки стека. Над этими инструментами по-прежнему нужно потрудиться, чтобы они могли работать со службами SOAP, основанными на сообщениях, либо с чрезвычайно сложными службами, основанными на RPC.

Конечный итог двоякий: во-первых, он должен заинтриговать и заинтересовать вас. Когда появятся инструменты, которые могут работать с этими более сложными типами, станет проще генерировать WSDL-документы даже для сложных служб SOAP. После чего те же инструменты приобретут способность генерировать код клиентов, обращающихся к службам. Представьте себе поиск службы в реестре UDDI, загрузку описания WSDL и применение такого инструмента для создания клиента, общающегося со службой. Лишь незначительно изменив

код под свои нужды, вы сразу сможете работать с новой службой. Веб-службам (а также книгам, посвященным этой теме, я думаю!) обеспечено блестящее будущее.

Второй результат заключается в том, что по-прежнему приходится брать телефон и разговаривать (хотя иногда и недолго) с людьми об их службах. Когда кто-то опишет сигнатуру метода, с которым можно взаимодействовать, или отправит ее по электронной почте (мы, программисты, не очень-то склонны к личному общению), вы сможете написать код клиента, и я расскажу, как это сделать.

Пишем код клиента

Найдя службу, набор методов и сообщения, которые можно посылать, мы готовы к созданию клиента. В примере 13.6 приведен код клиентского приложения, с которым можно работать (предварительно его скомпилировав).

Пример 13.6. Клиент CDOrder

```
package javaxml2;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URL;
import javax.xml.parsers.DocumentBuilder;

// Импортируем SAX и DOM
import org.w3c.dom.Document;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;

// Импортируем SOAP
import org.apache.soap.Constants;
import org.apache.soap.Envelope;
import org.apache.soap.SOAPException;
import org.apache.soap.messaging.Message;
import org.apache.soap.transport.SOAPTransport;
import org.apache.soap.util.xml.XMLParserUtils;

public class CDOrderer {

    public void order(URL serviceURL, String msgFilename)
        throws IOException, SAXException, SOAPException {

        // Анализируем XML-сообщение
        FileReader reader = new FileReader(msgFilename);
        DocumentBuilder builder = XMLParserUtils.getXMLDocBuilder();
        Document doc = builder.parse(new InputSource(reader));
        if (doc == null) {
            throw new SOAPException(Constants.FAULT_CODE_CLIENT,
                "Ошибка при анализе XML-сообщения.");
```

```
            }

            // Создаем конверт сообщения
            Envelope msgEnvelope = Envelope.unmarshall(doc.getDocumentElement());

            // Отправляем сообщение
            Message msg = new Message();
            msg.send(serviceURL, "urn:cd-order-service", msgEnvelope);

            // Обрабатываем ответ
            SOAPTransport transport = msg.getSOAPTransport();
            BufferedReader resReader = transport.receive();

            String line;
            while ((line = resReader.readLine()) != null) {
                System.out.println(line);
            }
        }

        public static void main(String[] args) {
            if (args.length != 1) {
                System.out.println("Использование: java javaxml2.CDOrderer " +
                    "[Имя файла с XML-сообщением]");
                return;
            }

            try {
                URL serviceURL =
                    new URL("http://localhost:8080/soap/servlet/messagerouter");

                CDOrderer orderer = new CDOrderer();
                orderer.order(serviceURL, args[0]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Это очень простой, тривиальный клиент. Он считывает переданное в командной строке XML-сообщение и преобразует его в SOAP `Envelope`. Объект `org.apache.soap.messaging.Message` инкапсулирует конверт и посылает его по заданному URN и имени узла службы. Клиент получает ответ через `SOAPTransport` из сообщения `Message`, а полученное сообщение просто выводится на экран (его можно выводить в другой XML-файл либо извлекать и подвергать воздействию DOM или JDOM для обработки, а затем продолжать обработку).

Вместо того чтобы непосредственно создавать экземпляры класса `org.apache.xerces.dom.DocumentImpl`, я воспользовался классом JAXP `DocumentBuilder` и классом SOAP `XMLUtils`, чтобы избавиться от кода, привязанного к платформе. Это хорошая привычка, и такой подход правильнее, чем реализованный в классе `OrderProcessor` (когда мы ссылались непосредственно на класс Xerces). Оба подхода приведены только

для иллюстрации различий. Измените код класса `OrderProcessor` в соответствии с приведенным тут кодом клиента.

Убедившись, что все необходимые классы клиентов SOAP находятся в путях к классам, и скомпилировав класс `CDOrderer`, мы готовы приступить к тестированию. Убедитесь, что установлены и доступны службы `urn:cd-order-service` и `urn:cd-catalog`. Кроме того, можно добавить один или два компакт-диска в документ *po.xml* из примеров 13.2 и 13.3. Я сначала добавил в каталог один из них, чтобы проверить работу успешного и неудавшегося заказа, а затем добавил оба, чтобы убедиться, что все работает и для пары компакт-дисков:

```
C:\javaxml2\build>java javaxml2.CDAdder
    http://localhost:8080/soap/servlet/rpcrouter
    "Little Lion" "Brooks Williams" "Signature Sounds"
Добавление компакт-диска 'Little Lion' исполнитель 'Brooks Williams',
                         on the label
    Signature Sounds
Успешное добавление компакт-диска.
```

Убедитесь, что вы сохранили XML-документ из примера 13.5, подходящий для SOAP. Я сохранил его под именем *poMsg.xml* в каталоге *xml*. Наконец, можно запустить клиентское приложение:

```
bmclaugh@GANDALF
$ java javaxml2.CDOrderer c:\javaxml2\ch13\xml\poMsg.xml
<?xml version='1.0' encoding='UTF-8'?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
<s:Body>
<response>
 <orderedCDs>
  <orderedCD artist="Brooks Williams" label="Signature Sounds"
             title="Little Lion"/>
 </orderedCDs>
 <failedCDs>
  <failedCD title="What You Whispered">Запрошенный компакт-диск не
доступен.</failedCD>
 </failedCDs>
</response>
</s:Body>
</s:Envelope>
```

Программа выдает XML-ответ от сервера. Я отформатировал его, чтобы он был более удобочитаемым. Вы должны получить подобный вывод при тестировании (в этом примере я добавил в каталог компакт-диск Brooks Williams и не добавил David Wilcox).

Сейчас вы уже должны уверенно создавать как серверы, так и клиентские приложения SOAP. Кроме того, вероятно, уже стало понятно, что UDDI и WSDL не так уж сложны. В случае применения с SOAP они обеспечивают удобную систему для веб-служб и взаимодействия при-

ложений. Рекомендую также рассмотреть некоторые из расширенных свойств Apache SOAP или той реализации SOAP, с которой работаете вы. Например, Apache SOAP поддерживает протокол SMTP (Простой протокол передачи почты, Simple Mail Transport Protocol) в качестве транспорта для RPC и передачи сообщений. Здесь мы не будем вдаваться в подробности, поскольку это более сложная тема, а спецификация SOAP пока что не предусматривает применение SMTP в качестве транспорта. Другими словами, такая возможность зависит от реализации, а я стараюсь избегать зависимости от платформы. Но как бы то ни было, владение особенностями используемой реализации SOAP лишь повысит эффективность ваших веб-служб.

Куда идти?

Те, кто похож на меня, вероятно, готовы прочитать еще три-четыре главы, посвященные этой теме. Одну главу о работе с реестрами UDDI из приложений, еще одну главу о работе с WSDL, еще несколько примеров... это было бы здорово. Конечно, тогда это была бы книга о веб-службах, а не о Java и XML. Однако в Интернете есть ресурсы, которые помогут перейти на следующий уровень. Во-первых, стоит обратиться к сайту *http://www.webservices.org*, на котором собрано немало дополнительных вводных материалов. Затем стоит посетить сайт IBM, посвященный этому вопросу, *http://www.ibm.com/developerworks/webservices*. Тем, кто работает с клиентами от Microsoft (C#, Visual Basic и объектами COM), стоит посетить сайт *http://msdn.microsoft.com/soap*. Другими словами, воспользуйтесь этой главой как надежной точкой опоры, посетите упомянутые тут сайты и ждите новых книг от O'Reilly, посвященных SOAP.

Что дальше?

Понимая, как работают веб-службы, вы готовы завоевывать мир, верно? Не совсем так. Как и во всем остальном в Java, существует столько же способов осуществления межкорпоративного взаимодействия и взаимодействия, основанного на службах, сколько существует людей, реализующих это взаимодействие. Не думаю, что последняя «идея фикс» (SOAP, UDDI и WSDL), – это конечное, единственное решение, поэтому в следующей главе я представил совершенно иной способ решения вопросов взаимодействия – при помощи различных языков, сервлетов и RSS (Rich Site Summary). Вам предстоит ощутимо пополнить багаж знаний.

14

- *Библиотека Foobar*
- *Компания mytechbooks.com*
- *Рассылка или запрос*
- *Что дальше?*

Объединение содержимого

В двух предшествующих главах я попытался представить точку зрения веб-служб. Другими словами, вы увидели, как создавать приложения, взаимодействие которых организовано при помощи различных технологий веб-служб, таких как WSDL, UDDI и SOAP. Но, как вы могли понять, некоторые из рассмотренных технологий по-прежнему не очень стабильны, например, автоматическое создание и поддержка WSDL-документов (при условии, что вы пользуетесь открытыми стандартами, такими как SOAP). В настоящее время существуют и другие варианты организации межкорпоративного взаимодействия. В этой главе представлено альтернативное решение, которое дополнит ваши знания.

Здесь рассмотрено использование различных спецификаций XML для обеспечения этого вида связи между приложениями и компаниями на примере фирм, придуманных как раз для этого. Начнем с Foobar Public Library – библиотеки, которая позволяет своим поставщикам вводить в режиме реального времени информацию о новых поступлениях. Информация помещается в хранилище данных библиотеки для дальнейшего пользования. К сожалению, библиотеке трудно найти хороших разработчиков на Java, поэтому она реализована в виде CGI-приложения, написанного на языке Perl. Поступившая информация о новых книгах сохраняется с помощью сценария, написанного на Perl. Можно заметить, что альтернативы веб-службам могли бы быть нелишними, в то время как найти хорошую реализацию SOAP для Perl не так просто (по крайней мере, пока!).

Также мы рассмотрим другую компанию, *mytechbooks.com*. Фирма *mytechbooks.com* продает техническую и компьютерную литературу (такую, как эта книга) в режиме реального времени, пользуясь различными партнерскими связями с большими книжными магазинами. Недавно они подписали соглашение с Foobar Public Library о получении книг из библиотеки. Они будут оплачивать доставку и стоимость книг

по каталогу, в то время как библиотека соглашается заказывать дополнительные книги со скидками. Эти дополнительные книги затем продаются компанией *mytechbooks.com*. Фирме *mytechbooks.com* нужно иметь доступ к информации о новых книгах, поступающих в библиотеку Foobar от поставщиков, чтобы знать, когда будут доступны новые поступления и когда начинать их рекламировать. Однако они не знают, как взаимодействовать с написанной на Perl системой библиотеки Foobar. К тому же, защищенных сетевых каналов между двумя организациями не существует, поэтому для связи должен использоваться обычный протокол HTTP. И чтобы у нас не было никаких шансов на применение веб-служб: компания *mytechbooks.com* хочет подождать, пока веб-службы не наберут вес и не будет реализована более полная интеграция с WSDL, поэтому им нужно более надежное решение (по крайней мере, такое, которое применялось в течение более длительного времени).

Наконец, мы рассмотрим покупателей фирмы *mytechbooks.com*. Книжный магазин предназначен для людей, которые работают в сети, поэтому руководство магазина хочет размещать рекламу на сайтах в Интернете, подобных Netscape Netcenter. Руководство желает дать людям возможность легко получать информацию о новых поступлениях на сайте. Но как и в случае с библиотекой Foobar, фирма *mytechbooks.com* не знает, как достичь этой цели. Поскольку они читали книги от O'Reilly и статьи на сайте *http://www.oreillynet.com*, они слышали от активиста (spec lead) RSS Рэйля Дорнфеста (Rael Dornfest), что это отличная технология, и хотят ее опробовать. Разумеется, Рэйль прав, так что именно об этой технологии и пойдет разговор в этой главе.

Мы начинаем работу с этим распространенным сценарием с библиотеки Foobar и изучения ее информационной системы, написанной на Perl. Затем, переходя к фирме *mytechbooks.com* и далее к клиентам магазина, мы посмотрим, как ввести в действие приложение, созданное по модели business-to-business (и business-to-customer), применив XML в качестве средства взаимодействия между каждым из его уровней.

Библиотека Foobar

Приступим к созданию системы business-to-business, для чего сначала рассмотрим информационную систему, в настоящее время реализованную в библиотеке Foobar. Однако прежде чем углубляться в ее код, нужно исследовать требования библиотеки, чтобы не создавать систему, которую невозможно поддерживать.

Оценка требований

Зачастую хорошее решение проблемы не является подходящим решением для компании, у которой эта проблема имеется. Библиотека

Foobar – отличный тому пример: конечно, Java-сервлет, взаимодействующий с сервлетами, созданными компанией *mytechbooks.com*, мог бы быстро решить проблемы двух организаций. Но такое решение не учитывает требований библиотеки. Библиотека предъявляет к решению следующие требования:

- Решение должно быть основано на языке Perl – в штате нет специалистов по языку Java.
- Решение не должно включать в себя установку новых программных продуктов или библиотек.
- Решение не должно повлиять на существующую систему приема заказов (не должно быть изменений в интерфейсе).

Хотя это не слишком строгие требования, они заставляют обратиться к решению, не связанному с Java-сервлетами. Нужно избегать использования Java в качестве решения. Конечно, читая книгу об XML, вы, вероятно, склонны предполагать, что хранение данных о новых книгах в формате XML позволит библиотеке передавать эти XML-данные клиентам через запрос HTTP, а те будут использовать данные по своему разумению. По сути дела, это решение гораздо предпочтительнее взаимодействия между сервлетами, поскольку данные в формате XML могут использоваться любой компанией или клиентом в своих приложениях, а реализация, привязанная к какой-либо платформе, ограничит взаимодействие библиотекой и еще одной конкретной компанией. Эти соображения и определяют нашу цель при модернизации информационной системы библиотеки Foobar: сохранить введенную информацию в виде XML-данных, а затем обеспечить клиентам и покупателям доступ к этим данным по протоколу HTTP.

Ввод информации о книгах

Необходимо изучить существующий HTML-интерфейс, позволяющий поставщикам вводить информацию о новых книгах в систему. В примере 14.1 приведен статический HTML-код, применяемый для создания этой формы.

Пример 14.1. Статический HTML-код для интерфейса библиотеки Foobar

```
<html>

<head>
  <title>Foobar Public Library: Add Books</title>
  <style>
<!--
body         { font-family: Arial }
h1           { color: #000080 }
-->
  </style>
</head>
```

```
<body link="#FFFF00" vlink="#FFFF00" alink="#FFFF00">
 <table border="0" width="100%" cellpadding="0" cellspacing="0">
  <tr>
   <td width="15%" bgcolor="#000080" valign="top" align="center">
    <b><i>
     <font color="#FFFFFF" size="4">Options</font>
    </i></b>
   <p><b>
     <font color="#FFFFFF">
      <a href="/javaxml/foobar">Main Menu</a>
     </font>
   </p></b>
   <p><b>
    <font color="#FFFFFF">
     <a href="/javaxml/foobar/catalog.html">Catalog</a>
    </font>
   </b></p>
   <p><b>
    <i><font color="#FFFF00">Add Books</font></i>
   </b></p>
   <p><b>
    <font color="#FFFFFF">
     <a href="/javaxml/foobar/logout.html">Log Out</a>
    </font>
   </p></td>
   <td width="*" valign="top" align="center">
    <h1 align="center">The Foobar Public Library</h1>
    <h3 align="center"><i>- Add Books -</i></h3>

<!-- Здесь должна быть ссылка на каталог CGI и конкретный сценарий, который
мы рассмотрим позже -->
    <form method="POST" action="/cgi/addBook.pl">

     <table border="0" cellpadding="5" width="100%">
      <tr>
       <td width="100%" valign="top" align="center" colspan="2">
        Title 
        <input type="text" name="title" size="20">
        <hr width="85%" />
       </td>
      </tr>
      <tr>
       <td width="50%" valign="top" align="right">Author 
        <input type="text" name="author" size="20">
       </td>
       <td width="50%" valign="top" align="left">Subject 
        <select size="1" name="subject">
         <option>Fiction</option>
         <option>Biography</option>
         <option>Science</option>
         <option>Industry</option>
         <option>Computers</option>
```

```
          </select></td>
         </tr>
         <tr>
          <td width="50%" valign="top" align="right">Publisher 
           <input type="text" name="publisher" size="20">
          </td>
          <td width="50%" valign="top" align="left">ISBN 
           <input type="text" name="isbn" size="20">
          </td>
         </tr>
         <tr>
          <td width="50%" valign="top" align="right">Price 
           <input type="text" name="price" size="20">
          </td>
          <td width="50%" valign="top" align="left">Pages 
           <input type="text" name="numPages" size="20">
          </td>
         </tr>
         <tr>
          <td width="100%" valign="top" align="center" colspan="2">
           Description 
           <textarea rows="2" name="description" cols="20"></textarea>
          </td>
         </tr>
        </table>
        <p>
         <input type="submit" value="Add this Book" name="addBook">
         <input type="reset" value="Reset Form" name="reset">
         <input type="button" value="Cancel" name="cancel">
        </p>
      </form>
     </td>
    </tr>
   </table>
  </body>
  </html>
```

Этот файл, сохраненный под именем *addBooks.html*, реализует ту часть ПО библиотеки, которая позволяет поставщикам добавлять информацию о новых книгах, отправляемых ими в библиотеку.

Примечание

В примере 14.1 и в оставшейся части главы код и листинги HTML будут приводиться полностью, чтобы вы могли создать описанное приложение и пройти по пути внедрения взаимодействия приложений на основе XML. Кроме того, следует оставить имена файлов, указанные в тексте, без изменения; в противном случае потребуется соответствующим образом исправить код и примеры. Чтобы читателям проще было разобраться в примерах, код, который может подвергнуться изменениям (в части ссылок на файлы или сценарии), выделен в текстах программ.

При обращении к HTML-коду из примера 14.1 через веб-броузер будет получен результат, представленный на рис. 14.1. Хотя мы не рассматриваем другие пункты меню, поставщик может также просмотреть каталог библиотеки, перейти в главное меню приложения, а также выйти из приложения, используя меню в левой части экрана.

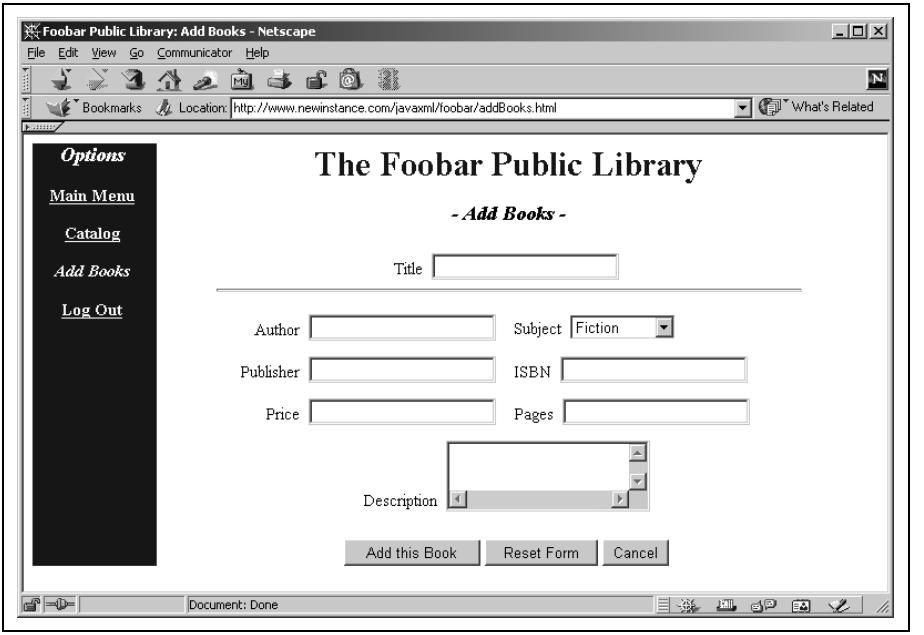

Рис. 14.1. Пользовательский HTML-интерфейс для библиотеки Foobar

Эта форма позволяет поставщику вводить подробности о каждой книге, которая передается в библиотеку. Поставщик заполняет выходные данные книги (название, автор, издательство, количество страниц и описание), а также тему (для классификации книги) и информацию по сбыту, включая цену и номер ISBN.

Введенная информация передается CGI-сценарию, написанному на Perl:

```
<form method="POST" action="/cgi/addBook.pl">
```

Сценарий должен сгенерировать результат в XML-формате. Самым простым решением была бы загрузка библиотеки Perl, выполняющей анализ XML, например Xerces-Perl. Однако помните, что одним из требований был запрет на добавление каких бы то ни было библиотек или программных продуктов. Хотя это может показаться неразумным и необоснованным, имейте в виду, что во многих компаниях на применяемые системы налагаются очень серьезные ограничения. В данном случае библиотека Foobar только начинает внедрять приложения в Интернете, и у нее нет ресурсов для поддержки дополнительного программного обеспечения.

К счастью, нужно лишь выдавать результат в формате XML. Задача элементарно решается прямолинейным созданием файла с данными о книгах. Все было бы гораздо сложнее, если бы потребовалось анализировать данные в формате XML. Поскольку библиотеке требуется организовать хранение информации обо всех существующих книгах, каждая новая запись добавляется в уже существующий файл. Написание программы на Perl практически тривиально, а полный код программы для чтения параметров запроса и добавления данных в существующий файл приведен в примере 14.2.

Пример 14.2. CGI-сценарий на Perl для создания учетных записей поступлений в формате XML

```
#!/usr/local/bin/perl

# Каталог, в который вы хотите записывать файлы
$baseDir = "/home/bmclaugh/javaxml/foobar/books/";

# Имя используемого файла
$filename = "books.txt";

$bookFile = $baseDir . $filename;

# Получаем ввод от пользователя
use CGI;
$query = new CGI;

$title = $query->param('title');
$author = $query->param('author');
$subject = $query->param('subject');
$publisher = $query->param('publisher');
$isbn = $query->param('isbn');
$price = $query->param('price');
$numPages = $query->param('numPages');
$description = $query->param('description');

# Сохраняем книгу в файле в формате XML
if (open(FILE, ">>" . $bookFile)) {
  print FILE "<book subject=\"" . $subject . "\">\n";
  print FILE " <title><![CDATA[" . $title . "]]></title>\n";
  print FILE " <author><![CDATA[" . $author . "]]></author>\n";
  print FILE " <publisher><![CDATA[" . $publisher . "]]></publisher>\n";
  print FILE " <numPages>" . $numPages . "</numPages>\n";
  print FILE " <saleDetails>\n";
  print FILE "  <isbn>" . $isbn . "</isbn>\n";
  print FILE "  <price>" . $price . "</price>\n";
  print FILE " </saleDetails>\n";
  print FILE " <description>";
  print FILE "<![CDATA[" . $description . "]]>";
  print FILE "</description>\n";
  print FILE "</book>\n\n";
```

```
  # Выводим подтверждение
  print <<"EOF";
Content-type: text/html

  <html>
   <head>
    <title>Foobar Public Library: Confirmation</title>
   </head>
   <body>
    <h1 align="center">Book Added</h1>
    <p align="center">
     Thank you.  The book you submitted has been added to the Library.
    </p>
   </body>
  </html>
EOF

} else {
  print <<"EOF";
Content-type: text/html

  <html>
   <head>
    <title>Foobar Public Library: Error</title>
   </head>
   <body>
    <h1 align="center">Error in Adding Book</h1>
    <p align="center">
     We're sorry.  The book you submitted has <i>not</i> been added to
     the Library.
    </p>
   </body>
  </html>
EOF
}
close (FILE);
```

Эта программа, сохраненная в файле *addBook.pl*, вызывается при отправке формы (после нажатия кнопки «Add this Book»), когда поставщик вводит информацию по новой книге. Сценарий определяет файл, в который осуществляется запись, а затем сохраняет параметры запроса в локальных переменных:

```
$title = $query->param('title');
$author = $query->param('author');
$subject = $query->param('subject');
$publisher = $query->param('publisher');
$isbn = $query->param('isbn');
$price = $query->param('price');
$numPages = $query->param('numPages');
$description = $query->param('description');
```

Теперь работа с этими значениями максимально упростилась, и сценарий открывает определенный ранее файл в режиме добавления данных (о чем говорят символы >> перед именем файла) и записывает информацию о введенной книге в XML-формате в конец файла:

```
print FILE "<book subject=\"" . $subject . "\">\n";
print FILE " <title><![CDATA[" . $title . "]]></title>\n";
print FILE " <author><![CDATA[" . $author . "]]></author>\n";
print FILE " <publisher><![CDATA[" . $publisher . "]]></publisher>\n";
print FILE " <numPages>" . $numPages . "</numPages>\n";
print FILE " <saleDetails>\n";
print FILE "  <isbn>" . $isbn . "</isbn>\n";
print FILE "  <price>" . $price . "</price>\n";
print FILE " </saleDetails>\n";
print FILE " <description>";
print FILE "<![CDATA[" . $description . "]]>";
print FILE "</description>\n";
print FILE "</book>\n\n";
```

Тема книги используется в качестве значения атрибута объемлющего элемента `book`, а остальная информация отражается в виде элементов. Поскольку название, автор и описание книги, а также название издательства могут содержать кавычки, апострофы, амперсанды и другие символы, которые требуют экранирования, мы помещаем эти данные в секцию CDATA, чтобы не заботиться о маскировке символов.

Кроме того, вы должны были заметить, что здесь не создаются ни объявление XML, ни корневой элемент, потому что в одном файле будет существовать несколько таких блоков. Поскольку требуется некоторый труд для того, чтобы проверить существование файла, записать объявление XML и корневой элемент, если файл новый, и затем записать закрывающий элемент (который должен перезаписываться для каждой новой записи), мы делаем этот файл фрагментом XML-документа. Вот, например, как мог бы выглядеть файл после добавления двух книг:

```
<book subject="Computers">
 <title><![CDATA[Java Servlet Programming]]></title>
 <author><![CDATA[Jason Hunter]]></author>
 <publisher><![CDATA[O'Reilly & Associates]]></publisher>
 <numPages>753</numPages>
 <saleDetails>
  <isbn>0596000405</isbn>
  <price>44.95</price>
 </saleDetails>
 <description><![CDATA[This book is a superb introduction to Java
  servlets and their various communications mechanisms.]]></description>
</book>

<book subject="Fiction">
```

```
  <title><![CDATA[Second Foundation]]></title>
  <author><![CDATA[Isaac Asimov]]></author>
  <publisher><![CDATA[Bantam Books]]></publisher>
  <numPages>279</numPages>
  <saleDetails>
   <isbn>0553293362</isbn>
   <price>5.59</price>
  </saleDetails>
  <description><![CDATA[fter the First Foundation was taken over by the
   Mule, only the Second Foundation stood between order and the utter
   destruction the Mule would bring.]]></description>
</book>
```

Хотя этот фрагмент не является полным XML-документом, он корректен (well-formed) и может быть вставлен в любой XML-документ, в котором уже имеются пролог и корневой элемент. В действительности, когда в следующем разделе мы будем рассматривать формирование списка книг, именно так и будет осуществляться обработка вывода данного фрагмента.

Оставшаяся часть сценария выдает HTML-код, указывающий, удалось ли добавить книгу в список или произошла ошибка. После добавления книги в XML-каталог поставщик получает простое подтверждение, показанное на рис. 14.2.

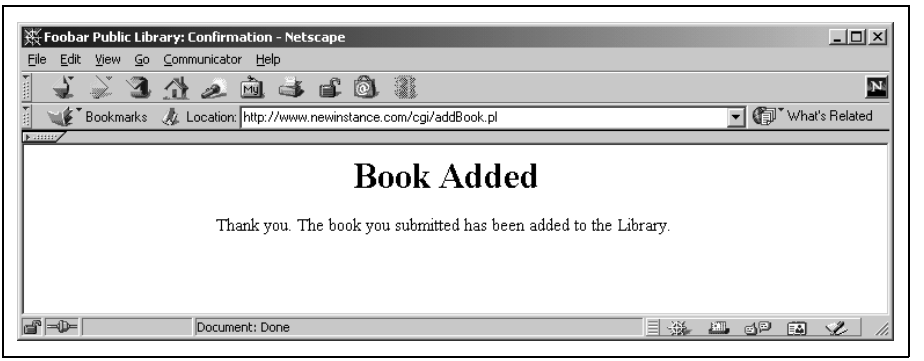

Рис. 14.2. Подтверждение о добавлении книги

Теперь, когда у нас имеется фрагмент XML-документа с данными о новых книгах, нужно взять этот файл и сделать его доступным для пользователей.

Создание списка доступных книг

В качестве механизма предоставления клиентам и заказчикам списка новых книг в формате XML можно снова воспользоваться языком Perl. Будем считать, что какая-то иная часть ПО библиотеки периоди-

чески читает XML-данные и обновляет каталог библиотеки; на этом этапе данный компонент приложения отвечает за удаление записей в файле (или самого файла), чтобы книги в нем больше не рассматривались как новые поступления. Таким образом, второй сценарий на Perl должен лишь прочитать фрагмент XML-кода и добавить все содержащиеся в нем данные в XML-документ, который выводится на экран. Как я уже говорил, сценарий также должен добавить объявление XML и корневой элемент, заключающий информацию о новых книгах. Этот новый сценарий, приведенный в примере 14.3, при поступлении HTTP-запроса считывает файл, созданный сценарием *addBook.pl*, и выдает содержимое в виде XML-документа.

Пример 14.3. CGI-сценарий на Perl, выводящий XML-документ со списком новых книг

```
#!/usr/local/bin/perl

# Каталог, в который вы хотите записывать файлы
$baseDir = "/home/bmclaugh/javaxml/foobar/books/";

# Имя используемого файла
$filename = "books.txt";

$bookFile = $baseDir . $filename;

# Сначала открываем файл
open(FILE, $bookFile) || die "Could not open $bookFile.\n";

# Сообщаем бр?узеру медиа-тип
print "Content-type: text/plain\n\n";

# Выводим пролог и корневой элемент
print "<?xml version=\"1.0\"?>\n";
print "<books>\n";

# Выводим книги
while (<FILE>) {
  print "$_";
}

# Закрываем корневой элемент
print "</books>\n";

close(FILE);
```

Этот сценарий, сохраненный в файле *supplyBooks.pl*, принимает запрос, считывает файл, созданный сценарием *addBook.pl*, и выводит результат в формате XML через запрос HTTP. Результат обращения к этому сценарию в броузере (с несколькими добавленными книгами) показан на рис. 14.3.

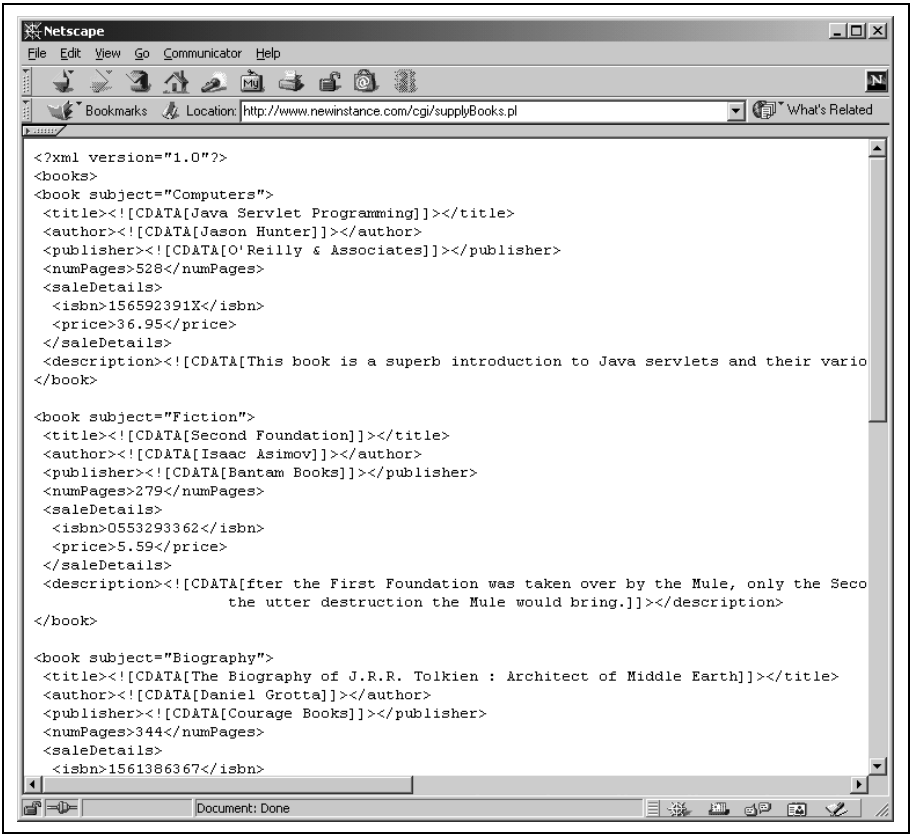

Рис. 14.3. Результат выполнения сценария supplyBooks.pl в формате XML

Как видите, мы легко превратили простое приложение библиотеки Foobar, написанное на Perl, в компонент, способный предоставлять полезную информацию клиентам библиотеки, включая магазин технической книги *mytechbooks.com*. К тому же, нам удалось это сделать без установки нового программного обеспечения, изменяющего архитектуру системы или приложения библиотеки, даже не написав ни единой строчки кода на Java!

Компания mytechbooks.com

Теперь библиотека Foobar предоставляет доступ к списку новых книг в формате XML, и компания *mytechbooks.com* становится ближе к своей цели – предоставлению актуальной информации своим клиентам. К тому же, в компании *mytechbooks.com* уже существует корпоративный стандарт применения Java для разработки приложений. Это делает процесс работы с XML-данными из библиотеки еще более простым,

поскольку в Java имеется отличная поддержка для XML, которую мы, собственно, и рассматриваем в этой книге. Сначала необходимо дать возможность компании *mytechbooks.com* получать в режиме реального времени список новых книг, а затем разработать способ автоматического предоставления этой информации покупателям.

Фильтрация XML-данных

Если вы помните, библиотека Foobar позволяла вводить в свою систему книги по различным темам; компании *mytechbooks.com* нужны только те книги, которые посвящены компьютерам. К счастью, для каждого поступления в библиотеку эта информация сохраняется в атрибуте `subject` элемента `book`. Таким образом, первым делом необходимо исключить все книги, тема которых отлична от «Computers». После этого данные по технической литературе следует отформатировать в виде HTML-страницы, которую можно представить покупателям, посещающим сервер *mytechbooks.com*.

В случае данной компании (и для данного приложения) нет возможности использовать статический HTML-код, поскольку страница, выводящая новые списки, должна генерироваться заново при каждом обращении к ней. А обработку ответов я собираюсь реализовать при помощи сервлета. Хотя система публикации Apache Cocoon была бы отличным выбором для преобразования XML-данных из библиотеки в ответ в формате HTML, компании *mytechbooks.com* требуется сделать список книг доступным в короткие сроки, и она не хочет наспех вносить серьезные изменения в свою систему. Вместо этого компания предпочла бы пока применять анализаторы и процессоры XML, а в качестве следующего шага ввести в эксплуатацию систему публикации Cocoon. Это значит, что нам придется преобразовывать данные из XML в HTML, а также фильтровать данные и добавлять различные объекты, такие как логотип компании и меню веб-приложения.

Но, собрав всю информацию об XML и XSL, имеющуюся в нашем распоряжении, мы вспомним, что даже без системы Cocoon можно применять XSL для преобразования XML-документа в формат HTML. Преобразование также позволяет отфильтровать книги, не соответствующие тематическим критериям, определяемым в *mytechbooks.com*. Принимая это во внимание, можно просто создать таблицу стилей XSL, которую и применить к XML-данным, полученным от библиотеки Foobar. В примере 14.4 приведено начало этой таблицы стилей, отвечающей за создание HTML-кода, специфичного для сайта *mytechbooks.com*.

Пример 14.4. Таблица стилей XSL для списка книг от библиотеки Foobar

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
>
```

```
<xsl:template match="books">
 <html>
  <head>
   <title>mytechbooks.com - Your Computer Bookstore</title>
  </head>
  <body background="/javaxml/techbooks/images/background.gif"
        link="#FFFFFF" vlink="#FFFFFF" alink="#FFFFFF">
   <h1 align="center">
    <font face="Arial" color="#00659C">
     &lt;mytechbooks.com&gt;
    </font>
   </h1>
   <p align="center">
    <i><b>
     Your source on the Web for computing and technical books.
    </b></i>
   </p>
   <p align="center">
    <b><font size="4" color="#00659C">
     <u>New Listings</u>
    </font></b>
   </p>
   <table border="0" cellpadding="5" cellspacing="5">
    <tr>
     <td valign="top" align="center" nowrap="nowrap" width="115">
      <p align="center">
       <font color="#FFFFFF"><b>
        <a href="/javaxml/techbooks/">Home</a>
       </b></font>
      </p>
      <p align="center">
       <font color="#FFFFFF"><b>
        <a href="/javaxml/techbooks/current.html">Current Listings</a>
       </b></font>
      </p>
      <p align="center">
       <b><font color="#FFFFFF">
        <i>New Listings</i>
       </font></b>
      </p>
      <p align="center">
       <font color="#FFFFFF"><b>
        <a href="/javaxml/techbooks/contact.html">Contact Us</a>
       </b></font>
      </p>
     </td>
     <td valign="top" align="left">
      <table border="0" cellpadding="5" cellspacing="5">
       <tr>
        <td width="450" align="left" valign="top">
         <p>
```

```
            <b>
             Welcome to <font face="courier">mytechbooks.com</font>,
             your source on the Web for computing and technical books.
             Our newest offerings are listed on the left.  To purchase
             any of these fine books, simply click on the
             &quot;Buy this Book!&quot; link, and you will be taken to
             the shopping cart for our store.  Enjoy!
            </b>
           </p>
           <p>
            <b>
             You should also check out our current listings, information
             about the store, and you can call us with your questions.
             Use the links on the menu to the left to access this
             information.  Thanks for shopping!
            </b>
           </p>
          </td>
          <td align="left">

        <!-- Handle creation of content for each new *computer* book -->

          </td>
         </tr>
        </table>
       </td>
      </tr>
     </table>
    </body>
   </html>
  </xsl:template>

</xsl:stylesheet>
```

Хотя этот код еще не фильтрует входящие XML-данные и не преобразовывает их, он берет на себя ответственность за пользовательский интерфейс HTML. Зачастую гораздо проще позаботиться сначала о деталях представления данных, а затем уже добавлять логику, связанную с их преобразованием.

Примечание

При разработке таблиц стилей XSL, в частности, для веб-приложений, следует проверять результаты с помощью процессора XSLT, используя возможности работы из командной строки. Благодаря этому можно на каждом этапе разработки таблицы стилей быть уверенным, что она преобразует наш документ так, как мы ожидаем. Пытаться устранить проблемы в большой таблице стилей после ее завершения значительно труднее. В данном примере можно обратиться к сценарию *supplyBooks.pl* через броузер, сохранить результаты в XML-файле и тестировать его вместе с таблицей стилей по мере ознакомления с примерами.

Подобно приложению в библиотеке Foobar, этот код создает меню в левой части страницы. Меню содержит ссылки на другие части приложения, некоторую текстовую информацию о компании и ее услугах и оставляет правую колонку свободной для добавления списка новых книг.

Перед тем как фильтровать содержимое документа, необходимо добавить шаблон для формирования HTML-кода на основе содержимого отдельного элемента book. Как вы помните, эта запись выглядит примерно так:

```
<book subject="Computers">
 <title><![CDATA[Running Linux]]></title>
 <author><![CDATA[Matt Welsh]]></author>
 <publisher><![CDATA[O'Reilly & Associates]]></publisher>
 <numPages>729</numPages>
 <saleDetails>
  <isbn> 156592469X</isbn>
  <price>39.95</price>
 </saleDetails>
 <description><![CDATA[In the tradition of all O'Reilly books, Running
  Linux features clear, step-by-step instructions that always seem to
  provide just the right amount of information.]]></description>
</book>
```

Теперь можно преобразовать XML-данные в формат HTML при помощи следующего шаблона XSL:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
>

 <xsl:template match="books">
   <!-- Представление пользовательского интерфейса -->
 </xsl:template>

 <xsl:template match="book">
  <table border="0" cellspacing="1" bgcolor="#000000">
   <tr>
    <td>
     <table border="0" cellpadding="3" cellspacing="0">
      <tr>
       <td width="100%" bgcolor="#00659C" nowrap="nowrap" align="center">
        <b><font color="#FFFFFF">
         <xsl:value-of select="title" />
        </font></b>
       </td>
      </tr>
      <tr>
       <td width="100%" align="center" nowrap="nowrap" bgcolor="#FFFFFF">
        <font color="#000000"><b>
```

```
          Author: <xsl:value-of select="author" /><br />
          Publisher: <xsl:value-of select="publisher" /><br />
          Pages: <xsl:value-of select="numPages" /><br />
          Price: <xsl:value-of select="saleDetails/price" /><br />
          <br />
         </b></font>
         <xsl:element name="a">
          <xsl:attribute name="href">/servlets/BuyBookServlet?isbn=
           <xsl:value-of select="saleDetails/isbn" />
          </xsl:attribute>
          <font color="#00659C">Buy the Book!</font>
         </xsl:element>
        </td>
       </tr>
      </table>
     </td>
    </tr>
   </table>
   <br />
  </xsl:template>

</xsl:stylesheet>
```

Данный шаблон отбирает элементы `book`, а затем создает таблицу с заголовком в одной строке и содержанием во второй. Вся эта таблица помещается в другую таблицу с черным фоном, в результате чего таблица выдается на экран, окруженная черной объемной рамкой. Название книги помещается в заголовок таблицы, а данные о книге (автор, издательство, количество страниц и цена) добавляются в содержимое таблицы. Наконец, приводится ссылка на сервлет Java *BuyBookServlet*, позволяющий купить книгу в режиме реального времени. Значение элемента `isbn` передается в качестве аргумента этому сервлету, что дает ему возможность загрузить заказанную книгу.

Внимание

Следует убедиться, что в таблице стилей XSL строка, указывающая на сервлет *BuyBookServlet*, и строка с элементом `xsl:value-of`, выбирающая номер ISBN книги, в действительности расположены в одной строке. В противном случае в полученном URL могут присутствовать пробелы или символ возврата каретки, что приведет к передаче сервлету некорректной информации. В примере эта информация разбита на две строки из-за ограничений размеров печатной страницы.

Последнее, что необходимо сделать, работая с таблицей стилей, – это гарантировать, что новый шаблон применяется к документу и что он применяется только к тем книгам, темой которых являются компьютеры. В таблице стилей можно сослаться на значение атрибута `subject` с помощью символа `@` и отфильтровать запросы с помощью атрибута `select` элемента `xsl:apply-templates`:

```
    </td>
    <td align="left">

     <!-- Создание содержимого для каждой новой *компьютерной* книги -->
    <xsl:apply-templates select="book[@subject='Computers']" />

    </td>
   </tr>
  </table>
```

Мы ссылаемся на значение атрибута и сравниваем его с константой, заключенной в одинарные кавычки, потому что само выражение XPath заключено в двойные кавычки. Поскольку мы обращаемся к атрибуту вложенного элемента, то должны ссылаться на элемент по имени и заключить в квадратные скобки выражение, содержащее атрибут элемента. Это гарантирует, что шаблон будет применяться только к книгам с темой «Computers», и только такие книги будут включаться в конечную HTML-страницу. Завершив создание таблицы стилей, ее можно сохранить в файле *computerBooks.xsl* и ссылаться на нее программными средствами из сервлета Java, написание которого рассмотрено далее.

XSLT в сервлете

Подготовив таблицу стилей к применению, необходимо добавить код на Java, чтобы применить ее к XML-данным из библиотеки Foobar. Доступ к этим данным легко получить, применив класс Java `java.net.URL` для отправки HTTP-запроса информационной системе библиотеки. После этого останется лишь непосредственно применить XSL-преобразование в программе. В примере 14.5 приведен код сервлета Java, загружающего XML-данные из библиотеки Foobar, и указано, где следует вставлять код, отвечающий за преобразование.

Пример 14.5. Java-сервлет для преобразования списка книг в формат HTML

```
package com.techbooks;

import java.io.FileInputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.URL;
import javax.servlet.*;
import javax.servlet.http.*;

public class ListBooksServlet extends HttpServlet {

  /** Имя компьютера, к которому мы подключаемся для получения списка книг */
  private static final String hostname = "newInstance.com";
  /** Номер порта, к которому мы подключаемся для получения списка книг */
  private static final int portNumber = 80;
```

```
    /** Файл со списком книг (URI) */
    private static final String file = "/cgi/supplyBooks.pl";

    /** Таблица стилей, применяемая к XML */
    private static final String stylesheet =
        "/home/bmclaugh/javaxml/techbooks/XSL/computerBooks.xsl";

    public void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        res.setContentType("text/html");

        // Подключаемся и получаем список книг в формате XML
        URL getBooksURL = new URL("http", hostname, portNumber, file);
        InputStream in = getBooksURL.openStream();

        // Преобразование XML из потока ввода в HTML
    }
}
```

Этот простой сервлет обращается к приложению библиотеки Foobar с помощью HTTP-запроса и получает ответ в формате XML в поток `InputStream`.[1] Этот поток затем следует использовать как параметр для процессора XSLT наряду с таблицей стилей XSL, определенной в сервлете в виде константы.

В настоящее время не существует Java API, описывающего, каким образом преобразования XSLT могут происходить в программах; однако в каждом процессоре должны быть классы, которые позволяют вызывать такие преобразования из кода на Java. Я по-прежнему применяю процессор Apache Xalan; тем же, кто работает с другим процессором, необходимо проконсультироваться у разработчика по поводу метода или методов, которые следует вызывать в своих программах.

В случае процессора Apache Xalan класс `XSLTProcessor` из пакета `org.apache.xalan.xslt` служит именно для этой цели. При выполнении преобразования он использует в качестве параметров объект `XSLTInputSource`, являющийся контейнером обрабатываемого XML-файла, объект `XSLTInputSource`, являющийся контейнером применяемой таблицы стилей XSL, и объект `XSLTResultTarget` для выдачи результата преобразования. Все три этих вспомогательных класса также входят в пакет `org.apache.xalan.xslt`. Их удобно создавать, передавая конструктору объект `InputStream` (для `XSLInputSource`) или `OutputStream` (для `XSLTResultTarget`). Наш XML-документ представлен в виде объекта `InputStream`, таблицу стилей XSL можно представить в виде объекта `FileInputStream`, а Servlet API предоставляет нам простой доступ к объекту `ServletOutputStream` с помощью метода `getOutputStream()` объекта `HttpServletResponse`. Последняя деталь, которую необходимо уточнить, касается

[1] Дополнительную информацию о классе URL и организации ввода/вывода в Java можно найти в книге Эллиота Р. Гарольда (Elliotte Rusty Harold) «Java I/O», O'Reilly (Ввод/вывод на Java).

получения экземпляра класса XSLTProcessor. Поскольку существует несколько внутренних механизмов, которые можно применять для преобразования, экземпляр этого класса не создается непосредственно, а получается с помощью класса XSLTProcessorFactory, который также входит в пакет org.apache.xalan.xslt. Мы уже освоились с классами-фабриками, поэтому остается лишь импортировать требуемые классы и добавить в сервлет вызов методов, выполняющих преобразования:

```
package com.techbooks;

import java.io.FileInputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.URL;
import javax.servlet.*;
import javax.servlet.http.*;

// Импортируем компоненты XSLT-процессора Xalan
import org.apache.xalan.xslt.XSLTInputSource;
import org.apache.xalan.xslt.XSLTProcessor;
import org.apache.xalan.xslt.XSLTProcessorFactory;
import org.apache.xalan.xslt.XSLTResultTarget;

public class ListBooksServlet extends HttpServlet {

    /** Имя компьютера, к которому мы подключаемся для получения списка книг */
    private static final String hostname = "newInstance.com";
    /** Номер порта, к которому мы подключаемся для получения списка книг */
    private static final int portNumber = 80;
    /** Файл со списком  книг (URI) */
    private static final String file = "/cgi/supplyBooks.pl";

    /** Таблица стилей, применяемая к XML */
    private static final String stylesheet =
        "/home/bmclaugh/javaxml/techbooks/XSL/computerBooks.xsl";

    public void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        res.setContentType("text/html");

        // Подключаемся и получаем список книг в формате XML
        URL getBooksURL = new URL("http", hostname, portNumber, file);
        InputStream in = getBooksURL.openStream();

        // Преобразование XML из потока ввода в HTML
        try {
            XSLTProcessor processor = XSLTProcessorFactory.getProcessor();

            // Преобразование XML при помощи таблицы стилей XSL
            processor.process(new XSLTInputSource(in),
                              new XSLTInputSource(
                                  new FileInputStream(stylesheet)),
                              new XSLTResultTarget(
```

```
                                        res.getOutputStream()));

            } catch (Exception e) {
                PrintWriter out = res.getWriter();
                out.println("Error: " + e.getMessage());
                out.close();
            }
        }
    }
```

Примечание

Для выполнения этого преобразования я также мог применить TrAX API из JAXP 1.1. Однако JAXP 1.1 по-прежнему представляет собой новую технологию, и я встречал не много людей, уже признавших ее. Кроме того, большинство сред исполнения сервлетов, особенно Tomcat, по-прежнему поставляются с JAXP 1.0, а многие из них полагаются на поведение по умолчанию и не предоставляют новых версий JAXP.

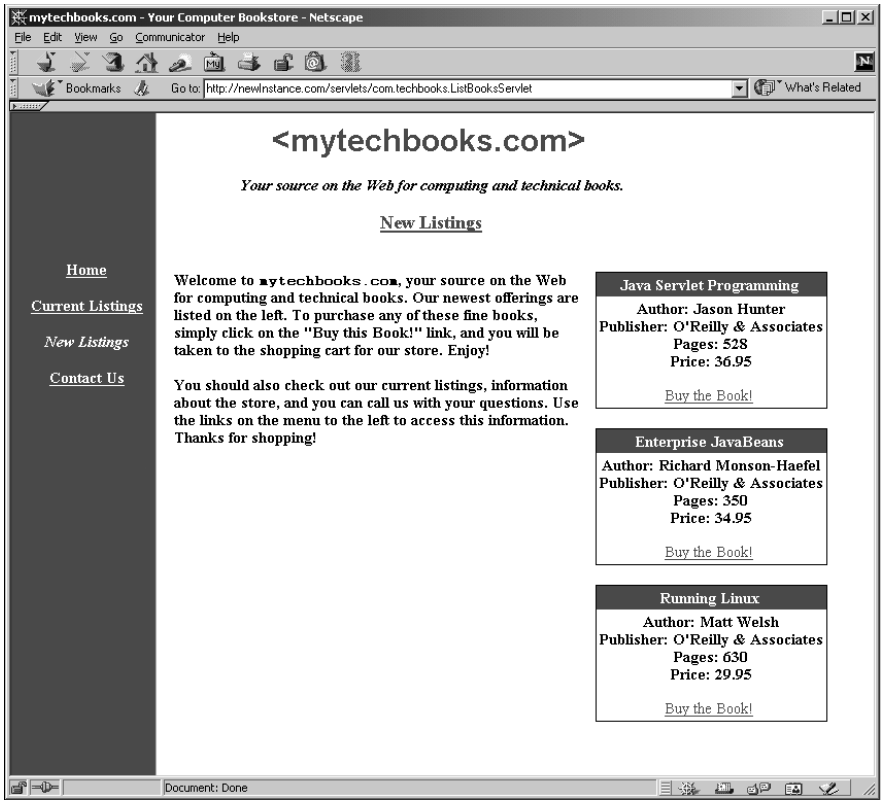

Рис. 14.4. Список новых книг от mytechbooks.com в формате HTML

Когда запрашивается этот новый сервлет, он, в свою очередь, запрашивает XML-данные из библиотеки Foobar. Затем эти данные (список вновь поступивших книг) преобразуются и выдаются на экран в формате HTML. Ответ сервлета должен выглядеть примерно так, как показано на рис. 14.4.

Наряду со ссылками в меню слева (не реализованными в примере) список новых книг выдается в изящном формате, с новейшими данными (благодаря изменениям в системе библиотеки Foobar!), а также со ссылками, позволяющими купить книгу с помощью нескольких щелчков мыши. Теперь клиенты фирмы *mytechbooks.com* могут легко просматривать списки новых книг в режиме реального времени. Осталось лишь организовать доставку данных клиентам так, чтобы им даже не нужно было вводить URL. Далее мы рассмотрим, как решить эту непростую проблему.

Рассылка или запрос

До сих пор мы рассматривали создание приложений, полагая, что клиенты приложения всегда будут запрашивать данные и содержимое документов. Другими словами, пользователь должен был набрать URL в броузере (в случае списка новых книг от *mytechhbooks.com*), или приложение, подобное сервлету компании *mytechbooks.com*, должно было выполнить HTTP-запрос XML-данных (в случае библиотеки Foobar). Хотя это вовсе не сложно, для компании типа *mytechbooks.com* это не всегда наилучший способ продажи книг. Клиенты, запрашивающие данные, должны регулярно посещать сайты, где они покупают книги, но часто они не делают этого по нескольку дней, недель и даже месяцев. Хотя эти клиенты могут приобретать товары и услуги в больших объемах (когда они об этом помнят), в среднем такие покупки не дают дохода, сравнимого с прибылями от мелких покупок, которые совершаются чаще.

Осознавая эту тенденцию, компания *mytechbooks.com* хочет иметь возможность рассылать данные своим клиентам. Рассылка данных включает уведомление клиента (не требуя действий с его стороны) о том, что поступили новые товары, или о том, что какие-либо товары заканчиваются. Благодаря этому клиент может делать покупки чаще без необходимости помнить о посещении веб-страницы. Однако рассылка данных клиентам в Сети затруднена, поскольку Интернет не является «толстым» клиентом: труднее посылать всплывающие сообщения или генерировать предупреждения для пользователей. Компания *mytechbooks.com* обнаружила, что настраиваемые стартовые страницы в Интернете, такие как My Netscape и My Yahoo, пользуются большой популярностью. Из консультаций со специалистами Netscape руководство *mytechbooks.com* узнало о технологии, называемой Rich Site Summary (RSS), и пришло к выводу, что данная технология может стать решением задачи по рассылке данных клиентам.

Rich Site Summary

Rich Site Summary (RSS) – это особый формат XML. Он имеет собственное DTD и определяет так называемые *каналы* (*channel*). Канал – это способ представления данных по определенной теме, он предусматривает название и описание канала, изображение или логотип, а также несколько *разделов* (*items*) канала. Каждый раздел представляет собой какую-либо интересную тему, связанную с каналом, предлагаемый продукт или услугу. Поскольку допустимые элементы раздела являются весьма общими (`title`, `description`, `hyperlink`), в качестве раздела канала можно представить практически все что угодно. Канал RSS предназначен не для предоставления доступа ко всему содержимому сайта, а только для краткой рекламы компании или услуги, уместной для публикации в системах типа порталов или внутри врезки на сайте. В действительности разнообразные «завлекалочки» (widgets) в Netscape Netcenter представляют собой каналы RSS, и фирма Netscape предоставляет возможность создания новых каналов, которые можно зарегистрировать с помощью Netcenter. У Netscape имеется также встроенная система для визуализации каналов RSS в формате HTML, что, конечно же, очень хорошо подходит для стартовых страниц Netcenter.

У вас может возникнуть некоторое беспокойство по поводу того, что RSS для фирмы Netscape является тем же, чем анализатор MSXML для Microsoft: его сложно интегрировать с другими инструментами или платформами. Хотя изначально технология RSS разрабатывалась фирмой Netscape специально для Netcenter, тот факт, что документ RSS следует структуре XML, делает его применимым в произвольном приложении, которое способно прочитать DTD. На деле многие приложения и сайты, разработанные в стиле порталов, начинают использовать RSS. В их числе проект Apache Jetspeed (*http://jakarta.apache.org/jetspeed*) – свободно распространяемая система для реализации информационных корпоративных порталов. Jetspeed использует тот же самый формат RSS, что и Netscape, но визуализирует его совершенно иначе. Благодаря четкой грамматике RSS это делается очень легко.

Поскольку многие пользователи имеют свои домашние или стартовые страницы, либо подобные веб-страницы, которые они часто посещают, фирма *mytechbooks.com* хотела бы создать канал RSS, представляющий каталог новых книг и позволяющий заинтересованным клиентам сразу же купить те книги, которые им приглянулись. Эта технология является эффективным средством рассылки данных, поскольку продукты, подобные Netcenter, автоматически обновляют канал RSS с той частотой, которая удобна пользователю.

Создание XML RSS документа

Первое, что нужно сделать, чтобы применить RSS, – это создать RSS-файл. Делается это очень просто: кроме указания правильного DTD

и следования этому DTD, в создании документа RSS нет ничего сложного. В примере 14.6 приведен RSS-файл, созданный компанией *mytechbooks.com*.

Пример 14.6. Пример RSS-документа для mytechbooks.com

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

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns="http://purl.org/rss/1.0/"
>
 <channel>
  <title>mytechbooks.com New Listings</title>
  <link>http://www.newInstance.com/javaxml2/techbooks</link>
  <description>
   Your online source for technical material, computers,
   and computing books!
  </description>

  <image rdf:resource="http://newInstance.com/javaxml2/logo.gif" />

  <items>
   <rdf:Seq>
    <rdf:li resource="http://www.newInstance.com/javaxml2/techbooks" />
   </rdf:Seq>
  </items>
 </channel>

  <image rdf:about="http://newInstance.com/javaxml2/logo.gif">
   <title>mytechbooks.com</title>
   <url>http://newInstance.com/javaxml2/logo.gif</url>
   <link>http://newInstance.com/javaxml2/techbooks</link>
  </image>

  <item rdf:about="http://www.newInstance.com/javaxml2/techbooks">
   <title>Java Servlet Programming</title>
   <link>
    http://newInstance.com/javaxml2/techbooks/buy.xsp?isbn=156592391X
   </link>
   <description>
    This book is a superb introduction to Java servlets
    and their various communications mechanisms.
   </description>
  </item>
</rdf:RDF>
```

Корневым элементом должен быть элемент `RDF` из пространства имен `RDF`, как показано в примере. Внутри корневого элемента должен присутствовать единственный элемент `channel`. Он содержит элементы, описывающие канал (`title`, `link` и `description`), необязательное графическое изображение, которое можно связать с каналом (а также информацию

об этом изображении) и до 15 элементов item[1], каждый из которых описывает один раздел, относящийся к каналу. В каждом элементе item имеются элементы title, link и description, названия которых говорят сами за себя. Еще можно добавить необязательное поле ввода текста и кнопку отправки информации, хотя их и нет в примере. Все подробности о допустимых элементах и атрибутах можно найти в спецификации RSS 1.0, доступной на сайте *http://groups.yahoo.com/group/rss-dev/files/specification.html*.

Примечание

Как и в предыдущих примерах, в реальных документах каналов RSS следует избегать пробелов в элементах link и url и хранить всю информацию в единой строке. В примере это не отражено из-за ограничений размера печатной страницы.

Тем не менее, существует один сложный момент, за которым нужно следить. Вы обратили внимание, что элемент item (или элементы) на самом деле не вложен в элемент channel? Для создания связей между разделами документа и каналом нужно использовать некоторые конструкции RDF (Resource Description Framework, потомком которой является RSS):

```
<items>
 <rdf:Seq>
  <rdf:li resource="http://www.newInstance.com/javaxml/techbooks" />
 </rdf:Seq>
</items>
```

В данном случае элемент items вложен в элемент channel. Затем конструкции li из пространства имен, определенного RDF, присваивается URI (посредством атрибута resource). Для каждого раздела, который необходимо связать с каналом, передайте атрибут about (опять же, в пространстве имен RDF) и присвойте ему тот же URI, который используется в описателе ресурса канала:

```
<item rdf:about="http://www.newInstance.com/javaxml/techbooks">
  <!-- Содержимое -->
</item>
```

Для каждого раздела с таким URI можно установить соответствие между разделом и каналом с тем же URI. Другими словами, мы только что связали канал из RSS-файла и его разделы. Этот же подход применим и для ссылки на изображение; необходимо использовать элемент image в элементе channel, задав URL изображения в качестве значения атрибута rdf:resource. Затем следует определить элемент image *вне* эле-

[1] Этот лимит установлен не в RSS 1.0, а используется для обратной совместимости с RSS 0.9 и 0.91.

мента channel, передав URL, описание и ссылку. Наконец, используйте атрибут rdf:about (как в элементе item) для задания того же URL, что и в элементе image канала. Разобрались? Все это немного отличается от RSS 0.9 и 0.91 (эти версии были рассмотрены в первом издании книги), так что будьте внимательны и не перепутайте старую спецификацию и новую.

Создавать RSS-файлы программными средствами достаточно просто – процедура аналогична той, которая применялась для генерации HTML-кода в случае сайта компаниии *mytechbooks.com*. Половина содержимого RSS-файла (информация о канале, а также об изображении) является статической; только элементы item должны генерироваться динамически. Однако пока вы собирались открыть редактор *vi* и начать создавать еще одну таблицу стилей XSL, возникло дополнительное требование: канал RSS будет храниться не на том сервере, который мы использовали в предыдущем примере, и на ней имеются лишь давно устаревшие версии библиотек Apache Xalan. Поскольку на этой же машине работают приложения, которые должны быть доступны непрерывно, такие как биллинговая система, комания *mytechbooks.com* не хочет обновлять эти библиотеки до тех пор, пока не будет пройден этап контроля изменений – процесс, занимающий неделю. Однако у них есть в наличии более новые версии библиотек Xerces (поскольку синтаксический анализ XML применяется в биллинговой системе), поэтому Java API для работы с XML имеется[1]. В данном примере для преобразования XML из библиотеки Foobar в формат RSS используется JDOM. Пример 14.7 выполняет именно эту задачу.

Пример 14.7. Java-сервлет для преобразования списка новых книг в документ RSS

```
package com.techbooks;

import java.io.FileInputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.URL;
import java.util.Iterator;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;
```

[1] Это в самом деле несколько глупая ситуация, и, может быть, ее возникновение не столь уж вероятно. Однако она дает возможность рассмотреть другие альтернативы для создания XML с помощью программных средств. Не слишком иронизируйте над абсурдностью данного примера: все примеры из этой книги, включая подобные этому, взяты из практического опыта консультирования реальных компаний. Может статься, что в вашем следующем проекте возникнут такие же требования.

```
// JDOM
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.input.SAXBuilder;

public class GetRSSChannelServlet extends HttpServlet {

    /** Имя компьютера, к которому мы подключаемся для получения списка
    книг */
    private static final String hostname = "newInstance.com";
    /** Номер порта, к которому мы подключаемся для получения списка книг */
    private static final int portNumber = 80;
    /** Файл со списком книг (URI) */
    private static final String file = "/cgi/supplyBooks.pl";

    public void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();

        // Подключаемся и получаем список книг в формате XML
        URL getBooksURL = new URL("http", hostname, portNumber, file);
        InputStream in = getBooksURL.openStream();

        try {
           // Запрашиваем реализацию SAX и используем анализатор по умолчанию
            SAXBuilder builder = new SAXBuilder();

            // Создаем документ
            Document doc = builder.build(in);

            // Выводим XML
            out.println(generateRSSContent(doc));

        } catch (JDOMException e) {
            out.println("Error: " + e.getMessage());
        } finally {
            out.close();
        }
    }

    /**
     * <p>
     * Данный метод создает RSS XML-документ на основе полученного
     *  документа JDOM <code>Document</code>.
     * </p.
     *
     * @param doc <code>Document</code> - исходный документ.
     * @return <code>String</code> - конечный RSS-файл.
     * @throws <code>JDOMException</code> генерируется при возникновении
                                          ошибки.
     */
```

```
private String generateRSSContent(Document doc) throws JDOMException {
    StringBuffer rss = new StringBuffer();

    rss.append("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n")
       .append("<rdf:RDF ")
       .append("xmlns:rdf=\"http://www.w3.org/1999/02/22-rdf-syntax-
               ns#\"\n")
       .append("         xmlns=\"http://purl.org/rss/1.0/\"\n")
       .append(">\n")
       .append(" <channel>\n")
       .append("  <title>mytechbooks.com New Listings</title>\n")
       .append("  <link>http://www.newInstance.com/javaxml2/techbooks")
       .append("</link>\n")
       .append("  <description>\n")
       .append("   Your online source for technical material,
               computers, \n")
       .append("   and computing books!\n")
       .append("  </description>\n\n")
       .append("  <image ")
       .append("rdf:resource=\"http://newInstance.com/javaxml2/
               logo.gif\"")
       .append(" />\n\n")
       .append("  <items>\n")
       .append("   <rdf:Seq>\n")
       .append("    <rdf:li ")
       .append("resource=\"http://www.newInstance.com/javaxml2/
                techbooks\"")
       .append(" />\n")
       .append("   </rdf:Seq>\n")
       .append("  </items>\n")
       .append(" </channel>\n\n")
       .append("  <image ")
       .append("rdf:about=\"http://newInstance.com/javaxml2/
               logo.gif\">\n")
       .append("   <title>mytechbooks.com</title>\n")
       .append("   <url>http://newInstance.com/javaxml2/logo.gif</
                  url>\n")
      .append("   <link>http://newInstance.com/javaxml2/techbooks</
                  link>\n")
       .append("  </image>\n\n");

    // Добавляем пункт для каждой новой книги по компьютерной тематике
    List books = doc.getRootElement().getChildren("book");
    for (Iterator i = books.iterator(); i.hasNext(); ) {
        Element book = (Element)i.next();
        if (book.getAttribute("subject")
                .getValue()
                 .equals("Computers")) {
            // Выводим пункт
            rss.append("<item rdf:about=\"http://www.newInstance.com/")
               .append("javaxml2/techbooks\">\n")
                // Добавляем заголовок
```

```
                    .append(" <title>")
                    .append(book.getChild("title").getContent())
                    .append("</title>\n")
                     // Добавляем ссылку для покупки книги
                    .append(" <link>")
                    .append("http://newInstance.com/javaxml2")
                    .append("/techbooks/buy.xsp?isbn=")
                    .append(book.getChild("saleDetails")
                                .getChild("isbn")
                                .getContent())
                    .append("</link>\n")
                    .append(" <description>")
                     // Добавляем описание
                    .append(book.getChild("description").getContent())
                    .append("</description>\n")
                    .append("</item>\n");

            }
        }

        rss. append("</rdf:RDF>");

        return rss.toString();
    }
}
```

Ничто в этом коде уже не должно казаться неожиданным: мы импортируем необходимые классы JDOM и классы ввода/вывода, а затем получаем доступ к приложению библиотеки Foobar, как это делалось в классе *ListBooksServlet*. Получаемый поток `InputStream` используется для создания объекта JDOM `Document` с помощью анализатора по умолчанию (Apache Xerces) и реализации JDOM Builder, основанной на SAX.

Затем мы передаем объект JDOM `Document` методу `generateRSSContentMethod()`, выводящему все статическое содержимое канала RSS. Далее этот метод получает элементы `book` из XML-документа из библиотеки и обходит их в цикле, игнорируя те из них, значение атрибута `subject` которых не равно «Computers».

Примечание

Опять же, кое-что сделано лишь для наглядности. Например, этот код непосредственно выводит XML-данные; с таким же успехом можно было бы создавать дерево JDOM и выводить его при помощи `XMLOutputter`. Конечно, так же можно было бы использовать DOM в большей части кода сервлета. Все это вполне допустимые варианты.

Наконец, каждый элемент, соответствующий критерию отбора, добавляется в канал RSS. Ничего особенно замечательного, не так ли? На рис. 14.5 показан пример результата обращения к сервлету `GetRSSChannelServlet`.

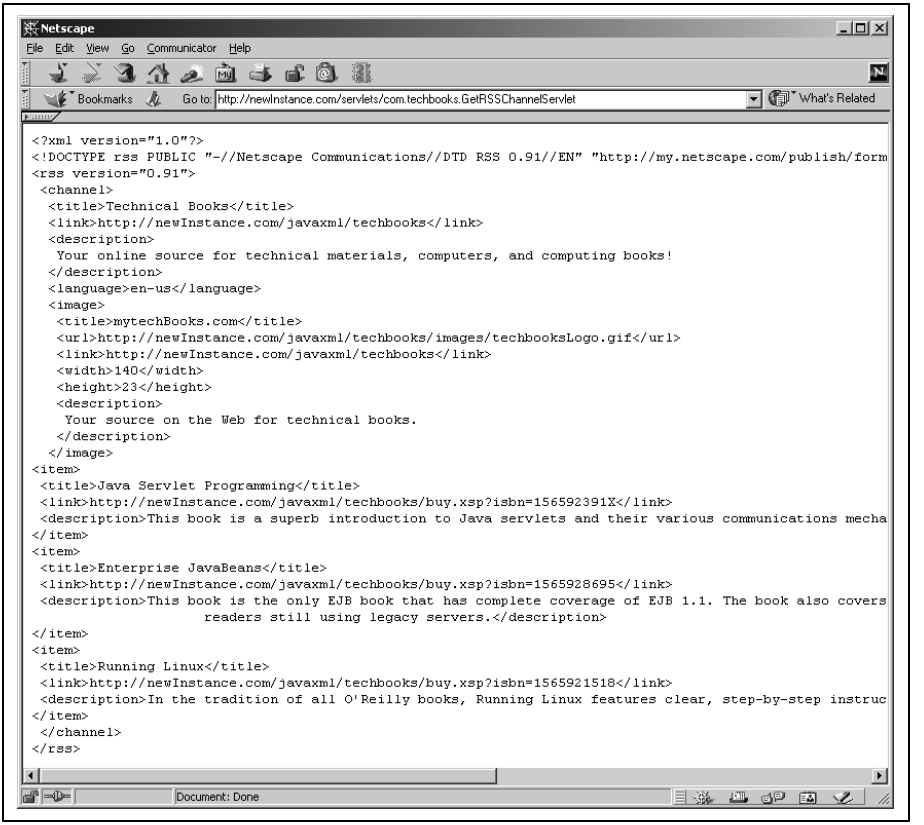

Рис. 14.5. RSS-канал, сгенерированный сервлетом GetRSSChannelServlet

Подготовив канал RSS к использованию, компания *mytechbooks.com* тем самым обеспечила доступ к своим данным любому поставщику услуг, который поддерживает RSS! Чтобы предоставить доступ клиентам к своему каналу, компания *mytechbooks.com* хотела бы проверить, являются ли действительными ее RSS-документы, и посмотреть на пример отображаемого HTML-кода (полагаю, вы тоже этого хотите).

Проверка работы

Теперь проверим это все в действии. Перейдите на сайт *http://www.redland.opensource.ac.uk/rss*. Здесь расположен инструмент для тестирования в режиме реального времени под названием «Redland RSS viewer», который проверит действительность RSS-канала, а также отобразит его в виде HTML. Необходимо гарантировать, что переданный RSS-документ каким-либо образом доступен в режиме реального времени, например посредством обращения к упомянутому сервлету. Введите URL сервлета или RSS-документа и выберите значение «Yes»

в поле «Format output as a box». Так вы укажете программе просмотра на необходимость отобразить канал в виде HTML-окна, подобного тем, которые можно увидеть на сайте Netscape's Netcenter или *http://www.oreilly.com*, на котором также доступны RSS-каналы. Полученный результат представлен на рис. 14.6.

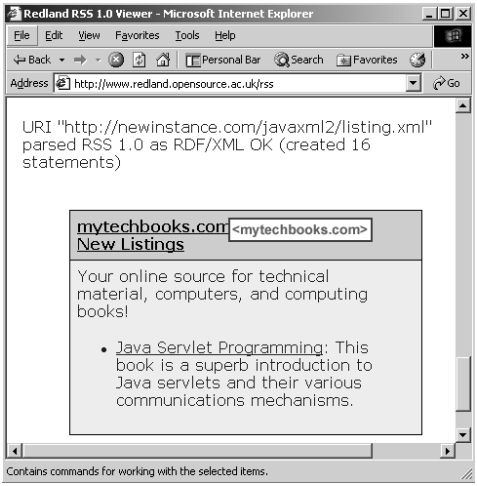

Рис. 14.6. RSS-документ, отформатированный в виде HTML

Также можно выбрать несколько других каналов RSS в программе просмотра и узнать, как они выглядят в формате HTML. Особый интерес представляет канал Meerkat, поскольку он использует практически все доступные в настоящее время возможности RSS. Кроме того, если в RSS-документе имеются ошибки, программа просмотра сообщит о них, что очень полезно при поиске ошибок, который необходимо выполнить перед тем, как начать использовать RSS-канал в реальной жизни.

В этой главе не приводится код, анализирующий и форматирующий RSS; помимо того, что он покажется вам очень простым, на каждом сайте понадобится реализовать различное форматирование для каналов RSS. Чтобы получить представление о различном оформлении RSS-каналов, посетите сайты *http://www.servlets.com* (справа снизу), *http://www.oreilly.com* и *http://www.xml.com*, на всех из них применяется различное форматирование. При чтении RSS-канала его, вероятно, следует рассматривать как XML и использовать SAX, DOM или JDOM для чтения данных и преобразования их в желаемый формат. Другими словами, нет причин воспринимать канал RSS иначе, чем другие XML-документы; просто формат известен заранее. Теперь вы готовы к тому, чтобы применять каналы RSS на собственных веб-сайтах.

Что случилось с Netcenter?

Читатели первого издания могут заинтересоваться, куда делся раздел, посвященный просмотру канала RSS в Netscape Netcenter. Объясню тем, кто не видел это издание, – Netscape предоставлял возможность публикации RSS в качестве каналов, которые можно добавить на сайт *my.netscape.com*. Это была потрясающая возможность, т. к. можно было легко форматировать и просматривать RSS в замечательном интерфейсе Netscape. Но после выхода RSS 1.0 Netscape удалил все ссылки на публикацию, и больше не предоставляет подобную возможность. Так что тем, кто ищет способ использовать подстроенные каналы на домашней странице Netcenter, не повезло.

Мы разобрались с тем, что такое RSS. Что более важно, вы, вероятно, поняли, что помимо SOAP и веб-служб существует множество способов взаимодействия с RSS. Изменяйте примеры и применяйте понятия из этой главы в собственных приложениях, реализуя оформление и бизнес-логику согласно собственным потребностям. Теперь у вас есть представление о том, как это сделать, и у вас будет свободное время, чтобы поиграть вечерком на гитаре (хм, это про меня... в любом случае, вы меня поняли).

Что дальше?

Мы познакомились с различными технологиями для взаимодействия, такими как публикация содержимого для клиента, применение SOAP для связи с удаленными приложениями и взаимодействие B2B. Теперь мы вновь сосредоточимся на коде. И в следующей главе поговорим о связывании данных XML, которое является более простым способом работы с XML-документами и объектами Java. Я также расскажу, как это связано с долгоживущими объектами и их настройкой – концепциями, которые часто применяются при разработке корпоративных приложений.

15

- *Основные принципы*
- *Castor*
- *Zeus*
- *JAXB*
- *Что дальше?*

Связывание данных

При рассмотрении инструментария XML я старался придерживаться определенной «гаммы». Когда мы начинали говорить о Java в главе 2, вы полностью контролировали ситуацию. SAX предоставлял лишь тончайший слой над XML и по существу обеспечивал систему, в которую интегрируются многочисленные пользовательские методы. В результате перехода к DOM и JDOM размеры полученной вами помощи немного увеличились, а могущество пошло на убыль. Эти модели, основанные на деревьях, хранимых в памяти, оказались более удобными, но за это пришлось заплатить некоторой потерей производительности. Перейдя к JAXP, мы продвинулись вверх по иерархии, получив еще один уровень абстракции. К этому времени вы все еще имели некоторый контроль, но работали строго в XML.

Затем я включил, так сказать, следующую передачу, и мы перешли к рассмотрению систем веб-публикации, XML-RPC, SOAP, а также межкорпоративных и веб-служб. Это была огромная разница по сравнению с простым «рукопашным» подходом, предпринятым в первой половине книги. По мере роста комфортабельности и специализации инструментов происходит потеря контроля над обработкой XML, и мы иногда не видим XML-кода вообще (как в XML-RPC). Это может помешать вам (как помешало мне) немного «запачкать» руки. Например, при работе с SOAP невозможно осуществлять тонкую подстройку значений из XML-документа, что было доступно при работе с SAX или JDOM. Однако удобство WSDL было заметным и имело преимущество перед неверным набором имени элемента в DOM. Короче говоря, мне хотелось найти золотую середину.

Связывание данных и представляет собой такую середину, но прежде чем говорить о ней, не помешает охватить взглядом весь спектр. В этой главе я представлю способ получения максимума возможностей низ-

коуровневого API с максимумом удобства системы приложений, основанной на XML (примером такой системы является SOAP). Вероятно, практически каждый в определенный момент не откажется от такого сочетания, при помощи которого можно решать различные задачи. В первую очередь рассмотрим основные принципы, на которых основана работа механизма связывания данных. Это будет общий обзор, поскольку различные реализации механизма связывания обладают своими собственными характерными методологиями. Когда вы получите представление об основах, мы вкратце рассмотрим два свободно распространяемых API для связывания данных, а затем новый API от Sun – JAXB. Так что пристегните ремни и поехали!

А как начет Quick? А как насчет JATO? А как насчет ...?

Не исключено, что кто-то будет разочарован (и даже расстроен), не найдя в книге ничего о своем любимом API. Однако мой выбор не был произвольным. Я остановился на двух API с открытыми исходными кодами, совместимых с JAXB и использующих те же принципы. Свободно распространяемые API выбраны потому, что они бесплатны, и с ними можно работать уже сейчас. Я не думаю, что кто-нибудь заплатит несколько тысяч долларов за продукт вроде Breeze XML Studio, чтобы запускать мои примеры.[a] И я выбрал Castor и Zeus потому, что для наложения ограничений они руководствуются стандартами XML – схемами XML и DTD.

Не ищите слов о JXQuick, потому что здесь их нет. Схемы, используемые в JXQuick для представления ограничений документов и классов Java (QIML, QJML и т. д.) не являются стандартами XML, а значит, они менее полезны при передаче информации об XML другим приложениям. Поэтому JXQuick исключен из рассмотрения. Однако информацию о нем можно получить на сайте *http://jxml.com/2001-12/products/quick/*.

То же справедливо и для JATO, расположенного на сайте *http://sourceforge.net/projects/jato*, который скорее является языком сценариев для отображений XML и Java. JATO реализует не столько связывание данных, сколько их отображение и плохо вписывается в рамки этой главы, как и не сочетается с моделью JAXB. Оба этих пакета можно найти в Интернете и принять собственное решение. Всегда выбирайте то, что лучше всего подходит для вашего проекта.

[a] Между прочим, это не критика Breeze. Просто я вообще считаю, что некий бесплатный продукт следует предпочесть такому же продукту, за который нужно платить. Если вы хотите попробовать коммерческий продукт, пожалуйста.

Основные принципы

Прежде чем переходить к конкретным пакетам и системам, реализующим связывание данных, необходимо получить представление о том, что собой представляет этот механизм. Правда, это довольно простое понятие, так что вы начнете писать код в два счета. Во-первых, возьмите старый добрый XML-документ, подобный тому, который приведен в примере 15.1.

Пример 15.1. XML-каталог Homespun tapes

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

<catalog xmlns="http://www.homespuntapes.com">
  <item id="VD-DOK-GT01" level="4">
    <title>Doc's Guitar Fingerpicking and Flatpicking</title>
    <teacher>Doc Watson</teacher>
    <guest>Pete Seeger</guest>
    <guest>Mike Seeger</guest>
    <guest>Jack Lawrence</guest>
    <guest>Kirk Sutphin</guest>
    <description>Doc Watson, a true master of traditional guitar styles,
            teaches,
      in detail, some of the most reuested fingerpicking and flatpicking
            tunes in
      his vast repertoire, for guitarists at all levels.</description>
  </item>
  <item id="VD-WLX-GT01" level="4">
    <title>The Guitar of David Wilcox</title>
    <teacher>David Wilcox</teacher>
    <description>Create fresh new guitar sounds with rich, ringing voicings!
            David
      even shows you how to invent your own tunings.</description>
  </item>
  <item id="VD-THI-MN01" level="3">
    <title>Essential Techniques for Mandolin</title>
    <teacher>Chris Thile</teacher>
    <description>Here's a lesson that will thrill and inspire mandolin
          players at
      all levels.</description>
  </item>
  <item id="CDZ-SM01" level="4">
    <title>Sam Bush Teaches Mandolin Repertoire and Techniques</title>
    <teacher>Sam Bush</teacher>
    <description>Learn complete solos to eight traditional and orignal tunes,
          each
      one jam-packed with licks, runs, and musical variations.</description>
  </item>
</catalog>
```

В предыдущих главах рассказывалось, как применять SAX, DOM, JDOM и JAXP для доступа к этому документу. Можно обрабатывать как его структуру (имена и порядок следования элементов, атрибутов и других лексических конструкций), так и содержимое (собственно данные). Однако в большинстве случаев доступ к структуре документа не нужен, поскольку необходимо работать только с данными.

В этом случае написание кода, анализирующего документ, выделяющего данные и преобразующего их в некий формат, который вы можете использовать, превращается в занятие, напоминающее стрельбу из пушки по воробьям; и кроме того, это утомительно. Гораздо приятнее запустить программу (или применить API... начинаете понимать?), которая сделала бы это за вас и выдала бы годные к употреблению классы Java. На самом деле именно эту задачу и решает механизм связывания данных. Связывание данных включает три различных процесса, которые могут следовать один за другим в различном порядке либо протекать независимо друг от друга. Я рассмотрю их все.

Порождение класса

Первый процесс – порождение классов – обеспечивает способ преобразования XML-документа в Java-представление. Основная задача механизма связывания данных при порождении Java-представления – обеспечить доступ только к данным документа. Кроме того, связывание данных сохраняет определенный уровень семантики информации из документа. Для Java-представления данных создаются методы доступа и методы-модификаторы[1], такие как `getItem()` и `setTeacher()`, – вместо `getElement()` и `setAttribute()`. В результате обработка документов, подобных приведенному в примере 15. 1, становится скорее делом бизнес-логики, чем делом Java, что, очевидно, хорошо. Однако прежде чем XML-документ можно будет превратить в экземпляр класса, эти чудесные классы Java следует сначала создать, потому возвратимся к порождению классов.

Порождение классов (*class generation*) – это процесс создания классов Java (а возможно, и интерфейсов) на основе набора ограничений XML. Представим себе это следующим образом: ограничения XML (подобные присутствующим в DTD или схеме XML) эквивалентны определениям классов Java. Они определяют способ представления данных. С другой стороны, XML-документ эквивалентен экземпляру класса в том отношении, что является просто набором данных, который удов-

[1] Говоря «метод доступа» (accessor), я имею в виду то, что многие называют устанавливающим методом (getter), говоря «метод изменения» (mutator) – подразумеваю устанавливающий метод (setter). Однако я знаю, что сеттер – это собака, а не метод в Java, поэтому я советую моим студентам избегать этого термина. Думаю, это просто идиосинкразия.

летворяет установленным ограничениям. Теперь еще раз медленно перечитайте этот абзац, и вы все поймете.

Все системы связывания данных, о которых мы говорим в этой главе, обладают способом представления ограничений документа (обычно, посредством DTD или схемы XML, но также существуют и другие возможности, которые рассматриваются в соответствующих разделах). Эти ограничения можно обработать инструментом порождения классов и получить исходный код на Java, готовый к компиляции. Будучи скомпилированным, этот код может применяться для создания экземпляра класса данных на основе XML-документа. В результате мы получим процесс, подобный тому, который представлен на рис. 15.1.

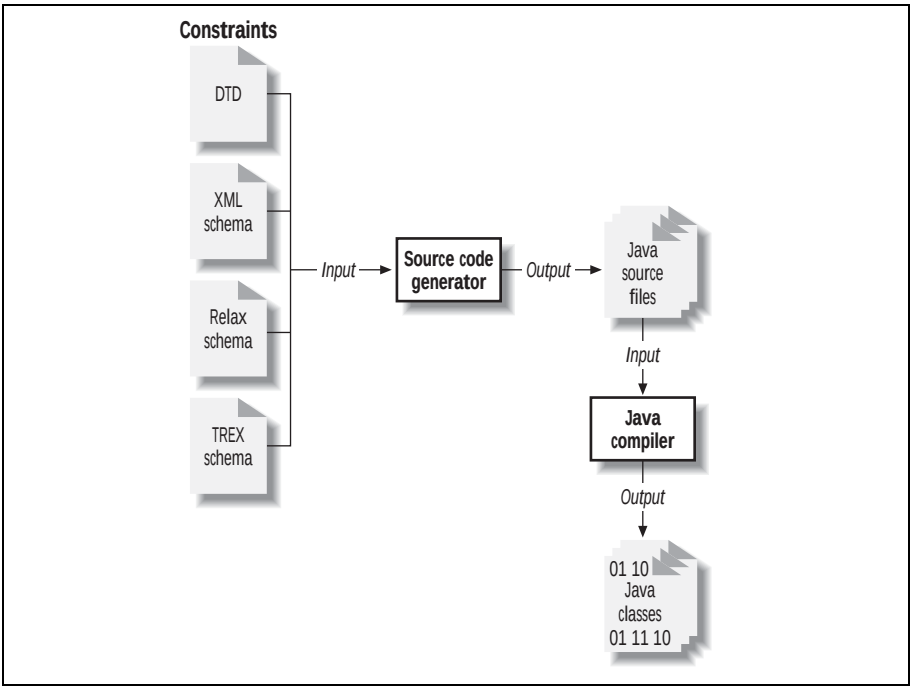

Рис 15.1. Порождение классов в связывании данных XML

Обратите внимание, что конечным результатом этого процесса могут быть конкретные классы, интерфейсы, интерфейсы и реализации или любые другие воплощения классов Java. В случае с примером 15.1 (предполагается, что ограничения представлены в некой произвольной форме) можно получить следующий интерфейс `Catalog`:

```
public interface Catalog {
    public List getItemList();
    public void addItem(Item item);
    public void setItemList(List items);
}
```

Далее можно получить интерфейс `Item`:

```
public interface Item {
    public String getID();
    public void setID(String id);
    public int getLevel();
    public void setLevel(int level);

    public String getTitle();
    public void setTitle(String title);
    public String getTeacher();
    public void setTeacher(String teacher);
    public List getGuests();
    public void addGuest(String guest);
    public void setGuestList(List guests);
    public String getDescription();
    public void setDescription();
}
```

Это существенно полезнее написания сотен строк методов обратных вызовов SAX. В результате работа с документом становится пустяковой задачей, а не тренировкой навыков в Java и XML API. Имейте в виду, что это всего лишь примеры, и они не обязательно отражают то, что будет получено в случае применения API, рассмотренных в этой главе. Однако в следующих разделах показано, как применять API и чего можно ожидать от этого.

Распаковка

Помните, что полученный набор порожденных классов можно применить не в таком уж большом количестве случаев. Конечно, можно прибегнуть к существующим низкоуровневым XML API для чтения XML-документа, выдачи данных, создания новых экземпляров порожденных классов и заполнения их данными из XML-документа. Но связывание данных решает все эти задачи, так зачем же о них думать? На самом деле системы связывания данных как раз и предназначены для решения этих задач. И в таком контексте *распаковка* (*unmarshalling*) – это процесс преобразования XML-документа в экземпляр класса Java.

Примечание

Многие (да и сам я тоже) путаются в терминологии, связанной с распаковкой и упаковкой. Я придерживаюсь терминологии, определенной в последней версии спецификации JAXB (Архитектура Java для связывания данных, Java Architecture for Data Binding) от Sun, которая, несомненно, станет стандартным словарем. В этой спецификации упаковка (marshalling) – это переход от Java к XML, а распаковка (unmarshalling) – переход от XML к Java. Рекомендовал бы читателям придерживаться этих определений.

Выясняется, что процесс достаточно простой – XML-документ передается инструменту или экземпляру класса из системы связывания данных, и в результате получается объект Java. Обычно это экземпляр

класса Java верхнего уровня, который представляет документ. Так что, вновь воспользовавшись примером 15.1, мы получим экземпляр класса `Catalog`. Как правило, понадобится выполнить приведение класса `java.lang.Object` к конкретному классу, который необходимо получить, поскольку система ничего не знает о ваших классах (потому что они были порождены). После приведения классов с объектом можно работать как с объектом `Catalog`, а не как с XML-документом. Для работы с данными можно воспользоваться различными методами доступа и модификации, а когда документ будет готов к преобразованию обратно в XML, его следует *упаковать*.

Упаковка

Упаковка (*marshalling*) – это процесс, обратный распаковке, и заключается он в преобразовании объекта Java и связанных с ним объектов в XML-представление. Во многих случаях упаковка является частью цикла преобразований из XML-формата и обратно и происходит в паре с распаковкой. В качестве примера рассмотрим рис. 15.2, иллюстрирующий типичный ход событий в приложении.

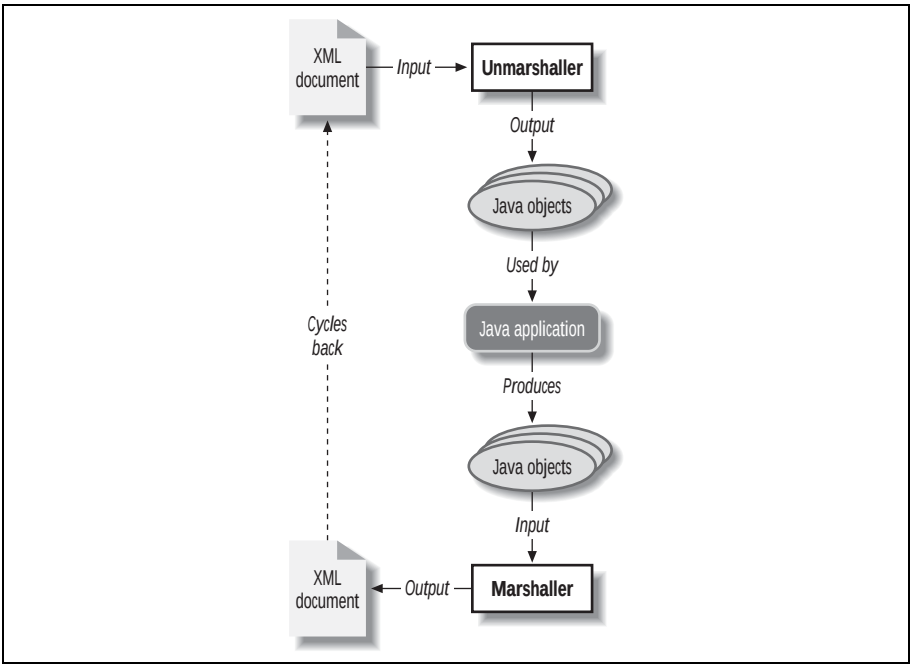

Рис. 15.2. Ход событий в приложении, использующем связывание данных XML

Существует два различных способа упаковать из объекта Java. Первый заключается в вызове метода `marshal()`. Этот метод обычно создается наряду с методами доступа и методами-модификаторами при порождении классов. Метод рекурсивно вызывает сам себя для всех связан-

ных объектов, и в результате получается XML-документ. Обратите внимание, что конечный XML-документ не обязан совпадать с исходным XML-документом; используя различные имена файлов в процессе упаковки, легко можно получить множество архивных XML-документов.

Другой подход к упаковке, а я предпочитаю именно его, заключается в том, что упаковка выполняется отдельным классом. Метод `marshal()` вызывается не для порожденного объекта, а для отдельного класса, передав ему объект для упаковки. Это полезно, т. к. помимо выполнения указанных выше задач, такой подход позволяет преобразовать в XML и те классы, которые не были получены путем распаковки XML. Можно считать, что механизм связывания данных, применяемый таким образом, становится системой создания долгоживущих (persistent) объектов. Любой объект со свойствами в стиле компонентов Bean (`setXXX()` и `getXXX()`) легко преобразуется в XML! В нашем распоряжении вся мощь связывания данных вместе с гибкостью долгоживущих объектов. Это удобная комбинация, и она поддерживается некоторыми из систем, описанных в этой главе.

Понятно, что тех, кто не знаком со связыванием данных, все это может несколько сбивать с толку, напоминая, скажем, разговор о химии. И я бы предпочел-таки что-нибудь повзрывать (ну, вы понимаете!), так что в оставшейся части главы я покажу, как пользоваться некоторыми из систем связывания данных. Поскольку нам предстоит рассмотреть четыре системы, то ни один из примеров не будет чрезвычайно сложным. Вместо этого сосредоточимся на том, как применять порождение классов, возможности упаковки и распаковки каждой из систем. Этого должно быть более чем достаточно, чтобы начать работу.

Прецеденты использования

В качестве последнего аргумента в пользу связывания данных приведу небольшую подборку прецедентов использования. Для каких-то прецедентов лучше всего подходят низкоуровневые API, такие как SAX или DOM, а для других идеально связывание данных. В табл. 15.1 перечислены распространенные прецеденты использования, типы подходящих API и короткие разъяснения, на которых основаны решения. Это поможет понять, как связывание данных вписывается в ландшафт XML.

Таблица 15.1. Прецеденты использования API

Прецедент использования	Подходящий API	Разъяснение
XML IDE	DOM или JDOM	Древовидное представление данных XML, как в IDE, четко соответствует древовидным моделям DOM и JDOM.
Сервер сообщений XML	SAX	Главным фактором здесь является скорость, а SAX обеспечивает самое быстрое чтение сообщений из потоков.

Таблица 15.1 (продолжение)

Прецедент использования	Подходящий API	Разъяснение
Конфигурационные данные	Связывание данных	Первостепенным является содержимое, а не модель. Связывание данных экономит время и упрощает работу с конфигурационными данными.
Преобразования XML	DOM или JDOM	Требуется изменение структуры (вычеркиваем связывание данных) и модификация содержимого (вычеркиваем SAX).
Клиент сообщений XML	Связывание данных	Если формат сообщения известен заранее, клиент может извлечь пользу из простых объектов Java, порожденных связыванием данных.

Очевидно, это лишь несколько распространенных приложений XML, но они должны помочь вам получить представление о том, когда лучше применять низкоуровневые API, а когда высокоуровневые.

Castor

Первая система связывания данных, о которой мы будем говорить, – это система Castor, доступная на сайте *http://castor.exolab.org*. Эта система существует уже некоторое время, и на момент написания этой книги автор работал с версией 0.9.2. Прежде всего, следует пояснить, что Castor предоставляет не просто связывание данных для XML. Этот пакет обеспечивает Java-представление не только данных XML – также можно работать с объектами LDAP, OQL для отображения SQL запросов в объекты, а также с объектами данных Java (Java Data Objects, JDO), довольно новой спецификацией от Sun, в которой идет речь о долгоживущих отображениях Java в RDBMS (Relational Database Management Systems, системы управления реляционными базами данных). Однако эта книга посвящена XML, поэтому я буду говорить только о связывании данных XML.

Установка

Castor можно загрузить со страницы *http://castor.exolab.org/download.html*. Здесь же приведены ссылки на FTP-сервер Exolab (на него можно перейти самостоятельно, как это сделал я) и перечислены доступные файлы. Лучше загрузить полную версию (тогда, если вы захотите, то сможете изучить OQL или JDO в дальнейшем). Поместите в пути к классам jar-файлы из архива и приступайте к работе.[1]

[1] На самом деле в дистрибутиве находятся два файла: *castor-0.9.2.jar* и *castor-0.9.2-xml.jar*. Первый является надмножеством для второго, так что вам нужен лишь первый файл. Если же вам нужен меньший архив, возьмите второй файл.

Примечание

В этом и последующих примерах предполагается, что анализатор XML, совместимый с SAX (например, Xerces), по-прежнему находится в путях к классам вместе с библиотеками, о которых говорится в этой главе. Если это не так, добавьте файл *xerces.jar* или jar-файл(ы) вашего анализатора в пути к классам вместе с используемой системой связывания данных.

Порождение исходного кода

Castor позволяет порождать классы на основе существующего набора ограничений. Необходимо иметь схему XML, ограничивающую данные документов. В примере 15.2 приведена такая схема для документа из примера 15.1.

Пример 15.2. Схема XML для примера 15.1 (используем Castor)

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

<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
        targetNamespace="http://www.homespuntapes.com">
  <element name="catalog">
    <complexType>
      <sequence>
        <element ref="item" minOccurs="1" maxOccurs="unbounded" />
      </sequence>
    </complexType>
  </element>

  <element name="item">
    <complexType>
      <sequence>
        <element name="title" type="string" />
        <element name="teacher" type="string" />
        <element name="guest" type="string" minOccurs="0"
           maxOccurs="unbounded" />
        <element name="description" type="string" />
      </sequence>
      <attribute name="id" type="string" />
      <attribute name="level">
        <simpleType>

          <restriction base="integer">
            <enumeration value="1" />
            <enumeration value="2" />
            <enumeration value="3" />
            <enumeration value="4" />
            <enumeration value="5" />
          </restriction>
        </simpleType>
      </attribute>
```

```
    </complexType>
  </element>
</schema>
```

Очевидно, что можно попробовать применить собственные схемы XML. Если они удовлетворяют спецификации схем XML, то будут работать для всех примеров из этого раздела.

Внимание

По крайней мере во время написания этой книги Castor поддерживал лишь проект рекомендации схемы XML, вышедший в октябре 2000 года, а не окончательную версию этой спецификации.[1] Поэтому может потребоваться внести незначительные изменения в существующие схемы, чтобы они удовлетворяли действующей спецификации. Текущий уровень совместимости со схемой XML можно выяснить по адресу *http://castor.exolab.org/xmlschema.html*.

Определив схему XML, можно создать классы для имеющихся ограничений. Свою схему, как видно из приводимых ниже инструкций, я назвал *catalog.xsd*.

При наличии схемы порождение классов с помощью Castor является пустяковой задачей. Для этого следует использовать класс `org.exolab.castor.builder.SourceGenerator`, как показано ниже:

```
java org.exolab.castor.builder.SourceGenerator -i castor/catalog.xsd
    -package javaxml2.castor
```

В данном примере команда запускается, когда схема находится в подкаталоге *castor/* текущего каталога. Схема указывается посредством ключа «-i», а пакет, генерирующий файлы, – при помощи ключа «-package». Существует множество других параметров, получить которые можно, просто указав имя класса без параметров. Класс перечислит допустимые параметры и ключи.

После выполнения команды мы получим путь к каталогу (могут возникнуть ошибки, связанные со схемой), соответствующий указанному пакету. В моем случае был создан каталог *javaxml2*, внутри которого находился подкаталог *castor*, содержащий исходные файлы *Catalog.java* и *CatalogDescriptor.java*, а также *Item.java* и *ItemDescriptor.java*. В большинстве ситуаций надо будет работать с первым файлом из каждой пары.

Также должен получиться подкаталог с именем *types*, содержащий некоторые дополнительные файлы. Они были сгенерированы потому, что пользователь определил тип для атрибута «level» в схеме XML. В результате создается класс `LevelType`. Существует только пять допустимых значений, поэтому Castor должен создать собственные клас-

[1] Сейчас Castor поддерживает спецификацию схемы XML от 2 мая 2001 года. – *Примеч. науч. ред.*

сы для обработки данного типа. С этими классами типов сложно работать, поскольку не существует способа сделать, например, следующее:

```
// Создаем новый тип со значением "1"
LevelType levelType = new LevelType(1);
```

Вместо этого необходимо получить значение, которое будет использоваться, и преобразовать его в строку. Затем можно вызвать статический метод `valueOf()` для получения экземпляра `LevelType` с верным значением:

```
LevelType levelType = LevelType.valueOf("1");
```

Конечно же, если к этому привыкнуть, это уже не кажется большой проблемой. Если это кажется несколько неясным, не беспокойтесь об этом сейчас, в следующем разделе вы увидите, как применять этот класс в практической ситуации. Скомпилировать файлы типов и другие исходные файлы, созданные системой Castor, можно при помощи простой команды:

```
javac -d . javaxml2/castor/*.java javaxml2/castor/types/*.java
```

Теперь классы можно использовать. Я не буду приводить здесь исходный код этих файлов, поскольку он достаточно велик (и вы можете взглянуть на него самостоятельно). Я лишь перечислю ключевые методы для класса `Catalog`, чтобы вы имели представление о том, чего можно ожидать:

```
package javaxml2.castor;

public class Catalog {

    // Добавляем новый элемент
    public void addItem();
    // Получаем элементы в виде перечисления
    public Enumeration enumerateItem();
    // Получаем все элементы
    public Item[] getItem();
    // Получаем число элементов
    public getItemCount();
}
```

Обратите внимание, что можно добавлять элементы и перебирать доступные элементы. Имена двух методов — `enumerateItem()` и `getItem()`, несколько необычны, так что будьте внимательны. Я не ожидал, что метод `getItem()` возвращает массив, и искал сначала методы `getItems()` или `getItemList()`. Когда порожденные классы получены, можно применять их в приложении.

Упаковка и распаковка

Когда классы, порожденные системой Castor, скомпилированы, их нужно добавить в пути к классам. Затем их можно использовать в соб-

ственных приложениях. В примере 15.3 приведена простая HTML-форма, позволяющая пользователю ввести информацию о новом элементе.

Пример 15.3. HTML-форма для добавления элементов

```
<HTML>
 <HEAD><TITLE>Добавление нового элемента в каталог</TITLE></HEAD>
 <BODY>
  <H2 ALIGN="CENTER">Добавьте новый элемент</H2>
  <P ALIGN="CENTER">
   <FORM ACTION="/javaxml2/servlet/javaxml2.AddItemServlet" METHOD="POST">
    <TABLE WIDTH="80%" CELLSPACING="3" CELLPADDING="3" BORDER="3">
     <TR>
      <TD WIDTH="50%" ALIGN="right"><B>Идентификатор элемента:</B></TD>
      <TD><INPUT TYPE="text" NAME="id" /></TD>
     </TR>
     <TR>
      <TD WIDTH="50%" ALIGN="right"><B>Уровень элемента:</B></TD>
      <TD><INPUT TYPE="text" NAME="level" SIZE="1" MAXLENGTH="1" /></TD>
     </TR>
     <TR>
      <TD WIDTH="50%" ALIGN="right"><B>Заголовок:</B></TD>
      <TD><INPUT TYPE="text" NAME="title" SIZE="20" /></TD>
     </TR>
     <TR>
      <TD WIDTH="50%" ALIGN="right"><B>Учитель:</B></B></TD>
      <TD><INPUT TYPE="text" NAME="teacher" /></TD>
     </TR>
     <TR><TD COLSPAN="2" ALIGN="CENTER"><B>Гости:</B></TD></TR>
     <TR>
      <TD COLSPAN="2" ALIGN="CENTER"><INPUT TYPE="text" NAME="guest" /></TD>
     </TR>
     <TR>
      <TD COLSPAN="2" ALIGN="CENTER"><INPUT TYPE="text" NAME="guest" /></TD>
     </TR>
     <TR>
      <TD COLSPAN="2" ALIGN="CENTER"><INPUT TYPE="text" NAME="guest" /></TD>
     </TR>
     <TR>
      <TD COLSPAN="2" ALIGN="CENTER"><INPUT TYPE="text" NAME="guest" /></TD>
     </TR>
     <TR><TD COLSPAN="2" ALIGN="CENTER"><B>Описание:</B></TD></TR>
     <TR>
      <TD COLSPAN="2" ALIGN="CENTER">
       <TEXTAREA NAME="description" COLS="30" ROWS="10"></TEXTAREA>
      </TD>
     </TR>
     <TR>
      <TD COLSPAN="2" ALIGN="CENTER"><INPUT TYPE="submit" value="Добавить" /
></TD>
     </TR>
```

```
    </TABLE>
   </FORM>
  </P>
 </BODY>
</HTML>
```

Среда исполнения сервлетов (например, Tomcat), вероятно, была установлена в процессе чтения предшествующих глав, поэтому я не буду вдаваться в подробности. Перенесите эту форму в одно из веб-приложений, а затем наберите и скомпилируйте код сервлета из примера 15.4.

Пример 15.4. Сервлет AddItemServlet для Castor

```
package javaxml2;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

// Классы Castor
import org.exolab.castor.xml.Marshaller;
import org.exolab.castor.xml.Unmarshaller;

// Классы, порожденные Castor
import javaxml2.castor.Catalog;
import javaxml2.castor.Item;
import javaxml2.castor.types.LevelType;

public class AddItemServlet extends HttpServlet {

    private static final String CATALOG_FILE =
        "c:\\java\\tomcat\\webapps\\javaxml2\\catalog.xml";

    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        PrintWriter out = res.getWriter();
        res.setContentType("text/html");

        // Получаем параметры ввода
        String id = req.getParameterValues("id")[0];
        String levelString = req.getParameterValues("level")[0];
        String title = req.getParameterValues("title")[0];
        String teacher = req.getParameterValues("teacher")[0];
        String[] guests = req.getParameterValues("guest");
        String description = req.getParameterValues("description")[0];
```

```
            // Создаем новый элемент
            Item item = new Item();
            item.setId(id);
            item.setLevel(LevelType.valueOf(levelString));
            item.setTitle(title);
            item.setTeacher(teacher);
            if (guests != null) {
                for (int i=0; i<guests.length; i++) {
                    if (!guests[i].trim().equals("")) {
                        item.addGuest(guests[i]);
                    }
                }
            }
            item.setDescription(description);

            try {
                // Загружаем текущий каталог
                File catalogFile = new File(CATALOG_FILE);
                FileReader reader = new FileReader(catalogFile);
                Catalog catalog =
                    (Catalog)Unmarshaller.unmarshal(Catalog.class, reader);

                // Добавляем элемент
                catalog.addItem(item);

                // Записываем измененный каталог
                FileWriter writer = new FileWriter(catalogFile);
                Marshaller.marshal(catalog, writer);

                out.println("Элемент добавлен.");
            } catch (Exception e) {
                out.println("Ошибка при загрузке/записи в каталог: " +
                e.getMessage());
            } finally {
                out.close();
            }
        }

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

Этот сервлет принимает параметры из формы, приведенной в примере 15.3. Сначала он считывает XML-файл, представляющий актуальный каталог (он называется `catalog.xml` и также находится в каталоге веб-приложения). Затем сервлет должен получить доступ к каталогу; конечно, можно было бы создать объемный код с использованием SAX, но зачем? Castor отлично справляется с работой. Для чтения XML-документа я предоставил объект `FileReader`, а для записи — `FileWriter`. Всю остальную работу берет на себя Castor. Когда сервлет получает зна-

чения из формы, он создает новый экземпляр Item (при помощи классов, созданных Castor) и устанавливает различные значения для этого класса. Вы заметили, что т. к. «level» является пользовательским типом (помните, я говорил об этом раньше?), то для преобразования строкового значения уровня элемента в корректный экземпляр класса LevelType сервлет использует статический метод LevelType.valueOf(String). Это один из незначительных недостатков Castor; поначалу классы для типов, определяемых пользователем, кажутся неуклюжими.

Когда сервлет имеет готовый к использованию экземпляр нового элемента Item, он применяет класс org.exolab.castor.Unmarshaller для получения доступа к каталогу. Проще и быть не может – сервлет передает класс, в который следует производить распаковку, и класс доступа к файлу (как я уже говорил, FileReader). Результатом является объект Java Object, который можно привести к указанному типу класса. Теперь добавить элемент совсем не сложно! Вы работаете с Java, а не XML, и можете просто вызвать метод addItem() и передать ему вновь созданный экземпляр Item. Затем процесс обращается. Метод marshal() класса Marshaller (из того же пакета, что и его родной брат Unmarshaller) применяется для записи экземпляра Catalog в XML-документ при помощи FileWriter. Все очень просто, не правда ли? Когда этот процесс завершен, вы получаете новую запись в XML-файле (вам может потребоваться остановить среду исполнения сервлетов, чтобы получить к нему доступ), которая выглядит примерно так:

```
<item id="CD-KAU-PV99" level="1">
  <title>Parking Lot Pickers, Vol. 3</title>
  <teacher>Steve Kaufman</teacher>
  <guest>Debbie Barbra</guest>
  <guest>Donnie Barbra</guest>
  <description>This video teaches you what to play when the
   singing stops, bluegrass style!</description>
 </item>
```

Вот, пожалуй, и все. Существует еще несколько параметров, используемых с классами Marshaller и Unmarshaller, так что обратитесь к документации по API. Чтобы убедиться, что вы сможете следовать дальше, приведу список нужных файлов и каталогов из веб-приложения, благодаря которым все работает:

```
$TOCAT_HOME/lib/xerces.jar
$TOMCAT_HOME/webapps/javaxml2/addItem.html
$TOMCAT_HOME/webapps/javaxml2/catalog.xml
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/lib/castor-0.9.2.jar
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/AddItemServlet.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/Catalog.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/Catalog.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    CatalogDescriptor.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    CatalogDescriptor$1.class
```

```
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/Item.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    ItemDescriptor.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    ItemDescriptor$1.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    ItemDescriptor$2.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    ItemDescriptor$3.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    ItemDescriptor$4.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    ItemDescriptor$5.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/
    ItemDescriptor$6.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/types/
    LevelType.class
$TOMCAT_HOME/webapps/javaxml2/WEB-INF/classes/javaxml2/castor/types/
    LevelTypeDescriptor.class
```

Еще многое можно сказать о Castor, как и о любом другом из рассмотренных мной пакетов. Это короткое введение поможет вам взяться за дело, а со всем остальным поможет сопутствующая документация. Особый интерес представляет возможность определять отображения, позволяющие, например, преобразовывать XML-элемент с именем «item» в переменную Java с именем «inventory». Это позволяет учитывать различия представления одних и тех же данных в Java и XML, и также может существенно помочь в преобразовании старых классов Java. Представьте себе преобразование старого класса Java в новый XML-формат и последующую распаковку этого нового XML-кода в новый класс Java. Два простых шага – и весь ваш старый код на Java преобразован в новый формат! Ловко, не правда ли? С другой стороны, отсутствие в Castor поддержки последней версии стандарта схем и порождение конкретных классов, а не интерфейсов – это недостаток. Попробуйте поработать с этой системой и посмотрите, понравится ли она вам. У каждой системы из этой главы есть свои достоинства и недостатки, так что вам предстоит решать, какая из них больше всего подходит для вас.

Zeus

Следующая система связывания данных – это Zeus. Этот проект существует в рамках Enhydra (*http://www.enhydra.org*) и основывается на цикле статей, написанных мной для IBM Developer Works (*http://www.ibm.com/developerWorks*). Это был продукт, созданный по необходимости иметь простое решение для связывания данных (в то время Castor казался чрезвычайно сложным решением довольно прямолинейной проблемы). С тех пор Zeus разросся до целого проекта в En-

hydra, и над ним работают несколько человек. Всю историю целиком можно найти на сайте *http://zeus.enhydra.org*. Я коснусь тех же тем, с которыми я имел дело в Castor, так что вы сможете использовать любой из пакетов в ваших приложениях.

Установка

Zeus по-прежнему находится в довольно ранней стадии функционального развития (он уже существует некоторое время, но некоторая серьезная работа все еще продолжается). В результате я рекомендовал использовать бы последнюю версию из CVS, а не скомпилированную версию. Так вы гарантированно получите самые последние и самые лучшие возможности. Полностью инструкции о том, как получить Zeus из CVS, можно найти на веб-сайте проекта. Но если вкратце, то сначала следует обзавестись клиентом CVS (например, взять его с сайта *http://www.cvshome.org*). После установки CVS наберите следующую команду:

```
cvs -d :pserver:anoncvs@enhydra.org:/u/cvs login
```

Введите в качестве пароля «anoncvs». Затем наберите:

```
cvs -d :pserver:anoncvs@enhydra.org:/u/cvs co toolsTech/Zeus
```

Вы получите все, что вам нужно. Перейдите в каталог *Zeus* (созданный CVS) и запустите процесс сборки для получения скомпилированной версии:

```
bmclaugh@GANDALF ~/dev/Zeus
$ ./build.sh
```

Или в Windows:

```
c:\dev\Zeus> build.bat
```

Также нужно собрать примеры, для этого нужно добавить в качестве аргумента «samples»:

```
bmclaugh@GANDALF ~/dev/Zeus
$ ./build.sh samples
```

После этого вы должны получить файл *zeus.jar* в каталоге *build/*. Добавьте его, а также файлы *jdom.jar* и *xerces.jar* из каталога *lib/* в пути к классам. Наконец, если вы собираетесь применять порождение DTD, следует также добавить в пути к классам файл *dtdparser113.jar*. А чтобы использовать примеры, добавьте и сам каталог *build/classes*, чтобы все было действительно просто. Итак, моя переменная `CLASSPATH` выглядит следующим образом:

```
bmclaugh@GANDALF ~/dev/Zeus
$ echo $CLASSPATH
 /dev/Zeus/lib/jdom.jar:/dev/Zeus/lib/xerces.jar:/dev/Zeus/lib/
dtdparser113.jar:
 /dev/Zeus/build/zeus.jar:/dev/Zeus/build/classes
```

Или, в Windows:

```
c:\dev\Zeus> echo %CLASSPATH%
c:\dev\Zeus\lib\jdom.jar;c:\dev\Zeus\lib\xerces.jar;
c:\dev\Zeus\lib\dtdparser113.jar;c:\dev\Zeus\build\zeus.jar;
c:\dev\Zeus\build\classes
```

Вот и все. Когда переменная CLASSPATH инициализирована, мы берем XML-документ, приведенный ранее в этой главе, и приступаем к работе.

Порождение классов

Основное различие между Zeus и такими системами, как Castor и даже JAXB от Sun, заключается в способе порождения классов. Из рис. 15.1 вы помните, что стандартный способ порождения классов заключается в том, что исходный код на Java выдается после чтения набора ограничений. И хотя это удобно, добавить поддержку других типов ограничений, таких как DTD или новые альтернативы схемам – RELAX и TReX, становится сложно. Чтобы обойти это ограничение, в Zeus добавлен промежуточный шаг. Происходит следующее: набор ограничений (в виде схемы, DTD и т. д.) преобразуется в набор эквивалентов Zeus. Эти эквиваленты не связаны с конкретными способами представления ограничений. Другими словами, если вы смоделируете набор ограничений в DTD, а затем снова смоделируете этот же набор ограничений в виде схеме XML, Zeus преобразует их в идентичный набор эквивалентов. Эти эквиваленты применяются для порождения исходного кода на Java. В результате получаем процесс, представленный на рис. 15.3.

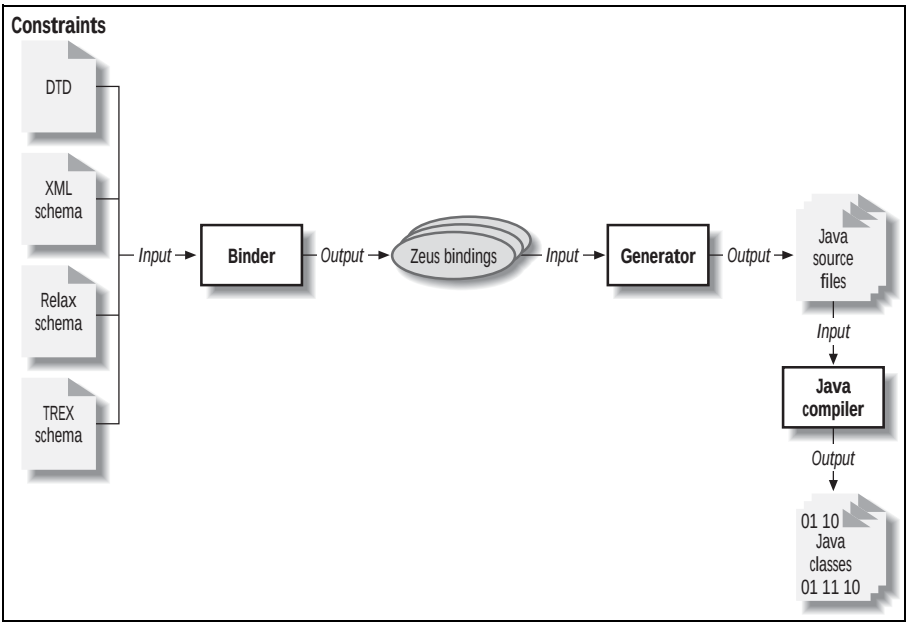

Рис. 15.3. Порождение классов в Zeus

Очень приятно то, что добавление поддержки нового метода создания ограничений, вроде TREX, становится очень простым делом. Можно написать класс, реализующий интерфейс `org.enhydra.zeus.Binder`, принимающий набор ограничений и создающий на их основе эквивалентные представления Zeus. Порождение классов выполняется без нашего участия, и нет необходимости работать с сотней раздражающих операторов `out.write()`. В Zeus входят два готовых класса — `DTDBinder` и `SchemaBinder`, оба из пакета `org.enhydra.zeus.binder`.

За исключением этого архитектурного изменения (оно вас не коснется при использовании стандартных методологий ограничения данных XML), Castor и Zeus сходным образом генерируют классы. В примере 15.5 приведено DTD, ограничивающее XML-документ из примера 15.1. Порождение классов для DTD я продемонстрирую на этом примере.

Пример 15.5. DTD для примера 15.1

```
<!ELEMENT catalog (item+)>
<!ATTLIST catalog
          xmlns   CDATA    #FIXED    "http://www.homespuntapes.com"
>

<!ELEMENT item (title, teacher, guest*, description)>
<!ATTLIST item
          id      CDATA    #REQUIRED
          level   CDATA    #REQUIRED
>

<!ELEMENT title (#PCDATA)>
<!ELEMENT teacher (#PCDATA)>
<!ELEMENT guest (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Zeus настроен так, что он позволяет приложениям вызывать генератор исходного кода, а потому в пределах API не предоставляет статической точки входа для порождения исходного. Но такая точка часто бывает нужна, и притом в состав примеров (помните сборку программы с аргументом «samples»?) входит класс, облегчающий порождение классов. Если пути к классам настроены так, как я показывал ранее, то нужно лишь создать новый каталог под названием «output». Программа по умолчанию помещает порожденные классы в этот каталог (это можно исправить, отредактировав код программы). Так что когда каталог *output* создан, можно запустить следующую команду:

```
C:\javaxml2\build>java samples.TestDTDBinder
    -file=c:\javaxml2\ch14\xml\zeus\catalog.dtd
    -package=javaxml2.zeus
    -quiet=true
```

Чтобы упростить восприятие, я слегка отформатировал строку команды. Я указал файл, для которого происходит порождение кода, пакет,

в который помещается исходный код, а также предписал не отображать отладочную информацию в процессе порождения. Аналогичным образом происходит порождение классов для схемы. Поскольку Zeus реализует последнюю версию рекомендации XML Schema, вам понадобится более новая версия XML Schema; обновите схему из раздела по Castor до текущей версии.

Пример 15.6. Обновление схемы XML из Castor

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

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.homespuntapes.com">
  <element name="catalog">
    <complexType>
      <sequence>
        <element ref="item" minOccurs="1" maxOccurs="unbounded" />
      </sequence>
    </complexType>
  </element>

  <element name="item">
    <complexType>
      <sequence>
        <element name="title" type="string" />
        <element name="teacher" type="string" />
        <element name="guest" type="string" minOccurs="0"
            maxOccurs="unbounded" />
        <element name="description" type="string" />
      </sequence>
      <attribute name="id" type="string" />
      <attribute name="level">
        <simpleType>
          <restriction base="integer">
            <enumeration value="1" />
            <enumeration value="2" />
            <enumeration value="3" />
            <enumeration value="4" />
            <enumeration value="5" />
          </restriction>
        </simpleType>
      </attribute>
    </complexType>
  </element>
</schema>
```

Теперь можно воспользоваться следующей командой для порождения классов:

```
C:\javaxml2\build>java samples.TestSchemaBinder
    -file=c:\javaxml2\ch14\xml\zeus\catalog.xsd
    -package=javaxml2.zeus
    -quiet=true
```

Вывод, полученный для схемы, будет идентичен результатам для DTD.

Внимание

Это небольшое преувеличение. На самом деле вывод будет идентичен, если удастся представить ограничения в DTD и в схеме XML идентичным образом. В данном примере в схеме «level» определяется как целое, в то время как в DTD это должно быть секцией `PCDATA`. В результате при использовании схемы вы получите типизированное поле (типа `int` Java), а в случае DTD – строку (`String`). В этом разделе я пользуюсь DTD, так что покажу вам последний вариант.

В каталоге *output* вы найдете подкаталог *javaxml2*, а в нем подкаталог *zeus* (соответствующий иерархии пакета). Можно взглянуть на находящиеся в нем сгенерированные исходные файлы.

Сразу же видно, что Zeus создал интерфейс (например, *Catalog.java*) и реализацию этого интерфейса по умолчанию (например, *CatalogImpl.java*). Объясняется это тем, что вы можете захотеть иметь собственные классы (возможно даже, что они уже существуют), реализующие интерфейс `Catalog`; в то время как в Castor создаются конкретные классы. В первом случае можно пользоваться собственной реализацией, извлекая при этом пользу из порождения классов. Я часто встречал разработчиков, применяющих Zeus для порождения классов, и затем создающих собственные реализации полученных интерфейсов в виде компонентов Enterprise JavaBeans (EJBs). Эти компоненты EJB затем сохраняются с помощью Zeus, что сделать очень просто (я продемонстрирую это в следующем разделе). Если бы реализации, определяемые пользователем, не были предусмотрены, для решения этой задачи потребовалось бы изменять порожденный исходный код, что могло бы нарушить распаковку и упаковку.

Другое важное отличие, о котором нужно помнить, – это имена методов. Вот, например, интерфейс `Item` (усеченный до простейшей формы):

```
package javaxml2.zeus;

public interface Item {

    public Title getTitle();
    public void setTitle(Title title);

    public Teacher getTeacher();
    public void setTeacher(Teacher teacher);

    public java.util.List getGuestList();
    public void setGuestList(java.util.List guestList);
    public void addGuest(Guest guest);
    public void removeGuest(Guest guest);

    public Description getDescription();
    public void setDescription(Description description);

    public String getLevel();
```

```
    public void setLevel(String level);

    public String getId();
    public void setId(String id);
}
```

Обратите внимание, что применяется система именования JavaBeans, и элементы, которые могут встретиться в документе несколько раз, возвращаются в форме коллекций. В любом случае посмотрите на этот исходный код, а затем скомпилируйте его:

```
javac -d . output/javaxml2/zeus/*.java
```

Теперь можно рассмотреть упаковку и распаковку при помощи Zeus.

Упаковка и распаковка

В плане упаковки и распаковки Zeus схож с системой Castor. Основой всех операций являются два класса: `org.enhydra.zeus.Marshaller` и `org.enhydra.zeus.Unmarshaller`. Как и в Castor, интерес представляют методы `marshal()` и `unmarshal()` из соответствующих классов. Однако тут существуют некоторые различия. Во-первых, классы `Marshaller` и `Unmarshaller` в Zeus не предоставляют статических точек входа. Это сделано намеренно и напоминает пользователю о необходимости указывать пакет, в который распаковывать XML, а также определять методы, которые должны игнорироваться при упаковке. Кроме того, в Zeus применяется ввод и вывод в стиле JAXP, т. е. варианты классов `Source` и `Result`, о которых я говорил в главе 9, предназначенные для механизма связывания данных. Как и JAXP, Zeus предоставляет несколько вариантов этих классов в пакетах `org.enhydra.zeus.source` и `org.enhydra.zeus.result`. Но даже и с этими изменениями процесс оказывается очень простым в обеих системах:

```
File catalogFile = new File("catalog.xml");
FileReader reader = new FileReader(catalogFile);
FileWriter writer = new FileWriter(catalogFile);

// Castor: Распаковка
Catalog catalog =
    (Catalog)org.exolab.castor.xml.Unmarshaller.unmarshal(Catalog.class,
     reader);

// Zeus: Распаковка
StreamSource source = new StreamSource(reader);
org.enhydra.zeus.Unmarshaller unmarshaller = new
org.enhydra.zeus.Unmarshaller();
Catalog catalog = (Catalog)unmarshaller.unmarshal(source);

// Castor: Упаковка
org.exolab.castorMarshaller.marshal(catalog, writer);

// Zeus: Упаковка
StreamResult result = new StreamResult(writer);
```

```
org.enhydra.zeus.Marshaller marshaller = new org.enhydra.zeus.Marshaller();
marshaller.marshal(catalog, result);
```

И хотя при использовании Zeus приходится писать больше кода, данная методология должна показаться более знакомой пользователям JAXP, а это и было целью. Также эта система позволяет реализовывать ввод/вывод для потоков SAX и деревьев DOM при помощи соответствующих реализаций `Source` и `Result`. Взгляните на пример применения Zeus (пример 15.7). Эта программа считывает каталог, заданный первым аргументом командной строки, выводит имена и идентификаторы элементов, изменяет существующий элемент, а затем упаковывает каталог в формат XML. Взаимодействия с пользователем тут очень мало, но вы должны иметь возможность увидеть, как работает Zeus.

Пример 15.7. Применение Zeus для работы с каталогом

```
package javaxml2;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Iterator;
import java.util.List;

// Классы Zeus
import org.enhydra.zeus.Marshaller;
import org.enhydra.zeus.Unmarshaller;
import org.enhydra.zeus.source.StreamSource;
import org.enhydra.zeus.result.StreamResult;

// Классы, порожденные Zeus
import javaxml2.zeus.Catalog;
import javaxml2.zeus.Guest;
import javaxml2.zeus.GuestImpl;
import javaxml2.zeus.Item;
import javaxml2.zeus.ItemImpl;

public class CatalogViewer {

    public void view(File catalogFile) throws IOException {
        FileReader reader = new FileReader(catalogFile);
        StreamSource source = new StreamSource(reader);

        // Преобразование из XML в Java
        Unmarshaller unmarshaller = new Unmarshaller();
        unmarshaller.setJavaPackage("javaxml2.zeus");
        Catalog catalog = (Catalog)unmarshaller.unmarshal(source);

        List items = catalog.getItemList();
        for (Iterator i = items.iterator(); i.hasNext(); ) {
            Item item = (Item)i.next();
            String id = item.getId();
```

```
            System.out.println("Item ID: " + id);
            String title = item.getTitle().getValue();
            System.out.println("Item Title: " + title);

            // Изменение элемента
            if (id.equals("CDZ-SM01")) {
                item.getTitle().setValue("Sam Bush Teaches Mandolin " +
                    "Repertoire and Technique, 2nd edition");
                Guest guest = new GuestImpl();
                guest.setValue("Bela Fleck");
                item.addGuest(guest);
            }
        }

        // Выводим результаты
        FileWriter writer = new FileWriter(new File("newCatalog.xml"));
        StreamResult result = new StreamResult(writer);
        Marshaller marshaller = new Marshaller();
        marshaller.marshal(catalog, result);
    }

    public static void main(String[] args) {
        try {
            if (args.length != 1) {
                System.out.println("Usage: java javaxml2.CatalogViewer " +
                    "[XML Catalog Filename]");
                return;
            }

            // Получаем доступ к каталогу XML
            File catalogFile = new File(args[0]);
            CatalogViewer viewer = new CatalogViewer();
            viewer.view(catalogFile);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Программа считывает указанный файл, а затем отображает идентификатор и заголовок для каждого элемента из каталога. Также она ищет определенный идентификатор («CDZ-SM01») и изменяет этот элемент, обновляя информацию до второго издания (похоже, что все любят вторые издания!). Наконец, она записывает каталог с измененной информацией в новый файл. Попробуйте запустить эту программу; только убедитесь, что уже скомпилированные классы, сгенерированные Zeus, находятся в путях к классам.

После того как вы изучили основы работы механизма связывания данных, сюрпризов осталось не так уж много. Вам лишь понадобится найти систему, соответствующую нуждам вашего приложения. Zeus хорошо выполняет задачу порождения интерфейсов и классов реализации,

что позволяет использовать уже существующие классы, внося в них небольшие изменения. Кроме того, Zeus предусматривает использование DTD, что понравится многим из вас, учитывая применение DTD в многочисленных существующих приложениях, построенных на основе XML. С другой стороны, Zeus появился позже, чем Castor, он обладает меньшей движущей силой, и над этой системой работает меньше разработчиков. Так что посещайте сайт и пользуйтесь этой системой, если она может вам помочь.

JAXB

Последняя, но не худшая система, которую я рассмотрю, – это JAXB (архитектура Java для связывания данных от Sun, Java Architecture for Data Binding). Вас может удивить то, что я рассматриваю JAXB в последнюю очередь, или даже то, что я вообще включил описание других API. Однако продукт, предлагаемый Sun, развивается очень медленно, и на момент написания книги были доступны лишь версии для тестирования. Вероятно, сейчас, когда вы читаете эту книгу, еще не вышла финальная версия JAXB, кроме того, эта архитектура имеет очень ограниченный набор возможностей. Конечно, в будущих версиях функциональность вырастет, но пока Sun пытается лишь создать стабильную версию продукта. В Castor и Zeus вы найдете возможности, которые попросту не доступны пока в JAXB. Поэтому я и показал вам не только систему JAXB. И конечно, я всегда считаю, что программное обеспечение с открытыми исходными кодами – это здорово!

Установка

Для начала нужно загрузить дистрибутив JAXB с сайта Sun *http://java.sun.com/xml/jaxb/*. Также я рекомендую получить документацию и спецификацию.

Внимание

Во время написания этой книги были доступны лишь самые первые версии JAXB. Перед тем как выйдет окончательная версия, некоторые вещи еще могут измениться, хотя это и не часто происходит в мире программного обеспечения EA (early access). Так что учтите, что может потребоваться изменить приведенный здесь код, чтобы работать с ним тогда, когда вы получите версию JAXB. Кроме того, в документации допущено много ошибок (по крайней мере, в самых первых версиях). Я старался избегать их здесь, так что примеры, приведенные в этом разделе, можно использовать для проверки синтаксиса, если при работе с инструкциями JAXB будут возникать ошибки (у меня они возникали).

Загрузив версию, просто поместите два *jar*-файла – *jaxb-xjc-1.0-ea.jar* и *jaxb-rt-1.0-ea.jar* – в пути к классам. Кроме того, для компиляции схем

в JAXB применяет сценарий *xjc*. Я не знаю, почему это не класс Java, но такова жизнь. Если есть желание поработать со сценариями интерпретатора, можно вызвать класс Java вручную. Однако в версии early access отсутствует сценарий, который можно запустить для системы Windows, что понадобилось бы многим из вас. В примере 15.8 приведен сценарий, совместимый с Windows и соответствующий версии для Unix.

Пример 15.8. Сценарий xjc.bat для платформы Windows

```
@echo off

echo JAXB Schema Compiler


if "%JAVA_HOME%" == "" goto errorJVM
if "%JAXB_HOME%" == "" goto errorJAXB

set JAXB_LIB=%JAXB_HOME%\lib
set JAXB_CLASSES=%JAXB_HOME%\classes

echo %JAVA_HOME%\bin\java.exe -jar %JAXB_LIB%\jaxb-xjc-1.0-ea.jar %1 %2 %3 %4
%5
%JAVA_HOME%\bin\java.exe -jar %JAXB_LIB%\jaxb-xjc-1.0-ea.jar %1 %2 %3 %4 %5

goto end

:errorJVM

echo ОШИБКА: Переменная JAVA_HOME не найдена.
echo Пожалуйста, установите переменную JAVA_HOME в значение,
echo соответствующее местоположению JVM, которую вы хотите использовать.
echo Например:
echo  set JAVA_HOME=c:\java\jdk1.3.1

goto end

:errorJAXB

echo ОШИБКА: Переменная JAXB_HOME не найдена.
echo Пожалуйста, установите переменную JAXB_HOME в значение,
echo соответствующее каталогу, в котором установлен JAXB.
echo Например:
echo  set JAXB_HOME=c:\java\jaxb-1.0-ea

:end
```

Поместите этот сценарий в подкаталог *bin* каталога, в котором установлен JAXB, вместе с файлом *xjc*; как видите, я назвал свой файл *xjc.bat*. Наконец, установите обе переменные окружения, `JAVA_HOME` и `JAXB_HOME` (на платформе Unix нужна только переменная `JAVA_HOME`). В моей системе Windows это выглядит так:

```
set JAVA_HOME=c:\java\jdk1.3.1
set JAXB_HOME=c:\javaxml2\jaxb-1.0-ea
```

Теперь можно работать дальше.

Порождение классов

Castor поддерживает лишь XML Schema, а JAXB поддерживает лишь DTD. Поэтому для порождения классов я воспользуюсь DTD из примера 15.5. Однако для порождения классов в JAXB также требуется наличие схемы связывания. Схема связывания указывает JAXB, как преобразовывать ограничения из DTD в класс (или классы) Java. Как минимум, нужно определить корневой элемент в DTD (или несколько корневых элементов, если они определены в DTD). В примере 15.9 приведена простейшая схема связывания для документа *catalog.xml*, используемого на протяжении этой главы.

Пример 15.9. Схема связывания для каталога

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

<xml-java-binding-schema version="1.0ea">
  <element name="catalog" type="class" root="true" />
</xml-java-binding-schema>
```

Приведенный код определяет документ как схему связывания и определяет корневой элемент. Сохраните этот файл под именем *catalog.xjc*.

Имея DTD (файл *catalog.dtd* из раздела о Zeus) и схему связывания можно заняться порождением классов Java. Выполните следующую команду:

```
xjc c:\javaxml2\ch14\xml\jaxb\catalog.dtd
c:\javaxml2\ch14\xml\jaxb\catalog.xjc
```

А в Unix или Cygwin:

```
xjc /javaxml2/ch14/xml/jaxb/catalog.dtd /javaxml2/ch14/xml/jaxb/catalog.xjc
```

Когда команда отработает, вы получите два класса в текущем каталоге: *Catalog.java* и *Item.java*. Они имеют типичный набор методов, как можно было ожидать:

```
public class Item {
    public String getLevel();
    public void setLevel();

    public String getId();
    public void setId(String id);

    public List getGuest();
    public void deleteGuest();
    public void emptyGuest();

    // И т.д...
}
```

Обратите внимание на то, как JAXB работает со списками. Это шаг вперед по сравнению с Castor (на мой взгляд), т. к. JAXB предоставляет доступ к списку гостей. Однако (опять же, это мое мнение) тут не хватает удобства метода addGuest(), предоставляемого Zeus. Вся работа со списком производится путем получения списка и непосредственной его обработки:

```
// В Zeus
item.addGuest(new Guest("Bela Bleck"));

// В JAXB
List guests = item.getGuest();
guests.add("Bela Fleck");
```

Все порожденные классы являются конкретными, что во многом напоминает модель, применяемую в Castor.

Схема связывания предоставляет несколько полезных возможностей (они относятся не к XML, но это не важно). Первая состоит в том, что можно указать, в каком пакете будут находиться порожденные классы. Делается это с помощью элемента options и его атрибута package. Добавьте следующую строку к схеме связывания:

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

<xml-java-binding-schema version="1.0ea">
  <options package="javaxml2.jaxb" />
  <element name="catalog" type="class" root="true" />
</xml-java-binding-schema>
```

Теперь сгенерированный исходный код будет помещен в каталог *javaxml2/jaxb* со структурой, соответствующей иерархии пакета. Затем давайте укажем, что атрибут level элемента item должен быть числом, а не строкой (как это было по умолчанию):

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

<xml-java-binding-schema version="1.0ea">
  <options package="javaxml2.jaxb" />
  <element name="catalog" type="class" root="true" />
  <element name="item" type="class">
    <attribute name="level" convert="int" />
  </element>
</xml-java-binding-schema>
```

Как видите, я сначала добавил объявление для элемента item. Это позволяет работать с атрибутом level при помощи конструкции attribute. Для определения типа данных я указываю желаемый тип (int) при помощи атрибута convert.

Продолжая рассматривать возможности, предоставляемые схемой связывания, нельзя не упомянуть об одной действительно интересной возможности – фактически, можно изменить имя атрибута, опреде-

ленного в DTD. Например, я ненавижу методы вроде getId(). Вместо него я предпочитаю метод getID(), который смотрится гораздо лучше. Так что, фактически, в Java я хочу называть свойство id из DTD как ID. Оказывается, при помощи JAXB сделать это очень просто:

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

<xml-java-binding-schema version="1.0ea">
  <options package="javaxml2.jaxb" />
  <element name="catalog" type="class" root="true" />
  <element name="item" type="class">
    <attribute name="level" convert="int" />
    <attribute name="id" property="ID" />
  </element>
</xml-java-binding-schema>
```

Закончив вносить все эти изменения, снова запустите компилятор схемы (*xjc*). Вы получите измененные классы, о которых я говорил и которые теперь можно скомпилировать:

```
javac -d . javaxml2/jaxb/*.java
```

Если у вас возникли проблемы, убедитесь, что файл *jaxb-rt-1.0-ea.jar* находится в путях к классам.

Схема связывания также предоставляет и другие возможности, о которых я не упомянул. На самом деле, многие из них не документированы, и я обнаружил их, изучая файл *xjc.dtd*, включенный в состав дистрибутива JAXB. Советую вам сделать то же самое, а также прочитать имеющуюся документацию. После порождения классов можно переходить к упаковке и распаковке.

Упаковка и распаковка

Процесс упаковки и распаковки оказывается третьим куплетом той же самой песни, о которой речь идет в этой главе. Поэтому я сразу же перейду к коду, приведенному в примере 15.10.

Пример 15.10. Класс Categorizer

```
package javaxml2;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

// Классы JAXB
import javax.xml.bind.UnmarshalException;
```

```
// Классы, порожденные JAXB
import javaxml2.jaxb.Catalog;
import javaxml2.jaxb.Item;

public class Categorizer {

    public void categorize(File catalogFile) throws IOException,
        UnmarshalException {

        //Преобразование из XML в Java
        FileInputStream fis = new FileInputStream(catalogFile);
        Catalog catalog = new Catalog();
        try {
            catalog = Catalog.unmarshal(fis);
        } finally {
            fis.close();
        }

        // Создаем новые каталоги для различных категорий
        Catalog fingerpickingCatalog = new Catalog();
        Catalog flatpickingCatalog = new Catalog();
        Catalog mandolinCatalog = new Catalog();
        List items = catalog.getItem();
        for (Iterator i = items.iterator(); i.hasNext(); ) {
            Item item = (Item)i.next();
            String teacher = item.getTeacher();
            if ((teacher.equals("Doc Watson")) ||
                (teacher.equals("Steve Kaufman"))) {
                flatpickingCatalog.getItem().add(item);
            } else if (teacher.equals("David Wilcox")) {
                fingerpickingCatalog.getItem().add(item);
            } else if ((teacher.equals("Sam Bush")) ||
                       (teacher.equals("Chris Thile"))) {
                mandolinCatalog.getItem().add(item);
            }
        }

        // Записываем  в XML-формате
        FileOutputStream fingerOutput =
            new FileOutputStream(new File("fingerpickingCatalog.xml"));
        FileOutputStream flatpickOutput =
            new FileOutputStream(new File("flatpickingCatalog.xml"));
        FileOutputStream mandolinOutput =
            new FileOutputStream(new File("mandolinCatalog.xml"));
        try {
            // Проверка действительности каталогов
            fingerpickingCatalog.validate();
            flatpickingCatalog.validate();
            mandolinCatalog.validate();

            // Вывод каталогов
            fingerpickingCatalog.marshal(fingerOutput);
            flatpickingCatalog.marshal(flatpickOutput);
```

```
            mandolinCatalog.marshal(mandolinOutput);
        } finally {
            fingerOutput.close();
            flatpickOutput.close();
            mandolinOutput.close();
        }
    }

    public static void main(String[] args) {
        try {
            if (args.length != 1) {
             System.out.println("Использование: java javaxml2.Categorizer " +
                    "[Каталог XML]");
              return;
            }

            // Получаем доступ к  XML-каталогу
            File catalogFile = new File(args[0]);
            Categorizer categorizer = new Categorizer();
            categorizer.categorize(catalogFile);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

Тут нет ничего интересного – вы это видели уже несколько раз. Однако JAXB кое-что выполняет по-другому. Прежде всего, методы `marshal()` и `unmarshal()` определены в самих порожденных классах, а не в статических классах `Marshaller` и `Unmarshaller`:

```
// Преобразование из XML в Java
FileInputStream fis = new FileInputStream(catalogFile);
Catalog catalog = new Catalog();
try {
    catalog = Catalog.unmarshal(fis);
} finally {
    fis.close();
}
```

Порожденные классы предоставляют статические методы, предназначенные для упаковки и распаковки. Эти статические методы возвращают экземпляр класса с данными из переданного файла. Однако *необходимо убедиться, что это возвращаемое значение присвоено реализованной переменной!* Следующий код приводит к очень обидной ошибке:

```
// Преобразование из XML в Java
FileInputStream fis = new FileInputStream(catalogFile);
Catalog catalog = new Catalog();
try {
```

```
        catalog.unmarshal(fis);
    } finally {
        fis.close();
    }
```

Обратите внимание на выделенную строку: пытаясь обратиться к переменной экземпляра `catalog` после данного отрывка кода, вы не получите никаких данных независимо от того, что находилось в переданном XML-файле. Дело в том, что метод `unmarshal()` статический и возвращает самостоятельный экземпляр (live instance) класса `Catalog`; в данном примере этот экземпляр нигде не сохраняется. Это может досаждать довольно сильно, так что будьте внимательны! Именно из-за этого в Castor и Zeus используются внешние классы `Marshaller` и `Unmarshaller`.

В данном примере, получив экземпляр каталога XML, я обхожу его в цикле. В зависимости от учителя код добавляет элемент в один из трех новых каталогов: *flat picking*, *finger picking* или *mandolin*. Затем каждый из этих каталогов опять упаковывается в новый XML-документ. В качестве примера покажу, что я получил в документе *mandolinCatalog.xml*:

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

<catalog>
  <item level="3" id="VD-THI-MN01">
    <title>Essential Techniques for Mandolin</title>
    <teacher>Chris Thile</teacher>
    <description>Here's a lesson that will thrill and inspire mandolin
       players at
    all levels.</description></item>
  <item level="4" id="CDZ-SM01">
    <title>Sam Bush Teaches Mandolin Repertoire and Techniques</title>
    <teacher>Sam Bush</teacher>
    <description>Learn complete solos to eight traditional and orignal tunes,
    each one jam-packed with licks, runs, and musical variations.</
description>
</item></catalog>
```

Пробелы, как обычно, добавлены для читаемости, так что переносы строк у вас могут отличаться. Применять упаковку и распаковку в JAXB очень просто. Если справиться с проблемой, связанной со статическими методами, о которой я говорил, то этот процесс практически идентичен тому, который можно наблюдать в двух других системах.

И хотя я советую посмотреть на JAXB уже сейчас, стоит поостеречься и не использовать его в реальных приложениях до того, как выйдет окончательная версия. В нем по-прежнему существует несколько недокументированных возможностей, которые вполне могут измениться до выхода окончательной версии. Кроме того, JAXB пока не поддерживает упаковку произвольных объектов (порожденных системой, от-

личной от JAXB). В других системах такая возможность есть, и она может оказаться критичной для ваших приложений. Также JAXB, как уже говорилось, не поддерживает XML Schema и пространства имен. Но с другой стороны, Sun, очевидно, добавит в JAXB множество возможностей, так что можно ожидать, что в ближайшее время JAXB будет «доведен до ума».

Как бы то ни было, читатели получили достаточно информации о возможностях связывания данных. Все подробности о конкретных системах вы можете найти самостоятельно, но даже основ, приведенных в этой главе, должно хватить для того, чтобы заставить работать любой упомянутый здесь проект или даже все проекты. Выбирая продукт, с которым вы будете работать, обязательно узнайте о том, что в нем работает, а что нет – это действительно важно для дальнейшей разработки, особенно при разработке программного обеспечения с открытым исходным кодом.

Что дальше?

Мы переходим к заключительным страницам книги, так что вас не должно удивить, что я собираюсь подводить итоги. В следующей главе я собираюсь подурачиться и расскажу, что, на мой взгляд, будет заслуживать внимания в ближайшие год-два. Глава 16 «Взгляд в будущее» включена в книгу по двум причинам. Первая причина очевидна: я хочу, чтобы вы опередили других. Вторая причина более интересна и заключается в том, чтобы показать, как быстро все меняется. Это особенно верно в мире Java и XML, и, уверяю вас, примерно через год я посмеюсь над некоторыми из предположений, сделанных в следующей главе. В любом случае это поможет вам задуматься над тем, что ожидается на горизонте, и подготовит вас к появлению новых технологий.

16

- *XLink*
- *XPointer*
- *Отображения для схем XML*
- *И прочее...*
- *Что дальше?*

Взгляд в будущее

Ну вот и подошло время завершать прогулку по миру Java и XML. Надеюсь, вам не было скучно. Но прежде чем вы останетесь наедине с размышлениями о многообразии открывшихся вам возможностей, заглянем ненадолго в магический хрустальный шар. Как и любой хороший программист, я всегда стараюсь предугадывать развитие технологий и использовать все преимущества таких прогнозов. Как правило, это подразумевает достаточно глубокие познания во всех технологиях, поэтому мне не составляет труда набрать скорость, если в чем-то мои прогнозы не оправдываются. В этой главе я собираюсь рассказать о некоторых интересных вещах, появляющихся на горизонте, и предоставить о них дополнительную информацию. Охотно допускаю, что некоторые из моих догадок будут совершенно неверными, другие же могут действительно оказаться стоящими. Ознакомьтесь с ними и готовьтесь действовать, если увидите, что предложенные технологии могут применяться в ваших приложениях.[1]

XLink

Первым в моем списке восходящих звезд мира XML стоит XLink. XLink определяет механизм связывания для XML, позволяющий ссылаться на другие документы. Тем, кто знаком с HTML, это может напомнить элемент «a», к которому вы уже привыкли:

```
<a href="http://www.nickelcreek.com">Check out Nickel Creek!</a>.
```

[1] Многие из разделов этой главы либо полностью, либо частично основаны на статьях и советах, написанных мной для IBM DeveloperWorks (*http://www.ibm.com/developer*). Спасибо Нэнси Данн и коллегам из IBM за то, что они разрешили мне обновить и переиздать часть этих статей.

Однако XLink предлагает гораздо больше, чем просто однонаправленное связывание. При помощи XLink можно создавать двунаправленные ссылки, определять правила их обработки и, что самое важное, осуществлять связывание из любого элемента XML (а не только из элемента «a»). По всем этим причинам XLink заслуживает упоминания.

В примере 16.1 приведен простой XML-документ, содержащий описание некоторых из моих гитар.

Пример 16.1. XML-документ, использующий XLink

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

<guitars xmlns="http://www.newInstance.com/about/guitars"
         xmlns:xlink="http://www.w3.org/1999/xlink">
 <guitar luthier="Bourgeois"
          xlink:type="simple"
          xlink:href="http://www.newInstance.com/about/guitars/bourgeoisOM">
  <descripton xlink:type="simple"
              xlink:href="http://www.newinstance.com/pics/
bougOM_front_full.jpg"
              xlink:actuate="onLoad" xlink:show="embed">
  А это настоящая красавица с небольшой декой. Хотя это оркестровая гитара,
  я играю на ней музыку "кантри" медиатором и извлекаю замечательные звуки
  без него.
  </description>
 </guitar>
 <guitar luthier="Bourgeois"
          xlink:type="simple"
          xlink:href="http://www.newInstance.com/about/guitars/
bourgeoisD150">
  <descripton xlink:type="simple"
              xlink:href="http://www.newinstance.com/pics/
bougd150_con_rim2.jpg"
              xlink:actuate="onLoad" xlink:show="embed">
   А это Святой Грааль в процессе сотворения. Дэйна Буржуа прямо сейчас
   работает над этим чудом из бразильского красного дерева и адирондака...
   вы поймете, что гитара закончена, когда от нескольких аккордов в вашем
   доме повылетают стекла!
  </description>
 </guitar>
</guitars>
```

Во-первых, обратите внимание, что я упоминаю пространство имен XLink, чтобы документ имел доступ к атрибутам и возможностям XLink. Во-вторых, я собираюсь рассматривать только ссылки XLinks типа «simple», что отражено в атрибуте `xlink:type`. Дело в том, что поддержка XLink-связывания в броузерах минимальна и существует лишь в Mozilla и Netscape 6 (у меня не было возможности проверять наличие такой поддержки в IE 6.0, но в 5.5 ее не было). А поэтому лучше придерживаться базовых возможностей.

Если отбросить формальности, XLink просто требует использования некоторых атрибутов для элементов, имеющих ссылки. Возьмите элемент `guitar` из моего документа. Он определяет мастера для каждой гитары (т. е. создателя гитары). Я уже говорил о назначении атрибута `xlink:type`, имеющего значение «simple». Следом мы задаем URL, на который необходимо сослаться при помощи XLink. А для задания URL используется атрибут `xlink:href`. Пока что все очень сильно похоже на HTML. Ничего особенного, верно? По умолчанию (считаем, что броузер поддерживает эту возможность, конечно же) содержимое текущего окна должно заменяться при переходе по ссылке. Если цель ссылки необходимо открыть в новом окне, следует добавить атрибут `xlink:show` со значением «new»; по умолчанию его значение равно «replace», что соответствует обычному поведению в HTML.

Конечно же, тут рассмотрены лишь основы связывания. Гораздо интереснее становится, когда нужно получить доступ к удаленным ресурсам, например добавить ссылки на фотографии. Взгляните на элемент `description`; в нем значение атрибута `xlink:show` установлено в «embed». Ресурс – в данном случае файл с изображением описываемой гитары – должен стать частью страницы. Броузеру, совместимому с XLink, предписывается вставить указанный документ в исходный XML-документ. По-настоящему интересные возможности открываются, когда мы понимаем, что так можно ссылаться не только на изображение, но и на *другой XML-документ*.

Продолжаем рассмотрение. Мы можем указать, *когда* следует отображать ресурс. Это делается при помощи атрибута `xlink:actuate`. Он определяет, когда ресурс будет прочитан и когда отображен. Если значение равно «onLoad», как в примере 16.1, ресурс должен загружаться при загрузке документа. Значение «onRequest» означает, что ресурс не будет отображаться до тех пор, пока ссылка не будет активирована. Это помогает снизить трафик, обеспечивая пользователю возможность просмотра только тех ресурсов, которые его заинтересуют.

Несомненно, XLink может сильно повлиять на следующее поколение XML-документов. Полную спецификацию можно найти по адресу *http://www.w3.org/TR/xlink*. Также рекомендуется следить за выходом новых версий броузеров, чтобы знать, когда появится полная поддержка XLink.

XPointer

XPointer – это еще одна технология связывания, фактически построенная на возможностях XLink. XLink, хоть и полезен сам по себе, позволяет лишь ссылаться на другой документ. Однако зачастую требуется сослаться на определенную часть другого документа. Это очень распространенная задача и напоминает она применение якорей в HTML. Выполнима она при помощи технологии XPointer, основанной

на XLink; эти технологии естественным образом подходят одна другой и предназначены для совместной работы. Прежде всего, следует изучить документ, на который вы собираетесь ссылаться. Если это возможно, проверьте, применяются ли в документе `id`. Это очень сильно упростит создание ссылок. В примере 16.2 приведен список некоторых гитар, сделанных Дэйной Буржуа (Dana Bourgeois), и каждый тип в этом примере имеет идентификатор.

Пример 16.2. Список гитар Буржуа

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

<guitarTypes xmlns="http://www.bourgeoisguitars.com">
 <type model="OM" ID="OM">
  <picture url="http://www.bourgeoisguitars.com/images/vvOM.jpg"/>
  <description>Оркестровая модель с небольшой декой.</description>
 </type>
 <type model="D" ID="D">
  <picture
   url="http://www.bourgeoisguitars.com/images/ricky%20skaggs%20model.jpg"/>
  <description>Мощная гитара "кантри" в классической форме "дредноута".</
description>
 </type>
 <type model="slopeD" ID="slopeD">
  <picture
   url="http://www.bourgeoisguitars.com/images/
slope%20d,%20custom%20version.jpg"/>
  <description>
    "Дредноут" с уменьшенным корпусом, идеальный для вокального
аккомпанемента.
  </description>
 </type>
</guitarTypes>
```

Предположим, что этот документ доступен по адресу *http://www.bourgeoisguitars.com/guitars.xml*. Вместо того чтобы ссылаться на весь документ целиком, что не слишком эффективно, XPointer позволяет создавать ссылки на определенные части документа. Помните атрибут `xlink:href`? Значение этого атрибута является целью ссылки XLink. Но можно добавить символ #, а затем выражения XPointer в эти URL-адреса. Например, выражение `xpointer(id("slopeD"))` указывает на элемент документа, идентификатор которого – «slopeD». Таким образом, чтобы сослаться на XML-код из примера 16.2, а затем на модель гитары Slope D, описанную в этом документе, нужно воспользоваться URL-адресом *http://www.bourgeoisguitars.com/guitars.xml#xpointer(id(«slopeD»))*. Достаточно просто. Ниже приводится измененная версия XML-документа с описанием моих гитар из раздела XLink (пример 16.1). В этой версии присутствует ряд ссылок XPointer. (Простите за странное форматирование – текста очень много, и он не помещается на отдельных строках.) Взгляните на пример 16.3.

Пример 16.3. Описание моих гитар в формате XML с использованием XPointer

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

<guitars xmlns="http://www.newInstance.com/about/guitars"
         xmlns:xlink="http://www.w3.org/1999/xlink">
 <guitar luthier="Bourgeois"
          xlink:type="simple"
          xlink:href=
         "http://www.bourgeoisguitars.com/
guitars.xml#xpointer(id(&apos;OM&apos;))">
  <descripton xlink:type="simple"
              xlink:href="http://www.newinstance.com/pics/
bougOM_front_full.jpg"
              xlink:actuate="onLoad" xlink:show="embed">
   А это настоящая красавица с небольшой декой.  Хотя это оркестровая гитара,
   я играю на ней музыку "кантри" медиатором и извлекаю замечательные звуки
   без него.

  </description>
 </guitar>
 <guitar luthier="Bourgeois"
         xlink:type="simple"
         xlink:href=
          "http://www.bourgeoisguitars.com/
guitars.xml#xpointer(id(&apos;D&apos;))">
  <descripton xlink:type="simple"
              xlink:href="http://www.newinstance.com/pics/
bougd150_con_rim2.jpg"
              xlink:actuate="onLoad" xlink:show="embed">
  А это Святой Грааль в процессе сотворения. Дэйна Буржуа прямо сейчас
  работает над этим кантри-чудом из бразильского красного дерева и
  адирондака... Вы поймете, что гитара закончена, когда от нескольких
  аккордов в вашем доме повылетают стекла!

  </description>
 </guitar>
</guitars>
```

Теперь этот документ может ссылаться на XML-данные по гитарам непосредственно от Дэйны Буржуа. Если он изменит эту информацию, беспокоиться все равно будет не о чем – мой документ останется актуальным, т. к. он просто содержит ссылку на эту информацию. Обратите внимание, что мне приходится экранировать кавычки в выражении XPointer при помощи сущности `&`, но не амперсанда (&). Однако это приводит к созданию более длинных URL для ссылок. Длинные URL, по моему опыту, приводят к досадным опечаткам (и некрасивому форматированию в книге!). К счастью, XPointer поддерживает сокращенную форму для ссылок на элемент с тегом `ID`. Вместо того чтобы применять форму `xpointer(id("D"))`, можно просто использовать значение идентификатора в качестве цели. В нашем случае – просто

«D». Благодаря этому я могу привести пример 16.3 к виду, представленному в примере 16.4, с более понятным синтаксисом ссылок.

Пример 16.4. Использование сокращенного формата XPointer для упрощения примера 16.3

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

<guitars xmlns="http://www.newInstance.com/about/guitars"
         xmlns:xlink="http://www.w3.org/1999/xlink">
 <guitar luthier="Bourgeois"
          xlink:type="simple"
          xlink:href="http://www.bourgeoisguitars.com/guitars.xml#OM" >
  <descripton xlink:type="simple"
              xlink:href="http://www.newinstance.com/pics/
bougOM_front_full.jpg"
              xlink:actuate="onLoad" xlink:show="embed">
      А это настоящая красавица с небольшой декой.  Хотя это оркестровая
      гитара, я играю на ней кантри-музыку медиатором и извлекаю
      замечательные звуки без него.
  </description>
 </guitar>
 <guitar luthier="Bourgeois"
         xlink:type="simple"
         xlink:href="http://www.bourgeoisguitars.com/guitars.xml#D" >
  <descripton xlink:type="simple"
              xlink:href="http://www.newinstance.com/pics/
                          bougd150_con_rim2.jpg"
              xlink:actuate="onLoad" xlink:show="embed">
   А это Святой Грааль в процессе сотворения. Дэйна Буржуа прямо сейчас
   работает над этим кантри-чудом из бразильского красного дерева и
   адирондака... вы поймете, что гитара закончена, когда от нескольких
   аккордов в вашем доме повылетают стекла!
  </description>
 </guitar>
</guitars>
```

Помимо указания ссылок непосредственно на элементы, разрешены также ссылки и на элементы по *относительному* пути. В качестве примера элементы `description` в примере 16.5 изменены и ссылаются на изображение, упомянутое в файле *bourgeois.xml* из примера 16.2.

Примечание

Чтобы этот длинный URL поместился на странице книги, я сократил *http://www.bourgeoisguitars.com* до *http://bg.com*. Этот URL не является действительным, но вполне подходит для примера.

Пример 16.5. Применение относительных ссылок

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

<guitars xmlns="http://www.newInstance.com/about/guitars"
```

```
        xmlns:xlink="http://www.w3.org/1999/xlink">
<guitar luthier="Bourgeois"
         xlink:type="simple"
         xlink:href=
"http://bg.com/guitars.xml#xpointer(id(&apos;OM&apos;))/
  descendant::picture[@url]">
 <descripton xlink:type="simple"
             xlink:href="http://www.newinstance.com/pics/
              bougOM_front_full.jpg"
             xlink:actuate="onLoad" xlink:show="embed">
  А это настоящая красавица с небольшой декой.  Хотя это оркестровая гитара,
  я играю на ней кантри-музыку медиатором и извлекаю замечательные звуки без
  него.
 </description>
</guitar>
<guitar luthier="Bourgeois"
        xlink:type="simple"
        xlink:href=
 "http://bg.com/guitars.xml#xpointer(id(&apos;D&apos;))/
   descendant::picture[@url]" >
 <descripton xlink:type="simple"
             xlink:href="http://www.newinstance.com/pics/
               bougd150_con_rim2.jpg"
             xlink:actuate="onLoad" xlink:show="embed">
 А это Святой Грааль в процессе сотворения. Дэйна Буржуа прямо сейчас
 работает над этим кантри-чудом из бразильского красного дерева и
 адирондака... вы поймете, что гитара закончена, когда от нескольких
 аккордов в вашем доме повылетают стекла!
 </description>
</guitar>
</guitars>
```

Вы видите, что как только найден элемент, на который указывает идентификатор, сразу же начинается поиск потомка этого элемента (задаваемого ключевым словом `descendant`) с именем «picture». Затем конечной целью ссылки становится атрибут «url» найденного элемента. Разумеется, это слишком короткое объяснение для первого раза, но если разобрать пример шаг за шагом, все встанет на свои места. Более подробно о богатых возможностях, предоставляемых технологией XPointer, можно узнать из спецификации XPointer по адресу *http://www.w3.org/TR/xptr*.

Примечание

Обратите внимание, что не использована сокращенная форма ссылок, о которых говорилось в последнем разделе. Дело в том, что такая форма допустима лишь для прямых ссылок. Без полной формы нет и более сложных ссылок (как те, что связаны с обходом дочерних элементов в примере 5).

XLink и XPointer могут коренным образом изменить процесс создания и связывания документов XML. Думаю, что мы увидим широкую поддержку этих технологий в Java API, как только поддержка спецификаций будет полностью реализована в броузерах; так что следите за ними.

Отображения для схем XML

Что касается вещей, связанных с Java, то я ожидаю увидеть один важный аспект XML-программирования – набор типов данных, определенных в Java и представляющих конструкции схем XML. В какой-то степени такое отображение похоже на частную реализацию для HTML в DOM Level 2, о которой шла речь в главе 6. На мой взгляд, это будет невероятно полезная технология. Поскольку схема XML сама по себе является XML-документом, ее можно анализировать и обрабатывать так же, как и любой другой документ. Однако если начать обрабатывать схему XML как старый добрый XML-документ, это может оказаться не очень приятным делом. Например, нельзя получить определение элемента, а значит, и выяснить, представляет ли он сложный тип. Вместо этого придется найти элемент, узнать, имеет ли он дочерние элементы, определить, есть ли среди дочерних элементов элемент под названием `complexType` и т. д. Все гораздо хуже, если в дело идут последовательности (sequences). Внезапно определение сложного типа оказывается *двумя* уровнями глубже.

Я же ожидаю увидеть (на самом деле до меня уже дошли соответствующие слухи) грамматику и набор объектов Java, специально построенных таким образом, чтобы соответствовать типам данных схем XML. Думаю, технология будет построена на основе таких существующих объектных API, как DOM или JDOM. Пока же, чтобы не быть голословными, предположим, что в DOM Level 4 или JDOM 1.1 такие объекты определены. Тогда можно будет увидеть подобный код:

```
// ЭТОТ КОД ГИПОТЕТИЧЕСКИЙ
XSDDocumentParser schemaParser =
    new org.apache.xerces.parsers.XSDParser();
parser.parse(mySchema);
XSDDocument doc = parser.getXSDDocument();
```

Теперь вместо того чтобы работать с корневыми элементами и атрибутами в XML, вы будете иметь дело с этим документом (где все классы имеют префикс XSD – от *XML Schema Datatypes*), использующим концепции схемы, как показано ниже:

```
// Получаем "корневой" элемент
XSDSchema schema = doc.getXSDSchema();

// Получаем пространство имен для этого документа
String targetNamespaceURI = schema.getTargetNamespace().getURI();
```

```
// Получаем определения некоторых элементов
XSDElementDef def = schema.getElementDef("movie");
if (def.isComplexType()) {
    List attributeDefs = def.getAttributeDefs();
    List nestedElementDefs = def.getElementDefs();
} else {
    XSDType elementType = def.getType();
}
```

Очевидно, что код несколько запутан, поскольку я выдумываю синтаксис на ходу. Тем не менее, понятно, что мой код на Java работает со схемой XML и использует преимущества семантики схемы. Я работаю не с базовой XML-семантикой (хотя если бы эти классы расширяли основные классы DOM или JDOM, можно было бы работать таким образом), но использую информацию о допустимых схемах, которая содержится в спецификации XML Schema, чтобы более эффективно работать с документом схемы. Надеюсь, в третье издание этой книги уже войдут подробности о таком весьма полезном API.

И прочее...

Еще много что я хочу рассмотреть и о многом могу рассказать. Однако если бы я привел здесь все, то заканчивать книгу пришлось бы еще в течение полугода, но тогда бы наступило время следующего переиздания! Вместо этого ограничимся кратким описанием технологий. Ниже перечислено то, что может оказаться полезным (или бесполезным), но эти технологии, несомненно, повлияют на Java и XML в течение следующих нескольких лет:

- Scalable Vector Graphics (SVG) и Apache Batik (*http://xml.apache.org*)
- MathML (Язык математической разметки, приложение XML)
- Спецификации, связанные с ebXML и построенные на основе этого стандарта (*http://www.ebXML.org*)
- Xerces 2.0 (*http://xml.apache.org*)
- JAXM, Java API for XML Messaging (*http://java.sun.com/xml/jaxm*)
- JAXRPC, Java API for XML-based RPC (*http://java.sun.com/xml/jaxrpc*)

Вот таков краткий перечень того, что станет актуальным через год (если не раньше). Вероятно, в следующем издании я откажусь от некоторых из них, но такова жизнь на скоростном шоссе, которым, вне всякого сомнения, является XML.

Что дальше?

А дальше приложения, предметный указатель, немного информации обо мне и об обложке и, вероятно, реклама других отличных книг от O'Reilly.

На самом деле, мы рассмотрели уже достаточно информации. Подождите несколько дней, пока прочитанный материал уляжется в голове, а затем применяйте полученные сведения в работе или в собственных проектах – это поможет довести до совершенства ваши знания XML. Вскоре вы станете знатоком XML и поймете, что ценность ваших приложений возрастает, т. к. они становятся более гибкими, конфигурируемыми и производительными. Ну и наконец, ценность ваших знаний для вашего начальника (и множества потенциальных начальников из других компаний!) невероятно возрастет, когда вы начнете создавать легкие в сопровождении и эффективные приложения. Развлекайтесь и оставайтесь расширяемыми. До встречи в Интернете!

A

Справочник по API

Данное приложение представляет собой краткий справочник по четырем низкоуровневым интерфейсам прикладного программирования Java и XML – SAX, DOM, JDOM и JAXP. Каждому из API посвящен самостоятельный раздел.

SAX 2.0

SAX 2.0 обеспечивает последовательный разбор XML-документа. SAX API, описанный в главе 3 «SAX» и в главе 4 «Расширенный SAX», определяет набор интерфейсов, которые могут быть реализованы и методы которых будут вызываться в процессе анализа данных XML. Далее подробно описаны пакеты SAX. Входящие в их состав классы и интерфейсы перечислены в алфавитном порядке. Большинство методов вспомогательных классов пакета `org.xml.sax.helpers` являются реализациями интерфейсов, уже определенных в базовом пакете SAX (`org.xml.sax`).

Пакет org.xml.sax

Этот пакет содержит базовые интерфейсы и классы SAX 2.0. Все определенные здесь интерфейсы, кроме `XMLReader` и `Attributes`, предназначены для реализации разработчиками на Java. Интерфейсы `XMLReader` и `Attributes` должны быть реализованы в программном продукте, производящем синтаксический анализ XML. Кроме того, пакет определяет несколько исключений, которые могут быть сгенерированы методами SAX. Некоторые из определенных здесь интерфейсов являются составной частью SAX версии 1.0, а также альфа-версии 2.0 и на сегодняшний день являются устаревшими.

AttributeList (устарел)

Этот интерфейс был определен в SAX 1.0 и в настоящее время является устаревшим. В реализациях SAX 2.0 вместо интерфейса AttributeList следует использовать интерфейс Attributes.

```
public interface AttributeList {
    public abstract int getLength ();
    public abstract String getName (int i);
    public abstract String getType (int i);
    public abstract String getValue (int i);
    public abstract String getType (String name);
    public abstract String getValue (String name);
}
```

Attributes

Этот интерфейс предоставляет доступ к атрибутам элементам XML. Он передается методам обратного вызова, связанным с началом элемента (метод startElement() интерфейса ContentHandler), и в чем-то сходен с Java-классом Vector. Предусмотрена возможность получения количества имеющихся атрибутов, а также различных форматов имен атрибутов (локального, комбинации префикса пространства имен и URI, а также необработанного имени) и их значений. Кроме того, имеются методы, предназначенные для поиска индекса атрибута по заданному имени. Главное отличие между этим интерфейсом и его предшественником AttributeList заключается в том, что этот интерфейс поддерживает пространства имен.

```
public interface Attributes {
    public abstract int getLength();
    public abstract String getURI(int index);
    public abstract String getLocalName(int index);
    public abstract String getQName(int index);
    public abstract String getType(int index);
    public abstract String getValue(int index);
    public int getIndex(String uri, String localName);
    public int getIndex(String qName);
    public abstract String getType(String uri, String localName);
    public abstract String getType(String qName);
    public abstract String getValue(String uri, String localName);
    public abstract String getValue(String qName);
}
```

ContentHandler

Этот интерфейс определяет методы обратного вызова, которые работают с содержимым анализируемого XML-документа. В частности, существует возможность получать уведомления о начале (предшествует всем остальным событиям) и завершении синтаксического анализа (следует за всеми остальными событиями), об инструкциях обработки,

а также о сущностях, которые могут пропускаться анализаторами, не выполняющими проверку действительности. Методы обратного вызова, связанные с элементами, поддерживают работу с отображениями пространств имен.

```
public interface ContentHandler {
    public void setDocumentLocator(Locator locator);
    public void startDocument() throws SAXException;
    public void endDocument() throws SAXException;
    public void startPrefixMapping(String prefix, String uri)
        throws SAXException;
    public void endPrefixMapping(String prefix)
        throws SAXException;
    public void startElement(String namespaceURI, String localName,
                             String qName, Attributes atts)
        throws SAXException;
    public void endElement(String namespaceURI, String localName,
                             String qName)
        throws SAXException;
    public void characters(char ch[], int start, int length)
        throws SAXException;
    public void ignorableWhitespace(char ch[], int start, int length)
        throws SAXException;
    public void processingInstruction(String target, String data)
        throws SAXException;
    public void skippedEntity(String name)
        throws SAXException;
}
```

DocumentHandler

Данный интерфейс был определен в SAX 1.0 и в настоящее время является устаревшим. В реализациях SAX 2.0 вместо интерфейса `DocumentHandler` следует использовать интерфейс `ContentHandler`.

```
public interface DocumentHandler {
    public abstract void setDocumentLocator(Locator locator);
    public abstract void startDocument() throws SAXException;
    public abstract void endDocument() throws SAXException;
    public abstract void startElement(String name, AttributeList atts)
            throws SAXException;
    public abstract void endElement(String name)
            throws SAXException;
    public abstract void characters(char ch[], int start, int length)
            throws SAXException;
    public abstract void ignorableWhitespace(char ch[], int start, int
            length)
            throws SAXException;
    public abstract void processingInstruction (String target, String data)
            throws SAXException;
}
```

DTDHandler

Данный интерфейс определяет методы обратного вызова, которые вызываются в процессе анализа DTD. Обратите винимание, что этот интерфейс предоставляет информацию не об ограничениях в DTD, а о ссылках на неанализируемые сущности и объявления NOTATION, описывающие компоненты, которые обычно не подлежат синтаксическому анализу.

```
public interface DTDHandler {
    public abstract void notationDecl(String name, String publicId,
                                      String systemId)
                 throws SAXException;
    public abstract void unparsedEntityDecl(String name, String publicId,
                                      String systemId,
                                      String notationName)
            throws SAXException;
}
```

EntityResolver

Этот интерфейс позволяет приложениям вмешиваться в процесс интерпретации внешних сущностей, например, в случае XML-документа, который ссылается на DTD или таблицу стилей. Реализация этого интерфейса позволяет возвращать автору вызова измененный или даже совершенно другой объект SAX InputSource. Передача в программу значения null предписывает выполнить стандартную интерпретацию для доступа к указанному ресурсу по соответствующему URI.

```
public interface EntityResolver {
    public abstract InputSource resolveEntity(String publicId,
                                    String systemId)
              throws SAXException, IOException;
}
```

ErrorHandler

Данный интерфейс позволяет организовать специальную обработку трех типов ошибочных ситуаций, которые могут возникнуть в процессе синтаксического анализа документа XML. Каждый из методов получает в качестве параметра объект типа SAXParseException, содержащий информацию об ошибке, приведшей к вызову метода. Исключение SAXException позволяет полностью остановить процесс синтаксического анализа.

```
public interface ErrorHandler {
    public abstract void warning(SAXParseException exception)
              throws SAXException;
    public abstract void error(SAXParseException exception)
              throws SAXException;
```

```
    public abstract void fatalError(SAXParseException exception)
              throws SAXException;
}
```

HandlerBase

Этот вспомогательный класс содержит пустые реализации всех основных интерфейсов-обработчиков SAX 1.0, и может быть расширен с целью быстрого добавления обработчиков путем переопределения методов, реализующих функциональность, определяемую приложением. Этот класс появился в SAX 1.0 и в настоящее время является устаревшим. В реализациях SAX 2.0 вместо класса `HandlerBase` следует использовать класс `org.xml.sax.helpers.DefaultHandler`.

```
public class HandlerBase implements EntityResolver, DTDHandler,
                                   DocumentHandler, ErrorHandler {

        // Реализация EntityResolver
        public InputSource resolveEntity(String publicId,
        String systemId);

        // Реализация DTDHandler
        public void notationDecl(String name, String publicId,
                            String systemId);
    public void unparsedEntityDecl(String name, String publicId,
                        String systemId, String notationName);

        // Реализация DocumentHandler
        public void setDocumentLocator(Locator locator);
    public abstract void startDocument() throws SAXException;
    public abstract void endDocument() throws SAXException;
    public abstract void startElement(String name, AttributeList atts)
              throws SAXException;
    public abstract void endElement(String name)
              throws SAXException;
    public abstract void characters(char ch[], int start, int length)
              throws SAXException;
    public abstract void ignorableWhitespace(char ch[], int start,
                                                        int length)
              throws SAXException;
    public abstract void processingInstruction(String target,
                                            String data)
              throws SAXException;

       // Реализация ErrorHandler
    public abstract void warning(SAXParseException exception)
              throws SAXException;
    public abstract void error(SAXParseException exception)
              throws SAXException;
    public abstract void fatalError(SAXParseException exception)
              throws SAXException;
}
```

InputSource

Данный класс инкапсулирует всю информацию о ресурсе, используемом при обработке XML. Информация может ограничиваться одной строкой или объектом `InputStream`, который используется для обнаружения источника данных, но может быть представлена сущностью с публичным и системным идентификатором либо URI-ссылкой (например, на общедоступное DTD-определение). Данный класс является предпочтительной оболочкой для передачи исходных данных SAX-совместимому анализатору.

```
public class InputSource {
    public InputSource();
    public InputSource(String systemId);
    public InputSource(InputStream byteStream);
    public InputSource(Reader characterStream);
    public void setPublicId(String publicId);
    public String getPublicId();
    public void setSystemId(String systemId);
    public String getSystemId();
    public void setByteStream(InputStream byteStream);
    public InputStream getByteStream();
    public void setEncoding(String encoding);
    public String getEncoding();
    public void setCharacterStream(Reader characterStream);
    public Reader getCharacterStream();
}
```

Locator

Данный интерфейс представляет собой дополнение к XML-документу или другим анализируемым конструкциям, поскольку он обеспечивает доступ к системному и публичному идентификатору документа, а также к данным о текущей позиции в обрабатываемом файле. Это особенно полезно для IDE-приложений и для идентификации мест, в которых происходят ошибки при анализе.

```
public interface Locator {
    public abstract String getPublicId();
    public abstract String getSystemId();
    public abstract int getLineNumber();
    public abstract int getColumnNumber();
}
```

Parser

Данный интерфейс был определен в SAX 1.0 и в настоящее время уже устарел. В реализациях SAX 2.0 вместо данного интерфейса следует использовать интерфейс `XMLReader`.

```
public interface Parser {
    public abstract void setLocale(Locale locale) throws SAXException;
```

```
    public abstract void setEntityResolver(EntityResolver resolver);
    public abstract void setDTDHandler(DTDHandler handler);
    public abstract void setDocumentHandler(DocumentHandler handler);
    public abstract void setErrorHandler(ErrorHandler handler);
    public abstract void parse(InputSource source)
              throws SAXException, IOException;
    public abstract void parse(String systemId)
              throws SAXException, IOException;
}
```

SAXException

Базовое исключение, генерируемое методами обратного вызова SAX и реализациями синтаксических анализаторов. Поскольку оно часто генерируется в результате возникновения других исключительных ситуаций, данный класс имеет конструктор, инициализируемый исключением более низкого уровня, а также метод доступа, возвращающий исходное исключение. Данный класс является базовым для классов всех прочих исключений SAX.

```
public class SAXException extends Exception {
     public SAXException(String message);
       public SAXException(Exception e);
       public SAXException(String message, Exception e);
       public String getMessage();
       public Exception getException();
       public String toString();
}
```

SAXNotRecognizedException

Данный класс позволяет реализации интерфейса `XMLReader` генерировать исключение, если анализатору передается неизвестный идентификатор. Чаще всего это происходит при вызове методов `setProperty()` и `setFeature()` (а также соответствующих методов доступа), когда указан URI, о котором анализатор не имеет никакой информации.

```
public class SAXNotRecognizedException extends SAXException {
    public SAXNotRecognizedException(String message);
}
```

SAXNotSupportedException

Данный класс обеспечивает реализации интерфейса `XMLReader` возможность генерировать исключение, когда анализатору передается знакомый, но не поддерживаемый идентификатор. Чаще всего это происходит при вызове методов `setProperty()` и `setFeature()` (а также соответствующих методов доступа), когда указан URI, для которого отсутствует код реализации в анализаторе.

```
public class SAXNotSupportedException extends SAXException {
    public SAXNotSupportedException(String message)
}
```

SAXParseException

Данный класс представляет исключение, возникающее в процессе синтаксического анализа. Информацию о местоположении ошибки в XML-документе можно получить с помощью методов доступа этого класса. Предпочтительным средством передачи этой информации объекту данного класса является указатель позиции в документе, тем не менее, номер строки и столбца документа, где имела место ошибка, можно передать непосредственно через перегруженные конструкторы. Кроме того, соответствующие параметры конструкторов позволяют передать данному классу также системный и публичный идентификаторы документа, в котором возникла ошибка.

```
public class SAXParseException extends SAXException {
    public SAXParseException(String message, Locator locator);
    public SAXParseException(String message, Locator locator,
                             Exception e);
       public SAXParseException(String message, String publicId,
                            String systemId, int lineNumber,
                                 int columnNumber);
    public SAXParseException(String message, String publicId,
                             String systemId, int lineNumber,
                             int columnNumber, Exception e);
    public String getPublicId();
    public String getSystemId();
    public int getColumnNumber();
}
```

XMLFilter

Данный интерфейс является аналогом XMLReader, но для него источником событий является другой интерфейс XMLReader, а не фиксированный документ или сетевой ресурс. Такие фильтры могут объединяться в цепи. Их основное применение заключается в модификации вывода интерфейса XMLReader, расположенного ранее в цепочке, обеспечивая фильтрацию данных, передаваемых методам обратного вызова, до получения уведомления о поступлении данных конечным приложением.

```
public interface XMLFilter extends XMLReader {
    public abstract void setParent(XMLReader parent);
    public abstract XMLReader getParent();
}
```

XMLReader

Этот базовый интерфейс определяет функциональность, связанную с синтаксическим анализом в SAX 2.0. Любой продукт, выполняющий

анализ XML, должен включать по крайней мере одну реализацию этого интерфейса. XMLReader заменяет интерфейс Parser, который был определен в SAX 1.0, и вводит поддержку пространств имен в элементах и атрибутах документа. Помимо обеспечения точки входа для процесса синтаксического анализа (с помощью системного идентификатора или потока InputSource, указывающего на исходные данные), он позволяет регистрировать разнообразные интерфейсы обработчиков, которые предусматриваются в SAX 2.0. Возможности и свойства, доступные в реализации SAX-совместимого анализатора, устанавливаются также с помощью этого интерфейса. Полный список возможностей и свойств содержится в приложении В «Возможности и свойства SAX 2.0».

```
public interface XMLReader {
    public boolean getFeature(String name)
        throws SAXNotRecognizedException, SAXNotSupportedException;
    public void setFeature(String name, boolean value)
              throws SAXNotRecognizedException, SAXNotSupportedException;
    public Object getProperty(String name)
              throws SAXNotRecognizedException, SAXNotSupportedException;
    public void setProperty(String name, Object value)
              throws SAXNotRecognizedException, SAXNotSupportedException;
    public void setEntityResolver(EntityResolver resolver);
    public EntityResolver getEntityResolver();
    public void setDTDHandler(DTDHandler handler);
    public DTDHandler getDTDHandler();
    public void setContentHandler(ContentHandler handler);
    public ContentHandler getContentHandler();
    public void setErrorHandler(ErrorHandler handler);
    public ErrorHandler getErrorHandler();
    public void parse(InputSource input)
              throws IOException, SAXException;
    public void parse(String systemId)
              throws IOException, SAXException;
}
```

Пакет org.xml.sax.ext

Данный пакет содержит расширения базовых классов и интерфейсов SAX. В частности, в нем определены дополнительные обработчики для менее распространенных операций, выполняемых в процессе синтаксического анализа. Поддержка этих расширений для реализаций XMLReader не является обязательной.

DeclHandler

Данный интерфейс определяет методы обратного вызова, позволяющие получать информацию, связанную с объявлениями DTD. Объявления элементов и атрибутов генерируют вызовы соответствующих методов, которым передаются их имена (а в случае атрибутов и имена

элементов), а также сведения об ограничениях. В то время как для атрибутов этот набор данных является строго определенным, в случае элементов модель ограничений передается в текстовом виде как одна строка. Кроме того, определены методы обратного вызова, позволяющие получать уведомления о внутренних и внешних сущностях.

```
public interface DeclHandler {
    public abstract void elementDecl(String name, String model)
              throws SAXException;
    public abstract void attributeDecl(String eName, String aName,
                                       String type, String valueDefault,
                                       String value)
              throws SAXException;
    public abstract void internalEntityDecl(String name, String value)
              throws SAXException;
    public abstract void externalEntityDecl(String name, String publicId,
                                            String systemId)
              throws SAXException;
}
```

LexicalHandler

Данный интерфейс определяет методы обратного вызова для различных событий, которые происходят на уровне документа (в плане его обработки), но не влияют на конечные данные в XML-документе. Например, обработка объявления DTD, комментариев, а также ссылок на сущности приводит к выполнению методов обратных вызовов, принадлежащих реализациям этого интерфейса. Кроме того, определен метод обратного вызова, сообщающий о том, когда начинается и когда заканчивается секция CDATA (данные, о которых идет речь, не изменяются).

```
public interface LexicalHandler {
    public abstract void startDTD(String name, String publicId,
                                  String systemId)
              throws SAXException;
    public abstract void endDTD()
              throws SAXException;
    public abstract void startEntity(String name)
              throws SAXException;
    public abstract void endEntity(String name)
              throws SAXException;
    public abstract void startCDATA()
              throws SAXException;
    public abstract void endCDATA()
              throws SAXException;
    public abstract void comment(char ch[], int start, int length)
              throws SAXException;
}
```

Пакет org.xml.sax.helpers

Данный пакет содержит расширения базовых классов и интерфейсов SAX. В частности, определены дополнительные обработчики для менее распространенных операций, выполняемых в процессе синтаксического анализа. Поддержка этих расширений для реализаций `XMLReader` не является обязательной.

Примечание

Для классов пакета, представляющих собой стандартные реализации базовых интерфейсов `org.xml.sax`, повторяющиеся методы опущены. Мои комментарии отражают, методы какого из интерфейсов реализованы.

AttributeListImpl

Данный класс определяет стандартную реализацию интерфейса `org.xml.sax.AttributeList`, и в SAX 2.0 является устаревшим. Он позволяет добавлять и удалять атрибуты, а также очищать их список.

```
public class AttributeListImpl implements AttributeList {
       public AttributeListImpl();
       public AttributeListImpl(AttributeList atts);

       // Реализация интерфейса AttributeList

       // Дополнительные методы
       public void setAttributeList(AttributeList atts);
       public void addAttribute(String name, String type, String value);
       public void removeAttribute(String name);
       public void clear();

}
```

AttributesImpl

Данный класс определяет стандартную реализацию интерфейса `org.xml.sax.Attributes`. Он позволяет добавлять и удалять атрибуты, а также очищать их список.

```
public class AttributesImpl implements Attributes {
       public AttributesImpl();
       public AttributesImpl(Attributes atts);

       // Реализация интерфейса Attributes

       // Дополнительные методы
    public void addAttribute(String uri, String localName,
                             String qName, String type, String value);
    public void setAttribute(int index, String uri, String localName,
                              String qName, String type, String value);
       public void clear();
}
```

DefaultHandler

Данный вспомогательный класс определяет пустые реализации всех базовых интерфейсов обработчиков SAX 2.0 и может быть расширен с целью быстрого добавления обработчиков путем переопределения методов, реализующих функциональность, определяемую приложением. Данный класс заменяет класс `org.xml.sax.HandlerBase`, существовавший в SAX 1.0.

```
public class DefaultHandler implements EntityResolver, DTDHandler,
                                       ContentHandler, ErrorHandler {

        // (Пустая) реализация интерфейса EntityResolver

        // (Пустая) реализация интерфейса DTDHandler

        // (Пустая) реализация интерфейса ContentHandler

        // (Пустая) реализация интерфейса ErrorHandler
}
```

LocatorImpl

Данный класс определяет стандартную реализацию интерфейса `org.xml.sax.Locator`. Он также предоставляет методы для принудительной установки значений номера строки и столбца.

```
public class LocatorImpl implements Locator {
        public LocatorImpl();
        public LocatorImpl(Locator locator);

        // Реализация интерфейса Locator

        // Дополнительные методы
        public void setPublicId(String publicId);
        public void setSystemId(String systemId);
        public void setLineNumber(int lineNumber);
        public void setColumnNumber(int columnNumber);
}
```

NamespaceSupport

Данный класс инкапсулирует функциональность, связанную с пространствами имен, что избавляет приложения от необходимости реализовывать ее самостоятельно (в случаях, когда это не требуется по соображениям производительности). Данный класс позволяет обрабатывать контексты пространств имен в модели стека, а также предусматривает возможность обработки уточненных имен XML 1.0.

```
public class NamespaceSupport {
        public NamespaceSuport();
        public void reset();
    public void pushContext();
```

```
    public void popContext();
    public boolean declarePrefix(String prefix, String uri);
    public String [] processName(String qName, String parts[],
                                 boolean isAttribute);
    public String getURI(String prefix);
    public Enumeration getPrefixes();
    public Enumeration getDeclaredPrefixes();
}
```

ParserAdapter

Данный вспомогательный класс представляет собой оболочку для класса `Parser` SAX 1.0 и расширяет его функциональность до возможностей интерфейса `XMLReader` SAX 2.0 (в частности, появляется поддержка пространств имен). Единственный метод, который не будет вести себя как следует, – это `skippedEntity()` интерфейса `ContentHandler`: он никогда не будет вызываться.

```
public class ParserAdapter implements XMLReader, DocumentHandler {
       public ParserAdapter() throws SAXException;
       public ParserAdapter(Parser parser);

       // Реализация интерфейса XMLReader

       // Реализация интерфейса DocumentHandler
}
```

ParserFactory

Данный класс содержит методы, которые динамически создают экземпляр класса, реализующего интерфейс `Parser`, по указанному имени класса. Если имя класса не указано, создается экземпляр класса, имя которого хранится в системном свойстве `org.xml.sax.driver`.

```
public class ParserFactory {
    public static Parser makeParser() throws ClassNotFoundException,
                IllegalAccessException, InstantiationException,
                NullPointerException, ClassCastException;
    public static Parser makeParser(String className)
                throws ClassNotFoundException, IllegalAccessException,
                InstantiationException, ClassCastException;
}
```

XMLFilterImpl

Данный класс предоставляет стандартную реализацию интерфейса `org.xml.sax.XMLFilter`.

```
public class XMLFilterImpl implements XMLFilter, EntityResolver,
                                      DTDHandler, ContentHandler,
                                      ErrorHandler {
       public XMLFilterImpl();
```

```
        public XMLFilterImpl(XMLReader parent);

        // Реализация интерфейса XMLFilter

        // Реализация интерфейса XMLReader

        // Реализация интерфейса EntityResolver

        // Реализация интерфейса DTDHandler

        // Реализация интерфейса ContentHandler

        // Реализация интерфейса ErrorHandler
}
```

XMLReaderAdapter

Данный вспомогательный класс представляет собой оболочку для реализации интерфейса `XMLReader` в SAX 2.0 и расширяет его функциональность до возможностей интерфейса `Parser` в SAX 1.0 (что делает недоступной поддержку пространств имен). Возможность, связанная с распознаванием пространств имен (*http://xml.org/sax/features/namespaces*), должна поддерживаться анализатором, иначе во время синтаксического анализа будут возникать ошибки.

```
public class XMLReaderAdapter implements Parser, ContentHandler {
        public XMLReaderAdapter () throws SAXException;
        public XMLReaderAdapter (XMLReader xmlReader);

    // Реализация интерфейса Parser

        // Реализация интерфейса ContentHandler
}
```

XMLReaderFactory

Данный класс содержит методы, которые динамически создают экземпляр класса, реализующего интерфейс `XMLReader`, по указанному имени класса. Если имя класса не указано, создается экземпляр класса, имя которого хранится в системном свойстве `org.xml.sax.driver`.

```
final public class XMLReaderFactory {
    public static XMLReader createXMLReader() throws SAXException;
    public static XMLReader createXMLReader(String className)
              throws SAXException;
}
```

DOM Level 2

Объектная модель документа (DOM) обеспечивает полное представление XML-документа, которое хранится в памяти. Модель DOM, разработанная консорциумом W3C, позволяет получать информацию по

структуре документа только после того, как проведен его полный синтаксический анализ. Хотя в DOM Level 3 будет описан специальный API для получения объекта DOM `Document`, в текущей версии DOM эта функциональность не определена. DOM и SAX устроены похоже, большинство базовых пакетов DOM состоит из интерфейсов, которые определяют структурные компоненты XML-документов и отображают их в конструкции языка Java (аналогичные отображения существуют для CORBA, JavaScript и других инструментов программирования).

Пакет org.w3c.dom

Данный пакет содержит базовые интерфейсы и классы DOM Level 2. Обычно программное обеспечение, выполняющее синтаксический анализ, обеспечивает реализацию этих интерфейсов, которые неявно используются прикладными программами.

Attr

Данный интерфейс представляет собой отображение атрибута элемента XML в конструкции языка Java. Он обеспечивает доступ к имени и значению атрибута, а также позволяет устанавливать его значение[1]. Метод `getSpecified()` позволяет определить, присутствует ли атрибут (и его значение) в XML-документе явным образом, или атрибуту присвоено значение по умолчанию согласно указанному в DTD. Наконец, с помощью данного интерфейса может быть получен элемент-«владелец» атрибута.

```
public interface Attr extends Node {
    public String getName();
    public boolean getSpecified();
    public String getValue();
    public void setValue(String value) throws DOMException;
    public Element getOwnerElement();
}
```

CDATASection

Данный интерфейс не определяет собственные методы, но он наследует все методы интерфейса `Text`. Однако наличие такого интерфейса (и, следовательно, соответствующего типа узла) дает возможность различать текст из секций `CDATA` XML-документа и обычный текст из элементов, не принадлежаший секции `CDATA`.

```
public interface CDATASection extends Text {
}
```

[1] Здесь и далее методы DOM вида `setXXX()` генерируют исключение `DOMException`, если сделана попытка изменить узел, имеющий статус «только для чтения».

CharacterData

Данный интерфейс представляет собой интерфейс-предок для всех типов текстовых узлов в DOM (Text, Comment и CDATASection). Он определяет методы для доступа к данным и изменения данных текстового узла, а также методы для работы непосредственно с символьными данными; реализовано получение длины, добавление, вставка и удаление данных, а также частичная и полная замена. Все эти методы генерируют исключительную ситуацию DOMException, если узел доступен только для чтения.

```
public interface CharacterData extends Node {
    public String getData() throws DOMException;
    public void setData(String data) throws DOMException;
    public int getLength();
    public String substringData(int offset, int count)
        throws DOMException;
    public void appendData(String arg) throws DOMException;
    public void insertData(int offset, String arg) throws DOMException;
    public void deleteData(int offset, int count) throws DOMException;
    public void replaceData(int offset, int count, String arg)
        throws DOMException;
}
```

Comment

Данный интерфейс реализует Java-представление комментариев XML. Аналогично интерфейсу CDATASection он не добавляет никаких собственных методов, но позволяет различать (на основе типа интерфейса) текст и комментарии в XML-документе.

```
public interface Comment extends CharacterData {
}
```

Document

Этот интерфейс является DOM-представлением полного XML-документа. Он также применяется в качестве основы для создания новых элементов, атрибутов, инструкций обработки и других конструкций XML. Интерфейс позволяет получить объявление DTD (метод getDocType()) и корневой элемент (метод getDocumentElement()), а также выполнять поиск конкретного элемента в дереве в упорядоченном режиме (метод getElementsByTagName()). Поскольку модель DOM требует, чтобы все реализации интерфейса Node были связаны с объектом, представляющим документ DOM, этот интерфейс предоставляет методы для создания различных типов узлов DOM. Каждый метод createXXX() имеет парный метод createXXXNS(), поддерживающий пространства имен. Кроме того, узлы можно импортировать в документ DOM с помощью метода importNode(). Логический параметр этого метода определеяет, следует ли рекурсивно импортировать и дочерние DOM-узлы.

```
public interface Document extends Node {
    public DocumentType getDoctype();
    public DOMImplementation getImplementation();
    public Element getDocumentElement();
    public Element createElement(String tagName) throws DOMException;
    public DocumentFragment createDocumentFragment();
    public Text createTextNode(String data);
    public Comment createComment(String data);
    public CDATASection createCDATASection(String data)
        throws DOMException;
    public ProcessingInstruction
        createProcessingInstruction(String target, String data)
        throws DOMException;
    public Attr createAttribute(String name) throws DOMException;
    public EntityReference createEntityReference(String name)
        throws DOMException;
    public NodeList getElementsByTagName(String tagname);
    public Node importNode(Node importedNode, boolean deep)
        throws DOMException;
    public Element createElementNS(String namespaceURI,
                                   String qualifiedName)
        throws DOMException;
    public Attr createAttributeNS(String namespaceURI,
                                  String qualifiedName)
        throws DOMException;
    public NodeList getElementsByTagNameNS(String namespaceURI,
                                           String localName);
    public Element getElementById(String elementId);
}
```

DocumentFragment

Данный интерфейс предназначен для работы с фрагментами документа DOM. В случае его применения нет необходимости целиком хранить в памяти дерево документа.

```
public interface DocumentFragment extends Node {
}
```

DocumentType

Данный интерфейс представляет собой отображение объявления `DOCTYPE`. Метод `getName()` возвращает имя элемента, следующее непосредственно за тегом `<!DOCTYPE`, предусмотрены также методы для получения системного и публичного индентификатора DTD. Кроме того, если присутствуют любые внутренние сущности или нотации, их можно получить с помощью соответствующих методов `getXXX()`.

```
public interface DocumentType extends Node {
    public String getName();
    public NamedNodeMap getEntities();
```

```
    public NamedNodeMap getNotations();
    public String getPublicId();
    public String getSystemId();
    public String getInternalSubset();
}
```

DOMException

Данный класс реализует исключение, генерируемое интерфейсами DOM при возникновении ошибок. В нем определен набор кодов ошибок, описывающий разнообразные проблемы, которые могут возникать при использовании DOM и приводить к генерации исключений.

```
public class DOMException extends RuntimeException {
       public DOMException(short code, String message);

    // Коды исключений
    public static final short INDEX_SIZE_ERR;
    public static final short DOMSTRING_SIZE_ERR;
    public static final short HIERARCHY_REQUEST_ERR;
    public static final short WRONG_DOCUMENT_ERR;
    public static final short INVALID_CHARACTER_ERR;
    public static final short NO_DATA_ALLOWED_ERR;
    public static final short NO_MODIFICATION_ALLOWED_ERR;
    public static final short NOT_FOUND_ERR;
    public static final short NOT_SUPPORTED_ERR;
    public static final short INUSE_ATTRIBUTE_ERR;
    public static final short INVALID_STATE_ERR;
    public static final short SYNTAX_ERR;
    public static final short INVALID_MODIFICATION_ERR;
    public static final short NAMESPACE_ERR;
    public static final short INVALID_ACCESS_ERR;
}
```

DOMImplementation

Данный интерфейс обеспечивает стандартную точку входа для доступа к фирменным реализациям DOM и позволяет создавать с их помощью объекты `DocumentType` и `Document`[1]. Также он предоставляет метод `hasFeature()`, позволяющий определить, поддерживает ли данная реализация конкретную возможность, например, модули `Traversal` и `Range` DOM Level 2.

```
public interface DOMImplementation {
    public boolean hasFeature(String feature, String version);
```

[1] К сожалению, для получения экземпляра `DomImplementation` требуется наличие объекта `Document` и вызов метода `getDOMImplementation()` либо явная загрузка классов конкретной реализации DOM. Это служит отправной точкой для задачки про курицу и яйцо. Более подробная информация на эту тему содержится в главах 5 и 6.

```
    public DocumentType createDocumentType(String qualifiedName,
                                           String publicId,
                                           String systemId)
        throws DOMException;
    public Document createDocument(String namespaceURI,
                                   String qualifiedName,
                                   DocumentType doctype)
        throws DOMException;
}
```

Element

Данный интерфейс реализует Java-отображение для элементов XML. В нем определены методы для получения имени и атрибутов элемента, а также для установки этих значений. Кроме того, он предоставляет несколько способов доступа к атрибутам элементов XML, включая версии методов `getXXX()` и `setXXX()`, поддерживающие пространства имен.

```
public interface Element extends Node {
    public String getTagName();
    public String getAttribute(String name);
    public void setAttribute(String name, String value)
        throws DOMException;
    public void removeAttribute(String name) throws DOMException;
    public Attr getAttributeNode(String name);
    public Attr setAttributeNode(Attr newAttr) throws DOMException;
    public Attr removeAttributeNode(Attr oldAttr) throws DOMException;
    public NodeList getElementsByTagName(String name);
    public String getAttributeNS(String namespaceURI, String localName);
    public void setAttributeNS(String namespaceURI, String qualifiedName,
                               String value)
        throws DOMException;
    public void removeAttributeNS(String namespaceURI, String localName)
                                  throws DOMException;
    public Attr getAttributeNodeNS(String namespaceURI, String localName);
    public Attr setAttributeNodeNS(Attr newAttr) throws DOMException;
    public NodeList getElementsByTagNameNS(String namespaceURI,
                                           String localName);
    public boolean hasAttribute(String name);
    public boolean hasAttributeNS(String namespaceURI, String localName);
}
```

Entity

Данный интерфейс реализует Java-отображение анализируемых и неанализируемых сущностей. Доступ к системному и публичному идентификаторам, а также к нотациям сущностей (из DTD) организован с помощью соответствующих методов.

```
public interface Entity extends Node {
    public String getPublicId();
```

```
    public String getSystemId();
    public String getNotationName();
}
```

EntityReference

Данный интерфейс предоставляет значение, полученное после интерпретации ссылки на сущность, и предполагает, что интерпретация ссылок на символьные и предопределенные сущности уже произошла, до того как приложение-клиент получило доступ к методам интерфейса.

```
public interface EntityReference extends Node {
}
```

NamedNodeMap

Данный интерфейс определяет список, во многом похожий на NodeList, каждый узел (Node) которого должен быть именованным узлом (таким как Element или Attr). Благодаря этому требованию имеется возможность определить методы для обеспечения доступа к элементам списка по их именам (с учетом или без учета поддержки пространств имен). Такой список предусматривает также удаление и модификацию элементов. Если соответствующий узел доступен только для чтения, методы генерируют исключительную ситуацию DOMException.

```
public interface NamedNodeMap {
    public Node getNamedItem(String name);
    public Node setNamedItem(Node arg) throws DOMException;
    public Node removeNamedItem(String name) throws DOMException;
    public Node item(int index);
    public int getLength();
    public Node getNamedItemNS(String namespaceURI, String localName);
    public Node setNamedItemNS(Node arg) throws DOMException;
    public Node removeNamedItemNS(String namespaceURI, String localName)
        throws DOMException;
}
```

Node

Данный интерфейс является базовым для всех объектов DOM. Он предоставляет мощный набор методов для доступа к информации об узле (Node) дерева DOM. Он также предусматривает обработку дочерних узлов данного узла (если они существуют). Хотя большинство методов не нуждаются в комментариях, некоторые из них требуют пояснений. Метод getAttributes() возвращает значащие данные только в том случае, если узел представляет собой элемент (Element). Метод cloneNode() предназначен для создания копии узла (с учетом или без учета вложенных элементов, если копируется элемент). Метод normalize() упорядочивает текстовые узлы таким образом, что все интерпретирован-

ные ссылки на сущности объединены в текстовые узлы и гарантируется отсутствие смежных текстовых узлов. Метод isSupported() предоставляет информацию о возможностях узла. Определены также методы для работы с пространствами имен (getNamespaceURI(), getPrefix() и getLocalName()). Наконец, предусмотрен набор констант для определения типа узла путем их сравнения с результатом, возвращаемым методом getNodeType().

```
public interface Node {
    public String getNodeName();
    public String getNodeValue() throws DOMException;
    public void setNodeValue(String nodeValue) throws DOMException;
    public short getNodeType();
    public Node getParentNode();
    public NodeList getChildNodes();
    public Node getFirstChild();
    public Node getLastChild();
    public Node getPreviousSibling();
    public Node getNextSibling();
    public NamedNodeMap getAttributes();
    public Document getOwnerDocument();
    public Node insertBefore(Node newChild,  Node refChild)
        throws DOMException;
    public Node replaceChild(Node newChild, Node oldChild)
                            throws DOMException;
    public Node removeChild(Node oldChild) throws DOMException;
    public Node appendChild(Node newChild) throws DOMException;
    public boolean hasChildNodes();
    public Node cloneNode(boolean deep);
    public void normalize();
    public boolean isSupported(String feature, String version);
    public String getNamespaceURI();
    public String getPrefix();
    public void setPrefix(String prefix) throws DOMException;
    public String getLocalName();
    public boolean hasAttributes();

    // Константы типов узлов
    public static final short ELEMENT_NODE;
    public static final short ATTRIBUTE_NODE;
    public static final short TEXT_NODE;
    public static final short CDATA_SECTION_NODE;
    public static final short ENTITY_REFERENCE_NODE;
    public static final short ENTITY_NODE;
    public static final short PROCESSING_INSTRUCTION_NODE;
    public static final short COMMENT_NODE;
    public static final short DOCUMENT_NODE;
    public static final short DOCUMENT_TYPE_NODE;
    public static final short DOCUMENT_FRAGMENT_NODE;
    public static final short NOTATION_NODE;
}
```

NodeList

Данный интерфейс является структурой DOM, аналогичной конструкциям Vector или List в Java. NodeList возвращается каждым методом, результатом работы которого является набор узлов DOM. Данный интерфейс прозволяет выполнять перебор узлов в цикле, а также предусматривает возможность получить узел с указанным индексом.

```
public interface NodeList {
    public Node item(int index);
    public int getLength();
}
```

Notation

Данный интерфейс представляет конструкцию DTD NOTATION, которая используется в определениях формата неанализируемых сущностей и объявлениях инструкций обработки. Интерфейс обеспечивает доступ к системному и публичному идентификаторам. Оба метода возвращают null, если идентификаторы отсутствуют.

```
public interface Notation extends Node {
    public String getPublicId();
    public String getSystemId();
}
```

ProcessingInstruction

Данный интерфейс представляет инструкцию обработки (PI) XML. В нем определены методы для получения целевого приложения и данных инструкции обработки. Обратите внимание, что нет способа получать отдельные пары «имя-значение» из инструкции обработки. ProcessingInstruction предоставляет возможность задавать данные инструкции обработки.

```
public interface ProcessingInstruction extends Node {
    public String getTarget();
    public String getData();
    public void setData(String data) throws DOMException;
}
```

Text

Данный интерфейс реализует Java-отображение текстовых данных элемента XML. Единственный метод, сверх определенных в интерфейсе CharacterData, разбивает узел на два. В исходном текстовом узле сохраняется текст до указанной позиции offset, а в узле, возвращаемом методом, – текст после указанной позиции. Как и в случае с другими методами-модификаторами, может генерироваться исключение DOMException, если узел доступен только для чтения.

```
public interface Text extends CharacterData {
    public Text splitText(int offset) throws DOMException;
}
```

JAXP 1.1

JAXP реализует уровень абстракции для создания экземпляра SAX- или DOM-совместимого анализатора, а также обеспечивает возможность выполнения преобразований, не зависящих от конкретной платформы.

Пакет javax.xml.parsers

Это единственный пакет в JAXP, и в нем определены классы, реализующие дополнительный уровень абстракции и заменяемости компонентов при анализе данных XML.

DocumentBuilder

Данный класс представляет собой оболочку для класса синтаксического анализатора. Он позволяет сделать синтаксический анализ независящим от платформы.

```
public abstract class DocumentBuilder {
    public Document parse(InputStream stream)
        throws SAXException, IOException, IllegalArgumentException;
    public Document parse(InputStream stream, String systemID)
        throws SAXException, IOException, IllegalArgumentException;
    public Document parse(String uri)
        throws SAXException, IOException, IllegalArgumentException;
    public Document parse(File file)
        throws SAXException, IOException, IllegalArgumentException;
    public abstract Document parse(InputSource source)
        throws SAXException, IOException, IllegalArgumentException;

    public abstract Document newDocument();
    public abstract boolean isNamespaceAware();
    public abstract boolean isValidating();
    public abstract void setEntityResolver(EntityResolver er);
    public abstract void setErrorHandler(ErrorHandler eh);
    public DOMmplementation getDOMImplementation();
}
```

DocumentBuilderFactory

Данный класс представляет собой фабрику для создания экземпляров класса `DocumentBuilder`. Он позволяет настраивать возможности, которые связаны с пространствами имен и проверкой действительности и используются при создании этих экземпляров.

```
public abstract class DocumentBuilderFactory {
    public static DocumentBuilderFactory newInstance();
    public abstract DocumentBuilder newDocumentBuilder()
        throws ParserConfigurationException;

    public void setAttribute(String name, Object value);
    public void setCoalescing(boolean coalescing);
    public void setExpandEntityReferences(boolean expand);
    public void setIgnoringComments(boolean ignoreComments);
    public void setIgnoringElementContentWhitespace(boolean
                ignoreWhitespace);
    public void setNamespaceAware(boolean aware);
    public void setValidating(boolean validating);

    public boolean isCoalescing();
    public boolean isExapandEntityReferences();
    public boolean isIgnoringComments();
    public boolean isIgnoreingElementContentWhitespace();
    public boolean isNamespaceAware();
    public boolean isValidating();
    public Object getAttribute(String name);
}
```

FactoryConfigurationError

Данный класс определяет ошибку, которая генерируется в том случае, если экземпляр класса не может быть создан фабрикой.

```
public class FactoryConfigurationException extends Error {
    public FactoryConfigurationError();
    public FactoryConfigurationError(String msg);
    public FactoryConfigurationError(Exception e);
    public FactoryConfigurationError(Exception e, String msg);
}
```

ParserConfigurationException

Данный класс определяет исключение, которое генерируется, если невозможно создать требуемый анализатор с указанными настройками проверки действительности и поддержки пространств имен.

```
public class ParserConfigurationException extends Exception {
    public ParserConfigurationException();
    public ParserConfigurationException(String msg);
}
```

SAXParser

Данный класс является оболочкой для класса анализатора, совместимого с SAX 1.0/2.0, и позволяет сделать синтаксический анализ независимым от платформы. Методы класса парны, один из методов пары предназначен для SAX 1.0, другой – для SAX 2.0.

```
public abstract class SAXParser {
    public void parse(InputStream stream, HandlerBase base)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(InputStream stream, HandlerBase base, String systemID)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(String uri, HandlerBase base)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(File file, HandlerBase base)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(InputSource source, HandlerBase base)
        throws SAXException, IOException, IllegalArgumentException;

    public void parse(InputStream stream, DefaultHandler dh)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(InputStream stream, DefaultHandler dh, String systemID)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(String uri, DefaultHandler dh)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(File file, DefaultHandler dh)
        throws SAXException, IOException, IllegalArgumentException;
    public void parse(InputSource source, DefaultHandler dh)
        throws SAXException, IOException, IllegalArgumentException;

    public Parser getParser() throws SAXException;
    public XMLReader getXMLReader() throws SAXException;

    public Object getProperty(String name);
    public void setProperty(String name, Object value);
    public boolean isNamespaceAware();
    public boolean isValidating();
}
```

SAXParserFactory

Данный класс представляет собой фабрику, используемую для создания экземпляров класса `SAXParser`**. Он позволяет настраивать возможности, связанные с пространствами имен и проверкой действительности и используемые при создании этих экземпляров.**

```
public abstract class SAXParserFactory {
    public static SAXParserFactory newInstance();
    public SAXParser newSAXParser()
        throws ParserConfigurationException, SAXException;

    public void setNamespaceAware(boolean aware);
    public void setValidating(boolean validating);
    public void setFeature(String name, boolean value);
    public boolean isNamespaceAware();
    public boolean isValidating();
    public boolean getFeature(String name);
}
```

Пакет javax.xml.transform

Данный пакет применяется в JAXP для преобразования XML-документов. Он позволяет сделать эти преобразования независящими от платформы при условии, что они используют интерфейсы TrAX (Transformations API for XML), определенные здесь.

ErrorListener

Данный интерфейс аналогичен интерфейсу `ErrorHandler` в SAX и позволяет получать уведомления об ошибках, возникающих при преобразованиях. Реализуйте его в собственных приложениях, использующих TrAX.

```
public interface ErrorListener {
    public void warning(TransformerException exception);
    public void error(TransformerException exception);
    public void fatalError(TransformerException exception);
}
```

OutputKeys

Этот класс представляет собой просто контейнер для набора констант, используемых в TrAX API.

```
public class OutputKeys {
    public static final String CDATA_SECTION_ELEMENTS;
    public static final String DOCTYPE_PUBLIC;
    public static final String DOCTYPE_SYSTEM;
    public static final String ENCODING;
    public static final String INDENT;
    public static final String MEDIA_TYPE;
    public static final String METHOD;
    public static final String OMIT_XML_DECLARATION;
    public static final String STANDALONE;
    public static final String VERSION;
}
```

Result

Данный интерфейс предназначен для вывода преобразований XML. В пакетах JAXP `javax.xml.transform.*` содержатся стандартные реализации этого интерфейса.

```
public interface Result {
    public static final String PI_DISABLE_OUTPUT_ESCAPING;
    public static final String PI_ENABLE_OUTPUT_ESCAPING;

    public String getSystemId();
    public void setSystemId();
}
```

Source

Данный интерфейс представляет исходные данные для преобразований XML. Реализации этого интерфейса по умолчанию представлены в пакетах JAXP `javax.xml.transform.*`.

```
public interface Source {
    public String getSystemId();
    public void setSystemId();
}
```

SourceLocator

Данный интерфейс аналогичен интерфейсу SAX `Locator` и представляет информацию о позиции в исходном документе в случае TrAX. Как и `ErrorListener`, он наиболее полезен при диагностике и обработке ошибок.

```
public interface SourceLocator {
    public int getColumnNumber();
    public int getLineNumber();
    public String getPublicId();
    public String getSystemId();
}
```

Templates

Данный интерфейс предназначен для выполнения оптимальных преобразований при помощи одной и той же таблицы стилей. Его методы позволяют создавать экземпляры класса `Transformer` и просматривать текущий набор свойств вывода.

```
public interface Tempaltes {
    public Properties getOutputProperties();
    public Transformer newTransformer();
}
```

Transformer

Это базовый (абстрактный) класс, предоставляющий возможности преобразования XML при использовании TrAX и JAXP. Помимо настройки различных свойств и объектов для интерфейса, можно выполнять собственно преобразования при помощи метода `transform()`.

```
public class Transformer {
    public void setErrorListener(ErrorListener errorListener);
    public ErrorListener getErrorListener();
    public void setURIResolver(URIResolver resolver);
    public URIResolver getURIResolver();

    public void setOutputProperties(Properties properties);
    public Properties getOutputProperties();
    public void setOutputProperty(String name, String value);
```

```
    public String getOutputProperty(String name);
    public void clearParmaters();
    public void setParameter(String name, String value);
    public Object getParameter(String name);

    public void transform(Source xmlSource, Result outputTarget);
}
```

TransformerFactory

Это вторая «половина» механизма преобразований в JAXP. Можно указать таблицу стилей, используемую для преобразований, а затем получить новые экземпляры экземпляра `Transformer`. Также можно создать новый объект `Templates`, а затем применить одну и ту же таблицу стилей для нескольких преобразований.

```
public class TransformerFactory {
    public TransformerFactory newInstance();
    public Transformer newTemplates(Source stylesheet);
    public Transformer newTransformer(Source stylesheet);
    public Transformer newTransformer();

    public Source getAssociatedStylesheet(Source source, String media,
                                          String title, String charset);
    public ErrorListener getErrorListener();
    public void setErrorListener(ErrorListener errorListener);
    public URIResolver getURIResolver();
    public void setURIResolver(URIResolver uriResolver);
    public Object getAttribute(String name);
    public void setAttribute(String name, String value);
    public boolean getFeature(String name);
}
```

URIResolver

Данный интерфейс отвечает за интерпретацию URI и схож с интерфейсом SAX `EntityResolver`.

```
public interface URIResolver {
    public Source resolve(String href, String base);
}
```

Пакет javax.xml.transform.dom

Данный пакет предоставляет два класса: `DOMResult` и `DOMSource`. Эти реализации интерфейсов `Result` и `Source` применяются в случаях, когда исходные данные и конечные данные преобразования представлены деревьями DOM. Здесь мы не будем рассматривать методы этих простых реализаций, поскольку их применение подробно описано в главе 9.

Пакет javax.xml.transform.sax

Данный пакет предоставляет два класса: `SAXResult` и `SAXSource`. Эти реализации интерфейсов `Result` и `Source` применяются в случаях, когда исходными и конечными данными для преобразования являются события SAX. Здесь методы этих простых реализаций не рассматриваются, поскольку их применение подробно описано в главе 9.

Пакет javax.xml.transform.stream

Данный пакет предоставляет два класса: `StreamResult` и `StreamSource`. Эти реализации интерфейсов `Result` и `Source` применяются в случаях, когда исходные и конечные данные преобразования представлены потоками ввода/вывода. Здесь методы этих простых реализаций не рассматриваются, т. к. их применение подробно описано в главе 9.

JDOM 1.0 (Beta 7)

Модель JDOM 1.0 (beta 7), описанная в главах 7 и 8, обеспечивает полное представление XML-документа в виде дерева. Хотя эта модель похожа на DOM, она не является столь же жесткой. Например, она позволяет напрямую устанавливать содержимое элемента вместо того, чтобы устанавливать значение его дочернего элемента. Кроме того, JDOM предоставляет разработчику готовые классы, а не интерфейсы, что позволяет создавать объекты непосредственно, а не с помощью фабрики. SAX и DOM используются в JDOM только для создания объекта JDOM `Document` на основе существующих XML-данных и входят в пакет `org.jdom.input`.

Пакет org.jdom

Данный пакет содержит базовые классы JDOM 1.0.[1] Он состоит из классов, моделирующих в Java конструкции XML, а также исключений, которые могут генерироваться при возникновении ошибок.[2]

---

[1] Обратите внимание, что хотя JDOM API достаточно стабилен, он все еще находится в стадии бета-тестирования. После выхода этой книги что-то может измениться. Пожалуйста, обратитесь к странице *http://www.jdom.org* за списком последних классов JDOM.

[2] Чтобы этот раздел не стал совершенно скучным, я опустил все исключения JDOM, кроме базового – `JDOMException`. Лучше я уделю время классам, чем странным исключительным ситуациям.

Attribute

Класс Attribute моделирует в Java атрибут XML и определяет связанную с ним функциональность. Посредством методов класса пользователь может получить значение атрибута, а также информацию о пространстве имен атрибута. Экземпляр класса можно создать, указав имя и значение атрибута либо пространство имен, локальное имя и значение атрибута. Предусмотрено несколько вспомогательных методов для автоматического преобразования значения атрибута.

```
public class Attribute {
    public Attribute(String name, String value);
    public Attribute(String name, String value, Namespace ns);

    public Element getParent();
    public String getName();
    public Namespace getNamespace();
    public void setNamespace(Namespace ns);
    public String getQualifiedName();
    public String getNamespacePrefix();
    public String getNamespaceURI();
    public String getValue();
    public void setValue(String value);

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();

    // Вспомогательные методы преобразования данных
    public String get StringValue(String default Value);
    public int getIntValue() throws DataConversionException;
    public long getLongValue() throws DataConversionException;
    public float getFloatValue() throws DataConversionException;
    public double getDoubleValue() throws DataConversionException;
    public boolean getBooleanValue() throws DataConversionException;
}
```

CDATA

Класс CDATA определяет функциональность, связанную с секциями CDATA в XML.

```
public class CDATA {
    public CDATA(String text);

    public String getText();

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Comment

Класс Comment является простым представлением комментария XML и инкапсулирует текст комментария XML.

```
public class Comment {
    public Comment(String text);

    public Document getDocument();
    public Element getParent();
    public String getText();
    public void setText(String text);

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

DocType

Класс DocType представляет объявление DOCTYPE XML-документа. Он включает информацию об именах элементов, а также о публичном и системном идентификаторах ссылки на внешнее DTD, если таковая присутствует.

```
public class DocType {
    public DocType(String elementName, String publicID, String systemID);
    public DocType(String elementName, String systemID);
    public DocType(String elementName);

    public Document getDocument();
    public String getElementName();
    public String getPublicID();
    public DocType setPublicID(String publicID);
    public String getSystemID();
    public DocType setSystemID(String systemID);

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Document

Класс Document моделирует XML-документ в Java. При создании экземпляра класса должен быть указан корневой элемент (Element), хотя впоследствии его можно изменить с помощью метода setRootElement(). Метод getContent() возвращает все содержимое документа, включая корневой элемент и все комментарии (Comments), которые могут существовать на уровне документа в XML.

```
public class Document {
    public Document(Element rootElement, DocType docType);
    public Document(Element rootElement);
    public Document(List content);
    public Document(List content, DocType docType);

    public Document addContent(Comment comment);
    public Document removeContent(Comment comment);
    public Document addContent(ProcessingInstruction pi);
    public Document removeContent(ProcessingInstruction pi);
    public Element getRootElement() throws NoSuchElementException;
    public Document setRootElement(Element rootElement);
    public DocType getDocType();
    public Document setDocType(DocType docType);
    public List getContent();
    public void setMixedContent(List content);

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Element

Класс `Element` является Java-представлением элемента XML. Он полностью поддерживает пространства имен, поэтому все методы принимают имя элемента в качестве аргумента, а также необязательную информацию о пространстве имен. Результатом вызова метода `getText()` всегда является строка – либо с текстовым содержимым элемента, либо пустая. Говорят, что элемент имеет смешанное содержимое, если он содержит комбинацию текстовых данных и вложенных элементов, а также необязательных комментариев, сущностей и инструкций обработки. Полный список (`List`) содержимого элемента можно получить с помощью метода `getContent()`, а полученные результаты в списке обрабатываются с помощью оператора `instanceof` для типов `String`, `Element` или `Comment`.

Методы `addXXX()` разработаны таким образом, чтобы их вызовы можно было объединять в цепи, и поэтому возвращают измененный элемент:

```
Element element = new Element("root");
element.addChild(new Element("child")
    .addChild(new Element("grandchild")
        .addAttribute("name", "value")
        .setContent("Hello World!"))
    .addChild(new Element("anotherChild"))
);
```

В результате выполнения этого кода получается следующий фрагмент XML-документа:

```
<root>
  <child>
    <grandchild name="value">
      Hello World!
    </grandchild>
  </child>
  <anotherChild />
</root>
```

Вот перечень методов API:

```
public class Element {
    public Element(String name);
    public Element(String name, String uri);
    public Element(String name, String prefix, String uri);
    public Element(String name, Namespace ns);

    public Document getDocument();
    public Element getParent();
    public Element detach();
    public String getName();
    public void setName(String name);
    public Namespace getNamespace();
    public Namespace getNamespace(String prefix);
    public void setNamespace(Namespace ns);
    public String getNamespacePrefix();
    public String getNamespaceURI();
    public String getQualifiedName();
    public void addNamespaceDeclaration(Namespace additionalNS);
    public void removeNamespaceDeclaration(Namespace additionalNS);
    public List getAdditionalNamespaces();

    public List getContent();
    public Element setMixedContent(List mixedContent);
    public Element addContent(CDATA cdata);
    public Element addContent(Comment comment);
    public Element addContent(Element element);
    public Element addContent(EntityRef entityRef);
    public Element addContent(ProcessingInstruction pi);
    public Element addContent(String text);
    public boolean removeContent(CDATA cdata);
    public boolean removeContent(Comment comment);
    public boolean removeContent(Element element);
    public boolean removeContent(EntityRef entityRef);
    public boolean removeContent(ProcessingInstruction pi);

    public boolean hasChildren();
    public Element getChild(String name);
    public Element getChild(String name, Namespace ns);
    public List getChildren();
    public List getChildren(String name);
    public List getChildren(String name, Namespace ns);
```

```
    public boolean removeChild(String name);
    public boolean removeChild(String name, Namespace ns);
    public boolean removeChildren();
    public boolean removeChildren(String name);
    public boolean removeChildren(String name, Namespace ns);
    public Element setChildren(List children);

    public Attribute getAttribute(String name);
    public Attribute getAttribute(String name, Namespace ns);
    public List getAttributes();
    public String getAttributeValue(String name);
    public String getAttributeValue(String name, Namespace ns);
    public boolean removeAttribute(String name);
    public boolean removeAttribute(String name, Namespace ns);
    public Element setAttribute(Attribute attribute);
    public Element setAttributes(List attributes);

    public String getChildText(String name);
    public String getChildText(String name, Namespace ns);
    public String getChildTextTrim(String name);
    public String getChildTextTrim(String name, Namespace ns);
    public String getText();
    public String getTextNormalize();
    public String getTextTrim();
    public Element setText(String text);

    public boolean isRootElement();

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

EntityRef

Данный класс определяет модель JDOM в части ссылок на сущности, которые могут встречаться в XML-документах. Он позволяет устанавливать имя ссылки, ее публичный и системный идентификаторы, а также изменять эти параметры.

```
public class EntityRef {
    public EntityRef(String name);
    public EntityRef(String name, String publicID, String systemID);

    public Document getDocument();
    public Element getParent();
    public String getName();
    public EntityRef setName(String name);
    public String getPublicID();
    public void setPublicID(String publicID);
    public String getSystemID();
    public void setSystemID(String systemID);
```

```
    public EntityRef detach();

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

JDOMException

Данный класс представляет базовое исключение JDOM, которое служит отправной точкой для прочих исключений JDOM. Он предусматривает получение сообщений об ошибках, а также может служить оболочкой для исходного исключения, если экземпляру `JDOMException` требуется хранить такую информацию.

```
public class JDOMException extends Exception {
    public JDOMException();
    public JDOMException(String message);
    public JDOMException(String message, Throwable rootCause);

    public Throwable getCause();
    public String getMessage();
}
```

Namespace

Класс `Namespace` отвечает за отображения пространств имен в объектах JDOM.

```
public class Namespace {
    public static Namespace getNamespace(String uri);
    public static Namespace getNamespace(String prefix, String uri);

    public String getPrefix();
    public String getURI();

    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

ProcessingInstruction

Класс `ProcessingInstruction` моделирует в Java инструкции обработки XML и предоставляет связанную с ними функциональность. Он предусматривает специальную обработку информации о целевом приложении и данных инструкции. Кроме того, многие инструкции обработки используют данные в форме пар «имя-значение» (аналогично атрибутам), поэтому предусмотрено также считывание и добавление пар «имя-значение». Так, для инструкции обработки `<?cocoon-process type="xslt"?>` вызов метода `getValue("type")` приведет к получению результата `'xslt'`.

```
public class ProcessingInstruction {
    public ProcessingInstruction(String target, Map data);
    public ProcessingInstruction(String target, String data);

    public ProcessingInstruction detach();
    public Document getDocument();
    public Element getParent();
    public String getTarget();
    public String getData();
    public ProcessingInstruction setData(String data);
    public ProcessingInstruction setData(Map data);
    public String getValue(String name);
    public ProcessingInstruction setValue(String name, String value);
    public boolean removeValue(String name);

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Text

Данный класс моделирует символьные данные, элементом-«владельцем» которых является элемент JDOM `Element`. Он обычно невидим для пользователя, поскольку класс `Element` выполняет его приведение к простой строке, когда запрашивается значение узла. Он виден в методе `getContent()` класса `Element`.

```
public class Text {
    public Text(String stringValue);

    public Element getParent();
    public void append(String stringValue);
    public String getValue();
    public void setValue(String stringValue);

    public Object clone();
    public boolean equals(Object obj);
    public int hashCode();
    public String toString();
}
```

Пакет org.jdom.adapters

Данный пакет содержит адаптеры, реализующие стандартный интерфейс для получения документа DOM с помощью любого DOM-совместимого анализатора (включая анализаторы, совместимые с DOM Level 1). Адаптеры могут быть легко добавлены, если необходимо реализовать поддержку JDOM для определенного анализатора.

AbstractDOMAdapter

Данный класс реализует стандартную функциональность для метода `getDocument()`, в качестве аргумента которого выступает имя файла, путем инкапсуляции файла в классе `FileOutputStream` и последующего вызова метода `getDocument(InputStream)`.

```
public abstract class AbstractDOMAdapter implements DOMAdapter {
    public abstract Document getDocument(InputStream in, boolean validate)
        throws IOException;
    public abstract Document getDocument(File filename, boolean validate)
        throws IOException;
    public abstract Document createDocument(DocType docType)
        throws IOException;
}
```

DOMAdapter

Данный класс определяет интерфейс, который должен реализовать любой адаптер. Интерфейс включает средства для создания документа DOM на основе имени файла или потока `InputStream`, а также нового, пустого документа DOM.

```
public interface DOMAdapter {
    public abstract Document getDocument(InputStream in, boolean validate)
        throws IOException;
    public abstract Document getDocument(File filename, boolean validate)
        throws IOException;
    public abstract Document createDocument(DocType docType)
        throws IOException;
}
```

Классы конкретных адаптеров здесь не описаны, поскольку в процессе публикации этой книги в код могут быть внесены дополнения и изменения. На момент написания этих строк полнофункциональные адаптеры предусмотрены для следующих анализаторов:

- Oracle Version 1 XML Parser
- Oracle Version 2 XML Parser
- Sun Project X Parser
- Sun/Apache Crimson Parser
- Apache Xerces Parser
- IBM XML4J Parser

Пакет org.jdom.input

Данный пакет определяет классы для создания объекта JDOM `Document` на основе различных источников данных, таких как потоки SAX и существующие деревья DOM. Также он предоставляет интерфейс, позво-

ляющий применять созданные разработчиком версии классов JDOM, например подклассы Element и Attribute.

BuilderErrorHandler

Стандартный обработчик ошибок, используемый при создании документа JDOM.

```
public class BuilderErrorHandler
    implements org.xml.sax.ErrorHandler {

    public void warning(SAXParserException exception);
    public void error(SAXParserException exception);
    public void fatalError(SAXParserException exception);
}
```

DOMBuilder

Данный класс обеспечивает возможность создания объекта JDOM Document на основе источника входных XML-данных посредством анализатора, который поддерживает DOM (объектную модель документа). При этом используются разнообразные адаптеры из пакета org.jdom.adapters, поэтому если указан анализатор, для которого отсутствует адаптер, возникает ошибка. Кроме того, предусмотрен метод для создания объекта JDOM Document на основе существующего дерева DOM (экземпляра класса, реализующего интерфейс org.w3c.dom.Document). При вызове конструктора класса DOMBuilder можно потребовать выполнить проверку действительности, а также указать имя класса применяемого адаптера. Если ни то ни другое не указано, используются настройки по умолчанию: проверка действительности не производится, применяется анализатор Apache Xerces.

Также можно указать фабрику (см. описание JDOMFactory), которая должна использоваться для создания классов JDOM.

```
public class DOMBuilder {
    public DOMBuilder(String adapterClass, boolean validate);
    public DOMBuilder(String adapterClass);
    public DOMBuilder(boolean validate);
    public DOMBuilder();

    public Document build(InputStream in);
    public Document build(File file);
    public Document build(URL url);
    public Document build(org.w3c.dom.Document domDocument);
    public Element build(org.w3c.dom.Element domElement);

    public void setValidation(boolean validate);
    public void setFactory(JDOMFactory factory);
}
```

JDOMFactory

Данный интерфейс позволяет разработчикам применять собственные фабрики, создающие конструкции JDOM (`Element`, `Attribute` и т. д.). Реализация фабрики, которая была передана методу `setFactory()`, применяется для создания новых конструкций JDOM. Таким образом, достигается максимальная гибкость процесса создания объектов JDOM.

В интересах уменьшения объема и сохранения прозрачности изложения здесь не приводится очень длинный список методов этой фабрики, а читатели отсылаются к документации Javadoc. Все существующие конструкции всех классов JDOM входят в этот класс, и каждый метод возвращает объект соответствующего типа.

SAXBuilder

Данный класс обеспечивает возможность создания объекта JDOM `Document` на основе источника входных XML-данных с помощью анализатора, который поддерживает SAX (простой API для XML). `SAXBuilder` может использовать реализацию любого анализатора, поддерживающего SAX 2.0. При вызове конструктора класса `SAXBuilder` можно потребовать выполнить проверку действительности и указать имя класса применяемого драйвера SAX. Если ни то ни другое не указано, используются настройки по умолчанию: проверка действительности не производится, применяется анализатор Apache Xerces.

Также можно указать фабрику (см. описание `JDOMFactory`), которая должна использоваться для создания классов JDOM.

```
public class SAXBuilder {
    public SAXBuilder(String saxDriverClass, boolean validate);
    public SAXBuilder(String saxDriverClass);
    public SAXBuilder(boolean validate);
    public SAXBuilder();

    public Document build(InputStream in);
    public Document build(InputStream in, String systemID);
    public Document build(InputSource inputSource);
    public Document build(Reader characterStream);
    public Document build(Reader characterStream, String systemID);
    public Document build(File file);
    public Document build(URL url);
    public Document build(org.w3c.dom.Document domDocument);
    public Element build(org.w3c.dom.Element domElement);

    public void setDTDHandler(DTDHandler dtdHandler);
    public void setEntityResolver(EntityResolver entityResolver);
    public void setErrorHandler(ErrorHandler errorHandler);
    public void setExpandEntities(boolean expandEntities);
    public void setXMLFilter(XMLFilter xmlFilter);
    public void setIgnoringElementContentWhitespace(boolean ignore);
```

```
        public void setValidation(boolean validate);
        public void setFactory(JDOMFactory factory);
    }
```

Пакет org.jdom.output

Этот пакет определяет функциональность, связанную с выводом объектов JDOM `Document`. Следует отметить класс `SAXOutputter`, который позволяет объекту JDOM `Document` генерировать события SAX для приложений, которым требуется функциональность SAX, и класс `DOMOutputter`, преобразующий структуры JDOM в DOM. И, конечно, `XMLOutputter` — самый распространенный класс вывода для объектов JDOM. Подобно некоторым классам из пакета `org.jdom.input`, эти три класса имеют буквально сотни методов. Чтобы не занимать следующие десять страниц книги довольно скучной информацией, я лучше снова вспомню о документации Javadoc (доступна на сайте *http://www.jdom.org*). Она позволяет получить актуальную информацию о возможностях любого класса JDOM.

B

Возможности и свойства SAX 2.0

В данном приложении описаны стандартные *возможности* (*features*) и *свойства* (*properties*) SAX 2.0. Хотя отдельные системы, предназначенные для анализа XML-документов, могут предоставлять дополнительные возможности, позволяющие реализовать специальную функциональность, в данном списке представлен базовый набор функциональных возможностей, который должна поддерживать любая реализация анализатора, совместимого с SAX 2.0.

Основные возможности

Здесь перечислен базовый набор возможностей, поддерживаемых реализациями интерфейса `XMLReader` в SAX 2.0. Эти возможности устанавливаются при помощи метода `setFeature()`, значение же позволяет получить метод `getFeature()`. Любая возможность может иметь режим доступа только для чтения или для чтения и записи. Возможности могут быть доступны для изменения только в процессе синтаксического анализа документа либо только тогда, когда анализ не производится. Более подробно о возможностях и свойствах в SAX можно узнать из главы 4.

Обработка пространств имен

Эта возможность предписывает анализатору осуществлять обработку пространств имен, которая делает доступными префиксы и URI пространств имен, локальные имена элементов в методах обратного вызова SAX `startPrefixMapping()` и `endPrefixMapping()`, а также в некоторых параметрах, передаваемых методам `startElement()` и `endElement()`. Если эта возможность имеет значение `true`, такая обработка будет производиться. Если эта возможность имеет значение `false`, обработка пространств имен не производится (подразумевается, что возможность

отслеживания префиксов пространств имен включена). По умолчанию в большинстве анализаторов данная возможность включена.

URI: *http://xml.org/sax/features/namespaces*

Режим доступа: только чтение, когда осуществляется синтаксический анализа документа; чтение и запись, когда анализ не производится.

Отслеживание префиксов пространств имен

Эта возможность предписывает анализатору сообщать об атрибутах, применяемых при объявлении пространств имен, таких как `xmlns:[URI пространства имен]`. Когда эта возможность отключена (имеет значение `false`), об атрибутах, связанных с пространствами имен, не сообщается, поскольку анализатор использует их для определения соответствий префиксов пространств имен и URI, и они обычно не имеют ценности для вызывающего приложения в таком контексте. Кроме того, когда обработка пространств имен включена, отслеживание префиксов обычно выключают. По умолчанию в большинстве анализаторов данная возможность отключена.

URI: *http://xml.org/sax/features/namespace-prefixes*

Режим доступа: только чтение, когда производится синтаксический анализ документа; чтение и запись, когда анализ не производится.

Канонизация строк

Данная возможность предписывает анализатору обрабатывать все полные и локальные имена элементов, префиксы и URI пространств имен с помощью метода `java.lang.String.intern()`. Когда эта возможность установлена в `false`, все компоненты XML остаются в первоначальном состоянии. В более новых, высокопроизводительных анализаторах эта возможность обычно отключена по умолчанию – в пользу прочих видов оптимизации работы с символьными данными.

URI: *http://xml.org/sax/features/string-interning*

Режим доступа: только чтение, когда осуществляется синтаксический анализ документа; чтение и запись, когда анализ не производится.

Проверка действительности

Эта возможность предписывает выполнение проверки действительности документа. При этом все ошибки, возникающие в результате нарушения ограничений, регистрируются посредством интерфейса SAX `ErrorHandler`, если существует его реализация, связанная с экземпляром анализатора. Если данная возможность установлена в `false` (как правило, это установка по умолчанию), проверка действительности не

производится. Чтобы выяснить, применима ли эта возможность к проверке действительности по DTD и/или схеме, обратитесь к документации своего анализатора.

URI: *http://xml.org/sax/features/validation*

Режим доступа: только чтение, когда осуществляется синтаксический анализ документа; чтение и запись, когда анализ не производится.

Обработка внешних сущностей (общих)

Эта возможность предписывает обработку всех общих (текстовых) сущностей в XML-документе. В большинстве анализаторов по умолчанию включена.

URI: *http://xml.org/sax/features/external-general-en*tities

Режим доступа: только чтение, когда производится синтаксический анализ документа; чтение и запись, когда анализ не производится.

Обработка внешних сущностей (параметрических)

Данная возможность предписывает выполнение анализа всех внешних параметрических сущностей, включая те из них, которые определены во внешних DTD. В большинстве анализаторов по умолчанию включена.

URI: *http://xml.org/sax/features/external-parameter-entities*

Режим доступа: только чтение, когда производится синтаксический анализ документа; чтение и запись, когда анализ не производится.

Базовые свойства

Свойства позволяют влиять на объекты, используемые в процессе анализа; в частности, речь идет о таких обработчиках, как `LexicalHandler` и `DeclHandler`, которые не входят в базовый набор обработчиков SAX 2.0 (`EntityResolver`, `DTDHandler`, `ContentHandler` и `ErrorHandler`). Любое свойство может иметь режим доступа «только чтение», либо «чтение/запись». Свойства также могут быть доступны для изменения только в процессе синтаксического анализа документа либо только тогда, когда анализ не производится.

Лексический обработчик

Данное свойство позволяет устанавливать и получать ссылку на реализацию интерфейса `LexicalHandler`, которая используется для обработки комментариев и указаний на DTD в XML-документе.

URI: *http://xml.org/sax/properties/lexical-handler*

Тип данных: `org.xml.sax.ext.LexicalHandler`

Режим доступа: чтение и запись, когда производится синтаксический анализ документа; чтение и запись, когда анализ не производится.

Обработчик объявлений

Данное свойство позволяет устанавливать и получать ссылку на реализацию интерфейса `DeclHandler`, используемую для обработки ограничений, определенных в DTD.

URI: *http://xml.org/sax/properties/declaration-handler*

Тип данных: `org.xml.sax.ext.DeclHandler`

Режим доступа: чтение и запись, когда производится синтаксический анализ документа; чтение и запись, когда анализ не производится.

Узел DOM

В процессе синтаксического анализа документа это свойство возвращает текущий узел DOM (если используется итератор DOM). Когда анализ не производится, данное свойство возвращает корневой узел DOM. Большинство анализаторов, которые я применял для тестированиия программ из этой книги, за исключением особых случаев не поддерживают это свойство. Я бы не стал полагаться на него, когда речь заходит о важной информации.

URI: *http://xml.org/sax/properties/dom-node*

Тип данных: `org.w3c.dom.Node`

Режим доступа: только чтение, когда производится синтаксический анализа документа; чтение и запись, когда анализ не производится.

Строковая константа XML

Данное свойство содержит строковую константу из XML-документа, которая вызвала событие, обрабатываемое в данный момент. Как и в случае с узлом DOM, я очень редко встречал поддержку этого свойства в анализаторах. Не стоит рассчитывать на него, особенно при смене анализатора.

URI: *http://xml.org/sax/properties/xml-string*

Тип данных: `java.lang.String`

Режим доступа: только чтение, когда производится синтаксический анализ документа; только чтение, когда анализ не производится.

Алфавитный указатель

Symbols

!, тег, 42
#PCDATA, ключевое слово, 43
& (амперсанд), 482
&amp, 482
* (модификатор повторения), 42
+ (модификатор повторения), 42
. (точка), разбиение имен свойств, 201
/, слеш, 32
:expr>, тег, 311
? (модификатор повторения), 42
@, префикс имени атрибута, 56

A

AbstractDOMAdapter, класс, 524
acceptNode(), метод, 174
addContent(), метод, 201, 210
addEvent(), метод, 338, 339
addHandler(), метод, 331
AddItemServlet, сервлет, 457
Adobe Acrobat, 285
Apache Axis, 362
Apache SOAP, 361
    WSDL, ограниченная функциональность, 391
API
    высокоуровневые, 25
    низкоуровневые, 25
API (интерфейс прикладного программирования), 21
appendChild(), метод, 159
apply-templates, конструкция, 55
ATTLIST, ключевое слово, 43
Attr, интерфейс, 502
Attribute, класс, 517
Attribute, класс (JDOM), 203
AttributeList, интерфейс (SAX), 489
AttributeListImpl, класс, 498
Attributes, интерфейс, 489
Attributes, интерфейс (SAX), 78
AttributesImpl, класс, 498
Axis SOAP, набор инструментов, 362

B

baseName, переменная, 212
BeanSerializer, класс, 379
binding, элемент, 391
BuilderErrorHandler, класс, 525

C

card, элемент, 294
Castor
    AddItemServlet, сервлет, 457
    классы, порождение, 453
    упаковка и распаковка, 455
    установка, 452
Castor, система, 452
CDAdder, класс, 372
    обновленная версия, 379
CDATA, класс, 517
CDATA, тип, 43
CDATASection, интерфейс, 502
CDLister, класс, 374
CDOrderer, клиент, 408
    тестирование, 410
CDs-R-Us, служба сообщений
    импортированные классы, 398
channel, элемент (RSS), 435
CharacterData, интерфейс, 503
characters(), метод, 80
    обработка пробельных символов, 82
    символьные массивы, 96
ClassNotFoundException, исключение, setDriver(), метод, 329
Cocoon
    Tomcat, настройка

контекст, 280
WML, вывод, 297
версия 2.0+, 316
настройка, 279
применение, 277
форматирование в зависимости от браузера, 290
Cocoon, система веб-публикации, 278
cocoon.properties, файл, 290
Comment, интерфейс, 503
Comment, класс, 209, 518
compareDocumentOrder(), метод, 185
compareTreePosition(), метод, 185
ContentHandler, интерфейс, 489
ContentHandler, интерфейс (SAX), 70
contents.xml, файл, 292
таблица стилей для WAP-устройств, 294
copy-of, конструкция, 57
count(), функция, 53
createDocType(), метод, 159
createDocument(), метод, 159, 167
createElement(), метод, 159
createElementNS(), метод, 167
createRange(), метод, 177
createTextNode(), метод, 159
createXMLRepresentation(), метод, 198, 199
CSS (каскадные таблицы стилей), 274

D

DataWriter, класс, 118
DeclHandler, интерфейс, 124, 496
decorators, 234
DefaultHandler, класс, 112, 261, 499
deleteContents(), метод, 178
DocType, класс, 518
Document, интерфейс, 141, 503
Document, класс, 518
Document, узел, 141
DocumentBuilder, класс, 409, 510
DocumentBuilderFactory, класс, 510
JAXP 1.1, 261
DocumentFragment, интерфейс, 504
DocumentHandler, интерфейс, 490
DocumentRange, интерфейс, 177
DocumentType, интерфейс, 504
doGet(), метод, 164
DOM, 21
отличия от SAX, 132
DOM (Объектная модель документа), 129
HTML, и, 182
JAXP
версии, 251
версия 1.0, 256
методы, Level 2, 263
Level 2, модули
Events, 179
Style, 182
Traversal, 171
Views, 181
Level 2, модуль
Range, 176
Level 3, 183
самозагрузка, 186
wrong document, исключение, 187
анализаторы, 135
вывод, 136
дерево
потребление памяти, 134
создание, 153
деревья
изменение, 160
изменяемость, 153
сериализация, 135
дополнительные темы, 152
замедленный DOM, 149
и память, 149
модули Level 2, 168
объявления пространств имен, 167
перегрузка, 164
поддержка пространств имен, Level_2, 164
предостережение при использовании большого объема данных, 149
распространенные проблемы, 187
реализация для Java, 130
свободные узлы, 188
спецификация, Level 1 и 2, 169
стандартизация, 218
структуры, сравнение, 186
частные реализации, 130
DOM (объектная модель документа)
Level 2, 501
анализаторы
Level_, 168
отличия от SAX, 129
DOMAdapter, интерфейс, 524
DOMBuilder, класс, 193, 525
производительность, 220

DOMException, класс, 505
DOMFaultListener, объект, 383
DOMImplementation, интерфейс, 505
dom-node, свойство, 106
DOMOutputter, 193, 227
DOMResult, класс, 515
DOMSerializer, класс, 138, 148
  каркас, 138
DOMSerializer.java, файл, 139
DOMSource, класс, 515
DOM-совместимые анализаторы
  генерация исключений SAX, 150
doPost(), метод, 154, 177
  пространства имен, 165
DTD
  и проверка действительности, 102
DTD (??????????? ???? ?????????)
  ELEMENT, ???????????, 42
DTD (Описание типа документа)
  дополнительные модели
    ограничений, 48
DTD (Определение типа документа), 40
  атрибуты, 43
  модификаторы повторения, 42
  сущности, 43
DTD (определение типа документа)
  RSS, 434
DTDHandler, интерфейс, 111, 491
  и проверка действительности, 126

E

Element, интерфейс, 506
Element, класс, 228, 519
  setAttribute(), метод (JDOM), 203
ELEMENT, ключевое слово, 42
Element, конструктор, 201
EMPTY, ключевое слово, 43
endDocument(), метод, 72
endPrefixMapping(), метод, 75
Entity, интерфейс, 506
ENTITY, ключевое слово, 43
EntityRef, класс, 225, 521
EntityReference, интерфейс, 507
EntityResolver, интерфейс, 107, 491
enumerateItem(), метод, 455
equals(), методы, 228
equalsNode(), метод, 185
ErrorHandler, интерфейс, 491
ErrorHandler, интерфейс (SAX), 87
ErrorListener, интерфейс, 265, 513
Events, модуль, 179
exclude-result-prefixes, атрибут, 297
execute(), метод, 334
expandEntities, флаг, 226
extractContents(), метод, 178

F

FactoryConfigurationException,
  класс, 511
fault.getString(), метод, 382
Foobar, библиотека, 413
FOP (процессор объектов
  форматирования), 284
for-each, конструкция, 56
FormattingNodeFilter, класс, 174

G

generateRSSContentMethod(),
  метод, 440
getAttribute(), метод, 265
getAttributes(), метод, 143
getAttributeValue(), метод, 216
getBody(), метод, 399
getChild(), метод, 201
getChildNodes(), метод, 142
getChildren(), метод, 212
getColumnNumber(), метод, 71
getContent(), метод, 215, 216
getDetailEntries(), метод, 383
getDocument(), метод, 137
getDocumentElement(), метод, 142
getDraftDate(), метод, библиотеки
  тегов, 311
getElementsByTagNameNS(), метод, 167
getException(), метод, 73
getFeature(), метод, 99, 263, 528
getFirstChild(), метод, 164
getInstance(), метод, 354
getItem(), метод, 455
getLineNumber(), метод, 71
getLocalName(), метод, 78
getNamespace(), метод (JDOM), 227
getNamespaceURI(), метод, 164
getNodeName(), метод, 142
getNodeType(), метод, 140
getNodeValue(), метод, 150
    текстовые данные, вывод, 144
getParent(), метод, 114
getPrefix(), метод, 164, 165
getProperty(), метод, 99, 198, 263

getRootElement(), метод, 212
getText(), метод, 213
getTextTrim(), метод, 212
getTitle(), метод, библиотеки тегов, 311
getURI(), метод, 78
getXMLReader(), метод, 261
getXPath(), метод, 239

H

HandlerBase, класс, 252, 492
Hashtable, класс, 195, 338, 348
helma.xmlrpc.WebServer, класс, 328
helma.xmlrpc.XmlRpcClient, класс, 332
helma.xmlrpc.XmlRpcServer, класс, 327
HTML и DOM, 182
HTTP протокол
  примнение для межкорпоративного взаимодействия, 413
HTTP, протокол
  RPC и, 323

I

ignorableWhitespace(), метод, 82
IIOP, протокол, 322
include, оператор (Java), 310
indentLevel, переменная, 140
InputSource, класс, 493
IOException, исключение, 151
isNamespaceAware(), метод, 255
isSameNode(), метод, 185
isValidating(), метод, 255
ItemSearcher, класс, 172

J

Jakarta Tomcat, среда исполнения сервлетов, 279
jar-файлы, 61
Java
  Castor, система, 452
  DOM, 130
  SerializerTest.java, исходный файл, 135
  XML, и, будущие разработки, 478
  реализация SOAP, 361
  совместимые операционные системы, 23
  типы данных, поддержка в XML-RPC, 327
  файлы свойств, 195
    XML, преобразование в, 196
  формы вывода, 138
java.io.IOException, класс, 335
java.net.MalformedURLException, исключение, 333
java.text.SimpleDateFormat, класс, 339
javax.xml.parsers, пакет, 510
javax.xml.transform, пакет, 513
javax.xml.transform.dom, пакет, 515
javax.xml.transform.sax, пакет, 516
javax.xml.transform.stream, пакет, 516
JavaXML.wml.xsl, файл, 295
JAXB (Архитектура Java для связывания данных), 469
JAXP (Java API для анализа XML), 510
  TrAX API, 263
  версия 1.1, 260
    DefaulstHandler, класс, 261
    изменения по сравнению с версией 1.0, 260
    обновления, 263
  классы, 250
JAXP (Java API для обработки XML), 249
  версии и анализаторы, 251
  версия 1.0, 251
    DOM, 256
    SAXParser, класс, 255
    SAXParserFactory, класс, 253
    анализаторы, смена, 259
  сравнение с другими API, 249
JCP (Java Community Process), 218
JDK, версии, 23
JDOM (Объектная модель документа в Java
  значения null, 219
JDOM (Объектная модель документа в Java), 222
  Attribute, класс, 203
  DOM, отличия, 219
  DOM, отличия от, 189
  Element, класс, 228
  EntityRef, класс, 225
  Java Collections, поддержка, 191
  JDOM Beta 7, 223
  JDOMNode, интерфейс, 234
  Namespace, класс, 226
  PropsToXML, класс, 195
  SimpleXPathViewer, класс, 242, 245
  Text, класс, 223
  XML, вывод, 204, 205

XML, дерево, 190
XML, сохранение, 208
XMLProperties, класс, 217
XpathDisplayNode, класс, 239
архитектура, 222
ввод/вывод, модель, 192
интерфейсы, и, 233
итерация и класс Text, 224
классы, 190
подклассы, 246
программа просмотра документа, реализация, 241
стандартизация, 217
фабрики, 228
создание, 229
JDOM (объектная модель документа в Java)
версия 1.0, 516
JDOM (Объектная модель документов в Java)
XMLProperties, класс, 206
JDOMException, класс, 522
JDOMFactory, интерфейс, 230, 526
JDOMNode, интерфейс, 234
JNDI (Java Naming and Directory Interface), 322
JRMP, протокол, 321
JSP (Серверные страницы Java), 300
JSP (серверные страницы Java), 274
JSR (Java Specification Request), 218

L

LDAP (облегченный протокол доступа к каталогам), 322
LexicalHandler, интерфейс, 497
LexicalHandler, класс, 120
Linux/Unix, Cocoon, скомпилированная версия, 279
list(), метод, 368
обработка ошибок, 383
load(), метод, 206, 211
loadFromElements(), метод, 212
Locator, интерфейс, 95, 493
Locator, класс, 71
LocatorImpl, класс (SAX), 499

M

mappings, элемент, 378
mapTypes(), метод, 381
marshal(), метод, 475
marshall(), метод, 402
message, элемент, 391
Microsoft
Windows, Cocoon, скомпилированная версия, 279
реализация SOAP, 362

N

NamedNodeMap, интерфейс, 507
NamedNodeMap, объект, 143
Namespace класс (JDOM), 522
Namespace, класс (JDOM, 226
NamespaceSupport, класс, 499
newSAXParser(), метод, 253
newTransformer(), метод, 266
Node, интерфейс, 140, 507
NodeFilter, интерфейс, 174
NodeIterator, интерфейс, 172
not(), функция, 53
Notation, интерфейс, 509
NoteList, интерфейс, 509

O

OrderProcessor, класс, 409
org.apache.soap.Body, класс, 399
org.apache.soap.Envelope, класс, 399
org.jdom, пакет, 516
org.jdom.adapters, пакет, 523
org.jdom.input, пакет, 524
org.jdom.output, пакет, 527
org.w3c.dom, пакет, 502
org.w3c.dom.Document, объекты, 137
org.w3c.dom.Node, интерфейс, 150
org.xml.sax, пакет, 103, 488
org.xml.sax.Attributes, интерфейс, 77
org.xml.sax.ContentHandler, 70
org.xml.sax.DTDHandler, 70
org.xml.sax.EntityResolver, 70
org.xml.sax.ErrorHandler, 70
org.xml.sax.ext, пакет, 496
org.xml.sax.HandlerBase, класс, 252
org.xml.sax.helpers,, 498
org.xml.sax.helpers.ParserAdapter, класс, 93
OutputKeys, класс, 513

P

-package, ключ, 454
parse(), метод, 137

SAXParser, интерфейс, 261
Parser, интерфейс, 493
ParserAdapter, класс, 500
ParserConfigurationException, класс, 511
ParserFactory, класс, 252, 500
PDF (переносимый формат документов)
получение из XML, 284
po.xsdXMLSchema, 393
portType, элемент, 391
ProcessingInstruction класс (JDOM), 522
ProcessingInstruction, интерфейс, 509
ProcessingInstruction, интерфейс (DOM), 144
propertyNames(), метод, 198
PropsToXML, класс, 195
каркас, 196
PUBLIC, ключевое слово, 30

R

Range, модуль, 176
removeChild(), метод, 163
removeEvent(), метод, 338, 339
request, переменная (XSP), 306
resolveEntity(), метод, 126
Result, интерфейс, 513
RMI (Удаленный вызов методов)
вызов методов, 330
RMI (удаленный вызов методов), 321
RPC, сравнение, 321
RPC (Удаленные вызовы процедур)
SOAP, применение, 367
RPC (Удаленный вызов процедур), 322
обработчики, регистрация, 331
ограничения, 323
RPC (удаленный вызов процедур), 320
RMI, сравнение, 321
RSS (Rich Site Summary), 434
каналы, 434

S

save(), метод, 206
SAX, 21, 59
SAX (Простой API для XML)
HandlerBase, класс, 252
JAXP, версии, 251
ParserFactory, класс, 252
XML-RPC сервер, драйвер для, 328
версия 2.0, 488
возможности, 528
свойства, 530
исключение, генерация DOM-совместимыми анализатор, 150
ограничения при работе с XML, 153
стандартизация, 218
фильтры, 113, 118
SAX (простой API для XML)
XSLT, ограничения, 134
анализаторы, 62
версия 2.0
отсутствие поддержки в анализаторах, 93
классы, 61
одноуровневые элементы и, 133
последовательная модель, 132
причины применения, 134
SAX 2.0
стандарты XML, поддержка, 99
SAXBuilder, класс, 193, 526
SAXException, исключения, 73
SAXException, класс, 73, 494
SAXNot SupportedException, исключение, 103
SAXNotRecognizedException, исключение, 103
SAXNotRecognizedException, класс, 494
SAXNotSupportedException, класс, 494
SAXOutputter, 206, 227
SAXOutputter, класс, 193
SAXParseException, класс, 87, 495
SAXParser, интерфейс
parse(), метод, 261
SAXParser, класс, 255, 511
SAXParserFactory, класс, 253, 512
JAXP 1.1, 261
Scheduler, класс, 338
SchedulerClient, класс, 349
schemaLocation, атрибут, 40
search(), метод, 175
serialize(), метод, 139, 140, 147
serializeNode(), метод, 140
SerializerTest, класс, 136
server.xml, файл, 280
service, элемент, 391
setAttribute(), метод, 159, 203, 265
setContentHandler(), метод, 85
setDocumentLocator(), метод, 95
setDriver(), метод, 329, 332
ClassNotFoundException, исключение, 329

setEndAfter(), метод, 178
setErrorHandler(), метод, 89
setErrorListener(), метод, 265
setExpandEntities(), метод, 226
setFactory(), метод, 231
setFeature(), метод, 99, 103, 262, 528
SetNameSpaceProcessing(), метод, 103
setParent(), метод, 114
setProperty(), метод, 99, 263
setRootPart(), метод, 402
setStartBefore(), метод, 178
setText(), метод, 201
  элементы, вызов, 223
SimpleXPathViewer, класс, 242, 245
SOAP (Простой протокол доступа к объектам), 357
  Apache Axis, 362
  Apache, stackTrace, элемент, 384
  DOMFaultListener, 383
  IBM SOAP4J, 362
  RPC службы
    обновление, 370
    развертывание, 370
  RPC-вызовы, 367
  RPC-клиенты, 371
  вызов, 360
  кодирование, 359
  конверт, 359
  обработка ошибок, 382
  пользовательские параметры, 376
    encodingStyle, атрибут, 378
  проект системы, 366
  различия реализаций, 363
  реализация Microsoft, 362
  реализация от Apache, 361
  уст
    -_, 365
  установка, 363
    клиенты, 363
    сервлет-маршрутизатор, 365
SOAP (Простой протокол доступа к объектам)основные компоненты, 358
SOAP, протокол, 22
SOAPMappingRegistry, класс, 381
SOAP-RPC, вызовы, обязательные действия, 371
Source, интерфейс, 514
SourceLocator, интерфейс, 514
startDocument(), метод, 72
startElement(), метод, 77
startPrefixMapping(), метод, 75
store(), метод, 206, 209, 215
StreamResult, класс, 268
StreamSource, класс, 267
Style, модуль, 182
switch, конструкция (Java), 141
SYSTEM, ключевое слово, 30

T

template, ключевое слово, 55
Templates, интерфейс, 514
test(), метод, 137
TestXMLProperties, класс, 215
Text, интерфейс, 509
Text, класс, 523
Text, класс (JDOM), 223
text, конструкция, 56
Tomcat, среда исполнения сервлетов, 279
  SOAP, установка сервлетов, 365
  библиотеки, загрузка, 280
    jar-файлы, 282
Tomcat, среда исполнения серлетов
  контекст, настройка, 280
transform(), метод, 267
Transformer, интерфейс, получение экземпляра, 266
Transformer, класс, 514
TransformerFactory, класс, 264, 515
Traversal, модуль, 171
TrAX API, 263
TreeWalker, интерфейс, 176
try/catch, блоки (JDOM), 220
types, элемент, 391

U

UDDI, 22
UDDI (Universal Discovery, Description, and Integration), 387
  реализации, 388
  реестр, 387
    поиск, 405
unmarshal(), метод, 475
UpdateItemServlet, класс, 154, 177
URI
  идентификатор свойства и возможности, 99
URIResolver, интерфейс, 265, 515
URL
  API Javadocs, 62
  Xerces, анализатор, 60

V

value-of, конструкция, 55
Vector, класс, 338
Views, модуль, 181

W

W3C, стандартизация XML, 18
WAP
  спецификация, 295
WAP (протокол беспроводных приложений), 299
WAP SDK, программный продукт, 295
WebServer, класс, 328
WML (Язык разметки для беспроводных устройств), 294
  источники дополнительной информации, 299
  таблицы стилей, 295
wml, элемент, 294
wrappers, 234
WSDL, 22
WSDL (Язык описания веб-служб), 389
  дальнейшая разработка, 407
  схемы XML, текущая версия, 391
  файлы, данные в, 389
  элементы, 391

X

Xalan, процессор, 430
Xerces, анализатор, 60
XLink, 478
xlink:actuate, атрибут, 480
xlink:href, атрибут, 480
xlink:show, атрибут, 480
xlink:type, атрибут, 479
XML
  недавние разработки, 9
  ограничения, 40
  пространства имен, 40
  технологии, 58
  элементы, 32
XML (Расширяемый язык разметки), 17
  HTML-формы, создание данных, 414
  Java, и, будущие разработки, 478
  XPointer, ссылки в документах, 481
  взаимодействие, 18
  возможности взаимодействия, 386
  отображения для схем XML, 485
  переносимость, 18
  преобразования, 266
  приложения business-to-business, реальная модель, 412
  пространства имен как средство идентификации, 38
  разделяемые данные, приложения и компании, 412
  связывание данных, 22, 444
    Zeus, 460
  системы веб-публикации, 273
  сообщения, 392
  стандартизация, 18
  форматирование в зависимости от браузер, 290
  фрагменты документа, работа с, 420
XML (расширяемый язык разметки)
  PDF-файлы, преобразование, 284
  ошибки, связанные с версией XML, 91
  просмотр в формате HTML, 284
XML (Расширяемый язык разметки), приложения на основе, 22
XML 1.0, рекомендация, 28
XML Schema
  SOAP, интеграция, 360
XML схема
  schemaLocation, атрибут, 40
  спецификация, 39
XML, объявления
  DOM Level 3, 184
XML, преобразования, 264
XMLFilter, интерфейс, 495
XMLFilter, класс, 113, 118
XMLFilterImpl, класс, 114, 500
XMLOutputter, 227
XMLOutputter, класс, 194, 205
XMLProperties, класс, 206, 217
  каркас, 206
  несовместимость версий, 214
  тестирование, 213
XMLReader, интерфейс, 495
  методы для установки, 99
XMLReader, интерфейс (SAX), 99
XMLReaderAdapter, класс, 501
XMLReaderFactory, класс, 501
XMLReaders, конвейер событий, 114
XML-RPC, 22, 320
  библиотеки, 325
  вызовы, 360
  и XML, 353
  классы сервера, 337

конфликт портов, 352
клиент, 332, 349
обработчик, класс, 327
обработчики, 326, 337
преимущества, 323
простейшее приложение, 324
сервер, 328
файл настройки, 344
сервлеты, сравнение, 355
типы данных Java, реализованные, 327
XmlRpc, класс, 328
XmlRpcClient, класс, 333
XmlRpcException, класс, 335
xml-string, свойства, 106
XMLUtils, класс, 409
XMLWriter, класс, 118
XML-анализаторы, 23
DOM возможности, поддержка, 170
XML-документы, 30
DTD, 41
анализ, 66
действительные, 33
корректные, 32
проверка действительности, 102
содержимое, обработка посредством связывания данных, 447
указатель позиции, 71
XPath, 52
XpathDisplayNode, класс, 239
XPointer, 480
сокращенная форма, 483
XSL, 49
деревья, 49
соответствие шаблонам, 54
форматирующие объекты, 50
циклы, 56
XSL (Расширяемый язык таблиц стилей)
WML-страницы, инструменты разработки, 295
веб-разработка, 275
XSL (расширяемый язык таблиц стилей)
таблицы стилей
в приложениях business-to-business, 424
шаблоны
в приложениях business-to-business, 427
xsl:apply-templates, конструкция, 134
xsl:stylesheet, элемент, 297
XSLT, 51, 134
XSLT (Преобразования XSL), 275
XSLTProcessor, класс, 430
XSLTProcessorFactory, класс, 431
XSP (Расширяемые серверные страницы), 300
библиотеки тегов, использование, 309
создание страниц, 302
таблицы стилей XSL, 301
xsp:include, структура, 310
xsp:logic, элемент, 301
xsp:structure, структура, 310

Z

Zeus
классы, порождение, 462
упаковка и распаковка, 466
установка, 461
Zeus, система, 460

А

Анализ
поток событий, 114
Анализ XML-документов, 66
Анализатор
получение, 60
Анализаторы, 23
DOM, 135
DOM Level_2, 168
Sun, 250
возможности, фирменные, 106
не поддеживающие SAX 2.0, 93
независимость от производителя, и, 251
свойства, фирменные, 106
стандартизация, недостаток, 99
анализаторы XML, DOM и, 131
Анализаторы без проверки действительности
пропускаемые сущности, 84
Архитектура, JDOM, 222
Атрибуты, 43, 264
атрибуты, 34
преимущества перед элементами, 35
атрибуты элементов (см. атрибуты), 34

Б

Библиотеки тегов

использование, 309
Быстродействие
анализаторы XML, 24

В

Веб-публикации система
веб-разработка, 275
Веб-публикации, система, 273
Веб-службы, 385
SOAP, 391
UDDI, регистрация, 403
альтернативы, 412
клиенты, создание, 408
пример, 392
реестр UDDI, поиск, 405
службы сообщений, 392
тестирование клиента, 410
Взаимодействие приложений, 386
Возможности, 100
канонизация строк, 105
проверка действительности, 101
Вызов в SOAP, 360

Г

генераторы, 318

Д

Данные
XML-документы, заголовки, 31
данные
в элементах, 80
Дескрипторы развертывания, 369
Действительность, проверка, 101
динамическое создание
наполнения, 275
Документ неверный, исключение, 187
Документы, начало/конец, 72
Древовидная структура
обработка XSL, 50

З

загрузка
WAP.SDK, программный
продукт, 295
зарезервированные символы,
маскировка, 36
Значения свойств, преобразование в
атрибуты, 203

И

Идентификатор, публичный, 146
Идентификатор, системный, 146
Идентификаторы, публичные, 30
Имена свойств, разбиение по точке, 201
имена элементов, 78
Инструкции обработки
DOM и, 144
XML-документы, 31
Интерфейс
DocumentRange, 177
Интерфейсы, 21
DeclHandler, 124
Document, 141
DTDHandler, 111, 126
EntityResolver, 107
ErrorListener, 265
JDOMFactory, 230
JDOMNode, 234
Node, 140
NodeFilter, 174
NodeIterator, 172
SAXParser, 261
TreeWalker, 176
URIResolver, 265
Исключения, перехват, 104

К

Каналы, 434
Канонизация строк, возможность
SAX, 529
Карта сайта, 317
Карты (WML), 294
Класс
регистрация (XML-RPC), 331
Класс, экземпляры
разделяемые, 354
Классы
Attribute, 203
BeanSerializer, 379
CDAdder, 372
обновленная версия, 379
CDLister, 374
Comment, 209
DataWriter, 118
DefaultHandler, 112
DefaultsHandler, 261
DocumentBuilder, 409
DocumentBuilderFactory, 261
DOMBuilder, 193

производительность, 220
DOMSerializer, 138, 148
Element
SetAttribute(), метод, 203
создание подклассов, 228
EntityRef, 225
FormattingNodeFilter, 174
HandlerBase, 252
Hashtable, 195
ItemSearcher, 172
JDOM, для, 190
LexicalHandler, 120
Namespace, 226
OrderProcessor, 409
ParserFactory, 252, 253
PropsToXML, 195
SAXBuilder, 193
SAXParser, 255
SAXParserFactory, 261
SerializerTest, 136
SimpleXPathViewer, 242, 245
SOAPMappingRegistry, 381
StreamResult, 268
StreamSource, 267
TestXMLProperties, 215
Text, 223
TransformerFactory, 264
UpdateItemServlet, 154, 177
XMLFilter, 113, 118
XMLFilterImpl, 114
XMLOutputter, 205
XMLProperties, 206, 217
каркас, 206
несовместимость версий, 214
тестирование, 213
XmlRpc, 328
XMLUtils, 409
XMLWriter, 118
XPathDisplayNode, 239
XSLTProcessor, 430
XSLTProcessorFactory, 431
обработчики, XML-RPC, 327
порождение, 447
Классы анализатора XML, 61
Классы обработчиков, файл настройки сервера XML-RPC, 344
Классы создания, 193
Кодирование, SOAP, 359
Колоды (WML), 294
Конверты (SOAP), 359
Конвейер, 114
Конец документа, 72
Конструкторы, класс JDOM Element, 230
Корневой элемент, 435
критические ошибки, 90

Л

Лексический обработчик, свойство SAX, 530

М

Методы
addContent(), 201, 210
addEvent(), 339
createDocument(), 159
createXMLRepresentation(), 198, 199
enumerateItem(), 455
equals(), 228
fault.getString(), 382
generateRSSContentMethod(), 440
getAttribute(), 265
getAttributeValue(), 216
getBody(), 399
getChild(), 201
getChildNodes(), 142
getChildren(), 212
getContent(), 215, 216
getDetailEntries(), 383
getDocumentElement(), 142
getFeature(), 263
getInstance(), 354
getItem(), 455
getNodeName(), 142
getProperty(), 198, 263
getRootElement(), 212
getText(), 213
getTextTrim(), 212
getXMLReader(), 261
getXPath(), 239
isNamespaceAware(), 255
isValidating(), 255
list(), 368, 383
load(), 206, 211
loadFromElements(), 212
mapTypes(), 381
marshal(), 475
marshall(), 402
newSAXParser(), 253
newTransformer(), 266
propertyNames(), 198

RMI, вызов, 321
save(), 206
setAttribute(), 203, 265
setDriver(), 329
setErrorListener(), 265
setExpandEntities(), 226
setFeature(), 262
setProperty(), 263
setText(), 201
store(), 206, 209, 215
transform( ), 267
unmarshall(), 475
XML, вывод, форматирование, 205
XMLReader, интерфейс, 99
XML-RPC, вызов, 327
названия и службы сообщений, 395
удаленные вызовы, 330
Модель содержимого, 42
Модель форматирования, 50
Модификаторы повторения, 42
Модуль кода, 367

Н

наборы узлов, 53
Начало документа, 72
неанализируемые данные, 37
некритические ошибки, 90

О

Обработка внешних сущностей, возможность SAX, 530
Обработка ошибок, SOAP, 382
Обработка сущностей, 105
Обработчик объявлений, свойство SAX, 531
Обработчики
DTDHandler, 111
EntityResolver, 107
XML-RPC, 327
Обработчики ошибок, 87
Обработчики содержимого, 69
Объявление XML
DOM Level 2, и, 142
Ограничения, 40
Операционные системы, совместимость Java, 23
Относительные пути, 52
Отображение префикса, 74
Отслеживание префиксов пространств имен, возможность SAX, 529
ошибки
версия XML и, 91
критические, 90
некритические, 90

П

Пакеты
com.sun.xml.parser, 250
helma.xmlrpc, 325
javax.xml.parsers, 250
javax.xml.transform, 264
org.apache.xalan.xslt, 430
org.enhydra.zeus.binder, 463
org.jdom.input, 193
org.jdom.output, 193
org.w3c.dom, 130
org.w3c.dom.css, 182
org.w3c.dom.events, 179
org.w3c.dom.html, 182
org.w3c.dom.ranges, 177
org.w3c.dom.stylesheets, 182
org.w3c.dom.traversal, 172
org.w3c.dom.views, 181
org.xml.sax, 103
org.xml.sax.ext, 120
SAX, 488
Память и DOM, 134
память, DOM и, 149
Подклассы, JDOM, 246
Потоки, 343
предупреждения, 89
Преобразования, 48
Преобразования XSL, 51
Пробелы
RSS и, 436
пробельные символы
обработка, 82
Проверка действительности, возможность SAX, 529
Пролог, 30
пропускаемые сущности, 84
SAX и, 93
Пространства имен, 102, 164
XML, 40
методы SAX, 74
объявления, 167
пространства имен
идентификацияXML-дано, 38
Пространства имен, методы, 164
Пространства имен, обработка,

возможность SAX, 528
Пространства имен, объявление
WML, 297
Пространство имен
спецификация схемы, 39
Пространство имен, по умолчанию (JDOM), 227
процессоры, 318
Пункт, 434

Р

Распаковка, 449
Рассылка данных клиентам, 433
регистраторы ошибок, 370
Регистрация служб, 388, 403
Реестр служб, 386

С

Свойства
строковые константы, 106
узлы DOM, 106
свойства, 100
Свойства и возможности, методы для установки, 99
Связывание данных, 22, 444
JAXB, 469
порождение классов, 447
система Castor, 452
классы, порождение, 453
упаковка и распаковка, 455
установка, 452
система Zeus, 460
классы, порождение, 462
упаковка и распаковка, 466
установка, 461
упаковка и распаковка, 449
серверные страницы Java (JSP), 274, 300
Сервлеты
XSLT, из, 429
разработка, 274
сравнение с XML-RPC, 355
Сериализация, 138
рекурсия, 140
типы узлов DOM, 140
системные ресу_,, 149
Системы веб-публикации
выбор, 275
поддержка анализатора, 276
список продуктов, 276
устойчивость, 276
поддержка DOM, 277
поддержка SAX, 277
установка, 278
Системы публикации, 22
Службы сообщений, 392
XML, ответ сервера, 400
названия методов, 395
События
добавление/удаление, 338, 339
получение списка, 339
сортировка, 340
события
сортировка, 344
Спецификации XML, соответствие
анализаторы XML, 24
Списки рассылки
анализатор Xerces, 61
списки рассылок
Cocoon, 318
Ссылки на сущности
JDOM, 225
статическое создание наполнения, 275
Строки, канонизация, 105
Строковая константа, свойство SAX, 531
Сущности, 43
сущности
пропускаемы (. _, 84
Схема XML, 44
Castor, классы, порождение, 454

Т

Таблицы логики, 310
Таблицы стилей
CSS, 274
WML, 295
XSP, 301
Теги
вложенные, 32
закрытые, 32
объединение открывающего и закрывающего в элементе, 34
открытые, 32
текстовые данные, 80
Типы данных, определяемые пользователем (SOAP), 359

У

Удаленный вызов процедур (RPC), 320

Узел DOM, свойство SAX, 531
Узлы, определение типа, 140
Узлы, сравнение, DOM Level 3, 184
Указатель позиции в документе, 71
Упаковка, 450
Уровни DOM, 130
Уровни абстракции, 249

Ф

Фабрики, JDOM, 228
    создание, 229
Файлы настройки
    XML-RPC, сервер, 344
Файлы свойств, 195
Файлы свойств, Java
    XML, преобразование в, 196
Фильтры, 113, 118
фирменные свойства и
    возможности, 106
Форматирующие объекты, 50

Ц

Цели, 31
Циклы, 56

Ш

Шаблоны, 54

Э

Элементы, 32
    JDOM, создание, 198

закрытие открытых, 32
объединение открывающих и закрывающих тегов, 34
префиксы и пространства имен, 165
пустые, 33, 43
регистр символов, чувствительность, 32
соглашения об именах, 32
элементы
дочерний и родительский, 81
методы обратного вызова SAX, 77
преимущества перед атрибутами, 35
содержащиеся данные, 80
Элементы, корневые, 31

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-018-9, название «Java и XML» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.