415613

bhv®

Алексей Старыгин

XML

Разработка Web-приложений

+ CD-ROM

- Полное описание языка XML
- Взаимодействие с XSLT и XPath
- Интеграция с HTML, CSS и Java
- Тонкости клиент-серверных решений
- Практические примеры Web-разработок

МАСТЕР ПРОГРАММ

Содержание

Предисловие **13**
Введение **15**

Часть I. XML от А до Я 41

Глава 1. Язык XML **43**

1.1. Что такое документ? 43
1.2. Структура XML-документа 45
1.3. Содержимое XML-документа 47
1.4. Заголовок 48
1.5. Инструкции обработки 49
1.6. Элемент 50
1.7. Имена тегов 55
1.8. Атрибут 57
1.9. Секция *CDATA* 59
1.10. Комментарий 60
1.11. Сущность 61
1.11.1. Символьные сущности 63
1.11.2. Сущности со смешанным контентом 65
1.11.3. Неанализируемые сущности 65
1.11.4. Внутренние и внешние сущности 66
1.12. Пролог документа и DTD 67
1.13. Особенности Cocoon 70

Глава 2. Язык XSLT **72**

2.1. Схема работы XSLT 72
2.2. Пространство имен 76

Функция *string substring-before(string, string)* ... 186
Функция *string substring-after(string, string)* ... 187
Функция *string substring(string, number, number)* ... 187
Функция *number string-length(string)* ... 188
Функция *string normalize-space(string)* ... 189
Функция *string translate(string, string, string)* ... 190
3.5.3. Функции обработки булевых значений ... 191
Функция *boolean boolean(object)* ... 191
Функция *boolean not(boolean)* ... 192
Функция *boolean true()* ... 192
Функция *boolean false()* ... 193
Функция *boolean lang(string)* ... 193
3.5.4. Функции обработки чисел ... 194
Функция *number number(object)* ... 194
Функция *number sum(node-set)* ... 195
Функция *number floor(number)* ... 196
Функция *number ceiling(number)* ... 196
Функция *number round(number)* ... 197
3.6. Дополнительные функции ... 197
3.6.1. Внешние наборы узлов ... 198
3.6.2. Ключи ... 200
3.6.3. Форматирование чисел ... 205
3.6.4. Различные дополнительные функции ... 207
Функция *node-set current()* ... 207
Функция *string unparsed-entity-uri(string)* ... 208
Функция *string generate-id(node-set)* ... 208
Функция *object system-property(string)* ... 209
3.7. Модель данных ... 209

Глава 4. Комплексный пример ... 212

4.1. Главная страница ... 216
4.2. Генерирование списка исполнителей ... 226
4.3. Поиск по названию композиции ... 235
4.4. Поиск по году записи композиции ... 244
4.5. Поиск по жанру ... 252
4.6. Многокритериальный поиск ... 262

Глава 5. Необходимые дополнения ... 280

5.1. Упрощенный синтаксис для элемента стиля ... 280
5.2. Встроенные стили ... 282
5.3. Нерассмотренные механизмы языка XML ... 283
5.3.1. DTD ... 283
5.3.2. Несколько слов о пространствах имен ... 284
5.3.3. Расширения и откат ... 285

ЧАСТЬ П. ПРАКТИЧЕСКАЯ РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ ... 289

Глава 6. Средства создания Web-приложения ... 291

6.1. Документы и технология клиент-сервер ... 292
6.2. XML как модель данных ... 298
6.3. Многослойная архитектура разработки XML-документов ... 309
6.4. Логический слой: Java ... 313
6.4.1. Java: основные возможности ... 315
6.4.2. Пакет разработчика ... 317
Апплеты ... 317
Графика ... 318
Операции ввода/вывода ... 320
Интернационализация ... 321
Сетевые возможности ... 323
Безопасность ... 324
Поддержка архивов ... 327
Взаимодействие с базами данных ... 328
Поддержка служб каталогов ... 330
6.4.3. Java и Internet: дополнительные возможности ... 332
Сервлеты ... 332
Java Server Pages (JSP) ... 335
Дополнительное ПО ... 336
6.5. Распределение задач между сервером и клиентом ... 339
6.6. Инструментарий публикаций Cocoon ... 342

Глава 7. Каскадные стилевые таблицы ... 352

7.1. Что такое CSS? ... 352
7.2. Для чего используются CSS? ... 353
7.3. Основы синтаксиса CSS ... 354
7.4. Привязка таблиц CSS к документу ... 355
7.5. Синтаксис селекторов ... 357
7.5.1. Отбор по атрибутам ... 358
7.5.2. Отбор по родственным связям ... 359
7.5.3. Отбор с использованием псевдоклассов ... 361
7.5.4. Отбор с использованием псевдоэлементов ... 363
7.6. Разрешение конфликтов между правилами ... 366
7.7. Наследование свойств ... 367
7.8. Комментарии ... 368
7.9. Типы устройств ... 368
7.10. Поддерживаемые типы значений ... 369
7.10.1. Числа ... 369
7.10.2. Единицы длины ... 369
7.10.3. Проценты ... 370
7.10.4. URL + URN = URI ... 370

7.10.5. Счетчики ... 371
7.10.6. Цвета ... 371
7.10.7. Строки ... 372
7.11. CSS и размещение объектов на Web-странице ... 372
7.11.1. Блочные, внутритекстовые и невидимые элементы ... 373
7.11.2. Box-модель и таблицы ... 376
7.11.3. Позиционирование областей ... 379
Нормальная (normal) схема позиционирования ... 380
Схема относительного (relative) позиционирования ... 380
Схема абсолютного (absolute) позиционирования ... 383
Схема фиксированного (fixed) позиционирования ... 384
7.11.4. Схема размещения плавающих (float) областей ... 385
7.11.5. Третье измерение ... 387
7.12. Поддержка CSS 1 и 2 в MSIE 5.5 ... 389
7.12.1. Определение стиля для элементов ... 390
7.12.2. Динамические стили ... 390
7.13. Совместное использование CSS и XML ... 394
7.14. Практический пример: создание кнопки ... 396

Глава 8. Методика обработки данных ... 399

8.1. Проектирование HTML-формы ... 399
8.1.1. Элемент *form* ... 399
8.1.2. Элемент *input* ... 401
8.1.3. Элемент *textarea* ... 404
8.1.4. Элемент *select* ... 405
8.1.5. Законченная форма ... 406
8.2. Сервлеты и обработка HTML-формы ... 410
8.3. Обработка параметров: метод *GET* ... 416
8.3.1. Обработка русскоязычных параметров ... 420
8.4. Обработка параметров: метод *POST* ... 422
8.5. Обработка данных на стороне клиента ... 430
8.6. Операции над файловой системой на стороне клиента ... 434
8.6.1. Операции над дисками ... 435
8.6.2. Операции над папками ... 437
8.6.3. Операции над файлами ... 438
8.6.4. Манипулирование путями и именами файлов ... 442
8.6.5. Коллекции дисков, папок и файлов ... 443

Глава 9. Представление данных ... 445

9.1. Графический пользовательский интерфейс ... 445
9.2. Архитектура GUI в терминах технологий XML и HTML ... 449
9.3. Многоуровневая разработка компонентов GUI ... 451
9.4. События ... 451
9.4.1. *onabort* ... 453
9.4.2. *onbeforecopy* ... 453
9.4.3. *onbeforecut* ... 453
9.4.4. *onbeforeeditfocus* ... 453
9.4.5. *onbeforepaste* ... 453
9.4.6. *onbeforeunload* ... 454
9.4.7. *onblur* ... 454
9.4.8. *onchange* ... 454
9.4.9. *onclick* ... 454
9.4.10. *oncontextmenu* ... 455
9.4.11. *oncopy* ... 455
9.4.12. *oncut* ... 455
9.4.13. *ondblclick* ... 455
9.4.14. *ondrag* ... 456
9.4.15. *ondragend* ... 456
9.4.16. *ondragenter* ... 456
9.4.17. *ondragleave* ... 457
9.4.18. *ondragover* ... 457
9.4.19. *ondragstart* ... 457
9.4.20. *ondrop* ... 457
9.4.21. *onerror* ... 458
9.4.22. *onfocus* ... 458
9.4.23. *onhelp* ... 458
9.4.24. *onkeydown* ... 458
9.4.25. *onkeypress* ... 459
9.4.26. *onkeyup* ... 459
9.4.27. *onload* ... 459
9.4.28. *onlosecapture* ... 459
9.4.29. *onmousedown* ... 460
9.4.30. *onmouseenter* ... 460
9.4.31. *onmouseleave* ... 460
9.4.32. *onmousemove* ... 461
9.4.33. *onmouseout* ... 461
9.4.34. *onmouseover* ... 461
9.4.35. *onmouseup* ... 462
9.4.36. *onpaste* ... 462
9.4.37. *onpropertychange* ... 462
9.4.38. *onreadystatechange* ... 462
9.4.39. *onreset* ... 463
9.4.40. *onresize* ... 463
9.4.41. *onresizeend* ... 463
9.4.42. *onresizestart* ... 463
9.4.43. *onscroll* ... 464
9.4.44. *onselect* ... 464
9.4.45. *onselectionchange* ... 464
9.4.46. *onselectstart* ... 464

9.4.47. *onsubmit* ... 464
9.4.48. *onunload* ... 465
9.5. Обработка структуры HTML-документа ... 466
9.5.1. Абсолютная адресация узлов ... 466
9.5.2. Относительная адресация узлов ... 469
9.5.3. Идентификация и характеристики узла ... 470
9.5.4. Поиск узла ... 471
9.5.5. Проверка на наличие контента и атрибутов ... 471
9.5.6. Создание, удаление и модификация узлов ... 472
Создание и копирование узлов ... 473
Встраивание узлов в дерево документа ... 474
Удаление и замена узлов из дерева документа ... 475
9.5.7. Обработка контента элемента ... 476
9.5.8. Обработка таблиц ... 478
9.5.9. Обработка URL ... 479
9.5.10. Обработка таблиц стилей ... 480
9.6. Законченный пример ... 481
9.6.1. Постановка задачи ... 482
9.6.2. Реализация приложения ... 484
Файл example.xml ... 485
Файл form.xml ... 486
Файл form.xsl ... 488
Файл form_handler.xml ... 495
Файл windows.xml ... 499
Файл windows.xsl ... 501
Файл example.js ... 502
9.6.3. Создание окна ... 503
9.6.4. Удаление окна ... 504
9.6.5. Перемещение окна ... 505
9.6.6. Переход к соседнему окну ... 507
9.6.7. Сохранение данных ... 508
9.6.8. Загрузка данных ... 511
9.7. Заключение ... 517

ЧАСТЬ III. ПРИЛОЖЕНИЯ ... 519

Приложение 1. Некоторые свойства объекта *style* (о поддержке CSS-браузером MSIE 5.5) ... **521**

Приложение 2. Описание демонстрационных Web-приложений ... **537**
П2.1. Элемент управления Selector ... 537
П2.2. Элемент управления Chooser ... 540
П2.3. Меню ... 543
П2.4. Дерево ... 546
П2.5. Web-приложение, реализующее обработку документов на стороне клиента ... 552

П2.6. Web-приложение, демонстрирующее совместную работу пяти предыдущих компонентов ... 556
П2.6.1. Файлы, содержащиеся в папке *complex* ... 559
П2.6.2. Файлы, содержащиеся в папке *form1* ... 565
П2.6.3. Файлы, содержащиеся в папке *form2* ... 569
П2.6.4. Файлы, содержащиеся в папке *form3* ... 571

Приложение 3. Установка программного обеспечения и файлов примеров ... **573**
П3.1. Установка Tomcat ... 573
П3.1.1. Tomcat как автономное приложение ... 573
П3.1.2. Tomcat+Apache ... 574
П3.2. Установка Cocoon ... 575
П3.3. Конфигурирование примеров ... 575
П3.4. Запуск серверных приложений ... 577
П3.5. Установка Cocoon2 ... 578
П3.5.1. Особенности установки Cocoon2 под Linux ... 578

Приложение 4. Описание компакт-диска ... **580**

Предметный указатель ... **582**

Предисловие

В разного рода публикациях часто можно встретить фразу: "В эпоху бурного развития Internet...", в окончании которой расписывается, насколько "прочно Internet вошел в нашу жизнь". Но если задаться вопросом, в какой степени Internet заменяет те средства, которые использовались до его появления, то можно сказать, что его развитие носит скорее количественный, нежели качественный характер.

Нет сомнений в том, что количество сайтов самой разнообразной тематики в последнее время растет чуть ли не экспоненциально. Гораздо медленнее, но тем не менее, оперативно расширяется спектр самых разных Internet-технологий. Все это бесспорно хорошо, но почему-то многие Web-мастера не спешат применять эти технологии в своих разработках. Разумеется, многие из этих технологий достаточно трудно использовать по целому ряду причин: дороговизна, сложность, малая популярность у пользователей и т. п. К этому списку можно добавить, пожалуй, еще и частое несоответствие "раскрученности" какого-либо продукта его актуальности.

Все это так. Но ведь есть и другие средства, не обладающие вышеперечисленными недостатками, для того, чтобы сделать Web-сайт красивее, гибче в навигации и функциональнее. Существующий на сегодняшний день инструментарий способен не только обеспечить привлекательность какого-либо ресурса Web для пользователей, но и на порядок упростить его разработку. На наш взгляд, основной причиной медленного внедрения передовых Internet-технологий является отсутствие у разработчика Web-проектов исчерпывающей информации о том, для чего предназначена некоторая конкретная Internet-технология, а также внятной документации о ее использовании.

Для того чтобы хотя бы частично заполнить этот пробел, и написана эта книга. И хотя основной ее темой является язык XML, в ней затрагиваются также некоторые другие средства разработки Web-приложений. В самом деле, ведь появление XML не означает отказа от таких инструментов, как

HTML, CSS, JavaScript и т. п. Более того, в этой книге мы покажем, что применение XML только вместе с ними способно приносить максимальный эффект. С другой стороны, использование "чистого" XML хотя и возможно, но настолько неэффективно, что может быть ограничено очень узким кругом задач, которые вряд ли будут интересны рядовому пользователю.

Аббревиатура XML вынесена в заголовок книги, и это говорит только о том, что данный язык мы опишем если не полностью, то, по крайней мере, очень детально. Но на самом деле ему посвящается только первая часть книги, а именно: в *главе 1* представлено описание непосредственно языка XML, в *главе 2* — языка XSLT трансформации XML-документов и в *главе 3* — языка XPath адресации узлов дерева XML-документа. Кроме этого, первая часть содержит подробно разобранный комплексный пример, содержащийся в *главе 4,* и описание некоторых специфических особенностей языков, представленных в первых трех главах книги. Этим особенностям посвящена последняя — пятая — глава первой части.

Если первую часть книги можно назвать теоретической, то вторая полностью посвящена практическим разработкам Web-приложения. В ней рассматривается многослойная архитектура Web-приложения, а также делается попытка наиболее широко раскрыть все возможности по реализации каждого его слоя. Здесь также описывается инструментарий, использующийся для разработок вместе с демонстрацией различных аспектов его применения.

В частности, *глава 6* помимо многослойной архитектуры Web-приложения посвящена инструментарию публикаций Cocoon 1.8.2, который мы использовали для своих разработок. В *главе 7* описываются основы оформления конечного документа (как в формате XML, так и HTML) с помощью каскадных стилевых таблиц. *Глава 8* содержит информацию об основном средстве обмена данными между клиентом и сервером — HTML-форме. Причем в этой главе мы рассказываем не только о том, как спроектировать эту форму, но и о том, как обработать ее параметры, переданные с помощью двух методов отправки HTTP-запроса GET и POST.

И, наконец, *глава 9* посвящена главным образом обработке HTML-документа. В ней содержится информация о том, как можно изменить его структуру с помощью JavaScript, а также комплексный пример, предназначенный для демонстрации возможностей JavaScript, используемых для представления данных исходного документа. Хотя, разумеется, это только основная задача примера.

Для того чтобы книга не выглядела голословной, вместе с ней поставляется компакт-диск, на котором находятся шесть Web-приложений, иллюстрирующих материал книги, а также все программное обеспечение, требуемое для его запуска. Все необходимые инструкции по его установке, а также по установке примеров приводятся в файле Readme на компакт-диске и в конце книги, в *приложении 3.* Сами же Web-приложения кратко описаны в *приложении 2.*

Приятного чтения.

Введение

Для кого эта книга

Эта книга предназначена, прежде всего, для опытных Web-разработчиков, имеющих навыки работы с HTML, Java и JavaScript. Технология XML (Extensible Markup Language — расширяемый язык разметки) принадлежит к числу технологий типа "клиент-сервер", поэтому, если вы всерьез решили заняться разработкой Web-приложений, то вам необходимо иметь представление о том, какими средствами создается как клиентская, так и серверная часть этого приложения.

Для клиентской части таким средством является HTML, или, точнее, Dynamic HTML (DHTML), поскольку на данный момент не существует такого браузера, который мог бы удовлетворительно интерпретировать XML-документ.

Для серверной части наиболее подходящим представляется язык Java как кроссплатформенный и в последнее время бурно развивающийся язык программирования. На платформе этого языка реализованы такие технологии, как сервлеты, поддержка сервиса имен и каталога, инструменты для работы с HTTP-соединением и многое другое. Эти и другие темы, в общем-то, мало касающиеся языка XML, будут затронуты в различных главах данной книги. Но исчерпывающего изложения подобных вопросов, выходящих за рамки основного содержания книги, читатель здесь не найдет. Предполагается, что этот материал уже освоен или будет изучен с помощью соответствующей литературы.

Если вы обладаете всеми вышеперечисленными навыками, но не уверены, стоит ли изучать еще и XML: "вдруг XML — это просто очередная новомодная технология?", тогда давайте вместе подробно проанализируем текущее положение дел с Internet-технологиями и выясним, какое место занимает XML в мире Web.

Общие характеристики Web-приложения

Что же такое Web-приложение? Прежде всего, это нечто, гораздо большее, чем то, что мы понимаем под Web-сайтом или Web-страницей. Обычный сайт состоит из одного или нескольких гипертекстовых документов, с которыми пользователь взаимодействует в одностороннем или почти одностороннем порядке, т. е. в режиме монолога. На взгляд автора, заполнение полей регистрационной формы или бланка заказа трудно назвать полноценным диалогом с сервером. А ведь именно такие средства общения чаще всего встречаются в Internet. Web-приложение в этом смысле отличается гораздо большей интерактивностью.

Работая с Web-приложением, пользователь имеет более широкие возможности по обработке информации, хранящейся на Web-сервере. А именно, ему доступны такие операции, как добавление, удаление, редактирование, переупорядочивание, поиск и др. с разнородными данными, организованными различными способами и, может быть, различными пользователями Web-приложения.

Кроме того, характерной особенностью Web-приложения является область его применения. Если Web-сайт служит, в основном, для широкого распространения какой-либо информации через Internet, то Web-приложение подходит скорее для Intranet, т. е. для внутрикорпоративного использования, и круг его пользователей, чаще всего, конкретно определен и ограничен.

В самых общих чертах структуру такого приложения можно определить так: существует единое информационное хранилище, состоящее из данных различного формата, находящихся на сервере, и множество пользователей, обрабатывающих эти данные, т. е. клиентов, взаимодействующих с хранилищем через Web-интерфейс или браузер. Здесь следует сразу отметить, что слово "хранилище" не то же самое, что и база данных. Оно включает в себя документы, которые жестко организованы и имеют четкую внутреннюю структуру, таким документом может быть, в том числе, и таблица базы данных.

Но тут сразу возникает вопрос: если информационное хранилище не база данных, а точнее, не только база данных, то что еще оно из себя представляет? Сначала давайте взглянем на информационное хранилище глазами пользователя. Какие услуги или сервисы оно может предоставить? Рассмотрим пример.

Предположим, что наш пользователь — служащий отдела кадров, причем не в мелкой фирме, а в серьезной компании, имеющей собственную кадровую политику. В обязанности этого служащего входит регистрация и прием резюме желающего устроиться на работу в эту компанию. Остальные функции служащего нас пока интересовать не будут.

Итак, для того, чтобы оформить устраивающегося на работу, служащий должен сформировать два документа: подробнейшую регистрационную карточку с множеством полей и резюме, представляющее собой форматированный текст. После того, как эти документы оформлены и их электронные версии помещены на сервер, наступает этап их использования другими пользователями.

С регистрационной карточкой все более или менее понятно, т. к. вся содержащаяся в ней информация формализована и различные отделы компании, например, бухгалтерия, могут получить к ней полный или частичный доступ как к полям базы данных. Эта база данных может состоять из одной или нескольких таблиц с различными уровнями доступа. Отметим здесь только один момент: с регистрационной карточкой могут работать служащие разных отделов, и совсем не обязательно для каждого из них иметь перед глазами всю информацию, содержащуюся в карточке. Таким образом, регистрационная карточка и ее части представляют собой так называемые разделяемые ресурсы. Конечно, здесь имеются в виду документы, а не аппаратные ресурсы.

С резюме дело обстоит немного сложнее. Содержащаяся в нем информация не может быть помещена (кроме как целиком) в реляционную или какую-либо другую базу данных. Но, тем не менее, эта информация также должна быть анализируемой и разделяемой. Разрешением этой проблемы мы займемся несколько позже. А сейчас самое время определить общие требования, которым должно удовлетворять приложение, позволяющее эффективно работать с описанными документами.

- Очевидно, что такое приложение должно быть сетевым и многопользовательским, причем права пользователей должны быть определены для каждого документа и, может быть, для его частей.
- Приложение должно быть защищено как от несанкционированного доступа, так и от сбоев системы.
- Должен быть разработан эффективный способ хранения разнородных документов. То есть должны быть обеспечены следующие возможности: одновременный доступ к ресурсу, компактное хранение данных, быстрое выполнение всех пользовательских операций над документами.
- Приложение должно обладать удобным и, желательно, приятным для пользователя интерфейсом.

Совершенно очевидно, что с этим комплексом задач справится только Web-приложение (подробнее об этом читайте в *части II).* Хотя, разумеется, в любом случае можно изобрести свой велосипед.

Теперь, сказав воображаемому пользователю "до свидания" и получив общее представление о том, что такое Web-приложение, вспомним название книги и поговорим о Web-интерфейсе.

Программное обеспечение с Web-интерфейсом

Прогуливаясь по Сети, читатель наверняка обратил внимание на многообразие стилей оформления Web-сайтов. Если нет, то, обратившись за помощью к постоянным обитателям виртуального мира, он получит массу впечатлений от огромного количества картинок, звуков, анимации и даже видеоизображений. И, в то же время, насколько разнообразен и пестр дизайн страничек Internet, настолько же сер и непривлекателен интерфейс программ, основанных на стандартном API (Application Program Interface — программно-прикладной интерфейс) какой-либо операционной системы. Достаточно просто сравнить, например, пользовательское меню Internet Explorer с тем меню, которое находится на одной из Web-страниц официального сайта его разработчика.

В последнее время все чаще появляются различные сетевые сервисы, снабженные Web-интерфейсом. Это такие службы, как DNS, почтовый сервис, средства маршрутизации и т. п. Некоторые компании, производящие серверы с предустановленной операционной системой Linux, снабжают свои машины Web-управляемым интерфейсом, позволяющим настраивать многие параметры системы. Можно сказать, что там, где требуются многопользовательские, распределенные, удобные для пользователей системы, уверенно занимает свое место Web-приложение. Если же вспомнить о таких электронных предприятиях, как электронная биржа или интернет-магазин, то красота и эстетичность оформления их внешнего вида становится первостепенным вопросом.

Говоря о различии Web-сайта и Web-приложения, мы делаем упор на интерактивность, а значит и на функциональность последнего. Можно ли создать такой интерфейс, чтобы он одновременно был и функционален, и красиво оформлен? Да. Другими словами, если удастся создать основные элементы управления (такие, как меню, дерево, список...) средствами HTML, то можно считать задачу построения интерфейса выполненной. Ведь после создания элементов управления, которые обеспечивают функциональность, можно для красоты оформления к каждому пункту меню или вершине дерева применить все возможности языка HTML.

Понятие "Web-интерфейс" слишком объемно и может быть разбито на несколько разных составляющих. Но перед тем как о них говорить, необходимо представлять, как устроен просто интерфейс, без приставки Web.

Из чего состоит и как строится интерфейс

Принципы организации и работы пользовательского интерфейса любого приложения более-менее схожи, вне зависимости от того, для какой операционной системы оно было написано. Если, конечно, не рассматривать какие-нибудь экстравагантные или устаревшие приложения и операционные системы.

Все составляющие интерфейса можно условно разделить на видимые и невидимые. Видимые компоненты интерфейса представляют собой те части приложения, которые могут быть отображены на экране дисплея. Это пункты меню, кнопки панели управления, курсор мыши, заголовок и строка состояния окна и т. д. К невидимым компонентам относятся средства взаимодействия видимых компонентов друг с другом и с пользователем. Составляющими невидимой части интерфейса могут быть, например, события от мыши или клавиатуры, механизм обработки этих событий, способ изменения внешнего вида приложения в ответ на событие и т. д.

При построении интерфейса мы будем пользоваться понятием "элемент управления", которое представляет собой совокупность некоторых видимых и невидимых конструкций. Если рассмотреть в качестве примера выпадающее меню, то его видимой частью будут изображения пунктов и подпунктов этого меню, а невидимой — реакция на события от мыши, т. е. перерисовка внешнего вида меню и выполнение команды, сопоставленной с активизированным пунктом этого меню. Для того чтобы не перегружать книгу сведениями, имеющими отдаленное отношение к XML, мы сосредоточим свое внимание, главным образом, на визуальной части интерфейса. Невидимые компоненты также будут приведены и снабжены комментариями, но ровно настолько, насколько это необходимо для работоспособности примеров, в основе которых лежит XML-код.

При создании как видимой, так и невидимой частей интерфейса обычно используются соответствующие средства той операционной системы, для которой создается приложение. В программно-прикладной интерфейс ОС, как правило, входят такие стандартные элементы управления, как окна, меню, списки, кнопки, деревья и т. д. Помимо изображений этих элементов, API содержит также и средства управления ими — невидимую часть интерфейса — механизм диспетчеризации сообщений, функции перерисовки графических объектов и т. п.

Если программист, пользующийся инструментарием данного API, попробует создать собственные элементы управления, то в качестве "строительных блоков" он может применить стандартные средства API. К сожалению, в случае Web-приложения это невозможно, т. к. не существует единой операционной системы для среды Web. Можно, конечно, воспользоваться таким средством, как пакет Java-классов Swing или каким-либо другим, но на данный момент все подобные Internet-технологии еще далеки от совершенства и не имеют широкого распространения.

Задача создания универсального инструментария для построения интерфейса выходит за рамки данной книги. Но в ней будут продемонстрированы

основные приемы использования XML для решения более узкой задачи — построения Web-интерфейса. Хотя это и менее широкая постановка вопроса, но за основу мы все же возьмем самые общие принципы организации интерфейса, описанные ранее.

Web-интерфейс и динамический HTML

Со времен появления Internet Web-разработчиками были написаны миллионы байт HTML-кода. Для создания Web-сайтов и Web-приложений использовались и продолжают использоваться три основные средства разработки. Это непосредственно теги HTML, какой-либо интерпретируемый (скриптовый) язык (JScript, JavaScript, VBScript и т. п., везде далее - JavaScript, который мы будем использовать в рабочих примерах, а исключения оговаривать отдельно) и каскадные стилевые таблицы (CSS). Помимо этих основных средств разработки в содержимое Web-приложения могут входить изображения различного формата, анимация и т. д., которые, в общем, компонентами HTML не являются.

Давайте рассмотрим Web-интерфейс с точки зрения HTML или, точнее, DHTML. Если рассмотреть код, из которого состоит не самым примитивным образом оформленная Web-страница, можно увидеть, что теги HTML отвечают за то, какие элементы должны присутствовать на странице, CSS — за отображение этих элементов и, наконец, код JavaScript — за все изменения, которые могут происходить с этими элементами. Таким образом, если говорить о создании Web-интерфейса, то можно сказать, что его видимую составляющую представляют теги HTML и конструкции CSS, а невидимую — операторы языка JavaScript.

Такая система разработки Web-приложения обладает некоторыми недостатками. Давайте вернемся к примеру со служащим отдела кадров и попытаемся представить себе Web-интерфейс приложения, которое позволяет ему обрабатывать регистрационные карточки и резюме устраивающихся на работу. Web-приложение, позволяющее только лишь обработать поля форм и редактировать текст, само по себе не будет слишком сложной программой или, что точнее сказать, страницей.

Но если принять во внимание то, что служащему необходимо создавать новые документы (например, заводить карточки), удалять старые, просматривать список уже имеющихся документов, то сразу видно, что функциональность такого Web-приложения становится гораздо выше. Оно уже должно содержать меню, каждый пункт которого сопоставлен с командами удаления, добавления и т. д. документов; отдельное окно со списком документов, а на практике, чаще всего, с деревом, каждая вершина которого представляет собой либо документ, либо папку с документами.

Представьте себе коллектив разработчиков, который пытается создать Web-интерфейс для этого приложения или для любой другой, столь же комплексной задачи. При этом они используют HTML, CSS и JavaScript, под совокупностью которых мы и будем подразумевать DHTML.

Во-первых, разработчики-программисты в связи с большим объемом кода JavaScript будут вынуждены написать некоторое количество вспомогательных функций, которые, скорее всего, будут содержать часто повторяющийся код. Затем они должны будут включить этот код (непосредственно или путем подключения дополнительного файла) в каждую страницу приложения, которая содержит хотя бы один элемент управления. Очевидно, что любая из таких страниц не станет от этого загружаться быстрее. А если на странице находится форма, гиперссылка или что-либо еще, заставляющее эту страницу перерисовываться при активировании этого объекта пользователем?

Поэтому можно сделать вывод, что малая скорость загрузки и работы Web-приложения, созданного исключительно средствами DHMTL, и большой его объем являются главными недостатками. То же самое касается и конструкций CSS или форматирующих тегов (это зависит от предпочтений разработчиков), описывающих стили оформления наиболее часто встречающихся элементов, например, пунктов меню.

Во-вторых, вспомним, что разработчиками являются не только программисты, но и дизайнеры, люди, имеющие к программированию несколько отдаленное отношение. Им для того, чтобы встроить в интерфейс, допустим, статичное или анимированное изображение, нужно обращаться к Web-мастерам и программистам, у которых, между нами говоря, свои проблемы. Если дело касается того, чтобы просто вставить рисунок, то это — еще полбеды. Представляете выражение лица программиста, которого в двадцать четвертый раз просят сдвинуть рисунок, скажем, на два миллиметра вправо, да еще непонятно с какой целью? Отсюда следует, что при таких средствах разработки у коллектива полностью отсутствует разделение труда, т. е. каждый из них должен помимо своей профессии иметь представление о многих вещах, с ней не связанных.

В-третьих, отметим, что существующие на сегодняшний день средства отладки DHTML весьма бедны. А средства автоматического или визуального построения элементов управления отсутствуют полностью. Пусть даже у разработчиков не возникает головной боли по поводу управления памятью или приведения типов, но что если в громоздком HTML-документе нужно найти причину не совсем корректного отображения какой-то его части? Таким образом, третьим недостатком DHTML является его поначалу кажущаяся достоинством вольность интерпретации. Ведь ни один HTML-Документ сам по себе не откажется загружаться и не выдаст сообщение об ошибке, если, например, какой-либо его открывающий тег (блочный или встроенный — не важно) не имеет закрывающего.

Преимущества XML

Как вы уже, наверное, догадываетесь, язык XML в полной мере компенсирует все эти недостатки. Так как пример со служащим отдела кадров не является всеохватывающим, на самом деле таких недостатков гораздо больше. Если же ограничиться рамками этого примера, то, задействовав XML, получим следующие решения вышеперечисленных проблем.

Обычно XML-документ содержит не только какую-либо структурированную информацию, но и правила ее трансформации, т. е. представляет собой, как минимум, два файла. Один из этих файлов содержит данные, структурированные с помощью тегов, названия которых придуманы создателем документа. Второй — правила интерпретации этих тегов. Первый файл обычно имеет расширение xml, а второй — xsl. Такое свойство XML-документов называется разделением данных и их представления.

Программист-разработчик может извлечь из этой ситуации несколько полезных усовершенствований для своего способа программирования. Во-первых, если возникает необходимость в частом использовании какой-либо громоздкой HTML-конструкции (например, элемента текста, или рисунка, или и того и другого вместе с множеством правил форматирования), то при использовании XML достаточно создать один тег или группу тегов, единственный раз задать правила их форматирования и далее вместо всей конструкции использовать только эти созданные теги. Для разработчика такая модульность кода продукта может оказаться немаловажной.

Рассмотрим следующий код:

```
<card number="111">
 <surname> Иванов </surname>
 <name> Сергей </name>
 <telephone> 01-01-01 </telephone>
...
</card>
```

Согласитесь, что оперировать тегами с такими именами как card, name и т. д. гораздо проще и понятнее, чем тегами языка HTML. И, кроме того, каждый из этих тегов может представлять собой набор правил форматирования любой степени сложности. Это не означает, что нужно совсем отказаться от HTML, ведь, говоря о правилах форматирования, мы подразумеваем использование именно этого языка. Но, с другой стороны, если работать только на HTML, то результатом будет замена каждого тега из вышеприведенного примера на его HTML-интерпретацию. Таким образом, объем кода для каждой карточки резко возрастет, а это, согласитесь, не очень удобно.

Во-вторых, для дизайнеров такое положение вещей также приносит некоторую, хотя и весьма небольшую, выгоду. Мы говорили, что разработчики приложений или сайтов на языке HTML, программисты или дизайнеры должны владеть не только своей профессией, но и смежными специальностями. Откровенно говоря, XML не решает эту проблему и не собирается этого делать в будущем. По крайней мере, от разработчиков стандартов данного языка не поступает никаких внятных предложений по этому поводу.

Единственное, хотя и очень небольшое преимущество XML состоит в том, что XML-документ состоит не из одного файла, а из нескольких, и предполагается, что можно организовать работу программистов и дизайнеров так, чтобы каждый работал над своей группой файлов. Но на практике их деятельность неизбежно пересекается, и поэтому определенно можно сказать только то, что XML решает проблему разделения труда лучше, чем HTML.

Что касается средств отладки XML-кода, то, как видно из примера, каждая единица информации заключена в логические скобки — открывающий и закрывающий теги. В языке XML закрывающий тег необходим всегда, как и кавычки, в которые заключены значения атрибутов. XML изначально задумывался как язык разметки, не содержащий двусмысленностей интерпретации, и это отражено в соответствующих стандартах и спецификациях. Такая жесткость правил синтаксиса языка позволяет значительно сократить количество ошибок в документе, а также отыскать ту часть кода, которая отвечает за несоответствующим образом отображающуюся часть документа. При несоблюдении правил документ просто не будет отображаться в окне браузера, вместо этого появится сообщение об ошибке.

Здесь мы рассмотрели только некоторые недостатки HTML и то, как XML справляется с ними. Но существует ряд задач, для которых HTML абсолютно не предназначен, и это нельзя назвать его недостатком. Основной функцией этого языка является только лишь отображение информации, в то время как язык XML охватывает ее смысл и структуру. В этом и состоит революционность XML и суть этой технологии, но подробнее об этом несколько позже.

Сопоставление XML и HTML

Как нам удалось показать, XML не может выступать отдельно от HTML в том случае, когда информация, содержащаяся в XML-документе, нуждается в некотором представлении, или, проще говоря, в отображении (точнее говоря, может, но его совместное использование с HTML несколько "уютнее"). Теперь следует обозначить границу между этими двумя языками. Мы уже можем сделать вывод, что язык XML никогда не вытеснит HTML,

но и не оставит его неизменным. Влияние XML на HTML выражается в появлении такого стандарта как XHTML, который представляет собой, грубо говоря, HTML, дополненный жесткими правилами разметки языка XML. И если между XML и HTML не может быть конкурентной борьбы в силу различия их функций, то эта борьба возможна и уже происходит между HTML и XHTML.

Для того чтобы наилучшим образом проиллюстрировать грань между XML и HTML, вспомним два типа текстовых редакторов. Первый тип представляет собой редакторы, работающие по принципу WYSIWYG (What You See Is What You Get — что ты видишь, то ты и получишь). Согласно данному принципу, вид текста при вводе будет идентичен отображению на экране или распечатке на принтере. Примером такого редактора может служить Notepad. Примером реализации второго типа редакторов (когда вводимый и отображаемый документы не совпадают) могут быть редактор TeX или, скажем, SGML+DSSSL. Существование этих двух типов редакторов говорит о том, что способ хранения информации может отличаться от способа ее отображения.

Вспомнив о требованиях к Web-приложению, о которых мы говорили ранее, мы отметили, что информация, находящаяся на Web-сервере, должна храниться в наиболее компактной форме. И в то же время она, если это потребуется, должна выглядеть по-разному для разных пользователей. В примере со служащим отдела кадров мы говорили, что к регистрационной карточке устраивающегося на работу могут иметь доступ пользователи, выполняющие разные функции. Поэтому кому-то из них могут понадобиться паспортные данные, другому — виды деятельности на прежних местах работы и т. д. Таким образом, разные пользователи, обращаясь к регистрационной карточке, могут увидеть разные данные об устраивающемся на работу. А может быть и те же самые, но по-разному отформатированные.

Если мы говорим о текстовом документе, с которым работает только один человек, то для его обработки наиболее подходит редактор, работающий по принципу WYSIWYG. Но если речь заходит о множественном доступе к документу, т. е. о многопользовательском приложении, то информация должна быть либо продублирована соответствующим образом для каждого пользователя (что не удовлетворяет требованию компактности), либо храниться в некоем универсальном виде, отличном от того, в котором отображается.

Если глубже проанализировать различия двух типов текстовых редакторов, то можно заметить, что те из них, которые относятся к типу WYSIWYG, обрабатывают весь текст документа равноправно, т. е. вне зависимости от того, каким смысловым содержанием наделен этот текст (паспортные ли это данные или что-либо еще). Второй тип редакторов, хоть и на примитивном уровне в случае нашего примера, уделяет некоторое внимание структуре данных, содержащихся в документе.

Впрочем, текстовые редакторы — это всего лишь аналогия, но, тем не менее, она позволяет отделить друг от друга HTML и XML. Если первый занимается только представлением информации, то второй — еще и ее хранением в некоторой специфической форме. Таким образом, можно вывести формулу:

XML = структурированная информация + способ(ы) ее представления,

где роль последних отводится языку HTML.

Проект XML: прорисовка формы

Итак, что же такое язык XML и чем он не является?

Несмотря на то, что в названии (расширяемый язык разметки) слово "разметка" присутствует, XML не содержит ни одного правила или набора символов, который можно было бы назвать "разметкой". Вообще говоря, разметка представляет собой описание логической структуры документа и предназначена для определения частей документа и введения над ними отношений порядка. Например, в HTML тег <P> означает, что заключенный в него текст является параграфом, в TeX команда `/frac{...}{...}` — о том, что текст в первых фигурных скобках является числителем, а во вторых — знаменателем дроби. Поэтому не совсем верно называть XML языком разметки. Он представляет собой скорее инструментарий для создания такого языка, который необходим для разметки обрабатываемого класса документов. Примерами таких классов могут быть математические или химические публикации, гипертекст и обычный текст и все, что угодно.

Для великого множества типов документов, будь то газетные статьи, книги, научные публикации или различные формуляры, и был создан язык XML - универсальный способ для описания методов хранения и управления информацией любого вида, единственным ограничением на которую является возможность ее вербализации. Это описание представляет собой, фактически, моделирование некоего исходного, неформализованного текста.

Модель XML-документа строится в зависимости от того, какую смысловую нагрузку несут части текста и как они взаимосвязаны друг с другом. Типы связей при этом могут быть самыми разными. Например, в случае книги, такой связью может быть относительное расположение названий или нумерации частей, глав и содержащегося в них текста. Или это может быть связь по типу гиперссылки, когда документ непосредственно не содержит какую-то свою часть, а только ссылается на нее.

Модель или логическая структура XML-документа служит только одной цели, состоящей в том, чтобы иметь возможность работать с этим документом, как с набором динамических (изменяемых во времени, а не формируе-

мых по требованию) данных, а не просто как со статическим текстом, который можно только просматривать. При наличии жесткой структуры и строгой однозначности синтаксиса, в XML-документе очень просто найти и получить доступ к любой содержащейся в нем единице информации. Работая, например, с таблицей и используя механизм адресации XPath, можно легко получить и обработать содержимое произвольной ее ячейки. В то время как средствами HTML то же самое действие осуществить значительно труднее, да, в общем-то, и не нужно. Язык HTML послужит в данном случае только для того, чтобы эту таблицу отобразить.

Как мы уже говорили, XML-документ состоит из двух частей. Причем вторая часть, не являясь обязательной, служит средством отображения или публикации того, что содержится в первой. Первая же часть и содержит сам XML-код, т. е. текст и его разметку, созданную пользователем. Оперируя такими инструментами, как адресация любого элемента или его атрибута, создание "на лету" любой конструкции XML, а также всеми возможностями языка запросов к XML-документу, мы превращаем его в базу данных с более сложной, но четко определенной структурой.

Здесь под языком запросов мы понимаем возможность чтения и, самое главное, записи содержимого элемента или его атрибута, а также удаления элемента, атрибута или их содержимого, причем все эти операции проводятся над внешним документом. Хотя нужно отметить, что на сегодняшний день не существует удовлетворительного языка запросов, осуществляющего запись и удаление в XML-документе. Большинство из распространяемых сегодня языков стремятся к тому, чтобы производить сложные многокритериальные выборки из одного или нескольких XML-документов, при этом сохраняя простоту и гибкость запроса.

Несмотря на широкий спектр возможностей языка XML, следует понимать, что он не стремится заменить собой ни базы данных, ни существующие средства создания Web-приложений, ни даже средства публикаций. Этот язык не поглощает функций ни одной из этих технологий, но объединяет некоторые из них. И, в то же время, он сам является отдельной технологией хранения и публикации данных, а также служит для организации взаимодействия некоторых других технологий и направлений, как правило, имеющих отношение к Internet. Для того чтобы говорить о роли, которую играет XML в виде одной серверной технологии среди группы других, нужно сначала определить предпосылки его возникновения и проследить путь его эволюционного развития.

W3-консорциум и X-технологии

Рассматривая язык XML в перспективе, мы не станем затрагивать такие моменты, как общественная деятельность различных рабочих групп, географические пути распространения языка и т. п. Мы проанализируем только лишь побудительные причины возникновения XML как некоей платформы, на основе которой развилось множество взаимосвязанных и взаимодополняющих технологий.

Временем рождения XML можно считать 1996 год, в конце которого появился черновой вариант спецификации языка, или 1998, когда эта спецификация была утверждена. Разумеется, для понимания проблем современности необходимо знание истории, но, говоря об истории XML, предоставим каменный век историкам и начнем сразу с 1986 года — года появления языка SGML.

SGML (Standard Generalized Markup Language — Стандартный Обобщенный Язык Разметки) заявил о себе как гибкий, комплексный и всеохватывающий мета-язык для создания языков разметки. И хотя понятие гипертекста появилось в 1945 году, этот язык не имеет гипертекстовой модели. Создание SGML можно с уверенностью назвать попыткой, объять необъятное, т. к. он объединяет в себе такие возможности, которые крайне редко используются все вместе. В этом и состоит его главный недостаток — сложность и, как следствие, дороговизна этого языка ограничивает его использование только крупными компаниями, которые могут позволить себе купить соответствующее программное обеспечение и нанять высокооплачиваемых специалистов. Кроме того, у небольших компаний редко возникают настолько сложные задачи, чтобы привлекать к их решению SGML.

Наиболее широко SGML применяется для создания других языков разметки, именно с его помощью был создан язык разметки гипертекстовых документов — HTML, спецификация которого была утверждена в 1992 году. Его появление было связано с необходимостью организации стремительно увеличивающегося массива документов в сети Internet. Бурный рост количества подключений к Internet и, соответственно, Web-серверов повлек за собой такую потребность в кодировке электронных документов, с которой не мог справиться SGML вследствие высокой трудности освоения. Появление HTML — очень простого языка разметки — быстро решило эту проблему, легкость в изучении и богатство средств оформления документов сделали его самым популярным языком для пользователей Internet.

Но, по мере роста количества и изменения качества документов в Сети, росли и предъявляемые к ним требования, и простота HTML превратилась в его главный недостаток. Ограниченность количества тегов и полное безразличие к структуре документа побудили разработчиков в лице консорциума W3C к созданию такого языка разметки, который был бы не столь сложен, как SGML, и не настолько примитивен, как HTML. В результате, сочетая в себе простоту HTML, логику разметки SGML и удовлетворяя требованиям Internet, появился на свет язык XML. И несмотря на молодость (всего-то 4 года), уже успел превратиться в целый набор технологий,

призванных выполнять самые разные действия по обработке информации как в сети Internet, так и вне ее.

Посетив сайт консорциума W3C **(www.w3.org),** вы можете познакомиться с несколькими десятками этих, так называемых, X-технологий. Здесь мы перечислим и кратко опишем некоторые из них, в том числе и, собственно, XML, которому посвящена книга.

XML

Расширяемый язык разметки (extensible Markup Language, XML) описывает класс объектов XML document, а также частично работу компьютерных программ, обрабатывающих объекты с данными, реализующими этот класс. XML — это прикладной уровень или усеченная форма SGML, Стандартного Обобщенного языка разметки [ISO 8879]. По своему построению, XML-документ является полноценным SGML-документом.

XSL

Расширяемый язык стилевых таблиц (extensible Stylesheet Language, *XSL)* состоит из двух частей: языка для трансформации XML-документов и из XML-словаря, определяющего семантику форматирования. Стилевая таблица XSL определяет представление класса XML-документов, описывая, как представитель класса, трансформируемый в XML-документ, использует словарь форматирования.

XSLT

Язык XSLT (XSL Transformations) разработан как часть XSL. XSL определяет стилизацию XML-документа, задействуя XSLT для объяснения того, как документ трансформируется в другой XML-документ на основе словаря форматирования. Кроме того, XSLT может использоваться независимо от XSL. Однако XSLT не является единым комплексным языком трансформации XML-документов. Он, скорее, предназначен для тех видов трансформаций, которые необходимы, когда XSLT используется как часть XSL.

XPath

Главной задачей XPath является адресация частей XML-документа. Для этой цели XPath представляет XML-документ как некоторое дерево узлов различного типа. Поэтому помимо адресации в XPath обеспечиваются минимальные возможности по обработке данных различных типов. Третьей задачей XPath можно назвать динамическое генерирование контента, если этот контент не может быть создан до первого обращения к документу.

XSL Formatting Objects

XSL FO (Formatting Objects — форматирующие объекты) — это набор классов, в терминах которых отражена семантика форматирования. Они представляют собой узлы дерева, полученного в результате XSL-трансформации. Классы форматирующих объектов определяют такие полиграфические термины, как страница, параграф и т. п. Лучший контроль над представлением этих объектов обеспечивается множеством свойств форматирования. Это такие свойства, как отступы, тени, промежутки между словами и буквами и т. п. В XSL классы свойств и объектов форматирования обеспечивают словарь для отражения цели представления.

XHTML

Расширяемый HTML (extensible HTML, XHTML) является результатом применения правил синтаксиса XML к стандарту HTML. Расширяемость XHTML заключается в возможности определения таких конструкций, как элементы, атрибуты, сущности и т. п. с помощью подключения определения типа документа (Document Type Definition, DTD).

RDF

Инструментарий описаний ресурсов (Resource Description Framework, RDF) — это набор инструментов для работы с метаданными. Он обеспечивает единую, стандартизированную среду управления внутренним (без вмешательства человека) взаимодействием приложений, которые обмениваются в Web информацией, понимаемой машинами. RDF делает ударение на легкость автоматизированной обработки Web-ресурсов. Метаданные RDF могут быть использованы в самых разных областях работы приложений. В общем, RDF обеспечивает основу для элементарных инструментов авторизации, поиска и редактирования Данных, понимаемых машинами, в Web, содействуя, таким образом, трансФормации Web в аппаратно обрабатываемое хранилище информации.

XML Schema

Язык XML Schema используется для объявления элементов и атрибутов в XML-документе с целью его структуризации. Кроме того, этот язык предоставляет расширяемые возможности для определения типов данных элементов и атрибутов.

XBase

XML Base представляет собой аналог элемента BASE из языка HTML. Он описывает механизм предоставления сервисов базовых URI (Uniform Resource Locator, универсальное местоположение ресурса) для XLink.

XLink

Расширяемый язык связывания (XML Linking Language, XLink) определяет конструкции, которые могут быть вставлены в XML-документы для описания связей между объектами. Он использует синтаксис языка XML для создания структур, которые могут служить как для описания простых однонаправленных гиперссылок, определяемых в HTML, так и для более сложных связей.

XPointer

Расширяемый язык указателей (XML Pointer Language, XPointer) — язык, разработанный для совместного использования с XLink. XPointer определяет конструкции, которые поддерживают адресацию во внутренних структурах XML-документов. В частности, он предназначен для специфических ссылок на элементы, символьные строки и другие части XML-документов.

XInclude

Расширяемый язык встраивания (XML Include Language, XInclude) предназначен для объединения блоков информации в формате XML (XML infosets) в единый составной информационный блок. Спецификация документов XML (или информационных блоков), которые должны быть объединены, также процесс объединения описываются с помощью дружественного языку XML синтаксиса (элементов, атрибутов, ссылок на URI).

XQL

Расширяемый язык запросов (XML Query Language, XQL) — это нотация для адресации и фильтрации элементов и текста в XML-документах. XQL является естественным расширением схемы (pattern) синтаксиса XSL. Он обеспечивает выразительную и простую нотацию для указания (pointing) на специфические узлы, а также для поиска узлов со специальными, частными характеристиками. Этот язык, базируясь на возможностях XSL, обеспечивает идентификацию классов узлов путем добавления булевой логики, фильтров, индексации в коллекциях узлов и т. д.

SOAP

Простой протокол доступа к объектам (Simple Object Access Protocol, SOAP) представляет собой облегченный протокол для обмена информации в децентрализованной, распределенной среде. Этот базирующийся на XML протокол состоит из трех частей:

- конверт, который служит для определения содержимого сообщения и способов его обработки;
- набор правил кодировки (encoding rales), предназначенных для описания определенных в приложении типов данных;
- соглашение о вызовах удаленных процедур (Remote Procedure Call, RPC) и об их результатах.

В SOAP заложены потенциальные возможности взаимодействия с другими протоколами, такими как HTTP и его расширенным инструментарием (HTTP Extension Framework).

MathML

Математический язык разметки (Mathematical Markup Language, MathML) — базирующийся на XML язык описания математических формул. Представляет собой расширение XML, предназначенное как для отображения, так и для обработки структуры математических публикаций. Этот язык предназначен для создания инструментария разметки математики в Web.

CML

Химический язык разметки (Chemical Markup Language, CML) является аналогом MathML для публикаций в Web статей по химической тематике.

На самом деле, Internet-технологий, разными способами обрабатывающих данные в Сети и для Сети, гораздо больше. Мы не сильно ошибемся, если всю их совокупность будем обозначать одним термином — XML.

XML среди других серверных технологий

Несмотря на многообразие возможностей, которые предоставляет язык XML, было бы заблуждением думать, что программное обеспечение, необходимое для создания Web-приложения, состоит только из синтаксического анализатора (парсера) XML. Большинство функций этого языка направлено на обработку данных, содержащихся внутри документа, но как обстоит дело с операциями над самими документами? Кроме того, информация, содержащаяся в документе, может быть структурирована различными способами, в зависимости от специфики задачи. Зададимся вопросом, для работы с какими типами таких структур подходит язык XML, а для каких нет?

Мы уже определили в общих чертах, из чего состоит и каким требованиям Должно удовлетворять Web-приложение. Не будем здесь развивать эту тему более подробно, т. к. она представляет собой целую философскую систему и

выходит за рамки этой книги. Просто рассмотрим совокупность функций Web-приложения и языка XML. Для этого предположим, что рассматриваемое нами Web-приложение достаточно компактно, пусть оно обслуживает не более чем несколько сотен пользователей, и в задачи этих пользователей входит только лишь работа с документами. Пусть также наше приложение не производит ни научных расчетов, ни громоздких операций с документами в режиме реального времени. И вместе с тем, оно успешно справляется с поставленными перед ним задачами и делает это с удовлетворительной скоростью.

Прежде всего, заметим, что если мы решили воспользоваться помощью XML для построения Web-приложения, то это еще не значит, что нам необходимо отказаться от программного обеспечения, служащего для представления HTML-документов. Ведь некоторые документы, обрабатываемые приложением, могут служить только для представления информации. Единственная операция, которую может осуществить над ними пользователь - просмотр, поэтому логическая структура таких документов не имеет никакого значения.

Следовательно, для того, чтобы приложение имело возможность обрабатывать статические документы, одним из его компонентов должен быть Web-сервер. Можно, разумеется, перевести все HTML-документы в формат XML, но тогда логично будет заключить, что таким образом Web-приложение будет работать медленнее, т. к. ему понадобится время на преобразование XML в HTML. Хотя нужно признаться, что на практике такое замедление работы случается не всегда.

Основная функция языка XML, как мы уже говорили, состоит в структуризации информации, т. е. создавая XML-документ, мы фактически строим формализованную модель данных. Если при этом количество этих данных слишком велико для того, чтобы они образовывали один документ, то возникает множество таких документов, средствами коммуникации между которыми служат языки запросов.

Даже в том случае, когда количество документов составляет несколько тысяч или десятков тысяч, совокупность их описаний также представляет собой набор данных, который может быть структурирован. Другими словами, можно создать один или несколько XML-документов, которые бы описывали все множество документов, хранящихся на сервере. В них можно включить информацию об уровнях доступа пользователей этого сервера, простые или сложные индексы документов и даже некоторый механизм обработки транзакций.

Но проделав всю эту работу, мы всего-навсего изобретем велосипед под названием "служба каталога". В этом нет никакой необходимости, т. к. на данный момент существует достаточное количество соответствующих серверов на любой вкус как коммерческих, так и бесплатных. Единственным исключением является ситуация, когда количество документов слишком мало для привлечения услуг этой службы.

Так как Web-приложение помимо статических данных обрабатывает еще и динамические, то должен существовать механизм, позволяющий обновлять связи между документами. Это означает, что при изменении какой-либо информации о документе или содержащейся в документе, на которую ссылается другой документ, в этом другом документе также должны произойти соответствующие изменения. Например, создание одного XML-документа может повлечь за собой обновление общего списка этих документов, т. е. другого XML-документа. Другим примером может быть генерирование отчетов — документов, содержащих сводную информацию из других источников.

Опять же, язык XML может быть привлечен для решения этой задачи, но когда речь заходит о каких-либо действиях произвольного характера (то есть не только представление и работа со структурой), то лучше воспользоваться каким-либо языком программирования. Ведь такая простая операция как получение текущей даты не может быть выполнена средствами языка разметки, каким бы расширяемым он не был.

В самом начале мы договорились использовать Java в качестве языка программирования, который, как и XML, представляет собой платформу для создания различных Internet-технологий. Одной из таких технологий является выполненный средствами Java инструмент под названием "сервлет" (servlet), который грубо можно определить как "атомарный" код, выполняющийся на сервере и возвращающий клиенту результаты своего выполнения. Причем клиент и сервер не обязательно должны находиться на различных машинах и взаимодействовать по сети.

Благодаря механизму сервлетов, мы можем объединить функциональность языка XML в плане обработки документов с мощной логикой языка программирования. Подробнее о взаимодействии XML, Java и сервлетов см. *главы 6 и 8.*

Как XML-документы, так и сервлеты представляют собой множество объектов, которое может быть упорядочено. Но поскольку сервлеты представляют собой наборы действий, т. е. программы, и в свою очередь являются приложениями, то для их обслуживания необходим отдельный сервер, который так и называется — сервер приложений (application server). Вообще в задачу сервера приложений входит не только обработка сервлетов, но и любых Других приложений, взаимодействующих с сервером и друг с другом по некоторому единому протоколу.

Так же, как и в случае службы каталога, сферой деятельности сервера приложений является обслуживание подключающихся к нему клиентов (политика пользователей, механизм транзакций и т. п.), а также некоторые специфические функции, связанные с обслуживанием приложений (время жизни при-

ложения, способ хранения информации, которой обмениваются приложения, и т. д.). Как правило, серверы приложений включают в себя и некоторые механизмы защиты от взлома и сбоев оборудования. Среди них средства шифрования, аутентификации, резервного копирования и многое другое.

Здесь мы привели список основных компонентов, образующих Web-приложение для небольшого предприятия (несколько сотен рабочих мест). Вопрос о выборе модели данных и языка для ее описания тесно связан со спецификой решаемой задачи. Поэтому для начала рассмотрим задачи, решаемые с помощью XML.

Круг задач, решаемых с помощью XML

Можно выделить три основных области применения языка XML, если как отправной точкой воспользоваться существующими на сегодняшний день типами моделей данных. Воспользовавшись классификацией баз данных, выделим среди них те, которые наиболее часто используются для решения практических задач:

- одно- или многомерный список (реляционная база данных);
- иерархическая структура (дерево);
- сетевая структура (граф).

В *главе 6* мы более подробно рассмотрим, как именно можно применить язык XML для организации хранилища информации реляционного и иерархического типов.

А здесь мы определим первый класс задач, для которых можно воспользоваться услугами XML. Если данные, с которыми предстоит работать Web-приложению, можно описать с помощью одного из обозначенных типов, и по каким-либо причинам нет необходимости привлекать соответствующее программное обеспечение (например, вследствие малого объема данных, или их простой структуры, или дороговизны этого ПО и т. д.), то для полной или частичной организации информации вполне подойдет язык XML.

Вторым классом задач является ситуация, когда XML берет на себя управление какой-либо частью Web-приложения. Даже при создании Web-сайта, который состоит хотя бы из нескольких HTML-документов, иногда бывает необходимо эти документы как-то упорядочить. Примером, когда использование XML может сильно сэкономить время, является создание карты сайта. Другим примером может быть, в частности, построение пользовательского интерфейса, практической реализации которого посвящена эта книга. Таким образом, в этих и им подобных задачах язык XML служит одним из посредников между массивом данных и к ним обращающимися пользователями или приложениями.

Говоря о том, что главной особенностью языка XML является его способность работать с документом как с множеством динамических данных, а не просто как с текстом, мы вовсе не имели в виду, что эти данные не могут представлять собой текст. Хранение информации в виде текста или гипертекста наиболее естественно для человека, и, вместе с тем, такой способ хранения не укладывается ни в одну из схем моделей данных, описанных ранее. Для более четкого представления того, о чем идет речь, рассмотрим такой пример. Представим себе набор текстов на английском языке, каждый из которых иллюстрирует использование в контексте некоторых содержащихся в нем слов. Изучающий английский язык может на основе этих текстов получить представление о корректном использовании английских слов с неоднозначным переводом (идиом).

Если посмотреть на Web-приложение, осуществляющее таким образом обучение английскому языку нескольких учащихся одновременно, то можно выделить следующие его услуги своему администратору:

- добавление текстового документа;
- удаление документа;
- пометка слова, содержащегося в документе, как ключевого (идиомы);
- удаление этой пометки;
- поиск документа по одному или нескольким ключевым словам (идиомам).

Такого рода задача является примером третьего класса задач, решаемых с помощью XML. Очевидно, что таким образом организованные данные плохо поддаются формализации, и если ключевые слова образуют древовидную структуру, то как быть с остальным текстом? Сразу сознаемся, что язык XML рассматривает и текст и ключевые слова в виде отдельных узлов дерева, но при этом формализованные данные имеют легко читаемый текстовый вид. Кроме того, в подобном случае, чтобы как-либо оперировать текстом или его частями, нет необходимости приводить эти данные к другому, специальному виду. Таким образом, обработка текстовых документов является основным классом задач, для которых язык XML изначально проектировался и, следовательно, подходит лучше всего.

Что содержит эта книга

Разобравшись с тем, что представляет собой Web-приложение и какую роль в его построении играет язык XML, перейдем к вопросам практического Использования языка разметки и к их освещению в данной книге.

Ее первая часть целиком посвящена трем основным составляющим XML: языку стилевых таблиц XSLT, языку адресации XPath и непосредственно

XML, рассмотренным в *главах 1—3*. Эта часть составлена на основе соответствующих спецификаций и может быть использована в качестве справочника. Каждая описываемая конструкция снабжена одним или несколькими примерами, а также подробными объяснениями. В *главе 4* рассматривается и анализируется комплексный пример совместного использования всех трех составляющих XML. Остальные X-технологии не описываются *и не применяются* в последующих главах книги.

Во второй части детально описывается анатомия Web-приложения, предназначенного для небольшого предприятия. Кроме того, здесь сопоставляются различные типы моделей данных и возможности XML. Основой второй части является подготовка читателя к изготовлению пользовательского интерфейса. Здесь излагаются такие практические сведения, как применение каскадных стилевых таблиц для оформления документа, затрагиваются некоторые аспекты использования Java, рассматривается технология клиент-сервер и взаимодействие между клиентом и сервером по протоколу HTTP.

Содержащаяся в *главах 6—8* второй части информация не претендует на полноту освещения тех вопросов, которые здесь затронуты в силу того, что они имеют отдаленное отношение к основной теме — языку XML. Но тем не менее, эта информация или некоторая ее часть необходимы для решения тех задач, которые ставит перед собой книга.

После того, как описаны язык XML и основы построения Web-приложения, можно приступать непосредственно к созданию Web-интерфейса средствами XML, чему и посвящена *глава 9* второй части книги. В этой главе рассматриваются общие принципы построения пользовательского интерфейса, а также представлен комплексный пример Web-приложения, реализующего этот интерфейс.

Таким образом, можно сказать, что книга рассматривает как теоретический, так и практический аспекты технологии XML. Она может быть использована и как справочник (первая часть), и как практическое руководство (часть вторая). Для того чтобы разобраться с теми XML-документами или их частями, которые приводятся в качестве примеров, давайте рассмотрим некоторые специфические особенности используемого программного обеспечения. С этой целью обратимся к соответствующей документации.

Парсеры: Cocoon 1.8.2

Проект Cocoon представляет собой инструментарий публикаций, полностью написанный на Java, который опирается на рекомендованные W3 консорциумом технологии (такие, как DOM, XML и XSL), служащие для обработки Web-информации.

Цель Cocoon заключается в изменении существующих способов создания, представления и хранения данных в Web. Новая парадигма Cocoon основывается на том факте, что содержимое документа (content), его стиль (style) и логика (logic) зачастую создаются различными людьми или рабочими группами. Проект Cocoon стремится к полному разделению эти трех уровней разработки, позволяя им быть раздельно оформленными, созданными и управляемыми. Это создает предпосылки для сокращения времени управления разработками и увеличения количества таких работ, результаты которых можно использовать повторно.

Несмотря на то, что наиболее частое использование Cocoon представляет собой автоматическое создание HTML-кода путем обработки статических или динамически сгенерированных XML-файлов, Cocoon также способен выполнять более сложное форматирование. Это может быть XSL:FO преобразование в PDF-файлы, трансформации, зависящие от клиентского устройства такие, как WML-форматирование для устройств, поддерживающих WAP, или прямое предоставление XML-документа для клиентского программного обеспечения, непосредственно воспринимающего документы XML и XSL.

Для обеспечения этих свойств модель Cocoon разделяет разработку Web-контента на три отдельных уровня.

- Создание XML-документа (XML creation) — XML-файл создается владельцами этого контента. Причем им нет необходимости обладать какими-либо специфическими знаниями о том, какой дальнейшей обработке подвергнется документ. Вместо этого им просто нужно выбрать заранее подготовленный для этого DTD или пространство имен (namespace). Это действие обуславливается или вмешательством человека, или путем динамического генерирования.
- Обработка XML-документа (XML processing) — запрашиваемый XML-файл обрабатывается, и выполняются те действия (логика), которые содержатся в его логических таблицах (logicsheet). В отличие от других динамических генераторов контента, в данном случае логика отделена от остального содержимого файла.
- XSL-представление (XSL rendering) — созданный документ трансформируется с помощью применения к нему стилевой таблицы XSL и форматируется в зависимости от выбранного типа представления (HTML, PDF, XML, WML, XHTML).

В отличие от остальных XML-проектов внимание Cocoon сосредоточено на Разрешении проблем, связанных с инфраструктурой публикаций. В этом отношении можно сказать, что существует множество крупных производителей, которые озабочены исключительно разработкой и внедрением необходимых им технологий, нежели способами их интеграции в единый инструментарий публикаций.

Более подробно о внутреннем устройстве Cocoon, о том, какие технологии он поддерживает и как взаимодействует с некоторыми другими серверными технологиями, можно прочитать в *главе 6*. В этой главе представлен необходимый объем технической информации, согласованный с существующими на сегодняшний день стандартами.

Выбор Web-обозревателя: MSIE 5.5

Помимо лексического анализатора нам необходимо выбрать также программное обеспечение, находящееся на стороне клиента. Как ни удивительно, для раскрытия темы книги нам будет достаточно только лишь одного Web-обозревателя, или, в просторечии, браузера. Хотя с одной стороны, книга посвящена построению интерфейса, а с другой — язык Java предлагает для этого достаточно мощные средства (такие, как Swing), мы ими пользоваться не будем. Перечисленного выше инструментария вполне хватит для удовлетворения наших нужд, но если особо ревностные поклонники Java пожелают установить на стороне клиента виртуальную машину и обогатить свой код разнообразными апплетами, мы не станем против этого возражать...

Итак, браузеры. Ни для кого не секрет, что на данный момент существует два основных вида браузеров: от фирмы Netscape и от Microsoft. Остальные типы браузеров рассматривать не стоит в силу их редкого использования, а также потому, что ни один из них по своим возможностям не превосходит браузеры первых двух типов. Между тем, проводя различия между этими двумя типами, следует отметить, что Microsoft Internet Explorer (MSIE) намного превосходит браузеры Netscape. Наверное, именно поэтому в Сети часто можно встретить предупреждение, что Web-сайт должен просматриваться MSIE версии не ниже 4.0. Некоторые организации, проводящие социологические опросы пользователей Internet, называют в качестве оценки частоту использования MSIE цифру 95%, хотя, может быть, это и происки! одного весьма богатого гражданина США.

Руководствуясь этими и другими соображениями практического характера, мы и выбрали для своих разработок данный браузер. Для того чтобы пояснить свои предпочтения, давайте в общих чертах рассмотрим, что он собой представляет с точки зрения так называемой объектной модели.

Объектная модель представляет собой механизм доступа или программирования документа. Объектная модель DHTML, поддерживающаяся: в MSIE 4.0, обеспечивает доступ почти ко всем элементам и ко всем атрибутам элемента. Начиная с версии 5, объектной модели уже доступен абсолютно каждый элемент и атрибут.

Объектная модель документа (Document Object Model, DOM) является независимым как от платформы, так и от языка интерфейсом, который позволяет языку сценариев (скриптов) обращаться к определенным этой моделью конструкциям и корректировать контент, структуру и стиль документа. DOM включает в себя механизм, с помощью которого происходит объединение стандартного множества объектов HTML и XML, а также инструментарий для доступа к этим объектам и манипуляций с ними.

Ключевым преимуществом DOM является возможность доступа ко всему в документе: для того, чтобы произвести многочисленные изменения контента, и для того, чтобы работать с контентом в раздельных фрагментах документа. Работая совместно с объектной моделью DHTML, DOM расширяет для Web-мастеров возможности построения и управления сложными документами и данными. Многие задачи, например, перемещение объекта из одной части документа в другую, выполняются очень легко и эффективно пользователями DOM.

DOM-интерфейс в Internet Explorer позволяет Web-мастеру получать доступ к различным узлам дерева документа. В качестве этих узлов выступают элементы, атрибуты или текст. Но в то время как DOM стремится к полному решению многих задач, в том числе и выполняемых средствами DHTML, он не содержит всех составляющих объектной модели DHTML. Примером могут служить события (events).

Объектная модель DHTML предоставляет четыре ключевых средства, которые позволяют Web-мастерам создать действительно динамические страницы:

- доступ ко всем элементам таблицы;
- мгновенное обновление страницы;
- полная модель событий;
- изменение текста на странице.

Очень важно объяснить ключевую особенность DHTML. Изменения, которым может подвергнуться страница, могут произойти в любое время: перед загрузкой, в процессе загрузки, после загрузки, когда пользователь нажмет кнопку мыши, когда он переместит указатель мыши или когда на часах будет 12:30.

DHTML представляет собой группу технологий, предназначенных для создания и отображения интерактивных Web-страниц. Они включают в себя динамические стили, динамический контент, двумерное заполнение, привязку данных и эффекты мультимедиа. Объектная модель DHTML делает все эти особенности программируемыми.

Если воспользоваться всеми ими вместе, то результатом будет лучшая производительность со стороны как клиента, так и сервера. Меньшее количество обращений к серверу обуславливается тем, что значения атрибутов или содержимое элементов обновляется "на лету" на стороне клиента. Таким образом, Web-страницы становятся более простыми в разработке и по-настоящему интерактивными для пользователя.

Иллюстративность задачи

Итак, основу специфики наших примеров составляют парсер Cocoon 1.8.2 и браузер MSIE5.5. Иными словами, практическая часть книги реализована с учетом особенностей этих двух программных продуктов. Но это не означает, что без них книга теряет свою практическую ценность. Ведь поддержка как XML, так и HTML многими продуктами реализуется очень похоже. Использование, например, MSIE версии 6.0 может быть обеспечено, если соответствующим образом отредактировать стилевые таблицы CSS в предоставляемых документах.

Совокупность рассматриваемых нами программных продуктов не исчерпывается двумя названными выше приложениями. Детальное описание программной среды, в которой обрабатываются предлагаемые нами XML-документы, можно найти в *главе 6* книги. Подробности установки соответствующего программного обеспечения изложены в *приложении 3,* а описания листингов — в *приложении 2.*

Заметим, что при создании примеров мы преследовали цель — продемонстрировать возможности языка XML, а также показать всю простоту его освоения. Ранее мы примерно очертили тот круг задач, для решения которых может быть использован этот язык, и выяснили то, как с его помощью можно реализовать функции некоторых других серверных служб. Таких, например, как служба каталога.

Разумеется, на практике для решения специфических задач лучше всего использовать специально предназначенное для этих целей программное обеспечение, а не обращаться к возможностям, предоставляемым XML там, где это только возможно. Тем не менее, мы иногда будем применять этот язык не совсем по назначению, но только для того, чтобы наилучшим образом продемонстрировать какую-либо из его возможностей или возможностей Cocoon. Иными словами, предоставляемые нами XML-коды носят чисто иллюстративный характер, т. е. нуждающийся в некоторой доработке для его использования в реальных условиях.

Часть I

XML от А до Я

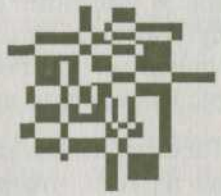
Глава 1. Язык XML

Глава 2. Язык XSLT

Глава 3. Язык XPATH

Глава 4. Комплексный пример

Глава 5. Необходимые дополнения



ГЛАВА 1

Язык XML

1.1. Что такое документ?

В повседневной жизни мы часто сталкиваемся с самыми разными видами документов. В быту это газеты, журналы, книги, письма, документы, удостоверяющие личность и, может быть, какие-либо ее права или полномочия (паспорт, водительские права, свидетельства об уровне образования) и многие другие. В профессиональной деятельности типология обрабатываемых документов еще более многообразна. Кроме того, не стоит упускать из виду, что бумага не является единственным носителем информации и что каждый такой носитель определяет специфику обработки и хранения своего документа. Очевидно, что, например, использование гиперссылки невозможно при работе с бумажными документами.

Проще всего было бы определить документ как некий объем информации, зафиксированный в произвольном виде на каком-либо носителе. Но, понятное дело, такое определение не конструктивно, т. к. предлагает только лишь некоторое поле для философских рассуждений, что плохо согласуется с практическим уклоном стоящих перед нами задач. Поэтому в первом приближении мы определим документ как последовательность символов, содержащую какую-либо информацию.

Любой документ, будь то газета, прайс-лист, письмо и т. д., подпадет под наше определение, и здесь уже можно говорить о создании, удалении или какой-либо еще обработке этого документа. Заметим, что информация, содержащаяся в подобных документах, неоднородна. Газета содержит заголовки Различных уровней, сноски, подписи к фотографиям и т. п. Конверт письма несет информацию об адресе отправителя и получателя. Разделение ин Формации на различные типы и какое-либо выделение этих типов дает нам возможность более тонкой обработки документа, чем просто его созда-

ние и удаление. Если представить себе книгу без оглавления, в которой отсутствуют абзацы, и весь текст, включая заголовки, набран одним и тем же шрифтом, то становится понятным, что наличие даже простейшей структуры в документе придает ему значительную гибкость в обработке.

Рассматривая в качестве примера такие документы, как письмо, мы исходили из того, что работать с ними предстоит человеку. Но если поручить их обработку компьютеру, т. е. коснуться электронных документов, то интуитивно понятной структуры здесь уже недостаточно. Для однозначного определения адреса здесь необходимо четко указать название города, улицы, номера дома и квартиры. Таким образом, должны существовать однозначные обозначения для соответствующих групп символов.

Теперь мы уже можем говорить о документе как о последовательности символов, некоторые группы которых выделены, т. е. интерпретируются особым образом, в зависимости от способа их выделения или, иными словами, от того, как они обозначены. Но и сейчас еще рано говорить о жесткой структуре документа. Помечая различные слова или части текста, мы ничего не сказали о том, как они должны интерпретироваться в зависимости от их местоположения в тексте или относительно друг друга. Возникающую в этой ситуации неоднозначность можно проиллюстрировать следующими двумя примерами.

Если в тексте присутствуют заголовки различной степени вложенности, то для их пометки как заголовков потребуются различные обозначения, причем для заголовка каждого уровня вложенности понадобится отдельное обозначение. Если воспользоваться одним обозначением для всех заголовков, то необходимо будет задать такие правила обработки этого обозначения, которые учитывали бы место вхождения обозначения относительно других его вхождений. Иначе интерпретатор текста не сможет определить для себя степень вложенности заголовков, и все они будут отображены одинаково.

Второй пример заключается в обработке омонимов — слов, которые могут иметь несколько значений в зависимости от контекста. Пример омонима — слово "коса". Оно может означать речную косу, женскую прическу и сельскохозяйственное орудие. В документах чаще всего встречаются омонимы другого рода, когда одно и то же слово используется непосредственно по своему значению и как имя собственное. Например, слово "труд" может быть названием газеты и нарицательным именем. Для того чтобы избавится от неоднозначности, мы, как и в первом примере, можем ввести для каждого смысла омонима уникальное обозначение и использовать его при каждом вхождении этого омонима во всем тексте. Или же мы можем описать правила интерпретации одного единственного обозначения, зависящие от местоположения омонима в этом тексте. Правило может быть,

например, таким: если слово "труд" встречается в самом начале текста (газеты), то его следует трактовать как название газеты, если слово встречается где-либо еще, то его нужно интерпретировать по своему смыслу.

Рассмотрев эти два примера, мы постараемся максимально уточнить понятие документа, насколько это можно сделать, не вдаваясь в строгие технические описания, такие как спецификация языка XML. Хотя, собственно говоря, в этой спецификации строгое и полное определение термину "документ" отсутствует как таковое. В ней сказано лишь нечто вроде: "корректным XML-документом является то, что удовлетворяет данной спецификации". Перед тем, как перейти к рассмотрению XML-документов, давайте остановимся на следующем определении непосредственно документа, безотносительно XML.

Итак, под документом мы будем подразумевать последовательность символов, в которой ни одна из групп этих символов (в т. ч. и вся последовательность целиком) не используется отдельно от какого-либо обозначения (то есть никак не обозначенные наборы отсутствуют), причем для каждого такого набора могут быть заданы правила его интерпретации.

Именно к таким документам мы и будем в дальнейшем применять язык XML. Но перед этим мы более формально рассмотрим понятие XML-документа, опирающееся на приведенное выше определение. Если это определение и связанные с ним рассуждения выглядят слишком запутанными и сложными, то в этом нет большой беды. Суть рассматриваемых концепций станет гораздо более прозрачной при рассмотрении приведенных далее примеров XML-кода, снабженных подробными пояснениями.

1.2. Структура XML-документа

Во введении мы отметили, что будем рассматривать не всю совокупность технологий, связанных с XML, а только лишь те из них, которые имеют отношение к механизму преобразования кода в формате XML, находящегося на сервере, в код формата HTML, или точнее XHTML, который переправляется клиенту и интерпретируется браузером. Поэтому опытным Web-разработчикам, имеющим представление о древовидной структуре HTML-документа или даже о его объектной модели (DOM), структура XML-документа будет более понятна. На интуитивном уровне принципы организации этих двух видов моделей документов очень похожи в силу того, что они имеют одного общего предка — язык SGML.

Исходя из определения документа, мы можем условно разделить содержимое некоторого документа на две составляющие: группы символов, несущие информацию, которую должен представлять документ, и специ-

альные наборы символов, служащие для специфического выделения или пометки некоторых частей первой группы. В XML-документах первая группа носит название "текст" или "контент", а вторая, представляющая собой систему обозначений, которые поддаются интерпретации, называется разметкой. В языке HTML, к примеру, к разметке относятся все теги, без исключения.

Разметка XML-документа представляет собой несколько более сложное понятие, чем в языке HTML. Это связано с тремя следующими обстоятельствами:

- XML-документ может содержать инструкции обработки самого себя. Это означает, что обработчик XML-документа (процессор) является посредником между этим документом и некоторым приложением или приложениями. Такими приложениями могут быть базы данных, сервер каталога, язык программирования, работающий на стороне сервера, и т. п.
- Синтаксис языка XML определен весьма жестко и не допускает многозначного толкования своих правил. Причем любое отклонение от этих правил влечет за собой отказ процессора обрабатывать весь документ. Поэтому в языке XML количество ограничений на синтаксис документа, о которых необходимо помнить разработчику, гораздо больше, чем в языке HTML.
- И наконец, в отличие от языка HTML, разработчик XML-документов сам определяет названия тегов (элементы) и, может быть, правила их интерпретации. Если такие правила присутствуют, то они отделены от непосредственно XML-кода и образуют так называемый сценарий XSLT. Проще говоря, если XML-файл служит не только для хранения каких-либо данных, но и для их представления, то способы представления этих данных вынесены в отдельный XSL-файл.

Согласно тому положению, что структура информации, содержащейся в XML-файле, представляет собой иерархическую модель данных, в корректном документе должен присутствовать единственный корневой элемент. Все остальные элементы являются его непосредственными потомками или потомками других его потомков. Процесс обработки такого документа начинается с первого или корневого элемента, затем обрабатываются узлы дерева второго уровня, затем — третьего и т. д. Здесь важно запомнить, что вся символьная информация, будь то разметка или контент, содержащаяся в XML-документе, представляет собой совокупность узлов некоторого дерева. Подавляющее большинство ошибок возникает в результате того, что наборы символов размечены не однозначно, т. е. процессор не может определить, к какому именно узлу следует отнести данный набор.

л 3. Содержимое XML-документа

Любой законченный неэлектронный документ можно разбить на две основные части. Первая представляет собой так называемую "шапку", в которую входят название документа и какие-либо его характеристики. Вторая — непосредственно "тело" документа. Всем нам приходилось неоднократно писать самые разные заявления — о приеме на работу, об увольнении, о получении денег и т. д. Каждое из них было примерно таким: справа вверху пишется, на чье имя и от кого это заявление, затем идет название документа, слово "Заявление", далее — текст самого заявления и, наконец, дата и подпись. В этом документе телом является текст, идущий после слова "Заявление". Все, что располагается перед телом документа, — шапка.

Точно так же устроен и XML-документ. Давайте рассмотрим следующий пример XML-файла, который называется application.xml (листинг 1.1), т. е. "заявление". В дальнейшем, говоря "XML-файл", мы будем иметь в виду XML-документ без его XSL-составляющей. Таким образом, с технической точки зрения под XML-документом мы будем понимать совокупность файлов с содержимым двух типов — XML и XSL.

Листинг 1.1. Документ application.xml

```
<?xml version="1.0"?>
<?xml-stylesheet href = "application.xsl" type = "text/xsl"?>

<application number="123">

<description>
Директору ООО "Предприятие"
<for>Иванову И. И.</for>
программиста отдела продаж
<from>Петрова П. П.</from>
<!-- от кого и на чье имя заявление -->
</description>

<title>
Заявление
<!-- название документа -->
</title>
```

```
<body>
Прошу предоставить мне отпуск. . .
<!-- тело документа (содержимое заявления) -->
</body>

<date/>
<!-- дата -->

</application>
```

В этом листинге шапкой являются первые две строки, все остальное представляет собой тело документа, имеющее древовидную структуру. Помимо этих двух строк шапка может содержать также одну или несколько инструкций обработки, точное определение которым мы дадим позже.

Теперь, последовательно рассматривая строки листинга, определим назначение и правила синтаксиса каждой встретившейся нам конструкции.

1.4. Заголовок

Первая строка листинга 1.1 представляет собой заголовок, или так называемый *пролог* XML-документа. Согласно спецификации языка XML, заголовок подчиняется следующим правилам синтаксиса:

- заголовок должен начинаться с символов <?;
- перед начальными символами заголовка не должно быть других символов;
- заголовок должен заканчиваться символами ?> ;
- после начальных символов должно стоять слово xml;
- указание версии с помощью конструкции version =". . ." обязательно;
- номер версии на данный момент — 1.0;

О номер версии должен быть заключен в кавычки.

Помимо номера версии заголовок XML-документа может включать в себя объявление кодировки документа, которая определяется с помощью конструкции encoding =". . .". Например:

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

или


```
<?xml encoding="koi8-r" version="1.0"?>.
```

Заголовок может содержать также конструкцию вида standalone =". . .", значение которой может быть yes или по. Для того чтобы объяснить, что она означает необходимо дать некоторые пояснения относительно пролога документа, который может быть несколько больше, чем в нашем примере. Мы предоставим эти пояснения после того, как определим все остальные конструкции, которые могут встретиться в XML-документе.

Здесь мы заметим только, что в дальнейшем не будем использовать ни декларацию типа документа, ни DTD, ни конструкцию standalone =". . .", т. к. для достижения целей книги у нас в этом нет никакой необходимости. На сегодняшний день они используются крайне редко. Гораздо чаще вместо DTD выступает XML Schema, которую, впрочем, мы также не будем использовать. Напомним, что XML Schema применяется в основном для двух целей: описания жестких правил использования элементов и атрибутов при правке XML-документа, а также для указания типов данных, содержащихся в этом документе.

Вторая строка листинга 1.1 служит для подключения к XML-файлу сценария XSLT, который содержит правила интерпретации конструкций, образующих структуру "Заявления". Описание синтаксиса этой строки относится к спецификации языка XSLT и будет приведено в *главе 2,* целиком посвященной этому языку. Здесь мы можем рассмотреть данную строку только как пример инструкции обработки.

1.5. Инструкции обработки

Итак, как уже говорилось ранее, инструкции обработки (processing instruction — PI), служат для указания приложению совершить какие-либо действия. В случае нашего примера это действие заключается в подключении сценария XSLT. Еще один пример — это инструкция обработки парсера Cocoon:

```
<?cocoon-process type="xslt"?>
```

которая указывает ему, что обрабатываемый XML-документ должен быть преобразован в формат HTML.

Для инструкций обработки существует только одно синтаксическое ограничение — она, как и заголовок, должна начинаться с <? и заканчиваться соответственно символами ?>. Кроме того, рекомендуется, чтобы инструкция обработки начиналась адреса того приложения, которому она предназначена. При этом адреса xml, XML и аналогичные им являются зарегистрированными для всех версий спецификации языка XML.

Сходство синтаксиса пролога и инструкций обработки не случайно. Оно восходит к правилам разбора языка SGML, который может обрабатывать XML-документы как некоторый частный случай SGML-документов. Здесь и определяется назначение пролога — указать процессору формат документа.

1.6. Элемент

Понятие элемента (element) является главнейшим в языке XML. Элемент представляет собой логические скобки, в которые помещается информация, выделенная из общего контента документа. Эти логические скобки являются открывающим и закрывающим тегами, либо, в том случае, когда у элемента отсутствует содержимое, это будет тег пустого элемента. Синтаксис записи тегов почти ничем не отличается от их аналога в языке HTML. Сначала идет открывающая угловая скобка (<), затем название тега и закрывающая угловая скобка (>):

```
<tag_name>
```

В случае закрывающего тега после открывающей скобки ставится прямой слеш (/):

```
</tag_name>
```

Если элемент не содержит внутри себя никаких данных, то он называется пустым. Он записывается, как и закрывающий тег, только слеш ставится перед закрывающей скобкой:

```
<tag_name/>
```

В листинге 1.1 это тег `<date/>`, представляющий в документе дату написания заявления.

В силу того, что язык XML является чувствительным к регистру символов то как открывающий, так и закрывающий теги должны быть записаны символами в одном регистре.

Неверно	Верно
`<tag> content </Tag>`	`<tag> content </tag>`
	`<TAG> content </TAG>`

Кроме того, не допускается один или несколько пробелов перед открывающей скобкой тега, хотя в любом другом месте пробелы допустимы. То же касается и символов концов строк. Таким образом, будут верными, например, такие два варианта кода:

```
<tag_name att1="1" att2="2">...</tag_name>
```

и


```
<tag_name
   att1="1" att2="2"
>...
</tag_name>
```

Вот, пожалуй, и все, что касается основ синтаксиса написания тегов в языке XML. Тем, кто не является новичком в области Web-разработок, эти правила написания будут более чем понятны.

Но если с виду теги языков XML и HTML почти ничем не отличаются, то отличия в их предназначении имеют принципиальное значение. В языке HTML теги используются так, чтобы документ имел такой внешний вид, который отвечает заданным требованиям, т. е. служат только для отображения данных. Теги XML предназначены в первую очередь для определения структуры документа. И хотя их порядок влияет на представление, можно сказать, что это их второстепенная функция.

Результатом обработки (processing) листинга 1.1 может быть следующий конечный документ:

```
                                    Директору ООО "Предприятие"
                                              Иванову И. И.
                                  программиста отдела продаж
                                              Петрова П. П.

                          Заявление

Прошу предоставить мне отпуск...
...

                                                Дата: 01.01.01
```

Исходя из получившегося представления, мы можем легко визуально разбить весь этот документ на составные части. Но для электронной обработки XML-документа недостаточно только визуального отделения его частей друг друга. Сравнивая листинг и его представление, уже можно интуитивно уловить, как именно язык XML используется для структурирования данных.

На примере листинга 1.1 мы можем более формально проиллюстрировать древовидную структуру XML-документа "Заявление". Рассмотрим рис. 1.1.

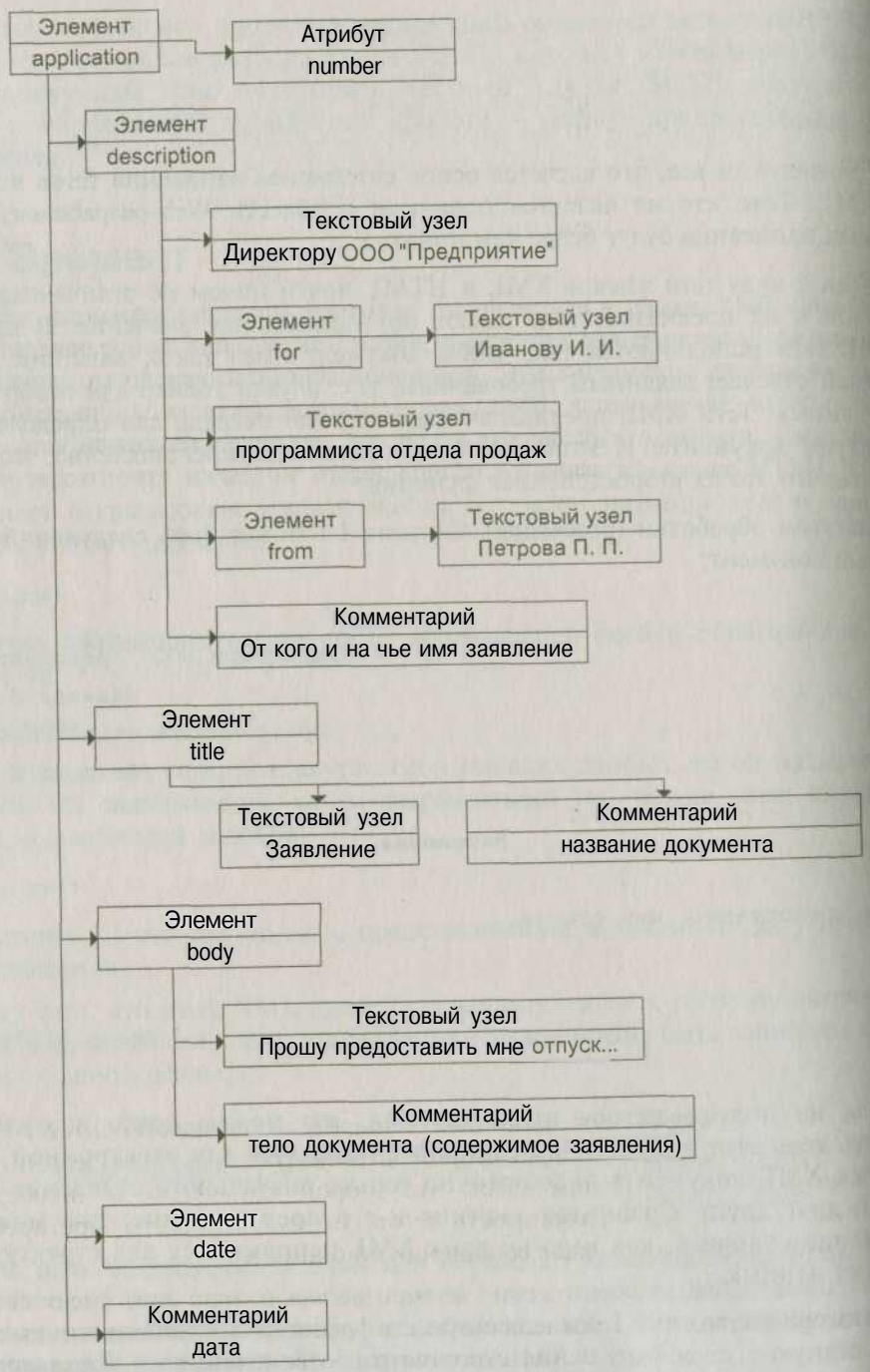


Рис. 1.1. Древовидная структура XML-документа

Корневым элементом XML-документа здесь является элемент <application>...</application>, который содержит в себе все остальные элементы или является предком для любой другой вершины дерева. Заметим здесь, каждый элемент в языке XML, как корневой так и не корневой, может содержать любые другие конструкции XML (кроме, естественно, пролога документа). В нашем случае корневой элемент содержит как текст, так и разметку; другие элементы — комментарии и непосредственно текст.

В частности, элемент application (заявление) включает в себя четыре других элемента: description (описание), title (заголовок), body (тело) и date (дата). Первые три элемента представлены своими открывающим и закрывающим тегами. Здесь четко видно, как эти теги однозначно отделяют друг от друга части документа, обладающие различной смысловой нагрузкой. Эта однозначность в данном случае обуславливается так называемой правильной вложенностью тегов. Сравним два варианта записи некоторой части нашего документа. Первый, демонстрирующий правильную вложенность тегов:

```
<description>
...
</description>
...
<title>
...
</title>
```

И второй, ошибочный вариант:

```
<description>
...
<title>
...
</description>
...
</title>
```

Если в первом варианте мы легко можем определить, что относится к описанию, а что к заголовку, то во втором возникает неоднозначность интерпретации разметки. Добавим здесь, что второй вариант кода, иллюстрирующий неправильную вложенность тегов, не будет проинтерпретирован, вместо этого процессор выдаст сообщение об ошибке.

В зависимости от местоположения элемента, каждый из них имеет собственную характеристику, определяющую его "родственное" отношение к другим элементам. Эти отношения строятся в соответствии со строгой иерархичностью Древовидной модели документа. Каждая вершина дерева документа называется узлом (node).

Перечисленные четыре элемента — description, title, body и date — являются по отношению к корневому элементу application его непосредственными потомками или детьми (child), а по отношению друг к другу — соседними элементами или братьями (sibling). Если мы спустимся по направлению к листьям дерева на один шаг от узла description, то попадем на элементы for и from. Для того чтобы обозначить все родственные связи между этими и некоторыми другими элементами, рассмотрим табл. 1.1, в ячейках которой перечислены родственные отношения элементов в крайней левой колонке по отношению к элементам в первой строке.

Таблица 1.1. *Родственные связи между элементами*

	application	description	for	from	title
application	self	parent	ancestor	ancestor	parent
description	child	self	parent	parent	sibling
for	descendant	child	self	sibling	-
from	descendant	child	sibling	self	-
title	child	sibling	-	-	self

Как можно увидеть, существует всего шесть типов родственных отношений между элементами:

- self — сам элемент; в этом родственном отношении элемент находится по отношению к самому себе;
- child — ребенок или непосредственный потомок;
- parent — родитель или непосредственный предок;
- sibling — брат или соседний элемент;
- ancestor — предок, не обязательно непосредственный;
- descendant — потомок, не обязательно непосредственный.

Такие обозначения используются не только в качестве прозрачной терминологии, но и служат для образования конструкций языка адресации XPath, описанного в *главе 3*.

Завершая описание элементов в языке XML, скажем несколько слов о том, что такое пустой элемент. В листинге 1.1 такой элемент представляет тег <data/>, который служит для вставки текущей даты в документ. Предположим, что рассматриваемый документ является типовым, т. е. пользователь, работая с ним, не создает его от начала и до конца, а просто заполняет не-

который шаблон. При этом какие-то данные в документе могут быть оставлены без изменения, например, на чье имя пишется заявление. То же самое касается и элемента data.

После того как документ сформирован, все его элементы интерпретируются в соответствии со сценарием XSLT. В частности, тег <data/> может быть заменен текущей датой, определяемой по таймеру компьютера. Или какой-либо другой датой, в зависимости от соответствующих директив XSLT. Таким образом, можно сказать, что пустые элементы служат для вставки в документ каких-либо постоянных данных, т. е. таких, которые пользователь не изменит непосредственно. Кроме того, пустые элементы могут употребляться в качестве дополнительных средств разметки, определяющих, например, конец страницы, и во многих других случаях.

1.7. Именатегов

В спецификации XML вся совокупность символов для удобства разбивается на буквы, цифры и остальные символы. И в ней говорится, что имя элемента может начинаться с буквы или с одного из некоторых знаков пунктуации. Для тех, кому не хватает алфавита для того, чтобы назвать элемент, скажем, что из этого правила можно сделать следующий вывод: имя элемента не может начинаться с цифры. Кроме того, заметим, что под "одним из некоторых знаков пунктуации" подразумевается, скорее всего, только один знак — подчеркивание.

"то касается языка, на котором может быть задано имя элемента, то спецификация говорит, что это практически любой язык. Хотя нужно отметить, что не все разработчики синтаксических анализаторов для языка XML утруждают себя поддержкой многоязычное™. На наш взгляд, этот вопрос также не является очень важным. В дальнейшем мы будем использовать английский алфавит и, может быть, цифры для именования элементов. Для написания контента вам вполне достаточно будет поддержки русского языка в Различных кодировках.

Допускает русский язык для именования элементов и атрибутов, использование для этих целей английского языка, по нашему мнению, позволяет при чтении легче отделять разметку от контента. К тому же, при работе с Другим парсером с русскоязычными элементами и атрибутами могут возникнуть проблемы.

Но переносимость XML-документов обеспечивается далеко не только применением Для разметки англоязычного алфавита. Здесь главная проблема заключается в том, что для разных разработчиков естественно использовать Для обозначения частей текста со сходными функциями одни и те же названи...

• Например, для пометки заголовка очень многие выберут элемент

с именем title. В том случае, когда два таких документа начнут взаимодействовать (например, если один из них включает в себя другой), возникает конфликт, связанный с тем, что для элементов с одинаковым именем определены различные способы интерпретации.

Если XML-документ полностью или частично формируется из некоторых XHTML-документов или содержит XHTML-разметку, то такие элементы как title, head, body и т. п. обязательно начнут конфликтовать друг с другом. Для разрешения этих конфликтов существует способ указать для каждого элемента, какой набор правил интерпретации или трансформации следует для него использовать. С этой целью имена элементов объединяют в некоторые группы, которые называются пространством имен (namespace). Каждое из таких пространств характеризуется некоторым префиксом, который ставится непосредственно перед именем из данного пространства.

Примерами пространств имен могут быть следующие наборы имен с префиксами:

- xsl — пространство имен, включающее в себя все директивы языка XSL;
- `xml` — префикс зарегистрирован в спецификации;
- `fo` — (formatting objects) эти имена служат для описания структуры документов, трансформирующихся в формат PDF.

Имена элементов, для которых указан префикс, называются полными именами (qualified name). Их синтаксис строится следующим образом: сначала, после открывающей скобки, идет префикс пространства имен, затем, без пробелов и других разделителей, — двоеточие и имя элемента, и далее — по правилам написания произвольного элемента. В элементе

```
<xsl:value-of select="."/>
```

префиксом пространства имен является xsl, все остальное — сам элемент.

Перед тем, как использовать какое-либо пространство имен, его необходимо объявить до использования имени элемента из этого пространства. Это делается с помощью атрибута с префиксом xmlns: (о том, что такое атрибут, можно узнать в *разд. 1.8*). Значением этого атрибута является URL, который обычно ссылается на организацию, зарезервировавшую это пространство имен. Вообще говоря, это значение может быть произвольным:

```
<my_element xmlns:myns="http:/www.mycompany.com">
...
</my_element>
```


т. к процессор не обращается по этой ссылке. Она служит только для идентификации элемента. Все потомки этого элемента считаются принадлежащими пространству имен myns, если, конечно, для них не указан префикс другого пространства. В этом примере мы объявили свое собственное пространство имен. Как подключить уже существующие, в частности xsl, мы рассмотрим позднее.

Подводя итоги, скажем, что использование стандартных префиксов, таких как `xml`, xsl, `fo` и других, а также двоеточия не допускается при создании пользовательского элемента. Впрочем, с помощью двоеточия можно указать элемент из пространства имен, определенного пользователем. Так как мы не собираемся создавать в этой книге новый стандарт наподобие MathML и не будем подключать DTD к своим XML-файлам, то вопрос создания собственного пространства имен мы оставляем на усмотрение читателя.

1.8. Атрибут

Атрибут в языке XML служит для предоставления дополнительной информации об элементе и его содержимом (в листинге 1.1 атрибутом является номер заявления — `number="123"`). Он выполняет примерно те же функции, что и прилагательное в разговорном языке. При этом роль существительного отводится элементу. Если, например, мы хотим сказать, что зима в этом году теплее, чем обычно, мы говорим "Зима теплая", но можем сказать: Зима. Тепло". Аналогией в XML для первого варианта будет:

```
<season weather="warm">winter</season>
```

хотя будет правильным и

```
<season>
 <weather>warm</weather>
 winter
</season>
```

Точно так же, как прилагательные обогащают разговорную речь, так и атрибуты делают язык разметки более разнообразным. Но в общем, не существует каких-либо указаний в спецификации, что предпочесть, элемент или атрибут. Этот вопрос целиком оставляется на усмотрение разработчика.

Формально атрибут в языке XML представляет собой пару "имя-значение" и записывается следующим образом. Сначала идет имя атрибута, затем, в кавычках, — его значение: weather = "warm". Единственным синтаксическим ограничением для атрибута является уникальность его имени по отношению к содержащему его элементу, т. е. один и тот же элемент не может обладать

двумя атрибутами с одинаковыми именами. При обработке приведенного ниже кода будет выдано сообщение об ошибке:

```
<season weather = "warm" weather = "dirty"/>
```

Но следующий вариант будет верным, хотя и бессмысленным:

```
<season weather = "warm">
 <season weather = "dirty"/>
</season>
```

Пара имя="значение" может содержать пробелы и символы конца строки, но следует помнить, что если они находятся в кавычках, то значение атрибута при этом не является тем же самым, как и без них, т. е. "warm" и " warm " не одно и то же.

В языке XML определены два зарезервированных атрибута, имеющих ограничения на свое значение, т. е. такие, которые пользователь может применять только так, как описано в спецификации. Атрибуты с этими именами употребляются не слишком часто, но, тем не менее, вот они:

- xml:lang — этот атрибут указывает язык, на котором написано содержимое элемента. Примерами его значений могут быть: en, en-us, ru и т. п. Чаще всего этот атрибут используется для выбора процессором для обработки только тех элементов, язык которых понятен пользователю или какому-либо приложению. В документ вставляются элементы с одним и тем же контентом, но набранным на различных языках. Перед просмотром пользователь выбирает язык, и парсер представляет только соответствующие ему элементы. На самом деле процессор не выполняет никаких действий со встретившимся ему атрибутом xml:lang, весь вышеописанный механизм его применения является только одним из возможных и целиком лежит на плечах разработчика.
- xml:space — этот атрибут указывает способ, которым должны обрабатываться пробелы, символы табуляции и конца строки в контенте элемента, обладающего этим атрибутом. Он может принимать два значения: preserve И default. Если атрибут имеет значение preserve, ТО В< вышеперечисленные символы остаются в контенте так, как они были набраны. Если же он имеет значение default, то процессор может поступить с ними по своему усмотрению. Например, синтаксический анализатор языка HTML убирает лишние пробелы, заменяя их одним, символы конца строки игнорирует вовсе. Для каждого элемента языка XML, для которого не задан атрибут xml:lang, значением по умолчанию считается preserve. Это означает, что элемент без этого атрибута ведет себя так, как будто этот атрибут задан, и его значение — preserve.

Рассмотрев основные конструкции XML, такие, как элемент и атрибут, можем перейти к описанию менее функциональных и не таких значительных инструментов этого языка.

1.9. Секция CDATA

Синтаксические правила существуют для того, чтобы процессор мог отличить разметку от контента. Например, для того, чтобы выделить некоторый элемент в виде группы символов, используются угловые скобки. Но как быть в случае, если эти скобки сами являются контентом? Если, например, в тексте встречается что-нибудь вроде "это <слово> не является элементом"? Такая ситуация может возникнуть, в частности, при создании электронной статьи о языке XML, написанной также на языке XML. Именно для подобных ситуаций и предназначена секция CDATA (сокращение от character data). Напомним, что все содержимое XML-документа подразделяется на разметку (markup) и текст (character data). Все, что находится в этой секции, рассматривается процессором при разборе документа как обычный текст. Примером может служить следующий код:

```
<?xml version="1.0"?>
<?xml-stylesheet href = "example.xsl" type = "text/xsl"?>

<example>
```

Это пример самого короткого XML-документа:

```
 <![CDATA[
 <?xml version="1.0"?>

<empty/>
]]>
</example>
```

после разбора которого в браузере будет выведен следующий текст (если, конечно, соответствующим образом задать директивы трансформации в Файле example.xsl):

```
Это пример самого короткого XML-документа:
<?xml version="1.0"?>

<empty/>
```

Здесь мы имеем дело с довольно простым синтаксисом. Секция CDATA представляет собой логические скобки, заключающие в себе не анализируемый процессором текст, или, иными словами, текст, не содержащий разметки. Открывающей скобкой является конструкция <![CDATA[, а закрывающей —]]>, между которыми может находиться все, что угодно, кроме последовательности, закрывающей секцию CDATA. О том, как разместить в документе последовательность символов вида]] >, или одну уг-

ловую скобку, или некоторые другие одиночные символы, можно прочитать далее, *в разд. 1.11.*

В силу чувствительности языка XML к регистру символов разметки слово CDATA должно быть набрано в верхнем регистре. Кроме того, как открывающая, так и закрывающая скобки не должны содержать других символов (пробелов, символов конца строки и т. п.). Завершая этот раздел, отметим, что секция CDATA по своим функциям очень напоминает тег <pre> из языка HTML.

1.10. Комментарий

В языке XML комментарии очень напоминают секции CDATA. Они отличаются только тем, что текст комментария не только не подвергается разбору, но и полностью игнорируется процессором. В листинге 1.1 представлено несколько комментариев, и, как видно из внешнего вида получившегося документа, они полностью отсутствуют в его тексте. Еще один пример комментария:

```
<article>
 <title>... </title>
 <!-- Здесь должен быть заголовок статьи -->
...
</article>
```

Комментарий используется в основном в двух целях:

- для сопровождения кода пояснительным текстом;
- для отладки кода путем превращения некоторых его частей в пропускаемый анализатором текст.

Синтаксис комментария также достаточно прост. Четыре символа (<!--) служат для обозначения его начала, и три (-->) — для его завершения. В текст комментария можно помещать все, что угодно, однако, процитируя спецификацию: "в целях совместимости не следует помещать в текст комментария двойной дефис (--)". Иначе документ перестает быть корректным.

Кроме того, пользуясь комментариями, следует учитывать, что процессор полностью выбрасывает его из документа, не заменяя какими-либо символами (пробелом или концом строки). Поэтому если написать следующий код:

```
<paragraph>
Язык <term>XML</term> - расширяемый<!-- extensible -->язык<!-- Markup
-->разметки<!-- Language -->.
</paragraph>
```

то в результате мы получим следующий текст:

```
Язык XML - расширяемыйязыкразметки.
```

Что, естественно, является неправильным. Поэтому лучше всего, чтобы один или многострочный комментарий содержал целое число строк, т. е. другими придерживаться следующего правила: если какая-либо строка содержит комментарий, то эта строка больше не должна ничего содержать.

1.11. Сущность

Сущности служат для замещения обозначений произвольного контента самим контентом. Они могут быть использованы где угодно в теле документа, но перед этим должны быть объявлены в его прологе.

Чаще всего сущности применяются для того, чтобы заменить часто повторяющиеся фрагменты документа их обозначением. Механизм их использования выглядит примерно так. Некоторый фрагмент документа объявляется как сущность, этой сущности присваивается имя или обозначение, после чего в документ встраивается не сам фрагмент документа, а его обозначение. При этом анализатор, разбирая документ и встретив сущность, подставляет вместо нее ее содержимое — соответствующий фрагмент документа. После этого анализатор возвращается к началу этого фрагмента и продолжает разбор. Если этот фрагмент сам содержит сущности, то весь процесс повторяется до тех пор, пока все сущности в документе не будут разобраны.

Примерами часто повторяющихся частей документа могут быть название компании, ее логотип, торговая марка продукта (или любой другой символ, не представленный на клавиатуре) и т. п. Определить их как сущности можно примерно так:

```
<?xml version="1.0"?>
<!DOCTYPE letter [
 <!ENTITY company "mycompany">
<!ENTITY logo SYSTEM "logo.gif" NDATA GIF>
]>
```

и затем использовать в документе:

```
<letter>
&logo;
<title> Рекламное письмо </title>
 Компания &company; приглашает к сотрудничеству региональных дилеров...
...
&logo;
</letter >
```

К к видно из этого кода, синтаксис объявления сущности состоит из пяти компонент:

- а открывающая объявление логическая скобка объявления (<!ENTITY);
- а имя сущности в нашем случае это — company и logo);
- содержимое сущности (контент для замены) в кавычках ("mycompany" и "1 o if") синтаксис содержимого сущности нечувствителен к пробелам, символам табуляции и конца строки;
- опциональный тип сущности и нотация;
- завершающая логическая скобка объявления сущности (>).

Причем объявление одной или нескольких сущностей должно находиться в объявлении типа документа (<!DOCTYPE root-element [...]>), на котором мы более подробно остановимся в *разд. 1.12*. Если одна и та же сущность объявлена более одного раза, то процессор обязан использовать ее первое объявление.

После того как сущность объявлена, мы можем использовать ссылку на нее в теле документа. Для этого в том месте документа, где должен появиться замещающий контент, необходимо установить так называемую ссылку на сущность (&company; или &logo;). Результатом разбора нашего документа будет следующее:

```
рисунок (logo.gif)

Рекламное письмо

Компания mycompany приглашает к сотрудничеству региональных дилеров...

...

рисунок (logo.gif)
```

Ссылка на сущность состоит из трех частей, записываемых без пробелов и других разделителей: сначала ставится амперсант (&), затем — имя сущности, и завершает ссылку точка с запятой (;).

Если обратить внимание на то, как записываются объявления сущностей в нашем примере, мы можем заметить, что сущности отличаются друг от друга по типу содержимого. Первая содержит текст, вторая — рисунок. Я самом деле классификация сущностей охватывает гораздо большее количество типов и имеет разветвленную структуру (рис. 1.2).

Рис. 1.2. Типы сущностей

Сущности отличаются друг от друга не только синтаксисом объявления, но и синтаксисом ссылки. Помимо описанной формы записи ссылки на сущность (&entity_name;), существует И еще одна (%entity_name;). Это Т. Н. параметрическая сущность, которую мы рассмотрим отдельно.

Теперь же (см. рис. 1.2), рассмотрим по порядку всевозможные типы сущностей.

1.11.1. Символьные сущности

Символьные сущности служат для вставки символов, не представленных на клавиатуре, либо специальных символов. Всего существует три подтипа символьных сущностей: нумерованные, предопределенные и именованные.

Нумерованные сущности предназначены для вставки символа по его коду в наборе символов Unicode. Этот набор содержит 65535 символов, представляющих собой алфавиты различных языков, знаки пунктуации, а также некоторые специальные символы: значок торговой марки (™), символ вложенности (c) и т. п. Ссылки на эти сущности можно записать двумя способами, отличающимися друг от друга системой счисления, в которой записан код этого символа. Система счисления может быть десятичной или шестнадцатеричной. Синтаксис ссылки на нумерованные сущности отличается наличием решетки (#) после амперсанта: Þ (десятичная система) или â (шестнадцатеричная система, о которой говорит "икс"

(x) перед кодом символа). Нумерованные сущности не нуждаются в объявлении, корректная их замена соответствующим символом в соответствии со спецификацией должна обеспечиваться процессором XML.

Вторым типом символьных сущностей является предопределенный тип. В *разд. 1.9,* описывающем секцию CDATA, мы обещали рассмотреть способ вставки в документ таких одиночных символов, как, например, угловая скобка. Проблема состояла в том, чтобы указать процессору, где такая скобка является частью разметки, а где частью контента. Можно использовать для ее разрешения секцию CDATA, но такой вариант выглядит слишком громоздким. В самом деле, не очень удобно для вставки такой скобки каждый раз писать:

```
<![CDATA[<]]> или <![CDATA[>]]>
```

Именно для вставки таких символов и предназначены предопределенные сущности. Их всего пять, и они перечислены в табл. 1.2.

Таблица 1.2. *Предопределенные сущности*

Имя	Контент	Значение
Lt	<	less then — знак меньше, чем
Gt	>	greater then — знак больше, чем
Amp	&	ampersand — амперсант
Quot	"	quotation — кавычки
Apos	'	apostrophe — апостроф

Таким образом, для вставки в документ, скажем, знака "меньше" (<) достаточно вместо него написать ссылку на соответствующую сущность (<).

Предопределенные сущности, как и нумерованные, также не нужно объявлять в документе.

Последним типом сущностей являются именованные сущности, функции которых ничем не отличаются от функций нумерованных. Отличия и только в синтаксисе. Если нумерованная сущность определяется символа, то именованная — его именем, заданным пользователем. Например, ссылка на сущность, содержащую символ авторских прав может быть &copy_right; или &cr;. Из того, что имя такой сущности определяет пользователь, следует, что этот тип сущности нуждается в объявлении. В нашем случае это будет:

```
<!ENTITY copy_right "&#169">
```


Таким образом, вопрос предпочтения между нумерованными и именованными сущностями заключается в следующем: в отличие от нумерованных сущностей, именованная сущность не требует запоминать код символа, но ее необходимо объявлять.

11.2. Сущности со смешанным контентом

Сущности этого типа позволяют вставлять в документ не одиночный символ, а целые наборы символов. Эти наборы могут содержать как текст, так и разметку. Причем разметка должна быть корректной, в силу того, что, как мы говорили ранее, после замещения сущности ее содержимым процессор переходит к разбору этого содержимого.

Сущности со смешанным контентом нуждаются в объявлении вида:

```
<!ENTITY mc "<my_company>My company name</my_company>">
```

Заметим, что содержимое сущности записано "как есть", т. е. мы используем разметку непосредственно, вместо того, чтобы пользоваться предопределенными сущностями для вставки угловых скобок. Это связано с некоторыми особенностями синтаксиса пролога, о которых мы поговорим позже.

1.11.3. Неанализируемые сущности

Неанализируемые сущности в языке XML представлены для вставки в документ нетекстового контента, являющегося некоторым "сырым" потоком данных, который может обработать приложение, взаимодействующее с этим документом. В качестве примера мы приводили объявление сущности, служащей для вставки рисунка в формате GIF:

```
<!ENTITY logo SYSTEM "logo.gif" NDATA GIF>
```

Данном случае приложением, умеющим обрабатывать контент этой сущности, является браузер. Кроме браузера, таким приложением будет также любая программа, которую этот браузер может вызвать, или какое-либо серверное приложение.

Приведенную выше сущность делает неанализируемой ключевое слово NDATA (Not Data — не данные), после которого следует так называемый идентификатор нотации (notation identifier). Этот идентификатор сообщает процессору, что, во-первых, процессор не должен сам обрабатывать контент сущности и, во-вторых, о том, в каком формате записан этот контент. Формальных правил для того, что именно должен предпринимать процессор, если встретит неанализируемую сущность, не существует. Также Не известно, что он должен делать с информацией о формате контента. Таким образом, на усмотрение конкретной реализации процессора оставляет-

ся вопрос о передаче (или отказе от нее) этого контента какому-либо приложению в зависимости от формата. Ясно только то, что процессор не будет подвергать разбору контент неанализируемой сущности. Именно отсюда и следует ее название.

1.11.4. Внутренние и внешние сущности

Помимо сущностей, представленных на рис. 1.2, существует еще два их типа, которые определяются в зависимости от того, где находится контент сущности. Если объявление и содержимое сущности содержится внутри XML-документа, то такая сущность называется внутренней. Если же сущность служит обозначением для какого-либо внешнего объекта или сама объявлена во внешнем наборе DTD, то она называется внешней.

Предопределенные и именованные сущности не подразделяются на внутренние и внешние, т. к. они вовсе не нуждаются в объявлении. Неанализируемые сущности, в свою очередь, не могут быть внутренними по определению, т. к. они ссылаются на данные, которые процессор не должен разбирать, и, следовательно, эти данные не могут содержаться внутри XML-документа. Все остальные сущности могут быть как внутренними, так и внешними.

Синтаксис внешних сущностей характеризуется наличием ключевых слов PUBLIC и/или SYSTEM, которые называются соответственно публичным (или открытым) и системным идентификаторами:

```
<!ENTITY logo SYSTEM "logo.gif" NDATA GIF>

<!ENTITY external PUBLIC "-//something//somewhere//EN">

<!ENTITY ext_entity "http://www.mycompany.com/needed.xml">

<!ENTITY ext_ent PUBLIC "-//something//somewhere//EN"
"http://www.mycompany.com/needed.xml">
```

Системный идентификатор используется для указания явным образом адреса внешнего источника, содержащего контент сущности. Как видно из приведенных выше примеров, этим адресом является URL источника. Публичный идентификатор предназначен для тех случаев, когда внешний источник данных не может быть обнаружен по этому URL. Такое может произойти, например, когда URL источника задан неверно или просто был изменен владельцами.

Стандарт для публичного идентификатора еще не завершен, но предполагается, что он будет ссылаться на запись в неком едином реестре ресурсов,

которая не будет зависеть от изменения URL этих ресурсов. Поэтому на данный момент для внешних сущностей можно использовать только системный идентификатор и тот адрес, который представляет собой URL ресурс, хранящийся на том же сервере, что и обрабатываемые XML-документы.

1.12. Пролог документа и DTD

Напомним, что для решения стоящих перед нами задач, мы не будем использовать ни пролог документа, ни файлы DTD, а также любые другие конструкции, с ними связанные (например, атрибут standalone). Поэтому данный раздел носит описательный характер и не будет большой беды, если он будет пропущен.

Ранее, когда мы говорили о прологе документа, мы упоминали две конструкции. Первая, которую мы условно назвали заголовком, называется объявлением XML, а вторая (необязательная) — объявлением типа документа.

```
<?xml version="1.0"?>
 <!-- Объявление XML -->

<!DOCTYPE doc [
 <!ENTITY cmp "My company name">
 <!ENTITY extdata SYSTEM "http://www.mycmp.com/data/thisdata.xml">
...
]>
 <!-- Объявление типа документа -->
```

Объявление XML предоставляет процессору информацию о документе в целом, объявление типа документа, в свою очередь, служит для описания его структуры. До этого момента мы использовали объявление типа документа только в качестве контейнера для объявлений сущностей, хотя функции этой конструкции значительно шире. Давайте рассмотрим их более подробно.

Синтаксис объявления типа документа выглядит так:

```
<!DOCTYPE
root_element
uri_of_dtd
[
...
]>
<!-- конец объявления типа документа -->
```

```
<root_element>
...
</root_element>
```

Первая строка представляет собой начало объявления типа документа и состоит из девяти символов (простой способ указать отсутствие пробелов, знаков табуляции и конца строки) — <!DOCTYPE. Во второй строке указывается имя корневого элемента документа. Третья строка содержит уникальный идентификатор ресурса (URI), который представляет собой файл DTD. В этом файле содержится все то, что не вошло во внутренние объявления, находящиеся в квадратных скобках.

Перед тем как перейти к рассмотрению того, что может находиться в этих скобках (или в файле DTD), сделаем несколько замечаний.

- Наличие URI на файл DTD не является обязательным.
- Синтаксис URI файла DTD совпадает с синтаксисом адреса источника данных для внешних сущностей. То есть URI представляет собой системный и/или публичный идентификатор.
- Описанный ранее атрибут standalone (автономный), принимающий значение yes или no и содержащийся в объявлении XML, говорит о том, имеет ли документ внешние источники данных (внешние сущности и файл DTD) или является автономным.

Определение типа документа (DTD — Document Type Definition) предназначено, в основном, для наложения ограничений на использование элементов и атрибутов в документе. Описывая элементы и атрибуты, мы говорили, что их имена, порядок использования, тип содержимого и т. п. остаются целиком на усмотрение пользователя. Но в то же время мы определили XML не как некий язык разметки, а как средство для создания такого языка. Чтобы проиллюстрировать эти высказывания, рассмотрим небольшой фрагмент файла DTD:

```
<!ENTITY % QName "NMTOKEN">
<!ENTITY % NCName "NMTOKEN">

<!ELEMENT simpleType ((annotation)?,
          (restriction|list|union))>
<!ATTLIST simpleType
          name %NCName; #IMPLIED
          id ID #IMPLIED>
<!ELEMENT list ((annotation)?,(simpleType)?)>
<!ATTLIST list itemType %QName; #IMPLIED
          id ID #IMPLIED>
```


В начале этого фрагмента определяется набор нескольких параметрических сущностей, синтаксис которых отличается от синтаксиса общих сущностей наличием знака процента (%) вместо амперсанта. Параметрические сущности, точно так же, как и общие, служат для обозначений часто повторяющегося контента, но используются исключительно в DTD, т. е. если процессор встретит ссылку на параметрическую сущность где-либо вне внутреннего или внешнего набора DTD, то будет зафиксирована ошибка.

Далее в листинге мы сталкиваемся с двумя конструкциями <!ELEMENT...> и <!ATTLIST...>, которые позволяют описать элементы и их атрибуты. Разберем первое вхождение в листинг этой пары:

```
<!ELEMENT simpleType ((annotation)?,
          (restriction|list|union))>
<!ATTLIST simpleType
          name %NCName; #IMPLIED
          id ID #IMPLIED>
```

В объявлении свойств говорится, что элемент должен иметь имя simpleType, в котором содержится необязательный элемент annotation и какой-либо один обязательный элемент из элементов restriction, list или union.

Далее для элемента simpleType идет объявление его атрибутов. В частности, определяется атрибут name, имеющий тип NMTOKEN (тип образуется путем замены ссылки на параметрическую сущность %NCName; на ее содержимое — NMTOKEN). Тип NMTOKEN называется меткой имени. Он говорит о том, что значение атрибута этого типа должно начинаться с буквы и содержать буквы, цифры и некоторые знаки препинания. Кроме типа атрибут характеризуется еще и режимом, который накладывает ограничение на появление в документе атрибута и его значение. Режим #IMPLAIED означает, что атрибут необязателен и им можно пренебречь. В противоположность ему режим #REQUIRED требует от пользователя задания значения атрибута для данного элемента.

Мы не станем перечислять все типы и режимы атрибутов и описывать синтаксис каких-либо конструкций DTD. Для того чтобы получить некоторое представление о его функциях, достаточно в общих чертах рассмотренного примера. Мы лишь перечислим список некоторых из этих функций:

- объявление сущностей, элементов и атрибутов;
- определение порядка следования элементов;
- определение содержимого элементов;
- определение типа и режима появления атрибутов;
- объявление нотаций для задания типов данных атрибутов и неанализируемых сущностей;

В этой PI содержится указание XSP-процессору сгенерировать из исходного XML-документа Java-код и откомпилировать его. При использовании этой PI несколько меняется механизм взаимодействия клиентского программного обеспечения с XML-документом. Без этой инструкции браузер (или другой клиент) обращаясь к документу, заставляет парсер преобразовывать XML в HTML и отсылать результат к себе. В противном случае HTML-код генерируется не парсером, а байтовым кодом Java, получившимся в результате предыдущего преобразования XML в Java-код и последующей его компиляции.

для более ясного понимания принципа работы этих PI нам понадобится описать инфраструктуру парсера Cocoon, а также рассмотреть, что такое XSP-процессор. Сведения об этом и некоторая другая информация о Cocoon будут приведены в *главе 6*.

- объявление условных секций, позволяющих включать или исключать фрагменты DTD (используется, в основном, в целях отладки как "условный" комментарий).

Напомним еще раз, что содержимое файла DTD может находиться и в прологе документа. При этом документ, использующий DTD, называется действительным, а тот, который DTD не обладает, соответственно — недействительным. В свою очередь, процессоры, обрабатывающие как внутренний так и внешний наборы DTD, называются проверяющими, а остальные — непроверяющими. При этом если документ содержит внутренний набор DTD, то непроверяющий процессор должен обработать некоторую его часть, удовлетворяющую определенным условиям, которые мы описывать не будем. Здесь нужно обратить внимание, что действительный (valid) документ и корректный (well-formed) — это не одно и то же.

В качестве примера использования DTD можно привести язык HTML. Этот язык разметки был создан средствами языка SGML, путем создания соответствующего DTD. Все элементы HTML имеют фиксированные имена тегов и атрибутов, и их типы также строго определены. В этом и заключается разница между языком разметки (таким, как HTML) и средствами для его создания (такими, как XML и SGML).

1.13. Особенности Cocoon

Более подробно о возможностях парсера Cocoon мы поговорим несколько позднее. Здесь мы перечислим только некоторые его особенности, лежащие в плоскости непосредственно XML, т. е. безотносительно XSLT и XPath. Эти его свойства созданы для внутреннего использования процессором. Они не предназначены для замены или расширения каких-либо стандартных конструкции языка. Проще говоря, это несколько инструкций обработки, знакомых этому конкретному синтаксическому анализатору. Рассмотрим простой пример инструкции обработки.

```
<?cocoon-format type="text/xslfo"?>
```

Эта инструкция (Processing Instruction, PI) сообщает Cocoon о том, что содержимое XML-файла должно быть преобразовано в формат PDF в соответствии со спецификацией Formatting Objects. Данная PI приведена здесь только "для полноты картины", и в отличие от двух последующих, мы не будем ею пользоваться.

```
<?cocoon-process type="xslt"?>
```

Эта инструкция заставляет процессор использовать для интерпретации XML-файла сценарий XSLT, который находится в отдельном файле.

```
<?cocoon-process type="xsp"?>
```

- объявление условных секций, позволяющих включать или исключать фрагменты DTD (используется, в основном, в целях отладки как "условный" комментарий).

Напомним еще раз, что содержимое файла DTD может находиться и в прологе документа. При этом документ, использующий DTD, называется действительным, а тот, который DTD не обладает, соответственно — недействительным. В свою очередь, процессоры, обрабатывающие как внутренний так и внешний наборы DTD, называются проверяющими, а остальные — непроверяющими. При этом если документ содержит внутренний набор DTD, то непроверяющий процессор должен обработать некоторую его часть, удовлетворяющую определенным условиям, которые мы описывать не будем. Здесь нужно обратить внимание, что действительный (valid) документ и корректный (well-formed) — это не одно и то же.

В качестве примера использования DTD можно привести язык HTML. Этот язык разметки был создан средствами языка SGML, путем создания соответствующего DTD. Все элементы HTML имеют фиксированные имена тегов и атрибутов, и их типы также строго определены. В этом и заключается разница между языком разметки (таким, как HTML) и средствами для его создания (такими, как XML и SGML).

1.13. Особенности Cocoon

Более подробно о возможностях парсера Cocoon мы поговорим несколько позднее. Здесь мы перечислим только некоторые его особенности, лежащие в плоскости непосредственно XML, т. е. безотносительно XSLT и XPath. Эти его свойства созданы для внутреннего использования процессором. Они не предназначены для замены или расширения каких-либо стандартных конструкции языка. Проще говоря, это несколько инструкций обработки, знакомых этому конкретному синтаксическому анализатору. Рассмотрим простой пример инструкции обработки.

```
<?cocoon-format type="text/xslfo"?>
```

Эта инструкция (Processing Instruction, PI) сообщает Cocoon о том, что содержимое XML-файла должно быть преобразовано в формат PDF в соответствии со спецификацией Formatting Objects. Данная PI приведена здесь только "для полноты картины", и в отличие от двух последующих, мы не будем ею пользоваться.

```
<?cocoon-process type="xslt"?>
```

Эта инструкция заставляет процессор использовать для интерпретации XML-файла сценарий XSLT, который находится в отдельном файле.

```
<?cocoon-process type="xsp"?>
```


В этой PI содержится указание XSP-процессору сгенерировать из исходного XML-документа Java-код и откомпилировать его. При использовании этой PI несколько меняется механизм взаимодействия клиентского программного обеспечения с XML-документом. Без этой инструкции браузер (или другой клиент) обращаясь к документу, заставляет парсер преобразовывать XML в [illegible] и отсылать результат к себе. В противном случае HTML-код генерируется не парсером, а байтовым кодом Java, получившимся в результате предыдущего преобразования XML в Java-код и последующей его компиляции.

Для более ясного понимания принципа работы этих PI нам понадобится описать инфраструктуру парсера Cocoon, а также рассмотреть, что такое XSP-процессор. Сведения об этом и некоторая другая информация о Cocoon будут приведены в *главе 6*.

- объявление условных секций, позволяющих включать или исключать фрагменты DTD (используется, в основном, в целях отладки как "условный" комментарий).

Напомним еще раз, что содержимое файла DTD может находиться и в прологе документа. При этом документ, использующий DTD, называется действительным, а тот, который DTD не обладает, соответственно — недействительным. В свою очередь, процессоры, обрабатывающие как внутренний так и внешний наборы DTD, называются проверяющими, а остальные — непроверяющими. При этом если документ содержит внутренний набор DTD, то непроверяющий процессор должен обработать некоторую его часть, удовлетворяющую определенным условиям, которые мы описывать не будем. Здесь нужно обратить внимание, что действительный (valid) документ и корректный (well-formed) — это не одно и то же.

В качестве примера использования DTD можно привести язык HTML. Этот язык разметки был создан средствами языка SGML, путем создания соответствующего DTD. Все элементы HTML имеют фиксированные имена тегов и атрибутов, и их типы также строго определены. В этом и заключается разница между языком разметки (таким, как HTML) и средствами для его создания (такими, как XML и SGML).

1.13. Особенности Cocoon

Более подробно о возможностях парсера Cocoon мы поговорим несколько позднее. Здесь мы перечислим только некоторые его особенности, лежащие в плоскости непосредственно XML, т. е. безотносительно XSLT и XPath. Эти его свойства созданы для внутреннего использования процессором. Они не предназначены для замены или расширения каких-либо стандартных конструкций языка. Проще говоря, это несколько инструкций обработки, знакомых этому конкретному синтаксическому анализатору. Рассмотрим простой пример инструкции обработки.

```
<?cocoon-format type="text/xslfo"?>
```

Эта инструкция (Processing Instruction, PI) сообщает Cocoon о том, что содержимое XML-файла должно быть преобразовано в формат PDF в соответствии со спецификацией Formatting Objects. Данная PI приведена здесь только "для полноты картины", и в отличие от двух последующих, мы не будем ею пользоваться.

```
<?cocoon-process type="xslt"?>
```

Эта инструкция заставляет процессор использовать для интерпретации XML-файла сценарий XSLT, который находится в отдельном файле.

```
<?cocoon-process type="xsp"?>
```

В этой PI содержится указание XSP-процессору сгенерировать из исходного XML-документа Java-код и откомпилировать его. При использовании этой PI несколько меняется механизм взаимодействия клиентского программного обеспечения с XML-документом. Без этой инструкции браузер (или другой клиент) обращаясь к документу, заставляет парсер преобразовывать XML в HTML и отсылать результат к себе. В противном случае HTML-код генерируется не парсером, а байтовым кодом Java, получившимся в результате предыдущего преобразования XML в Java-код и последующей его компижции.

Для более ясного понимания принципа работы этих PI нам понадобится описать инфраструктуру парсера Cocoon, а также рассмотреть, что такое XSP-процессор. Сведения об этом и некоторая другая информация о Cocoon будут приведены в *главе 6.*

ГЛАВА 2

Язык XSLT

2.1. Схема работы XSLT

В первой главе книги мы описали, из чего состоит и как работает язык XML. Вообще говоря, под языком XML можно понимать две вещи. Обычно, когда говорят об XML, имеют в виду всю совокупность технологий, связанных с разметкой и имеющей в своем названии слово extensible — расширяемый. Другое понимание языка XML подразумевает только то, что определено в соответствующей спецификации, т. е. непосредственно XML, безотносительно XSLT, XPath и т. п. На наш взгляд оба эти толкования имеют право на жизнь, но мы для удобства будем пользоваться, как правило, только первым из них. И если где-либо мы отступим от этого правила, то смысл понятия "язык XML" будет ясен из контекста.

Сведения, содержащиеся в первой главе, позволяют использовать XML только для хранения информации. Мы можем с его помощью структурировать электронные данные, поместить их в один или несколько файлов и затем посредством парсера обращаться ко всему документу целиком или к каким-либо его частям. Применяя средства, реализующие механизм запросов к данным в формате XML и некоторые другие приложения, мы сможем создать полный набор операций над документами и информацией, содержащейся внутри документов. Здесь имеются в виду такие операции, как создание и удаление документа, а также создание, удаление и редактирование узлов дерева, образующего внутреннюю структуру содержимого этого документа.

Но все эти операции мы будем вынуждены выполнять вслепую. Это означает, что спецификацией языка XML не предусмотрены никакие средства для представления содержащихся в документе данных. И хотя инструментарий, рассмотренный в первой главе, предусматривает широкий спектр операций над этими данными, он не описывает ни одного

способа увидеть их или результаты операций над ними. Именно этим целям и служит язык XSLT (extensible Stylesheet Language for Transformation —расширяемый язык стилей для трансформаций), которому целиком посвящена данная глава.

Как уже говорилось ранее, более общим, чем XSLT, средством для представления данных в формате XML является XSL. Он представляет собой самый полный стандарт в этой области, и язык XSLT является его подмножеством. Напомним, что XSL описывает стилевую таблицу (stylesheet), в которой для каждой конструкции XML, содержащейся в документе, заданы правила его представления. Язык XSLT, в свою очередь, служит для преобразования одного XML-документа в другой. Можно сказать, что XSL трансформирует документ с самыми разными целями и в самые разные форматы (например, в PDF), а XSLT в качестве формата для результата трансформации использует исключительно XML.

Мы можем воспользоваться этим свойством XSLT и преобразовать заранее созданный XML-документ в формат XHTML, который уже умеет себя представлять. Напомним, что XHTML — это результат применения к стандарту HTML 4.0 синтаксиса XML (для каждого открывающего тега задан закрывающий тег, правильная вложенность тегов, значения атрибутов указаны в кавычках и т. д.). Рассмотрим, как происходит это преобразование.

Мы уже неоднократно говорили, что доступный для просмотра XML-документ состоит, как минимум, из двух файлов: непосредственно XML-файла, в котором хранятся данные, и XSLT-файла, содержащего правила их интерпретации. Как же происходит эта интерпретация? Путем трансформации исходного дерева, представленного в XML-файле, в конечное дерево в формате XHTML с помощью директив XSLT.

Вспомним листинг 1.1, описывающий документ под названием "Заявление". Во второй строке файла application.xml расположена инструкция обработки, с указанием процессору воспользоваться для трансформации документа набором правил, содержащихся в файле application.xsl:

```
<?xml-stylesheet href = "application.xsl" type = "text/xsl"?>
```

Спецификация XSLT не конкретизирует, как именно стилевые таблицы привязываются к документу XML, на этот счет существуют лишь рекомендации. Поэтому в реализации других парсеров для этих целей может быть выбрана другая инструкция обработки, но скорее всего ее вид будет не сильно отличаться от вышеприведенного.

В примере 1 помимо листинга мы также привели один из возможных способов представления этого документа, т. е. показали, как он может выглядеть. Рассмотрим соответствующий сценарий XSLT, который и определяет такой внешний вид документа (листинг 2.1).

Листинг 2.1. Сценарий application.xsl

```
<?xml version="1.0" encoding="koi8-r"?>

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

<xsl:template match="application">
 <xsl:processing-instruction name="cocoon-format">
type="text/html"</xsl:processing-instruction>
 <html>
  <body>
   <xsl:apply-templates/>
  </body>
 </html>
</xsl:template>

<xsl:template match="description">
 <div style="text-align:right;">
  <div style="text-align:left;width:200px">
   <xsl:apply-templates/>
  </div>
 </div>
</xsl:template>

<xsl:template match="for|from">
 <b>
  <xsl:apply-templates/>
 </b>
 <br/>
</xsl:template>

<xsl:template match="title">
 <h1 style="text-align:center">
  <xsl:apply-templates/>
 </h1>
</xsl:template>

<xsl:template match="body">
 <p style="text-indent:1.5em">
  <xsl:apply-templates/>
 </p>
</xsl:template>

<xsl:template match="date">
Дата: 01.01.01
</xsl:template>

</xsl:stylesheet>
```

для полной работоспособности пары файлов, представленных листингами 1.1 и 2.1, в XML-файл необходимо вставить одну из инструкций обработки, описанных в конце *разд. 1.5:*

```
<?cocoon-process type="xslt"?>
```

Ее можно вставить в промежуток между первой и второй строкой. Кроме того, для поддержки русского языка в заголовке каждого файла, содержащего русский текст, нужно указать соответствующую кодировку, определяемую атрибутом encoding, например, так:

```
<?xml version="1.0" encoding="koi8-r"?>
```

Теперь, если XML- и XSL-файлы (обозначенные здесь и далее по своим расширениям) набраны верно, в окне браузера можно будет увидеть документ, который изображен на рис. 2.1.

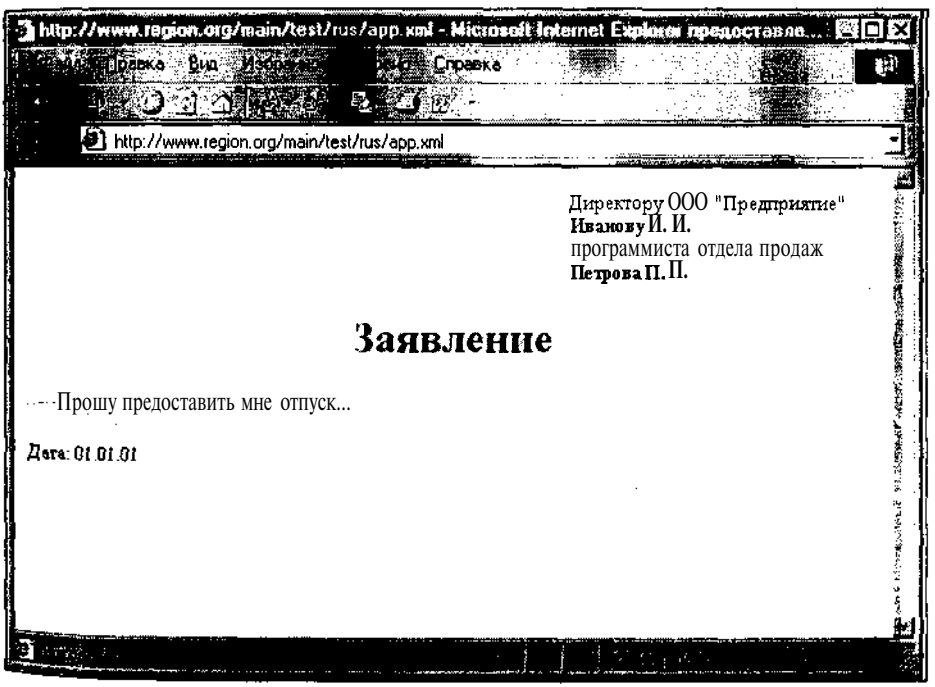


Рис. 2.1. Внешний вид документа "Заявление"

Беглый просмотр листинга 2.1 показывает, что в нем содержатся имена всех элементов из XML-файла. Это говорит о том, что для каждого элемента заданы инструкции его преобразования в конструкции XHTML. Но если для какого-либо документа такие инструкции были бы пропущены, то документ, представляющий сценарий XSLT, не потерял бы своей корректности.

Листинг 2.1. Сценарий application.xsl

```
<?xml version="1.0" encoding="koi8-r"?>

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

<xsl:template match="application">
 <xsl:processing-instruction name="cocoon-format">
type="text/html"</xsl:processing-instruction>
 <html>
  <body>
   <xsl:apply-templates/>
  </body>
 </html>
</xsl:template>

<xsl:template match="description">
 <div style="text-align:right;">
  <div style="text-align:left;width:200px">
   <xsl:apply-templates/>
  </div>
 </div>
</xsl:template>

<xsl:template match="for|from">
 <b>
  <xsl:apply-templates/>
 </b>
 <br/>
</xsl:template>

<xsl:template match="title">
 <h1 style="text-align:center">
  <xsl:apply-templates/>
 </h1>
</xsl:template>

<xsl:template match="body">
 <p style="text-indent:1.5em">
  <xsl:apply-templates/>
 </p>
</xsl:template>

<xsl:template match="date">
Дата: 01.01.01
</xsl:template>

</xsl:stylesheet>
```

Для полной работоспособности пары файлов, представленных листингами 1.1 и 2.1, в XML-файл необходимо вставить одну из инструкций обработки, описанных в конце *разд. 1.5:*

```
<?cocoon-process type="xslt"?>
```

Ее можно вставить в промежуток между первой и второй строкой. Кроме того, для поддержки русского языка в заголовке каждого файла, содержащего русский текст, нужно указать соответствующую кодировку, определяемую атрибутом encoding, например, так:

```
<?xml version="1.0" encoding="koi8-r"?>
```

Теперь, если XML- и XSL-файлы (обозначенные здесь и далее по своим расширениям) набраны верно, в окне браузера можно будет увидеть документ, который изображен на рис. 2.1.

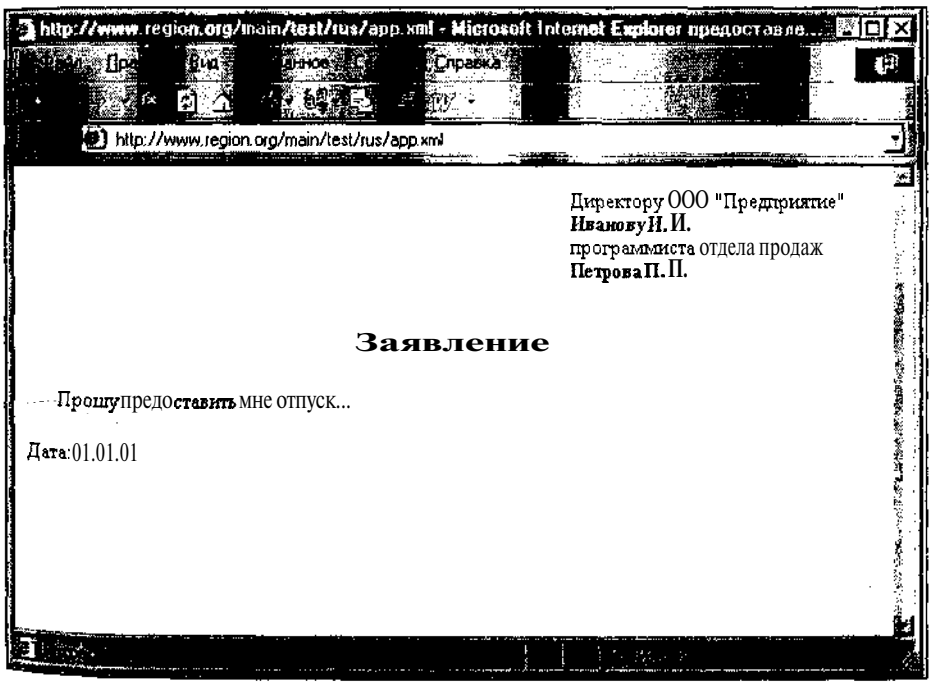


Рис. 2.1. Внешний вид документа "Заявление"

Беглый просмотр листинга 2.1 показывает, что в нем содержатся имена всех элементов из XML-файла. Это говорит о том, что для каждого элемента заданы инструкции его преобразования в конструкции XHTML. Но если для какого-либо документа такие инструкции были бы пропущены, то документ, представляющий сценарий XSLT, не потерял бы своей корректности.

Это означает, что результатом обработки документа был бы, пусть не совсем красиво выглядящий, но все же текст, а не сообщение об ошибке.

Процесс преобразования исходного дерева в конечное осуществляется путем сопоставления так называемых образцов и шаблонов. Образец сравнивается с элементами исходного дерева, и по правилам шаблона происходит преобразование в набор узлов конечного дерева. Причем исходное и конечное деревья полностью отделены друг от друга.

Просмотр элементов исходного дерева направлен сверху вниз, от корня к листьям. Это свойство, а также эффективно организованная древовидная форма структуры данных позволяют разбивать описание представления документа на части и редактировать каждую из них по отдельности. Помимо редактирования у разработчика есть возможность использовать механизм взаимодействия этих частей. Это означает, что из любого места (с учетом сохранения корректности) сценария XSLT можно "вызвать" некоторый где-либо описанный набор правил и применить к текущему узлу. Такая модульность и возможность организации рекурсии добавляют значительную гибкость стилевым таблицам.

Кроме того, при описании правил трансформации для какого-либо узла, существует возможность указать также некоторые стороны представления для его потомков. Можно, например, объявить их элементами списка или таблицы и среди правил для текущего узла описать их окружение (формат списка, некоторые дополнительные, создаваемые на этапе разбора столбцы или строки таблицы).

Но обо всем по порядку.

2.2. Пространство имен

Все элементы языка XSLT принадлежат зарезервированному пространству имен:

```
http://www.w3.org/1999/XSL/Transform
```

и имеют префикс `xsl:`, который не используется по умолчанию (то есть для всех элементов данного пространства имен должны быть заданы полные имена). Здесь первая часть URL представляет собой ссылку на сайт W3 консорциума, каталог 1999 — год, когда была разработана спецификация. Способ организации иерархии каталогов, следующих далее, подчеркивает вложенность XSL Transformations в XSL.

Это пространство имен действует только в сценарии XSLT и не распространяется на XML-файл, представляющий собой исходное дерево документа.

Согласно спецификации XSLT, имена всех элементов таблиц стилей пишутся строчными буквами, в качестве разделителя используется дефис:

```
<xsl:apply-templates/>
```

2.3. Структура таблицы стиля

Сценарий XSLT должен соответствовать правилам синтаксиса XML, т. к. он сам представляет собой XML-документ. И, как и любой документ, должен содержать заголовок. В нашем случае (см. листинг 2.1) это

```
<?xml version="1.0" encoding="koi8-r"?>
```

В качестве элемента документа (корневого элемента) в стилевых таблицах используется один из двух вариантов: `xsl:stylesheet` или `xsl:transform`, которые являются синонимами и, соответственно, обладают одинаковым набором атрибутов. Каждый из этих элементов мы будем в дальнейшем называть элементом стиля. Приведем примеры с элементом первого типа:

```
<xsl:stylesheet
version = number
id = ID
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
extension-element-prefixes = tokens
exclude-result-prefixes = tokens>
<!-- Content -->
</xsl:stylesheet>
```

и второго:

```
<xsl: transform
version = number
id = ID
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
extension-element-prefixes = tokens
exclude-result-prefixes = tokens>
<!-- Content -->
</xsl:transform>
```

Здесь number, id и tokens — типы данных, являющихся значениями соответствующих атрибутов. Таким образом, эти атрибуты означают следующее:

- version — обязательный номер версии (на данный момент 1.0);
- id — уникальный идентификатор элемента. Уникальность гарантирует соответствующий тип атрибута (ID);
- `xmlns:xsl` — служит для указания пространства имен. В коде указан явно;
- extension-element-prefixes — служит для перечисления через пробел префиксов пространств имен, которые рассматриваются как так называемые пространства имен расширений. Расширения (мы не будем подробно на них останавливаться) бывают двух типов: элементы расширения и функции расширения. Они предназначены для того, чтобы указать процессору, как он должен реагировать, если ему встретятся префиксы пространства имен, определенные разработчиками или пользователями синтаксического анализатора;
- `exclude-result-prefixes` — содержит список префиксов пространств имен, разделенных пробелами, которые определяют элементы, не включаемые в дерево результирующего документа. Здесь имеются в виду также пространства имен, определенные разработчиками или пользователями синтаксического анализатора или, возможно, те, которые будут определены в последующих версиях XSLT.

Примечание

Атрибуты `extension-element-prefixes` И `exclude-result-prefixes` служат для весьма специфических целей, далее в наших листингах использоваться не будут, поэтому их описание мы приводить не станем.

Непосредственными потомками элемента стиля могут быть следующие элементы: `xsl:import`, `xsl:include`, `xsl:strip-space`, `xsl:preserve-space`, `xsl:output`, `xsl:key`, `xsl:decimal-format`, `xsl:namespace-alias`, `xsl:attribute-set`, `xsl:variable`, `xsl:param` И `xsl:template`. Остальные элементы могут встречаться только в содержимом вышеперечисленных элементов, которые называются элементами верхнего уровня.

2.4. Включение и импорт таблиц стилей

Комбинирование таблиц стилей не только определяет некоторую парадигму программирования сценариев XSLT, но и позволяет полностью отделить друг от друга использование XML- и XSL-составляющих документа. С одной стороны, один и тот же XSL-файл, будучи однажды созданным, может быть использован с различными XML-файлами. С другой, один XML-файл может быть обработан как с помощью некоторой совокупности XSL-файлов, так и путем их выборочного подключения.

Например, при обработке интернационального XML-документа, мы можем для его преобразования создать набор таблиц стилей, каждая из которых соответствует определенному языку. И при запросе документа, к документу будет применен набор правил, соответствующий языку пользователя, который этот документ запрашивает. Выбор языка пользователем может быть осуществлен, например, с помощью щелчка мышью по специальной гиперссылке. Таким образом, информация о языке может быть доставлена XML-документу среди параметров HTTP-запроса. Подробнее об этом можно прочитать в *главах 6* и *8*.

Набор правил XSLT может располагаться в нескольких файлах. При работе с большими проектами бывает удобно разбить всю совокупность кода на несколько составляющих. Такая модульность придает гибкость не только использованию сценариев XSLT, но и их разработке. Создавая правила преобразования для каждого элемента из файла XML, мы можем, во-первых, разбить эти правила на тематические группы и редактировать каждую из них по отдельности. Во-вторых, используя механизм включения и/или импортирования стилей, можно конструировать сценарий поэтапно. Сначала мы можем создать несколько простейших правил шаблона, затем, взяв их за основу, построить более сложные правила и т. д.

2.4.1. Включение стилевых таблиц

Включение (include) таблицы стилей в сценарий представляет собой механизм, принцип работы которого во многом напоминает внешние сущности. Для его реализации служит элемент `xsl:include`:

```
<xsl:include href = URIресурса />
```

Здесь атрибут href содержит в себе относительную ссылку на включаемую таблицу стилей.

Во время обработки документа этот URI преобразуется в абсолютный URI ресурса с использованием так называемого базового URI. Базовый URI связан с каждым узлом дерева XML-документа. Если такой узел встречается где-либо в документе, то его базовый URI представляет собой URI самого Документа, если же он находится во внешней сущности, то представляет собой URI этой сущности.

Ресурс, на который указывает атрибут href, обрабатывается как XML-Документ, и все непосредственные потомки элемента `xsl:stylesheet` из включаемой таблицы стилей замещают соответствующий элемент `xsl:include`. Проводя аналогию с внешними сущностями, можно сказать, что элемент `xsl:include` подобен ссылке на сущность, а включаемая таблица стилей — на содержимое этой сущности.

Но если ссылка на сущность может присутствовать где угодно в теле документа и встречаться произвольное количество раз, то элемент xsl: include может быть только элементом верхнего уровня *(см. разд. 2.3)*. Кроме того, каждая таблица стилей может быть включена в документ только один раз. иначе из-за дублирования деклараций может возникнуть ошибка. Причем многократное включение одной и той же таблицы стилей может быть как прямым, так и косвенным, которое труднее отследить. Вариант с многократным косвенным включением может возникнуть, например, в следующем случае: ssl.xsl включает ss2.xsl и ss3.xsl, а ss2.xsl и ss3.xsl включают ss4.xsl. Таким образом, ss4.xsl включается в ssl.xsl два раза.

Этой ситуации можно избежать, если перегруппировать правила трансформации в этих файлах так, чтобы каждое из них встречалось только один раз, Заметим здесь, что ошибка такого рода встречается на практике не очень часто, т. к. структура последовательности включений, как правило, достаточно проста, и поэтому все декларации легко разбить на тематические группы. Еще реже встречается ситуация, когда стилевая таблица прямо или косвенно включает сама себя, что также является ошибкой.

2.4.2. Импорт стилевых таблиц

Механизм импорта (import) стилевых таблиц обладает несколько большей сложностью, чем включение. Он осуществляется с помощью синтаксиса, аналогичного включению, в котором слово include заменено на import:

```
<xsl:import href = URI ресурса />
```

Но несмотря на схожесть синтаксисов, на этом отличия включения и импорта не исчерпываются. И отметив здесь, что атрибут href обладает теми же функциями, а элемент импортирования xsl:import может быть так же, как и xsl:include, только элементом верхнего уровня, обратим внимание на роль местоположения xsl:import в документе. Тот факт, что результат импорта зависит от места, где располагается элемент импортирования, хорошо иллюстрирует принципиальную разницу между этими двумя способами комбинирования стилевых таблиц.

Дело в том, что с импортом тесно связано понятие приоритета таблиц стилей. И если во время включения при дублировании деклараций возникает ошибка, то при импорте обрабатывается та декларация, которая имеет более высокий приоритет. Правило определения приоритета для импортированных стилевых таблиц можно сформулировать следующим образом.

При импортировании стилевых таблиц строится так называемое дерево импорта, корневым элементом которого является импортирующая таблица, элементами второго (следующего) уровня дерева — таблицы, импортированные в таблицу, представляющую корневой узел дерева импорта, третьего уровня — эле-

менты, импортированные в таблицы, представляющие узлы второго уровня дерева и т. д. Приоритет импортируемой таблицы стилей определяется номером уровня узла дерева импорта. Самым высоким приоритетом обладает исходная таблица, которую никто не импортирует и которая представляет собой корень дерева импорта или его узел первого уровня.

Здесь может возникнуть вопрос: а как же определяется приоритет для таблиц одного уровня? И также может возникнуть ответ: "в зависимости от того где в документе располагается элемент `xsl:import`, например, если элемент импортирования одной стилевой таблицы встречается ранее элемента импортирования другой стилевой таблицы, то первая обладает большим приоритетом". Но на самом деле это не так.

На этот счет спецификация XSLT провозглашает, что механизм обработки многократного импортирования в ней не описывается, т. е. оставляется на усмотрение разработчика синтаксического анализатора XML. Таким образом, если в документе возникнет конфликт между декларациями одного и того же приоритета (или, иными словами, стилевыми таблицами, представленными узлами дерева одного и того же уровня дерева импорта), то не исключено, что эта ситуация сможет вызвать у синтаксического анализатора состояние аффекта.

По этой причине, а также в силу того, что механизм импорта стилевых таблиц работает, как правило, гораздо медленнее включения, мы не рекомендуем его использовать. Предлагаем ограничиться лишь включением. Заметим, что, несмотря на достаточный объем кода, реализующий поставленные в книге задачи, нам ни разу не понадобилось прибегать ни к импорту, ни к включению. Хотя вполне возможно, что и зря.

Заканчивая раздел, перечислим не указанные ранее ограничения, накладываемые на импорт стилевых таблиц.

- Точно так же, как и в случае включения, спецификация запрещает прямое или косвенное импортирование стилевой таблицы самой в себя.
- Элемент `xsl:import` должен располагаться перед элементами `xsl:include`. Кроме того, здесь спецификация рекомендует следующее: при вставке элемента импорта его следует располагать до всех элементов `xsl:include`, а также ПОСЛе Всех ИМеЮЩИХСЯ `xsl:import`.

Заметим здесь также, что элементы включения обрабатываются раньше, чем элементы импорта.

2.5. Выражения. Язык XPath

Так как язык XSLT используется для преобразования одного XML-документа в другой XML-документ, которые представлены соответственно Исходным и результирующим деревьями, то в нем необходимо одно-

значно указывать, какой именно элемент исходного дерева трансформируется в соответствующее поддерево конечного дерева. При этом для указания (отбора) узлов исходного дерева служит язык XPath, представляющий собой некоторую систему адресации узлов. Этому языку посвящена *глава 3* книги. Расположение же результирующих поддеревьев в конечном дереве зависит от типа и взаимного расположения правил трансформации в сценарии XSLT.

Всего существует семь типов узлов как исходного, так и конечного деревьев (в силу того, что оба дерева представляют документы, обладающие синтаксисом XML):

- **корень** — является непосредственным предком (родителем) для корневого элемента (или по-другому, элементом документа, который представляет тело документа) и для инструкций обработки, комментариев, а также текстовых узлов, состоящих из одних пробельных символов *(см. разд. 2.17.1)*. Корень не имеет предков и содержит весь контент документа кроме пролога. Он представляет собой только обозначение, и правила трансформации для него не могут быть представлены в сценарии XSLT;

- **элемент** — узел, который может быть предком для элементов, представляющих любые типы узлов, кроме корня. Для элемента могут быть заданы правила трансформации;

- **атрибут** — узлы этого типа являются листьями дерева, т. е. не могут иметь потомков. Значение атрибута является его частью, но не потомком. Для атрибута правила трансформации могут быть определены среди правил, заданных для его родительского элемента, и не могут присутствовать в сценарии непосредственно;

- **текст** — для узлов этого типа будет верным все то, что касается атрибутов, кроме, естественно, той части, которая связана со значением атрибута. Если в тексте присутствует, например, комментарий, то такой контент разделяется на три узла: текст до комментария, комментарий и текст после комментария. Все эти три узла будут иметь одного родителя, причем такого, который имел бы текст без этого комментария;

- **комментарий** — этот узел не содержит потомков и не трансформируется никакими правилами, т. к. игнорируется процессором;

- **инструкция обработки** — также не имеет потомков и ни во что не может быть трансформирована, поскольку является указанием сделать что-либо, имеющим смысл только для конкретных реализаций процессора;

- **пространство имен** — является отдельным типом узла только потому, что оказывает воздействие на остальной документ. Не трансформируется

непосредственно, но как и инструкции обработки может повлиять на трансформацию узлов остальных типов.

Помимо отбора узлов в исходном дереве язык XPath может служить для определения способа, которым должен быть обработан какой-либо узел. Иначе говоря, он применяется для описания некоторых условий, в зависимости от выполнения которых выбирается тот или иной способ трансформации узла.

третьей целью, для которой может служить язык XPath, является генерирование текста для подстановки в конечное дерево. Иногда существует необходимость вставить в текст некоторую служебную информацию. Например, может потребоваться пронумеровать узлы, являющиеся соседними, или некоторую их часть. Используя XPath, мы легко можем получить номер текущего узла в списке его непосредственных соседей (sibling), как-либо его отформатировать и затем с помощью специальной директивы XSLT вставить в конечный документ. Разумеется, средствами XPath можно сгенерировать и текст, не содержащий служебной информации, например, присвоить атрибуту конечного дерева преобразованное или оставленное без изменений значение атрибута исходного дерева.

Важно запомнить, что язык XPath не влияет непосредственно на трансформацию документа. Он представляет собой только лишь средство адресации или способ описания чего-либо. Когда мы говорили, например, об отборе узлов, мы имели в виду, что в данном случае конструкции XPath только описывают эти узлы, задавая их местоположение в исходном дереве. Сам отбор как действие осуществляет какая-либо директива XSLT, после чего она совершает над этим набором те операции, для которых предназначена. Это означает, что выражения XPath не могут появляться непосредственно где-либо в документе, но могут выступать как параметры или часть некоторого параметра для директив XSL или каких-то других средств обработки XML-документов. Кроме того, именно в этом заключается отличие языка адресации XPath от каких-либо языков запросов.

2.6. Правила шаблона

Понятия "шаблон" и "правило шаблона" являются ключевыми при написании сценариев XSLT. Рассматривая простой файл, представляющий такой сценарий, в первом приближении, можно заметить, что он состоит из заголовка и элемента стилевой таблицы (`xsl:stylesheet`). Содержимое этого элемента можно разбить на две части. Первая, если присутствует, состоит из элементов включения и импортирования таблиц стилей, вторая — из правил шаблона.

Можно сказать, что большая часть разработки сценария XSLT заключается в настройке элементов, представляющих эти правила.

Преобразование исходного дерева документа в конечное происходит с помощью повторения (количество повторений равно числу узлов исходного дерева) операции, состоящей из трех этапов: сначала в исходном дереве определяется набор узлов, соответствующий некоторому заданному образцу (pattern), затем среди этих узлов по некоторым критериям выбирается один узел, далее к нему применяется правило шаблона. В языке XSLT правило шаблона представляет элемент xsl: template, рассмотрим его синтаксис:

```
<xsl:template
   math = образец
   name = имя
   priority = число
   mode = режим>
<!-- шаблон -->
</xsl:template>
```

Как видно, значением первого атрибута является образец, на основе которого в исходном дереве происходит отбор узлов, которые трансформируются с помощью шаблона, представляющего собой контент элемент; `xsl:template`.

Что же такое образец? Правила написания образцов являются некоторым подмножеством синтаксиса XPath. Рассмотрим табл. 2.1, содержащую некоторые примеры образцов, и для каждого из них укажем, какие узлы исходного дерева с их помощью идентифицирует правило шаблона.

Таблица 2.1. *Примеры образцов*

Образец	Идентифицируемые узлы
Para	Любой элемент (все элементы) с именем `para`
*	Любой элемент (все элементы)
chapter I appendix	Элементы с именами chapter и appendix
`olist/item`	Элемент item, чьим родителем является `olist`
`appendix//para`	**Элемент** para, **чьим предком является** appendix
/	Корень

Таблица 2.1 *(окончание)*

Образец	Идентифицируемые узлы
text()	Любой текстовый узел
processing-instruction()	Любая инструкция обработки
node()	Любой узел, кроме атрибута и корня
para[1]	Элемент para, являющийся для своего родителя первым встретившимся непосредственным потомком (child) с именем para
*[position()=1 and self::para]	Элемент para, являющийся для своего родителя первым непосредственным потомком (child) с именем para
para [last 0=1]	Любой элемент para, являющийся для своего родителя единственным непосредственным потомком
items/item[position() > 1]	Элемент item, имеющий родителя items, но не являющийся для него первым непосредственным потомком
item[position() mod 2 = 1]	Все нечетные (в нумерации непосредственных потомков родителя элемента item) элементы item
div[@class="appendix"]	Элемент div, обладающий атрибутом class, значение которого равно appendix
@class	Атрибут class
@*	Любой атрибут

Например, правило шаблона, записанное следующим образом:

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

будет применено к элементу description, а следующее правило

```
<xsl:template match = "for|from">
...
</xsl:template>
```

будет применено к обоим элементам — for и from.

Здесь мы только обозначим примерную схему работы образцов. Подробное ее рассмотрение мы отложим до третьей главы, целиком посвященной синтаксису языка XPath, без знания которого полное понимание сути образцов невозможно.

Прежде чем перейти к описанию того, каким может быть сам шаблон, рассмотрим второй этап преобразования. Этот этап представляет собой некоторые действия, направленные на разрешение конфликтов между правилами шаблона. Или, иными словами, на этом этапе не всегда однозначно находится ответ на вопрос: какое из правил шаблонов должно работать в том случае, если наборы узлов, определяемые их образцами, содержат общие узлы?

2.6.1. Разрешение конфликтов между правилами

Критерий предпочтения одного правила шаблона другим строится на основе так называемого приоритета правила шаблона. Этот приоритет либо вычисляется процессором, либо указывается явным образом с помощью атрибута priority. Его значение представляет собой действительное (с плавающей точкой) 64-битное число двойной точности.

Примечание

Действительные числа в XPath могут содержать несколько специальных значений: Not-a-Number (NaN), положительная и отрицательная бесконечности, а также положительный и отрицательный нули.

Ниже приведены шаги алгоритма выбора правила шаблона, которое должно быть применено к рассматриваемому узлу.

1. Сначала из рассмотрения исключаются правила шаблона с более низким приоритетом импорта *(см. разд. 2.4)*. Правило шаблона, содержащееся в импортируемой стилевой таблице с более высоким приоритетом импорта, предпочитается правилу шаблона из таблицы стилей, обладающей более низким приоритетом.
2. Далее отсеиваются правила шаблона с меньшим значением атрибута priority. Если значение этого атрибута не указано явным образом, то в его качестве выступает значение по умолчанию. При вычислении этого значения более высокий приоритет присваивается тем правилам шаблона, для которых заданный образец определен более конкретно. Правила шаблона с более общим образцом имеют менее высокий приоритет. Формально значение приоритета по умолчанию вычисляется следующим образом:
   - Если образец содержит несколько альтернатив, разделенных вертикальной чертой (|), то все эти альтернативы считаются имеющим

один и тот же приоритет, как если бы для каждой альтернативы существовало отдельное правило шаблона.

- Если образец описывает узел в виде полного имени *(см. разд. 1.7)*, т. е. определенного типа и с определенным расширенным именем, то приоритет соответствующего правила шаблона равен 0.
- Если образец описывает узел в виде неполного имени, т. е. определенного типа и с расширенным именем, из пространства имен с определенным URI, то приоритет соответствующего правила шаблона равен -0.25.
- Правила шаблона, чьи образцы описывают просто узлы определенного типа, имеют приоритет −0.5.
- Остальные, более специфические образцы определяют приоритет для своих правил шаблонов, равный 0.5.

Если после применения этого алгоритма для какого-либо узла остались конфликтующие правила шаблона, то, согласно спецификации XSLT, процессор обязан либо сгенерировать ошибку, либо разрешить ситуацию по своему усмотрению, т. е. расширить вышеприведенный алгоритм.

2.6.2. Применение правил шаблонов

После того, как для данного узла в соответствии с образцом выбрано правило шаблона, начинается третий этап обработки этого узла — в результирующем дереве для него по шаблону создается соответствующее поддерево. Рассмотрим пример работы правила шаблона:

```
<xsl:template match="title">
 <h1 style="text-align:center">
  <xsl:apply-templates/>
 </h1>
</xsl:template>
```

Для узла, представленного элементом с именем `title`, в конечном дереве будет сформировано следующее поддерево

```
<h1 style="text-align:center">
...Здесь находится контент элемента title
</h1>
```

Таким образом, каждое вхождение элемента `title` будет заменено заголовком первого уровня, текст в котором отцентрирован. Заметим здесь, что такая трансформация произойдет только в том случае, если правило шаблона для `title` не будет конфликтовать с таким же правилом шаблона, обладающим более высоким приоритетом.

В этом примере мы употребили новый элемент `xsl:apply-templates`, который служит здесь для указания процессору применить к непосредственным потомкам элемента `title` соответствующие правила шаблона. Вообще говоря, синтаксис `xsl:apply-templates` предоставляет возможности для более широкого использования этого элемента:

```
<xsl:apply-templates
   select= адрес набора узлов
   mode= режим />
```

Атрибут `select` этого элемента с помощью выражения XPath позволяет указать, к каким именно узлам исходного дерева должны быть применены соответствующие правила шаблона. В качестве значения этого атрибута могут выступать образцы из табл. 2.1, например

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

или

```
<xsl:apply-templates select= "para[last()=1]" />
```

Можно сказать, что наличие атрибута `select` дает возможность "вызвать" из одного правила шаблона другое правило. Причем делать это можно неоднократно:

```
<xsl:template match="title">
 <h1 style="text-align:center">
  <xsl:apply-templates/><br/>
  <xsl:apply-templates/><br/>
  <xsl:apply-templates/><br/>
 </h1>
</xsl:template>
```

Такой шаблон трижды вызывает правило шаблона для детей элемента `title`. И если предположить, что контентом `title` является, например, слово "Документ", то в результате мы получим следующий XHTML-код:

```
<h1 style="text-align:center">
  Документ<br/>
  Документ<br/>
  Документ<br/>
 </h1>
```

Не следует забывать, что при трансформации одного дерева в другое древовидные модели как исходного, так и конечного документа должны удовлетворять спецификации XML. Это означает, в частности, что у результирующего дерева должен быть только один корневой элемент. Поэтому сле-

дующий код, представляющий собой правило шаблона для корневого элемента исходного дерева, вызовет ошибку:

```
<xsl:templateroatch="root-element">
  <p> Некоторый текст </p>
  <xsl:apply-templates/><br/>
</xsl:template>
```

Ошибка возникнет в силу того, что у результирующего поддерева в данном случае будет, как минимум, три корневых элемента: элемент p, один или несколько элементов, полученных в результате применения правил шаблона, вызванных директивой `<xsl:apply-templates/>`, а также элемент `br`.

Для того чтобы избежать этой ошибки, необходимо все элементы, принимающие непосредственное участие в трансформации (исключением может быть, например, инструкция обработки) и образующие шаблон для преобразования корневого элемента, сделать контентом для какого-либо другого единственного элемента. Правильным будет, например, следующий вариант:

```
<xsl:template match="root-element">
 <html>
  <p> Некоторый текст </p>
  <xsl: apply-templates/xbr/>
 </html>
</xsl:template>
```

Напомним, что такая ситуация может возникнуть только при создании правила шаблона для корневого элемента.

Еще одной возможной ошибкой может быть явная или неявная организация рекурсии. Следующий код вызовет бесконечный цикл:

```
<xsl:template match="some-element">
  <xsl:apply-templates select="."/>
</xsl:template>
```

поскольку здесь осуществляется вызов правила шаблона из самого себя. И если такие явные циклы процессор теоретически может отследить, то более сложные (такие, например, когда ряд правил, начиная с первого, последовательно вызывают друг друга, а последнее вызывает первое) могут вызвать ошибку типа "отказ в обслуживании". Единственным средством против такой ошибки является только внимательность разработчика.

2.6.3. Переопределение правил шаблона

Если при импортировании между правилами шаблона возникают конфликты, то они разрешаются так, как было описано ранее. При этом обрабатывается то правило шаблона, для которого был вычислен максимальный приоритет импорта. Но это не означает, что остальные правила шаблона, участвовавшие в конфликтной ситуации, отбрасываются процессором. Если рассмотреть два правила шаблона для одного и того же узла с разными приоритетами, то можно сказать, что тот из них, который обладает наивысшим приоритетом, т. е. находится в текущем документе, переопределяет то правило шаблона, которое импортируется.

Существует возможность применить импортируемое правило шаблона совместно с переопределенным. Это можно сделать с помощью директивы

```
<xsl:apply-imports/>
```

При обработке стилевой таблицы в каждой ее точке определяется текущее правило шаблона. Исключение составляет тот случай, когда обрабатывается контент директивы `xsl:for-each`, при этом правило шаблона является нулевым.

Директива `xsl:apply-imports` используется для обработки текущего узла, применяя к ним импортированные (в текущее правило) правила шаблона. Если при этом текущее правило шаблона является нулевым, возникает ошибка.

Предположим, что стилевая таблица import.xsl содержит следующее правило

```
<xsl:template match="para">
   <i>
     <xsl:apply-templates/>
   </i>
</xsl:template>
```

Тогда при обработке другой стилевой таблицы current.xsl, импортирующей первую,

```
<xsl:import="import.xsl"/>

<xsl:template match="para">
   <P>
     <xsl:apply-imports/>
   </p>
</xsl:template>
```

будет сгенерирован следующий HTML-код:

```
<p> <i>... </i> </p>
```

2.6.4. Встроенные правила шаблона

До этого момента мы рассматривали только те ситуации, когда для каждого узла исходного дерева было задано одно или несколько правил шаблона. Но что произойдет, если для какого-либо узла такие правила не определены вовсе? Вообще говоря, такой случай не делает сценарий XSLT некорректным как XML-документ, т. е. такой сценарий не вызовет ошибки и будет специальным образом обработан. Для подобных узлов спецификацией XSLT предусмотрен ряд умолчаний, которые зависят от типа этого узла. Ниже следует список правил, действующих по умолчанию, которые еще называются встроенными.

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

Это правило шаблона выполняется для элемента документа, а также для любого другого элемента. Иными словами, если для корневого или любого другого элемента не задано правило шаблона, то сработает вышеприведенное правило, которое вызовет обработку всех элементов исходного дерева. Таким образом, для каждого элемента с помощью директивы `<xsl:apply-templates/>` будет применено соответствующее правило шаблона.

```
<xsl:template match="*|/" mode="m">
  <xsl:apply-templates/>
</xsl:template>
```

Этот код описывает точно такое же встроенное правило шаблона, но предназначенное для правил с режимом т. О том, что такое режим и для чего он нужен, можно узнать в следующем подразделе.

Также существует встроенное правило шаблона для атрибутов и текстовых узлов, которое просто копирует текст или, соответственно, значение атрибута:

```
<xsl:template math="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

О том, как работает директива `<xsl:value-of select = "..."/>`, далее в этой главе.

Для комментария и инструкций обработки процессор не должен делать ничего:

```
<xsl:template match="comment()|processing-instruction()"/>
```

Это правило шаблона является встроенным правилом шаблона для узлов Пространства имен, для которых не существует собственного встроенного правила постольку, поскольку не существует образца, с помощью которого эти узлы можно было бы адресовать.

Все встроенные правила шаблона имеют самый низкий приоритет — приоритет импорта, т. е. работают так, как если бы они были импортированы. Поэтому сначала выполняются все те правила шаблона, которые разработчик задал явным образом в сценарии XSLT.

2.6.5. Режимы

Мы не зря так много говорили о конфликтах, возникающих вследствие дублирования деклараций XSLT. Довольно часто возникают ситуации, когда к элементу с одним и тем же именем могут применяться различные правила шаблона. Например, элемент `title`, обозначающий заголовок, может по-разному интерпретироваться в зависимости от своего уровня. Он может быть заголовком части, главы или раздела документа. В таких случаях говорят, что трансформация элемента зависит от его контекста.

В подобных ситуациях можно поступить двумя способами. Первый заключается в том, чтобы использовать не один элемент, а несколько, т. е. для каждого способа трансформации определить отдельный элемент со своим собственным именем, например, `title1`, `title2`... или `part`, `chapter` и т. п. Но это не совсем удобно, т. к. способов трансформации может быть много и большое количество разноименных элементов приведет к путанице.

Гораздо удобнее второй вариант, когда в сценарии XSLT для каждого возможного контекста элемента создается собственное правило шаблона, что достигается грамотным конструированием его образца. В случае заголовка мы можем заметить, что каждый раздел содержится в какой-либо главе, а каждая глава, в свою очередь, принадлежит какой-либо части. Поэтому мы можем, пользуясь одним элементом `title` и предполагая, что элемент, соответствующий части (part), является непосредственным потомком некоего элемента book (это условие может быть и другим, нам оно необходимо для определенности), задать три правила шаблона:

```
<!-- правило шаблона для части -->
<xsl:template match="book/title">
...
</xsl:template>

<!-- правило шаблона для главы -->
<xsl:template match="book/title/title">
...
</xsl:template>

<!-- правило шаблона для раздела -->
<xsl:template match="book/title/title/title">
...
</xsl:template>
```

таким образом, мы можем использовать один элемент для заголовков любого уровня, не опасаясь возникновения конфликтной ситуации.

Но если имеется возможность определить разные правила шаблонов для различного контекста, то что делать, когда элементы отличаются не своим расположением, а своим содержимым? Предположим, что нам необходимо создать электронную форму, в которой пользователь должен указать либо свой телефон, либо адрес электронной почты, либо факс, либо какое-нибудь еще из десятка существующих или несуществующих средств связи. Разумеется, мы можем создать форму, содержащую поля, представляющие все эти способы связи, но ведь нам достаточно только одного. Теперь представим себе, что пользователь не заполняет поля формы, а непосредственно или с помощью чего-либо редактирует один XML-документ. Естественно, ему будет гораздо более удобно пользоваться одним элементом с именем, скажем, connection, нежели десятком (telephone, fax, e-mail и т. п.) или более других.

Таким образом, возникает вопрос: как описать правила шаблона, из которых в каждом случае применяется только один, в зависимости от структуры контента (ведь телефон и адрес e-mail записываются поразному)? Для решения этой задачи и предназначены так называемые режимы.

Режим определяется значением атрибута mode у элементов `xsl:template` и `xsl:apply-templates`:

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

<xsl:template match="connection" mode="telephone">
...
</xsl:template>

<xsl:template match="connection" mode="e-mail">
...
</xsl:template>
...

<xsl:template match="some_element">
  <xsl:apply-templates mode="mode">
</xsl:template>
```

В правиле шаблона, представленном образцом `some_element` и являющимся прямым предком элемента connection, мы можем с помощью некоторых директив XSLT проанализировать контент элемента connection и в зависимости от полученных результатов вызвать правило шаблона в том

или ином режиме. Кроме того, используя директиву xsl:apply-templates без указания шаблона, мы можем вызвать правило шаблона по умолчанию (не путать со встроенными правилами). Это первое правило в нашем сценарии. Правилом по умолчанию в данном контексте его делает отсутствие атрибута mode.

Заметим, что если some_element не является прямым предком элемента connection или если их родственные связи вовсе не определены, то для вызова правила шаблона в необходимом режиме можно воспользоваться атрибутом select **В директиве** xsl:apply-templates:

```
<xsl:apply-templates select="//connection" mode="mode">
```

В этом атрибуте нужно только указать доступный для определения в текущей ситуации образец.

Для значения режима существует одно вполне естественное ограничение: это значение может быть любым, но не совпадающим с другими значениями режима. Здесь следует также запомнить, что процессор, находясь в каком-либо режиме, не рассматривает правила без указания режима; вместо них он использует встроенные правила. Поэтому при использовании режимов необходимо для каждого элемента, который должен быть обработан, задать правило шаблона с соответствующим режимом. В этом смысле указание режима действует точно так же, как указание пространства имен по умолчанию. Мы можем хорошо проиллюстрировать эту ситуацию, если в листинге 2.1 укажем режимы для правила шаблона с образцом description И ДЛЯ директивы xsl:apply-templates В правиле шаблона для application. При этом можно указать режим как в одном месте, так и в обоих.

2.6.7. Именованные шаблоны

Рассмотрим ситуацию, когда сценарий XSLT содержит большое количество мало отличающихся друг от друга правил шаблона. Такое может случиться, если эти правила шаблона обладают некоторой общей частью. В языке программирования, будь то Java или C, мы могли бы выделить часто повторяющийся код и сформировать из него отдельную функцию. Затем, вместо многократного переписывания кода, мы могли бы просто вызывать эту функцию. В языке XSLT этим целям служат именованные правила шаблона или, для краткости, именованные шаблоны.

Для того чтобы присвоить правилу шаблона имя, служит атрибут name, принадлежащий элементу xsl:template. При этом использование атрибута match не исключается, но и не обязательно.

```
<xsl:template name="...">
...
</xsl:template>
```

Для вызова именованного шаблона также применяется атрибут name, но для директивы xsl:apply-templates:

```
<xsl:apply-templates name="..."/>
```

Кроме нее для вызова именованного шаблона может быть указана директива xsl:call-template, которая отличается от xsl:apply-templates тем, что не меняет текущий узел и текущий набор узлов. Это означает, что все директивы, находящиеся в именованном шаблоне, будут выполнены так, как будто они находились в точке вызова:

```
<xsl:call-template name="..."/>
```

Кроме того, атрибут name для директивы xsl:call-template обязателен.

Если сценарий XSLT содержит несколько правил шаблона с одинаковым именем и одним и тем же приоритетом импорта, то согласно спецификации процессор должен генерировать ошибку.

Проводя далее аналогию именованных шаблонов с функциями, заметим, что как именованный, так и неименованный шаблон может быть вызван с параметрами. Здесь понятие "параметр" полностью подобно параметру (аргументу) функции. Но поскольку перед использованием параметрического правила шаблона необходимо отдельно определить сами параметры, то об этом мы поговорим несколько позже, в *разд. 2.9.*

2.7. Внутренняя организация шаблона

Итак, мы подробно обсудили, что собой представляют правила шаблона и какие операции над ними мы можем проводить. Теперь зададимся вопросом, чем является содержимое правила шаблона, т. е. что может быть контентом элемента xsl:template, помимо директив вызова или применения других правил шаблона: xsl:apply-templates И xsl:call-template? Рассматриваемый контент как раз и называется шаблоном.

Хотя и трудно представить себе такую ситуацию, но для полноты картины скажем, что шаблон может быть пустым. В этом случае встроенное правило шаблона для соответствующего узла выполняться не будет, следовательно, поскольку для этого узла не определено никаких способов его трансформации, этот узел в результирующем дереве присутствовать не будет вовсе. Если Для следующего XML-кода:

```
<phrase>
```

```
Те узлы <tree>исходного</tree> дерева, для которых задано пустое правило
шаблона, в <tree>конечном</tree> дереве представлены не будут.

</phrase>
```

создать такие правила шаблона:

```
<xsl:template match = "phrase">
 <P>
  <xsl:apply-templates/>
 </p>
</xsl:template>


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

то результатом будет следующее представление:

```
Те узлы дерева, для которых задано пустое правило шаблона, в дереве пред-
ставлены не будут.
```

Пустые правила шаблона могут понадобиться для подавления трансформации узлов в каких-либо целях.

Следующим рассматриваемым нами видом шаблонов будут такие, которые состоят или из никак не размеченного текста, или текста в формате XHTML. Для XML-кода, состоящего только из одного элемента (кроме, разумеется, пролога документа, который мы опускаем):

```
<phrase/>
```

можно написать следующее правило шаблона:

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

Те узлы исходного дерева, для которых задано пустое правило шаблона, в
конечном дереве представлены не будут.

</xsl:template>
```

или, например, такое:

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

<p style="text-indent:1.5em">Те узлы <i>исходного</i> дерева, для которых
задано пустое правило шаблона, в <i>конечном</i> дереве представлены не
будут.</p>

</xsl:template>
```

Такие правила шаблона вполне допустимы, несмотря на то, что выглядят несколько бессмысленно. Они могут быть применены в тех случаях, когда в документе содержится слишком много обычного текста или гипер

текста, не имеющего какой-либо структурированной информации, подлежащей разметке с помощью XML. Такой контент для удобства можно разбить на отдельные части с тем, чтобы работать с каждой из них по отдельности. Результатом применения последнего правила шаблона будет такой текст:

```
Те узлы исходного дерева, для которых задано пустое правило шаблона, в конечном дереве представлены не будут.
```

Если бы всевозможные правила шаблона исчерпывались вышеперечисленными видами, то не было бы никакой нужды в использовании языка XML, и его преимущества перед HTML были бы весьма сомнительны. Далее мы будем делать упор на те шаблоны, которые помимо текста и HTML-разметки содержат еще и директивы XSLT. Наличие этих директив, как мы увидим далее, предоставляет широкий спектр операций над узлами как исходного, так и результирующего деревьев, что и составляет основу гибкости и функциональности языка XML и XSLT в частности.

2.8. Вычисление значения поддерева

Первой директивой, которую мы рассмотрим, будет одна из наиболее часто применяемых конструкций XSLT:

```
<xsl:value-of
 select = выражение XPath
 disable-output-escaping = "yes" | "no"/>
```

Эта директива выводит значение узла исходного дерева в конечный документ. Адрес узла, как видно из кода, определяется выражением XPath, являющимся значением атрибута select. Процессор находит поддерево, заданное этим адресом, вычисляет его значение, преобразует в строку (о том, как именно происходит это преобразование и о роли в нем атрибута disable-output-escaping *см. разд. 2.15.4)* и вставляет таким образом сгенерированный текст в конечное дерево как узел текстового типа. Вычисление значения поддерева происходит следующим образом.

Для элемента (в т. ч. и корневого) — в качестве исходного текста, выбирается объединение всех текстовых узлов элемента, затем в этот текст встраивается текст непосредственных потомков элементов, далее в получившийся текст встраивается текст потомков следующего уровня, и так повторяется до тех пор, пока все поддерево, корнем которого является узел, соответствующий рассматриваемому элементу, не будет разобрано. Рассмотрим следующий XML-код:

```
<root_element>
  <some_element>
   01 текст элемента 0-го уровня 01
    <first_levell>
     11 текст элемента 1-го уровня 11
    </first_levell>
   02 текст элемента 0-го уровня 02
    <first_level2>
     12 текст элемента 1-го уровня 12
     <second_level>
      2 текст элемента 2-го уровня 2
     </second_level>
     13 текст элемента 2-го уровня 13
    </first_level2>
   03 текст элемента 0-го уровня 03
  </some_element>
</root_element>
```

Текст, который будет сгенерирован для элемента some_element с помощью директивы

```
<xsl:value-of select = "some_element" />
```

будет выглядеть так:

```
01 текст элемента 0-го уровня 01
11 текст элемента 1-го уровня 11
02 текст элемента 0-го уровня 02
12 текст элемента 1-го уровня 12
2 текст элемента 2-го уровня 2
13 текст элемента 2-го уровня 13
03 текст элемента 0-го уровня 03
```

только без разрывов строк, которые мы поместили здесь для удобства просмотра.

Примечание

Если значением атрибута select является не один узел, а набор узлов (см. *главу 3*), то директива xsl:value-of сгенерирует текст только для первого узла из этого набора, а не для каждого из них.

Для атрибута будет сгенерирован текст, представляющий собой значение этого атрибута. Так директива

```
<xsl:value-of select = "@name"/>
```

для следующего XML-кода

```
<city name = "Moscow"/>
```

создаст текстовый узел с содержимым MOSCOW.

Результатом выполнения вычисления поддерева для текстового узла будет текстовый узел, являющийся непосредственным потомком для текущего элемента. То есть для

```
<xsl:template match = "some_element">
  <xsl:value-of select = "/some_element/text()"/>
</xsl:template>
```

результатом будет

01 текст элемента 0-го уровня 01

Для того чтобы **вывести следующий текстовый узел** (02 текст элемента 0-го уровня 02), **являющийся непосредственным потомком** `some_element`, нужно написать

```
<xsl:value-of select = "/some_element/text()[position()=2]"/>
```

Здесь `text()` является идентификатором типа узла, и точно так же для адресации остальных конструкций языка XML используются соответствующие идентификаторы типов (подробнее *см. главу 3*):

`comment()` — для адресации комментариев;

`processing-instruction()` — **инструкций обработки;**

`text()` — текстовых узлов;

`node()` — узлов произвольного типа.

В следующем примере:

```
<some_element>
  <another_element>
 ...
  </another_element>
  <!  комментарий 1-->
  <!  комментарий 2-->
</some_element>
```

адресом **первого комментария может быть** `some_element/comment()` ИЛИ `another_element/following-sibling::comment()`, **любой из которых** может быть образцом для `xsl:value-of`. Для адресации второго комментария

можно использовать some_element/comment() [position() =2] или просто some_element/comment()[2].

Результатом вычисления значения поддерева для таких узлов будет:

- для инструкций обработки — весь контент элемента, являющегося этой PI;
- для комментария — текст внутри комментария;
- для пространства имен — URI пространства имен.

Кроме того, для подобных и всех остальных узлов результат работы директивы xsl:value-of зависит от того, где эта директива располагается. Если она является частью шаблона для some_element из последнего примера, то может быть вычислено значение одного поддерева, если она принадлежит правилу шаблона для another_element, то — для другого. Это происходит вследствие того, что выражение XPath, являющееся значением для атрибута select, вычисляется относительно текущего узла (кроме тех случаев, когда это выражение представляет собой абсолютный адрес). Подробнее этот момент мы разберем в следующей главе.

С помощью директивы xsl:value-of можно вычислять не только значения поддеревьев, но и явно заданных арифметических выражений. Например, код

```
<xsl:value-of select = "2+3"/>
```

сгенерирует текст 5.

С помощью директивы xsl:value-of мы можем передать вычисленное значение поддерева в конечное дерево. Но как быть в том случае, когда мы хотим передать это значение некоторому атрибуту? В силу синтаксических ограничений языка XML мы не можем написать что-нибудь вроде:

```
<div id = "<xsl:value-of select = "@name"/>">
...
</div>
```

т. к. получается, что открывающий тег div содержит угловые скобки, что недопустимо. Для такой ситуации в XSLT предусмотрена конструкция, состоящая из фигурных скобок, которые действуют только при определении в результирующем дереве значений атрибутов. Верным будет следующий код:

```
<div id = "{@name}">
...
</div>
```

Фигурные скобки могут располагаться где угодно в значении атрибута. Таким образом, корректными будут следующие определения:

```
<div id = "_{@name}"/>
```

```
<div id = "_{@name1}_and_{@name2}_"/>
<div id = "{prefix}_{@name}_{postfix}"/>
```

Единственным ограничением на синтаксис фигурных скобок является то, что они не могут быть использованы рекурсивно. Вне зависимости от того, правильно вложены эти скобки друг в друга или нет, подобное их использование не разрешается.

2.9. Переменные и параметры

Естественным предназначением переменных и параметров в любом языке является их использование для обозначения часто повторяющихся данных. В XSLT у них есть несколько дополнительных функций. Мы будем их применять, в частности, для построения таких выражений XPath, которые без переменных не могут быть описаны вовсе. Кроме того, параметры служат для определения значений по умолчанию для именованных шаблонов, в чем, кстати, и состоит их единственное отличие от переменных.

Синтаксис определения и использования как переменных, так и параметров одинаков. Для переменной:

```
<xsl:variable
 name = имя
 select = выражение XPath>
<!-- Значение переменной -->
</xsl:variable>
```

Для параметра:

```
<xsl:param
 пате = имя
 select = выражение XPath>
<!  Значение переменной -->
</xsl:param>
```

Обе директивы `xsl:variable` и `xsl:param` имеют обязательный атрибут `name`, который задает имя соответственно переменной и параметра. Далее в разделе мы будем везде употреблять слово "переменная", подразумевая их аналогию с параметрами.

2.9.1. Типы переменных

Значение переменной может обладать одним из четырех типов, определенных в языке XPath: строковый, числовой, булево выражение и набор узлов. Значения этих типов могут быть заданы с помощью атрибута select:

```
<xsl:variable name = "name" select = "'some_value'"/>
<xsl:variable name = "name" select = "111"/>
<xsl:variable name = "name" select = "2&gt;1"/>
<xsl:variable name = "name" select = "some_element"/>
<xsl:variable name = "name" select = "'some_element'"/>
```

Для вывода значения переменной можно воспользоваться директивой xsl:value-of. При этом для обозначения последовательности символов как имени переменной служит символ $, который должен располагаться непосредственно перед этим именем. Если мы последовательно применим эту директиву к определенным выше переменным (<xsl:value-of select = "$name"/>), то получим следующий результат (см. табл. 2.2).

Таблица 2.2. *Типы значений переменной* name

Значение	Тип
'some_value'	**Строковый**
111	**Числовой**
True	Булево выражение
контент элемента some_elenient	Набор узлов
'some_element'	**Строковый**

Кроме вышеперечисленных типов переменных существует еще один тип данных, который называется фрагментом конечного дерева (result tree fragment). Значение этого типа переменная имеет в том случае, если значение определяется контентом элемента xsl:variable, а не присваивается через атрибут select. Таким образом, значения следующих двух переменных будут отличаться друг от друга:

```
<xsl:variable name="var1" select = "2"/>
<xsl:variable name="var2">2</xsl:variable>
```

Тип фрагмента конечного дерева представляет собой частный случай типа набор узлов. Он является набором узлов, содержащим не более одного узла, являющегося текстовым. Кроме того, к фрагменту конечного дерева можно применять только те операции, которые разрешены для строк. В частности, недопустимыми являются следующие операции: /, //, []. Поэтому, var2 можно использовать как переменную числового типа:

```
<xsl:value-of select = "item[$var1]"/>
```

а var2 нет:

```
<xsl:value-of select - "item[position O = $var2]"/>
```

Оба эти варианта являются верными, причем второй вариант будет верен и в том случае, если переменную var2 заменить var1.

Если значение переменной не задано ни с помощью атрибута select, ни с помощью контента элемента xsl:variable, то ее значение считается пустым. Следующие два определения эквивалентны:

```
<xsl:variable name = "var"/>
<xsl:variable name = "var" select = "''"/>
```

2.9.2. Область видимости

В XSLT не существует средств для переопределения переменных, поэтому их вернее было бы назвать константами. Но при этом одна переменная может маскировать другую точно так же, как в языке программирования локальная переменная маскирует глобальную. Элемент определения переменной может быть элементом верхнего уровня:

```
<xsl:variable name = "style" select = "top:20px;left:20px;"/>

<xsl:template match = "region">
 <div style = "$style">
  <xsl:apply-templates/>
 </div>
</xsl:template>
```

(в этом случае переменная является глобальной и видима всюду в сценарии XSLT после своего определения), а может содержаться в правиле шаблона:

```
<xsl:template match = "region">
 <xsl:variable name = "style" select = "top: 20px;left:20px;"/>
 <div style = "$style">
  <xsl:apply-templates/>
 </div>
</xsl:template>
```

При этом областью видимости переменной являются элементы, соседние (sibling) по отношению к элементу xsl:variable, кроме тех, которые расположены перед ним, а также все их потомки. Это означает, что в любом месте этой области (в контенте перечисленных элементов и их атрибутах) могут находиться ССЫЛКИ на ЭТУ переменную (*$var_name*).

Переменные могут маскировать друг друга в случаях, если:

- имена этих переменных совпадают;
- определение одной переменной находится в области видимости другой переменной;
- элементы определения переменных не имеют общего родителя (дублирование переменных).

Рассмотрим пример:

```
<xsl:variable name = "style" select = "top:10px; left:10px; "/>

<xsl:template match = "region">
 <xsl:variable name = "style" select = "top:20px; left:20px; "/>
 <div style = "$style">
  <xsl:apply-templates/>
 </div>
</xsl:template>
```

В этом примере в качестве стиля для div будет использоваться значение локальной переменной (top:20px;left:20px). Напомним, что переменные и параметры почти эквивалентны, поэтому в данном примере можно где-либо заменить xsl:variable на xsl:param. Также в силу схожести этих понятий следующий код вызовет ошибку:

```
<xsl:template match = "region">
 <xsl:variable name = "style" select = "top:20px; left:20px; "/>
 <xsl:param name = "style" select = "top:10px; left:10px; "/>
 <div style = "$style">
  <xsl:apply-templates/>
 </div>
</xsl:template>
```

Ошибка дублирования переменных возникает, вообще говоря, в тех случаях, когда определения одноименных переменных имеют одного и того же родителя и обладают одним и тем же приоритетом импорта.

Еще одной ошибкой, связанной с переменными, является их рекурсивное определение:

```
<xsl:variable name = "var">
  <xsl:value-of select = "$var"/>
</xsl:variable>
```

Таким образом, можно сформулировать правило: во избежание зацикливания ссылка на переменную может быть использована только после определения этой переменной. Точнее говоря, после закрывающего тега

</xsl:variable> или в том случае, если элемент xsl:variable является пустым, — после тега пустого элемента <xsl:variable.../>.

2.9.3. Параметрические правила шаблона

Как мы уже говорили, для правил шаблона можно задать некоторые значения по умолчанию — параметры, которые представляют собой аналогию аргументов функции в языках программирования. Для того чтобы задать параметры правила шаблона, нужно просто воспользоваться директивой xsl:param (не xsl:variable!) в контенте этого правила:

```
<xsl:template name = "func">
  <xsl:param name = "top">10px</xsl:param>
  <xsl:param name = "left">10px</xsl:param>
  <div style = "top:{$top};left:{$left}">
  ...
  </div>
</xsl:template>
```

Здесь мы определили два значения по умолчанию: отступ сверху (top) для некоторой области, равный 10px, и отступ слева (left) для этой области, равный также Юрх.

Для того чтобы вызвать правило шаблона с каким-либо параметром, служит директива

```
<xsl:with-param
  name = name
  select = выражение XPath>
<!  Новое значение параметра -->
</xsl:with-param>
```

которая используется ТОЧНО так же, как xsl:variable И xsl:param. Атрибут name указывает имя того параметра, значение которого мы переопределяем. Новое значение при этом задается либо с помощью атрибута select, либо как контент элемента xsl:with-param:

```
<xsl:call-template name = "func">
  <xsl:with-param name = "top" select = "30px" />
  <xsl:with-param name = "left" select = "40px" />
</xsl:call-template>
```

Таким образом, новыми значениями для параметров top и left будут соответственно зорх и 40px, и в результате вызова правила шаблона func характеристиками области, определяемой элементом div, будут именно эти, новые значения.

Для параметрического правила шаблона не будет ошибкой его вызов с параметром, который не определен в этом шаблоне с помощью элемента xsl:param.

2.10. Циклы

Если в документе встречается регулярное повторение одного и того же контента, то иногда полезно задать для него циклическую обработку. Для этих целей служит конструкция вида:

```
<xsl:for-each
  select = набор узлов для обработки />
  <!-- действия над каждым узлом из набора -->
</xsl:for-each>
```

С помощью обязательного атрибута select задается набор узлов, каждый из которых последовательно обрабатывается в соответствии с контентом элемента xsl:for-each. Результатом работы директивы является трансформация исходного набора в конечный, причем исходный набор может значительно отличаться от результирующего.

Циклы пригодны, например, для построения различных списков или таблиц. Предположим, у нас есть следующий XML-код:

```
<caption> </caption>
<table>
 <row>
  <flower> тюльпан </flower>
  <color> красный </color>
 </row>
 <row>
  <flower> фиалка </flower>
  <color> какой угодно </color>
 </row>
 <row>
  <flower> сирень </flower>
  <color> сиреневый </color>
 </row>
</table>
```

то после применения к нему следующих правил шаблона:

```
<xsl:template match = "caption">
  <h1> Цветы и цвета </h1>
</xsl:template>
```

```
<xsl:template match = "table">
  <table border="1">
   <tr> <td> № </td> <td> Цветок </td> <td> Цвет </td>
   <xsl:for-each select = "row">
    <tr><td> <xsl:value-of select = "position()"/> </td>
        <td> <xsl:apply-templates select = "flower"/> </td>
        <td> <xsl:apply-templates select = "color"/>  </td> </tr>
   </xsl:for-each>
  </table>
</xsl:template>
```

мы получим результат, представленный в табл. 2.3.

Таблица 2.3. Цветы и цвета

№	Цветок	Цвет
1	Тюльпан	Красный
2	Фиалка	Какой угодно
3	Сирень	Сиреневый

Все выражения XPath, являющиеся образцами для элементов в контенте `xsl:for-each`, кроме абсолютных, задаются относительно рассматриваемого в данный момент узла из набора узлов, заданного атрибутом select. В нашем сценарии это иллюстрируется тем, что мы пишем `<xsl:apply-templates select = "flower"/>` вместо `<xsl:apply-templates select = "row/flower"/>`, т. е. текущим является узел row, а не table.

Примечание

Написанный таким образом сценарий дает возможность редактировать исключительно XML-файл, без обращения к самому сценарию. Поэтому добавление новых пар "цветок-цвет" или их удаление вызовет автоматическое преобразование таблицы и, в частности, нумерации в ней.

2.11. Логические структуры

Логические структуры в языке XSLT служат для обработки контента с Условием. В зависимости от того, какое именно условие выполнено, выбирается та часть шаблона, которая будет участвовать в трансформации исходногодерева.

В XSLT существует всего два вида логических структур. Первый представляет собой реализацию простейшей обработки с условием "если-то". Если условие выполнено, то предпринимаются какие-либо действия, если нет, то

не происходит ничего. Конструкции этого типа представлены следующей директивой:

```
<xsl:if
  test = условие>
  <!-- контент, который участвует в трансформации при выполнении условия
-->
</xsl:if>
```

Здесь атрибут test определяет условие, которое задано на языке XPath и после разбора принимает значение true или false. В следующем примере первая страница некоего многостраничного доклада не нумеруется:

```
<xsl:template match = "page">
  <xsl:if test = "position() &gt; 1">
    <xsl:value-of select = "position()"/>
  </xsl:if>
</xsl:template>
```

Обработка с условием может находиться в теле какого-либо цикла. В этом случае директива xsl:if может быть применена или ко всем конструкциям XML (элементам, атрибутам и т. д.), перебираемым в этом цикле, или только к некоторым из них. Вот код, соответствующий первому варианту:

```
<xsl:for-each select = "">
  <xsl:if test = "">
  ...
  </xsl:if>
</xsl:for-each>
```

Но как правило, расположение директивы xsl:if в контенте xsl:for-each является излишним. Тот образец, который определяет набор узлов, последовательно перебираемых в цикле, может содержать и то дополнительное условие, которое определятся атрибутом test директивы xsl:... Иными словами, условия, представленные директивой цикла и директивой обработки с условием, можно переформулировать так, чтобы цикл перебирал только те узлы, которые удовлетворяют обоим этим условиям.

Но тем не менее, такая избыточность кода допустима в тех случаях, когда либо в цикле присутствуют какие-либо директивы помимо xsl:if, либо образец, представляющий собой объединение условий, слишком велик, и эти условия легче реализовать по отдельности. Здесь следует запомнить, что совмещение циклов и логических структур часто является предпосылкой к возникновению больших нагромождений кода, преследующих цель сформировать критерии отбора узлов для обработки. Этого можно избежать с помощью грамотного построения этих критериев, а именно с помощью пре-

дельной минимизации образцов, результат обработки которых имеет тип булева значения.

Второй вид обработки с условием представлен директивой xsl:choose, с помощью которой реализуется выбор одного из многих вариантов. Для отбора этих вариантов служат директивы xsl:when и xsl:otherwise, которые могут быть использованы только внутри контента xsl:choose:

```
<xsl:choose>
  <!-- контент, который может содержать только
       одну директиву xsl:when и
       одну или ни одной xsl:otherwise -->
</xsl:choose>
```

Синтаксис директив xsl:when и xsl:otherwise:

```
<xsl:when
  test = условие>
<!-- шаблон, который будет участвовать в
     трансформации в том случае, если выполнено
     условие, заданное атрибутом test -->
</xsl:when>

<xsl:otherwise>
<!-- шаблон, который будет применен по умолчанию,
     т. е. если не выполнится ни одно из условий,
     представленных в каких-либо xsl:when -->
</xsl:otherwise>
```

Директива xsl:choose может содержать только список различных вариантов, каждый из которых представлен директивой xsl:when или xsl:otherwise. Эти варианты последовательно просматриваются, и при выполнении условия, заданного атрибутом test у какого-либо из них, в трансформации принимает участие тот шаблон, который является контентом соответствующей директивы xsl:when. Если не выполняется ни одно из этих условий, то к исходному дереву применяется контент директивы xsl:otherwise.

В следующем примере определяется, какое слово для обозначения лет ("год", "года", "лет") должно быть использовано при указании возраста (для простоты от 21 до 30):

```
<xsl:value-of select = "$age"/>
<xsl:choose>
<xsl:when test = "$age = 21">
```

```
год
</xsl:when>
<xsl:when test = "$age &gt; 21 and $age &lt; 25 ] ">
года
</xsl:when>
<xsl:otherwise>
лет
</xsl:otherwise>
</xsl:choose>
```

Таким образом, если возраст, определяемый значением переменной age, будет равен 21, то результатом выполнения директивы xsl:choose будет текст "21 год", если возраст находится в промежутке от 22 до 24, то результатом будет соответственно "22 (23, 24) года", и для всех остальных случаев - "25 (26, 27, 28, 29) лет".

2.12. Нумерация и числа

Для представления чисел в языке XSLT существует директива xsl:number. С ее помощью можно выводить, однако, только натуральные числа, т. к. ее основная функция заключается в подсчете узлов. Кроме того, числа с плавающей точкой можно выводить с помощью директивы xsl:value-of. Директива xsl:number может быть также использована для того, чтобы указать формат и некоторые другие характеристики выводимых чисел:

```
<xsl:number
  value = числовое значение
  grouping-separator = разделитель для групп цифр
  grouping-size = количество цифр в одной группе
  format = формат вывода
  lang = язык символьной нумерации
  letter-value = языковая специфика символьной нумерации
  level = уровень нумерации
  count = элементы для нумерации
  from = адрес начала нумерации />
```

Если директива xsl:number применена для вывода конкретного числа, то его нужно указать с помощью атрибута value, который в других случаях не является обязательным. Но простой вывод числа можно указать с помощью директивы xsl:value-of в том случае, если это число представляет собой некоторую служебную информацию и вычисляется на этапе разбора. Таким числом может быть, например, номер какого-либо узла (position()) или количество узлов в некотором наборе (count()) и т. п. Если же число известно заранее, то нет не-

обходимости в использовании и xsl:value-of, т. к. мы можем вставить это число в XML-файл или сценарий XSLT непосредственно.

2.12.1. Вывод числа

В чем же заключаются преимущества директивы xsl:number? С ее помощью мы можем отделить друг от друга наборы цифр одинаковой длины, составляющие все это число в целом. При этом мы можем указать символ-разделитель для этих групп. Например, следующая конструкция

```
<xsl:number value = "100000000"
   grouping-separator = "."
   grouping-size = "4"/>
```

выведет число 100000000 в виде 1.0000.0000. Как следует из примера, для указания разделителя используется атрибут grouping-separator, а для указания размера отделяемых групп — grouping-size, который обычно равен трем. Эти два атрибута могут быть использованы только совместно. Если какой-нибудь из них опущен, то это не будет ошибкой, но второй атрибут будет проигнорирован.

2.12.2. Уровни нумерации

Как мы уже говорили, директива xsl:number применяется главным образом Для нумерации элементов. В этом варианте атрибут value не используется Вовсе и, как правило, атрибуты grouping-separator И grouping-size также не нужны, просто потому, что вряд ли в документе будет настолько большое количество нумеруемых узлов (хотя бы десятки или сотни тысяч), чтобы возникала потребность в разбиении цифр, составляющих число, на группы.

При нумерации указывается один или несколько из атрибутов: count, from и level. Атрибут count задает образец (pattern) для нумеруемых узлов. Если, например, нам необходимо пронумеровать главы и части книги, то его значением может быть "part | chapter". Атрибут from служит для указания того места, с которого начинается нумерация. Его значение представляет собой образец, адресующий корневой узел того поддерева, в котором проводится нумерация. Если мы хотим пронумеровать все главы в третьей части книги, то нам необходимо задать значения атрибутов count и from примерно так: from = "part[@number = '3']" И count = "chapter".

Атрибут level заслуживает отдельного рассмотрения. Он предназначен для того, чтобы определить, как будет зависеть нумерация узлов, заданных с помощью count, от их взаимного расположения в документе. При нумерации частей и глав книги мы рассматриваем элементы одного уровня. Это

означает, что элементы part являются братьями (sibling), элементы chapter также являются соседними внутри части. Поэтому если нас мало интересует формат вывода, для их нумерации можно воспользоваться конструкцией <xsl:value-of select = "position()"/>.

Но при этом такие две нумерации будут полностью совпадать, т. е. как первая часть, так и первая глава будут иметь номер 1. В некоторых случаях это недопустимо. Предположим, например, что в книге отсутствуют обозначения "часть" и "глава", вместо них используются только их названия. Поэтому, встретив в тексте ссылку на что-нибудь вроде "1. Разметка", мы не будем знать, какая из структурных единиц книги имеется в виду: часть, глава или что-то еще.

Атрибут level позволяет разрешить ситуацию, когда в документе имеется несколько слоев нумерации, причем нумерация одного уровня зависит от нумерации другого. Этот атрибут может принимать три значения: single, multiple и any, каждое из которых соответствует отдельному способу нумерации.

2.12.3. Значение *single*

Значение single используется в тех случаях, когда нумеруются узлы, находящиеся на одном уровне, и является значением по умолчанию. Рассмотрим алгоритм формирования номера узла при этом значении атрибута level.

1. Составляется список узлов, состоящий из всех предков текущего узла, а также самого этого узла. Другими словами, первым узлом в списке становится текущий узел, вторым — его родитель, третьим — родитель родителя и т. д. (в языке XPath такой набор узлов выражается обратной осью ancestor-or-self::current_element, где current_element — ИМЯ текущего узла).
2. В этом списке отбирается первый узел, соответствующий образцу count.
3. Подсчитывается количество узлов, которые по отношению к узлу, отобранному на шаге 2:
   - являются его братьями (sibling);
   - находятся перед этим узлом (preceding-sibling — предшествующие братья);
   - принадлежат поддереву, корневым узлом которого является узел, определяемый образцом, заданным необязательным атрибутом from.
4. Искомый номер узла получается прибавлением единицы, соответствующей самому этому узлу, к числу узлов, подсчитанных на третьем шаге (т. е. если на третьем шаге был получен 0, то искомый номер равен 1).

Таким образом, значение single предназначено для одноуровневой нумерации узлов. Для кода:

```
<section>
  <number>
</section>
<section>
  <section>
    <number/>
  </section>
  <section>
    <number/>
  </section>
</section>
```

после применения правила шаблона:

```
<xsl:template match = "number">
  Номер текущей секции <xsl:number count = "section"/>.<br/>
</xsl:template>
```

получим следующий вывод:

```
Номер текущей секции 1.
Номер текущей секции 1.
Номер текущей секции 2.
```

Как видим, для каждого уровня вложенности элементов section была создана отдельная нумерация. Первая строка вывода соответствует первому вхождению элемента number, вторая — второму и т. д., поэтому легко заметить, что нумерации разных уровней не зависят друг от друга.

2.12.4. Значение *multiple*

Рассмотрим ситуацию, когда такая зависимость имеется. Это может быть в том случае, когда, например, части книги нумеруются 1, 2, 3..., а главы, в зависимости от принадлежности к той или иной части, — 1.1, 1.2, 1.3,..., 2.1, 2.2,... Иными словами, нумерация глав книги зависит от нумерации частей.

Для определения многоуровневых и взаимозависимых нумераций, атрибут level предусматривает значение multiple. Алгоритм формирования номера Узла повторяет алгоритм для одноуровневой нумерации за исключением второго шага. На этом шаге из списка предков, созданного на предыдущем шаге, отбираются все узлы, соответствующие образцу, представленному значением атрибута count, а не первый встретившийся.

Не будем фантазировать и рассмотрим пример, который приводится в спецификации XSLT. В этом примере нумеруются заголовки (title) различного уровня, в зависимости от того, к какой структурной единице книги они принадлежат. При этом структура книги определяется следующим образом:

1. книга состоит из глав (chapter) и приложений (appendix);
2. каждая глава и приложение делятся на секции (section);
3. каждая секция делится на подсекции (subsection).

Следующие два правила шаблона реализуют эту нумерацию:

```
<xsl:template match = "title">
  <div>
    <xsl:number level = "multiple"
                count = "chapter| section | subsection"
                format = "1.1"/>
    <xsl:apply-templates/> <br/>
  </div>
</xsl:template>

<xsl:template match = "appendix//title" priority = "1">
  <div>
    <xsl:number level = "multiple"
                count = "appendix | sectionI subsection"
                format = "A.1"/>
    <xsl:apply-templates/> <br/>
  </div>
</xsl:template>
```

Здесь для каждого элемента title отыскивается первый предок с именем appendix, которому присваивается номер А. Всем соседним (sibling) с этим предком элементам присваиваются последовательно номера в, с, D и т. д. Затем в контенте каждого элемента нумеруются элементы section. Их номера: A.1, A.2, ..., B.1, B.2, ... После этого в каждой секции нумеруются элементы subsection, номера которых, в свою очередь, A.I.I, A. 1.2,..., A.2.1, A.2.2,..., в. 1.1, в. 1.2,... Затем, в соответствии с приоритетом правил шаблона, точно так же нумеруются главы, их секции и подсекции, с той разницей, что номера глав — 1, 2, з, ..., а не А, в, с...

В этом примере каждое правило шаблона определяет три уровня нумерации, каждая из которых влияет на остальные две. Различия в форматах номеров для глав, приложений и их составляющих мы опишем немного позд-

нее. А сейчас рассмотрим последнее значение, которое может принимать атрибут level.

2.12.5. Значение any

Это значение используется также при многоуровневых нумерациях, но при этом номер узла не зависит от уровня его вложенности. Это означает, что номера присваиваются узлам последовательно от первого вхождения заданного атрибутом count узла.

Примером может послужить нумерация сносок или примечаний в книге. Сноски могут находиться в любом месте книги. Предположим, что их нумерация не зависит от местоположения, т. е. начавшись в первой главе книги, продолжается непрерывно до ее конца. Для того чтобы определить номер находящейся где-либо сноски, можно воспользоваться следующей конструкцией:

```
<xsl:number count = "note"
            level = "any"/>
```

Итак, если level имеет значение any, подсчет узлов, определенных атрибутом count, происходит от начала документа или от узла, указанного в атрибуте `from`, до текущего узла, т. е. того, для которого вычисляется номер.

2.12.6. Формат нумерации

Форматирование номера или преобразование числа в строку осуществляется с помощью группы атрибутов: format, `lang` и letter-value, каждый из которых является необязательным. При этом определяющее значение имеет **атрибут** format.

Строка, являющаяся значением этого атрибута, представляет собой набор групп символов (лексем), отделенных друг друга сепаратором. Каждая из этих лексем может быть представлена как цифрами, так и буквами. Она задает тот способ, которым будут пронумерованы узлы того уровня вложенности, для которого она определена. Например, если строка формата состоит из трех лексем, то первая из них определяет формат нумерации узлов первого уровня вложенности (A, `B`,...), вторая — второго (АЛ, `A.2`,...) и третья, соответственно, — третьего уровня (`A.1.1`, `A.1.2`,...). Если при этом уровней вложенности нумеруемых узлов больше, чем лексем, то для оставшихся с неопределенным способом форматирования уровней действует последняя в списке лексема.

В качестве составляющих лексемы символов могут быть цифры и буквы английского и некоторых других, представленных в Unicode, алфавитов. Нумерация при этом происходит последовательно по алфавиту или по ряду натуральных чисел и начинается с лексемы, указанной в атрибуте format. При этом количество символов, составляющих номер узла, не может быть меньшим, чем количество символов, составляющих лексему, — для лексемы 007 узлы будут нумероваться так: 008, 009, 010, 011,..., 099, 100, 101,..., 999, 1000 ... Значением по умолчанию является лексема 1, которая соответствует ряду натуральных чисел.

Если к следующему XML-коду:

```
<chapter>
  <title/>
  <section>
    <title/>
    <subsection> <title/> </subsection>
    <subsection> <title/> </subsection>
  </section>
  <section>
    <title/>
    <subsection> <title/> </subsection>
  </section>
</chapter>
<chapter>
  <title/>
   <section>
    <title/>
    <subsection> <title/> </subsection>
  </section>
</chapter>
<appendix>
   <title/>
   <section>
    <title/>
    <subsection> <title/> </subsection>
    <subsection> <title/> </subsection>
   </section>
   <section> <title/> </section>
</appendix>
```

```
<appendix>
   <title/>
   <section>
    <title/>
    <subsection> <title/> </subsection>
  </section>
</appendix>
```

мы применим правила шаблона из *разд. 2.12.4* "Значение multiple", то в результате получим:

1
1.1
1.1.1
1.1.2
1.2
1.2.1
2
2.1
2.1.1
A
AЛ
A.1.1
A.1.2
A.2
B
B.1
B.1.1

о табл. 2.4 приведены некоторые возможные лексемы и их интерпретации.

Таблица 2.4. *Примеры лексем*

Лексема	Интерпретация
1	1, 2, 3, ...
4	4, 5, 6, ...
01	01, 02, 03, ..., 09, 10, 11, ...
a	a, Ь, c, ..., z, ab, ac, ...

Таблица 2.4 (окончание)

Лексема	Интерпретация
A	A, B, C, . . . , Z, AB, AC, ...
I	I, ii, iii, iv, v, vi, . . .
I	I, II, III, IV, V, VI, . . .
II	II, III, IV, V, VI, . . .

Как видно из табл. 2.4, лексема i интерпретируется как римские цифры, а не i, k, l, m, ... Дело в том, что в некоторых языках существуют специфические форматы нумераций. Для того чтобы определить, как именно проводить нумерацию — в алфавитном порядке или по правилам, традиционным для определенного языка, предназначен атрибут letter-value, который может принимать соответственно два значения: alphabetical и traditional. Указание языка осуществляется с помощью атрибута lang, который может принимать те же значения, что и встроенный xml:lang *(см. разд. 1.8)*.

Значение по умолчанию для атрибута letter-value определяется реализацией процессора. Такая свобода выбора предоставлена для неанглоязычных разработчиков. Для атрибута format существует несколько предопределенных значений. Вот некоторые из них:

- format = "ア" — задает нумерацию Katakana;
- format = "๑" — задает нумерацию тайскими цифрами;
- format = "а" — задает старославянскую нумерацию.

В качестве сепараторов для лексем, помимо точки (.), могут выступать запятая (,), символ подчеркивания (_), вертикальная черта (|) и некоторые другие символы.

2.13. Сортировка

В XSLT для сортировки используется директива:

```
<xsl:sort
  select = ключ сортировки
  lang = язык
  data-type = способ интерпретации сортируемых данных
  order = направление сортировки
  case-order = направление сортировки в зависимости от регистра
символов/>
```

Эта директива может быть исключительно непосредственным потомком (child) элементов xsl:apply-templates и xsl:for-each. Сортировка может проводиться одновременно по многим критериям. В этом случае какая-либо из этих директив должна содержать несколько элементов xsl:sort, причем первая из них представляет собой первичный ключ сортировки, вторая — вторичный и т. д.

Если одна или несколько конструкций xsl:sort используется внутри xsl:for-each, то они должны стоять перед всеми остальными директивами. При обработке правила шаблона, содержащего директиву xsl:sort, сначала производится сортировка узлов и лишь затем управление передается соответствующей директиве xsl:apply-templates ИЛИ xsl:for-each. При ЭТОМ текущим набором узлов является тот, который представлен атрибутом select какой-либо из этих директив.

В приведенном ниже примере сортировка файлов производится по трем критериям: имени файла, расширению и размеру. Причем сортировка по имени и расширению происходит в нисходящем порядке, а по размеру — в восходящем.

```
<xsl:template match = "files">
  <p> Отсортированный список файлов: </p>
  <xsl:apply-templates>
    <xsl:sort select ="name"/>
    <xsl:sort select ="ext"/>
    <xsl:sort select ="size" order = "descending"/>
  </xsl:apply-templates>
</xsl:template>
```

Атрибут select директивы xsl:sort представляет образец, определяющий набор узлов для сортировки. Выражение XPath, которое он содержит, задается относительно текущего набора узлов, который мы указали выше. По Умолчанию содержит значение текущего узла (.).

Атрибут lang указывает процессору язык для сортируемых узлов. Он может принимать те же значения, что и встроенный атрибут xml:lang *(см. разд. 1.8)*.

Для указания типа данных сортируемых узлов используется атрибут data-type, который может принимать три значения:

- text — указывает процессору, что узлы должны быть упорядочены лексически в соответствии с особенностями языка, заданного атрибутом lang;
- number — означает, что перед сортировкой данные должны быть преобразованы в число (как если бы к ним была применена функция number о языка XPath, подробнее *см. главу 3);*

- некоторое расширенное имя, которое идентифицирует тип данных, спецификация XSLT не конкретизирует поведение процессора в этом случае. Этот тип данных может быть использован для поддержки стандарта XML Schema.

Для указания направления сортировки используется атрибут order. Он может принимать значение ascending в случае нисходящего направления (сортировка по убыванию) и descending — в случае восходящего (первое значение самое маленькое).

В нашем примере сортировка происходит сначала по имени файла. Файлы располагаются в алфавитном порядке. Далее точно так же сортировка идет по расширению, т. е. если встречаются два файла с одинаковыми именами, первым в упорядоченном списке будет следовать тот, у которого расширение находится выше в отсортированном списке расширений. После этого происходит сортировка по размеру: если встречаются два файла с одинаковыми именами и расширениями, то первым в упорядоченном списке файлов становится тот, чей размер меньше.

И, наконец, атрибут case-order применяется для того, чтобы определить, какой из двух одинаковых символов разного регистра должен стоять первым, а какой — вторым. Этот атрибут может принимать два значения: upper-first и lower-first. Если установлено значение upper-first, то прописные буквы ставятся раньше строчных. В этом случае в списке из двух слов "age" и "Age" после сортировки первым будет стоять "Age". Если же установлено значение lower-case, то первым из этих двух слов будет "age", а "Age" — вторым.

2.14. Сообщения

Сообщения в языке XSLT используются в основном для прерывания обработки документа, но могут применяться и для того, чтобы зафиксировать какую-либо информацию во внутреннем или во внешнем источнике (например, log-файле).

```
<xsl:message
  terminate = флаг прерывания>
<!   XML-контент сообщения -->
</xsl:message>
```

Атрибут terminate, принимающий значения yes или no, указывает процессору, надо ли прерывать обработку документа после обработки сообщения. Если выбрано значение по умолчанию no, то в конечное дерево вставляется обработанный контент сообщения, процессор совершает некоторые специ-

фические для него действия (например, записывает текст сообщения в log-файл) и больше ничего не предпринимает.

Следующий сценарий XSLT описывает правило шаблона, выдающее различные сообщения, а также его применение:

```
<xsl:template name = "messages">
  <xsl:param name = "message_type">warn</xsl:param>
  <xsl:param name = "message"/>
 <xsl:choose>
  <xsl:when test = "$message_type = 'warn' ">
    <xsl:message>
      Предупреждение.
      <xsl:value-of select = "$message"/>
    </xsl:message>
  </xsl:when>
  <xsl:when test = "$message_type = 'err' ">
    <xsl:message>
      Ошибка.
      <xsl:value-of select = "$message"/>
    </xsl:message>
  </xsl:when>
  <xsl:when test = "$message_type = 'fatal_err' ">
      <xsl:message terminate = "yes">
        Фатальная ошибка.
        <xsl:value-of select = "$message"/>
      </xsl:message>
  </xsl:when>
 </xsl:choose>
</xsl:template>

<xsl:template match = "some_element">
...
 <xsl:call-template name = "messages">
   <xsl:with-param name = "message_type">err</xsl:with-param>
   <xsl:with-param name = "message">
     Орфографическая ошибка.
   </xsl:with-param>
 </xsl:call-template>
...
</xsl:template>
```

В том случае, когда message_type будет равен fatal_err, после вывода сообщения о фатальной ошибке обработка документа будет прервана. В остальных случаях не произойдет ничего, кроме вставки в конечное дерево тела сообщения.

2.15. Создание узлов дерева документа

Преобразование исходного дерева XML-документа в конечное происходит путем последовательного создания узлов в этом конечном дереве. Сначала создается корневой элемент с помощью правила шаблона для корневого элемента исходного дерева. Затем с помощью различных правил создаются его потомки. Ими могут быть те узлы, которые получены в результате обработки атрибутов корневого элемента исходного дерева, его непосредственных потомков, а также HTML-контент правила шаблона для корневого элемента.

Но иногда может возникнуть ситуация, когда некоторые узлы исходного дерева не могут быть определены до начала обработки. Предположим, что некоторый атрибут какого-либо элемента исходного дерева должен содержать значение параметра HTTP-запроса. Такой параметр может быть передан как соответствующая часть URL, адресующий рассматриваемый XML-документ, или как часть заголовка HTTP-запроса. Пусть структура XML-документа сильно зависит от значения этого параметра. Если при этом он может принимать достаточно большое количество значений, то обработка документа не может состоять из перебора всевозможных вариантов контента документа, каждый из которых соответствует одному из значений параметра. Тем более что спецификация языка XML не предусматривает средств организации такого перебора в XML-файле.

Таким образом, будет проще, если на этапе обработки у какого-либо элемента возникнет атрибут, значение которого является параметром HTTP-запроса. Или же, если появится элемент, контентом которого будет этот параметр. Вообще говоря, существуют причины, по которым может возникнуть надобность для создания в исходном дереве узла любого типа, не только элемента или атрибута, но и, допустим, инструкции обработки.

До этого момента мы не создавали таких узлов непосредственно. Но неявно мы это делали. В частности, мы создавали текстовые узлы с помощью Директив xsl:value-of, xsl:number И xsl:message. Ведь два следующих фрагмента XML-документов аналогичны, т. е. в результате их обработки мы получим один и тот же документ.

- Первый фрагмент.

  XML-составляющая:

```
<some_element some_attribute = "some_value"/>
```

В том случае, когда message_type будет равен fatal_err, после вывода со, общения о фатальной ошибке обработка документа будет прервана. В остальных случаях не произойдет ничего, кроме вставки в конечное дерево тела сообщения.

2.15. Создание узлов дерева документа

Преобразование исходного дерева XML-документа в конечное происходит путем последовательного создания узлов в этом конечном дереве. Сначала создается корневой элемент с помощью правила шаблона для корневого элемента исходного дерева. Затем с помощью различных правил создаются его потомки. Ими могут быть те узлы, которые получены в результате обработки атрибутов корневого элемента исходного дерева, его непосредственных потомков, а также HTML-контент правила шаблона для корневого элемента.

Но иногда может возникнуть ситуация, когда некоторые узлы исходного дерева не могут быть определены до начала обработки. Предположим, что некоторый атрибут какого-либо элемента исходного дерева должен содержать значение параметра HTTP-запроса. Такой параметр может быть передан как соответствующая часть URL, адресующий рассматриваемый XML-документ, или как часть заголовка HTTP-запроса. Пусть структура XML-документа сильно зависит от значения этого параметра. Если при этом он может принимать достаточно большое количество значений, то обработка документа не может состоять из перебора всевозможных вариантов контента документа, каждый из которых соответствует одному из значений параметра. Тем более что спецификация языка XML не предусматривает средств организации такого перебора в XML-файле.

Таким образом, будет проще, если на этапе обработки у какого-либо элемента возникнет атрибут, значение которого является параметром HTTP-запроса. Или же, если появится элемент, контентом которого будет этот параметр. Вообще говоря, существуют причины, по которым может возникнуть надобность для создания в исходном дереве узла любого типа, не только элемента или атрибута, но и, допустим, инструкции обработки.

До этого момента мы не создавали таких узлов непосредственно. Но неявно мы это делали. В частности, мы создавали текстовые узлы с помощью директив xsl:value-of, xsl:number И xsl:message. Ведь два следующих фрагмента XML-документов аналогичны, т. е. в результате их обработки мы получим один и тот же документ.

- Первый фрагмент.

  XML-составляющая:

```
<some_element some_attribute = "some_value"/>
```

XSLT-составляющая:

```
<xsl:template-match = "some_element">
  <xsl:value-of select = "some_element"/>
</xsl:template>
```

р Второй фрагмент.

XML-составляющая:

```
<some_element some_attribute = "some_value">
  some_value
</some_value>
```

XSLT-составляющая:

фрагмент соответствующего сценария XSLT пуст, т. к. нам здесь будет достаточно применения к some_element встроенного правила шаблона *(см. разд. 2.6.4)*.

Во втором варианте у элемента some_element есть непосредственный потомок — текстовый узел со значением some_value, в первом мы создаем этот узел с помощью директивы xsl:value-of. Точно так же текстовые узлы создаются С ПОМОЩЬЮ xsl:message И xsl:number.

Здесь очень важно заметить, что так, как описано выше, текстовые узлы создаются в результирующем дереве, а не в исходном. В спецификациях XML и XSLT не описано ни одного средства для динамического формирования XML-документа, т. е. для создания узлов в исходном дереве. Это означает, что созданные узлы будут представлены "как есть", без применения к ним каких-либо правил шаблона. Это происходит потому, что язык XPath, который используют директивы XSLT, описывает только узлы исходного Дерева и не может адресовать узлы результирующего.

Далее мы разберем, как средствами XSLT создавать узлы произвольного типа в результирующем дереве, а также некоторые расширения, предоставляемые Cocoon, позволяющие создавать узлы также произвольного типа, в исходном дереве.

2.15.1. Создание элемента

В XSLT для создания элементов используется следующая директива:

```
<xsl:element
  name = имя нового элемента
  namespace = пространство имен нового элемента
  Use-attribute-set = перечень имен наборов атрибутов для элемента >
<!-- контент нового элемента -->
</xsl:element>
```

Здесь обязательный атрибут name содержит имя того элемента, который будет создан. Необязательный атрибут namespace указывает, к какому пространству имен будет принадлежать новый элемент. Также необязательный атрибут use-attribute-sets может содержать список именованных наборов атрибутов *(см. разд. 2.15.3),* которыми будет обладать создаваемый элемент.

В следующем примере мы продемонстрируем создание элемента абзаца (иными словами, в результирующем HTML-коде появится тег <p>). Если к XML-составляющей:

```
<article note = "под редакцией проф. Воловича.">
Текст статьи.
</article>
```

мы применим следующее правило шаблона:

```
<xsl:template match = "article">
 <div>
  <xsl:element name = "p">
    Примечание:
    <xsl:value-of select = "@note">
  </xsl:element>
  <xsl:apply-templates/>
 </div>
</xsl:template>
```

то в результате получим такой **HTML-код:**

```
<div>
<p>Примечание: под редакцией проф. Воловича.</p>
Текст статьи.
</div>
```

Если же у нас возникнет необходимость создать элемент в исходном дереве. то мы можем воспользоваться директивой xsp:element, определенной Б Cocoon. Эта директива также имеет обязательный атрибут name, указывающий имя создаваемого элемента:

```
<some_element>
  <xsp:element name = "new_element"> some_value </xsp:element>
  Некоторый текст
</some_element>
```

Заметим, что в отличие от директивы xsl:element, мы поместили xsp:element в XML-файл, а не в сценарий XSLT. Далее приводится часть этого сценария:

```
<xsl:template match = "some_element">
  <b><xsl:value-of select = "new_element"/></b>
  <p><xsl:apply-templates select = "text()"></p>
</xsl:template>
```

После обработки документа мы получим следующий HTML-код:

```
<b>some_value</b>
<p>Некоторый текст</p>
```

Мы не будем формально описывать синтаксис тех директив, которые предлагает Cocoon, т. к. они не определены в спецификациях W3-консорциума. Но тем не менее, мы покажем, как пользоваться некоторыми из них. Подробнее о Cocoon и предлагаемых им расширениях можно прочитать в *главе 6.*

2.15.2. Создание атрибута

Для того чтобы создать атрибут, в результирующем дереве используется следующая директива:

```
<xsl:attribute
  name = имя нового атрибута
  namespace = пространство имен для имени нового атрибута>
<!-- значение атрибута -->
</xsl:attribute>
```

Директива `xsl:attribute` может быть применена для создания атрибутов как фиксированных элементов (т. е. определенных явно в XML-файле), так и тех, которые были созданы с помощью директивы `xsl:element`. Если у элемента, для которого создается атрибут, уже присутствует атрибут с таким именем, то это не вызовет ошибки дублирования атрибутов. Вместо этого значение данного атрибута будет заменено на новое.

Как видно из синтаксических правил, директива `xsl:attribute` имеет два атрибута. После создания новый атрибут будет иметь имя, заданное обязательным атрибутом name, которое принадлежит пространству имен, заданному необязательным атрибутом namespace.

В следующем фрагменте сценария XSLT для элемента p задается отступ, если представляемый им абзац является первым в рассматриваемом блоке текста:

```
<xsl:template match = "para">
 <p>
  <xsl:if test = "../para[1] =.">
   <xsl:attribute name = "style">text-indent = 1.5em</xsl:attribute>
  </xsl:if>
```

```
  <xsl:apply-templates/>
 </p>
</xsl:template>
```

Примечание

Обратите внимание, что мы использовали образец "../para =." вместо "position () = 1" в силу того, что у элементов para могут быть соседи (sibling) с другим именем, которые также будут сосчитаны. В следующем XML-коде:

```
<article>
  <para>
   Некоторый текст
  </para>
  <para>
   Какой-либо еще текст
  </para>
</article>
```

номер позиции первого элемента para будет равен 2, а второго 4. Позиции с номерами 1 и 3 будут занимать текстовые узлы, состоящие из символа перевода строки.

Кроме этой ситуации при создании атрибутов могут возникать следующие, нарушающие правильное представление документа ошибки.

- Атрибут добавляется к элементу после того, как к этому элементу был добавлен непосредственный потомок. В нашем примере такая ошибка возникнет в том случае, если мы сразу после директивы `xsl:if` вставим текст или какую-либо директиву XSLT, которая порождает непустой контент.
- Атрибут добавляется к узлу, не являющемуся элементом.
- Значение создаваемого атрибута не является текстовым. Такая ситуация может возникнуть, например, если в контенте директивы `xsl:attribute` мы будем использовать буквы русского алфавита, но при этом в прологе документа не укажем соответствующую кодировку (encoding = "...").

Согласно спецификации XSLT при возникновении этих ошибок процессор должен либо сигнализировать об этом, либо проигнорировать директиву создания атрибута.

При необходимости создания атрибута в исходном дереве можно воспользоваться директивой xsp:attribute, определенной в Cocoon. Рассмотрим, как это происходит.

XML-Фрагмент:

```
<some_element>
  <xsp:attribute name = "attr1"> value1</xsp:attribute>
  <xsp:attribute name = "attr2"> value2</xsp:attribute>
</some_element>
```

XSLT-фрагмент:

```
<xsl:template match = "some_element">
  Значение атрибута attrl равно
   <xsl:value-of select = "@attr1"/><br/>
  Значение атрибута attr2 равно
   <xsl:value-of select = "@attr2"/>
</xsl:template>
```

После обработки этого кода получим:

Значение атрибута attrl равно value1
Значение атрибута attr2 равно value2

С помощью директивы xsp: attribute атрибут может быть добавлен, так же как и с помощью xsl: attribute, либо к фиксированному элементу, либо к элементу, созданному директивой xsp:element. Содержимым директивы xsp:attribute может быть только XML-текст или xsp:expr — еще одна из дополнительных директив Cocoon. О том, что представляет собой директива xsp:expr, можно прочитать в *главе 6.*

2.15.3. Именованные наборы атрибутов

Если в сценарии XSLT создается много новых атрибутов, то иногда бывает полезно объединить некоторые директивы создания этих атрибутов в группы по какому-либо критерию. Предположим, что нам необходимо создать XML-документ, содержащий информацию о геометрических фигурах трех типов: треугольник, прямоугольник и квадрат. Фигура каждого типа характеризуется длинами своих сторон, т. о. квадрат будет иметь одну характеристику, прямоугольник — две и треугольник — три характеристики. Рассмотрим, как именованные наборы атрибутов могут быть использованы для решения этой задачи.

Для определения именованного набора атрибутов служит директива

```
<xsl:attribute-set
  name = имя набора
  use-attribute-sets = имена других используемых наборов >
<!-- директивы создания атрибутов (xsl:attribute) -->
</xsl:attribute-set>
```

Здесь атрибут name задает имя для создаваемого набора атрибутов, а use-attribute-sets — имена других, созданных ранее, наборов атрибутов, которые войдут в состав данного набора. Контент директивы xsl:attribute-set состоит из одного или нескольких определений атрибутов xsl:attribute **ИЛИ может быть ПУСТЫМ.**

Пусть для описания множества геометрических фигур в XML-файле используется элемент figure с атрибутом type, задающим тип фигуры, а длины сторон каким-либо образом вычисляются на этапе обработки документа и их значения хранятся в переменных value1, value2 и value3. Код будет таким:

```
<xsl:attribute-set name = "square">
  <xsl:attribute name = "side1">
   <xsl:value-of select = "$value1"/>
  </xsl:attribute>
</xsl:attribute-set>

<xsl:attribute-set name = "rectangle"
  use-attribute-sets = "square">
  <xsl:attribute name = "side2">
   <xsl:value-of select = "$value2"/>
  </xsl:attribute>
</xsl:attribute-set>

<xsl:attribute-set name = "triangle"
  use-attribute-sets = "rectangle">
  <xsl:attribute name = "side3">
   <xsl:value-of select = "$value3"/>
  </xsl:attribute>
</xsl:attribute-set>

<xsl:template match = "figure">
  <xsl:element name = "@type"
    use-attribute-sets = "@type"/>
</xsl:template>
```

Таким образом, в зависимости от значения атрибута type будет создан элемент С каким-либо **ИЗ Имен** square, rectangle ИЛИ triangle, **обладающий** соответствующим набором атрибутов. При этом предполагается, что обработчик конечного дерева (в нашем случае синтаксический анализатор HTML) умеет обрабатывать элементы с такими именами.

Мы уже говорили, что если создается атрибут с таким именем, которое уже задано для какого-либо атрибута фиксированного элемента, то ошибки не будет. В данном случае этот атрибут просто поменяет свое значение на новое:

```
<xsl:template match = "para">
  <p style = "text-indent:1.5em">
   <xsl:attribute name = "style">text-indent:2em</xsl:attribute>
   <xsl:apply-templates/>
  </p>
</xsl:template>
```

С помощью этого правила шаблона в результирующем дереве будет создано следующее поддерево:

```
<p style = "text-indent:2em">
...
</p>
```

Но, тем не менее, при использовании директив `xsl:attribute` и `xsl:attribute-set` в некоторых случаях могут возникать ошибочные ситуации. Рассмотрим порядок присоединения создаваемых атрибутов к элементу:

1. Сначала к элементу подключается атрибуты из наборов, указанных в директиве use-attribute-sets.
2. Затем подключаются фиксированные конечные атрибуты, т. е. указанные в сценарии явно без использования `xsl:attribute` ИЛИ use-attribute-set.
3. И наконец, к элементу подключаются те атрибуты, которые определены **С ПОМОЩЬЮ Директивы** `xsl:attribute`.

Таким образом, при совпадении имен некоторых атрибутов, явно заданные атрибуты перекрывают атрибуты, указанные в use-attribute-set, но и сами они могут быть перекрыты атрибутами, созданными с помощью директивы `xsl:attribute`.

Определение набора атрибутов аналогично применению соответствующего количества директив `xsl:attribute`. Это значит, что вместо определения набора атрибутов может быть использована директива `xsl:attribute` для определения каждого атрибута из этого набора. Причем последовательность создания атрибутов как в первом, так и во втором случае одинакова. Поэтому если в определении набора атрибутов или где-либо в аналогичной ему

последовательности директив xsl:attribute встретятся определения атрибутов с одним и тем же именем, то будет зафиксирована ошибка. Процессор может как сигнализировать об этой ошибке, так и обработать ее самостоятельно.

Ошибка дублирования определений атрибутов (как и любая другая ошибка дублирования чего-либо в сценарии XSLT) может возникнуть только в том случае, когда директивы определения имеют один и тот же приоритет *(см. разд. 2.6.1)*. Кроме того, для именованных наборов атрибутов имеет значение местоположение их определения. В нашем примере мы определяли свои наборы с помощью директив верхнего уровня (то есть непосредственных потомков элемента xsl:stylesheet), хотя могли бы это сделать и внутри какого-либо правила шаблона. Таким образом, для именованных наборов атрибутов ошибка дублирования атрибутов может возникнуть только в том случае, если их определения обладают одной и той же областью видимости *(см. разд. 2.9.2)*.

Последний тип ошибочной ситуации, которая может возникнуть в случае именованных наборов атрибутов, представляет собой рекурсивное определение этих наборов:

```
<xsl:attribute-set name = "some_attributes"
  use-attribute-sets = "some_attributes">
...
</xsl:attribute-set>
```

Причем рекурсия может быть не только прямой, но и косвенной.

2.15.4. Создание текстового узла

При формировании текстовых узлов в конечном дереве можно ввести текст непосредственно (с клавиатуры), а можно воспользоваться директивой:

```
<xsl:text
  disable-output-escaping = индикатор обработки текста >
<!-- текст -->
</xsl:text>
```

Эта директива, как правило, используется для ввода текста, содержащего либо разметку (угловые скобки, амперсант и т. д.), либо те неслужебные символы, которые не должны обрабатываться процессором (несколько пробелов подряд, символы конца строки и т. д.). Для корректного отображения такого текста нужно установить значение yes для необязательного атрибута disable-output-escaping. При таком использовании директива xsl:text будет выполнять те же функции, что и секция CDATA.

Следующие два фрагмента аналогичны:

```
<xsl:text disable-output-escaping = "yes">
 if (x > 0 && x < 10)  y = x;
</xsl:text>

<xsl:text>
 if (x &gt; 0 &amp;&amp; x &lt; 10)  y = x;
</xsl:text>
```

Атрибут disable-output-escaping может принимать только два значения: yes и no, при этом no является значением по умолчанию.

2.15.5. Создание инструкции обработки

Для создания инструкций обработки в результирующем дереве используется директива

```
<xsl:processing-instruction
 name = имя инструкции обработки >
<!-- текст PI -->
</xsl:processing-instruction>
```

В следующем примере создается инструкция обработки, указывающая процессору, что для обработки документа необходимо использовать таблицу стилей CSS:

```
<xsl:processing-instruction name = "xml-stylesheet">
  href = "mycss.css"
  type = "text/ess"
</xsl:processing-instruction>
```

После обработки этого шаблона мы получим такую инструкцию обработки:

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

Для создания инструкций обработки в исходном дереве Cocoon предлагает собственную Директиву xsp:pi. В следующем шаблоне создается инструкция обработки, которая подключает к XML-документу стилевую таблицу:

```
<xsp:pi name = "xml-stylesheet">
  href = "myxsl.xsl"
  type = "text/xsl"
</xsp:pi>
```

Результатом его обработки будет:

```
<?xml-stylesheet href = "myxsl.xsl" type = "text/xsl"?>
```

2.15.6. Создание комментария

Комментарий в конечном дереве создается с помощью следующей директивы:

```
<xsl:comment>
  <!-- текст комментария -->
</xsl:comment>
```

ЕСЛИ В результате обработки директив xsl:comment ИЛИ xsl:processing-instruction их контент не будет текстовым, то процессор может либо сигнализировать об ошибке, либо исправить ее по своему усмотрению. Кроме этого ошибочной будет ситуация, когда текст комментария после обработки будет содержать двойной дефис (--) или будет на дефис (-) заканчиваться.

В результате обработки шаблона

```
<xsl:comment>
  начало <xsl:number count = "part"/>-ой части
</xsl:comment>
```

будет создан следующий комментарий:

```
<!-- начало л-ой части -->
```

где *n* — номер соответствующей части книги.

Для создания комментария в исходном дереве в Cocoon определена директива xsp:comment, СПОСОб ИСПОЛЬЗОВанИЯ КОТОРОЙ сходен С xsl:comment. Разница между ними заключается только в том, что xsl:comment применяется в сценарии XSLT, а директива xsp:comment может быть использована и в XML-файле.

2.16. Копирование узлов и поддеревьев

Для копирования текущего узла в XSLT существует простой способ, реализующийся с помощью директивы

```
<xsl:copy
  use-attribute-sets = именованные наборы атрибутов>
<!-- контент для копируемого узла -->
</xsl:copy>
```

Директива xsl:copy используется, как правило, в двух случаях: для подключения к какому-либо элементу некоторых именованных наборов атрибутов (с помощью use-attribute-sets, подробнее об этом атрибуте *см. разд. 2.15.3),* а также в тех случаях, когда к нескольким узлам конечного дерева требуется подключить один и тот же набор узлов. Проиллюстрируем каждый из этих вариантов.

предположим, что в конечном дереве присутствует несколько абзацев, представленных элементом p, каждый из которых имеет какой-либо из двух наборов атрибутов — att_list1 и att_list2, которые влияют на способ отображения этих абзацев. Следующее правило шаблона регулирует эту ситуацию:

```
<xsl:template match = "P">
  <xsl:choose>
   <xsl:when test = "view1">
    <xsl:copy use-attribute-sets = "att_list1"/>
   </xsl:when>
   <xsl:when test = "view2">
    <xsl:copy use-attribute-sets = "att_list2"/>
   </xsl:when>
   <xsl:otherwise>
    <xsl:copy/>
   </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Тем самым, при способе отображения view1 к элементу p добавляется набор атрибутов с именем att_list1, при view2 — att_list2, и никаких наборов атрибутов не добавляется во всех остальных случаях. Кроме этого, контент элемента p из исходного дерева, а также его атрибуты в конечное дерево не копируются.

Следующий шаблон демонстрирует второй вариант директивы xsl:copy:

```
<xsl:for-each select = "@style">
  <xsl:copy/>
</xsl:for-each>
```

Здесь атрибуты style исходного дерева копируются в элементы конечного дерева. Элементами конечного дерева могут быть, например, p или DIV. Такой прием может понадобиться при выборочном копировании. Например, в том случае, когда копируются только те атрибуты, которые удовлетворяют определенным условиям. Или тогда, когда копирование происходит не для всех типов элементов. Заметим, что копироваться могут не только атрибуты, но и сами элементы. Тогда контент директивы xsl:copy может содержать XHTML-код для копирования.

Как мы уже говорили, директива xsl:copy служит для копирования только самого узла, без его потомков. Если же возникает необходимость скопировать поддерево или набор узлов конечного дерева в конечное дерево, то применяется директива

```
<xsl:copy-of
  select = адрес копируемой структуры />
```

Здесь атрибут select содержит выражение XPath, описывающее те узлы, которые подлежат копированию вместе со всеми своими потомками (ими могут быть элементы, атрибуты и узлы пространства имен). Можно сказать, что директива xsl:copy-of делает то же самое, что и xsl:value-of, только без преобразования результата в строку. При этом узлы копируются в том порядке, в котором они указаны в атрибуте select или (в случае единственного узла) так, как они расположены в адресуемом поддереве. Если же адрес не является ни набором узлов, ни каким-либо поддеревом, то он преобразуется в строку так, как если бы использовалась директива xsl:value-of.

2.17. Особенности обработки документа

Итак, мы разобрали основные конструкции языка XSLT. Последнее, что нам осталось сделать, — это рассмотреть в целом некоторые особенности преобразования исходного дерева в конечное. В частности, в этом разделе главы мы опишем, как процессор XML может обрабатывать пробельные символы, а также различные формы вывода конечного дерева.

В *разд. 1.8* мы уже касались вопроса обработки пробелов. Мы использовали для этого атрибут xml:space. В XSLT существуют конструкции со сходными функциями. Но прежде чем перейти к их рассмотрению, мы должны выяснить, как процессор обрабатывает пробельные символы по умолчанию, т. е. безотносительно каких-либо конкретных инструкций.

2.17.1. Обработка пробельных символов

В языке XML пробельными считаются следующие четыре символа: непосредственно сам пробел (код символа #x2o), символ табуляции (#x9), символы конца строки и возврата каретки (#xA и #xD). Напомним, что сценарий XSLT создается в соответствии с синтаксисом XML, и поэтому эти символы являются пробельными как для XML-файла, так и для подключенного к нему сценария XSLT.

При обработке пробельных символов из документа (после построения исходного и конечного деревьев, но перед их обработкой) могут быть вычищены (stripping) некоторые узлы, содержащие исключительно эти символы. Узлы, которые не вычищаются из документа, должны удовлетворять какому-либо из следующих трех условий.

- Узел представляет собой элемент, имя которого включено в список элементов, сохраняющих пробельные символы.

- Узел содержит хотя бы один пробельный символ.
- Элемент имеет предка, для которого установлен атрибут xml:space, и его значением является preserve. Причем для данного элемента нет более близкого предка, для которого xml:space принимает значение default.

Все остальные узлы из дерева вычищаются.

В сценарии XSLT для того, чтобы включить элемент в список элементов, сохраняющих пробелы, служит директива

```
<xsl:preserve-space
  elements = список элементов />
```

Для исключения элементов из списка предназначена директива

```
<xsl:strip-space
  elements = список элементов />
```

По умолчанию все элементы исходного дерева являются сохраняющими пробельные символы. При помощи директивы xsl:strip-space можно указать те элементы, пробельные символы в которых сохраняться не будут. Для этого необходимо присвоить атрибуту elements данной директивы строковое значение, представляющее собой перечень имен таких элементов, отделенных друг от друга пробелом.

Директива xsl:preserve-space является прямой противоположностью по отношению к xsl:strip-space и используется точно так же. Ее атрибут elements должен содержать имена элементов, для которых пробельные символы сохраняются. Обе директивы должны быть элементами верхнего уровня.

Если имя некоторого элемента встречается в обоих списках, представленных этими двумя директивами, то возникающая при этом конфликтная ситуация Разрешается точно так же, как и в случае правил шаблонов *(см. разд. 2.6.1)*. Таким образом, сначала из рассмотрения исключаются те элементы, которые имеют более низкий приоритет импорта, затем те, которые имеют более низкий приоритет по умолчанию. Если конфликтная ситуация сохраняется, то процессор должен либо сигнализировать об ошибке, либо выбрать тот элемент, чье имя встречается в последнем из конфликтующих списков.

2.17.2. Управление выводом

При формировании конечного дерева существует возможность получить в результате обработки документ, который может не являться XML-документом. В XSLT определена директива xsl:output, которая позволяет повлиять на вывод конечного документа. Здесь под выводом подразумевается следующее. После преобразования исходного дерева в конечное тот

документ, который получился в результате преобразования (назовем условно его новым), может быть представлен различными способами, например, отображен в окне браузера или, если в результате получился документ в формате XML, то он может быть подвергнут дальнейшей обработке с помощью еще какого-либо сценария. Вот этот процесс представления конечного дерева мы и будем называть выводом, который естественным образом зависит от структуры и наполнения конечного документа.

Таким образом, по-разному определяя некоторые аспекты структуры и содержания документа, мы можем повлиять на способы его представления. Вообще говоря, процессор не обязан поддерживать различные формы вывода, но если он это делает, то строго в соответствии с тем, как это определено ДИРеКТИВОЙ `xsl:output`.

Примечание

Cocoon не поддерживает директиву `xsl:output`. И хотя это не имеет большого значения для тех целей, которым служит книга, мы опишем ее действие настолько подробно, насколько это необходимо для объяснения различных способов вывода и, в частности, того из них, который реализован в Cocoon.

Эта директива представляет собой пустой элемент верхнего уровня и подчиняется следующим правилам синтаксиса:

```
<xsl:output
  method = форма вывода
  version = версия
  encoding = кодировка
  omit-xml-declaration = флаг присутствия объявления XML-документа
  standalone = флаг автономности документа
  doctype-public = публичный ключ
  doctype-system = системный ключ
  cdata-section-elements = список имен элементов
  indent = флаг наличия дополнительных пробелов
  media-type = MIME тип контента />
```

Главным атрибутом этой директивы является method, который идентифицирует способ, с помощью которого производится вывод конечного документа. Он может либо принимать одно из предопределенных значений — `xml`, html и text, либо указывать тот способ вывода, который поддерживается данным конкретным процессором. Если в сценарии XSLT директива `xsl:output` отсутствует вовсе или в том случае, когда атрибут method опущен, используется его значение по умолчанию. Это значение равно `html`, если:

- корень (не корневой элемент!) документа имеет своим непосредственным потомком элемент;
- первый элемент, являющийся непосредственным потомком корня, имеет имя html, причем это имя может состоять из произвольного сочетания прописных и строчных букв;
- все текстовые узлы, являющиеся непосредственными потомками корня и находящиеся перед первым элементом, состоят только из пробельных символов.

Если какое-либо из этих условий не выполнено, то в качестве значения по умолчанию выбирается `xml`.

Остальные атрибуты директивы `xsl:output` определяют различные параметры метода вывода, причем не все из них определены при некоторых значения атрибута method. Далее приведем общее описание каждого атрибута:

- version — указывает номер версии метода вывода;
- indent — указывает, могут ли появляться при выводе дополнительные пробелы, принимает значение yes или no;
- encoding — указывает вид кодировки, которую должен использовать XSLT-процессор при выводе документа;
- media-type — указывает тип MIME контента данных, полученных при выводе;
- `doctype-public` — указывает публичный идентификатор, который должен использоваться в объявлении типа документа;
- `doctype-system` — указывает системный идентификатор, который должен использоваться в объявлении типа документа;
- `omit-xml-declaration` — указывает, должна ли быть выведена декларация XML, принимает значения yes или no;
- standalone — указывает, должна ли быть декларация XML декларацией одиночного элемента, принимает значения yes или no;
- `cdata-section-elements` — содержит список имен элементов, чей контент должен выводиться с помощью секций CDATA.

Если в документе присутствует несколько директив `xsl:output`, то конфликты между значениями их атрибутов разрешаются точно так же, как и в случае с правилами шаблона *(см. разд. 2.6.1)*. Специфика некоторых из перечисленных атрибутов зависит от выбранного метода вывода, описания которых приводятся ниже.

Метод вывода *xml*

Если выбран этот метод, то после обработки выводится корректный XML-документ, т. е. документ, удовлетворяющий всем синтаксическим правилам

языка XML. Для этого метода значения всех вышеперечисленных атрибутов могут быть определены. Причем если какое-либо из этих значений не имеет смысла для процессора, но является корректным, то вместо него используется значение по умолчанию.

Рассмотрим табл. 2.5, описывающую данные атрибуты.

Таблица 2.5. *Атрибуты элемента* `xsl:output` *при методе вывода XML*

Атрибут	Пояснение	Значение по умолчанию
Version	Версия XML	1.0
Indent	Без специфики	по
Encoding	Без специфики	UTF-8 ИЛИ UTF-16
media-type	Без специфики	text/xml
doctype-public	Если указан, то в объявлении типа документа должен присутствовать публичный идентификатор	отсутствует
doctype-system	Если указан, то в объявлении типа документа должен присутствовать системный идентификатор	отсутствует
omit-xml-declaration	Используется только при методе вывода xml	по
Standalone	Если присутствует декларация XML, то standalone в ней имеет значение данного атрибута	отсутствует
cdata-section-element	Каждый элемент из списка, представляющего собой значение этого атрибута, будет обработан так, как если бы его контент был заключен в секцию CDATA	отсутствует

При использовании метода вывода xml в результате вывода конечного дерева может быть получено новое дерево, узлы которого могут отличаться от конечного иным порядком атрибутов. Кроме того, новое дерево может содержать узлы пространства имен, которые не присутствовали в конечном дереве.

Метод вывода *html*

В наших разработках этот метод используется по умолчанию, поэтому рассмотрим его подробно.

Метод html используется для представления конечного дерева в формате HTML, в соответствии со стандартом HTML 4.0. Далее мы последовательно перечислим все те синтаксические правила этого стандарта, в соответствии с которыми должно быть представлено или выведено конечное дерево.

- Для пустых элементов не должны выводиться закрывающие теги. Согласно стандарту ПУСТЫМИ Элементами В HTML ЯВЛЯЮТСЯ: area, base, basefont, br, col, frame, hr, img, input, isindex, link, meta И param. Например, элемент, представленный тегом <br/>, должен выводиться как <br>.
- Названия узлов в выводимом дереве не должны зависеть от регистра символов.
- Контент элементов script и style должен выводиться как содержимое секции CDATA, например, < должно быть выведено как <.
- Атрибут indent директивы xsl:output по умолчанию имеет значение yes. В данном случае это означает, что процессор может добавлять или удалять пробельные символы так, чтобы браузер (или другой клиент) был не чувствителен к этим операциям добавления и удаления (то есть чтобы эти операции не влияли на отображение документа в этом клиенте).
- Процессор должен игнорировать символы, не принадлежащие набору ASCII.
- Любая ссылка на сущность должна быть заменена содержимым этой сущности, если данная сущность поддерживается стандартом HTML.
- Инструкции обработки должны заканчиваться символом >, а не ?>.
- Булевы атрибуты, т. е. такие, значение которых определяется только наличием или отсутствием самого атрибута, должны быть представлены в сокращенной форме. Например, конструкция <OPTION selected = "selected"> должна быть представлена как <OPTION selected>.
- Атрибут encoding, если он присутствует, должен указывать кодировку Для вывода документа. Если при этом присутствует элемент с именем HEAD, то этот элемент должен содержать элемент META, указывающий эту кодировку. Например,

```
<HEAD>
  <META http-equiv = "content-type" content = " text/html; charset =
koi8-r">
...
</HEAD>
```

- Для атрибутов doctype-system и doctype-public, если они указаны, сразу перед первым элементом должно идти объявление типа документа, содержащее соответственно системный и публичный идентификаторы.

Например,

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html40/strict.dtd">
<HTML>
...
```

- Для метода вывода html можно использовать атрибут media-type, значением по умолчанию для которого является text/html.

Во всех листингах этой книги подразумевается метод вывода html. Подробнее о том, как именно Cocoon представляет конечное дерево, можно прочитать в *главе 6*.

Метод вывода *text*

Метод вывода text представляет конечное дерево, выводя значение для каждого его текстового узла в том порядке, в котором эти узлы следуют в документе. При этом не учитывается, является ли символ специальным (например, угловые скобки) или нет.

Для этого метода вывода могут быть определены два атрибута директивы xsl:output. Первый — encoding — определяет кодировку выводимого документа, второй — media-type — его тип MIME. Значением по умолчанию ДЛЯ media-type является — text/plain.

ГЛАВА 3

Язык XPath

3.1. Структура XPath

Язык XPath представляет собой механизм, при помощи которого осуществляется связь между технологиями XML и XSLT. Конструкции XML используются для структурирования информации, директивы XSLT — для обработки этой информации, а XPath — для описания тех частей дерева XML, которые должны быть обработаны отдельно взятой директивой XSLT. Это описание строится с помощью некоторой системы адресации, синтаксис которой отличается от синтаксиса языка XML, но не отменяет его правила. Это означает, что, например, при использовании некоторой конструкции XPath в качестве значения атрибута, мы не можем использовать угловые скобки (< и >), вместо них должны применяться ссылки на соответствующие сущности (`<` И `>` соответственно).

Для придания гибкости системе адресации в XPath определены числа, строки, булевы значения и предоставлен инструментарий для манипулирования ими. Такие возможности могут понадобиться при создании новых и модификации уже имеющихся текстовых узлов. В частности, с помощью директивы `xsl:value-of` мы можем вычислять арифметические выражения, выполнять разбиение и конкатенацию строк и затем вставлять получившиеся результаты в конечный документ.

Конструкции языка XPath, которые после обработки возвращают число, строку или булево значение, принято называть выражениями (expressions). Те же из них, которые возвращают значение типа набор узлов (node-set), имеют собственное название — пути адресации (location path), хотя также являются выражениями. Далее в этой главе мы будем рассматривать выражения и пути адресации и по отдельности, и как единое Целое.

3.2. Пути адресации

Итак, путь адресации — это такое выражение XPath, которое в результате обработки возвращает значение типа "набор узлов". И хотя во второй главе мы часто использовали эти конструкции, здесь приведем еще несколько примеров, взятых из спецификации XPath. Заметим, что приведенные ниже пути адресации могут быть записаны не так громоздко с помощью т. н. сокращенного синтаксиса, который мы опишем позднее.

- `child::para` — адресует элемент para, являющийся непосредственным потомком текущего узла;
- `child::*` — описывает все элементы, являющиеся непосредственными потомками текущего узла;
- `child::text()` — описывает все текстовые узлы, являющиеся непосредственными потомками текущего узла;
- `child::node()` — описывает все непосредственные потомки текущего узла независимо от их типа (элементы, атрибуты и т. д.);
- `attribute::name` — описывает атрибут с именем name, принадлежащий текущему узлу;
- `attribute::*` — описывает все атрибуты текущего узла;
- `descendant::para` — описывает все элементы para, являющиеся потомками текущего узла;
- `ancestor::para` — описывает все элементы para, являющиеся предками текущего узла;
- `ancestor-or-self::para` — описывает тот же набор узлов, что и `ancestor::para`, плюс текущий узел, если его имя para;
- `descendant-or-self::para` — то же, что и `descendant::para`, плюс текущий узел, если его имя para;
- `self::para` — адресует текущий элемент, если его имя para, в противном случае возвращает пустой набор узлов;
- `child::chapter/descendant::para` — среди всех потомков элемента chapter, являющегося непосредственным потомком текущего узла, выбирает все элементы с именем para;
- `child::*/child::para` — описывает все элементы para, являющиеся потомками текущего узла во втором поколении;
- `/` — адресует корень документа;
- `/descendant::para` — адресует все элементы para, находящиеся в текущем документе;

ft /descendant: : olist/child: : item — **адресует все элементы** item, **находящиеся** где-либо в документе и чьими родителями является olist;

- child::para[position()=1] — адресует первый непосредственный потомок текущего узла с именем para;
- child::para[position()=last()]— адресует последний непосредственный потомок текущего узла с именем para;

p child::para[position()=last()-1] — адресует предпоследний непосредственный потомок текущего узла с именем para;

P child::para[position()>1] — описывает Все элементы para, являющиеся непосредственными потомками текущего узла, кроме первого;

O following-sibling::chapter[position()=1] — адресует элемент chapter, который имеет с узлом контекста общего родителя и следует сразу после него;

- preceding-sibling::chapter[position()=1] — адресует элемент chapter, который имеет с узлом контекста общего родителя и находится непосредственно перед ним;

Q /descendant: : figure [position () =42] — **адресует В текущем документе СОРОК ВТОрОЙ Элемент** figure;

Q /child::doc/child::chapter[position() =]/child::section[position() 2] — описывает ВТОрОЙ элемент section в пятом элементе chapter, который является непосредственным потомком элемента документа doc;

Q child::para[attribute::type = "warning"] — **описывает все Элементы** para, являющиеся непосредственными потомками текущего узла и имеющие атрибут с именем type и значением warning;

- child::para[attribute::type = "warning"][position()=5] — описывает элемент para, являющийся пятым непосредственным потомкам текущего узла и имеющий атрибут type, чье значение равно warning;
- child::para[position() = 5][attribute::type = "warning"] — **возвращает ТО же, ЧТО И** child::para[attribute::type= "warning"][position()=5], но отличается от него порядком обработки;
- child::chapter[child::title = "Introduction"] — описывает Элемент title, являющийся непосредственным потомком chapter и имеющий значение introduction результатом обработки своего контента (иными словами, мы получим это значение, если выполним <xsl:value-of select = "child::title"/>);
- child::chapter[child::title] — описывает элемент chapter, ЯВЛЯЮЩИЙСЯ непосредственным потомком текущего узла и имеющий среди своих непосредственных потомков хотя бы один элемент с именем title;

- `child::*[self::chapter or self::appendix]` — адресует Все элементы chapter и appendix, являющиеся непосредственными потомками текущего узла;
- `child::*[self::chapter or self::appendix][position() = last()]` — описывает последний из всех элементов chapter и appendix, являющихся непосредственными потомками текущего узла.

Пути адресации бывают относительными и абсолютными. Относительный путь адресации строится из т. н. шагов адресации, отделенных друг от друга прямым слешем (/). Как видно из примеров, каждый путь адресации зависит от текущего узла. Иными словами, результатом обработки выражения XPath могут быть различные значения, которые зависят от того, где именно в сценарии XSLT встретилось это выражение. Рассмотрим, например, такой XML-код:

```
<doc>
  <title> Название документа </title>
  <title> Альтернативное название документа </title>
  Некоторый текст
</doc>
```

Если для трансформации этого кода используется следующий фрагмент сценария XSLT:

```
<xsl:template match = "doc">
  Первое название:
  <xsl:value-of select = "child::title[1]"/> <br/>
  Второе название:
  <xsl:value-of select = "child::title[2]"/> <br/>
  <xsl:apply-templates/>
</xsl:template>
```

то для выражения `child::title` текущим узлом будет элемент doc, для которого записано правило шаблона. Поэтому когда мы используем это выражение, процессор будет обрабатывать только те элементы **title**, которые являются непосредственными потомками doc.

Текущим не всегда является узел, описываемый значением атрибута match директивы `xsl:template`. Некоторые директивы XSLT могут менять один текущий узел на другой. Примером такой директивы может служить `xsl:for-each`:

```
<xsl:template match = "doc">
  <xsl:for-each select = "title">
    Заголовок №
```

```
  <xsl:number count = "title"/>:
  <xsl:value-of select = "."/> <br/>
 </xsl:for-each>
</xsl:template>
```

Здесь для пути адресации, состоящего только лишь из точки (напомним, что точка адресует текущий узел), текущим узлом является уже не doc, а title.

Путь адресации состоит из одного или нескольких шагов адресации, которые отделены друг от друга прямым слешем (/). На первом шаге адресации относительно текущего узла выбираются те узлы, которые этому шагу соответствуют. Например, результатом отбора выражением child::doc будут все элементы doc, которые являются непосредственными потомками текущего узла. В данном случае выражение child::doc представляет собой путь адресации, состоящий из одного шага.

Если немного усложнить это выражение, добавив к нему еще один шаг, например, child::title, то в результате мы получим выражение child::doc/child::title. Это выражение адресует все элементы title, которые являются непосредственными потомками элемента doc, который в свою очередь является непосредственным потомком текущего узла. Иными словами, на первом шаге адресации отбираются все непосредственные потомки текущего узла, и среди них выбираются элементы с именем doc. На втором шаге для каждого узла из набора узлов, полученного на первом шаге, снова строится список всех его непосредственных потомков, и в результирующий набор узлов попадают все элементы с именем title из объединенного списка непосредственных потомков всех отобранных элементов doc.

Формально алгоритм отбора узлов при обработке всей последовательности шагов адресации можно описать так:

1. Выбираются узлы, описанные на первом шаге адресации (child::doc). Выражение XPath, представляющее собой рассматриваемый путь адресации, читается слева направо. Таким образом, выражение child::doc является первым шагом адресации, а child::title — вторым.
2. Все узлы из набора узлов, полученных на первом шаге, последовательно становятся текущими, и к каждому из них применяется выражение, представленное вторым шагом (child::title). В результате получается набор узлов, соответствующий второму шагу адресации.
3. К полученному набору применяется правило третьего шага и т. д. Процесс продолжается до тех пор, пока не будут обработаны все шаги последовательности.

Абсолютный путь адресации отличается от относительного тем, что в качестве текущего узла первого шага адресации выбирается корень документа. Выражение, представляющее собой абсолютный путь, всегда начинается со слеша (/). Примером может быть конструкция `/child::doc/child::title` или, если мы воспользуемся сокращенным синтаксисом, — `/doc/title`. Первый слеш (/) обозначает корень документа, doc — корневой элемент или элемент документа, второй слеш — разделитель шагов адресации, и `title` — непосредственный потомок корневого элемента.

Все, что было сказано про относительные пути адресации, будет верным и для абсолютных за исключением того, что значение, возвращаемое в результате обработки абсолютного пути, не зависит от текущего узла или, иными словами, от местоположения в документе выражения, представляющего собой этот абсолютный путь адресации.

3.2.1. Шаги адресации

Рассмотрим теперь, из чего состоит шаг адресации. В качестве примера возьмем выражение `child::title[position()=1]`. В предыдущем подразделе, когда мы описывали обработку какого-либо шага адресации, мы последовательно рассматривали составляющие его части. Причем как и все выражение целиком, мы читали каждый шаг также слева направо. Итак, для выражения `child::title[position()=1]` будет проделано следующее. Сначала будут отобраны все узлы, являющиеся непосредственными потомками текущего, затем среди них будут выбраны элементы с именем `title`, и наконец, среди всех выбранных элементов `title` в качестве результирующего будет выбран первый в этом наборе. Таким образом, в конце обработки получим элемент `title`, который является первым непосредственным потомком текущего узла.

Каждое из трех проделанных нами действий соответствует одной из трех частей, на которые подразделяется выражение, представляющее собой шаг адресации. И каждая из этих частей имеет свое название. Рассмотрим их слева направо:

1. Первая часть представляет собой т. н. ось. В нашем случае это child - ось, адресующая всех непосредственных потомков текущего узла. Если бы нам понадобилось описать все, а не только непосредственные потомки текущего узла, мы могли бы воспользоваться осью descendant. Все оси, которые мы можем использовать в выражениях XPath, будут подробно описаны в следующем разделе данной главы.
2. После названия оси следует правило отбора (проверки) узлов, которое отделено от нее двойным двоеточием (::). В нашем выражении это правило

представлено именем элемента title, которое говорит о том, что среди всех узлов, определяемых осью, следует выбрать только те, которые являются элементами и носят имя title. Если бы мы захотели адресовать первый непосредственный потомок среди элементов с произвольными именами, то воспользовались бы выражением child::*[position()=1].

3. за правилом отбора следует один или несколько т. н. предикатов, каждый из которых заключен в квадратные скобки. В отличие от правила отбора узлов, предикат позволяет задать более специфические условия отсева, чем ограничения на тип и/или имя адресуемого узла. Предикат position()=1 говорит о том, что среди всех узлов необходимо выбрать только один — первый непосредственный потомок текущего узла. Для выбора, например, второго узла мы бы использовали предикат position()=2, а ДЛЯ последнего — position()=last().

Примечание

Иногда для описания элементов, занимающих определенное место в наборе, мы использовали более короткую форму записи предиката. Например, вместо child::title[position()=1] мы записывали child::title[1]. Такая форма записи связана с некоторыми особенностями предикатов *(см. разд. 3.2.4),* а не с правилами сокращенного синтаксиса.

Из описанных выше частей шага адресации обязательной является только вторая — правило проверки узлов. Предикаты можно опустить полностью. Если не указана ось, то будет использоваться ось по умолчанию — child. Кроме того, вместо названия оси можно воспользоваться конструкциями сокращенного синтаксиса. В том случае, когда выражение состоит только из одного шага адресации и путь адресации является относительным (то есть не начинается со слеша), оно может представлять собой один предикат, записанный без квадратных скобок. Результатом обработки

```
<xsl:value-of select = "position() &gt; 10"/>
```

будет булево значение true или false, преобразованное в строку и подставленное в конечный документ.

3.2.2. Оси

Ось описывает характер взаимоотношений между текущим узлом и тем набором узлов, который отбирается на данном шаге адресации. Например, ось ancestor строит дерево всех предков текущего узла, т. е. говорит о том, что на рассматриваемом шаге адресации следует выполнить такие действия:

1. Выбрать набор узлов (дерево), являющихся предками.
2. Относительно текущего узла построить дерево предков.

В этом подразделе мы опишем те оси, которые являются допустимыми в языке XPath. Для каждой из них мы приведем один или несколько примеров, которые нужно рассматривать следующим образом. Вместо того чтобы описывать какие-либо XML-фрагменты и соответствующие им сценарии XSLT, мы для каждого примерного выражения XPath предоставим XML-код, где полужирным шрифтом выделим те узлы, которые описываются данным выражением.

child

Отбор узлов, реализующийся с помощью этой оси, осуществляется среди непосредственных потомков текущего узла.

Пример. Выражение: `child::bbbbb` **Текущий узел:** aaaaa

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
</aaaaa>
```

Пример. Выражение: `child::*` **Текущий узел: aaaaa**

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
</aaaaa>
```

descendant

Представляет собой множество всех потомков текущего узла. Это множество состоит из всех непосредственных потомков текущего узла, всех непосредственных потомков непосредственных потомков текущего узла и т. д. Ось descendant не содержит атрибутов и узлов пространства имен.

Пример. Выражение: `descendant::*` **Текущий узел:** /

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
</aaaaa>
```

Пример. Выражение: `descendant::*` Текущий узел: ЬЬЬЬЬ

```
<aaaaa>
    <bbbbb>
        <ddddd>
            <eeeee/>
        </ddddd>
        <fffff/>
        <ggggg/>
    </bbbbb>
    <ccccc>
        <hhhhh/>
    </ccccc>
</aaaaa>
```

Пример. Выражение: `descendant::eeeee` Текущий узел: ЬЬЬЬЬ

```
<aaaaa>
    <bbbbb>
        <ddddd>
            <eeeee/>
        </ddddd>
        <eeeee/>
        <ggggg/>
    </bbbbb>
    <ccccc>
        <bbbbb>
            <eeeee/>
        </bbbbb>
    </ccccc>
</aaaaa>
```

parent

Адресует непосредственного предка (родителя) текущего элемента, если такой предок имеется.

Пример. Выражение: `parent::*` **Текущий узел: eeeee** 1

```
<aaaaa>
    <bbbbb>
        <fffff/>
        <ddddd>
            <eeeee/>
        </ddddd>
        <eeeee/>
    </bbbbb>
    <ccccc>
        <eeeee>
            <bbbbb>
                <eeeee>
                    <ggggg/>
                </eeeee>
            </bbbbb>
        </eeeee>
    </ccccc>
</aaaaa>
```

ancestor

Образует множество всех предков текущего узла, которое строится по аналогии с множеством потомков, адресуемых осью descendant. Оно состоит из родителя текущего узла, родителя для родителя текущего узла и т. д. Разница в построении этих множеств заключается только в том, что потомков произвольного типа у какого-либо узла может быть несколько, а родитель только один. Ось ancestor всегда включает в себя корень документа, кроме того единственного случая, когда корень является текущим узлом.

Пример. Выражение: `ancestor::*` **Текущий узел: /**

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
</aaaaa>
```

Пример. Выражение: `ancestor::*` **Текущий узел:** `fffff`

```
<aaaaa>
    <bbbbb>
        <ccccc>
            <eeeee/>
            <ggggg/>
        </ccccc>
        <ddddd>
            <fffff/>
        </ddddd>
        <hhhhh/>
    </bbbbb>
    <iiiii/>
</aaaaa>
```

Пример. Выражение: `ancestor::bbbbb` **Текущий узел:** `ddddd`

```
<aaaaa>
    <bbbbb>
        <ccccc>
            <bbbbb>
                <ddddd/>
            </bbbbb>
        </ccccc>
        <eeeee/>
        <fffff/>
    </bbbbb>
</aaaaa>
```

following-sibling

Адресует все следующие за текущим узлы, которые имеют с ним общего родителя. Если текущий узел является атрибутом или узлом пространства имен, то ось following-sibling **будет Пустой.**

Пример. Выражение: `following-sibling::*` **Текущий узел:** ccccc

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
    <ddddd/>
    <eeeee/>
    <fffff/>
</aaaaa>
```

Пример. Выражение: `following-sibling::ddddd` **Текущий узел:** ccccc

```
<aaaaa>
    <bbbbb>
        <ccccc/>
        <ddddd/>
        <eeeee/>
    </bbbbb>
    <fffff>
        <ddddd/>
        <ccccc/>
        <ggggg>
            <hhhhh/>
        </ggggg>
        <ddddd>
            <iiiii>
                <ddddd/>
            </iiiii>
        </ddddd>
        <ddddd/>
    </fffff>
</aaaaa>
```

preceding-sibling

Адресует все предшествующие текущему узлы, которые имеют с ним общего родителя. Если текущий узел является атрибутом или узлом пространства имен, то ось preceding-sibling будет пустой.

Пример. Выражение: `preceding-sibling::*` Текущий узел: ddddd

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
    <ddddd/>
    <eeeee/>
    <fffff/>
</aaaaa>
```

Пример. Выражение: `preceding-sibling::ddddd` Текущий узел: eeeee

```
<aaaaa>
    <bbbbb>
        <ccccc/>
        <ddddd/>
        <eeeee/>
    </bbbbb>
    <fffff>
        <ddddd/>
        <ccccc>
            </ddddd>
        </ccccc>
        <ddddd>
            <iiiii>
                <ddddd/>
            </iiiii>
        </ddddd>
        <eeeee/>
        <ddddd/>
    </fffff>
</aaaaa>
```

following

Описывает все узлы, располагающиеся после текущего, кроме прямых (но не непосредственных) потомков и предков текущего узла. В адресуемый

осью following набор узлов также не попадают атрибуты и узлы простра ства имен.

Пример. Выражение: following::* Текущий узел: ccccc

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
    <ddddd>
        <eeeee/>
    </ddddd>
    <fffff/>
</aaaaa>
```

Пример. Выражение: following::* Текущий узел: ccccc

```
<aaaaa>
    <bbbbb>
        <ccccc>
            <eeeee/>
            <fffff/>
        <ccccc/>
        <ddddd>
            <eeeee/>
        </ddddd>
    </bbbbb>
    <fffff>
        <ggggg/>
    </fffff>
    <hhhhh/>
</aaaaa>
```

preceding

Описывает все узлы, располагающиеся перед текущим, кроме прямых (н непосредственных!) потомков и предков текущего узла. В адресуемь осью preceding набор узлов также не попадают атрибуты и узлы пр странства имен.

Пример. Выражение: `preceding::*` Текущий узел: fffff

```
<aaaaa>
    <bbbbb/>
    <ccccc>
        <ddddd/>
    </ccccc>
    <eeeee/>
    <fffff/>
    <ggggg/>
</aaaaa>
```

! Пример. Выражение: `preceding::bbbbb` Текущий узел: fffff

```
<aaaaa>
    <bbbbb>
        <bbbbb/>
        <ccccc/>
    </bbbbb>
    <bbbbb/>
    <bbbbb>
        <ccccc>
            <eeeee/>
            <bbbbb/>
            <fffff/>
        </ccccc>
        <ddddd/>
        <bbbbb/>
    </bbbbb>
    <bbbbb/>
    <ggggg/>
</aaaaa>
```

Примечание

В определении осей preceding и following под прямыми предками и потомками следует понимать "кровные" родственные связи, которые можно представить в виде цепочки, состоящей из родственных отношений типа "отец-сын". Таким образом, например, родственное отношение деда к брату внука не будет прямым, а к самому внуку — будет.

attribute

В том случае, если текущий узел является элементом, описывает его атрибуты, иначе ось является пустой.

Пример. Выражение: `attribute::*` Текущий узел: ccccc

```
<aaaaa>
    <bbbbb attr1 = "value1"/>
    <ccccc
        attr1 = "value1"
        attr2 = "value2" />
    <ddddd/>
</aaaaa>
```

Пример. Выражение: `//attribute::attr1` Текущий узел: / (абсолютная адресация)

```
<aaaaa>
    <bbbbb attr1 = "value11" />
    <ccccc>
        <ddddd
            attr1 = "value21"
            attr2 = "value22" />
        </ddddd>
    </ccccc>
</aaaaa>
```

self

Описывает только сам текущий узел.

Пример. Выражение: `self::*` Текущий узел: ccccc

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
    <ddddd/>
</aaaaa>
```

descendant-or-self

Описывает текущий узел, а также всех его потомков. Представляет собой объединение наборов узлов, адресуемых осями descendant и self.

Пример. Выражение: `descendant-or-self::*` Текущий узел: ccccc

```
<aaaaa>
    <bbbbb/>
    <ccccc>
        <ddddd>
            <eeeee/>
        </ddddd>
    </ccccc>
</aaaaa>
```

Пример. Выражение: `descendant-or-self::*` Текущий узел: ccccc

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
    <ddddd>
        <eeeee>
            <ccccc/>
        <eeeee/>
    </ddddd>
    <fffff>
        <ддддд/>
        <ccccc/>
            <hhhhh/>
            <iiiii/>
        <ccccc/>
        <jjjjj/>
    </fffff>
</aaaaa>
```

ancestor-or-self

Описывает текущий узел, а также всех его предков. Представляет собой объединение наборов узлов, адресуемых осями ancestor и self.

Пример. Выражение: `ancestor-or-self::*` **Текущий узел:** ccccc

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
    <ddddd/>
</aaaaa>
```

Пример. Выражение: `ancestor-or-self::*` **Текущий узел:** ddddd

```
<aaaaa>
    <bbbbb/>
    <ccccc>
        <ccccc/>
        <ddddd/>
    </ccccc>
    <ccccc/>
    <ddddd/>
    <ccccc/>
    <eeeee>
        <fffff/>
        <ggggg>
            <ddddd>
                <hhhhh/>
            </ddddd>
            <iiiii/>
        </ggggg>
    </eeeee>
</aaaaa>
```

namespace

Описывает элементы того пространства имен, которому принадлежит текущий узел. Если текущий узел не является элементом, то ось namespace будет пустой.

Как мы уже говорили ранее, для достижения целей наших разработок мы будем создавать собственные пространства имен. Поэтому, а также в связи с тем, что реализация поддержки пространства имен, не описанных в спец

фикациях консорциума W3C, зависит от конкретного процессора, мы не б,дем определять эти пространства, а также описывать их использование.

Примечание

Оси ancestor, descendant, following, preceding и self относительно произвольно выбранного текущего узла все вместе адресуют все множество узлов документа за исключением атрибутов и узлов пространства имен. Причем наборы узлов, соответствующие каждой из этих осей, не пересекаются, т. е. не содержат общих узлов.

3.2.3. Проверка узлов

Во время обработки для каждой оси ставится в соответствие так называемый основной тип узлов, который определяется в зависимости от названия оси и правила отбора (проверки) узлов. Правило проверки узлов следует после названия оси и отделяется от нее двойным двоеточием (::). Как мы уже говорили ранее, это правило определяет не только тип отбираемых узлов, но также может задавать для них имя.

Рассмотрим такие правила проверки узлов, которые влияют только на тип, и в качестве примера возьмем ось child, адресующую непосредственных потомков текущего узла (табл. 3.1).

Таблица 3.1. *Правила проверки, опирающиеся на типы узлов*

Шаг адресации	Типы адресуемых узлов
child::text()	Текстовые узлы
child::comment()	**Комментарии**
child: : processing-instruction ()	**Инструкции обработки**
child: : node ()	**Узлы произвольного типа**

Когда правило проверки узлов не является одной из предопределенных конструкций text(), comment O, processing-instruction () ИЛИ node O, ТО основной тип узлов оси определяется следующим образом. Если набор узлов, адресуемых данной осью, может содержать элементы, то эта ось адресует только элементы, иначе она является одной из специфических осей: attribute или namespace, адресующих атрибуты или узлы пространства имен соответственно.

Иными словами, если в качестве правила проверки узлов мы возьмем имя узла, например, child: :title, **то все оси кроме** attribute И namespace будут адресовать элементы с именем title. Если же мы используем одну из

лексем, приведенных в табл. 3.1, то ось будет адресовать узлы соответствующего типа. Например, выражение `child::title/child::text()` будет адресовать текст заголовка, представленного элементом `title`.

Помимо описанных выше конструкций правило отбора узлов может представлять собой лексему, состоящую только из одного символа — звездочки (*). Эта лексема говорит о том, что ось должна описывать узлы с произвольным именем. При этом основной тип узлов определяется точно так же.

Напомним, что узлы дерева документа могут быть одним из семи типов (корень, элемент, атрибут, текстовый узел, узел пространства имен, инструкция обработки и комментарий). В этом подразделе мы описали, как из них выбирается основной тип для узлов, описываемых какой-либо осью. Единственный тип, который мы пропустили, — это корень. Но по определению корень также является элементом и, следовательно, адресуется точно так же, как и любой другой элемент.

3.2.4. Предикаты

После того как построен набор узлов, адресуемый какой-либо осью на данном шаге адресации, и из этого набора отобраны узлы в соответствии с правилом проверки узлов, из получившегося таким образом набора дополнительно отсеиваются те узлы, которые не удовлетворяют условиям, представленным т. н. предикатами. Предикат — это то, что записывается в квадратных скобках сразу после правила проверки узлов. Рассмотрим табл. 3.2, содержащую несколько примеров.

Таблица 3.2. *Примеры предикатов*

Выражение	Предикат	Значение
`child::item[position()=2]`	`position()=2`	Элемент item, являющийся вторым непосредственным потомком текущего узла
`child::item[2]`	2	То же, что и первый предикат
`child::item[last()]`	last()	Элемент item, являющийся последним непосредственным потомком текущего узла
`child::item[position() > 2]`	`position()>2`	Отличается от первого предиката тем, что описываются все потомки, начиная с третьего
`child::item[position() mod 2 = 1]`	`position() mod 2=1`	Все нечетные непосредственные потомки текущего узла

На каждом шаге адресации может присутствовать либо один, либо ни одного, либо несколько предикатов:

```
descendant::item[text()='some_text'][attribute::name='some_name']
```

С помощью предикатов, определенных в этой конструкции, можно отыскать такого потомка (не обязательно непосредственного), который имеет атрибут name со значением some_name и текстовый узел, являющийся первым непосредственным потомком item и содержащий текст some_text.

Оси делятся на прямые и обратные. Прямые оси описывают узлы по направлению от корня к листьям дерева, а обратные — в противоположном направлении. Иными словами, если ось описывает узлы, находящиеся в дереве ниже, чем текущий узел, то такая ось будет прямой, остальные оси (кроме self, которая описывает сам текущий узел, и поэтому не имеет значения, прямая она или обратная) являются обратными. Таким образом, оси ancestor, ancestor-or-self, preceding И preceding-sibling ЯВЛЯЮТСЯ обратными осями, все остальные — прямыми.

В прямых осях порядок узлов в адресуемом наборе имеет то же направление, что и их порядок следования в документе. Соответственно, в обратных осях порядок узлов имеет противоположное направление. Первый узел в наборе имеет номер 1, нулевого узла не существует. Рассмотрим следующий код

```
<list>
  <item1/>
  <item2/>
  <item3/>
  <item4/>
  <item5/>
</list>
```

Если мы выберем в качестве текущего узла элемент item3, то выражение following-sibling::*[1] будет адресовать элемент item5, а выражение Preceding-sibling::*[1] — элемент item2.

На этапе обработки значение предиката вычисляется для каждого узла из описываемого набора, которое затем приводится к типу boolean (булево выражение, принимающее какое-либо из двух значений — true или false). Например, предикат position()=5 будет принимать значение true только для узлов, занимающих пятую позицию в том наборе узлов, к которому применяется данный предикат. Если в результате вычисления значения предиката получается число, то значение true будет возвращено для тех узлов, которые имеют в рассматриваемом наборе номер, равный этому числу.

ПОЭТОМУ выражения `child::item[position()=2]` И `child::item[2]` будут идентичными, и если в данном наборе непосредственных потомков имеется всего два узла, то им будут также идентичны еще два выражения `child::item[position()=last()]` И `child::item[last()]`.

Примечание

В том случае, когда предикат содержит сравнение каких-либо двух величин, представленных конструкциями XPath, результат вычисления значения такого предиката зависит от того, какой тип имеет каждая из сравниваемых величин (число, строка, булево выражение или набор узлов). Использующийся при этом механизм приведения типов несколько специфичен, он описывается в *разд. 3.3.*

В том случае, когда значение предиката не является булевым значением или числом, оно также приводится к типу boolean. Приведение к этому типу осуществляется так, как если бы к значению предиката мы применили функцию boolean о *(см. разд. 3.5.3).* Например, выражение chapter [title будет описывать элемент с именем chapter, являющийся непосредственным потомком текущего узла, причем chapter должен в качестве непосредственного потомка иметь элемент **title**.

Если мы условно разделим шаг адресации на две части, вторая из которых представляет собой предикаты, а первая — все остальное, то можно будет сказать, что условие, определяемое второй частью, накладывается на набор узлов, адресуемый первой частью, а не только лишь на тот набор, который представляет собой правило проверки узлов. Поэтому, например, выражение `/descendant::div/descendant::title[2]` будет описывать те элементе **title**, которые представляют собой вторые потомки в *любом* поколении для каждого вхождения элемента div. Значит, предикат применяется ко всей конструкции `descendant::title` целиком, а не к одному **title**, и было бы неправильным утверждать, что среди всех потомков с именем **title** выбираются только те, которые являются вторыми *непосредственными* потомками для своего родителя с произвольным именем.

Принцип применения предикатов тесным образом зависит от того, какие конструкции мы используем для записи шага адресации. Эти тонкости мы опишем в соответствующих разделах. В частности, обработка предикатов в зависимости от типа используемой оси, будет описана в *разд. 3.3,* применение сокращенного синтаксиса, также влияющего на состав набора узлов, описываемых выражением с предикатами, рассматривается в следующем подразделе.

3.2.5. Сокращенный синтаксис

Те выражения XPath, которые состоят из трех или более шагов адресации, являются слишком громоздкими. Для того чтобы избежать слишком длин-

ных конструкций, в некоторых ситуациях можно использовать предусмотренный в языке XPath сокращенный синтаксис.

Приведем несколько примеров из спецификации XPath:

- para — адресует непосредственный потомок текущего узла с именем para;
- * — адресует все элементы, являющиеся непосредственными потомками текущего узла;
- text() — все текстовые узлы, являющиеся непосредственными потомками текущего узла;
- @name — атрибут name текущего узла;
- @* — все атрибуты текущего узла;
- para[1] — первый непосредственный потомок текущего узла с именем para;
- para[last()] — последний непосредственный потомок текущего узла с именем para;
- */para — все потомки текущего узла во втором поколении, имеющие ИМЯpara;
- /doc/chapter[5]/section[2] — элемент section, ЯВЛЯЮЩИЙСЯ вторым непосредственным потомком элемента chapter, являющегося пятым непосредственным потомком элемента документа doc;
- chapter//para — элементы para, являющиеся потомками в любом поколении элемента chapter, который является непосредственным потомком текущего узла;
- //para — все элементы para в документе;
- //olist/item — все элементы item, являющиеся непосредственными потомками находящихся где-либо в документе элементов olist;
- . — текущий узел;
- .//para — элемент para, являющийся потомком в любом поколении текущего узла;
- .. — родительский узел (непосредственный предок текущего узла);
- ../@lang — атрибутlang, принадлежащий родительскому элементу;
- Para[@type = 'warning'] — все элементы para, являющиеся непосредственными потомками текущего узла и имеющие атрибут type со значениемwarning;
- Para[@type = 'warning'][5] — **Элемент** para, ЯВЛЯЮЩИЙСЯ ПЯТЫМ непосредственным потомкам текущего узла и имеющий атрибут type со значением warning;

- para[5] [@type = 'warning'] — то же, что и предыдущее выражение, отличие заключается в порядке отбора узлов (сначала находится пятый непосредственный потомок, а затем проверяется наличие у него атрибута type и его значение);
- chapter[title="Introduction"] — **элемент** chapter, **ЯВЛЯЮЩИЙСЯ** непосредственным потомком текущего узла и, в свою очередь, имеющего непосредственного потомка, который является элементом с именем **title** и чье текстовое значение вычисляется *(см. разд. 2.8)* как Introduction;
- chapter[title] — элемент chapter, являющийся непосредственным потомком текущего узла и имеющий хотя бы одного непосредственного потомка, представляющего собой элемент с именем **title**;
- employee[@secretary and @assistant] адресует элемент employee. ЯВЛЯЮЩИЙСЯ непосредственным потомком текущего узла и имеющий два атрибута С именами secretary И assistant.

Наиболее часто применяемым правилом сокращенного синтаксиса является возможность опустить название оси child, которая используется по умолчанию. Таким образом, выражения child::list/child::item И list/item являются эквивалентными. Здесь адресуются те элементы item, которые являются непосредственными потомками элемента **list**.

Как мы уже говорили ранее, с помощью звездочки (*) можно выделять все те узлы, которые соответствуют основному типу используемой оси. Например, выражение attribute::* адресует все атрибуты текущего узла. Но звездочку можно использовать без указания оси, и, т. к. осью по умолчанию является child, адресующая узлы типа "элемент", следовательно, звездочка будет описывать все элементы, представляющие собой непосредственные потомки текущего узла. В частности:

- list/* адресует все элементы с произвольными именами, являющиеся непосредственными потомками элемента **list**;
- list/*/item описывает такие элементы item, которые являются потомками **list** во втором поколении.

Таким образом, звездочка представляет собой сокращение для конструкции node().

ТОЧНО Так Же ДЛЯ КОНСТРУКЦИЙ text(), comment(), processing-instruction() и node(), записанных без указания оси, осью по умолчанию будет child. Например, выражение item/text() адресует первый непосредственный потомок элемента item, который является текстовым узлом.

Сокращением ДЛЯ КОНСТРУКЦИИ /descendant-or-self::node()/ является двойной прямой слеш (//). Например, выражение //title описывает эл

мент title, находящийся где-либо среди потомков текущего узла. Если title является единственным шагом адресации, то такое выражение адресует элемент title, находящийся где-либо в документе, причем если title является корневым элементом, то он также будет включен в адресуемый набор узлов. В результате обработки пути адресации doc//title, который эквивалентен doc/descendant-or-self::node()/child::title, будут описаны все элементы title, находящиеся где-либо среди потомков элемента doc.

Примечание

Предикаты, используемые вместе с выражениями // и /descendant-or-self, будут отсеивать разные наборы узлов. В частности, выражения //title[2] и /descendant-or-self::title[2] будут описывать все элементы **title**, находящиеся где-либо среди потомков текущего узла и являющиеся для своего родителя вторым непосредственным потомком. Вторая конструкция, соответственно, адресует все те элементы **title**, которые являются вторыми, не обязательно непосредственными потомками для каждого элемента, представляющего каждое вхождение текущего узла. Так происходит в силу того, что, как мы уже говорили ранее, предикат отсеивает узлы, отталкиваясь от оси, и в данном случае конструкции //title[2] и //child::**title**[2] будут эквивалентны. Таким образом, в первой конструкции предикат будет действовать в соответствии с определением оси **child**, а во второй — оси descendant-or-self.

Если необходимо адресовать текущий узел, то вместо пути адресации self::node() можно использовать точку (.). Точка точно так же, как и self::node(), описывает текущий узел произвольного типа.

Две точки, стоящие подряд (..), эквиваленты выражению parent::node(). Сокращения для текущего (.) и родительского (..) узлов можно эффективно использовать в сочетании с //.

Выражение ..//* будет адресовать тот же набор узлов, что и

```
parent::node()/descendant-or-self::node()/child::node()
```

Для выражения .//title эквивалентным будет

```
self::node()/descendant-or-self::node()/child::title
```

При адресации атрибутов вместо оси attribute можно использовать символ @. Таким образом, выражения attribute::name и @name будут адресовать одно и то же. Вместе с @ можно употреблять звездочку. Конструкция @* будет описывать все атрибуты текущего узла, являющегося элементом. Выражение chapter[@lang] будет описывать элементы chapter, имеющие атрибут lang. Выражение chapter[@lang = 'RU'] будет описывать элементы chapter с атрибутом lang, который имеет значение RU. Для последнего пути адресации эквивалентом, записанным по правилам полного синтаксиса, будет конструкция child::chapter[attribute::lang = "RU"].

3.3. Выражения

Выражения языка XPath в результате обработки могут возвращать значения четырех типов — число, строка, булево значение, набор узлов. В последнем случае выражение XPath называется путем адресации. Если же происходит вычисление значения этого набора с последующим его приведением к одному из первых трех типов, то тогда это просто выражение.

Выражения и пути адресации отличаются также по своему предназначению. Пути адресации, как было сказано раньше, служат для описания или адресации некоторого множества поддеревьев исходного дерева. Эта адресация нужна для того, чтобы поставить в соответствие данному множеству, представляющему собой некий набор узлов, некоторую последовательность команд, которая должна быть выполнена либо для каждого узла в этом наборе, либо над всей совокупностью узлов. Гибкость путей адресации заключается в том, что у нас нет необходимости переписывать одни и те же шаблоны или их фрагменты для каждого узла, к которому они должны быть применены. Вместо этого мы можем указать все эти узлы сразу и обрабатывать всю их совокупность целиком.

Выражения, в отличие от путей адресации, предназначены для генерирования текста и подстановки его в конечный документ.

Примечание 3

Напомним еще раз, что пути адресации также являются выражениями XPath, и их отличия друг от друга весьма условны и определяются контекстом, в котором используется выражение XPath. Например, XPath-выражение `descendant::title` при использовании в конструкции `<xsl:value-of select = "descendant::title"/>` представляет собой выражение, а в конструкции `<xsl:apply-templates select = "descendant::title"/>` — путь адресации.

Здесь под текстом понимается не значение строкового типа, а часть результирующего дерева, которая может представлять собой разметку, символьные данные или и то, и другое.

Генерируемый таким образом текст может выступать в качестве имен узлов результирующего дерева (создание разметки):

```
<xsl:template match = "element_creation">
  <xsl:element name = "{@name}">
</xsl:template>
```

В этом примере будет создан элемент с именем, которое представляет собой значение атрибута name элемента `element_creation` исходного дерева. Точно так же может быть создан узел произвольного типа, например, атрибут с помощью директивы `xsl:attribute`.

Для создания символьных данных выражения используются в качестве значения атрибута select директивы xsl:value-of или в том случае, когда эта директива применяется для определения значения атрибутов результирующего документа и не может быть записана непосредственно *(см. разд. 2.8)*. Выражения, генерирующие значение атрибута, заключаются в фигурные скобки:

```
<div>
  <xsl:value-of select = "//element_creation/text()">
</div>
<p style = "{/doc/styles/para_style/@bold_para}">
  <xsl:apply-templates/>
</p>
```

Кроме директивы xsl:value-of, для генерирования символьных данных пригодны и некоторые другие директивы, например, xsl:number или xsl:message.

В выражениях точно так же, как и при записи путей адресации, могут быть использованы оси, правила проверки узлов и предикаты. Если в предикатах есть переменные, то во время обработки их идентификаторы будут заменены соответствующими значениями:

```
<xsl:variable name="style" select="text-indent:1.5em"/>
<xsl:variable name="company_name">"My company"</xsl:variable>
<div style="$style">
  <xsl:apply-templates/>
  <p>Примечание: авторские права принадлежат компании
    <xsl:value-of select = "$company_name"/>
  </p>
</div>
```

Результатом обработки этого фрагмента будет:

```
<div style="text-indent:1.5em">
  Текст, представленный в XML-файле
<p Примечание: авторские права принадлежат компании My company</p>
</div>
```

Если при этом для какого-либо идентификатора переменной не определено ее значение, фиксируется ошибка.

Помимо переменных, в выражениях допускаются вызовы функций, которые представлены своим именем и следующим за ним списком атрибутов в скобках:

```
<xsl:value-of select = "item[position0=2] "/>
```

Ошибка при использовании функций может возникнуть в каком-либо из трех случаев.

- Имя вызываемой функции не принадлежит к списку основных или дополнительных функций. Пользовательские функции средствами XML, XSLT или XPath создать невозможно, но какие-либо реализации парсеров такие функции определять могут.
- Количество аргументов функции не совпадает с тем, которое определено в спецификации.
- Тип какого-либо аргумента не соответствует заданному в спецификации и не может быть к нему приведен. Приведение типов осуществляется так *(см. разд. 3.5):*
  - к строковому типу — функцией string();
  - К ЧИСЛОВОМУ — number();
  - к булеву значению — boolean().

3.3.1. Наборы узлов

Когда в качестве выражения XPath используется путь адресации, результатом обработки такого выражения является набор узлов. К типу "набор узлов" не приводится ни один из остальных типов, определенных в языке XPath (число, строка, булево значение).

Для объединения наборов узлов, представляющих собой поддеревья исходного дерева, служит оператор | (вертикальная черта). В следующем примере

```
<xsl:template match="//chapter/body | //part/body | //appendix/body">
  <xsl:apply-templates
       select = "abstract/descendant::* |
       term[text()=preceding::term/text()]"/>
  <!-- шаблон -->
</xsl:template>
```

представлено правило шаблона для такого элемента body, который является непосредственным потомком какого-либо из элементов chapter, part или appendix, находящихся где-либо в документе. Здесь выражение представляет собой объединение трех наборов узлов, каждый из которых состоит только из одного узла — элемента body.

Выражение, которое является значением атрибута select директивы xsl:apply-templates, состоит из двух наборов узлов, причем каждый из этих наборов описывает соответствующее поддерево исходного документа

Первый адресует дерево потомков элемента abstract, второй — все, предшествующие текущему элементу body, элементы term, обозначающие термины, использующиеся в контенте текущего элемента body и описанные ранее в тексте документа (то есть элементы term, предшествующие данному body, такие, что их строковое значение совпадает со строковым значением какого-либо из элементов term, встречающихся в текущем элементе body). Например, для XML-кода:

```
<body>
  <term>term1</term>
  <term>term2</term>
  <term>term3</term>
</body>
<body>
  <term>term4</term>
  <term>term2</term>
  <term>term1</term>
</body>
```

элементами ИЗ набора узлов `term[text()=preceding::term/text()]` будут второй и третий элементы term (содержащие текст term2 и term1 соответственно), находящиеся во втором элементе body, который является текущим в нашем примере (он будет текущим, например, в случае применения конструкции `<xsl:apply-templates select="//body[2]"/>`).

Точно так же, как и в путях адресации, в выражениях могут использоваться *I* и //, которые в этом случае называются операторами. Символ / служит в качестве разделителя для шагов адресации, а также для адресации корня. Двойной слеш является сокращением для выражения `/descendant-or-self::node()/`.

Примечание

При применении в выражениях предикатов большое значение имеет тип используемой оси (напомним, что оси бывают прямые и обратные). Например, выражение `preceding::item[1]` будет возвращать первый элемент item среди всех описываемых элементов с этим именем и расположенных в *обратном* порядке относительно документа. Иными словами, в силу того, что порядок адресуемых данной осью элементов будет *обратным,* в результате будет адресован элемент item, ближайший к текущему узлу, т. е. *последний в* списке с *прямым* направлением. Если же мы перепишем это выражение как (preceding: : item) [1], то в результате получим *первый* элемент item в *прямом* списке одноименных элементов, т. к. элементы в этом списке располагаются точно так же, как в документе, а не в обратном порядке.

3.3.2. Булевы значения

Предикат или выражение XPath целиком (в том случае, когда оно состоит только лишь из одного предиката) во время обработки приводится к типу "булево значение". Это приведение типа производится так, как если бы для выражения была вызвана функция `boolean()` *(см. разд. 3.5.3)*. Значение, возвращаемое этой функцией, всегда равно true или false.

Предикат может содержать операторы, которые мы для удобства разобьем на три группы: логические операторы (or, and и функция not()), операторы сравнения (= и !=) и операторы, реализующие отношения порядка (>, <, >= и <=). Приведем описание каждой группы.

Логические операторы

- Обработка выражения or сводится к приведению типа каждого операнда к типу boolean с последующей его проверкой на истинность. Если хотя бы один из операндов имеет значение true, то и результат обработки or также имеет значение true. Обработка операндов производится последовательно слева направо, и если встретился операнд, имеющий значение true (после приведения его типа), то дальнейшая обработка не производится и в качестве результата возвращается true, в противном случае возвращается false.
- Обработка выражения and сводится к приведению типа каждого операнда к типу boolean с последующей его проверкой на истинность. Если хотя бы один из операндов имеет значение false, то и результат обработки and также имеет значение false. Обработка операндов производится последовательно слева направо, и если встретился операнд, имеющий значение false (после приведения его типа), то дальнейшая обработка не производится и в качестве результата возвращается false. в противном случае возвращается true.
- Роль оператора not в языке XPath играет функция not() *(см. разд. 3.5.3)*. Иными словами, в XPath оператор not отсутствует, но иногда мы будем называть оператором соответствующую функцию.

Операторы сравнения

В том случае, когда используется оператор сравнения (= и !=), перед тем, как произвести операцию сравнения, процессор пытается привести тип каждого из операндов к типу "набор узлов". Если для какого-либо операнда это не удается сделать, производится приведение к типу булева значения. Если и это не удается, то приведение типа производится к числовому типу и, в случае неудачи, — к строковому типу.

Если оба операнда могут быть приведены к типу набора узлов, то при применении оператора = будет возвращено значение true тогда, когда оба набора узлов содержат узлы, которые, будучи приведены к типу булева значения (как если бы для них была вызвана функция boolean о), будут иметь одинаковые значения. Если подобных узлов не нашлось, производится поиск таких узлов, чьи значения совпадают после их приведения к числовому типу (как если бы для них была вызвана функция number о). Если такие узлы нашлись, то возвращается true и поиск прекращается, иначе будет произведена попытка найти в наборах такие узлы, у которых совпадают строковые значения (как если бы для них была вызвана функция string о). Если попытка оказывается неудачной, то операция сравнения заканчивает работу, и в качестве результата возвращается false.

Если какой-либо из операндов не является набором узлов, то производится приведение его типа так, как было описано выше. Рассмотрим те случаи, когда только один из операндов представляет собой набор узлов, а второй — нет.

- Если один из операндов имеет тип boolean, а другой — набор узлов, то результатом сравнения этих операндов будет true только тогда, когда в этом наборе узлов имеется хотя бы один узел, чье значение приводится к типу boolean и совпадает со значением второго операнда.
- Если один из операндов имеет числовой тип, а другой — набор узлов, то результатом сравнения этих операндов будет true только тогда, когда в этом наборе узлов имеется хотя бы один узел, чье значение приводится к числовому типу и совпадает со значением второго операнда.
- Если один из операндов не приводится ни к типу набора узлов, ни к булеву значению, ни к числовому типу, то его значение преобразуется в строку. Результатом сравнения набора узлов со строкой будет true только в том случае, когда среди данного набора имеется хотя бы один узел, чье строковое значение совпадает со значением второго операнда.

Примечание

Тот факт, что при сравнении набора узлов со значением любого типа для истинности результата этого сравнения достаточно только одного узла из этого набора, чье значение после приведения типа совпадает со значением второго операнда (или значением какого-либо узла из второго набора атрибутов), очень важен. Значение true можно получить при сравнении весьма различающихся наборов узлов или некоего набора узлов, например, с числом. В частности, набор узлов, представляющий собой все узлы исходного дерева //* будет равен пустому символу, если документ содержит хотя бы один пустой элемент.

Оператор !=, который читается как "не равно", является обратным по отношению к = и возвращает true только в том случае, когда = возвращает false, и наоборот, != возвращает false только тогда, когда = возвращает true.

Примечание

Рассмотрим пример из спецификации, который очень наглядно иллюстрирует операцию сравнения, применяя для двух, казалось бы, идентичных выражений операторы = и !=. Пусть $x — набор узлов. Следующие два выражения не эквивалентны: $x = 'foo' и not($x != 'foo'). Первое из них возвращает true только в том случае, когда *хотя бы один* узел из набора $x имеет строковое значение foo. Второй возвращает true, когда все узлы в этом наборе имеют строковое значение foo.

Отношения порядка

Мы решили выделить эти операторы в отдельную группу, т. к. сравниваемые значения могут иметь только числовой тип. Если в выражении встречается оператор <, то true будет возвращено только в том случае, когда результат преобразования в число первого операнда будет меньше результата преобразования в число второго операнда. Оператор <= возвращает true только тогда, когда первое значение меньше либо равно второму. Операторы > и >= возвращают true, когда < и <= для тех же операндов возвращают false.

По правилам синтаксиса языка XML операторы, реализующие отношения порядка, должны быть записаны с использованием соответствующих ссылок на предопределенные именные сущности *(см. разд. 1.11.1)*. Вместо < нужно использовать <, а вместо > — >. Исключения составляют те случаи, когда знаки "меньше" и "больше" находятся в секциях CDATA или в контенте директивы xsl:text, ДЛЯ которой атрибут disable-output-escaping имеет значение yes. Таким образом, вместо x < y следует писать x < y.

Согласно спецификации все операторы (логические, сравнения и реализующие отношения порядка) имеют приоритет, определяющийся их положением в следующем далее списке:

or

and

=, !=

<=, <, >=, =

Оператор, стоящий первым в списке, имеет наименьший приоритет, последний — наивысший.

Все вышеперечисленные операторы имеют ассоциативность слева. Это означает, что если выражение содержит несколько операторов одинакового приоритета, то сначала рассматривается первый из них. Разбор такого выражения происходит слева направо. Для двух операндов, соответствующих этому оператору, вычисляется результат его применения. Далее этот результат, имеющий тип булева значения, сравнивается с третьим операндом. Т

...ом используется второй оператор. Процесс продолжается либо до тех пор, ...ка не будет разобрано все выражение, либо (в случае операторов or и and) ...тех пор, пока для какого-нибудь из операндов не будет получено true или ...lse соответственно.

Например, выражение 3>2>1 будет обработано, как (3>2)>1, и его результатом ...ет false, т. к. выражение 3>2 вернет true, которое после преобразования в число *(см. разд. 3.5.4)* станет равным 1, и таким образом 1>1 вернет false.

3.3.3. Числа

Числа в языке XPath могут принимать любое положительное и отрицательное значение двойной точности с плавающей точкой в 64-битном диапазоне нуля. Кроме этих значений число может быть положительной или отрицательной бесконечностью (строковое значение infinity), положительным или отрицательным нулем (строковое значение 0), а также принимать специальное значение "не число" — NaN (сокращение от Not-a-Number, строковое значение NaN).

Для чисел определены унарные и бинарные операторы вычитания (-) и сложения (+), а также бинарные операторы умножения (*), деления (div) и вычисления остатка от усекающего деления (mod).

В табл. 3.3 приведены примеры обработки различных числовых выражений **Директивой** `xsl:value-of`.

Таблица 3.3. *Примеры обработки чисел с помощью директивы* `xsl:value-of`

Выражение	Результат вычисления
-2	-2
-2+3	1
2*3	6
6 div 3	2
5 mod 2	1
5 mod -2	1
-5 mod 2	-1
-5 mod -2	-1

3.3.4. Строки

Строковый тип является самым простым в языке XPath. Все то, что не может быть приведено к какому-либо другому типу, преобразуется в строку.

Исключения составляют те символы, которые процессор рассматривает как специальные (например, угловые скобки), а также символы, не включенные в набор допустимых для данного процессора либо его разработчиком, либо пользователем. Такими символами могут быть, например, буквы русского языка, если не подключена или не установлена по умолчанию соответствующая кодировка.

3.3.5. Лексическая структура

В языке XPath в качестве служебных используются некоторые символы, которые могут интерпретироваться неоднозначно, например, они могут быть использованы для задания имен узлов исходного или конечного деревьев, что допускается спецификацией XML. Далее приведем перечень рекомендаций и правил, устраняющих эту неоднозначность.

1. Оператору вычитания, как правило, должен предшествовать пробельный символ, т. к. знак "минус" может быть частью имени узла. При этом выражение x-y будет интерпретироваться как имя узла, а x - y — как операция вычитания.
2. Символ * не является оператором умножения, когда предыдущей лексемой будет какой-либо оператор или один из следующих символов 8, `::`, `(`, `[` (звездочка служит также сокращением для `node()`).
3. Если за набором символов, состоящим из букв и цифр, следует (сразу или через пробельный символ) символ `(`, то такая лексема должна рассматриваться как вызов функции (скобки могут быть также использованы для указания приоритета разбора выражения XPath, например, выражения `ancestor::item[1]` И `(ancestor::item)[1]` адресуют разные узлы, если у текущего элемента предок с именем item не единственный).
4. Если за набором символов, состоящим из букв и цифр, следует (сразу или через пробельный символ) символ `::`, то такая лексема должна распознаваться как имя оси.
5. Во всех остальных случаях лексема не должна рассматриваться как **обозначение Типа узла** (`text()`, `comment()`, `processing-instruction()`, `node()`), оператор *(см. разд. 3.3.2 и 3.3.3),* имя функции или имя оси.

3.4. Образцы

Еще одним подтипом выражений являются так называемые образцы (patterns). Они предназначены для использования вместе с правилами шаблона (`xsl:template`), для нумерации (`xsl:number`) и при декларации

ключей (xsl:key). Образцы представляют собой выражения XPath, отвечающие некоторым ограничениям, и интерпретируются несколько иначе.

Три вышеперечисленные директивы в качестве значений своих атрибутов (для xsl:template И xsl:key ЭТИМИ атрибутами является match, а ДЛЯ xsl:number — это атрибуты count и from) используют пути адресации, с помощью которых устанавливается соответствие между этими путями и теми узлами исходного дерева, которые им удовлетворяют. После того, как это соответствие установлено, данные узлы трансформируются с помощью соответствующих директив. Таким образом, можно сформулировать первое ограничение, которому отвечают образцы: результатом их обработки всегда будет набор узлов.

Когда мы записываем, например, такую конструкцию <xsl:template match - "some_node"/>, то этим мы указываем процессору, что данное правило шаблона должно быть применено к таким узлам, которые соответствуют образцу some_node, причем эти узлы могут находиться где угодно в исходном дереве. Это происходит в силу того, что для образцов понятие текущего узла не имеет смысла (но имеет смысл понятие текущего элемента). Или иными словами, если где-либо в документе существует такой узел, для которого образец, интерпретируемый как путь адресации относительно этого узла, будет описывать непустой набор узлов, то искомым будет первый узел из этого набора. Например, для XML-кода

```
<aaaaa>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <occcc/>
</aaaaa>
```

и образца ccccc такими узлами будут aaaaa и bbbbb. Иначе говоря, если мы Рассмотрим образец ccccc как относительный путь адресации, то в документе существуют такие узлы (aaaaa и ьььььь), относительно которых (полагая их текущими), он будет адресовать элементы ccccc.

Примечание

Образец можно рассматривать как путь адресации, если к этому образцу добавить слева //. Например, образец ccccc будет эквивалентен пути адресации //ccccc. То есть можно сказать, что образец — это такой путь адресации, к которому процессор по умолчанию добавляет слева двойной слеш.

Точно так же, как и пути адресации, образцы разбиваются на шаги адресации. При этом на каждом шаге адресации допускаются только две оси: child И attribute. И ХОТЯ ОСЬ descendant-or-self ИСПОЛЬЗОВать нельзя,

допустимо выражение //. Для отделения друг от друга шагов адресации также применяется /. На наш взгляд ограничение на оси введено для того, чтобы если не однозначно, то хотя бы максимально жестко идентифицировать те узлы, к которым применяется соответствующая директива.

Действительно, если бы такие оси, как, например, descendant или ancestor, были допустимыми в образцах, то процессору было очень трудно разрешать проблемы дублирования деклараций. Наборы узлов, адресуемые с использованием подобных осей относительно различных текущих узлов как множества могут иметь весьма сложные пересечения. Для каждой из вышеперечисленных трех директив и для каждого узла из этого пересечения соответственно, процессор должен был бы сложным образом вычислять их приоритет для корректного применения этих директив.

Как нам кажется, по этой же причине в качестве пары имя оси::правило проверки узлов, представляющей собой один шаг адресации образца, не допускается ссылка на переменную. Поэтому запись

```
<xsl:key match = "$var" use = "expression"/>
```

является примером неверной записи директивы, декларирующей ключ.

Образцы могут содержать один или несколько предикатов, на которые не накладывается никаких специфических для образцов ограничений. Иными словами, предикаты в образцах используются и интерпретируются точно так же, как и в выражениях и путях адресации.

Примечание

В предикатах, использующихся вместе с образцами, допускаются оси с произвольными именами (определенные в спецификации XPath), а также ссылки на переменные.

Например, образец

```
part//note [position() = $var]
```

будет допустимым, если он соответствует таким элементам note, которые встречаются где-либо в документе, имеют среди своих предков элемент part и в списке непосредственных потомков своего родителя имеют номер позиции, равный значению переменной var.

При записи образцов может быть использован символ вертикальной черты (|), обозначающий альтернативу. Результатом обработки таких образцов будет объединение результатов обработки каждой альтернативы по отдельности. При этом каждая альтернатива рассматривается как отдельный образец, т. е. должна удовлетворять всем вышеперечисленным требованиям для образцов. Если какая-либо из альтернатив не соответствует ни од-

ному из узлов исходного дерева, то результатом ее обработки будет пустой набор узлов, что не сделает всю совокупность альтернатив недопустимым образцом.

Если ли образец начинается с функции, то этими функциями могут быть только `id()` и `key()`, остальные функции недопустимы. Все шаги адресации образца кроме первого не могут начинаться ни с какой функции.

3.5. Основные функции

функции, описанные в этом разделе, согласно спецификации XPath, должны быть обязательно представлены в любой реализации синтаксического анализатора.

Так как спецификация не предусматривает создания пользовательских функций, мы не будем описывать синтаксис какой-либо абстрактной функции, а просто перечислим их все. Здесь мы скажем только, что синтаксис каждой определенной функции строится следующим образом. Сначала записывается имя функции, далее в круглых скобках следует список ее аргументов. Если у функции имеются аргументы, то их значения должны быть приводимы к тому типу, который указан в описании функции, в противном случае иногда может возникать ошибка (пример возникновения подобной ошибки см. в *разд. 3.3.4)*.

Везде далее при определении какой-либо функции мы будем указывать сначала тип возвращаемого ею значения, набранный курсивом, далее — имя функции, затем в круглых скобках список типов ее аргументов, набранных также курсивом. Причем

- *node-set* — будет соответствовать типу "набора узлов";
- *string* — строковому типу;
- *number* — числовому типу;
- *boolean* — типу "булево значение".

3.5.1. Функции обработки набора узлов

Функция *number last()*

Возвращает число, равное количеству узлов, расположенных на том уровне дерева, на котором находится текущий узел. Выражением XPath, описывающим этот набор узлов, будет `preceding-sibling::*|following-sibling::*|self::*`.

Пример. Выражение: `bbbbb[last()]` **Текущий узел:** aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb/>
</aaaaa>
```

Пример. Выражение: `descendant::ccccc[last()]` **Текущий узел:** bbbbb

```
<aaaaa>
    <bbbbb>
        <ccccc/>
        <ccccc>
            <ddddd>
                <ccccc/>
            </ddddd>
        </ccccc>
    </bbbbb>
    <ccccc/>
</aaaaa>
```

Функция *number* **position()**

Возвращает число, равное номеру позиции текущего узла в списке узлов, представляющем собой те узлы, которые расположены на том уровне дерева, на котором находится сам текущий узел. Выражением XPath, описывающим этот набор узлов, будет `preceding-sibling::*|following-sibling::*|self::*`. Вызов функции `position()` для данного набора узлов будет эквивалентен вызову функции `last()` для набора узлов, описываемых **выражением** `preceding-sibling::*|self::*`.

Пример. Выражение: `bbbbb[position()=2]` **Текущий узел: aaaaa**

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb/>
</aaaaa>
```

Пример. Выражение: bbbbb[position()=last()] **Текущий узел: aaaaa**

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb/>
</aaaaa>
```

Пример. Выражение: bbbbb[position() mod 2 = 1] **Текущий узел: aaaaa**

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb>
        <ddddd/>
    </bbbbb>
    <bbbbb/>
    <bbbbb/>
    <bbbbb/>
</aaaaa>
```

Функция *number count(node-set)*

Возвращает количество узлов основного типа в наборе, представленном аргументом этой функции. Данная функция подсчитывает все узлы набора, т. е. безотносительно каких-либо родственных отношений, связывающих эти узлы как друг с другом, так и с остальными узлами дерева.

Для предыдущего примера директива

```
<xsl:value-of select = "count(bbbbb[position() mod 2 = 1])"/>
```

сгенерирует значение 3, а результатом обработки

```
<xsl:value-of select = "count(*)"/>
```

будет 6.

Пример. Выражение: bbbbb [count (descendant::*) > 1]
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc/>
        <ccccc/>
    </bbbbb>
    <bbbbb/>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb>
        <ccccc>
            <ddddd/>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Функция *string local-name(node-set)*

Возвращает локальную часть расширенного имени. Расширенное имя представляет собой совокупность собственно имени (которое в этом контексте называется локальным) и пространства имен, которому оно принадлежит. Так как мы не собираемся создавать собственные пространства имен, а также производить какие-либо действия над теми узлами, для которых заданы пространства имен отличные от http://www.w3.org/1999/XSL/Transform, везде далее дай получения имени узла мы будем пользоваться вызовом функции name().

Функция *string namespace-uri(node-set)*

Возвращает URI (уникальный идентификатор ресурса) из расширенного имени узла, который встретится первым в наборе, представленном аргументом функции. В следующих случаях namespace-uri() возвращает пустую строку.

- Аргумент функции представляет собой пустой набор узлов.
- Первый встретившийся в наборе узел не имеет расширенного имени.
- URI пространства имен в расширенном имени является нулевым.
- Первый узел из набора не является элементом или атрибутом.

Если аргумент отсутствует, то вместо него используется текущий узел.

Функция *string name(node-set)*

Для первого узла, встретившегося в наборе, представленном аргументом, возвращает расширенное имя. Если данный набор пуст или первый встретившийся узел не имеет расширенного имени, то возвращается пустая строка. Если аргумент опущен, то по умолчанию в качестве набора узлов используетсятекущийузел.

Пример. Выражение: `*[name() = 'bbbbb']` Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <ccccc/>
    <ddddd/>
    <bbbbb/>
</aaaaa>
```

Пример. Выражение: `bbbbb[descendant::*[name()='ccccc']]` Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb>
        <ddddd/>
    </bbbbb>
    <bbbbb>
        <ddddd>
            <ccccc/>
        </ddddd>
        <ddddd/>
    </bbbbb>
    <bbbbb/>
</aaaaa>
```

3.5.2. Функции обработки строк

Функция *string string(object)*

Возвращает результат преобразования объекта в строку. Здесь под объектом понимается значение, имеющее один из определенных в XPath типов (на-

бор узлов, булево значение, число и строка). Преобразование объекта в строку происходит следующим образом.

- Набор узлов преобразуется в объединение строковых значений всех текстовых узлов из этого набора. Данные строковые значения располагаются в том порядке, который соответствует их порядку следования в документе *(см. разд. 3.7)*.
- В зависимости от своего значения число преобразуется в строку так:
  - NaN преобразуется в строку NaN;
  - положительный ноль преобразуется в 0;
  - отрицательный ноль преобразуется в 0;
  - положительная бесконечность преобразуется в Infinity;
  - отрицательная бесконечность преобразуется в -infinity;
  - если число целое, то результат его преобразования в строку не содержит точки. Если оно отрицательное, то первым символом (не считая пробельных символов) результата его преобразования должен быть минус;
  - если число не целое, то результат его преобразования в строку содержит точку. Перед точкой и после нее должна быть хотя бы одна цифра. Если число отрицательное, то первым символом (не считая пробельных символов) результата его преобразования должен быть минус.
- Булево значение false преобразуется в строку false. Булево значение true преобразуется в строку true.

Если аргумент функции опущен, то вместо него используется текущий узел.

Примечание

Функция string() не предназначена для представления чисел. Она не влияет на то, как будет отображен результат преобразования числа в строку. Для представления чисел служит функция format-number() *(см. разд. 3.6.3)*, а также директива xsl:number *(см. разд. 2.12)*.

Пример. Выражение: bbbbb[string() = 'BBBBB'] Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>AAAAA</bbbbb>
    <bbbbb>BBBBB</bbbbb>
    <ccccc>BBBBB</ccccc>
</aaaaa>
```

Пример. Выражение: bbbbb[string(ccccc) = 'ABC'] Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>
        <ccccc>ABC</ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>     ABC     </ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc><ddddd/>ABC</ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>
            ABC
        </ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>A<ddddd>B</ddddd><eeeee>C</eeeee></ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>AB
            <ddddd>C</ddddd>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Функция *string concat(string, string, string)*

Возвращает объединение своих аргументов, количество которых должно быть больше одного.

Пример. Выражение: bbbbb[string() = concat('AAA','BBB','CCC','DDD')] Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>BBBAAACCC</bbbbb>
    <bbbbb>AAABBBCCCDDD</bbbbb>
    <ccccc>DDDBBBCCC</ccccc>
</aaaaa>
```

Пример. Выражение: `bbbbb[string(ccccc) = concat(ddddd,'111')]`
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>
        <ccccc>AAA</ccccc>
        <eeeee>222</eeeee>
        <ddddd>111</ddddd>
    </bbbbb>
    <bbbbb>
        <ccccc>AAA</ccccc>
        <ddddd>222</ddddd>
        <ddddd>111</ddddd>
    </bbbbb>
</aaaaa>
```

Функция *boolean starts-with(string, string)*

Возвращает результат проверки, начинается ли строка, представленная первым аргументом, с того набора символов, который представлен вторым аргументом. Если в начале первой строки присутствует последовательность символов, составляющая вторую строку, то результатом будет true, в противном случае будет false.

Пример. Выражение: `bbbbb[starts-with(string(),'ABC')]`
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>abcde</bbbbb>
    <bbbbb>ABCDE</bbbbb>
    <bbbbb>ABC
        <ccccc>
            abcde
        </ccccc>
    </bbbbb>
    <bbbbb>
ABC</bbbbb>
</aaaaa>
```

Пример. Выражение: bbbbb[starts-with(self::*,'ABC')]
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>abcde</bbbbb>
    <bbbbb>ABCDE</bbbbb>
    <bbbbb>ABC
        <ccccc>
            abcde
        </ccccc>
    </bbbbb>
    <bbbbb>
ABC</bbbbb>
</aaaaa>
```

Пример. Выражение: bbbbb[starts-with(.,'ABC')] Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>abcde</bbbbb>
    <bbbbb>ABCDE</bbbbb>
    <bbbbb>ABC
        <ccccc>
            abcde
        </ccccc>
    </bbbbb>
    <bbbbb>
ABC</bbbbb>
</aaaaa>
```

Функция *boolean contains(string, string)*

Возвращает true, если строка второго аргумента [illegible] в строке первого аргумента. В противном случае возвращается false.

Пример. Выражение: bbbbb[contains(.,'ABC')] Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>abcde</bbbbb>
    <bbbbb>ABCDE</bbbbb>
```

```
<bbbbb>abcABCdef
    <ccccc>
        abcde
    </ccccc>
</bbbbb>
<bbbbb>
    ABC
</bbbbb>
<bbbbb>
    <ccccc>A</ccccc>
    <ddddd>B</ddddd>
    <eeeee>C</eeeee>
</bbbbb>
<bbbbb>
    <ccccc>
        <ddddd>A</dddddxeeeee>B</eeeeexfffff>C</fffff>
    </ccccc>
</bbbbb>
</aaaaa>
```

Функция *string* substring-before(string, *string*)

Возвращает ту часть строки, представленной первым аргументом, которая предшествует первому вхождению в нее подстроки, представленной вторым аргументом. Если вторая строка не содержится в первой (contains (strl, str2) =false), то возвращается пустая строка.

Пример. Выражение: `bbbbb[substring-before(.,'ABC')='ab']`
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>ABC</bbbbb>
    <bbbbb>abABC</bbbbb>
    <bbbbb>ABCab</bbbbb>
    <bbbbb>abABCdef</bbbbb>
    <bbbbb>cdABCabABC</bbbbb>
</aaaaa>
```

Функция *string substring-after(string, string)*

Возвращает ту часть строки, представленной первым аргументом, которая следует сразу после первого вхождения в нее подстроки, представленной вторым аргументом. Если вторая строка не содержится в первой (`contains(str1,str2)=false`), то возвращается пустая строка.

Пример. Выражение: ЬЬЬЬЬ `[substring-after(.,'ABC')='ab']`
Текущий узел: ааааа

```
<aaaaa>
    <bbbbb>ABC</bbbbb>
    <bbbbb>ABCab</bbbbb>
    <bbbbb>abABC</bbbbb>
    <bbbbb>defABCab</bbbbb>
    <bbbbb>ABCdeABCab</bbbbb>
</aaaaa>
```

Функция *string substring(string, number, number)*

Возвращает строковый результат вырезания из строки, представленной первым аргументом, начиная с символа, номер которого равен значению второго аргумента, такого количества символов, которое задано значением второго аргумента. Если третий аргумент отсутствует, вырезаются все символы, начиная с того, который задан значением второго аргумента, и до конца строки.

Символы нумеруются слева направо. Номер позиции первого символа равен 1.

Второй и третий аргументы функции substring() могут быть произвольными числами, отвечающими спецификации XPath. Иными словами, они могут быть положительными и отрицательными, целыми и числами с плавающей точкой, а также принимать одно из специальных значений *(см. разд. 3.3.3)*. Если значением какого-либо из последних двух аргументов является число с плавающей точкой, то оно будет округлено в соответствии с правилами округления, выполняемого функцией `round()`.

Если после округления второй аргумент функции меньше 1, то символы вырезаются, начиная с первого, и их количество в этом случае равно сумме второго и третьего аргументов минус 1 (`arg1 + arg2 - 1`). Если при этом третий аргумент не задан, то в результате будет возвращен первый аргумент (вся строка целиком). Если после округления второй аргумент функции больше либо равен 1, то функция работает так, как было описано в первом абзаце этого подраздела.

Далее приведем несколько примеров, иллюстрирующих нестандартное поведение функции substring(), часть которых была взята нами из спецификации XPath:

- substring('12345', 1.5, 2.6) возвращает '234';
- substring('12345', 0, 3) возвращает '12';
- substring('12345', -40, 30) возвращает '';
- substring('12345', -40, 50) возвращает '12345';
- substring('12345', -40, 1 div 0) возвращает '12345';
- substring('12345', 0 div 0, 3) возвращает '';
- substring('12345', 1, 0 div 0) возвращает '';
- substring('12345', -1 div 0, 1 div 0) возвращает ''.

Примечание

Такое поведение функции тесно связано с правилами вычисления выражений, либо содержащих, либо возвращающих специальные значения. Заметим, что:

1 div 0 возвращает Infinity;

-1 div 0 возвращает -Infinity;

0 div 0 возвращает NaN;

(1 div 0) + (-1 div 0) возвращает NaN (отрицательная бесконечность плюс положительная бесконечность есть не число).

Пример. Выражение: bbbbb[substring(.,1,3)='ABC']
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>ABC</bbbbb>
    <bbbbb>ABCab</bbbbb>
    <bbbbb>abABC</bbbbb>
    <bbbbb>defABCab</bbbbb>
    <bbbbb>ABCdeABCab</bbbbb>
</aaaaa>
```

Функция *number string-length(string)*

Возвращает количество символов в строке. Если аргумент опущен, то в качестве значения по умолчанию берется результат применения функции string() к текущему узлу.

Пример. Выражение: `bbbbb[string-length()=4]` **Текущий узел:** `aaaaa`

```
<aaaaa>
    <bbbbb>ABCD</bbbbb>
    <bbbbb>ABC </bbbbb>
    <bbbbb>abABC</bbbbb>
    <bbbbb> ABC</bbbbb>
    <bbbbb> AB </bbbbb>
</aaaaa>
```

Функция *string normalize-space(string)*

Возвращает нормализованную строку аргумента. Нормализация заключается в удалении пробельных символов в начале и конце строки, т. е. после нормализации первым и последним символом строки не является пробел. Последовательности пробельных символов внутри строки заменяются при этом единственным пробелом. Напомним, что пробельными считаются четыре символа: собственно сам пробел (`#x20`), символ табуляции (`#x9`), а также символы конца строки и возврата каретки (`#xA` и `#xD`).

Если аргумент функции опущен, то в качестве значения по умолчанию берется результат применения функции `string()` к текущему узлу.

Пример. Выражение: `bbbbb[normalize-space()='ABC')]` **Текущий узел:** `aaaaa`

```
<aaaaa>
    <bbbbb>
        <ccccc>ABC</ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>     ABC     </ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc><ddddd/>ABC</ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>
            ABC
        </ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>A<ddddd>B</ddddd><eeeee>C</eeeee></ccccc>
```

```
    </bbbbb>
    <bbbbb>
        <ccccc>AB
            <ddddd>C</ddddd>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Функция *string translate(string, string, string)*

Возвращает строку, представленную первым аргументом, в которой символы, указанные в строке, представленной вторым аргументом, заменяются символами из строки, представленной третьим аргументом, причем позиции заменяемых символов в значениях второго и третьего аргументов совпадают. Например, `translate(abcdefab, abc, ABC)` в качестве результата вернет **Строку** ABCdefAB.

Если вторая строка длиннее, чем третья, то те символы, которые в ней представлены и отсутствуют в третьей строке, будут удалены из первой строки вовсе. Например, translate (abcdefab, abcde, ABC) в качестве результата вернет строку ABCfAB.

Если какой-либо символ встретился во второй строке несколько раз, то используется правило для его первого вхождения, т. е. из третьей строки для замены будет взят такой символ, позиция которого равна позиции первого вхождения повторяющегося во второй строке символа.

Если длина строки третьего аргумента больше, чем длина строки второго аргумента, то те символы, номер позиции которых больше, чем длина строки второго аргумента, не используются.

Примечание

Функция translate () используется главным образом для изменения регистра символов. Так как язык XML чувствителен к регистру, то строки, состоящие из одних и тех же символов, но разной высоты, не будут идентичными. Таким образом для сравнения подобных строк безотносительно регистра составляющих их символов функция преобразования translate () оказывается очень удобной.

I Пример. Выражение: ЬЬЬЬЬ [translate(normalize-space(),abc,ABC)='abc')] **Текущий узел: aaaaa**

```
<aaaaa>
    <bbbbb>
        <ccccc>ABc</ccccc>
    </bbbbb>
    <bbbbb>
```

```
      <ccccc>      abC      </ccccc>
   </bbbbb>
   <bbbbb>
      <ccccc><ddddd/>aBC</ccccc>
   </bbbbb>
   <bbbbb>
      <ccccc>
         ABC
      </ccccc>
   </bbbbb>
   <bbbbb>
      <ccccc>A<ddddd>B</ddddd><eeeee>c</eeeeex/ccccc>
   </bbbbb>
   <bbbbb>
      <ccccc>AB
         <ddddd>C</ddddd>
      </ccccc>
   </bbbbb>
</aaaaa>
```

3.5.3. Функции обработки булевых значений

Функция *boolean* **boolean(object)**

Возвращает результат приведения типа объекта, представленного обязательным аргументом, к типу булево значение. Преобразование объекта происходит в соответствии с его типом:

- строка преобразуется в true только, если она имеет ненулевую длину;
- число преобразуется в true только в том случае, если его значением не является NaN, положительный или отрицательный нули;
- набор узлов преобразуется в true только тогда, когда он не пуст.

Пример. Выражение: bbbbb [boolean (descendant: : ddddd)]
Текущий узел: aaaaa

```
<aaaaa>
   <bbbbb/>
   <bbbbb>
      <ddddd/>
```

```
    </bbbbb>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb>
        <ccccc>
            <ddddd/>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Функция *boolean not(boolean)*

Возвращает true только тогда, когда результат применения функции boolean о к ее обязательному аргументу возвращает false. В противном случае возвращает false.

j Пример. Выражение: `bbbbb[not(descendant::ddddd)]` Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ddddd/>
    </bbbbb>
    <bbbbb>
        <ccccc/>
    </bbbbb>
    <bbbbb>
        <ccccc>
            <ddddd/>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Функция *boolean true()*

Всегда возвращает true.

j Пример. Выражение: `bbbbb[true()]` Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <ddddd/>
```

```
<bbbbb>
    <ccccc/>
<bbbbb/>
<eeeee/>
<bbbbb>ABC</bbbbb>
</aaaaa>
```

функция *boolean false()*

Всегда возвращает `false`.

Пример. **Выражение:** `bbbbb[false()]` **Текущий узел:** aaaaa

```
<aaaaa>
    <bbbbb/>
    <ddddd/>
    <bbbbb>
        <ccccc/>
    <bbbbb/>
    <eeeee/>
    <bbbbb>ABC< /bbbbb>
</aaaaa>
```

Функция *boolean lang(string)*

Возвращает true только тогда, когда значение атрибута `xml:lang` текущего узла равно строковому значению, представленному обязательным аргументом функции. Если у текущего узла отсутствует атрибут `xml:lang`, то такой атрибут ищется у его ближайшего предка. Если, в конце концов, атрибут не будет найден, то возвращается false. В противном случае происходит сравнение аргумента и значения этого атрибута, причем регистр операндов не учитывается, и если значение атрибута имеет суффикс, начинающийся с дефиса, то этот суффикс может быть отброшен.

Рассмотрим предлагаемый спецификацией XPath пример. Функция `lang('en')` будет возвращать true, если узлом контекста является любой из четырех перечисленных ниже элементов:

```
<para xml:lang = "en"/>
<div xml:lang = "en"><para/></div>
<para xml:lang = "EN"/>
<para xml:lang = "en-us"/>
```

3.5.4. Функции обработки чисел

Функция *number number(object)*

Возвращает результат приведения типа объекта, представленного обязательным аргументом, к типу "число". Преобразование объекта происходит в соответствии с его типом:

- строка, которая после нормализации не сдержит пробелов и начинается с необязательного знака "минус", преобразуется в число, ближайшее по своему значению со значением этой строки (приводится к значению 64-битного диапазона, нулю или к одному из специальных значений), для всех остальных строк возвращается NaN;
- булево значение true преобразуется в 1, булево значение false преобразуется в 0;
- набор узлов сначала преобразуется в строку, как при вызове для него функции string o, затем получившийся результат преобразуется в число по правилам преобразования строки.

Если аргумент отсутствует, то в качестве значения по умолчанию используется текущий узел.

Пример. Выражение: `bbbbb[number() = 222]` Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>111</bbbbb>
    <bbbbb>222</bbbbb>
    <ccccc>333</ccccc>
</aaaaa>
```

Пример. Выражение: `bbbbb[number(ccccc) = '123']` Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>
        <ccccc>123</ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>
            <ddddd/>
            123
        </ccccc>
```

```
    </bbbbb>
    <bbbbb>
        <ccccc>1<ddddd>2</ddddd><eeeee>3</eeeee></ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>12
            <ddddd>3</ddddd>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Функция *number* sum(node-set)

Возвращает сумму всех узлов из набора, представленного аргументом функции. Перед суммированием каждый узел набора приводится к числовому типу так, как если бы для него была вызвана функцияи number о. Суммируются те наборы, которые разделяются вертикальной чертой (|). Если какой-либо из этих наборов содержит несколько чисел, то их строковые значения будут объединены, как при вызове функции concat (), а затем преобразованы в число.

Пример. Выражение: `bbbbb[sum(ccccc[1]|ccccc[2]) = '30']`
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc>10</ccccc>
        <ccccc>20</ccccc>
        <ccccc>
            10
            <ddddd>20</ddddd>
        </ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>10</ccccc>
        <ccccc>
            <ddddd>10</ddddd><ccccc>10</ccccc>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Пример. Выражение: `bbbbb[sum(ccccc[1]|ccccc[2])='1020']`
Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb/>
    <bbbbb>
        <ccccc>10</ccccc>
        <ccccc>10<ddddd>10</ddddd>
        </ccccc>
    </bbbbb>
    <bbbbb>
        <ccccc>10</ccccc>
        <ccccc>
            <ddddd>10</dddddxccccc>10</ccccc>
        </ccccc>
    </bbbbb>
</aaaaa>
```

Функция *number* floor(number)

Возвращает наибольшее число, не превышающее значения обязательного аргумента number и являющееся целым. Алгоритм работы функции не зависит от знака ее аргумента.

Пример. Выражение: `bbbbb[floor(.) = '5']` **Текущий узел: aaaaa**

```
<aaaaa>
    <bbbbb>4.2</bbbbb>
    <bbbbb>5.3</bbbbb>
    <bbbbb/>
    <bbbbb>5.9</bbbbb>
    <bbbbb>4.8</bbbbb>
</aaaaa>
```

Функция *number* ceiling(number)

Возвращает наименьшее целое число, которое не меньше значения обязательного аргумента number. Принцип работы функции не зависит от знака ее аргумента.

Пример. Выражение: bbbbb[ceiling(.) = '5'] Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>4.2</bbbbb>
    <bbbbb>5.3</bbbbb>
    <bbbbb/>
    <bbbbb>5.9</bbbbb>
    <bbbbb>4.8</bbbbb>
</aaaaa>
```

функция *number* round(number)

Возвращает целое число, ближайшее к значению аргумента. Если аргумент представляет собой какое-либо из специальных значений (NaN, положительная и отрицательная бесконечности, положительный и отрицательный нули), то возвращается это специальное значение. Если аргумент меньше нуля, но больше, либо равен -0.5, то возвращается отрицательный нуль.

Примечание

В тех двух случаях, когда функция round() возвращает отрицательный нуль, принцип ее работы отличен от применения функции floor() к тому же аргументу с прибавленным к нему числом 0.5.

Пример. Выражение: bbbbb[round(.) = '5'] Текущий узел: aaaaa

```
<aaaaa>
    <bbbbb>4.2</bbbbb>
    <bbbbb>5.3</bbbbb>
    <bbbbb/>
    <bbbbb>5.9</bbbbb>
    <bbbbb>4.8</bbbbb>
</aaaaa>
```

3.6. Дополнительные функции

Язык XPath изначально разрабатывался для создания единого синтаксиса и семантики, совместно применяемых языками XSLT и XPointer. Поэтому некоторые конструкции XPath описываются в спецификациях тех средств, которые его используют. Как мы уже говорили во введении, мы не будем описывать в этой книге какие-либо X-технологии кроме XML, XSLT и

XPath, поэтому рассмотрим здесь лишь те конструкции XPath, которые на предоставляет спецификация XSLT.

Хотя этот раздел называется "Дополнительные функции", мы опишем здесь не только функции, расширяющие возможности XPath, но и некоторые ди-рективы XSLT, использующиеся совместно с этими функциями. Все описы-ваемые в данном разделе конструкции, как и основные функции XPath, обязательно должны поддерживаться любой реализацией синтаксического анализатора.

3.6.1. Внешние наборы узлов

Существует возможность адресовать наборы узлов, находящиеся в каком-либо внешнем источнике, а не только в текущем документе. Для этих целей служит функция

node-set `document(`*`object`*`,` *node-set)*

которая возвращает набор узлов из внешнего источника, описываемого с помощью ее аргументов.

Если первый аргумент функции представляет собой строку, то он интерпретируется, как относительный или абсолютный URI внешнего источника информации. Если при этом аргумент представляет собой относительный URI, то он должен быть преобразован в абсолютный URI относительно либо базового URI документа, либо URI первого узла из набора, представленного вторым, необязательным аргументом функции. Иными словами, второй аргумент функции необходим только тогда, когда, во-первых, первый аргумент задает относительный URI источника, во-вторых, его преобразование в абсолютный URI должно происходить не относительно базового URI документа, а каким-либо специальным образом. Этот аргумент функции, как правило, не используется.

Внешний источник информации может представлять собой не только какой-либо XML-файл, но и некоторый его фрагмент. При этом фрагменты могут быть заданы либо с помощью DTD (уникальные идентификаторы, определяемые атрибутами типа ID), либо с помощью каких-либо дополнительных X-технологий (например, XLink). В силу того, что процессор не обязан поддерживать различные схемы URI, везде далее мы не будем использовать идентификаторы фрагментов. Вместо этого мы всегда будем считывать корень того документа, который представляет собой внешний источник. Второй аргумент функции нам также не потребуется.

Рассмотрим пример. Предположим, что мы хотим обработать узлы дерева, которое представлено файлом doc.xml, содержащим корректный XML-документ. Причем этот файл не является текущим документом, т. е. представляет собой внешний источник. Далее следует его фрагмент:

```
<employees>
   <employee>
      <name>name1</name>
      <surname>surname1</surname>
      <department>dep1</department>
      <note>note1</note>
   </employee>
   <employee>
      <name>name2</name>
      <surname>surname2</surname>
      <department>dep2</department>
      <note>note2</note>
   </employee>
</employees>
```

Для того чтобы получить корень этого документа (например, присвоить значение его адреса какой-нибудь переменной), мы должны вставить в сценарий XSLT следующий код:

```
<xsl:variable name = "var" select = "document('doc.xml')"/>
```

Далее мы можем использовать ссылку на эту переменную ($var) так же, как и произвольный внутренний набор узлов. Например, мы можем его трансформировать:

```
<xsl:apply-templates select="$var"/>
```

Кроме того, применяя в качестве первого шага адресации либо саму функцию document(), либо переменную, содержащую возвращаемое ею значение, можно адресовать какие-либо отдельные узлы из этого набора. Например,

```
<xsl:value-of select="$var/employees/employee[2]/name"/>
```

в конечном дереве создаст текстовый узел с контентом name2.

Первым аргументом функции может являться набор узлов. Тогда функция document() будет возвращать результат объединения вызовов этой функции для строкового значения каждого узла из этого набора. Иными словами, для каждого узла из набора процессор вычислит его строковое значение, а затем вызовет функцию document(), используя это значение в качестве ее первого аргумента. После этого результаты этих вызовов будут объединены и возвращены как результат вызова функции, первый аргумент которой представляет собой рассматриваемый набор узлов.

Таким образом, если текущий документ содержит код

```
<files>
    <file>doc1.xml</file>
    <file>doc2.xml</file>
</files>
```

то В результате вызова document(files/*) ИЛИ document (files/file[1] | files/file[2]), что одно и то же, мы получим набор узлов, который является объединением корней документов, представленных файлами doc1.xml и doc2.xml. Спецификация не накладывает ограничений на порядок следования узлов в результирующем наборе в зависимости от порядка следования узлов, представленных первым аргументом функции. Должно выполняться только одно правило: при многократных вызовах функции document() для одного и того же набора узлов порядок следования узлов в результирующем наборе должен быть одинаковым для каждого вызова.

3.6.2. Ключи

В языке XML существует возможность связывать элементы с помощью перекрестных ссылок. Для того чтобы пометить элемент для последующей ссылки на него, используются атрибуты типа ID. Тип атрибута указывается во внешнем или внутреннем наборе DTD. Для ссылки на этот элемент служат атрибуты типа IDREF или IDREFS.

Примечание

В HTML аналогом атрибута типа ID является конструкция <a name = "">. Аналогом атрибута типа IDREF является <a href = "">.

Для ссылки на единственный элемент используется атрибут типа IDREF, значением которого является соответствующее значение атрибута типа ID. Атрибут типа IDREFS служит для ссылок сразу на несколько элементов, и его значением является несколько значений атрибутов типа ID (разделенных пробелами), соответствующих тем элементам, на которые реализуется ссылка.

Спецификация XML не указывает, какое именно отношение устанавливается для элементов, связываемых с помощью атрибутов типа ID/IDREF/IDREFS. Утверждается только сам факт существования некоторого отношения.

Данный механизм связывания имеет ряд существенных недостатков.

- Атрибуты типа ID/IDREF/IDREFS должны быть декларированы во внутреннем или внешнем наборе DTD. В том случае, когда они определены во внешнем DTD, соответствующие им элементы будут связанны только, если процессор прочтет этот внешний DTD. Однако, согласно спецификации XML, он это делать не обязан. И тем более, данные атрибуты не будут работать в том случае, когда имеет место декларация ОДИНОЧНОГО документа — standalone = "yes".

- Документ может иметь только один набор уникальных атрибутов типа ID.
- Уникальный идентификатор элемента может быть задан исключительно атрибутом этого элемента.
- Значение атрибута типа ID должно подчинятся тем же правилам синтаксиса XML, в соответствии с которыми присваиваются имена узлам. Например, оно не должно содержать внутри себя пробельных символов.
- Каждый элемент может содержать не более одного атрибута типа ID.
- Если для какого-либо атрибута типа ID задано уникальное значение, то в документе может присутствовать только один элемент, которому принадлежит данный атрибут с этим значением. Иными словами, значение атрибута типа ID не должно повторяться даже для тех атрибутов типа ID, которые имеют разные имена.

Гораздо более гибким является механизм ключей, определенный спецификацией XSLT. Ключи так же, как и атрибуты типа ID/IDREF/IDREFS, служат для связывания узлов, но при этом лишены всех вышеперечисленных недостатков.

Ключ в XSLT определяется тремя параметрами:

- узлом (не обязательно элементом), для которого задан ключ;
- именем (синтаксис имени подчиняется правилам синтаксиса, определенным для имен узлов спецификацией XML);
- значением (синтаксис значения ключа *не обязан* подчиняться правилам синтаксиса, определенным для имен узлов спецификацией XML).

Для создания ключей в XSLT определена следующая директива:

```
<xsl:key
  name = имя ключа
  roatch = набор узлов (образец), которым будет принадлежать ключ
  use = значение ключа />
```

Здесь атрибут name задает имя ключа, атрибут match описывает тот набор ...ов, которые после обработки директивы xsl:key будут владельцами это... ключа, и атрибут use, значением которого является выражение XPath, определяет значение генерируемого ключа. Если в качестве значения атрибута match выступает не единственный узел, а набор узлов, то значение use б... вычислено отдельно для каждого узла из набора. Например, конструкция <xsl:key name ="key1" match ="item" use ="@type"/> для каждого эл...ента item сгенерирует ключ с именем key1 и значением, равным значению атрибута данного элемента item.

В том случае, когда значением атрибута match является относительный адрес, он преобразуется в абсолютный с помощью добавления к нему слева

строки /descendant-or-self()::node()/ ИЛИ, согласно правилам сокращенного синтаксиса, — двойного слеша (//). Короче говоря, значение атрибута match представляет собой образец.

Если значением атрибута use является набор узлов, то в качестве значения ключа берется результат преобразования в строку данного набора узлов. К правило, значение атрибута use представляет собой относительный адрес. При преобразовании этого адреса в абсолютный в качестве текущего рассматривается тот узел из набора, определяемого атрибутом match, для которого в данный момент производится вычисление атрибута use. Как мы уже говорили, значение атрибута use вычисляется последовательно для каждого узла из набора, заданного в match.

Директива xsl:key должна быть элементом верхнего уровня. Если какой-либо из атрибутов match или use содержит ссылку на переменную ($*variable_name*),фиксируется ошибка.

Один и тот же узел может иметь несколько ключей с одинаковыми именами, но разными значениями. Кроме того, в документе может быть несколько ключей с одинаковыми именами, одинаковыми значениями, но соответствующих разным узлам.

Для того чтобы сформировать набор узлов, для которых определены ключи с заданным именем и заданным значением, служит функция

node-set key(*string, object)*

Оба атрибута являются обязательными. Первый аргумент функции служит для указания имени ключа. Если второй аргумент не является набором узлов, то его значение приводится к строковому типу так, как если бы для него была вызвана функция string(). После этого преобразования функция key() возвращает набор узлов, для которых определен ключ с именем, представленным ее первым аргументом, и значением, если этим аргументом является строка. Если второй аргумент функции key() является набором узлов, то возвращаемым значением будут объединенные результаты ее последовательного вызова для каждого узла из этого набора. Причем в качестве первого аргумента будет использован первый аргумент исходного вызова функции, а в качестве второго — результат преобразования в строку того узла, для которого осуществляется вызов.

Рассмотрим несколько примеров.

```
<xsl:key name = "key1" match = "aaaaa" use = "'aaaaa'"/>
```

Результатом обработки директивы

```
<xsl:value of select = "key('key1', 'aaaaa')"/>
```

будет строковое значение элемента aaaaa.

Следующая директива создаст ключи для всех элементов aaaaa, причем значением каждого ключа будет значение атрибута attr соответствующего элемента aaaaa

```
<xsl:key name = "key1" match = "aaaaa" use = "@attr"/>
```

Если данные элементы aaaaa входят в представленный ниже XML-фрагмент, то в результате обработки <xsl:apply-templates select="key('key1','111')"/> соответствующее правило шаблона будет применено к тем узлам, которые выделены полужирным шрифтом:

```
<aaaaa attr = "111">
   <bbbbb/>
   <aaaaa attr = "222"/>
   <ccccc>
      <aaaaa attr = "111"/>
   </ccccc>
</aaaaa>
```

Рассмотрим более сложный пример. Предположим, что наш XML-документ состоит из двух частей. Первая представляет собой описание всех служащих некоей компании, вторая — список некоторых из этих служащих, сформированный по какому-либо критерию.

```
<doc>
 <employees>
   <employee>
      <name>Имя</name>
      <surname>Фамилия</surname>
      <department>Отдел</department>
      <note>Некоторые пояснения</note>
   </employee>
   <employee>
      <name>Имя</name>
      <surname>Фамилия</surname>
      <department>Отдел</department>
      <note>Некоторые пояснения</note>
   </employee>
   <!-- и т. д. по списку -->
 </employees>
```

```
  <list>
    <employee>
      <name>Имя</name>
      <surname>Фамилия</surname>
    </employee>
    <employee>
      <name>Имя</name>
      <surname>Фамилия</surname>
    </employee>
    <!-- и т. д. по списку -->
  </list>
</doc>
```

Как следует из кода, каждый служащий в первой части документа представлен своим именем (элемент name), фамилией (элемент surname), отделом, в котором он работает (элемент department), а также некоторой дополнительной информацией (элемент note). Вторая часть документа — простое перечисление некоторых служащих, каждый из которых уникально идентифицируется своим именем и фамилией.

Предположим, что для каждого служащего из второй части мы хотим получить название отдела, в котором он работает, и соответствующую ему пояснительную информацию. Этой цели мы можем достичь, создав правило шаблона каким-либо из двух способов:

```
<xsl:template match = "//list/employee">
  <xsl:variable name = "name" select = "name"/>
  <xsl:variable name = "surname" select = " surname"/>
  <xsl:value-of select = "//employees/employee[surname = $surname and
name = $name]/department"/>
<xsl:value-of select = "//employees/employee[surname = $surname and name
- $name]/note"/>
</xsl:template>
```

Если вторая часть документа представляет собой не линейный список, какую-либо нерегулярную структуру, то вышеприведенный шаблон можн сделать именованным и вызывать его по мере необходимости с параметр ми, имеющими значения соответствующих атрибутов name и surname.

Далее следует альтернативное правило шаблона, использующее заранее со данные ключи:

```
<xsl:key name = "employee" match = "//employees/employee" use =
"concat(name, surname)"/>
```

```
<xsl:template match = "employee">
  <xsl:value-of select = "key ( 'employee', concat(name, surname))/department"/>
  <xsl:value-of select = "key('employee', concat(name, surname))/note"/>
</xsl:template>
```

Второй вариант более компактен, кроме того, вместо создания именованного шаблона, из любого места в документе мы можем вызвать функцию key() с соответствующими параметрами. Действительно, если нам понадобится совершить различные действия над элементом employee из первой части (а не только лишь выводить их в конечный документ с помощью директивы xsl:value-of), идентифицируемым какой-либо парой атрибутов name-surname, то для каждого такого действия нам придется создавать отдельный именованный шаблон. С другой стороны, воспользовавшись функцией key(), мы получим требуемый набор узлов, с которым далее можно делать все, что угодно.

С помощью ключей можно индексировать узлы, находящиеся во внешнем XML-документе. В этом случае директива xsl:key и функция key() используются наряду с функцией document(). Предположим, что каждую часть документа из предыдущего примера мы поместили в отдельный XML-файл (docl.xml и doc2.xml соответственно), причем текущий узел находится в том документе, который содержит те элементы employee, которые имеют родителем list (docl.xml). Тогда для получения соответствующих им элементов из второго XML-файла можно воспользоваться следующим кодом:

```
<xsl:key name = "employee" match = "//employees/employee" use = "concat(name, surname) "/>
<xsl:template match = "employee">
  <xsl:variable name = "name" select = "name"/>
  <xsl:variable name = "surname" select = "surname"/>
  <xsl:for-each select = "document('doc2.xml')">
    <xsl:value-of select = "key('employee', concat($name, $surname))/department"/>
    <xsl:value-of select = "key( 'employee', concat($name, $surname) )/note"/>
  </xsl:for-each>
</xsl:template>
```

3.6.3. Форматирование чисел

В предыдущей главе мы рассмотрели директиву xsl:number, которая служит для... Представления исключительно целых чисел. Для того чтобы указать

способ отображения произвольного числа, допустимого в XPath, используется функция

string format-number(*number*, string, string)

Эта функция возвращает преобразованное в строку число, заданное первым аргументом. При этом в качестве шаблона форматирования используется строка, представленная обязательным первым аргументом. Третий, необязательный аргумент функции предназначен для указания именованного десятичного формата.

Примечание

Шаблон форматирования задается согласно правилам синтаксиса, определенного для класса DecimalFormat, стандартом JDK 1.1. Мы не будем здесь приводить эти правила в силу их громоздкости. И точно так же, как и спецификация XPath, сошлемся на вышеупомянутый стандарт.

Для создания именованного десятичного формата используется директива

```
<xsl:decimal-format
  name = имя создаваемого формата
  decimal-separator = char
  grouping-separator = char
  infinity = string
  minus-sign = char
  NaN = string
  percent = char
  per-mille = char
  zero-digit = char
  digit = char
  pattern-separator = char />
```

Здесь char означает, что значением данного атрибута должен быть единственный символ. Если задан атрибут name, то декларируется именованный шаблон форматирования, в противном случае декларируется шаблон, который будет использоваться по умолчанию. Если именованный шаблон форматирования или шаблон по умолчанию декларируются несколько раз, ошибка фиксируется даже тогда, когда эти декларации обладают разным приоритетом импорта. Однако такое дублирование деклараций возможно при условии, что в каждой из них заданы одни и те же значения для соответствующих атрибутов.

Ниже приведем список описаний остальных атрибутов директивы xsl:decimal-format:

- decimal-separator — задает символ, обозначающий десятичную точку по умолчанию используется точка (.);

- grouping-separator — задает символ, предназначенный для отделения друг от друга групп цифр (например, тысяч), по умолчанию используется запятая (,);
- infinity — задает строку для обозначения бесконечности, по умолчанию используется Infinity;
- minus-sign — задает символ для обозначения знака "минус", по умолчанию используется -;
- NaN — задает строку для представления значения NaN, по умолчанию используется строка NaN;
- percent — определяет символ процента, по умолчанию используется %;
- per-mille — определят символ промилле, по умолчанию используется символ Unicode (#2030);
- zero-digit — задает обозначение нуля, по умолчанию используется цифра o;
- digit — задает символ для обозначения цифры в шаблоне форматирования, по умолчанию используется #;
- pattern-separator — задает символ для отделения друг от друга положительной и отрицательной части шаблона форматирования, по умолчанию используется точка с запятой (;).

Примечание

Атрибуты директивы `xsl:decimal-format` **соответствуют полям класса** `DecimalFormatSymbols` **определенного стандартом JDK 1.1.**

3.6.4. Различные дополнительные функции

Функция *node-set current()*

Возвращает текущий узел. Таким образом, директивы

```
<xsl:value-of select = "."/>
```

И

```
<xsl:value-of select = "current()"/>
```

будут генерировать идентичные текстовые узлы. Однако в некоторых выражениях, использующих предикаты, значение текущего узла может меняться. Рассмотрим следующее правило шаблона:

```
<xsl:template match = "aaaaa">
  <xsl:value-of select = "bbbbb/ccccc[@attr =./@attr]"/>
</ xsl:template>
```

Здесь делается попытка вывести значение такого элемента ccccc, котор является непосредственным потомком bbbbb, который, в свою очередь, я ляется непосредственным потомком aaaaa, при этом предполагается, ч текущим узлом, обозначенным точкой, будет aaaaa, и следовательно бу выведено значение такого ccccc, значение атрибута attr которого совпал ет со значением атрибута attr элемента aaaaa. Но здесь по правилам отн сительной адресации текущий узел был изменен, и поэтому приведенны выше шаблон будет эквивалентным с

```
<xsl:value-of select = "bbbbb/ccccc[@attr = @attr]"/>
```

а следовательно, условие, определенное предикатом, всегда будет истинны Для того чтобы результат обработки этой директивы был таким, как зад мывалось, необходимо записать

```
<xsl:value-of select = "bbbbb/ccccc[@attr = current()/@attr]"/>
```

Именно для такого применения и определена функция current().

Функция *string unparsed-entity-uri(string)*

Как следует из ее названия, возвращает URI неанализируемой (неразобран ной) сущности *(см. разд. 1.11.3),* имя которой представляет аргумент фун ции. Сначала процессор пытается воспользоваться публичным идентифика тором сущности, а если таковой отсутствует, то рассматривается системны идентификатор, причем если он представляет собой относительный URI, т он должен быть преобразован в абсолютный. Если в документе, в которо присутствует текущий узел (в шаблоне которого находится вызов функции отсутствует неанализируемая сущность с заданным именем, то возвращается пустая строка.

Функция *string generate-id(node-set)*

Возвращает уникальный идентификатор строкового типа для такого узла и набора узлов, представленного необязательным аргументом функции, кото рый первым встретится в документе. Для идентификатора должны выпол няться такие же правила синтаксиса XML, согласно которым задаются име на узлов. Спецификация XPath не определяет способ, с помощью которог должно быть сгенерировано строковое значение, но при этом накладывае на него ряд ограничений:

- для одного и того же узла генерируется один и тот же идентификатор;
- для различных узлов генерируются разные идентификаторы;
- для одного и того же узла при каждой трансформации могут генериро ваться разные идентификаторы.

Если аргумент функции представляет собой пустой набор узлов, то должна быть сгенерирована пустая строка, если же аргумент отсутствует вовсе, то в качестве значения по умолчанию берется текущий узел. Рассмотрим пример:

```
<xsl:value of select = "generate-id(bbbbb//ccccc)"/>
```

Данная директива создаст текстовый узел, значением которого будет уникальный идентификатор первого встретившегося потомка элемента ьььььь, имеющего имя ссссс.

функция *object* system-property(string)

Возвращает значение системного свойства, представленного необязательным аргументом функции. Тип возвращаемого значения зависит от того, какое свойство было указано. Спецификацией XSLT, имеющей версию 1.0, предусматривается три предопределенных системных свойства, описанные в табл. 3.4.

Таблица 3.4. Значения, возвращаемые функцией *system-property o*

Свойство	Значение	Тип	Пояснение
xsl:version	1.0	Число	Версия спецификации XSLT
xsl:vendor	-	Строка	Идентификатор разработчика
xsl:vendor-url	-	Строка	URL разработчика (как правило, домашняя страница)

3.7. Модельданных

Язык XPath рассматривает XML-документ как некоторое множество данных, имеющее древовидную структуру. При этом в документе нет таких Данных, которые не являются вершиной этого дерева или ее частью. Исключение составляют декларация XML (заголовок) и объявление типа документа. Каждая вершина дерева называется узлом. Узел может иметь один из следующих семи типов:

- корень;
- элемент;
- атрибут;
- инструкция обработки;
- комментарий;

- текстовый узел;
- узел пространства имен.

Корень является предком для всех остальных узлов дерева. Его непосредственными потомками могут быть: единственный элемент, называющийся корневым элементом или элементом документа, произвольное количество комментариев и инструкций обработки. Кроме вышеперечисленных типов узлов непосредственными потомками корня могут быть такие текстовые узлы, которые после нормализации содержат только лишь пробел *(см. разд. 2.17.1)*. Это означает, что комментарии, инструкции обработки и элемент документа можно отделять друг от друга пробелами, а также символами табуляции и конца строки.

Непосредственными потомками элементов могут быть другие элементы, комментарии, инструкции обработки и текстовые узлы. Элементы не могут владеть совместно узлом какого-либо из вышеперечисленных типов. Иными словами, узлы этих типов могут иметь только одного родителя.

Узлы, не являющиеся корнем или элементом, не имеют непосредственных потомков. Узлы, являющиеся атрибутом или пространством имен, хотя и имеют родителя, но не являются его непосредственными потомками. Если взять в качестве примера следующий XML-код:

```
<aaaaa attr="value">
    <!— текст комментария -->
    <bbbbb/>
    <?some_pi pi_value="value"?>
    Некоторый текст
</aaaaa>
```

ТО результатом Обработки директивы `<xsl:value-of select = "node () [2] "/>` (текущий узел aaaaa) будет строковое значение комментария, т. е. текст комментария. Позиция комментария окажется второй потому, что первым будет текстовый узел, соответствующий пробельным символам, находящимся между открывающим тегом элемента aaaaa и узлом комментария.

В данном случае элемент aaaaa имеет семь непосредственных потомков, т. е. результатом обработки директивы `<xsl:value-of select="count(node())"/>` будет число 7. Тремя из этих потомков будут комментарий, элемент и инструкция обработки, а остальными — текстовые узлы, находящиеся между ними (два текстовых узла, содержащие только пробельные символы), текстовый узел, находящийся между открывающим тегом элемента aaaaa, и текстовый узел, располагающийся между инструкцией обработки и закрывающим тегом элемента aaaaa (единственный текстовый узел, содержащий не только пробельные символы).

Как видно из примера, атрибут attr не является первым непосредственным потомком элемента aaaaa. Точно так же и узел пространства имен не является непосредственным потомком своего родителя. Именно поэтому в языке XPath для узлов этих двух типов специально определены соответствующие оси: attribute и namespace, в то время как для узлов остальных типов тип определяется в соответствии с правилом проверки узлов. Если это правило не содержит явно идентификатор типа (text(), comment(), processing-instruction() или node()), то по умолчанию данные узлы рассматриваются как элементы.

Порядок следования узлов в документе должен соответствовать следующим правилам:

1. Первым в документе располагается корень.
2. Непосредственные потомки корня располагаются в документе в зависимости от своей близости к началу документа, т. е. первым будет тот узел, который находится ближе всего к прологу документа.
3. Потомки в n-ом поколении предшествуют потомкам в m-ом поколении, если $n<m$.
4. Потомки в одном поколении (непосредственные потомки) располагаются в документе в зависимости от своей близости к открывающему тегу их элемента-родителя.
5. Порядок следования атрибутов внутри множества атрибутов, принадлежащих одному и тому же элементу, зависит от реализации процессора.
6. Порядок следования узлов пространства имен внутри множества узлов пространства имен, принадлежащих одному и тому же элементу, зависит от реализации процессора.
7. Атрибуты и узлы пространства имен предшествуют непосредственным потомкам их содержащего элемента, причем узлы пространства имен предшествуют атрибутам.

Вышеописанный порядок следования узлов называется прямым. В спецификации содержится также понятие обратного порядка следования узлов, при котором узлы дерева перебираются, начиная с листьев этого дерева и заканчивая его корнем.

Примечание

Порядок следования узлов в документе, чаще всего, имеет значение при последовательной обработке узлов из некоторого набора. Например, с помощью Директив `xsl:for-each` или `xsl:apply-templates`. Кроме того, отсев узлов, производимый с помощью предикатов, сильно зависит от того набора узлов, к которому он применяется, и, в частности, от используемой оси *(см. разд. 3.2.4).*

ГЛАВА 4

Комплексный пример

Теперь, когда используемый нами инструментарий формально описан, мы можем попробовать применить полученный опыт на практике. В этой главе мы рассмотрим пример, в котором задействовано максимальное количество директив XSLT, расширений, предлагаемых парсером Cocoon, а также активно используются самые разные выражения XPath. Для удобства мы разбили наш код на несколько листингов, сложность которых постепенно увеличивается.

Рассмотрим следующую постановку задачи. Пусть на нашем сервере хранится множество аудиофайлов, и мы хотим создать их каталог. При этом необходимо обеспечить возможность просмотра как всего каталога целиком, так и некоторой его части, сформированной из аудиофайлов, отобранных по некоторым критериям. Договоримся для каждого из аудиофайлов рассматривать следующие его характеристики:

- Исполнитель (artist);
- Название композиции (title);
- Название альбома (album);
- Год, когда была записана композиция (year);
- Жанр (genre).

Примечание

Хотя данные характеристики являются полями заголовка файла в формате MP3, мы не будем пытаться получить к ним непосредственный доступ, скажем, с помощью Java. Основной целью будет демонстрация возможностей XML в рамках нашей задачи.

Реализацией этой задачи будет несколько Web-страниц, связанных между собой с помощью гиперссылок. Для каждой из этих страниц мы создадим удобный, но не изощренный в плане оформления интерфейс. И если

для кого-то разработка каталога является актуальной проблемой, то он может, взяв за основу нижеприведенный код, создать для него такой дизайн, который сочтет достаточно эстетичным. Со своей стороны мы определим некоторые минимальные стили для некоторых используемых нами тегов HTML, но сделаем это с учетом того, что CSS2 не является темой нашей книги.

Прежде всего, нам необходимо определить способ хранения реквизитов наших аудиофайлов. Самым простым будет задать отдельный элемент для каждого файла, а также для каждой его характеристики. Причем элементы, чьим текстовым контентом будут значения характеристик, мы сделаем непосредственными потомками элемента, содержащего соответствующий аудиофайл.

Листинг 4.1 Файл storage.xml 1

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

<audiofiles>
  <audiofile>
    <artist>artistl</artist>
    <title>title3</title>
    <album>album1</album>
    <genre>jazz</genre>
    <year>1992</year>
  </audiofile>
  <audiofile>
    <artist>artist2</artist>
    <title>title2</title>
    <album>album3</album>
    <genre>rock</genre>
    <year>1990</year>
  </audiofile>
  <audiofile>
    <artist>artist3</artist>
    <title>title3</title>
    <album>album3</album>
    <genre>industrial</genre>
    <year>1998</year>
  </audiofile>
  <audiofile>
```

```
    <artist>artist4</artist>
    <title>titlel</title>
    <album>album2</album>
    <genre>classic</genre>
    <year>1996</year>
  </audiofile>
  <audiofile>
    <artist>artist5</artist>
    <title>title3</title>
    <album>album1</album>
    <genre>rock</genre>
    <year>1992</year>
  </audiofile>
  <audiofile>
    <artist>artist6</artist>
    <title>title2</title>
    <album>album3</album>
    <genre>rock</genre>
    <year>1992</year>
  </audiofile>
  <audiofile>
    <artist>artist7</artist>
    <title>title3</title>
    <album>album3</album>
    <genre>rock</genre>
    <year>1998</year>
  </audiofile>
  <audiofile>
    <artist>artist8</artist>
    <title>titlel</title>
    <album>album2</album>
    <genre>jazz</genre>
    <year>1996</year>
  </audiofile>
  <audiofile>
    <artist>artist9</artist>
    <title>title3</title>
    <album>album1</album>
```

```
    <genre>rock</genre>
    <year>1992</year>
  </audiofile>
  <audiofile>
    <artist>artist10</artist>
    <title>title2</title>
    <album>album3</album>
    <genre>rock</genre>
    <year>1990</year>
  </audiofile>
  <audiofile>
    <artist>artist11</artist>
    <title>title3</title>
    <album>album3</album>
    <genre>rock</genre>
    <year>1998</year>
  </audiofile>
  <audiofile>
    <artist>artist12</artist>
    <title>title1</title>
    <album>album2</album>
    <genre>rock</genre>
    <year>1996</year>
  </audiofile>
</audiofiles>
```

Здесь корневым элементом является audiofiles, содержащий описания всех имеющихся в наличии аудиофайлов, каждый из которых представлен элементом audiofile. Элементами, представляющими реквизиты файла, являются соответственно: для исполнителя — artist, для названия композиции — title, для названия альбома — album, для года записи — year и для жанра — genre.

Заметим, что файл storage.xml только хранит некоторую информацию, но не отвечает за то, каким образом она будет представлена пользователю. Поэтому в листинге 4.1 отсутствует подключение стилевой таблицы. Здесь единственной конструкцией (помимо определения элементов), на которую стоит обратить внимание, является заголовок или, выражаясь формально, объявление XML. Он представлен первой строкой листинга 4.1 <?xml version="1.0"?>.

Примечание

Текстовые значения элементов могут быть заданы и на русском языке. Но т. к. обработку русскоязычных параметров HTTP-запроса нельзя назвать тривиальной задачей (хотя это и не слишком сложно), мы не будем в этом примере описывать обработку таких параметров, чтобы не загромождать код и описание примера. Мы опишем этот механизм несколько позже. Но тем не менее, некоторые из нижеприведенных листингов будут содержать русский контент. Поэтому эти листинги, в отличие от листинга 4.1, будут иметь в объявлении XML атрибут encoding.

4.1. Главная страница

Перед тем, как приступить к реализации какого-либо поиска, нам необходимо сформировать главную страницу, которая позволяла бы осуществлять навигацию по созданному сайту или, проще говоря, выбирать ту Web-страницу, которая соответствует нужному режиму поиска. В нашем примере таких режимов будет пять.

Путешествуя по Internet, очень часто можно встретить такую структуру сайта, при которой он разбит на три вертикальные колонки, перед которыми располагается какой-нибудь заголовок и, может быть, сопутствующие ему логотипы. Эту структуру в грубом приближении мы и возьмем за основу для реализации нашей задачи. Левая колонка у нас будет содержать ссылки на остальные пять Web-страниц, средняя — текст, разбитый на две части: общую, присутствующую на каждой Web-странице (в т. ч. и на главной), и ту, которая представляет собой описание Web-страницы, загруженной в данный момент. Помимо текста описания средняя часть будет содержать также те элементы управления, взаимодействуя с которыми, пользователь сможет осуществить поиск по необходимому критерию.

Правая часть, соответствующая третьей колонке, согласно выбранному нами сценарию, будет содержать некоторые ресурсы Internet. Как правило, данная колонка содержит ссылки на сайты сопутствующей тематики, на рекламируемые сайты, на какие-либо публикации и т. п. Мы расположим в ней просто несколько гиперссылок, которые представляют собой ресурсы, использованные для наших разработок. Там можно найти как программное обеспечение, так и документацию по различным Web-сервисам, в т. ч. Cocoon и Java.

Такую трехколоночную структуру будут иметь и все остальные (не только главная) Web-страницы. Так как мы договорились, что каждая из этих страниц будет содержать в своей средней колонке некоторую общую текстовую часть, то будет удобным вынести эту часть в отдельный файл для того, чтобы избежать многократного дублирования. Кроме того, это позволит наглядно продемонстрировать модульность языка XML, которую в нашем случае обеспечивает директива `xsl:import`.

Заметим, что левая колонка будет повторяться на каждой странице почти полностью, она будет содержать ссылки на все остальные Web-страницы, кроме главной и той, которая загружена в данный момент. Ведь загрузив какую-либо из имеющихся страниц, нет необходимости активизировать гиперссылку, ссылающуюся на ту же самую страницу. Правая колонка будет полностью повторяться на каждой Web-странице. Мы создали ее только для частичного соответствия разрабатываемого нами сайта тем, которые мы можем увидеть в Web.

Следовательно, нам будет нужно вынести в отдельный документ не только общую текстовую часть из средней колонки, но также левую и правую колонки целиком. Но хватит общих описаний. Лучше один раз увидеть, чем сто раз услышать, поэтому ниже мы приводим листинги, соответствующие главной странице, а также рис. 4.1, на котором изображены результаты обработки XML-документа, содержащегося в этих листингах.

Листинг 4.2, *а*. Главная страница. XML-составляющая (main.xml)

```
<?xml version="1.0" encoding="koi8-r"?>
<?xml-stylesheet href = "main.xsl" type = "text/xsl"?>
<?cocoon-process type = "xslt"?>

<main/>
```

Листинг 4.2, *б*. Главная страница. XSL-составляющая (main.xsl)

```
<?xml version="1.0" encoding="koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:include href = "common.xsl"/>
<xsl:variable name = "panels" select = "document('panels.xml')/panels"/>

<xsl:template match = "main">
  <xsl:processing-instruction name = "cocoon-
f-rmat">type="text/html"</xsl:processing-instruction>
 <html>
  <body>
    <h1 align = "center">Хранилище аудиофайлов</h1>
    <xsl:apply-templates select = "$panels/controls"/>
    <span style="top:50px;width:60%;vertical-align:top;margin-right:5%">
      <xsl:apply-templates select = "$panels/description"/>
```

```
    </span>
    <xsl:apply-templates select = "$panels/resources"/>
  </body>
 </html>
</xsl:template>

<xsl:template match = "link">
 <a href ="{@to}">
  <xsl:apply-templates/>
 </a><br/>
</xsl:template>

</xsl:stylesheet>
```

Примечание

Те строки, которые содержат символ ↪, разорваны в тексте при его форматировании, но на самом деле они не должны содержать разрывов.

Далее следуют листинги, с помощью которых мы только один раз определяем многократно встречающиеся фрагменты XML- и XSLT-кода. Здесь левая правая колонки, а также та часть средней колонки, которая совместно используется всеми Web-страницами, находятся в файле panels.xml (листинг 4.3). соответствующие им правила шаблона, которые также будут применяться однократно, располагаются в файле common.xsl (листинг 4.4).

Листинг 4.3. Файл panels.xml

```
<?xml version="1.0" encoding="koi8-r"?>

<panels>

 <controls>
   <link to ="artist.xml">Исполнитель</link>
   <link to ="title.xml">Название</link>
   <link to ="year.xml">Год</link>
   <link to ="genre.xml">Жанр</link>
   <link to ="custom.xml">Многокритериальный</link>
 </controls>
<description>
  <para>
```

Предлагаем вашему вниманию каталог имеющихся в нашем распоряжении аудиофайлов. Здесь вы можете просмотреть их полный или частичный список. Получившийся в результате преобразования данного XML-документа HTML-код вы можете просмотреть, выбрав в строке меню Internet Explorer раздел "Вид", а в нем подраздел "В виде HTML"
</para>
<para>
Как видно, данная страница разбита на три части. Такая структура сайта довольно широко распространена в Internet. Первая из них, располагающаяся слева в окне браузера, представляет собой навигационную панель, с помощью которой вы можете выбрать требуемый критерий отбора файлов. Она представляет собой набор гиперссылок, каждая из которых связана с тем XML-документом, который производит поиск файла по критерию, соответствующему тексту данной гиперссылки.
</para>
<para>
Вторая часть — это та часть, которую вы в данный момент читаете. Она представляет собой описание функций, которые выполняет данная страница (начало текста для каждой страницы совпадает, и поэтому сейчас может быть загружена любая из шести имеющихся в наличии: main.xml, artist.xml, title.xml, genre.xml, year.xml и custom.xml). Элемент, контентом которого является вторая часть страницы, носит имя description.
</para>
<para>
И наконец, третья часть, которая находится справа, содержит некоторые ресурсы Internet. Здесь, в частности, мы поместили ссылки на сайты тех компаний (или открытых проектов), программным обеспечением и документацией которых мы активно пользовались при ведении наших разработок.
</para>
</description>

```
 <resources>
   <link to="http:\\www.apache.org">www.apache.org</link>
   <link to="http:\\xml.apache.org">xml.apache.org</link>
   <link to="http:\\jacarta.apache.org">jacarta.apache.org</link>
   <link to="http:\\www.sun.com">www.sun.com</link>
 </resources>

</panels>
```

Листинг 4.4. Файл common.xsl

```
<?xml version="1.0" encoding="koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match ="controls">
 <span style="top:50px;width:15%;margin-right:5%">
  <p>Выберите критерий для поиска</p>
  <xsl:apply-templates/>
 </span>
</xsl:template>

<xsl:template match ="description">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match ="resources">
 <span style="top:50px;width:10%;vertical-align:top;">
  <p>Ресурсы</p>
  <xsl:apply-templates/>
 </span>
</xsl:template>

<xsl:template match ="para">
 <p style="text-indent:1.5em">
  <xsl:apply-templates/>
 </p>
</xsl:template>

</xsl:stylesheet>
```

После внимательного изучения рис. 4.1, отображающего результаты обработки этих файлов, можно приступить к последовательному описанию того, как работают вышеприведенные сценарии XSLT.

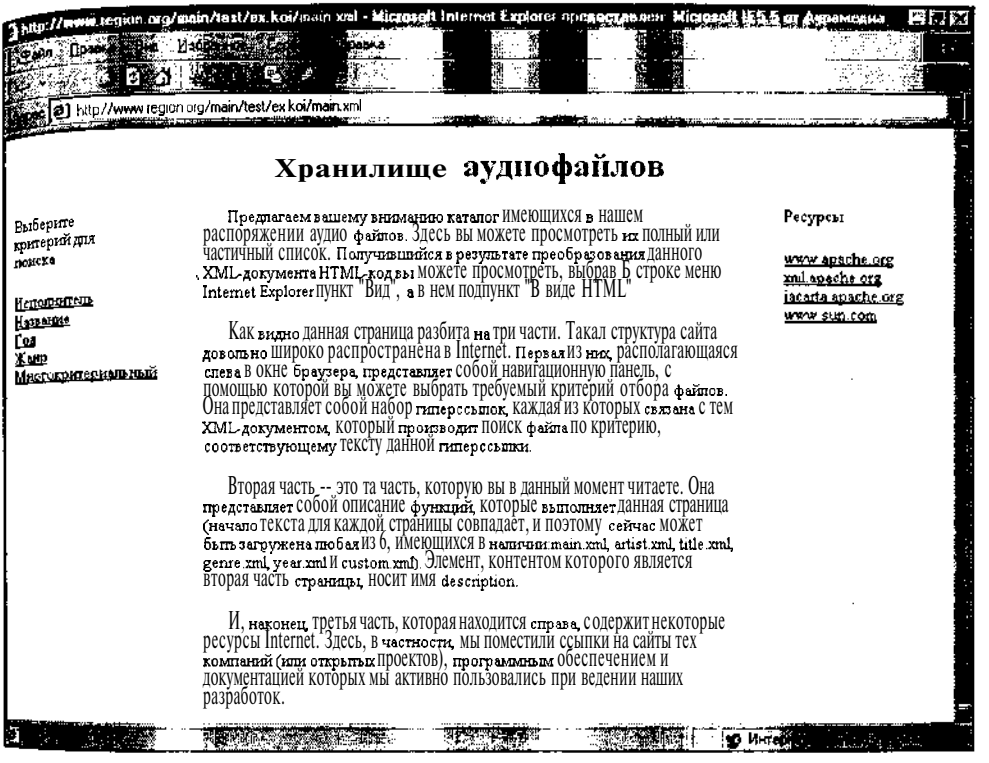


Рис. 4.1. Главная страница

Листинг 4.2, *а* не нуждается в подробных комментариях, т. к. корень этого Документа содержит только одного потомка — элемент документа main, представляющий главную страницу. Это обусловлено тем, что данный документ не предоставляет никаких средств поиска, он служит лишь для навигации между остальными страницами.

Некоторого внимания заслуживает лишь пролог этого документа. Заметим, что в объявлении XML мы указали используемую нами кодировку, во второй строке подключили стилевую таблицу main.xsl, файл которой должен находиться в том же каталоге, что и main.xml. И наконец, в третьей строке мы определили инструкцию обработки cocoon-process, значение которой `type="xslt"` говорит Cocoon о том, что данный XML-файл должен быть преобразован в соответствии с правилами XSLT (напомним, что XSLT — это не единственное расширение XSL, мы могли бы преобразовать данный файл в соответствии, например, с XSL Formatting Objects).

Рассмотрим построчно файл main.xsl (см. листинг 4.2, б). После заголовка точно такого же, как и у main.xml, следует элемент стилевой таблицы (или как его еще иногда называют, элемент стиля) xsl:stylesheet, который объявляет пространство имен XSLT. Далее, с помощью директивы xsl:include в данную таблицу включается еще одна стилевая таблица — common.xsl. Как следует из ее названия (common — общий), она будет включена во все остальные пять Web-страниц нашего сайта. В этих (а также в главной) Web-страницах мы используем одни и те же элементы, для которых определены одни и те же правила шаблона. Эти правила шаблона мы собрали в файле common.xsl.

В следующей строке с помощью директивы xsl:variable мы определяем переменную с именем panels. Значением этой переменной будет набор узлов, состоящий из единственного (т. к. в документе он один) элемента panels, который является элементом документа, представленного файлом panels.xml. Для того чтобы адресовать этот набор узлов, мы воспользовались путем адресации document('panels.xml')/panels, который состоит из двух шагов адресации: document('panels.xml') И panel Первый шаг адресации (слева направо) описывает корень документа, представленного файлом panels.xml. На втором шаге мы адресуем все элементы с именем panels, являющиеся непосредственными потомками корня документа.

Прежде, чем продолжить описание сценария main.xsl, рассмотрим файлы panels.xml (см. листинг 4.3) и common.xsl (см. листинг 4.4).

В файле panels.xml определяются те три колонки или панели, из которых будет состоять каждая Web-страница нашего сайта. Левая колонка представлена элементом controls, средняя — description, правая — resources соответственно. Все эти три элемента являются непосредственными потомками корневого элемента panels.

Левая колонка содержит ссылки на остальные Web-страницы сайта. Каждая из этих ссылок представлена элементами с именем link, являющимися непосредственными потомками элемента controls. Контент элемента li представляет собой текст соответствующей гиперссылки, а файл Web-страницы, на которую она ссылается, представлен значением атрибута Из точно таких же элементов link состоит и контент элемента resources, представляющего правую колонку.

Средняя колонка представлена элементом description (описание), контент которого содержит только три непосредственных потомка с именем para (параграф). Каждый из элементов para содержит только текстовый узел и никаких других потомков. Элемент description определяет текст, разбитый на три параграфа, который будет представлен на каждой Web-странице и помещенный поэтому в отдельный файл.

Так как каждая из Web-страниц будет состоять из трех колонок, содержимое которых либо полностью совпадает, либо незначительно отличается, то в каждой из этих страниц мы определим переменную panels так, как мы это сделали в сценарии main.xsl. Это избавит нас от многократного перепи-сывания XML-кода, описанного в двух предыдущих абзацах.

После того, как мы поместили многократно используемый XML-код в файл panels.xml, нужно создать правила шаблона для этого кода. Для этих целей и служит сценарий common.xsl. В нем определены правила шаблона соответственно для **элементов** controls, description, resources И para. **Заметим** здесь, что правило шаблона для элемента link в этом сценарии не задано, т. к содержимое левой колонки будет немного отличаться для каждой Web-страницы. Рассмотрим каждое из этих правил по отдельности.

Согласно шаблону, являющемуся контентом конструкции `<xsl:template match = "controls">... </xsl:template>`, элемент controls ИСХОДНОГО дерева будет трансформирован в элемент span конечного дерева. Причем для данного элемента span определен атрибут style, а также контент, состоящий из элемента p конечного дерева и результатов трансформации всех непосредственных потомков элемента controls, за что отвечает директива `xsl:apply-templates`. Это означает, что после элемента p в конечное дерево будут вставлены как трансформированные непосредственные потомки controls, так и результаты применения ко всем элементам link соответствующего правила шаблона.

Правило шаблона для элемента description содержит только одну инструкцию — трансформировать данный элемент исходного дерева, обработав всех его непосредственных потомков. Иными словами, в конечном дереве на месте элемента description будут находиться результаты применения правила шаблона ко всем элементам para, являющимся непосредственными потомками элемента description.

Элемент resources трансформируется так же, как и элемент controls. Различия состоят только в контенте элемента p конечного дерева, а также в значении атрибута style элемента span также конечного дерева. Действительно, на рис. 4.1 легко заметить, что левая и правая колонки отличаются только своим подзаголовком и местоположением на Web-странице (слева и справа).

Последнее правило шаблона в сценарии common.xsl определяет правило трансформации для элемента para. Как видно из листинга 4.4, этот элемент трансформируется в элемент p конечного дерева, для которого задан определенный стиль, представленный значением атрибута style. Контентом элемента p будет значение текстового узла элемента para. Поскольку para включает только текст и не содержит разметку, то во время обработки этого элемента контент станет контентом элемента p.

Теперь мы можем продолжить описание сценария main.xsl. Он содержит только два правила шаблона: для элемента документа main и для элемента link, определенного в файле panels.xml.

Первой строкой шаблона трансформации элемента main является директива создания инструкции обработки с именем cocoon-format и значением text/html. Эта инструкция сообщает процессору, что форматом вывода конечного документа должен быть text/html. Она является эквивалентом предлагаемым Cocoon, для директивы xsl:output, которую последний не поддерживает.

Примечание

Для более детального управления выводом конечного дерева в Cocoon предназначены специальные настроечные файлы, которые мы не будем описывать в этой книге, т. к., во-первых, существует директива xsl:output, определенная в спецификации XSLT, и мы, выбирая между спецификацией какого-либо компонента XML и его конкретной реализацией, отдаем предпочтение первой (остальные расширения Cocoon мы описываем, потому что они в спецификации не определены вовсе, и, вместе с тем, являются необходимыми при разработке нетривиальных задач). Во-вторых, информацию по конфигурированию Cocoon можно найти на сайте разработчика.

Мы не будем подробно описывать эту инструкцию обработки потому, что формат вывода не имеет большого значения в нашей задаче (а также всех остальных, поставленных далее в книге).

После создания инструкции обработки формируется корневой элемент конечного дерева с именем html, который имеет одного непосредственного потомка (не считая текстовых узлов, состоящих лишь из пробельных символов) с именем body. Контентом последнего в конечном дереве будет:

1. Заголовок Web-страницы, представленный элементом конечного дерева h1.
2. Результат обработки элемента controls, определенного в panels.xml и заданного выражением $panels/controls, которое является значением атрибута select первой встретившейся в файле common.xsl директивы xsl:apply-templates. Как видим, здесь демонстрируется использование ссылки на переменную в качестве шага адресации, что нигде ранее не запрещалось.
3. Элемент конечного дерева span, контентом которого будет результат обработки элемента description, определенного в panels.xml.
4. И наконец, результат обработки элемента resources, определенного также в panels.xml.

Вторым и последним правилом шаблона будет правило для элемента link. Данный элемент трансформируется в гиперссылку, которая представлена эле-

ментом а конечного дерева. Причем контентом элемента *a* будет контент link, а значением атрибута href — значение атрибута to, принадлежащего link.

Скажем еще несколько слов о совместно используемых XML-фрагментах и соответствующих им правилах шаблона. Для хранения адреса набора узлов исходного дерева мы объявили переменную, далее для адресации всего этого набора или каких-либо его частей в соответствующих выражениях XPath служила ссылка на эту переменную. Чтобы применить определенные во внешнем источнике правила шаблона, мы применяли механизм включения стилевых таблиц.

Здесь возникает вопрос: а почему бы эту стилевую таблицу не подключить непосредственно к совместно используемому XML-файлу? Другими словами в код файла panels.xml не добавить строку

```
<?xml-stylesheet href = "common.xsl" type = "text/xsl"?>
```

Так тоже можно было бы сделать, но тогда нам пришлось бы каким-либо образом заставить обработать файл panels.xml вместе с common.xsl, а мы можем это сделать (не выходя за рамки используемых спецификаций), только обратившись к модифицированному panels.xml по гиперссылке. В то же время, каждая из остальных страниц сайта помимо общей для всех части имеет и свое собственное содержимое, которое представлено только на ней. Таким образом, получается, что должна быть создана такая гиперссылка, которая адресовала бы сразу два файла: panels.xml и тот, который соответствует Web-странице с нужным пользователю режимом поиска.

Очевидно, что таких гиперссылок не существует. Кроме того, не существует такого браузера, который был бы способен загружать одновременно две Web-страницы или какую-либо их комбинацию.

Примечание

В подобной ситуации мы можем лишь попытаться создать такой XML-документ, который мог бы в зависимости от параметров HTTP-запроса обрабатываться по-разному, т. е. генерировать различный вывод. Иными словами, в результирующем документе должны быть представлены только те трансформированные узлы исходного дерева, которые удовлетворяют условиям, определенным параметрами HTTP-запроса.

Скажем сразу, что средствами, очерченными спецификацией XML, такой XML-документ, трансформация которого происходит в зависимости от внешних условий (сложившихся вне документа, например, определенных действием пользователя, посетившего сайт), в то время как обработка этих условий является внутренней (то есть определена в самом XML-документе), создать невозможно. Даже с помощью каких-либо дополнительных XML-технологий, не описанных в этой книге (таких, например, как XLink или XInclude), этого сделать не удастся. Более подробно об этой проблеме, а также о средствах ее решения можно прочитать в *главах 8 и 9*.

4.2. Генерирование списка исполнителей

Как видно из рис. 4.1, первая гиперссылка в левой колонке предлагает в качестве критерия поиска имя исполнителя композиции. Таким образом при активизировании этой гиперссылки пользователь перейдет на Web-страницу, обеспечивающую поиск аудиофайлов, представляющих такие композиции, исполнитель которых соответствует заданному образцу.

На самом деле с помощью XML-документа, соответствующего этой Web-странице, мы производить поиск не будем. Вместо этого мы просто выведем отсортированный список всех исполнителей, которые представлены в файле storage.xml. Для начала этого будет вполне достаточно, тем более что в остальных XML-документах тот или иной вид поиска мы реализуем.

Рассмотрим листинги данного XML-документа.

Листинг 4.5, *а*. Список исполнителей. XML-составляющая (artist.xml)

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type = "xsp"?>
<?cocoon-process type = "xslt"?>
<?xml-stylesheet href = "artist.xsl" type = "text/xsl"?>

<xsp:page language = "java"
          xmlns:fp="http://apache.org/cocoon/XSP/FP/1.0"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

<main>
 <description>
  <para>
   Эту часть описания вы можете увидеть только,
   если загружена страница, представленная
   результатом преобразования файла artist.xml.
   Хотя эта страница имеет заголовок "Поиск по
   исполнителю", никакого поиска здесь не ведется.
   Вместо этого просто выводится отсортированный
   список всех исполнителей, сведения о которых
   содержатся в файле storage.xml.
   Обратите внимание, что в левой панели сейчас
   отсутствует ссылка на artist.xml. Так сделано потому,
   что этот файл загружен в данный момент.
```

```
</para>
</description>
<artists_list/>
</main>
</xsp:page>
```

Заметим, что файл artist.xml по своей структуре не сильно отличается от main.xml, как, впрочем, и все приводимые ниже XML-файлы. Но несколько принципиальных отличий все же имеется.

Во-первых, добавлена представленная второй строкой листинга 4.5, *а* инструкция обработки с именем cocoon-process и значением xsp. Она указывает процессору Cocoon о том, что в результате обработки данного XML-документа должен быть сгенерирован ему соответствующий Java-код, который будет скомпилирован только один раз — во время первого обращения пользователя к документу. Таким образом, если при обращении к main.xml Cocoon трансформировал XML-документ в формат HTML, а затем предоставлял результат пользователю, то при обращении к файлу, содержащему вышеописанную инструкцию обработки, Cocoon перенаправит запрос соответствующему байтовому коду Java, который и сгенерирует документ в формате HTML. Подробнее об этом механизме можно прочитать в последнем разделе *главы 6.*

Необходимым условием для трансформации документа в Java-код является то, что именем корневого элемента должно быть `xsp:page`, который и представлен в рассматриваемом листинге. Об атрибутах этого элемента, а также о подключаемом им пространстве имен с префиксом `xsp:` также можно прочесть в *главе 6.*

Как видно из листинга 4.5, *а*, в отличие от main.xml, элемент main здесь уже не пуст. Он содержит описание данной Web-страницы, которое будет видно только на ней. Это описание представлено единственным параграфом para, являющимся непосредственным потомком элемента description.

Кроме описания в main находится элемент artists_list, представляющий собой узел исходного дерева, который в результате обработки будет трансформирован в список исполнителей (см. рис. 4.2).

Листинг 4.5, *б*. Список исполнителей. XSL-составляющая (artist.xsl)

```
<?xml version="1.0" encoding="koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:include href = "common.xsl"/>
<xsl:variable name = "panels" select = "document('panels.xml')/panels"/>
<xsl:variable name = "afiles"
              select = "document('storage.xml')/audiofiles/audiofile"/>
```

```
<xsl:template match = "main">
 <xsl:processing-instruction name = "cocoon- for-
mat">type="text/html"</xsl:processing-instruction>
 <html>
  <body>
    <h1 align = "center">Поиск по исполнителю</h1>
    <p align = "center">
       <a href = "main.xml">Главная</a>
    </p>
    <xsl:apply-templates select = "$panels/controls"/>
    <span style="top:50px;width:60%;vertical-align:top;margin-right:5%">
      <xsl:apply-templates select = "$panels/description/para"/>
      <xsl:apply-templates/>
    </span>
    <xsl:apply-templates select = "$panels/resources"/>
    <p align = "center">
       <a href = "main.xml">Главная</a>
    </p>
  </body>
 </html>
</xsl:template>

<xsl:template match = "artists_list">
 <div>
   Список исполнителей<br/><br/>
   <ol>
     <xsl:apply-templates select = "$afiles/artist">
        <xsl:sort select = "."/>
     </xsl:apply-templates>
   </ol>
 </div>
</xsl:template>

<xsl:template match = "artist">
  <xsl:if test = "not(../preceding-sibling::*/artist
                  [text()=current()/text()])">
      <li>
        <b><xsl:apply-templates/></b>
      </li><br/>
```

```
  </xsl:if>
</xsl:template>

<xsl:template match = "link">
  <xsl:if test = "not(normalize-space(@to)='artist.xml')">
    <a href = "{@to}">
      <xsl:apply-templates/>
    </a><br/>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Более серьезного внимания, чем файл XML, заслуживает сценарий, представленный в файле artist.xsl (см. листинг 4.5, *б)* и подключаемый к XML-файлу в четвертой строке листинга 4.5, *а* (получившийся в результате HTML-код см. далее в листинге 4.6). Точно так же, как и main.xsl, данный сценарий обеспечивает трехколоночную структуру Web-страницы посредством включения стилевой таблицы common.xsl и определения переменной, адресующей XML-фрагмент файла panels.xml.

Но кроме этих конструкций в artist.xsl, определен еще один внешний источник данных. Подобно переменной panels здесь определена еще и переменная `afiles`, которая адресует все элементы `audiofile`, являющиеся непосредственными потомками элемента `audiofiles`, который, в свою очередь, является непосредственным потомком корня Документа, представленного файлом storage.xml (см. листинг 4.1). Использование этой переменной мы опишем несколько позже, чтобы не нарушать последовательность изложения.

Итак, по порядку. Рассмотрим, в чем состоят отличия правила шаблона для элемента main, описанные в листингах 4.2, *б* и 4.5, *б.* Во-первых, мы, естественно, изменили заголовок Web-страницы, в листинге 4.5, *б* она называется "Поиск по исполнителю". Во-вторых, в начале и в конце шаблона мы добавили **ссылку на главную страницу** (`<a href = "main.xml">Главная</a>`), которая будет располагаться вверху и внизу средней колонки, и, в-третьих, tow теперь по-другому формируем внутреннее содержимое этой колонки. Это можно заметить, если соотнести контенты представляющих среднюю колонку элементов span в сравниваемых листингах.

Рассмотрим, как строится содержимое средней колонки согласно листингу 4.5, *б.* Сначала в элемент span вставляется общая часть, представленная результатом обработки набора узлов, заданных выражением `$Panels/description/para`. Далее в нее встраивается результат последо-

вательной обработки всех непосредственных потомков элемента . Это означает, что если пронумеровать все компоненты средней колонки сверху вниз, то этот список будет выглядеть так:

1. Заголовок.
2. Ссылка на главную страницу.
3. Общая часть.
4. Описание данной страницы.
5. Список исполнителей.
6. Ссылка на главную страницу.

Возможно, вставка в каждую страницу описания всего сайта и не является хорошей идеей, но с другой стороны, этим мы можем продемонстрировать, как сформировать конечный документ из данных, находящихся как в самом этом документе, так и во внешних источниках. Здесь под данными подразумеваются и XML-фрагменты, и правила шаблона, расположенные во включаемых таблицах стилей.

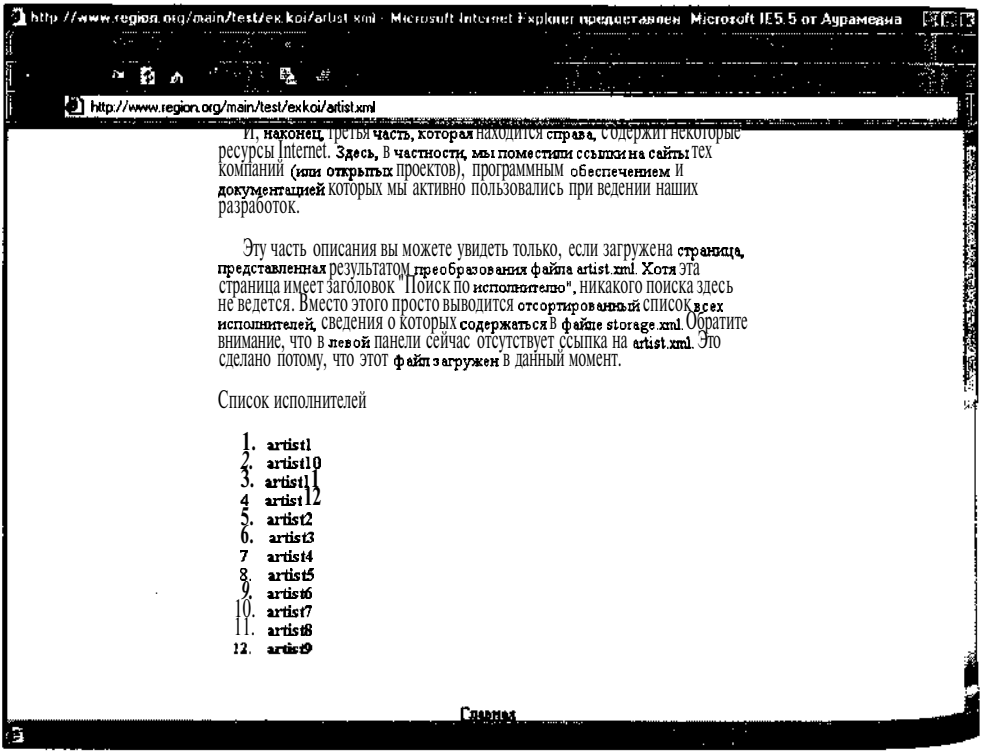


Рис. 4.2. Сгенерированный список исполнителей

Следующим в artist.xsl будет правило шаблона для элемента artists_list. В результате трансформации этого элемента в конечное дерево будет добавлен элемент div, содержащий подзаголовок "Список исполнителей" и непосредственно сам этот список, который, в свою очередь, будет представлен элементом ol. Далее будет сформирован контент ol посредством трансформации всех элементов artist, являющихся непосредственными потомками узлов из набора, адрес которого содержится в переменной afiles, т. е. всех элементов artist из файла storage.xml. Причем эти элементы будут отсортированы с помощью директивы xsl:sort.

В последующих двух правилах шаблона для элементов artist и link соответственно, примечательным является использование директивы xsl:if. В первом случае правило шаблона трансформирует элемент только тогда, когда до этого не был обработан элемент artist с точно таким же контентом. Иными словами, файл storage.xml содержит несколько аудиофайлов одного и того же исполнителя, следовательно, при выводе списка без такой проверки исполнители в этом списке будут повторяться, т. е. один и тот же исполнитель встретится несколько раз под разными номерами.

Рассмотрим выражение, представляющее собой значение атрибута test директивы xsl:if в правиле шаблона для artist. Это значение представляет собой функцию not() с некоторым аргументом, что говорит о том, что если результат вычисления значения аргумента истинный (true), то условие xsl:if не будет выполнено и соответствующий шаблон окажется пустым, г. е. элемент artist будет представлен в конечном дереве пустым набором узлов. И наоборот, artist будет обработан тогда, когда аргумент функции not о будет вычислен как false.

Аргументом функции not о является некоторый путь адресации, который будет преобразован в false только тогда, когда адресуемый им набор узлов не пуст. Разбирая слева направо путь адресации ../preceding-sibling::*/artist[text()=current0 /text o], определим этот набор узлов.

Итак, первым шагом адресации является лексема .., которая, согласно правилам сокращенного синтаксиса, эквивалентна parent::node o. Текущим элементом является artist (т. к. директива xsl:if расположена в соответствующем ему правиле шаблона), следовательно, лексема .. будет описывать родителя обрабатываемого в данный момент элемента artist. Как видно из листинга 4.1 родителем каждого artist является audiofile.

```
<audiofile>
   <artist>... </artist>
   <title>... </title>
  ...
</audiofile>
```

Следовательно, на втором шаге адресации (preceding-sibling::*) будут отобраны все братья данного узла audiofile, которые ему предшествуют.

```
<audiofiles>
    <audiofile>
       ...
    </audiofile>
...
    <audiofile>
       ...
    </audiofile>
    <audiofile>
        <artist>... </artist>
        <title>... </title>
       ...
    </audiofile>
    <audiofile>
       ...
    </audiofile>
   ...
</audiofiles>
```

Далее, на третьем шаге адресации (artist[text()=current()/text o]) для каждого из элементов audiofile, отобранных на предыдущем шаге, будет выбран такой его непосредственный потомок, именем которого является artist и чей текстовый контент совпадает с текстовым контентом текущего узла (то есть обрабатываемого в данный момент элемента artist).

Таким образом, если в данный момент обрабатывается такой элемент artist. что где-либо перед ним в документе существует аудиофайл с тем же исполнителем, то он не должен быть обработан или, как мы уже говорили ранее, этот artist будет представлен в конечном дереве пустым набором узлов.

Рассмотрим теперь принцип работы директивы xsl:if, расположенной в правиле шаблона для элемента link. Значением ее атрибута test является not(normalize-space(@to)='artist.xml'). Как работает функция not() мы уже разобрали в предыдущих абзацах, поэтому не будем останавливаться на ней подробно, а сразу перейдем к ее аргументу. Его значение представляет собой сравнение двух операндов: normalize-space(@to) И 'artist.xml'.

Первый представляет собой результат преобразования в строку и нормализации значения атрибута to текущего узла, которым является обрабатываемый в данный момент элемент link. Напомним, что нормализация заключается в удалении пробельных символов в начале и конце строки.

Значением второго атрибута является строка artist.xml. Таким образом, условие xsl:if будет выполнено, когда нормализованное значение атрибута to, текущего элемента link не будет совпадать со строковым значением artist.xml. Иными словами, т. к. текущим документом (представляющим загруженную в данный момент Web-страницу) является artist.xml, то у нас нет необходимости среди ссылок в левой колонке располагать гиперссылку на сам этот документ, т. е. загружать тот документ, который уже загружен.

Далее следует листинг, представляющий собой результирующий HTML-код. В нем следует обратить внимание, во-первых, на структуру получившейся Web-страницы, во-вторых, на те гиперссылки, которые представлены в левой панели, а также на отсортированный список исполнителей в средней колонке.

листинг 4.6. Результат обработки artist.xml и artist.xsl

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html40/strict.dtd">
<html><body>
<h1 align="center">Поиск по исполнителю</h1>
<p align="center"><a href="main.xml">Главная</a></p>
<span style="top:50px;width:15%;margin-right:5%">
  <p>Выберите критерий для поиска</p>
  <a href="title.xml">Название</a><br>
  <a href="genre.xml">Жанр</a><br>
  <a href="year.xml">Год</a><br>
  <a href="custom.xml">Многокритериальный</a><br>
</span><span style="top:50px;width:60%;vertical-align:top;margin-
right:5%"><p style="text-indent:1.5em">
Предлагаем вашему вниманию каталог имеющихся в нашем распоряжении аудио-
файлов. Здесь вы можете просмотреть их полный или частичный список. Полу-
чившийся в результате преобразования данного XML-документа HTML-код вы
можете просмотреть, выбрав в строке меню Internet Explorer пункт "Вид", а
в нем подпункт "В виде HTML".
</p><p style="text-indent:1.5em">
Как видно, данная страница разбита на три части. Такая структура сайта
довольно широко распространена в Internet. Первая из них, располагающаяся
слева в окне браузера, представляет собой навигационную панель, с помощью
которой вы можете выбрать требуемый критерий отбора файлов. Она представ-
ляет собой набор гиперссылок, каждая из которых связана с тем XML-
документом, который производит поиск файла по критерию, соответствующему
тексту данной гиперссылки.
```

```
</p><p style="text-indent:1.5em">
```

Вторая часть — это та часть, которую вы в данный момент читаете. Она представляет собой описание функций, которые выполняет данная страница (начало текста для каждой страницы совпадает, и поэтому сейчас может быть загружена любая из шести имеющихся в наличии: main.xml, artist.xml, title.xml, genre.xml, year.xml и custom.xml). Элемент, контентом которого является вторая часть страницы, носит имя description.

```
</p><p style="text-indent:1.5em">
```

И наконец, третья часть, которая находится справа, содержит некоторые ресурсы Internet. Здесь, в частности, мы поместили ссылки на сайты тех компаний (или открытых проектов), программным обеспечением и документацией которых мы активно пользовались при ведении наших разработок.

```
</p><p style="text-indent:1.5em">
```

Эту часть описания вы можете увидеть только, если загружена страница, представленная результатом преобразования файла artist.xml. Хотя эта страница имеет заголовок "Поиск по исполнителю", никакого поиска здесь не ведется. Вместо этого просто выводится отсортированный список всех исполнителей, сведения о которых содержатся в файле storage.xml. Обратите внимание, что в левой панели сейчас отсутствует ссылка на artist.xml. Так сделано потому, что этот файл загружен в данный момент.

```
</p>
<div> Список исполнителей
    <br><br><ol><li><b>artist1</b><br>
                <lixb>artist10</bxbr>
                <lixb>artist11</bxbr>
                <li><b>artist12</bxbr>
                <lixb>artist2</bxbr>
                <lixb>artist3</bxbr>
                <lixb>artist4</bxbr>
                <lixb>artist5</bxbr>
                <lixb>artist6</b><br>
                <lixb>artist7</b><br>
                <lixb>artist8</b><br>
                <lixb>artist9</b><br>
    </olx/div>
</span>
<span style="top:50px;width:10%;vertical-align:top;">
   <p>Ресурсы</p>
   <a href="http:\\www.apache.org">www.apache.org</a><br>
   <ahref="http:\\xml.apache.org">xml.apache.org</a><br>
```

```
  <ahref="http:\\jacarta.apache.org">jacarta.apache.org</a><br>
  <a href="http:\\www.sun.com">www.sun.com</a><br>
</span>
  <p align="center"><a href="main.xml">Главная</a></p>
 </body>
</html>
<!__ This page was served in 2128 milliseconds by Cocoon 1.8.2 -->
```

4.3. Поиск по названию композиции

Вторая гиперссылка на главной странице предоставляет возможность пользователю отыскать такие аудиофайлы, название композиции которых полностью совпадает с введенным значением. Рассмотрим соответствующие листинги.

Листинг 4.7, а Поиск по названию композиции. XML-составляющая (title.xml)

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type = "xsp"?>
<?cocoon-process type = "xslt"?>
<?xml-stylesheet href = "title.xsl" type = "text/xsl"?>

<xsp:page language = "java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

<main>
 <description>
  <para>
   Эту часть описания вы можете увидеть только,
   если загружена страница, представленная
   Результатом преобразования файла title.xml.
   Здесь вы можете просмотреть список, состоящий
   из реквизитов всех тех аудиофайлов, у которых
   название композиции совпадает с вводимым далее
   значением. И если при поиске по исполнителю мы
   пользовались директивой &lt;xsl:value-of
   select ="<i>выражение</i>"/&gt; для вывода только
   лишь непосредственно имени исполнителя, то в данном
   случае мы задействовали директиву &lt;
```

```
    xsl:apply-templates select="<i>выражение</i>"/&gt;
    для получения всех характеристик файла, которому
    соответствует введенное ниже название композиции.
  </para>
 </description>
 <title_list>
   <xsp:attribute name = "title">
     <xsp:expr>request.getParameter("title")</xsp:expr>
   </xsp:attribute>
   <list/>
   <confirm/>
 </title_list>
</main>
</xsp:page>
```

Практически все конструкции этого файла мы уже описали ранее. Здесь внимания заслуживает только элемент `title_list`, а также его контент. Этот элемент с помощью соответствующего правила шаблона будет преобразован в форму для ввода названия композиции, соответствующий которой аудиофайл(ы) требуется найти, а также в том случае, когда поиск был уже произведен. Таким образом, элемент `title_list` помимо формы будет генерировать список найденных аудиофайлов.

Контент элемента `title_list` содержит три конструкции. Первая из них представляет собой элемент `xsp:attribute` *(см. главу 6),* являющийся расширением Cocoon. Этот элемент позволяет в исходном дереве до трансформации документа создать атрибут с заданным именем и значением. Для задания имени атрибута служит атрибут `name` элемента `xsp:attribute`, а для присвоения ему значения — контент этого элемента. В данном случае элемент `xsp:attribute` используется для передачи параметров HTTP-запроса. Для получения значений этих параметров служит метод `getParameter()` объекта `request` *(см. главы 6 и 8).*

После элемента `xsp:attribute` следует элемент `list`, представляющий собой отсортированный список всех имеющихся в хранилище композиций. Третьей конструкцией контента `title_list` является элемент `confirm`, который трансформируется в кнопку подтверждения ввода для формы. Элементы `list` и `confirm` трансформируются в компоненты формы, в которую, в свою очередь, трансформируется элемент `title_list`.

Рассмотрим принцип работы этой страницы, заметив, что все описанные далее Web-страницы этого примера работают точно так же. После загрузки в окне браузера появляется содержимое страницы в виде трех колонок.

Средняя колонка после текста описания включает в себя список всех имеющихся названий композиций, содержащихся в файле storage.xml. Этот список представлен элементами select и option языка HTML. После списка расположена кнопка **Искать** (см. рис. 4.3).

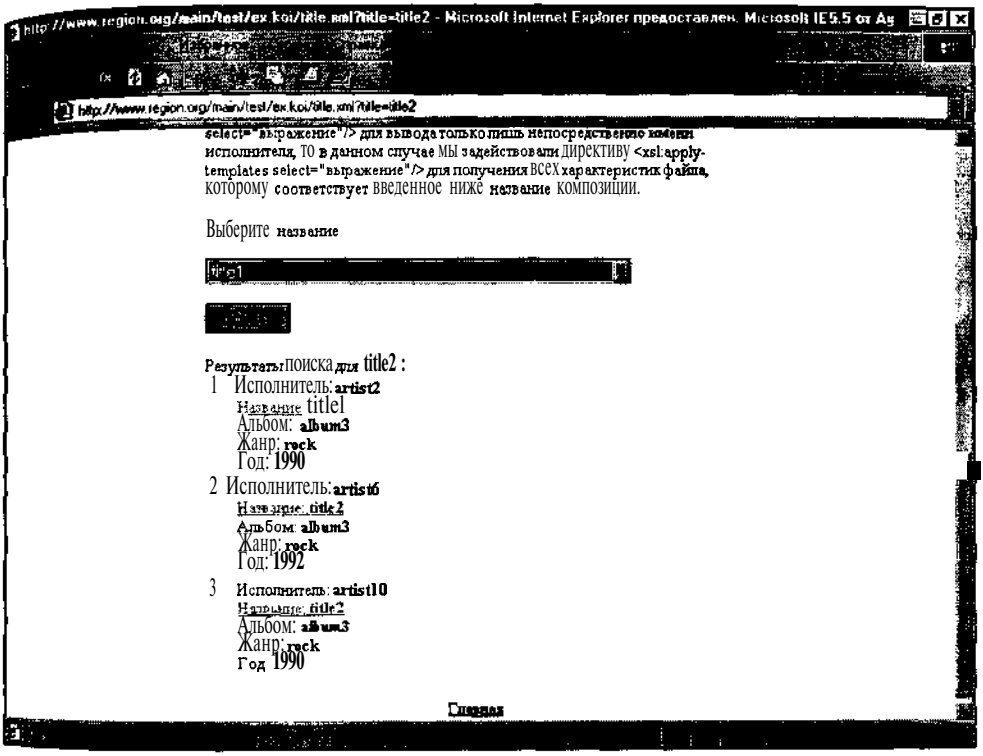


Рис. 4.3. Поиск по названию композиции

После того как требуемое название композиции выбрано, пользователь должен нажать кнопку подтверждения ввода. Тогда название композиции будет передано файлу title.xml в качестве параметра HTTP-запроса. Далее страница перезагрузится и будет содержать помимо формы (для возможного продолжения поиска) еще и список найденных аудиофайлов. Более подробно о формах и передаваемых с помощью них параметрах можно прочитать в *главе 8*.

Обработчиком формы является та страница, которая эту форму содержит. Так как мы в качестве значения атрибута `title` элемента `title_list` передали название искомой композиции, то в результате обработки документа будет сгенерирован список тех элементов `audiofile`, у которых контент непосредственного потомка `title` совпадает со значением вышеуказанного атрибута.

Листинг 4.7, б. Поиск по названию композиции. XSL-составляющая (title.xsl)

```
<?xml version="1.0" encoding="koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:include href = "common.xsl"/>
<xsl:include href = "audiofile.xsl"/>
<xsl:variable name = "panels" select = "document( 'panels.xml' ) /panels"/>
<xsl:variable name = "afiles" select =
              "document('storage.xml')/audiofiles/audiofile"/>

<xsl:template match = "main">
 <xsl:processing-instruction name = "cocoon-
format">type="text/html"</xsl:processing-instruction>
 <html>
  <body>
    <h1 align = "center">Поиск по названию</h1>
    <p align = "center">
       <a href = "main.xml">Главная</a>
    </p>
    <xsl:apply-templates select = "$panels/controls"/>
    <span style="top:50px;width:60%;vertical-align:top;margin-right:5%">
      <xsl:apply-templates select = "$panels/description/para"/>
      <xsl:apply-templates/>
    </span>
    <xsl:apply-templates select = "$panels/resources"/>
    <p align = "center">
     <a href ="main.xml">Главная</a>
    </p>
  </body>
 </html>
</xsl:template>
<xsl:template match = "title_list">
  <a name = "search"/>
  <form action = "title.xml#search">
    <xsl:apply-templates/>
  </form>
  <xsl:if test = "normalize-space(@title)!='null'">
```

```
        Результаты поиска для
    <b><xsl:value-of select = "@title"/></b>:
    <xsl:apply-templates select =
         "$afiles[title= normalize-space (current()/@title) ]"/>
  </xsl:if>
</xsl:template>

<xsl: template match = "list">
   Выберите название<br/><br/>
   <select name = "title" style="width: 80%">
     <xsl:for-each select = "$afiles">
       <xsl:sort select = "title"/>
       <xsl:if test = "not (preceding-sibling::*/title [text()=
                      current ()/title/text()])">
          <option>
           <xsl:value-of select = "title"/>
          </option>
       </xsl:if>
     </xsl:for-each>
   </selectxbr/xbr/>
</xsl:template>

<xsl: template match = "confirm">
  <input type = "submit" value = "Искать"/>
</xsl:template>

<xsl: template match = "title">
  <div style="text-decoration:underline;color:#00AA00;">
   Название:
   <b>
      <xsl:apply-templates/>
   </b><br/>
  </div>
</xsl:template>

<xsl: template match = "link">
 <xsl:if test = "not(normalize-space(@to)='title.xml' )">
  <a href = "{@to}">
   <xsl:apply-templates/>
```

```
  </axbr/>
 </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Для того чтобы разобраться, как осуществляется поиск и происходит генерирование списка найденных аудиофайлов, рассмотрим соответствующий сценарий XSLT (листинг 4.7, *б*). Как видно, в этом сценарии присутствует множество уже описанных нами ранее конструкций, отвечающих за структуру Web-страницы, содержимое колонок и т. д. Мы не будем на них останавливаться, а рассмотрим только три наиболее интересных правила шаблона ДЛЯ элементов **title_list**, **list** И `confirm`.

В самом начале шаблона трансформирования располагается якорь `<a name = "search"/>`, который введен исключительно для удобства навигации. После подтверждения ввода страница будет перезагружена, причем первой отображающейся в окне строкой будет не первая строка этой страницы, а первая строка формы. Это становится понятным, если обратить внимание на атрибут action следующего за якорем элемента form.

Содержимым формы будет, как видно из листинга, результат обработки непосредственных ПОТОМКОВ Элемента form, КОТОРЫМИ ЯВЛЯЮТСЯ **list** И confirm. Далее за формой следуют результаты поиска в том случае, если он производился. Проверка того, производился этот поиск или нет, осуществляется с помощью директивы `xsl:if`, которая проверяет, равно ли нормализованное значение созданного ранее атрибута **title** строковому значению null. Оно будет равно null только тогда, когда страница загружается в первый раз, т. е. когда в строке HTTP-запроса (query string) параметр **title** отсутствует.

Если значение **title** не равно `'null'`, то с помощью директивы `xsl:value-of` выводится подзаголовок, сообщающий пользователю, для какого названия композиции производился поиск, и далее следует список найденных аудиофайлов. Для его создания используется директива `xsl:apply templates`, значение атрибута **select** которой представляет собой выражение `$afiles[title = normalize-space(current()/@title)]`. Эта директива применяет соответствующее правило шаблона только к искомым элементам.

Переменная `afiles`, определенная в начале сценария, адресует все элементы audiofile из файла storage.xml. Следовательно, вышеприведенное выражение адресует все элементы audiofile, у которых текстовый контент непосредственного потомка **title** равен нормализованному значению атрибута **title** текущего узла, которым является `title_list`. Так как в этом атрибуте хранится название искомой композиции, можно сказать, что производится поиск тех аудиофайлов, у которых название композиции совпадает с заданным.

Правило шаблона для элемента list служит для создания списка всех имеющихся в наличии названий композиций. Контейнером для этого списка является элемент select языка HTML, а для представления каждого элемента этого списка служит элемент option, также определенный в HTML. Здесь для полного перебора всех названий композиций, представленных в storage.xml, мы воспользовались директивой xsl:for-each, атрибут select которой представляет собой путь адресации $afiles, т. е. выражение, адресующее набор узлов, состоящий из всех элементов audiofile, находящихся в storage.xml.

Заметим, что в контенте директивы xsl:for-each присутствует директива xsl:sort, отвечающая за сортировку названий композиций, а также xsl:if, которая не допускает среди них повторов. Директива xsl:if здесь работает точно так же, как было рассмотрено в предыдущем разделе.

Последним из рассматриваемых в этом сценарии будет правило шаблона для элемента confirm. Оно состоит всего из одной строки, которая создает кнопку подтверждения ввода. Эта кнопка представлена здесь элементом input языка HTML, значением атрибута type которого является submit. Надпись на кнопе определяется значением другого атрибута — value.

Ниже следует листинг, представляющий собой результат совместной трансформации элементов title_list, list и confirm. Мы не будем здесь приводить весь результирующий HTML-код, т. к. большая его часть повторяет листинг 4.6.

```
<a name="search"></a>
<form action="title.xml#search">
  Выберите название<br><br>
  <select name="title" style="width:80%">
    <option>titlel
    <option>title2
    <option>title3
  </select><brxbr>
  <input type="submit" value="Искать">
</form>

 Результаты поиска для<b>titlel</b>:
 <table width="50%"><tr><td valign="top" width="10%">1</td><td>
Исполнитель:  <b>artist4</b><br>
  <div style="text-decoration:underline;color:#00AA00;">
Название: <b>titlel</bxbr>
```

```
  </div>
Альбом: <b>album2</b><br>
Жанр: <b>classic</bxbr>
Год: <b>1996</bxbr>
  </tdx/trx/table>

<table width="50%"><tr><td valign="top" width="10%">2</td><td>
Исполнитель: <b>artist8</b><br>
    <div style="text-decoration:underline;color:#00AA00;">
Название:    <b>titlel</bxbrx/div>
Альбом: <b>album2</b><br>
Жанр:  <b>jazz</bxbr>
Год: <b>1996</b><br>
  </tdx/trx/table>
<table width="50%"><tr><td valign="top" width="10%">3</td><td>
Исполнитель:<b>artist12</b><br>
    <div style="text-decoration:underline;color:#00AA00;">
Название: <b>titlel</b><br></div>
Альбом: <b>album2</b><br>
Жанр:  <b>rock</bxbr>
Год: <b>1996</bxbr>
  </tdx/trx/table>
```

Данный листинг был сгенерирован при поиске композиции с названием `title1` (см. листинг 4.1). Для обработки элементов, представленных в файлах storage.xml (`audiofile`, **artist**, **title** и т. д.), мы в начале стилевой таблицы title.xsl с помощью директивы `xsl:include` включили сценарий audiofile.xsl, в котором и хранятся соответствующие правила шаблона (листинг 4.8).

Листинг 4.8. Файл audiofile.xsl

```
<?xml version="1.0" encoding="koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="audiofile">
 <table width="50%">
  <tr>
```

```
    <td width="10%" valign="top">
     <xsl:value-of select="position()"/>
    </td>
    <td>
     <xsl:apply-templates/>
    </td>
   </tr>
  </table>
</xsl:template>

<xsl:template match="artist">
 Исполнитель: <b><xsl:apply-templates/></b><br/>
</xsl:template>
<xsl:template match="title">
 Название: <b><xsl:apply-templates/></b><br/>
</xsl:template>

<xsl:template match="album">
 Альбом: <b><xsl:apply-templates/></b><br/>
</xsl:template>

<xsl:template match="genre">
 Жанр: <b><xsl:apply-templates/></b><br/>
</xsl:template>

<xsl:template match="year">
 Год: <b><xsl:apply-templates/></b><br/>
</xsl:template>

</xsl:stylesheet>
```

Заметим, что и в сценарии title.xsl и в audiofile.xsl (см. листинг 4.8) присутствует правило шаблона для элемента `title`. При этом будет применено то из них, которое находится в title.xsl, т. к. его приоритет импорта выше.

Фактически мы переопределили ранее заданное правило шаблона и сделали это для того, чтобы выделить среди элементов списка найденных аудиофайлов те их части, которые соответствуют названиям композиций, а также, чтобы продемонстрировать принцип действия приоритетов. В описываемых Далее Web-страницах мы поступим точно так же — в той из них, которая Реализует поиск по жанру, мы выделим строки, содержащие название жанра, а в той, в которой критерием поиска является год, мы выделим, соот-

ветственно, строки, содержащие значение искомого года. Выделенные компоненты у нас выводятся полужирным (bold) подчеркнутым (underline) шрифтом зеленого (00AA00) цвета.

4.4. Поиск по году записи композиции

В этом разделе мы покажем, как использовать простейшие конструкции JavaScript в сценариях XSLT. Кроме этого мы рассмотрим предикаты, содержащие логические условия, оперирующие числовыми значениями. Вообще говоря, мы создали отдельный XML-документ, реализующий поиск по году записи, только для того, чтобы не загромождать описание последующих двух компонентов сайта, в которых используется инструментарий данного раздела. Рассмотрим соответствующие листинги.

Листинг 4.9, а. Поиск по году записи. XML-составляющая (year.xml)

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type = "xsp"?>
<?cocoon-process type = "xslt"?>
<?xml-stylesheet href = "year.xsl" type = "text/xsl"?>

<xsp:page language = "java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

<main>
 <description>
  <para>
   Эту часть описания вы можете увидеть только,
   если загружена страница, представленная
   результатом преобразования файла year.xml.
   Здесь можно получить список реквизитов тех аудиофайлов,
   год записи которых не меньше одного значения и не
   больше другого. Значения начального и конечного
   годов записи вводятся ниже. Причем в том случае,
   когда начальный год строго больше конечного, будет
   выдано сообщение об ошибке и поиск производиться не
   будет. Для поиска файлов, записанных в каком-то
   конкретном году, нужно выставить одно и то же
   значение для начального и конечного годов, которое
```

```
    совпадает с данным конкретным годом.
  </para>
 </description>

 <year_list>
   <xsp:attribute name = "from">
     <xsp:expr>request.getParameter("from")</xsp:expr>
   </xsp:attribute>
   <xsp:attribute name = "to">
     <xsp:expr>request.getParameter("to")</xsp:expr>
   </xsp:attribute>
   <lists/>
   <confirm/>
 </year_list>
</main>
</xsp:page>
```

Отметим, что в листинге 4.9, *а* элемент year_list представляет собой список значений годов записи, содержащихся в storage.xml (см. листинг 4.1) аналогично тому, как в листинге 4.7, *а* элемент title_list представляет список названий композиций. Элементы list и confirm также выполняют функции, эквивалентные своим аналогам из этого листинга. Но, разумеется, все три вышеперечисленных элемента трансформируются несколько по-другому, в соответствии с предназначением рассматриваемого XML-документа.

В контенте элемента year_list присутствуют два элемента xsp:attribute, которые создают атрибуты с именами соответственно from и to. Значения этих атрибутов представляют собой тот диапазон лет, к которому должно принадлежать значение года искомого аудиофайла. Эти значения передаются XML-документу с помощью одноименных параметров HTTP-запроса.

Принцип поиска аудиофайлов, соответствующих заданному диапазону лет, мы Рассмотрим сразу после листинга сценария XSLT, реализующего этот поиск.

Листинг 4.9, *б*. Поиск по году записи. XSL-составляющая (year.xsl)

```
<?xml version = "1.0" encoding = "koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:include href = "common.xsl"/>
<xsl:include href = "audiofile.xsl"/>
<xsl:variable name = "panels" select = "document('panels.xml')/panels"/>
```

```
<xsl:variable name ="afiles"
              select = "document('storage.xml')/audiofiles/audiofile"/>

<xsl:template match = "main">
 <xsl:processing-instruction name = "cocoon-
format">type="text/html"</xsl:processing-instruction>
 <html>
 <head>
   <script>
    function check(obj) (
    if (obj.from.item(obj.from.selectedIndex).innerText &gt;
        obj.to.item(obj.to.selectedIndex).innerText)
        alert('Ошибка: Значение начального года &gt; значения конечно-
го!'); else
        obj.submit();   }
   </script>
 </head>
  <body>
    <hl align ="center">Поиск по году</h1>
    <p align = "center">
       <a href = "main.xml">Главная</a>
    </p>
    <xsl:apply-templates select = "$panels/controls"/>
    <spanstyle="top:50px;width:60%;vertical-align:top;margin-right:5%">
      <xsl:apply-templates select = "$panels/description/para"/>
      <xsl:apply-templates/>
    </span>
    <xsl:apply-templates select = "$panels/resources"/>
    <p align = "center">
     <a href = "main.xml">Главная</a>
    </p>
  </body>
 </html>
</xsl:template>

<xsl:template match = "year_list">
  <a name = "search"/>
  <form id = "frm" action = "year.xml#search">
```

```
    <xsl:apply-templates/>
  </form>
  <xsl:if test = "normalize-space(@from) !='null'">
    Аудиофайлы, записанные не ранее
    <b><xsl:value-of select = "@from"/></b>
     и не позднее
    <b><xsl:value-of select = "@to"/></b> года:
<xsl:apply-templates select =
      "$afiles[number(year) &gt; = number(normalize-↳
space(current()/@from))]
              [number(year) &lt; = number(normalize-↳
space(current()/@to))]"/>
  </xsl:if>
</xsl:template>

<xsl:template match = "lists">
  Выберите диапазон лет<br/><br/>
  <table>
    <tr>
      <td>от:</td>
      <td>до:</td>
    </tr>
    <tr>
      <td>
        <select name = "from">
          <xsl:for-each select = "$afiles">
            <xsl:sort select = "year"/>
            <xsl:if test ="not(preceding-sibling::*/year[text()=
                          current()/year/text()])">
              <option>
                <xsl:value-of select = "year"/>
              </option>
            </xsl:if>
          </xsl:for-each>
        </select>
      </td>
      <td>
        <select name = "to">
```

```
            <xsl:for-each select = "$afiles">
              <xsl:sort select = "year"/>
              <xsl:if test = "not(preceding-sibling::*/year[text()=
                              current()/year/text()])">
                <option>
                  <xsl:value-of select = "year"/>
                </option>
              </xsl:if>
            </xsl:for-each>
          </select>
        </td>
      </tr>
    </table>
</xsl:template>

<xsl:template match = "confirm">
  <input type = "button" value = "Искать" onclick = "check(frm);"/>
</xsl:template>
<xsl:template match = "year">
  <div style="text-decoration:underline;color:#00AA00;">
   Год:
   <b>
      <xsl:apply-templates/>
   </bxbr/>
  </div>
</xsl:template>
<xsl:template match = "link">
 <xsl:if test = "not(normalize-space(@to)='year.xml')">
  <a href = "{@to}">
   <xsl:apply-templates/>
  </axbr/>
 </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

В средней колонке получившейся Web-страницы располагаются два списка тех значений лет, которые имеются в файле storage.xml (см. лис-

тинг 4.1). Каждый список представлен элементом select языка HTML. Для того чтобы эти списки и соответствующие им подписи были выровнены относительно друг друга, мы расположили их в ячейках таблицы 2x2, представленной элементом table конечного дерева. Этот элемент является частью шаблона для lists. После таблицы располагается кнопка **Искать,** подтверждающая ввод формы, в которой находится сама таблица и вышеописанные списки.

После того, как в левом списке пользователь выберет год, соответствующий нижней границе диапазона лет, а в правом — верхней, и нажмет кнопку **Искать,** Web-страница будет перезагружена, и под этой кнопкой появится список найденных аудиофайлов. Если при этом нижняя граница вводимого диапазона будет строго больше, чем правая, то будет выдано соответствующее сообщение об ошибке и поиск производиться не будет, т. е. страница не перезагрузится.

На рис. 4.4 показана Web-страница с результатами поиска, при котором были найдены все аудиофайлы, записанные не раньше 1992 и не позже 1996 года.

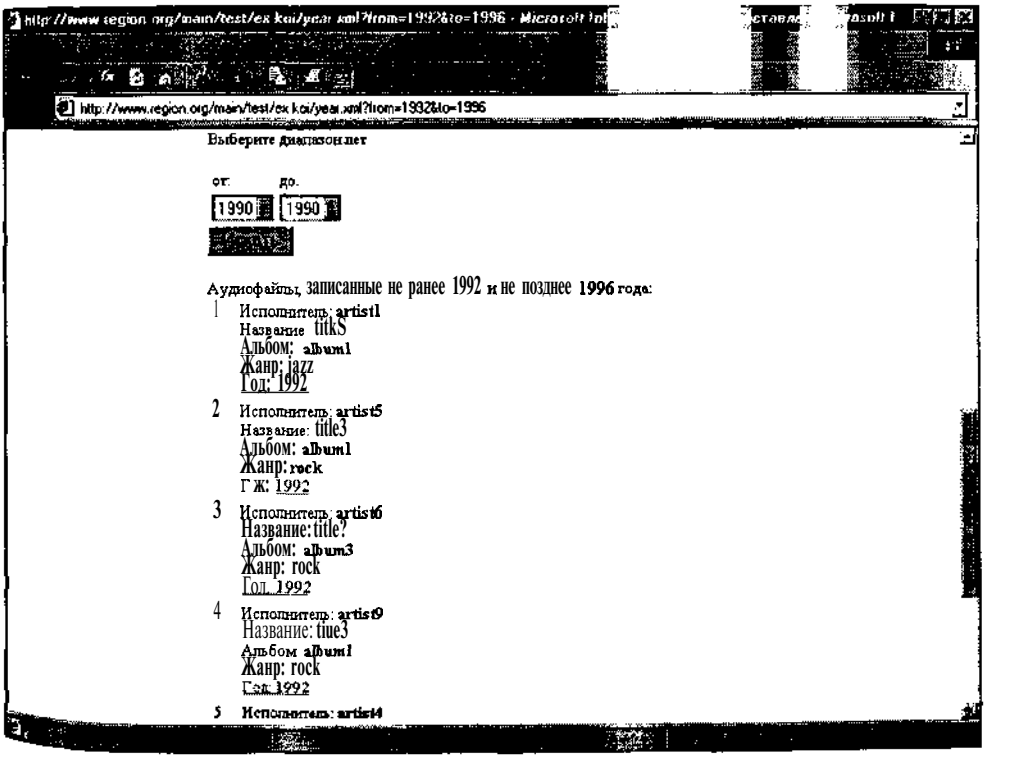


Рис. 4.4. Поиск по году записи

Примечание

Как видно из рис. 4.4, мы просматриваем только начало списка, продолжение которого находится ниже, что не очень удобно. Ведь найденных аудиофайлов может быть больше, если мы дополним файл storage.xml. Кроме того, после перезагрузки в списках, служащих для выбора диапазона лет (select), мы видим не введенные ранее значения 1992 и 1996, а исходные — 1990 и 1990 соответственно. Эти два недостатка мы устраним в двух последующих XML-документах.

Теперь перейдем к рассмотрению правил шаблона, с помощью которых создается Web-страница с вышеописанным содержанием. Как и в предыдущем разделе, мы не будем описывать их все, а только те, которые содержат новые конструкции. Элементами для этих правил шаблона являются `year_list`, `list` И `confirm`.

Правило шаблона для year_list содержит элемент form, представляющий форму для ввода диапазона лет. Здесь примечательным является его атрибут id, которым мы воспользуемся при написании сценария JavaScript. Далее следует директива `xsl:if`, шаблон которой будет участвовать в трансформации документа только тогда, когда соответствующий этому документу XML-файл загружен с параметром from HTTP-запроса. Директива `xsl:apply-templates`, являющаяся частью этого шаблона, указывает, к каким элементам audiofile из файла storage.xml должно быть применено соответствующее правило шаблона. Ее атрибут select как раз и адресует тот набор узлов (элементов `audiofile`), год записи которых лежит в заданном диапазоне.

Путь адресации, являющийся значением select, начинается со ссылки на переменную `afiles`, которая адресует все элементы audiofile из файла storage.xml. Далее следуют два предиката, задающие условия, которым должны соответствовать данные элементы.

Первый предикат обуславливает то, что преобразованный в число контент непосредственного потомка year рассматриваемого элемента audiofile должен быть больше либо равен числовому значению атрибута `from` текущего узла, которым является элемент `year_list`. Иными словами, год записи рассматриваемого аудиофайла должен быть не меньше, чем нижняя граница введенного диапазона лет.

По тому же самому принципу второй предикат определяет, превышает ли год записи рассматриваемого аудиофайла значение верхней границы этого диапазона или нет. Если для какого-либо из элементов хотя бы один из данных предикатов на этапе обработки будет преобразован в булево значение false, то такой элемент трансформироваться не будет, т. е. будет представлен в конечном дереве пустым набором узлов.

Правило шаблона для элемента list представляет интерес только построением таблицы, в которой содержатся элементы select и соответствующие

ним подписи. Конструкция каждого из этих элементов идентична той, которая была описана в предыдущем разделе — соответствующие списки отсортированы и не содержат повторяющихся значений. Отметим, что для каждого из данных элементов select задан атрибут name, имеющий значение from для элемента устанавливающего нижнюю границу года записи, и to — для элемента устанавливающего верхнюю границу. Этот атрибут мы также будем использовать в сценарии JavaScript.

Рассмотрим последовательно все атрибуты элемента input, представленного в шаблоне для элемента confirm. Атрибут type, имеющий значение button, говорит о том, что рассматриваемый элемент является кнопкой. Надпись доя нее задает атрибут value. Значением атрибута onclick является сценарий JavaScript, которому будет передано управление, когда пользователь щелкнет мышью по этой кнопке. В нашем случае этим сценарием является вызов функции check о с аргументом frm.

Данную функцию мы определили в контенте элемента head конечного дерева. Все, что находится в этом контенте (как правило, это сценарии JavaScript, подключаемые таблицы стилей CSS, стили, представленные элементом style и т. п.) будет загружено до того момента, как загрузится контент элемента body. Располагая там наш сценарий, мы получим уверенность в том, что когда пользователь нажмет кнопку **Искать,** этот сценарий будет уже загружен и функция check о будет выполнена.

Аргументом этой функции является идентификатор формы (атрибут id элемента form) frm, который представляет собой объект JavaScript. В теле функции check() он представлен именем ее аргумента obj.

Рассмотрим теперь, что содержит данная функция. Ее телом является проверка условия (if-else), не превышает ли введенное значение нижней границы диапазона лет, значения его верхней границы. Если это не так, то происходит подтверждение ввода формы (вызывается метод submit() объекта frm), в противном случае выдается сообщение об ошибке и подтверждения ввода не происходит.

Для того чтобы объяснить, как выполняется эта проверка, опишем используемые объекты DHTML, а также некоторые их методы (methods) и свойства (properties):

- select — соответствует элементу select языка HTML. Мы обращаемся к нему по его имени (from или to), которое задано его свойством name или одноименным атрибутом элемента select. Оба используемых объекта select являются свойствами объекта frm, представляющего форму. Типом объекта frm является объектный тип form, определенный в DHTML, поэтому мы обращаемся к ним с помощью obj.from и obj.to соответственно.

- item — свойство объекта select. Возвращает заданный своим аргументом элемент списка, представляющий собой объект типа option. Аргумент должен быть числом, которое является номером возвращаемого элемента списка.
- selectedIndex — свойство объекта select. Возвращает число, являющееся номером выбранного элемента списка, представленного данным элементом select.
- innerText — одно из стандартных свойств DHTML. Для элементов, имеющих контент, возвращает его текстовое значение. Если элемент содержит разметку HTML, то вычисляется строковое значение этого элемента, т. е. синтаксический анализатор HTML объединяет строковые значения всех текстовых узлов, являющихся его потомками.

Теперь рассмотрим условие, определяемое оператором if. Заметим, что вместо знака "больше" мы использовали соответствующую ссылку на сущность — >. Левый операнд операции сравнения возвращает строковое значение (innerText) элемента (item) списка (from) с номером, равным номеру позиции выбранного пользователем элемента (selectedIndex), т. е. возвращается текст текущего элемента списка. Точно так же для объекта select, служащего для ввода верхней границы диапазона лет (to), возвращается строковое значение выбранного пользователем элемента списка. Далее, если левый операнд больше правого (то есть значение нижней границы диапазона лет больше значения верхней), то выдается сообщение об ошибке, в противном случае вызывается метод submit() объекта frm.

4.5. Поиск по жанру

В XML-документе, соответствующем этому критерию поиска, мы продемонстрируем один из возможных способов порционного вывода информации. На практике список найденных значений может быть слишком велик для просмотра, поэтому было бы гораздо удобнее выводить не весь список целиком, а по частям. Реальным примером порционного вывода информации может послужить любой поисковый сервер, который выводит последовательно группы ссылок (например, по десять штук) на найденные сайты. В нашем примере мы будем выводить список аудиофайлов, соответствующих заданному значению жанра группами по три файла в каждой.

Перейдем к рассмотрению листингов.

Листинг 4.10, а. Поиск по жанру. XML-составляющая (genre.xml)

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type = "xsp"?>
```

```
<?cocoon-process type = "xslt"?>
<?xml-stylesheet href = "genre.xsl" type = "text/xsl"?>

<xsp:page language = "java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

<main>
 <description>
  <para>
   Эту часть описания вы можете увидеть только,
   если загружена страница, представленная
   результатом преобразования файла genre.xml.
   Здесь вы можете увидеть список всех аудиофайлов,
   соответствующих вводимому ниже жанру. Если количество
   аудиофайлов больше трех, то они будут выведены не все
   сразу, а по три. Просмотреть следующие три
   аудиофайла можно, нажав кнопку след. (если
   аудиофайлов меньше трех или в том случае, когда
   вы просматриваете последние 3, 2 или 1 из
   них, эта кнопка будет отсутствовать). Просмотреть
   предыдущие аудиофайлы можно, соответственно, нажав
   кнопку пред. (если аудиофайлов меньше трех или в
   том случае, когда вы просматриваете первые 3, 2 или 1
   из них, эта кнопка будет отсутствовать).
  </para>
 </description>
<genre_list>
   <xsp:attribute name = "genre">
     <xsp:expr>request.getParameter("genre")</xsp:expr>
   </xsp:attribute>
   <xsp:logic>
      if (request.getParameter("from")==null) {
         <xsp:attribute name = "from">
           <xsp:expr>0</xsp:expr>
         </xsp:attribute> } else {
         <xsp:attribute name = "from">
           <xsp:expr>request.getParameter("from")</xsp:expr>
         </xsp:attribute> }
   </xsp:logic>
```

```
   <list/>
   <confirm/>
 </genre_list>
</main>
</xsp:page>
```

В этом листинге конструкцией, не описанной нами ранее, является элемент xsp: logic, который позволяет вставлять код Java непосредственно в XML-документ. В частности, с помощью условного оператора if-else языка Java мы определяем, был ли задан параметр HTTP-запроса с именем from. Если этот параметр задан не был (то есть если вызов метода request.getParameter ("from") возвращает null), то мы для элемента genre_list создаем атрибут с именем from и значением о. В противном случае мы также создаем этот атрибут, но уже со значением, равным значению этого параметра. Подробнее о встроенном в XML-документ коде Java можно прочитать в *главе 6.*

Далее следует соответствующий сценарий XSLT.

Листинг 4.10, *б.* Поиск по жанру. XSL-составляющая (genre.xsl)

```
<?xml version = "1.0" encoding="koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:include href = "common.xsl"/>
<xsl:include href = "audiofile.xsl"/>
<xsl:variable name = "panels" select = "document('panels.xml')/panels"/>
<xsl:variable name = "afiles"
              select = "document('storage.xml')/audiofiles/audiofile"/>
<xsl:template match = "main">
 <xsl:processing-instruction name = "cocoon-
format">type="text/html"</xsl:processing-instruction>
 <html>
  <body>
    <hl align = "center">Поиск по жанру</h1>
    <p align = "center">
       <a href = "main.xml">Главная</a>
    </p>
    <xsl:apply-templates select = "$panels/controls"/>
    <span style="top:50px;width:60%;;vertical-align:top;margin-right:5%">
      <xsl:apply-templates select = "$panels/description/para"/>
      <xsl:apply-templates/>
```

```
    </span>
    <xsl:apply-templates select = "$panels/resources"/>
    <p align = "center">
     <a href = "main.xml">Главная</a>
    </p>
  </body>
 </html>
</xsl:template>

<xsl:variable name = "start"
              select = "number(normalize-space(//genre_list/@from))"/>
<xsl:template match = "genre_list">
  <a name = "search"/>
  <form id = "frm" action = "genre.xml#search">
    <input name = "from" type = "hidden"/>
    <xsl:apply-templates/>
  </form>
  <script>
   if ('<xsl:value-of select = "$start"/>'=='NaN')
      frm.firstChild.value = 0;
    else frm.firstChild.value=Number(<xsl:value-of select = "$start"/>);
  </script>
  <xsl:if test = "normalize-space(@genre)!='null'">
    <xsl:variable name = "afiles"
         select = "$afiles[genre = normalize-space(current()/@genre)]"/>
    Всего найдено аудиофайлов жанра
    <b><xsl:value-of select = "@genre"/></b>:
    <xsl:value-of select = "count($afiles)"/>
    <xsl:choose>
      <xsl:when test="$start=0">
         <xsl:apply-templates select = "$afiles[position() &lt; 4]"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:apply-templates select =
             "$afiles[($start &lt; position()) and
                      (position() &lt; ($start+4))]"/>
      </xsl:otherwise>
    </xsl:choose>
```

```
    <xsl:if test = "$start &gt; 0">
       <script>
         buttons.firstChild.style.visibility = 'visible';
       </script>
    </xsl:if>
    <xsl:if test = "count($afiles) Sgt; $start +3">
       <script>
         buttons.firstChild.nextSibling.style.visibility='visible;'
       </script>
    </xsl:if>
  </xsl:if>
</xsl:template>

<xsl:template match = "list">
   Выберите жанр<br/><br/>
   <xsl:variable name = "genre" select = "normalize-space(../@genre)"/>
   <select name = "genre" style="width:80%">
     <xsl:for-each select = "$afiles">
       <xsl:sort select = "genre"/>
       <xsl:if test = "not(preceding-sibling::*/genre[text()=
                          current()/genre/text()])">
         <xsl:choose>
          <xsl:when test = "normalize-space(genre)=$genre">
             <option selected = "true">
               <xsl:value-of select = "genre"/>
             </option>
          </xsl:when>
          <xsl:otherwise>
             <option>
               <xsl:value-of select = "genre"/>
             </option>
          </xsl:otherwise>
         </xsl:choose>
       </xsl:if>
     </xsl:for-each>
   </select><br/xbr/>
</xsl:template>

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

```
  <input type = "submit" value = "Вывести"/>
  <span id = "buttons" style="width:50%;text-align:right">
    <input style = "visibility:hidden" type = "submit" value = "пред."
       onclick="frm.firstChild.value=Number(frm.firstChild.value)-3;"/>
    <input style = "visibility:hidden" type = "submit" value = "след."
       onclick="frm.firstChild.value=Number(frm.firstChild.value)+3;"/>
  </span>
</xsl:template>

<xsl:template match = "genre">
  <div style="text-decoration:underline;color:#00AA00;">
   Жанр:
   <b>
      <xsl:apply-templates/>
   </b><br/>
  </div>
</xsl:template>
<xsl:template match = "link">
 <xsl:if test = "not(normalize-space(@to)='genre.xml')">
  <a href = "{@to}">
   <xsl:apply-templates/>
  </axbr/>
 </xsl:if>
</xsl:template>

<xsl:template match = "audiofile">
 <table width="50%">
  <tr>
   <td width="10%" valign = "top">
    <xsl:value-of select = "position()+$start"/>
   </td>
   <td>
    <xsl:apply-templates/>
   </td>
  </tr>
 </table>
</xsl:template>

</xsl:stylesheet>
```

Перед тем, как приступить к рассмотрению листинга 4.10, *б,* опише принцип работы данной Web-страницы. Для наглядности мы приведе рис. 4.5, на котором показана эта Web-страница, отображающая результаты поиска для жанра rock. Всего композиций этого жанра, содержащихся в storage.xml, восемь. На рис. 4.5 показана вторая тройка аудиофайлов, т. е. четвертый, пятый и шестой файл из списка найденных. Всего таких групп будет три, причем последняя будет содержать только два аудиофайла.

Как видно из рис. 4.5, для ввода жанра используется элемент select из языка HTML, являющийся компонентом формы ввода. Для подтверждения ввода служит кнопка **Вывести.** Как видно, форма и ее компоненты ничем не отличаются от своих аналогов из XML-документа, реализующего поиск по названию композиции. Отличия проявляются после того, как пользователь выбрал требуемый жанр и произвел подтверждение ввода.

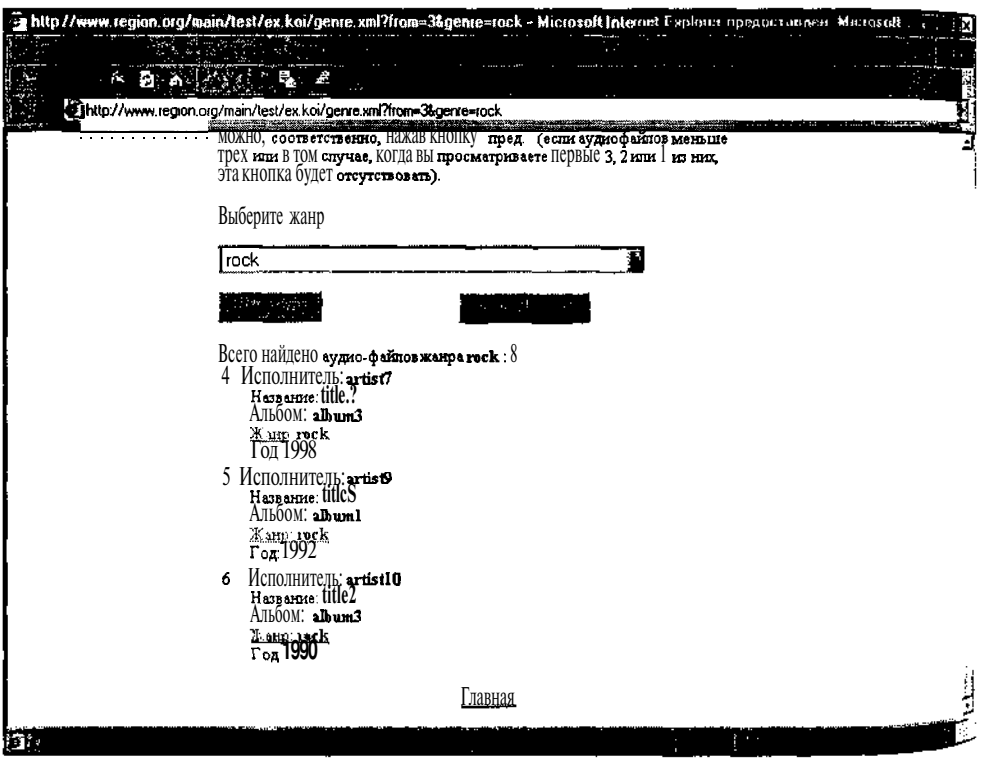


Рис. 4.5. Поиск по жанру

При первой загрузке Web-страницы, т. е. когда поиск еще не производилс эта Web-страница не содержит ничего после формы ввода жанра. После т

го, как выполнится поиск по жанру rock, под кнопкой появится список из первых трех аудиофайлов, соответствующих этому жанру. Кроме того, справа от этого списка появится кнопка **след.**, нажав на которую, пользователь в выведет следующую тройку. Если пользователь просматривает не первую из троек, то справа будет также находиться кнопка **пред.**, позволяющая, соответственно, вывести ту тройку найденных аудиофайлов, которая предшествует данной. Заметим, что если пользователь просматривает первую тройку, то кнопка **пред.** отсутствует, а когда он просматривает последнюю тройку, то отсутствует кнопка **след.**

рассмотрим правила шаблона для элементов `genre_list`, `list` и `confirm`.

Перед правилом шаблона для `genre_list` мы определили переменную start, значением которой является преобразованное в число значение атрибута from элемента `genre_list`. Эта переменная была нами введена для того, чтобы избежать многократного переписывания вызовов функций number() И `normalize-space()`.

В шаблоне для элемента `genre_list` точно так же, как и в листинге 4.9, *б,* с помощью элемента `form` создается форма. Атрибут id этого элемента задает ее идентификатор (frm), который используется в конструкциях JavaScript. Для того чтобы передать документу genre.xml номер, начиная с которого следует выводить найденные значения, в контенте form находится элемент input, значением атрибута type которого является hidden. Это значение обеспечивает отсутствие отображения на Web-странице данного элемента input, но с помощью этого элемента мы можем задать имя передаваемого параметра HTTP-запроса (оно указывается значением атрибута name), а также значение этого параметра, которое указывается значением атрибута value (или одноименного свойства объекта frm).

Сразу после элемента input следует сценарий JavaScript, представленный элементом script языка HTML. В этом сценарии задается начальное значение атрибута value элемента input. Это делается с помощью проверки условия if-then, в которой всегда выполняется только тот код, который соответствует секции else. Мы вставили этот сценарий для того, чтобы продемонстрировать совместное использование конструкций XSLT и кода JavaScript, а также показать взаимодействие с DOM (Document Object Model друг — объектная модель документа), реализованной в браузере MS Internet Explorer.

В Условии проверки используется директива `xsl:value-of`, с помощью которой в конечный документ подставляется строковое значение переменной start, которое сравнивается со строкой `'NaN'`. Поскольку в файле genre.xml мы Уже сформировали корректное значение атрибута from элемента `genre_list` (равное 0, если значение параметра `from` есть null, и значению этого параметра в противном случае), то условие проверки всегда будет

ложным. Исключением является тот случай, когда пользователь в адресной строке браузера сам задаст такое значение параметра from, которое преобразуется в NaN, т. е. не является числом. Иллюстрирующим эту ситуацию может быть, **например, такой URL:** http: //. . /genre.xml?from=str.

Объект frm представляет собой элемент form, который является контейнером для всех элементов формы. Первым непосредственным потомком этого элемента является элемент input. Для того чтобы к нему обратиться, мы использовали свойство firstChild, которым обладают все элементы HTML, которые могут содержать разметку (другие элементы). Иными словами, аналогом этого свойства в языке XML является child: :*[1]. Для того чтобы обратиться к следующему брату текущего элемента, спецификация DOM определяет свойство nextSibling, аналогом которой в XML является путь **адресации** following-sibling::*[1].

Для того чтобы задать значение атрибута value элемента input, мы обращаемся к свойству value, принадлежащему первому непосредственному потомку (firstchild) элемента form (объект frm), которым и является элемент input. Значение этого атрибута вычисляется с помощью конструктора Number объекта Number, определенного в JavaScript. Конструктор Number() создает новое числовое значение, преобразуя в число свой аргумент. Этот аргумент генерируется также с помощью директивы xsl:value-of. Результатом трансформации рассматриваемого сценария в формат HTML будет следующий код:

```
<script>
   if ('0'=='') frm.firstChild.value=0;
      else frm.firstChild.value=Number(0);
   alert('0 '+frm.firstChild.value);
</script>
```

Сразу после сценария следует проверка, представленная директивой xsl:if, с помощью которой проверяется, была ли загружена рассматриваемая Web-страница в первый раз. Если нет, то это означает, что пользователь задал требуемое значение жанра и произвел поиск, нажав кнопку **Вывести.** В этом случае в трансформации примет участие шаблон, являющийся контентом этой директивы. В самом начале этого контента определяется локальная переменная afiles, которая содержит адрес искомого набора узлов, состоящего из элементов audiofile. В рассматриваемом правиле шаблона в качестве ее значения будет использовано то, которое задано в нем, а не в начале сценария XSLT.

В этом шаблоне мы продемонстрировали, как работает директива выбора одного варианта из многих xsl:choose. Мы использовали эту директиву для того, чтобы проверить, какая из троек элементов из списка найденных должна быть загружена в данный момент. Здесь всего два варианта выбора: выводится первая тройка или какая-либо еще. Чтобы определить условие,

поверяющее номер тройки, применяется директива xsl:when, которая может находиться только в контенте директивы xsl:choose.

Атрибутом test этой директивы является выражение XPath, которое принимает значение true в том случае, когда загружена первая тройка элементов, т. е. когда равно нулю значение переменной start. Если это условие не выполнено, то в трансформации принимает участие контент директивы xsl:otherwise. В первом случае выводятся все найденные элементы artist, у которых номер позиции в списке найденных элементов больше 0 и меньше 4. Во втором — номер элемента в списке должен быть больше значения переменной start и меньше, чем значение start+4, таким образом выводится очередная тройка.

После директивы xsl:choose следуют две директивы проверки условия xsl:if, которые обеспечивают появление на Web-странице кнопок **пред. и след.** В том случае, когда выводится не первая тройка элементов (значение start больше о), должна быть видима кнопка **пред.** Мы делаем ее видимой, используя свойства firstChild и nextSibling для объекта buttons, представляющего элемент span, находящийся в правиле шаблона для элемента confirm исходного дерева. Если выводимая тройка не последняя (количество найденных аудиофайлов больше, чем позиция первого элемента в следующей тройке), то на Web-странице должна отображаться кнопка **след.**, которая делается видимой теми же средствами, что и **пред.**

Для того чтобы сделать видимым или невидимым какой-либо элемент HTML, в этом языке определено свойство visibility определенного в HTML объекта style, которое может принимать два значения visible (видимый) и hidden (скрытый).

Для того чтобы в элементе select формы ввода сделать выбранным то значение жанра, по которому производился поиск, мы также воспользовались директивой xsl:choose. Здесь мы проверяем, совпадает ли текст текущего элемента genre (являющегося непосредственным потомком Рассматриваемого в данный момент элемента audiofile) со значением атрибута genre элемента genre_list. Если да, то с помощью атрибута selected элемента option представляемый данным option элемент списка делается выбранным, т. е. после перезагрузки Web-страницы в окне объекта select мы увидим именно это значение жанра, а не то, которое является первым в отсортированном списке всех жанров, представленных в storage.xml.

Правило шаблона для элемента confirm содержит кнопку подтверждения ввода, представленную элементом input, а также контейнер (элемент span) для кнопок **пред. и след.**, которые изначально невидимы, т. к. атрибуты style, принадлежащие соответствующим им элементам input, содержат строку visibility:hidden.

Нумерация найденных элементов, которая производится в шаблоне для элемента audiofile, в том случае, когда выводится не первая тройка, начинается не с единицы, а с номера выводимого audiofile в общем списке найденных элементов. Это достигается тем, что в директиве xsl:value-of рассматриваемого шаблона в качестве значения ее атрибута select мы выбрали не position(), а position()+$start.

В этом разделе мы рассмотрели необычный способ порционного вывода информации. В *следующем* разделе мы покажем, как производить этот вывод так, чтобы он выглядел более традиционно.

4.6. Многокритериальный поиск

Последняя Web-страница создана большей частью путем объединения тех методов, которыми пользовались для поиска по какому-либо определенному критерию. XML-документ, ее представляющий, предназначен для тех читателей, которым все предыдущие коды показались слишком простыми (по мере изучения дальнейших материалов книги они такими и окажутся). На этой Web-странице будет реализован поиск по всем критериям сразу, причем некоторые из них будут сформулированы более сложно, чем в ранее приводившихся листингах.

Полное описание алгоритма работы этого документа было бы слишком громоздким, поэтому мы опишем здесь только те конструкции XSLT, которые не использовались нами ранее в этом примере. В частности, мы продемонстрируем применение ключей, именованных шаблонов с параметрами, а также еще один способ порционного вывода информации.

Далее следуют листинги соответствующего XML-документа.

Листинг4.11, *а*. Многокритериальный поиск. XML-составляющая (custom.xml)

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="custom.xsl" type="text/xsl"?>

<xsp:page language="java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">
<main>
 <description>
  <para>
   Эту часть описания вы можете увидеть только,
```

если загружена страница, представленная результатом преобразования файла custom.xml. Соответствующий данному файлу сценарий XSLT (custom.xsl) можно попытаться проанализировать тем, кому все остальные XML-документы, представленные в данном примере, показались слишком простыми. Данный XML-документ реализует многокритериальный поиск, т.е. поиск идет одновременно по нескольким параметрам.

</para>
<para>
В строковых значениях, представленных полями формы "Исполнитель", "Название" и "Альбом", можно использовать символ-заменитель *. Причем использование * должно подчиняться следующим правилам:

Поле может состоять только лишь из *
Поле может содержать * в качестве первого символа
Поле может содержать * в качестве последнего символа
Поле может содержать * в качестве первого и последнего символа
Другое использование * не допускается (оно не вызовет ошибки, но при этом * будет интерпретироваться как обычный символ)

Возможность использования символа-заменителя реализуется с помощью директивы xsl:key, в выражении которой присутствует по-настоящему большой предикат (точнее, несколько предикатов).
</para>
<para>
Кроме использования символа-заменителя, нами в список жанров введено специальное значение "unknown". Если оно выбрано, то поиск будет производиться по всем жанрам, представленным в файле storage.xml. Диапазон лет вводится точно так же, как и при поиске по году.
</para>
<para>
Еще одной особенностью данного документа является использование именного шаблона с параметром (см. *custom.xsl*). Кроме того, здесь реализован частичный вывод (по три) списка найденных вхождений, но здесь этот вывод сделан на качественно более высоком уровне, чем

```
    в документе, соответствующем поиску по жанру (см. genre.xsl).
  </para>
 </description>

 <form style="height:70%;width:60%;background-color:#8399B1;border-style:outset;border-width:2;padding:5;">
   <xsp:logic>
     if (request.getParameter("artist")==null) {
       <xsp:attribute name="artist">*</xsp:attribute>
       <xsp:attribute name="isFirstTime">yes</xsp:attribute> } else {
       <xsp:attribute name="isFirstTime">no</xsp:attribute>
       <xsp:attribute name="artist">
         <xsp:expr>request.getParameter("artist")</xsp:expr>
       </xsp:attribute> }
     if (request.getParameter("title")==null) {
       <xsp:attribute name="title">*</xsp:attribute> } else {
       <xsp:attribute name="title">
         <xsp:expr>request.getParameter("title")</xsp:expr>
       </xsp:attribute> }
     if (request.getParameter("album")==null) {
       <xsp:attribute name="album">*</xsp:attribute> } else {
       <xsp:attribute name="album">
         <xsp:expr>request.getParameter("album")</xsp:expr>
       </xsp:attribute> }
     if (request.getParameter("genre")==null) {
       <xsp:attribute name="genre">unknown</xsp:attribute> } else {
       <xsp:attribute name="genre">
         <xsp:expr>request.getParameter("genre")</xsp:expr>
       </xsp:attribute> }
     if (request.getParameter("from")==null) {
       <xsp:attribute name="from">
         <xsp:expr>1</xsp:expr>
       </xsp:attribute> } else {
       <xsp:attribute name="from">
         <xsp:expr>request.getParameter("from")</xsp:expr>
       </xsp:attribute> }
     <xsp:attribute name="year_from">
       <xsp:expr>request.getParameter("year_from")</xsp:expr>
     </xsp:attribute>
     <xsp:attribute name="year_to">
       <xsp:expr>request.getParameter("year_to")</xsp:expr>
```

```
    </xsp:attribute>
  </xsp:logic>
  <input name="artist">Исполнитель</input>
  <input name="title">Название композиции</input>
  <input name="album">Альбом</input>
  <genre_list style="width:80%"/>
  <year_lists/>
  <confirm/>
 </form>
</main>
</xsp:page>
```

Как видно из листинга 4.11, *а*, в нем нет никаких новых конструкций. Единственное, на что здесь стоит обратить внимание, — это использование конструкций HTML непосредственно в XML-коде. Этими конструкциями являются элемент `ol` (контейнер нумерованного списка), а также элементы `li` (элементы этого списка). К этим элементам будет применено правило шаблона по умолчанию, согласно которому они будут просто скопированы вконечноедерево.

Далее в контенте элемента form исходного дерева следует определение тех параметров, которые пользователь не ввел, заполняя поля формы. Причем в качестве значения по умолчанию для этих параметров используются некоторые специальные значения, а не нули или пустые строки.

Далее следует листинг соответствующего сценария XSLT.

Листинг 4.11, *б*. Многокритериальный поиск. XSL-составляющая (custom.xsl)

```
<?xml version="1.0" encoding="koi8-r"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:include href="common.xsl"/>
<xsl:include href="audiofile.xsl"/>
<xsl:variable name="panels" select="document('panels.xml')/panels"/>
<xsl:variable name="afiles" select="document('storage.xml')/audiofiles"/>

<xsl:template match="main">
<xsl:processing-instruction name="cocoon-format">type="text/html"</xsl:processing-instruction>
<html>
<head>
  <script>
```

```
    function check(obj) {
    if (obj.year_from.item(obj.year_from.selectedIndex).innerText &gt;
        obj.year_to.item(obj.year_to.selectedIndex).innerText)
        alert('Ошибка: Значение начального года &gt; значения ↳
        конечного!'); else
        obj.submit(); }
  </script>
 </head>
  <body>
    <h1 align="center">Многокритериальный поиск</h1>
    <p align="center">
       <a href="main.xml">Главная</a>
    </p>
    <xsl:apply-templates select="$panels/controls"/>
    <span style="top:50px;width:60%;;vertical-align:top;margin-right:5%">
      <xsl:apply-templates select="$panels/description/para"/>
      <xsl:apply-templates/>
    </span>
    <xsl:apply-templates select="$panels/resources"/>
    <p align="center">
     <a href="main.xml">Главная</a>
    </p>
  </body>
 </html>
</xsl:template>

<xsl:variable name="title" select="normalize-space(//form/@title)"/>
<xsl:variable name="artist" select="normalize-space(//form/@artist)"/>
<xsl:variable name="album" select="normalize-space(//form/@album)"/>
<xsl:variable name="genre" select="normalize-space(//form/@genre)"/>
<xsl:variable name="year_from"
              select="number(normalize-space(//form/@year_from))"/>
<xsl:variable name="year_to"
              select="number(normalize-space(//form/@year_to))"/>
<xsl:variable name="from"
              select="number(normalize-space(//form/@from))"/>
```

```
<xsl:key name="filter" use="genre"
        match="audiofile[($title='*') or
         ((substring($title,1,1)='*') and
         (substring($title,string-length($title),1)='*') and
          contains(title,substring($title,2,string-length($title)-
2))) or
         ((substring($title,1,1)='*') and
          contains(title,substring($title,2))) or
         ((substring($title,string-length($title),1)='*') and
         contains(title,substring($title,1,string-length($title)-1))) or
         ($title=title)]

         [($artist='*') or
         ((substring($artist,1,1)='*') and
         (substring($artist,string-length($artist),1)='*') and
          contains(artist,substring($artist,2,string-
length($artist)-2))) or
         ((substring($artist,1,1)='*') and
         contains(artist,substring($artist,2))) or
         ((substring($artist,string-length($artist),1)='*') and
         contains(artist,substring($artist,1,string-length($artist)-1)))
or
         ($artist=artist)]

         [($album='*') or
         ((substring($album,1,1)='*') and
         (substring($album,string-length($album),1)='*') and
         contains(album,substring($album,2,string-length($album)-2))) or
         ((substring($album,1,1)='*') and
         contains(album,substring($album,2))) or
         ((substring($album,string-length($album),1)='*') and
         contains(album,substring($album,1,string-length($album)-1))) or
         ($album=album)]

         [number(year) &gt;= $year_from]
         [number(year) &lt;= $year_to]"/>
```

```
<xsl:template match="form">
  <a name="search"/>
  <form id="frm" style="{@style}" action="custom.xml#search">
   <table width="100%">
    <tr>
     <td>
      <xsl:apply-templates select="input[1]"/>
     </td>
     <td style="text-align:right;">
      <xsl:apply-templates select="genre_list"/>
     </td>
    </tr>
    <tr>
     <td>
      <xsl:apply-templates select="input[2]"/>
     </td>
     <td style="text-align:right;">
      <xsl:apply-templates select="year_lists"/>
     </td>
    </tr>
    <tr>
     <td>
      <xsl:apply-templates select="input[3]"/>
     </td>
     <td style="text-align:right;">
      <xsl:apply-templates select="confirm"/>
     </td>
    </tr>
   </table>
  </form>
  <script>
   if (<xsl:value-of select="$from"/>==0) frm.firstChild.value=1;
      else frm.firstChild.value=Number(<xsl:value-of select="$from"/>);
  </script>
  <xsl:if test="normalize-space(@isFirstTime)='no'">
    Всего найдено вхождений:
    <xsl:for-each select="$afiles">
```

```
      <xsl:choose>
        <xsl:when test="$genre='unknown'">
          <xsl:call-template name="result">
            <xsl:with-param name="searched" select="audiofile/genre"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <xsl:call-template name="result">
            <xsl:with-param name="searched" select="$genre"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:if>
</xsl:template>
<xsl:template name="result">
  <xsl:param name="searched"/>
  <b><xsl:value-of select="count(key('filter', $searched))"/></b>
  <xsl:if test="count(key('filter', $searched)) &gt; 3">
   <br/>Выберите номер соответствующей тройки:
   <xsl:for-each select="key('filter', $searched)[position() mod 3 = 1]">
     <xsl:choose>
       <xsl:when test="((position()-1)*3+1) = $from">
         <span style="margin-left:10px;">
           <xsl:value-of select="position()"/>
         </span>
       </xsl:when>
       <xsl:otherwise>
         <a style="margin-left:10px;"
           href="custom.xml?from={(position()-1)*3+1}&amp;artist={$artist}&amp;title={$title}&amp;genre={$genre}&amp;album={$album}&amp;year_from={$yearfrom}&amp;yearto={$year_to}&amp;#search">
           <xsl:value-of select="position()"/>
         </a>
       </xsl:otherwise>
     </xsl:choose>
   </xsl:for-eachxbr/>
```

```
  </xsl:if>
  <xsl:apply-templates
       select="key('filter', $searched)
               [($from&lt;= position()) and (($from+3) &gt;
position())]"/>
</xsl:template>

<xsl:template match="input">
 <xsl:apply-templates/><br/>
 <input name="{@name}" type="text"
        value="{normalize-space(../@*[name()=current()/@name])}"/>
</xsl:template>
<xsl:template match="genre_list">
   Выберите жанр<br/>
   <xsl:variable name="genre" select="normalize-space(../@genre)"/>
   <select name="genre" style="{@style}">
     <option>unknown</option>
     <xsl:for-each select="$afiles/*">
       <xsl:sort select="genre"/>
       <xsl:if test="not(preceding-sibling::*/genre[text()=
                         current()/genre/text()])">
         <xsl:choose>
          <xsl:when test="normalize-space(genre)=$genre">
             <option selected="true">
               <xsl:value-of select="genre"/>
             </option>
          </xsl:when>
          <xsl:otherwise>
             <option>
               <xsl:value-of select="genre"/>
             </option>
          </xsl:otherwise>
         </xsl:choose>
       </xsl:if>
     </xsl:for-each>
   </select>
</xsl:template>
```

```
<xsl:template match="year_lists">
  Начальный и конечный годы<br/>
  <xsl:variable name="year_from" select="normalize-space(. • /@year_from)"/>
  <xsl:variable name="year_to" select="normalize-space(../@year_to)"/>
  <table>
    <tr>
      <td>от:</td>
      <td>до:</td>
    </tr>
    <tr>
      <td>
        <select name="year_from">
          <xsl:for-each select="$afiles/*">
            <xsl:sort select="year"/>
            <xsl:if test="not(preceding-sibling::*/year[text()=
                        current()/year/text()])">
              <xsl:choose>
                <xsl:when test="normalize-space(year)=$year_from">
                  <option selected="true">
                    <xsl:value-of select="year"/>
                  </option>
                </xsl:when>
                <xsl:otherwise>
                  <option>
                    <xsl:value-of select="year"/>
                  </option>
                </xsl:otherwise>
              </xsl:choose>
            </xsl:if>
          </xsl:for-each>
        </select>
      </td>
      <td>
        <select name="year_to">
          <xsl:for-each select="$afiles/*">
            <xsl:sort select="year"/>
```

```
            <xsl:if test="not(preceding-sibling::*/year[text()=
                        current()/year/text()])">
              <xsl:choose>
                <xsl:when test="normalize-space(year)=$year_to">
                  <option selected="true">
                    <xsl:value-of select="year"/>
                  </option>
                </xsl:when>
                <xsl:otherwise>
                  <option>
                    <xsl:value-of select="year"/>
                  </option>
                </xsl:otherwise>
              </xsl:choose>
            </xsl:if>
          </xsl:for-each>
        </select>
      </td>
    </tr>
  </table>
</xsl:template>

<xsl:template match="confirm">
  <input type="button" onclick="check(frm) " value="Искать"/>
</xsl:template>

<xsl:template match="link">
 <xsl:if test="not(normalize-space(@to)='custom.xml') ">
  <a href="{@to}">
   <xsl:apply-templates/>
  </a><br/>
 </xsl:if>
</xsl:template>

<xsl:template match="audiofile">
<table width="50%">
  <tr>
```

```
<td width="10%" valign="top">
  <xsl:value-of select="position()+$from -1"/>
</td>
<td>
  <xsl:apply-templates/>
 </td>
</tr>
</table>
</xsl:template>

</xsl:stylesheet>
```

Мы не будем здесь описывать принцип работы данной Web-страницы, т. к. он уже достаточно подробно описан в контенте элемента description листинга 4.11, *а*. Представление о том, как выглядит эта страница, можно получить из рис. 4.6.

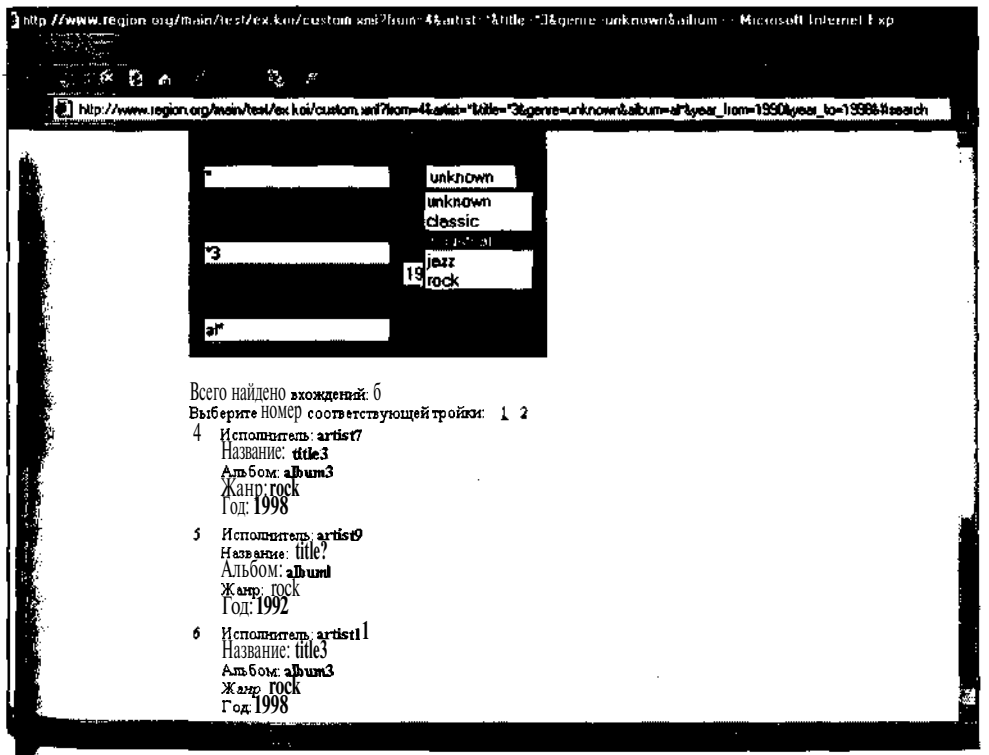


Рис. 4.6. Многокритериальный поиск

Как видно из листинга 4.11, *б*, для получения адреса искомых элементов мы воспользовались директивой `xsl:key`, которая определяет для них ключ с именем `filter`. Ключи в XSLT служат для индексации узлов исходного дерева. Иными словами, мы можем для некоторого множества узлов, обладающих неким общим признаком, который можно описать средствами XPath, определить ключ и далее, используя функцию `key()` вместе с именем этого ключа, адресовать это множество узлов.

Но в XSLT уже есть конструкция, которая выполняет те же самые функции, - директива `xsl:variable`. И действительно, мы можем задать имя переменной точно так же, как и имя ключа (атрибут name, имеющийся как у `xsl:key`, так и у `xsl:variable`), мы можем указать путь адресации, описывающий требуемый набор узлов (атрибут `select` у директивы `xsl:variable` и атрибут match у директивы `xsl:key`). После этого ссылку на переменную можно использовать точно так же, как и функцию `xsl:key`. Примером такой переменной может послужить panels, определенная в листингах 4.2, *б*, 4.5, *б*, 4.7, *б*, 4.9, *б*, 4.10, *б* и 4.11, *б*.

На самом деле сходство директив `xsl:variable` и `xsl:key` только кажущееся. У `xsl:key` имеется еще и атрибут use, т. е. ключ обладает тремя характеристиками: именем, значением и тем набором узлов, к которому он приписан. В то время как переменная — только двумя — именем и значением. Таким образом, можно сказать, что ключ — это переменная, параметризованная одним параметром. Кроме того, в отличие от ключей, переменная может принимать значение любого из четырех, определенных в языке XPath типов, тогда как функция `key()` возвращает только набор узлов.

Теперь рассмотрим тот ключ, который мы определили в листинге 4.11, *б*. Значение атрибута match соответствующей ему директивы хотя и выглядит огромным, на самом деле имеет довольно простую структуру. Как видно, образец, являющийся значением этого атрибута, состоит всего из одного шага адресации, в котором опущено имя оси по умолчанию (child), правило проверки узлов накладывает ограничение на имена отбираемых узлов (именем может быть только `audiofile`), предикаты, составляющие большую часть образца, служат для отбора узлов, удовлетворяющих критериям поиска.

Рассмотрим первый предикат образца. Этот предикат служит для отбора тех аудиофайлов, у которых название композиции соответствует введенным пользователем условиям. Иными словами, производится отбор таких элементов `audiofile`, у которых контент непосредственного потомка `title` частично или полностью совпадает со значением, заданным пользователем. При вводе этого значения пользователь может употребить символ-заменитель, о котором см. листинг 4.11, *а* (/main/description/para[2]), а также некоторое специальное значение для жанра (/main/description/para[3]).

Таким образом, после ввода названия композиции значение соответствующего параметра HTTP-запроса может содержать звездочку (*). Опишем ус-

ловия, накладываемые на название композиции в зависимости от местоположения * в этом значении:

1. Если все значение целиком состоит из *, т. е. не содержит никаких других символов, то это означает, что название композиции искомого аудиофайла может быть произвольным.
2. Если значение начинается с символа *, а далее следует некоторая последовательность символов, то название композиции искомого аудиофайла должно заканчиваться на эту последовательность.
3. Если значение заканчивается на *, которой предшествует некоторая последовательность символов, то название композиции искомого аудиофайла должно начинаться на эту последовательность символов.
4. Если значение и начинается на *, и заканчивается на *, между которыми расположена некоторая последовательность символов, то название композиции искомого аудиофайла должно содержать эту последовательность символов.

Понятно, что пользователь не может задать все четыре условия одновременно, т. е. символ * будет использован только одним из вышеперечисленных способов. Поэтому мы применим логический оператор `or`, операндами которого будут выражения XPath, представляющие эти способы.

Перед тем как построить эти выражения, нам нужно установить очередность проверки, какой именно из способов используется. Сначала мы проверим, не является ли значение параметра целиком состоящим из *. Если мы выполним такую проверку первой и в результате получим `true`, то надобность во всех остальных проверках отпадет и, более того, остальные проверки не должны будут производиться. Напомним, что просмотр операндов `or` производится слева направо и прекращается, как только найден первый операнд, значение которого вычисляется как `true`.

Далее мы должны будем проверить, не содержит ли значение параметра * в начале и в конце его представляющей строки. Действительно, если перед этой проверкой мы проверим условия 2 или 3, то такой поиск будет неверным. Предположим, что пользователь ввел, например, такое значение `*aaa*`. Если мы выполним для этого значения проверку, соответствующую условию 2, то будет произведен поиск аудиофайлов, название композиции которых заканчивается на `aaa*`, т. е. содержит звездочку. В случае проверки, соответствующей условию 3, проверяемое название композиции должно будет начинаться на `*aaa`, что также не соответствует критерию поиска.

Поэтому в рассматриваемом предикате мы расположим операнды `or` согласно следующему порядку соответствующих им условий: 1, 4, 2, 3. Отметим, что условия 2 и 3 равноправны, и не имеет значения, какое из этих двух условий будет проверено первым.

Перед директивой xsl:key мы определили несколько переменных, содержащих значения тех узлов дерева, которые мы не сможем адресовать, когда сменится текущий документ. Ведь поиск мы будем производить среди элементов, находящихся в файле storage.xml, который и станет текущим во время этого поиска. Точно так же, как и в предыдущих примерах, смена текущего документа произойдет в контенте директивы xsl:for-each, причем текущими станут поочередно все узлы из набора, адресуемого переменной afiles, определенной в начале документа. Напомним, что этот набор узлов содержит всего один узел — audiofiles — корневой элемент документа storage.xml.

Итак, первым операндом or рассматриваемого предиката будет $title='*', который принимает значение true только тогда, когда введенное пользователем значение названия композиции состоит из одного символа — звездочки.

Второй операнд будет преобразован в true при одновременном (and) выполнении трех условий:

1. Введенное пользователем название композиции имеет первым символом *: substring($title,1,1)='*'.
2. Введенное пользователем название композиции имеет последним символом *: substring($title,string-length($title),1)='*'.
3. Искомое название композиции содержит последовательность символов, располагающуюся между звездочками во введенном пользователем названии композиции:

```
contains(title, substring($title,2,string-length($title)-2)).
```

Третий операнд, соответствующий условию 2, будет преобразован в true, если введенное пользователем название композиции имеет первым символом *, (substring($title,1,1)='*'), после которого следует некоторая последовательность символов, и искомое название композиции будет заканчиваться символами, образующими эту последовательность (contains(title, substring($title,2))).

Четвертый операнд, соответствующий условию 3, будет преобразован в true, если введенное пользователем название композиции имеет последним символом *, (substring($title, string-length($title),1)='*'), перед которым располагается некоторая последовательность символов, и искомое название композиции будет начинаться этой последовательностью символов (contains(title,substring($title,1,string-length($title)-1))).

Пятый операнд будет преобразовываться в последнюю очередь, если ни один из первых четырех не будет истинным, т. е. если введенное пользователем значение вовсе не содержит звездочку или содержит, но где-то в середине строки, представляющей это значение. В этом случае введенное пользователем и проверяемое значения должны полностью совпадать

($title=title), иначе текущий audiofiie не попадет в список найденных аудиофайлов.

Таким образом, первый предикат для элемента audiofiie отбирает только те элементы, которые соответствуют заданному шаблону названия композиции. По тому же самому принципу работают второй и третий предикаты, которые отбирают элементы в соответствии с введенными значениями альбома (album) и исполнителя (artist). Последние два предиката накладывают ограничения на год записи искомых аудиофайлов. Механизм их работы описан в разделе, соответствующем поиску по году записи.

Перед тем как рассмотреть применение вышеописанного ключа, мы расскажем о том, для чего нами было введено специальное значение (unknown) для жанра искомых аудиофайлов. Использование этого значения тесно связано с именованным шаблоном, параметром которого оно является. Директива xsl:template, определяющая этот шаблон вместо атрибута match, содержит атрибут name, благодаря чему ее легко найти в листинге 4.11, *б*. Именем этого шаблона является result.

Специальное значение жанра unknown пользователю может понадобиться, когда ему не известен жанр искомых аудиофайлов. Это значение, как видно из рис. 4.6, располагается первым в списке всех жанров, представленных в файле storage.xml. Данный список представлен элементом select языка HTML.

Таким образом, при реализации поиска в сценарии XSLT должен быть осуществлен выбор одной из двух альтернатив: либо пользователь указал жанр конкретно, либо он выбрал специальное значение unknown. Этот выбор производится с помощью директивы xsl:choose в шаблоне трансформации для элемента form. И как мы уже говорили ранее, в зависимости от этого выбора задается параметр searched для именного шаблона result.

Значением для ранее определенного ключа мы выбрали контент элемента genre, являющегося непосредственным потомком индексируемых элементов audiofiie. Это означает, что данный ключ будет описывать набор узлов, соответствующих образцу match и имеющих в качестве контента своего непосредственного потомка genre то значение, которое задано вторым аргументом функции key(), возвращающей описываемый этим ключом набор узлов. Первым аргументом этой функции будет имя ключа, в нашем случае - filter.

Таким образом, если пользователь вводит конкретный жанр, например industrial, то аргументами функции key О, описывающей соответствующие аудиофайлы при прочих условиях, будут filter (имя ключа) и industrial (требуемое значение жанра). В нашем случае жанр задан переменной genre, поэтому мы получаем требуемый набор узлов с помощью key('filter',

$genre). Если же пользователь выбрал специальное значение, искомые узлы адресуются с помощью key('filter', audiofile/genre), т. е. при отборе нужных элементов audiofile контент элемента genre сравнивается сам с собой. Такое сравнение всегда будет истинным, и тем самым поиск будет производиться по аудиофайлам произвольного жанра.

Заметим, что приведенные в предыдущем абзаце вызовы функции key() не присутствуют непосредственно в листинге 4.11, *б*. Вместо этого их вызов происходит в именованном шаблоне result. Это сделано потому, что вне зависимости от того, выбрал ли пользователь конкретный жанр или специальное значение, вывод найденных элементов audiofile производится одинаково. Если бы мы не определили шаблон filter, то нам пришлось бы переписывать его контент для каждой альтернативы xsl:choose в шаблоне для form. Таким образом, один и тот же код в сценарии XSLT был бы продублирован.

Вместо этого мы, в зависимости от выбранного значения жанра, сформировали второй аргумент функции key() ($genre ИЛИ audiofile/genre) И передали его в качестве параметра searched именованному шаблону, в котором и производится вызов этой функции. При этом вместо двукратного переписывания одного и того же кода в контенте xsl:choose мы использовали вызов данного именованного шаблона. Этот вызов производится с помощью **Директивы** xsl:call-template.

В контенте именованного шаблона result формируется порционный вывод информации (по три найденных аудиофайла), о котором мы обещали рассказать в начале раздела. Как видно из рис. 4.6, здесь для этого вывода мы уже не используем кнопки **пред. и след.**, как это делалось в разделе, соответствующем поиску по жанру. Вместо этого мы выводим цифры 1, 2, 3, соответствующие номерам троек найденных аудиофайлов. При этом каждая из этих цифр, кроме той, которая соответствует просматриваемой в данный момент тройке, представляет собой гиперссылку, при активизировании которой Web-страница перезагружается и выводится тройка аудиофайлов с номером, равным выбранной цифре.

В начале этого шаблона с помощью директивы xsl:value-of выводится общее количество найденных аудиофайлов, соответствующих критериям поиска. Далее следует проверка условия xsl:if, с помощью которой мы выясняем, будет ли общее количество найденных аудиофайлов больше, чем Три (count(key('filter', $searched)) > 3). ЕСЛИ нет, ТО нет необходимости производить порционный вывод информации.

Если же условие, заданное проверкой, истинно, то с помощью директивы xsl:for-each строится ряд цифр, соответствующий номерам троек. Причем цикл, реализуемый данной директивой, перебирает только те найденные

аудиофайлы, номер позиции которых кратен трем (`key('filter', $searched)[position() mod 3 = 1]`). В контенте *xsl:*`for-each` мы с помощью директивы `xsl:choose` выясняем, соответствует ли позиция текущего элемента audiofile той позиции, с которой начинается вывод данной тройки (`((position()-1)*3+1) = $from`). Иными словами, мы проверяем, соответствует ли выводимая на данном шаге цикла цифра номеру текущей тройки. Если да, то эта цифра выводится как текстовый контент элемента span конечного дерева, в противном случае она становится текстовым контентом элемента a, т. е. выводится как гиперссылка.

В последнем случае для элемента, представляющего эту гиперссылку, мы формируем URL, являющийся значением атрибута href. Здесь мы указываем те параметры HTTP-запроса, при которых будет произведен поиск по тем же самым критериям, которые были введены пользователем во время последней загрузки Web-страницы. Здесь мы также задаем параметр from, указывающий, с какой позиции следует выводить список найденных элементов. Эта позиция вычисляется (`(position()-1)*3+1`) в зависимости от того, какая цифра представляет собой формируемую гиперссылку.

После `xsl:if` следует директива `xsl:apply-templates`, которая и отвечает за вывод списка найденных аудиофайлов. Она указывает процессору, к каким элементам audiofile следует применить соответствующее правило шаблона. Эти элементы, во-первых, должны удовлетворять критериям поиска, что обеспечивается функцией `key()`, а во-вторых, номер позиции выводимого элемента в общем списке найденных audiofile должен быть не меньше, чем `$from`, и меньше, чем `$from+3`. Второе условие и обеспечивает порционность вывода информации, которая заключается в том, что выводятся не все найденные элементы, а только те из них, номер которых лежит в заданном диапазоне.

Подробное рассмотрение конструкций, используемых в листинге 4.11, *б,* мы оставляем на самостоятельное изучение читателю. И если применение каких-либо из них, например, расширений Cocoon или инструкций CSS, вызывает вопросы, то к нему можно вернуться после изучения следующей части книги, в которой описаны некоторые аспекты создания Web-приложения, рассмотренные как со стороны клиента, так и со стороны сервера.

ГЛАВА 5

Необходимые дополнения

В этой главе, подводя итоги, мы рассмотрим те части спецификаций XML, XSLT и XPath, для описания которых используются все эти спецификации одновременно. Кроме того, здесь мы перечислим те их части, которые по тем или иным причинам не вошли в книгу. После этого можно будет сказать, что описание языка XML, представленное первой частью книги, является завершенным. Предназначение этой главы заключается не в соблюдении формализма изложения. Здесь мы продемонстрируем некоторые неочевидные возможности языка, в частности, мы покажем, например, как использовать пролог документа для манипулирования стилевыми таблицами (для простоты — стилями).

5.1. Упрощенный синтаксис для элемента стиля

Ранее мы говорили, что в качестве элемента стиля может выступать одна из двух директив `xsl:stylesheet` или `xsl:transform`, которые представлены в некотором внешнем файле, являющемся сценарием XSLT. О том, как создать XML-документ, содержащий в одном файле и XML-код, и правила шаблона для его трансформации, мы расскажем несколько позднее, а сейчас рассмотрим один частный случай.

Предположим, что для некоего XML-документа нам необходимо создать такой сценарий XSLT, который содержит только одно правило шаблона — для корня этого документа. При этом остальные узлы исходного дерева либо отсутствуют (присутствует только корневой элемент согласно правилам синтаксиса XML), либо используются только для хранения информации! т. е. не трансформируются, а значит, не представлены в конечном дереве.

Листинг 5.1. Файл doc.xml

```
<?xml version="1.0" encoding="koi8-r"?>
<?xml-stylesheet href="doc.xsl" type="text/xsl"?>
<?cocoon-process type="xslt"?>

<d/>
```

Листинг 5.2. Файл doc.xsl

```
<?xml version="1.0" encoding="koi8-r"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
 <html>
  <head>
    <title> Название документа </title>
  </head>
  <body>
    <p> Некоторый текст </p>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Для таких документов используется так называемый упрощенный синтаксис элементов стиля. Вместо объявления XML (первая строка листинга 5.1) и элементов `xsl:stylesheet` и `xsl:template` с соответствующими атрибутами используется элемент html, имеющий два обязательных атрибута: `xsl:version`, значением которого является `1.0`, а также `xmlns:xsl`, который декларирует пространство имен XSLT.

Далее следует пример сокращенного синтаксиса.

Листинг 5.3. Файл doc.xsl, упрощенный синтаксис

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <head>
    <title> Название документа </title>
  </head>
```

```
  <body>
    <p> Некоторый текст </p>
  </body>
</html>
```

Преобразование, выполненное с помощью сценария doc.xsl, соответствующего листингу 5.3, полностью идентично представленному в листинге 5.2.

Заметим, что рассматриваемый XML-файл содержит только один узел (не считая корня) — элемент документа d. Если бы этот элемент не был пустым, т. е. обладал бы каким-нибудь контентом или атрибутами, то к его содержимому можно было бы обратиться из контента элемента html с помощью различных директив XSLT. Например, если бы d имел своим непосредственным потомком элемент para, то мы могли бы записать, скажем, **В контенте Элемента** body **КОНСТРУКЦИЮ** `<xsl:value-of select="para"/>`. Вообще говоря, в контенте html допустимыми являются все директивы XSLT, за исключением элементов верхнего уровня.

5.2. Встроенные стили

До этого момента мы рассматривали сценарии XSLT, которые находились в некотором внешнем источнике (одном или нескольких файлах). Но существует возможность использовать стили, которые располагаются вместе с соответствующим им XML-кодом. Для этого используется атрибут id, принадлежащий Директивам `xsl:stylesheet` И `xsl:transform`. ТИПОМ ЭТОГО атрибута является ID, что говорит о том, что он будет использован для уникальной идентификации фрагмента документа.

Далее следует пример, иллюстрирующий применение такого стиля, который называется встроенным.

Листинг 5.4. Пример встроенного стиля

```
<?xml version="1.0" encoding="koi8-r"?>
<?xml-stylesheet type="text/xsl" href="#style"?>

<doc>
<head>
<xsl:stylesheet id="style"
                version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="doc">
```

```
  <html>
    <xsl:apply-templates select="body"/>
  </html>
</xsl:template>

<xsl:template match="body">
  <b><xsl:apply-templates/></b>
</xsl:template>

</xsl:stylesheet>
</head>

<body>
Некоторый текст
</body>
</doc>
```

Во второй строке листинга *5А* находится инструкция обработки, указывающая процессору, что для трансформации исходного дерева он должен воспользоваться фрагментом этого дерева с идентификатором style. Этот фрагмент является поддеревом с корневым элементом `xsl:stylesheet`, который обладает атрибутом id, со значением идентификатора style.

Заметим, что документ может содержать несколько стилей, каждый из которых должен быть представлен директивой `xsl:stylesheet` или `xsl:transform`, обладающей уникальным идентификатором.

5.3. Нерассмотренные механизмы языка XML

5.3.1. DTD

Как мы уже говорили ранее, DTD (Document Type Definition — Декларация Типа Документа) применяется для описания структуры документы. В частности, используя внутренний или внешний набор документа, можно задать очередность появления элементов в документе и содержимое элемента (текст, разметка), указать, элементы с какими именами могут находиться в этом содержимом, объявить имена и типы атрибутов, принадлежность атрибутов и их списков какому-либо элементу, а также определить сущности различного типа.

Фактически DTD применяется для создания языков разметки, содержащих конечный список элементов и атрибутов с фиксированными именами, т, е. таких, например, как HTML. В нашу задачу не входит построение такого языка, кроме того, существует еще несколько причин, по которым мы не стали приводить описание синтаксиса DTD.

Во-первых, в настоящее время как альтернатива DTD гораздо более широкое распространение получил стандарт XML Schema. Во-вторых, спецификация XML позволяет процессорам не обрабатывать внешние наборы DTD, в то время как внутренние наборы обрабатываются лишь частично. Таким образом, используя DTD, мы рискуем сильно сузить круг процессоров, способных обрабатывать наш код. В-третьих, правила синтаксиса DTD слишком объемны, т. е. мы не считаем нужным приводить громоздкое описание некоего устаревшего механизма, который процессоры не обязаны поддерживать.

В связи с этим мы исключили из рассмотрения функцию `id()` языка XPath, которая возвращает узел документа, уникально идентифицируемый аргументом функции. Но вместе с тем, мы привели подробное описание сущностей *(см. разд. 1.11),* которые должны быть обработаны любым процессором, если находятся во внутреннем наборе DTD. Причем полезность внутренних сущностей как механизма обозначения и подстановки некоторого фрагмента XML в отличие от остальных возможностей, предлагаемых DTD, является очевидной.

5.3.2. Несколько слов о пространствах имен

В этой книге встречаются несколько префиксов пространств имен `xml:`, `xsl:`, `fo:`, `xsp:`, `fp:`, хотя в некоторых случаях могут использоваться элементы из гораздо большего количества пространств имен. Существует множество всевозможных X-технологий (XLink, XPointer, XInclude...), а также различных устройств, служащих для вывода конечного документа. Каждая из этих технологий или средств взаимодействия с этими устройствами предоставляет инструментарий, состоящий из элементов и атрибутов, принадлежащих отдельным пространствам имен.

При трансформации этих узлов или при задании объявлений некоторых пространств имен могут возникать конфликты. Для разрешения подобных конфликтов, а также тех из них, которые связаны с совместимостью текущей и последующих версий соответствующих спецификаций, язык XML предлагает некоторые средства, которые мы рассмотрим далее. Причем мы сделаем это не слишком подробно, т. к. решение нашей задачи не является совершенным с точки зрения его практического применения. Кроме того, предлагаемый языком механизм разрешения конфликтов совместимости является, откровенно говоря, весьма бедным.

В тех случаях, когда возникает неоднозначность интерпретации кода, связанная с пространствами имен, можно использовать директиву

```
<xsl:namespace-alias
  stylesheet-prefix = префикс
  result-prefix = префикс />
```

которая служит для создания синонима заданного префикса пространства имен. При этом ее первый атрибут должен содержать этот префикс, а второй — имя соответствующего синонима. Например,

```
<xsl:namespace-alias
  stylesheet-prefix = "xsl"
  result-prefix = "my_xsl"/>
```

позволит везде в сценарии XSLT пользоваться директивами XSLT, как с **префиксом** `xsl:`, **так и с** `my_xsl:` **(например,** `<my_xsl:for-each select="...">`). Для того чтобы применение синонима в сценарии стало возможным, необходимо объявить соответствующее ему пространство имен (с помощью атрибута `xmlns` соответствующего элемента). Значением каждого из этих атрибутов может быть ключевое слово `#default`, обозначающее пространство имен по умолчанию.

5.3.3. Расширения и откат

Спецификация XSLT версии 1.0 не описывает механизм, с помощью которого можно расширить круг возможностей этого языка, но вместе с тем, она предоставляет средства (также весьма ограниченные), позволяющие объявить используемые конструкции расширениями. Здесь под расширениями понимаются дополнения, вносимые последующими версиями XSLT (номер версии больше, чем 1.0), с точки зрения текущей версии (номер которой равен 1.0). Иными словами, если процессор XSLT, имеющий версию 1.0, встречает конструкцию, описанную в более поздней спецификации, то при некоторых незначительных прочих условиях он должен ее рассматривать как расширение и обрабатывать ее соответствующим образом.

Спецификация XSLT не предоставляет никаких средств для создания расширений (что, как можно было убедиться, для нее характерно), но она определяет те действия, которые должны быть предприняты по отношению к известному ей расширению, а также те действия, которые она должна предпринять в том случае, когда рассматриваемое расширение недоступно.

Спецификация XSLT подразумевает два типа расширений: элементы-расширения и функции-расширения. Первый тип расширений можно обозначить как расширение с помощью атрибута extension-attribute-prefixes, значением которого будут префиксы тех пространств имен, разде-

ленные пробельным символом, которые являются пространством имен расширений. Этим атрибутом обладает директива `xsl:stylesheet`, `xsl:transform`, а также элемент html сокращенного синтаксиса для стилей (в последнем случае этот атрибут должен иметь префикс `xsl:`). Если процессору не известно, как именно должен быть трансформирован данный элемент, то для него должен быть вызван т. н. откат. Если необходимо объявить элемент, принадлежащий пространству имен по умолчанию, как элемент-расширение, то можно в значении атрибута использовать ключевое слово `#default` вместо указания префикса этого пространства имен.

Для определения доступности элемента спецификация XSLT определяет функцию XPath element-available(*string*), которая возвращает false ТОЛЬКО тогда, когда пространство имен, которому принадлежит имя элемента, заданное аргументом функции, имеет нулевой URI. С другой стороны, она будет возвращать true только в тех случаях, когда имя узла принадлежит пространству имен XSLT либо представляет собой элемент-расширение.

Точно так же для определения доступности какой-либо функции XPath, разработанной как дополнение к текущей документации, служит функция `function-available(`*string*`)`, которая возвращает false только тогда, когда имя функции не содержится в списке основных или дополнительных функций, а также не является функцией-расширением.

Если при обработке документа процессор не способен обработать какой-либо элемент или функцию как элемент-расширение или функцию-расширение, то он должен выполнить так называемый откат. При этом он не должен генерировать ошибку из-за того, что шаблон содержит элемент расширения, реализация которого ему недоступна. Откат осуществляется с помощью директивы

```
<xsl:fallback>
   <!-- Шаблон -->
</xsl:fallback>
```

Если при обработке какого-либо элемента, принадлежащего пространству имен XSLT или являющегося элементом-расширением, его реализация окажется недоступной процессору, то для этого элемента может быть произведен откат. Для этого рассматриваемый элемент должен иметь в качестве непосредственных потомков одну или несколько директив `xsl:fallback`, содержимое которых будет последовательно обработано. Если таких потомков нет, должна фиксироваться ошибка.

При формировании конечного дерева процессор относит каждый из рассматриваемых элементов к одному из четырех типов.

- Директивы. Это те элементы, которые принадлежат пространству имен XSLT.

- Элементы-расширения.
- Элементы, помеченные как исключенные.
- Конечные элементы. В нашем случае это элементы языка HTML, которые копируются в конечное дерево без обработки. Обрабатываются только их атрибуты и контент, который может быть шаблоном.

Мы рассмотрели все эти типы за исключением элементов, помеченных как исключенные. Такое разделение по типам служит для того, чтобы процессор мог определить, нужно ли их обрабатывать, и если нужно, то каким образом. Основные принципы поведения процессора в зависимости от типа этого элемента мы уже обсудили. Рассмотрим, как для созданного элемента формируются узлы пространств имен.

Если созданный элемент является директивой или элементом-расширением, то он не будет обладать соответствующими узлами пространств имен, т. е. узлом пространства имен XSLT или пространством имен, которому принадлежит элемент-расширение. В случае конечного элемента при его копировании в конечное дерево будут также скопированы и соответствующие ему узлы пространств имен.

Если нам не нужно, чтобы некоторые узлы пространств имен копировались в конечное дерево, мы можем пометить их как исключенные. Для этих целей служит атрибут `exclude-result-prefixes`, принадлежащий директивам `xsl:stylesheet` и `xsl:transform`. В случае упрощенного синтаксиса для этих же целей предназначен атрибут `xsl:exclude-result-prefixes`. Значением трех вышеперечисленных атрибутов должен быть список префиксов тех пространств имен, узлы которых мы хотим исключить. Префиксы должны быть разделены пробельным символом. Если с каким-либо из префиксов, перечисленных в этом списке, не связано никакое пространство имен (то есть оно не было объявлено), будет фиксироваться ошибка. Для исключения узла пространства имен по умолчанию вместо указания его префикса можно воспользоваться ключевым словом `#default`.

Часть II

Практическая разработка Web-приложений

Глава 6. Средства создания Web-приложений

Глава 7. Каскадные стилевые таблицы

Глава 8. Методика обработки данных

Глава 9. Представление данных

ГЛАВА 6

Средства создания Web-приложения

В первой части книги мы подробно описали синтаксис языка XML, а также показали, как с его помощью можно создать простейший Web-сайт. Иными словами, мы рассматривали XML как средство для описания документов. Но если рассмотреть этот язык с технической точки зрения, т. е. его программную реализацию, а также способы интеграции с другими приложениями, то можно уже говорить о нем как о средстве для построения Web-приложения.

Напомним, что мы определили условную границу между Web-сайтом и Web-приложением следующим образом: основной задачей Web-сайта является представление информации, хранящейся на сервере, в то время как главной задачей Web-приложения — обработка этой информации. Таким образом, пример, приведенный нами в *главе 4,* можно назвать, с одной СТОРОНЫ, примитивным Web-сайтом, т. к. взаимодействующий с ним пользователь вынужден работать с весьма бедным, как с точки зрения дизайна, так и с точки зрения функциональности, интерфейсом. С другой стороны, этот пример можно назвать и примитивным Web-приложением, т. к. кое-какие Действия на стороне сервера все же выполняются (например, по различным критериям делаются выборки информации), но вместе с тем, у пользователя отсутствуют такие возможности, как добавление, удаление и редактирование описаний аудиофайлов.

Проводя эту границу, можно сказать, что Web-сайт отличается от Web-приложения количеством программного кода, расположенного на сервере. Ведь в функции Web-приложения помимо обработки документов входят такие задачи, как обеспечение безопасности сервера, его взаимодействие с другими серверами и службами, отвечающими за хранение и обработку информации в формате, отличном от XML, и т. п.

Далее в книге мы не будем заниматься вопросами, касающимися администрирования данных серверов и служб. Вместо этого мы покажем, как, используя язык XML вместе с некоторыми другими инструментами, разработать относительно небольшое Web-приложение. Для его практического создания не потребуется привлекать много людей. Коллектив, состоящий не более чем из пяти человек, вполне способен справиться с подобной задачей. Причем каждый из этих разработчиков вовсе не обязан быть специалистом в какой-то узкой области, т. к. при работе с любой технологией очень редко используются все, предлагаемые ею возможности.

Поэтому, чтобы чрезмерно не усложнять задачу, далее мы опишем только основы тех технологий, которые будем применять в наших разработках. В частности, мы приведем описание некоторых конструкций Java, языка описания стилей CSS, а также рассмотрим возможности, предлагаемые DHTML.

Как мы уже говорили ранее, основным достоинством языка XML является его способность задавать и обрабатывать структуру документа. Исходя из этого, мы и определим те задачи, которые наше Web-приложение будет выполнять. И хотя даже для небольших Web-приложений часто требуется задействовать какие-либо дополнительные механизмы хранения и обработки информации, например базы данных, мы покажем, как можно обойтись и без них.

Небольшим Web-приложением мы будем считать такое, для которого нет необходимости обеспечивать высокий уровень безопасности и устойчивости (возможность обрабатывать одновременно несколько сотен, тысяч и более запросов) и не нужен большой коллектив высокопрофессиональных разработчиков. В этой главе мы попытаемся подробно проиллюстрировать все вышесказанное, а также на концептуальном уровне разберем тот класс задач, который вписывается в рассмотренный масштаб Web-приложения.

6.1. Документы и технология клиент-сервер

Ни для кого не секрет, что перенос коммерческой или общественной деятельности на электронную платформу позволяет снизить финансовые затраты на ее организацию и сопровождение. В самом деле, вместо того чтобы регулярно платить за аренду офисных и торговых помещений, гораздо проще создать Web-площадку, которая бы совмещала функции и тех, и других. Также не нужно тратиться на продавцов и другой обслуживающий персонал — интерактивность Web-сайта позволит решить и эту проблему. Кроме того, этот сайт при наличии красивого оформления может послужить и в рекламных целях.

Все эти, а также многие другие причины говорят о том, что сфера электронного бизнеса будет неуклонно расширяться. Но область применимости

Web-технологий по вполне понятным причинам не ограничивается реализацией товаров и услуг через Internet. С их помощью можно также организовать и внутрикорпоративную деятельность предприятия. Если это предприятие ведет достаточно активную деятельность, то как между его сотрудниками, так и при его взаимодействии с другими предприятиями и государством рождается великое множество документов.

Накапливаясь, эти документы сначала исчисляются папками, потом шкафами и, в конце концов, помещениями. Причем такое количество информации обуславливается вовсе не жаждой накопительства руководства или служащих предприятия. На практике достаточно часто возникает необходимость поднять отчетность за один или несколько прошлых лет. Например, если какой-нибудь сотрудник предприятия выходит на пенсию, то, чтобы вычислить ее размер, потребуется информация о его прошлых местах работы, стаже, зарплате в различные периоды времени и т. д.

Для решения подобных задач как нельзя лучше подходит Web-сервис. В качестве его основных достоинств можно назвать следующие:

- целостность — все данные могут храниться в одном месте — едином информационном хранилище;
- универсальный доступ — для тех пользователей Web-приложения, которые обладают одними и теми же правами, т. е. наделены одними и теми же функциями, доступ к ресурсам приложения будет одинаково быстр, удобен и функционален с любой рабочей станции, способной соединяться с Web-приложением;
- снижение затрат на организацию и поддержку деятельности предприятия;
- оперативность документооборота.

Помимо коммерческой ценности, Web-технологии, благодаря своей гибкости, простоте настройки и высокой функциональности, пользуются большой популярностью у разработчиков клиент-серверных приложений. В самом начале книги мы уже говорили о различных сетевых службах, администрируемых посредством Web-интерфейса. Среди них почтовые серверы, средства удаленного администрирования операционной системы и множество других.

Подводя итоги вышесказанному, можно сказать, что спектр возможностей Web-ориентированных приложений весьма высок. Но описывая его, мы не ставили перед собой цели созвать как можно больше пользователей и разработчиков информационных систем под транспаранты с надписями "Web", "Internet" и "Intranet". Мы хотели лишь обратить внимание читателя на один любопытный факт: в основе всех преимуществ, предлагаемых Web-сервисами, лежит понятие "документ". А также на вопрос, который вытекает из этого факта: как соотносится это понятие с привычными для нас листами бумаги, содержащими какой-то текст?

Здесь мы поступим точно так же, как и ранее. Добавим к слову "документ" приставку "Web-" и попытаемся раскрыть суть получившегося термина. Для этого нам понадобится взглянуть на данный термин с двух точек зрения: пользователя, обрабатывающего Web-документ, и разработчика, обеспечивающего всю его функциональность.

Рассмотрим весь процесс формирования и обработки бумажного документа от начала и до конца. Вообще говоря, этот процесс может быть как простым, так и очень сложным. Например, такой документ, как расписка о получении денег, содержит очень небольшое количество информации и создается очень быстро — на листке бумаги от руки записываются данные участвующих сторон и сумма полученных денег. С другой стороны, годовой отчет о деятельности какой-нибудь крупной фирмы может состоять из нескольких сотен или тысяч страниц, которые создаются трудом десятков людей. Далее мы не будем рассматривать такие сложные документы, а также какие-либо аналоги документов, взятых из реальной жизни. Вместо этого мы просто постараемся перечислить и описать все стадии, которые проходит документ от своего рождения до сдачи его в архив или уничтожения.

Итак, представим себе служащего некоего предприятия, перед которым лежит лист бумаги и ручка. После нескольких мучительных минут или часов размышлений он примется за работу и начнет составлять документ. Казалось бы, естественно начать описание документа именно с этого момента. Но здесь надо вспомнить и о существовании разработчика. Поясним, что мы здесь имеем в виду.

Предположим, что наш служащий начал свою нелегкую и очень нужную деятельность не с чистого листа бумаги, а с некоего бланка или формы, поля которой ему предстоит заполнить. Причем совершенно очевидно, что эту форму разработал не он, а кто-нибудь рангом повыше или, например, какой-нибудь государственный орган. Мы для простоты предположим, что эта форма зародилась где-то в недрах того же предприятия, на котором трудится тот человек, в чьи обязанности входит ее заполнение.

Полагая, что для каждой операции над документом необходим ровно один человек, можем сказать, что мы уже задействовали для работы над ним двух человек — разработчика и заполнителя бланка. После того, как этот бланк будет заполнен, к работе подключится еще один человек — тот, для кого этот документ предназначен. Предположим, что этот человек выполнит только две операции: проверит правильность заполнения формы и после этого поместит ее в общее хранилище документов. Проще говоря, достанет из шкафа соответствующую папку, положит в нее документ и поставит ее на место.

Пока процесс обработки документа выглядит достаточно просто, но и здесь могут возникнуть некоторые трудности. Если предположить, что разработчик формы в своей работе ограничился только перечислением некоторых

общих указаний и рекомендаций по структуре и внешнему виду документа, то может возникнуть такая ситуация, когда из-за неверного заполнения документ будет проходить все стадии обработки несколько раз. Служащий составляет документ, проверяющий находит ошибки и несоответствия и возвращает документ на доработку. Далее, после того, как служащий мягко поговорит на эту тему с разработчиком, документ дорабатывается как по структуре, так и по содержимому и снова отправляется на проверку. Здесь можно сделать такой вывод: для эффективной работы над документом функции каждого участника этой работы должны быть четко разграничены, но при этом обработчики документа должны сотрудничать, а не трудиться в полной изоляции друг от друга.

Теперь усложним задачу, предположив, что в формировании содержимого и структуры документа участвуют несколько человек, которые в своей деятельности оперируют некоторыми другими дополнительными документами. В этих условиях так же, как и в предыдущем случае, процесс обработки документа можно разбить на три стадии: разработка структуры документа, наполнение его информационным содержанием и помещение его в единое хранилище документов. Пусть каждой стадией занимается отдельная группа работников. Эти группы мы назовем соответственно разработчиками, пользователями и администраторами.

Рассмотрим схему работы этих рабочих групп. На первом этапе разработчики подготавливают первоначальный макет документа, который поступает группе пользователей. Далее, после того, как в соответствии с этим макетом в документ будет введена необходимая информация, он поступит на обработку группе администраторов. Если эта группа сочтет нужным включить в документ какую-либо дополнительную информацию, хранящуюся в других документах, то она отправит все эти (в том числе и формируемый) документы в группу разработчиков с требованием вставить по определенной схеме дополнительную информацию в создаваемый документ.

Следует отметить, что процесс обработки документа происходит и внутри каждой группы. Каждый из разработчиков, например, может отвечать за структуру какой-то одной определенной части документа, а каждый из пользователей — за информационное наполнение этой части. Таким образом, документ перемещается от одного сотрудника к другому, изменяя свою структуру и содержимое. Такой процесс обработки документа называется документооборотом. А последовательность служащих, через чьи руки он проходит, называется маршрутом прохождения документа.

Для формализации схемы вышеописанного документооборота перечислим те операции над документами, которые должны быть реализованы в ее рамках:

- обработка всей или части структуры документа (здесь и далее в списке под обработкой мы будем понимать такие элементарные операции, как Добавление, удаление и редактирование обрабатываемого контента);

- обработка любой части информационного содержимого документа;
- формирование структуры и содержимого документа на основе каких-либо других документов.

Далее следует список требований, которым должна удовлетворять реализация документооборота с такой схемой:

- все множество документов должно быть объединено в эффективно организованное единое хранилище документов;
- маршрут прохождения документа должен содержать минимальное число стадий его обработки;
- переход документа от одного служащего к другому должен занимать минимальное время;
- взаимодействие с единым хранилищем также должно занимать минимальное время (здесь под взаимодействием мы понимаем следующие операции: поиск одного или выборка нескольких документов, а также помещение в хранилище окончательно сформированного документа).

В начале главы мы описали те возможности, которые нам позволяет реализовать Web-сервис. И если вернуться к этому описанию, то можно заметить, что решение любой из приведенных задач очень хорошо вписывается в вышеописанную схему документооборота. Поэтому под этими словами мы далее будем подразумевать не только обработку документов внутри какого-либо предприятия, но и совокупность всех тех операций, которые способно производить над информацией любое Web-приложение. Здесь, правда, нужно сделать одну оговорку.

Говоря о Web-приложениях, мы среди его функций называли возможность производить какие-либо действия на стороне сервера. Например, администрировать какую-либо серверную службу, т. е. по запросу пользователя, сделанного посредством браузера, создавать или изменять настройки этой службы. Такая возможность не была включена в вышеописанную схему документооборота. Но с другой стороны, мы ведь описывали обработку бумажных, а не электронных документов, а бумага, как известно, самостоятельно никаких действий производить не может. Поэтому далее под документооборотом мы будем понимать документооборот в старом смысле, а также возможность осуществлять какие-либо действия на стороне сервера. При этом мы назовем его электронным документооборотом, и хотя далее для краткости мы будем опускать слово "электронный", под документооборотом мы будем подразумевать именно электронный документооборот.

Задавшись целью описать те средства, которые нам предоставляют Web-технологии и с помощью которых можно реализовать электронный документооборот, начнем с рассмотрения самого первого требования, предъявляемого его схеме, — займемся эффективной организацией единого храни-

лища документов. Для этого предположим, что вся имеющаяся информация оцифрована, т. е. отсканирована и отформатирована. Иными словами, теперь мы будем оперировать электронными документами, представленными файлами в текстовом формате.

В соответствии с какими-либо критериями создадим на сервере папки, по которым и распределим все имеющиеся файлы. Так как информация сохранена и находится в одном месте, можно сказать, что единое информационное хранилище создано. Теперь нам нужно организовать к ней доступ.

Самым простым способом организовать множественный доступ к серверу в локальной сети будет объявление вышеописанных папок сетевыми разделяемыми ресурсами. При грамотном описании прав доступа каждого пользователя к каждой папке можно утверждать, что с точки зрения простоты такая реализация хранилища документов достаточно эффективна. Но, вспомнив о существовании Internet, нам придется заняться чем-то более сложным, нежели сетевое администрирование папок.

Организуя документооборот в Internet, мы имеем возможность, выбрать для этой цели какую-либо одну или несколько из таких сетевых служб, как Web-сервер, сервер электронной почты, новостной сервер, сервер chat и т. д. Мы остановим свой выбор на Web-сервере и при этом предположим, что читатель осведомлен о том, какие функции выполняет каждая из указанных и неуказанных служб, и поэтому не станем аргументировать и обосновывать наш выбор.

При разработке представленных ниже приложений мы использовали Web-сервер Apache версии 1.3.20, на примере которого мы и опишем основные функции этой сетевой службы. Мы не будем здесь касаться вопросов установки и конфигурирования этого Web-сервера, т. к. подробнейшую информацию об этом можно найти на сайте разработчика **http://www.apache.org/docs/.** Итак, будем считать, что Web-сервер установлен и все его возможности изучены. Прописав в его конфигурационном файле те папки, в которых хранятся наши документы, и преобразовав эти документы в формат HTML, получим еще одну реализацию документооборота.

После того как созданная информационная система протестирована, мы можем перечислить те возможности, которые она предоставляет в связи с использованием Apache:

- организован множественный авторизованный доступ к документам;
- некоторые части документов можно генерировать динамически, с использованием языка PHP, интерфейса CGI, и при поддержке некоторых баз данных в документ на стороне сервера может быть вставлена какая-либо информация, формируемая на основе сделанного пользователем запроса;

- контент передается пользователю не в "сыром" виде, а в HTML-представлении, т. е. информация имеет определенное разработчиками оформление;
- все действия пользователей подробно протоколируются в log-файлах;
- для увеличения гибкости организации информационного хранилища может быть реализован виртуальный хостинг, позволяющий использовать сервер как несколько независимых серверов.

Разумеется, это далеко не полный список всех возможностей Apache. Здесь мы перечислили только те его функции, которые отвечают за выполнение некоторых, но не всех, требований, которым должна удовлетворять создаваемая информационная система.

Заметим, что Web-сервер не занимается непосредственно обработкой документов, вместо этого он просто передает имеющийся в наличии документ клиенту, которым в нашем случае является браузер. Иными словами, с помощью одного лишь Web-сервера мы не можем организовать эффективное прохождение документа по заданному маршруту. Более того, даже простейшие возможности редактирования документа им не поддерживаются. Единственными данными, которые пользователь в состоянии изменить, являются либо те из них, с которыми посредством CGI взаимодействует внешнее приложение, либо те, которые хранятся в базе данных.

Совершенно очевидно, что для создания эффективной системы документооборота возможности Web-сервера должны быть значительно расширены. Этим мы и займемся далее в этой главе. На данном этапе мы можем утверждать только то, что нами была заложена основа будущей информационной системы. Или, иными словами, информационная система была создана, далее последует расширение ее функциональности.

6.2. XML как модель данных

Прежде чем приступить к обработке информации, необходимо определить, в каком виде она будет храниться. Если отвлечься от обработки данных как процесса манипулирования документами, то перед нами встает более общая проблема: организация тех данных, которые фигурируют в рамках какой-либо поставленной задачи.

Организацию этих данных можно условно разбить на следующие этапы.

1. Отбор необходимых и отсев ненужных данных.
2. Структурирование данных, полученных на первом этапе.
3. Реализация этой структуры.

После того как информация собрана (первый этап), нужно каким-либо образом ее упорядочить (второй этап). Говоря о разных видах хранения информации, мы имели в виду такие способы ее представления, как, например документы Microsoft Word или Excel, HTML-документы и т. п. Кроме этих, существует великое множество подобных способов хранения данных, как простых, так и очень сложных. Для того чтобы систематизировать все многообразие способов представления информации, служат так называемые модели данных, для которых характерен некоторый формализм.

Второй этап организации данных как раз и представляет собой выбор для них наиболее подходящей модели. В настоящее время для этих целей (в том случае, когда количество информации достаточно велико) наиболее часто используются реляционные базы данных, в которых, как следует из их названия, используется реляционная модель. Грубо говоря, данные, реализованные в рамках подобной модели, представляют собой множество связанных между собой таблиц.

Возвращаясь к задаче построения документооборота средствами Web-технологий, вспомним, что описанный нами в первой части книги язык XML подразумевает древовидную структуру созданных с его помощью документов. Представление информации, при котором данные хранятся в узлах некоего дерева, принято называть иерархической моделью данных.

Вне всякого сомнения, помимо двух вышеупомянутых можно назвать еще несколько достаточно эффективных формальных способов представления информации. Но мы в этом разделе ограничимся сравнением и сопоставлением только реляционной и иерархической моделей, т. к. именно они используются чаще других и, кроме того, остальные модели данных представляются нам служащими для весьма специфических целей. Если попробовать поискать программное обеспечение, реализующее, например, объектно-ориентированную или сетевую модели, то список подобных программ будет весьма мал по сравнению с количеством всевозможных реляционных СУБД, HTML- и XML-редакторов и парсеров, а также других аналогичных им приложений.

Последнее утверждение ни в коей мере не должно послужить для аргументации предпочтения одних средств организации информации другим. Здесь мы рассмотрим только два, как нам кажется, самых распространенных из них, а именно область применимости и уровень взаимодействия Реляционных баз данных и конструкций языка XML. Причем мы не будем здесь как-либо затрагивать такие области знаний, как реляционная алгебра, а также общую теорию построения баз данных. Напомним, что нашей задачей является построение именно небольшого Web-приложения, поэтому от читателя не потребуется понимания даже такого процесса, как нормализация баз данных. Далее мы просто покажем, как данные, организованные в виде таблиц, могут быть тем же способом организованы средствами языка XML.

Рассмотрим следующий пример. Пусть требуется создать электронную ку- линарную книгу, которой будет пользоваться шеф-повар какого-нибудь рес- торана. Эта книга будет содержать линейный список рецептов, каждый из которых включает в себя перечень продуктов, необходимых для приготовления блюда, с указанием их количества, а также способ его приготовления. Говоря о линейности списка рецептов, мы имеем в виду, что не будем рас- сматривать реализацию перекрестных ссылок между рецептами, т. е. таких, например, как "соус готовится точно так же, как в рецепте таком-то", или каких-либо других связей между ними.

Легко заметить, что описание этого множества рецептов будет весьма труд. но осуществить с помощью реляционной СУБД. Действительно, в уважающем себя ресторане одних только специй может быть несколько сотен видов. И если мы попытаемся создать базу данных, состоящую только из одной таблицы, каждая запись которой будет представлять рецепт со списком ингредиентов, то в каждой такой записи большинство ячеек будут пустыми, что говорит об избыточности структуры данной базы (рис. 6.1). Кроме того, каждая попытка добавления нового ингредиента к данной базе повлечет за собой изменение ее структуры.

Рецепт / Ингредиент	Тесто для расстегая	Творожная масса	Заварной крем	Соус острый
Мука	500 г		20 г	40 г
Яйцо	3 шт.	2 шт.	1 шт.	
Сахар	100 г	200 г	250 г	2г
Молоко	200 г			
Творог		500 г		
Соль	2 г			5 г
Томатный соус				20 г
Бульон				200 г
Дрожжи	15г			
Перец				5г
Сл. масло		2г	300г	
Ванилин		2г		

Рис. 6.1. База данных рецептов в первом приближении

Как же мы можем избавиться от этой избыточности? Можно поступить сле- дующим образом: разделить все используемые продукты на несколько категорий так, чтобы каждая из них включала в себя те продукты, которые наиболее часто используются вместе. Примерами таких категорий могут быть: морепродукты, продукты, необходимые для выпечки, и т. д. Таким образом, вместо одной мы получим несколько таблиц, которые очень похожи на пер- вую (рис. 6.2).

Категория 1

Ингредиент \ Рецепт	Тесто для расстегая	Заварной крем
Мука	500 г	20 г
Яйцо	3 шт.	1 шт.
Сахар	100 г	250 г
Молоко	200 г	
Соль	2 г	
Дрожжи	15 г	
Сл. масло		300 г

Категория 2

Ингредиент \ Рецепт	Творожная масса
Яйцо	2 шт.
Сахар	200 г
Творог	500 г
Сл. масло	2г
Ванилин	2г

Категория 3

Ингредиент \ Рецепт	Соус острый
Мука	40 г
Сахар	2 г
Соль	5г
Томатный соус	20 г
Бульон	200г
Перец	5г

Рис. 6.2. Более эффективная реализация БД рецептов

Как видим, пустых ячеек стало гораздо меньше, но они не исчезли полностью. Кроме того, нам пришлось ввести дополнительную информацию — категории, которая вовсе не является необходимой. Это означает, что пользователь создаваемой информационной системы — шеф-повар — вовсе не обязан помимо названия какого-либо продукта вводить еще и его категорию просто потому, что так удобнее администратору базы данных.

Предположим, что единственной сложной операцией помимо элементарных (добавления, удаления и редактирования рецептов), выполняемой поваром, является следующая. Готовясь, например, к банкету и отобрав необходимые рецепты, он с помощью кулинарной книги хочет выяснить, в каком количестве ему понадобится тот или иной продукт. Допустим, выясняет, сколько необходимо яиц для приготовления всех выбранных блюд. В этих условиях, во втором предложенном решении мы ушли от одной избыточности, представляющей собой большое количество незаполненных ячеек первой таблицы, но пришли к другой, заключающейся в добавлении к базе служебной информации.

Рассмотрим третий вариант. Подходя к решению данной задачи формально, нормализуем имеющиеся данные. Здесь абсолютно не важно, что собой представляет и как реализуется процесс нормализации, поэтому мы не будем его описывать, а просто предъявим получившуюся третью нормальную Форму (рис. 6.3).

Рецепт	Ингредиент	Количество
Тесто для расстегая	Мука	500 г
Тесто для расстегая	Яйцо	3 шт.
Тесто для расстегая	Сахар	100 г
Тесто для расстегая	Молоко	200 г
Тесто для расстегая	Соль	2г
Тесто для расстегая	Дрожжи	15г
Творожная масса	Яйцо	2 шт.
Творожная масса	Сахар	200 г
Творожная масса	Творог	500 г
Творожная масса	Сл. масло	2г
Творожная масса	Ванилин	2г
Заварной крем	Мука	20 г
Заварной крем	Яйцо	1 шт.
Заварной крем	Сахар	250 г
Заварной крем	Сл. масло	300 г
Соус острый	Мука	40 г
Соус острый	Сахар	2г
Соус острый	Соль	5г
Соус острый	Томатный соус	20 г
Соус острый	Бульон	200г
Соус острый	Перец	5г

Рис. 6.3. Нормализованная БД рецептов

Теперь для подсчета количества какого-либо продукта повар должен создать еще одну таблицу, которая будет содержать список выбранных рецептов (рис. 6.4).

Рецепты
Тесто для расстегая
Творожная масса
Заварной крем

Рис. 6.4. Список рецептов блюд для банкета

После чего, записав соответствующий запрос SQL к таблице 1 (см. рис. 6.3), он сможет вывести требуемое значение. В том случае, когда повару нужно получить общее количество яиц, необходимое для приготовления всех блюд, представленных второй таблицей (см. рис. 6.4), по этому запросу в первой таблице будет подсчитана сумма значений тех ячеек, второе поле которой представляет собой значение `'яйца'`, а значение первого поля `'рецепт'` присутствует в какой-либо строке второй таблицы.

Последнее предъявленное решение будет оптимальным с точки зрения структуры построенной базы данных. Но будет ли оно таковым с точки зре-

ния естественности представления данных? Предположим, что у пользова-
теля перед глазами находятся таблицы, представленные на рис. 6.3 и 6.4.
Так как повар не обязан знать язык запросов SQL, самым удобным для него
будет проделать всю работу с помощью обычного просмотра обеих таблиц.
Иными словами, вооружившись карманным или электронным калькулято-
ром и по очереди сверяя значения этих таблиц, он после нескольких пере-
счетов узнает необходимое ему количество продукта.

Возвращаясь к первому и, может быть, второму решению задачи, заметим,
что, работая с соответствующими таблицами (см. рис. 6.1 и 6.2) и пользуясь
тем же калькулятором, повар получит ответ быстрее. К каким же выводам
приводят нас эти рассуждения? Неужели применение реляционных баз дан-
ных настолько не эффективно? И более того, неужели администратору этой
базы вместо оптимальной третьей нормальной формы следует использовать
какие-либо другие представления данных, пусть даже и избыточные?

И да, и нет. Разумеется, выдуманный нами шеф-повар, подсчитывающий
яйца с помощью калькулятора, водя пальцем по экрану монитора, выглядит
смешно. Конечно же, заботливый администратор баз данных предоставит
ему соответствующий пользовательский интерфейс, с помощью которого
легко можно будет как отредактировать имеющиеся данные, так и произве-
сти любые подсчеты. Поэтому, с одной стороны, ответом на вопросы пре-
дыдущего абзаца будет — нет.

Вернемся теперь к описанной в предыдущем разделе схеме электронного
документооборота. Очевидно, что шеф-повар, являющийся в нашем приме-
ре членом группы пользователей информационной системы, останется до-
волен результатами труда администратора баз данных. Последний в нашем
случае принадлежит группе разработчиков, т. к. именно он отвечает за
структуру документа, который заполняется поваром. Итак, если взглянуть
на данную информационную систему глазами разработчика, то можно ли
считать ее эффективной с его точки зрения?

Ранее мы обратили внимание на такой аспект представления данных, как
его естественность. Наиболее естественной формой записи рецептов являет-
ся обычный отформатированный текст. Здесь под форматированием пони-
мается его разбивка на абзацы и, может быть, разделы, главы и другие
структурные единицы. Но такое представление информации не функцио-
нально: очень трудным представляется не только такая относительно слож-
ная операция, как вычисление общего количества какого-либо ингредиента,
но и простой поиск рецепта. С другой стороны, функциональными являют-
ся реляционные базы данных с их языками запроса.

Для того чтобы объединить и функциональность обработки, и естествен-
ность представления данных, администратор базы данных должен, во-
первых, придать всей обрабатываемой информации структуру третьей нор-
мальной формы и, во-вторых, создать некий пользовательский интерфейс

для взаимодействия пользователя с этой информацией. Проще говоря, этот интерфейс служит для преобразования информации, находящейся в естественной форме, которая доступна пользователю, к виду, который доступен для обработки средствами СУБД. Кроме того, этот интерфейс должен обеспечивать и обратное преобразование.

Таким образом, можно утверждать, что предъявленное решение задачи весьма не эффективно с точки зрения разработчика, т. к. обе стоящие перед ним задачи — нормализация данных и организация пользовательского интерфейса для удобного к ним доступа — довольно сложны. Кроме того, при формулировке задачи мы отказались от реализации перекрестных ссылок между рецептами. Если бы мы этого не сделали, а наоборот, добавили бы это и еще несколько требований, которые могут возникнуть при решении реальной задачи, то выяснилось бы, что далеко не всякий из администраторов баз данных способен достичь поставленной цели, причем за приемлемое время.

Попробуем теперь решить данную проблему средствами языка XML. Для этого запишем один рецепт, используя синтаксис этого языка

```
<receipt>
 <title>Расстегай</title>
...
 Тесто для расстегая

Смешать <egg> <quantity>3</quantity> <measure>шт</measure>яйца</egg> со
<sugar> <quantity>100</quantity> <measure>г</measure>сахара</sugar>, добавить <salt> <quantity>2</quantity> <measure>г</measure> соли </salt>,
влить <milk> <quantity>200</quantity> <measure>г</measure> молока</milk>.
Затем добавить <meal> <quantity>200</quantity> <measure>г</measure> муки
</meal> и <yeast> <quantity>15</quantity> <measure>г</measure>дрожжей
</yeast>. Замесить негустое тесто. Когда тесто хорошо поднимется (увеличится примерно в два раза), добавить оставшиеся <meal>
<quantity>300</quantity> <measure>г</measure> муки, хорошо перемешать и оставить в теплом месте примерно на 30 мин. для того, чтобы тесто еще немного подошло. Теперь тесто готово для приготовления расстегая.
...
</receipt>
```

В таком виде рецепты могут храниться как в одном, так и в нескольких файлах XML. Для того чтобы подсчитать необходимое количество яиц, повару потребуется в каком-либо файле создать код:

```
<receipts>
 <receipt>Название рецепта</receipt>
 <receipt>Название рецепта</receipt>
...
```

```
<receipt>Название рецепта</receipt>
</receipts>
```

представляющий собой список рецептов тех блюд, которые необходимо приготовить для банкета.

После этого где-либо в документе, в зависимости от того, где должно выводиться искомое значение, ему будет нужно использовать конструкцию:

```
<xsl:value-of
    select="sum(//eggs[parent::receipt/title=//receipts/*/name())"/>
```

Здесь мы предполагаем, что элемент receipts находится в том же самом файле, что и сами рецепты. Легко видеть, что с помощью языка XML задача решается гораздо более эффективно. Ведь даже если формированием документов займется повар, а не разработчик, то ему это будет сделать гораздо легче, чем создавать и обрабатывать таблицы базы данных. Но на самом деле, ему не придется делать и этого.

Заметим, что естественность представления информации при использовании XML почти такая же, как и у текстового файла. Вместе с тем формирование документов также не представляет никакой трудности. После того, как разработчик создаст правила шаблона для всех элементов, которые могут встретиться в документе, он может отдать список этих элементов пользователю, причем не обязательно шеф-повару, с тем, чтобы последний заключил некоторые фрагменты ранее подготовленного текста в соответствующие теги. Так каждый рецепт и каждый ингредиент, например мясо, образуют соответственно контент элементов receipt и meat.

Заметим также, что функциональность информационной системы высока почти так же, как и СУБД. Язык XPath позволяет делать запросы, которые вряд ли можно назвать более сложными, чем их аналоги из SQL. Далее, в следующей части книги, будет показано, как обрабатывать данные, находящиеся во внешнем источнике информации. Причем под обработкой здесь понимается не только выборка информации, реализуемая средствами XPath, но и такие операции, как создание, чтение, запись и удаление наборов узлов, расположенных во внешнем XML-документе.

Проанализировав все вышесказанное, может сложиться впечатление, что автор предлагает уничтожить СУБД как класс и повсеместно использовать только язык XML. Разумеется, такие выводы нельзя делать, основываясь только на одном примере. В реальной жизни могут возникать такие задачи, для решения которых подходят самые разные средства. Чтобы не быть голословными, приведем еще один пример.

Рассмотрим некую гипотетическую компьютерную фирму, которая обслуживает нескольких постоянных клиентов. Деятельность этой фирмы заклю-

чается в том, что ее сотрудники, выезжая к клиентам, производят замену некоторых устаревших или вышедших из строя комплектующих, ранее этим клиентам поставленных. Причем профессиональный уровень сотрудников может быть как низким, так и высоким. Иными словами, если одни из них способны на настройку любого оборудования, то другие — только на конфигурирование простейших устройств.

Предположим, что глава данной фирмы каждый месяц требует отчета о проделанных операциях, причем этот отчет должен содержать помимо описания проделанной работы еще и сведения о том, кто из сотрудников ее выполнил и для какого клиента. Кроме этого, также должна быть предусмотрена возможность просмотра данных как по сотруднику, так и по клиенту.

На рис. 6.5 показан пример базы данных, в которой хранится вся вышеописанная информация.

Операции	Дата	**Сотрудник**	**Клиент**	Работа
Операция 1	01.01.01	Сотрудник 1	Клиент 1	Описание работы
...	...	...	...	...
Операция N	02.01.01	Сотрудник N	Клиент N	Описание работы

Работники	ФИО	Профессиональный уровень
Сотрудник 1	ФИО 1	Профессиональный уровень 1
...	...	...
Сотрудник N	ФИО N	Профессиональный уровень N

Клиенты	Компания	Телефон
Клиент 1	Компания 1	Телефон 1
...	...	...
Клиент N	Компания N	Телефон N

Рис. 6.5. База данных, описывающая деятельность фирмы

Легко видеть, что верхняя таблица, изображенная на рис. 6.5, представляет собой практически готовый отчет, который должен быть предоставлен руководству. Используя связи по полям "Сотрудник" и "Клиент", можно легко получить информацию как об одном, так и о другом. Таким образом, можно утверждать, что построенная информационная система легка в обработке для сотрудников любой рабочей группы (разработчиков, пользователей и администраторов).

Попробуем теперь реализовать эту систему с помощью XML. Рассмотрим следующий код:

```
<operations>
 <operation>
  <data>01.01.01</data>
  <worker>Сотрудник 1</worker>
  <client>Клиент 1</client>
  <!-- описание работы -->
 </operation>
...
 <operation>
  <data>02.01.01</data>
  <worker>Сотрудник N</worker>
  <client>Клиент N</client>
  <!-- описание работы -->
 </operation>
</operations>
```

описывающий все операции, проделанные сотрудниками фирмы в январе 2001 года. Далее следует код, в котором представлены сведения обо всех сотрудникахфирмы:

```
<workers>
 <worker>
  <id> Сотрудник 1 </id>
  <name>ФИО 1</name>
  <prof_level>профессиональный уровень 1</prof_level>
 </worker>
...
 <worker>
  <id> Сотрудник N </id>
  <name>ФИО N</name>
  <prof_level>профессиональный уровень N</prof_level>
 </worker>
</workers>
```

И во многом похожий на него код, описывающий клиентов фирмы:

```
<companies>
 <company>
  <id> Клиент 1 </id>
```

```
  <name>Компания 1</name>
  <telephone>Номер телефона 1</telephone>
 </company>
...
 <company>
  <id> Клиент 1 </id>
  <name>Компания N</name>
  <telephone>Номер телефона N</telephone>
 </company>
</companies>
```

Как видно из вышеприведенного кода, для решения поставленной задачи более всего подходят именно реляционные СУБД. Это следует хотя бы из того, что в данном коде гораздо больше разметки, которая, вообще говоря, является служебной информацией, чем самих данных. А для этой разметки еще нужно определить как способ отображения данных, так и все операции над ними. Между тем, вся требуемая функциональность системы может быть реализована с помощью встроенных возможностей реляционной СУБД.

Подвоя итоги, можно сказать, что для большинства задач применимы оба средства: и язык XML, и реляционная СУБД. Фактически мы здесь на примерах показали, как одна и та же информация может быть представлена как с помощью иерархической модели данных, так и с помощью реляционной. Выбор между этими двумя моделями не всегда однозначен. На наш взгляд, в тех случаях, когда должны обрабатываться документы, состоящие в основном из текста, в котором достаточно разреженно присутствуют данные, подлежащие анализу, лучше всего использовать язык XML. Если же форма организации информации близка к табличной, то здесь лучше всего воспользоваться средствами, предлагаемыми реляционными СУБД.

Кроме того, не стоит забывать и о том, что реляционные базы данных были созданы достаточно давно и, поэтому, содержат очень богатый инструментарий обработки информации, мощно развившийся за десятилетия разработок. С его помощью могут быть использованы такие возможности некоторых СУБД, как механизм транзакций, управление данными в реальном времени, компактная форма хранения данных и многое другое.

Здесь важно заметить, что рассмотренные средства организации данных не являются взаимоисключающими по отношению друг к другу. И действительно, вполне может сложиться ситуация, когда одна часть информации, подлежащей обработке, может быть более эффективно представлена с помощью иерархической модели данных, а другая — реляционной.

5.3. Многослойная архитектура разработки XML-документов

Предположим, что так или иначе мы определили структуру той информации, которую будет обрабатывать создаваемое Web-приложение. Теперь можно приступать непосредственно к его разработке.

рассмотрим тривиальный пример, реализующий описанную ранее схему документооборота. Допустим, что разработчик создал некую форму, поля которой должен заполнить пользователь, а также средства отображения введенных данных, т. е. отчет, который проверяется администратором. Наиболее часто встречающейся реализацией этой схемы в Internet на наш взгляд является заполнение пользователем регистрационной HTML-формы, в которой он указывает свои данные: фамилию, имя, город и страну проживания и т. п. Как правило, после этого ему становятся доступными некоторые дополнительные ресурсы соответствующего сервера.

Что же должен сделать разработчик для того, чтобы создать такое довольно простое Web-приложение? Можно сказать, что вся его деятельность будет проходить в три этапа: во-первых, он должен создать Web-страницу, содержащую непосредственно саму форму, во-вторых, им должна быть обеспечена обработка сервером введенных с ее помощью данных. Как правило, после того как данные введены и обработаны, пользователю предъявляется еще одна Web-страница, служащая для подтверждения их правильности. Создание этой Web-страницы и будет третьим, заключительным этапом разработки.

Не затрагивая вопроса обработки данных сервером, его мы коснемся несколько позже, рассмотрим процесс создания исходной и конечной Web-страниц. Начнем с первой. Напомним, что мы говорим о Web-приложении, которое должно функционировать в рамках небольшого предприятия. Поэтому, а также потому, что на практике задачи предприятия далеко не всегда являются такими простыми, мы подойдем к созданию формы весьма серьезно.

Безотносительно того, какой инструментарий использует разработчик (XML, HTML или что-либо еще), ему потребуется обеспечить соответствие Web-страницы следующим условиям:

1. Каждый компонент страницы должен быть красиво оформлен. Это условие подразумевает подбор шрифтов, создание статичных и анимационных картинок, кнопок и т. д.
2. Все компоненты Web-страницы должны быть соответствующим образом скомпонованы относительно друг друга. Здесь имеется в виду выбор местоположения этих компонент, подбор для них цветов в зависимости от этого местоположения, соответствующее оформление фона и т. д.

3. Ввод данных должен быть максимально удобным и функциональным. Здесь под функциональностью может пониматься, например, проверка введенных пользователем данных средствами JavaScript. Эта проверка несколько отличается от той, которая производится на сервере. В частности с помощью JavaScript можно организовать принудительное заполнение некоторых полей формы. Иными словами, если пользователь оставит эти поля пустыми, то данные отправляться на сервер не будут вовсе, вместо этого будет выводиться соответствующее сообщение.

Как видно из этого списка, разработчик должен быть одновременно и художником (первое условие), и дизайнером (второе условие), и программистом (третье условие). Так как в реальной жизни подобных многогранных личностей очень немного, то становится актуальным вопрос о разделении труда между тремя группами разработчиков (художников, дизайнеров и программистов). Далее мы дополним этот перечень еще двумя группами. Заметим, что по аналогии с художниками может быть введена еще и группа, отвечающая за разработку звуковых эффектов, но т. к. эта группа (композиторов) в принципе ничем не отличается от художников, ее мы рассматривать не будем.

Рассмотрим те средства, которые для этого разделения труда предлагает нам язык HTML. Для описания внешнего вида каждого компонента в этом языке служат так называемые каскадные стилевые таблицы (Cascading Style Sheet, CSS). С их помощью определение внешнего вида компонента, записанное с использованием специального синтаксиса, может быть либо вынесено в отдельный файл, либо расположено в заголовке документа, либо встроено непосредственно в описание самого компонента. Подробнее об использовании CSS мы расскажем в следующей главе. Здесь же мы отметим только одно: благодаря тому, что описания стилей могут находиться в одном или нескольких внешних источниках, для художников становится возможным отделенная от других групп работа над проектом.

Точно так же, как и описания стилей, во внешний источник данных можно вынести описание объектов и функций языка JavaScript, что позволяет отделить деятельность программистов от деятельности остальных групп. Таким образом, на долю дизайнеров остается только манипулирование непосредственно HTML-кодом, в котором вместо описания конкретных стилей встречаются лишь их названия, а вместо кода JavaScript — вызовы ранее описанных функций.

Вообще говоря, разделение труда между группами разработчиков, предлагаемое связкой CSS+HTML+JavaScript, является весьма условным. Действительно, для того чтобы дизайнер мог эффектно расположить в окне браузера компоненты Web-страницы, ему необходимо активно общаться с художником. Ведь художник не сможет по отдельности нарисовать эти компоненты, не зная о том, как их расположит дизайнер. В частности и хУ"

дожник., и дизайнер должны будут прийти к единому мнению относительно цвета каждого компонента, будь то шрифт, кнопка, фон и т. п.

Кроме того, для вставки в HTML-документ вызовов функций JavaScript или, например, небольших обработчиков событий необходимо, чтобы программист имел представление о структуре Web-страницы, определяемой дизайнером, а дизайнер — о том, как будет реагировать эта структура в зависимости от тех или иных действий пользователя, за обработку которых отвечает программист.

ранее мы говорили, что деятельность полностью изолированных друг от друга рабочих групп невозможна. Иными словами, разделение труда между ними подразумевает отдельное ведение каждой группой своей основной деятельности в тесном контакте с остальными группами. Говоря об условности достигнутого разделения труда, мы хотели подчеркнуть, что в описанных условиях члены каждой группы, помимо своей специализации, должны владеть еще и знаниями, являющимися специализациями других групп.

Заметим, однако, что, например, дизайнер не обязан уметь программировать на JavaScript, вместо этого он должен лишь иметь представление о предназначении и возможностях этого языка. Программист не должен знать, какими средствами пользуется художник для создания своих рисунков, ему будет достаточно знать их размеры. Вообще говоря, член любой рабочей группы не обязан владеть технологиями, использующимися членами другой рабочей группы так, как ими владеют они сами, но он должен иметь о них некоторое поверхностное представление. Следовательно, можно сделать вывод, что с помощью CSS, HTML и JavaScript деятельность художников, дизайнеров и программистов будет в должной мере разделена между этими группами.

Кроме того, заметим, что на практике даже самое отдаленное представление разработчика любой группы о деятельности других рабочих групп необязательно, т. к. для осуществления связи между группами в коллективах присутствуют так называемые координаторы, которые, зная "обо всем понемногу", и обеспечивают слаженность совместной работы этих групп над проектом.

Итак, мы вкратце рассмотрели те возможности, которые в этой связи нам предоставляет язык HTML. Для того чтобы в этом же аспекте рассмотреть Преимущества XML, перейдем к процессу создания второй Web-страницы, в которой должна быть представлена введенная пользователем информация или, иными словами, отчет.

При разработке первой Web-страницы мы разделили группу разработчиков на три подгруппы: художники, дизайнеры и программисты, но мы ничего не сказали о группе пользователей. Это произошло по той простой причине, что пользователи отвечают за информационное наполнение документа, а наша Web-страница как HTML-документ не содержала никакой информации. Другими словами, данные, подлежащие обработке, содержались не в самом документе, а вводились посредством формы. В случае со второй Web-страницей это не так, она уже содержит некоторые данные, которые были

введены ранее членом группы пользователей и теперь являются информационным наполнением документа или, для краткости, контентом.

Как и в предыдущем случае, рассмотрим, что должна сделать группа разработчиков для создания отчета. Сразу заметим, что т. к. вторая Web-страница не содержит формы и ее обработчика, то поле деятельности программистов будет гораздо более узким, чем в случае разработки первой Web-страницы. Еще одним отличием будет то, что на плечи разработчиков ляжет дополнительная задача — обработка контента. Так как для решения этой задачи не подходят ни художники, ни дизайнеры, ни тем более программисты, мы добавим к ним еще одну рабочую группу, которую назовем редакторами.

В обязанности этой группы будет входить правка текста, его форматирование, т. е. разбиение на разделы различного уровня вложенности и абзацы, объявление некоторых его фрагментов элементами списков и т. п. Кроме того, в общем случае редакторы должны обеспечить такую структуру информации, при которой было бы возможным быстрое выполнение таких операций, как поиск и выборка данных по какому-либо критерию, получение доступа к различным документам и их частям, а также все те операции, с помощью которых обработка контента становится максимально гибкой и функциональной.

Предположим, что после того, как пользователь ввел требуемую информацию, она была обработана сервером, в результате чего был сгенерирован файл, в котором и содержится эта информация. Рассмотрим две ситуации: в первой данные в файле будут сохранены в формате HTML, а во второй — в формате XML.

Не имеет смысла показывать, что для решения поставленной задачи XML подходит больше, чем HTML. Это очевидное утверждение напрямую следует из того факта, что язык XML имеет богатые возможности для работы со структурой документа, которые полностью отсутствуют в HTML. Примером может послужить приложение, представляющее собой электронную кулинарную книгу, описанное в предыдущем разделе. Возможность обозначения рецептов целиком, их названий и ингредиентов отдельными тегами позволяет решить все те задачи, которые ставятся перед редактором документа. С другой стороны, HTML предназначен только для работы с представлением данных и, следовательно, ставить перед ним задачу, связанную с их структурой, просто бессмысленно.

Поэтому мы здесь попытаемся ответить только на один вопрос: как с помощью XML можно организовать совместную деятельность различных рабочих групп по созданию одного проекта? Для этого предположим, что рассматриваемый нами XML-файл не был сгенерирован Web-приложением, а был создан одним из членов группы редакторов.

Несмотря на то, что для коллективной разработки Web-приложения мы выдумали большое количество всевозможных рабочих групп, на самом деле

все они (кроме редакторов) решают только одну задачу, а именно пытаются максимально эффективно реализовать представление данных. Теперь, когда в работу включился XML, будет обрабатываться еще и их структура, за которую и отвечает группа редакторов.

Рассуждая о достоинствах языка XML, принято говорить о разделении именно этих двух составляющих XML-документа: структурированной информации (контента) и ее представления.

Действительно, после того как данные структурированы с помощью разметки XML, или иными словами создан XML-файл, к обработке подключается рабочая группа, отвечающая за их представление, — создается сценарий XSLT, служащий для определения правил интерпретации этой разметки. Причем на этом этапе деятельность членов последней рабочей группы не исключает возможности их разделения на художников, дизайнеров и программистов. Ведь в сценарии XSLT могут совместно использоваться и CSS, и HTML, и JavaScript.

Заметим, что, как и при создании первой Web-страницы, рабочие группы не работают в полной изоляции друг от друга. Те разработчики, которые заняты обработкой контента, должны иметь представление об интерпретации созданной ими разметки, и наоборот, разработчики, занимающиеся интерпретацией, должны знать об особенностях форматирования XML-файла.

Подводя итоги, можно сказать, что полноценный XML-документ, т. е. документ, использующий всю функциональность языка XML, имеет многослойную архитектуру. Среди этих слоев принято выделять три. Два из них мы уже назвали. Первый отвечает за обработку контента, второй — за его представление. Третий слой отвечает за серверную обработку данных, которую мы пообещали описать в начале раздела. Этот слой — логический — можно назвать самым важным, т. к. именно он делает Web-приложение тем, чем оно является, — приложением, а не просто электронным средством массовой информации. Об этом слое — следующий раздел.

6.4. Логический слой: Java

Взаимодействие пользователя с Web-приложением заключается в том, что пользователь формирует некоторый пакет данных и, возможно, описывает некоторые особенности их обработки, после чего отправляет эту информацию Web-приложению. Далее Web-приложение обрабатывает полученную информацию, и результат отправляется обратно пользователю. Описанный процесс может повторяться многократно до тех пор, пока пользователь не сочтет полученные результаты удовлетворительными.

Примером такого взаимодействия может послужить описанное в *главе 4* Web-приложение, выполняющее различные операции над каталогом аудио-

файлов. Этот пример включает в себя несколько предопределенных действий, которые доступны для выполнения пользователем, а именно поиск аудиофайлов по различным критериям. Но серверная обработка информации не всегда является настолько простой. Если, например, предоставить пользователю возможность самому формировать критерии для этого поиска, то сложность Web-приложения возрастет на порядок, причем всего приложения целиком, а не только его серверной части.

Очевидно, что для обеспечения минимальной функциональности обработки данных серверная часть Web-приложения должна обладать такими способностями, как обработка этих данных с условием, циклическая обработка обработка данных, находящихся во внешнем источнике, и т. д. В первой части книги мы показали, как эти возможности реализуются в языке XML. Но язык XML предназначен в первую очередь для описания структуры данных, и поэтому его возможностей недостаточно. Это можно легко проиллюстрировать на таких задачах, как вычисление максимального значения среди элементов линейного списка или, например, их среднего арифметического. Читатель может попытаться решить эти задачи для списка, заданного следующим образом:

```
<list>
    <item>10</item>
    <item>5</item>
   ...
    <item>20</item>
    <item>15</item>
</list>
```

Несмотря на простоту структуры данных (ведь элементы списка могли бы быть разбросаны по разным частям документа или даже нескольких документов), нахождение максимального значения, в частности, сильно усложняется тем обстоятельством, что переменной, созданной с помощью `xsl:variable` ИЛИ xsl:param, Невозможно ПРИСВОИТЬ новое значение. Иными словами, все переменные XML являются константами.

Можно привести еще массу примеров, когда язык XML как средство серверной обработки данных оказывается частично или полностью несостоятельным. И здесь нам на помощь приходит Java. Разумеется, помимо Java существует еще несколько средств программирования, среди которых самыми распространенными, на наш взгляд, являются PHP, Perl и ASP. Мы не будем здесь рассматривать достоинства и недостатки этих средств по сравнению с Java. Задачей этого раздела является только демонстрация возможностей этого языка как альтернативы вышеперечисленным, а также другим инструментам серверной обработки данных.

6.4.1. Java: основные возможности

Java представляет собой объектно-ориентированный язык программирования, похожий на C++, но более простой в использовании. В более широком смысле Java можно назвать целой технологией разработки приложений, выполняющихся на самых разных устройствах, которые могут быть связаны между собой. Перед тем как перейти к сетевым возможностям Java, рассмотрим его просто в виде некоего языка программирования.

Итак, как мы уже сказали, язык Java обладает большей простотой, нежели C++. В чем же заключается эта простота? Главным образом это связано с некоторыми особенностями механизма управления памятью, реализованным в Java. Такие понятия, как указатели и ссылки, в Java отсутствуют полностью. Кроме того, разработчику нет необходимости выполнять высвобождение памяти, занятой ранее использованными переменными, Java с помощью сборщика мусора сделает это вместо него. В общем, можно сказать, что простота Java состоит в том, что многие функции, за выполнение которых в C++ отвечает разработчик, в Java являются автоматизированными.

Нет сомнений в том, что такие возможности, как управление памятью, множественное наследование, обработка данных самых разнообразных типов (в том числе и созданных разработчиком), значительно расширяют возможности языка. Но с другой стороны, это значительно увеличивает время его разработки. Можно сказать, что Java представляет собой язык программирования более высоко уровня, чем C++. Точно так же, как C++ является языком более высокого уровня по сравнению, например, с ассемблером. У каждого из этих языков есть своя область применимости, и если редко кому придет в голову задействовать в разработке Web-приложения язык C++, то точно так же редко, на наш взгляд, создаются автономные приложения на Java.

Вторым несомненным достоинством Java является его переносимость. Это означает, что приложение Java будет работать не только на той платформе, где оно было создано и отлажено, но и на многих других. Разработчики этого языка стремятся, чтобы приложения Java были способны выполняться на всех существующих на сегодняшний день устройствах, обладающих соответствующей аппаратной частью. К этим устройствам принадлежат не только всевозможные виды компьютеров, но и бытовая техника, средства связи, промышленное оборудование и многое другое. Но к сожалению, единая среда разработки приложений — пока только недостижимый идеал.

Со своей стороны добавим, что для автора кроссплатформенность Java позволила не участвовать в глупейших с точки зрения разработчика спорах о том, какая операционная система лучше: Windows или Linux.

Следующим достоинством Java является безопасность его кода. Здесь под безопасностью понимается стабильность работы Java-приложений, главным Фактором которой является ранее нами упомянутое свойство Java брать на

себя управление памятью. Не секрет, что главным источником головной боли для программистов на C++ является поиск кода, в котором производится либо удаление раннее удаленных переменных (как правило, экземпляров классов с помощью деструктора), либо попытка записи информации за пределы зарезервированной для этой цели области памяти (как правило, это присвоение какого-либо значения элементу массива, индекс которого превосходит размерность этого массива).

В Java такие ситуации исключены полностью. Виртуальная машина, взяв на себя управление памятью, проверяет код на наличие ошибок дважды — на этапе компиляции программы, а также во время ее исполнения. И как следствие, разработчику не нужно во всем приложении отыскивать код, повлекший за собой полную остановку приложения или даже всей операционной системы. Именно такого рода ошибки значительно снижают скорость разработки приложений на C++, и, соответственно, их отсутствие повышает скорость разработки приложений на Java.

Среди недостатков Java, как правило, упоминается избыточность байтового кода скомпилированного приложения, и как следствие, чрезмерное потребление ресурсов и невысокая скорость. Действительно, эти недостатки имели место у первых версий виртуальных машин, но на сегодняшний день как ресурсопотребление, так и скорость Java-приложений улучшились в десятки раз и являются приемлемыми для широкого круга задач, хотя, конечно, и не достигают уровня C++.

И напоследок, соотнесем скорость разработки Java-приложений с высоким уровнем этого языка. Разработчик, решая поставленные задачи, заинтересован, прежде всего, в том, чтобы создаваемое им приложение удовлетворяло всем выдвинутым заказчиком требованиям. И только во вторую очередь он занимается максимальной оптимизацией получившегося кода. Учитывая, что, как мы уже сказали, ресурсопотребление и скорость выполнения приложений Java находятся на приемлемом уровне, актуальность второй задачи снижается, и тем самым разработчик получает возможность целиком сосредоточиться на выполнении свих непосредственных задач.

В качестве иллюстрации к вышесказанному можно упомянуть механизм управления потоками в Java. Благодаря этому механизму разработчик, не сосредотачиваясь на таких низкоуровневых операциях, как организация буфера потока, его заполнение и очистка и т. п., может заняться непосредственно вводом/выводом данных из этого потока. Заметим, что определенные в Java классы потоков позволяют множеством разных способов манипулировать данными самых различных типов.

Другим примером может послужить реализованный на Java интерфейс доступа к приложениям различных производителей, выполняющимся параллельно друг с другом. На сегодняшний день почти каждый серьезный продукт содержит набор классов Java, называемый API (Application Program

Interface — Прикладной Программный Интерфейс), предназначенный для взаимодействия с этим приложением посредством Java. Причем сетевые возможности этого языка позволяют взаимодействовать приложениями, которые могут находиться где угодно в Internet.

6.4.2. Пакет разработчика

Перед тем как приступить к серверной обработке данных с помощью Java, нам необходимо установить программное обеспечение, которое предоставит возможность использовать конструкции этого языка. Это программное обеспечение состоит из двух частей: виртуальной машины Java, главной задачей которой является преобразование исходного кода в байтовый код, и библиотеки классов, реализующих набор низкоуровневых операций. В этом разделе мы не будем описывать процесс установки этого программного обеспечения. Мы расскажем здесь только о некоторых разделах библиотеки классов, позволяющих применять в разработке некоторые как стандартные, так и не совсем очевидные возможности языка.

Для наших разработок мы воспользовались программным обеспечением фирмы Sun Microsystems, Inc. Его полное название — Java™ 2 Software Development Kit Standard Edition, который мы для краткости обозначим как J2SE1.3, где J означает Java, SE — Standard Edition (стандартная редакция), 1.3 — номер версии используемого продукта. Этот продукт является бесплатным, и его, а также множество разнообразной документации, можно найти по адресу **http://java.sun.com.** Теперь рассмотрим некоторые выгодные особенности Java, реализованные теми наборами классов, которые включены в состав данного пакета разработчика.

Апплеты

Java-апплет представляет собой байт-код Java, выполняющийся на стороне клиента в клиент-серверном приложении. Так как в нашей книге клиентом является браузер Microsoft Internet Explorer, рассмотрим, как он взаимодействует с апплетами. Для вставки апплета в Web-страницу служит элемент `applet` языка HTML. Этот элемент обладает среди прочих атрибутами code, представляющим собой URL файла, содержащего Java-класс, являющийся апплетом, и codebase, в котором указывается URL папки, где находится этот класс. Более новым средством вставки апплета является элемент `object`, также обладающий атрибутами code и codebase.

В сети Web апплеты чаще всего представлены графическими индикаторами количества посещений Web-страницы, разнообразными часами, элементами управления и т. п. Но вообще говоря, апплеты не всегда бывают видимыми. Иногда они не являются интерактивными и служат для выполнения функций, зависящих от действий пользователя только косвенным образом. На-

пример, далее в книге будет описан Java-апплет, поддерживающий связь с сервером, который по требованию клиентской части Web-приложения загружает с этого сервера ту или иную информацию.

Для работоспособности любого Java-апплета нет необходимости устанавливать J2SE1.3 на сервер или на рабочие станции. Со стороны сервера он представлен файлом, содержащим байт-код, который просто выкачивается клиентом. Клиент же, которым является MSIE, в свою очередь, выполняет этот байт-код, используя встроенную в него виртуальную машину Java.

Примечание

MSIE, начиная с версии 6.0, по умолчанию не содержит виртуальной машины Java. Ее нужно скачивать с сайта разработчика отдельно.

Но J2SE1.3 понадобится для разработки апплета, а именно для компиляции исходного кода, результатом которой будет вышеупомянутый байт-код.

Графика

С помощью языка HTML можно создавать Web-сайты, обладающие самым изощренным дизайном. Технология DOM и язык JavaScript обеспечивают при этом весьма высокий уровень интерактивности этого сайта. Но с другой стороны, для придания Web-странице максимальной красочности и насыщенности, как правило, используются такие средства, как анимированные GIF-изображения, Flash-ролики и т. п. Это обусловлено тем, что HTML не содержит даже самого примитивного инструментария для работы с графикой.

Единственная вещь, которую способен выполнить HTML, — это просто вставить рисунок в Web-страницу. Такие задачи, как поворот изображения, обработка графических примитивов (отрезок, прямоугольник, круг и др.), использование различных эффектов вывода изображения этому языку недоступны. Поэтому для решений этих и гораздо более сложных задач по обработке графики применяются отдельные от HTML способы представления графических объектов. Два из них мы уже назвали, это — GIF-изображения и Flash-ролики. Разумеется, графических форматов и инструментария для создания рисунков гораздо больше, но мы рассмотрим только те возможности, которые в этой области нам предоставляет язык Java.

AWT (Abstract Window Toolkit — абстрактный инструментарий оконного интерфейса) служит для разработки GUI (Graphical User Interface — графический пользовательский интерфейс). Пакет классов java.awt, используемый при создании апплета, позволяет конструировать такие элементы управления, как кнопки, выпадающие меню, деревья, которые обладают как высокой степенью интерактивности, так и богатством графического оформления.

Кроме того, Java-апплеты очень тесно интегрируются с HTML и с языком JavaScript, что дает возможность разработчику организовать взаимодействие пользователя с апплетом не напрямую, а через какие-либо остальные компоненты Web-страницы. Например, нажатие кнопки мыши где-либо на Web-странице может вызвать перерисовку апплета, который находится вовсе не там, где была нажата эта кнопка. И наоборот, щелчок по апплету может повлечь за собой изменение внешнего вида той части Web-страницы, которая находится неподалеку от области, занимаемой апплетом.

Документация по J2SE1.3 сообщает о следующих достоинствах библиотеки AWT:

- множество компонент пользовательского интерфейса;
- мощная модель управления событиями;
- множество графических инструментов, включая классы форм (shapes), цветов (colors) и шрифтов (fonts);
- менеджеры схем расположения окон, не зависимые от конкретного размера окна и его разрешения;
- механизм drag-and-drop ("тащи и бросай"), позволяющий осуществлять перенос объектов с использованием буфера обмена той операционной системы, в которой выполняется приложение.

К этому можно добавить, что, применяя AWT, можно создать приложение, поддерживающее обработку изображений на нескольких мониторах. С помощью класса `java.awt.Robot` можно генерировать события от мыши и клавиатуры, что позволяет, например, создавать обучающие программы. Так как библиотека AWT слишком велика для того, чтобы приводить здесь хоть сколько-нибудь подробное ее описание, перейдем к следующему пакету классов по обработке графики — javax.swing.

Согласно документации проект Swing (Project Swing) представляет собой часть базовых классов Java (Java Foundation Classes, JFC), состоящую из множества компонент GUI со встраиваемым механизмом "look and feel". Этот механизм позволяет оформить некоторый набор компонент GUI так, чтобы они автоматически принимали такой внешний вид, который является естественным для той операционной системы, в которой выполняется приложение. Существует возможность установить внешний вид приложения какого-либо из следующих типов: Java Look & Feel (стандартный или по Умолчанию Look & Feel, подходит для любой операционной системы), Windows Look & Feel (L & F для Windows), CDE/Motif Look & Feel (L & F для платформ Sun), Mac OS Look & Feel и т. д.

Проект Swing включает в себя как версии существующих компонент AWT (кнопки, полосы прокрутки), так и великое множество высокоуровневых компонент, среди которых:

- `Japplet` — класс, представляющий собой расширение класса java.applet.Applet;
- `JButton` — класс, прорисовывающий и позиционирующий кнопку;
- `JComboBox` — класс, представляющий компонент, в котором объединены кнопка или текстовое поле и прокручиваемый список;
- `JDialog` — класс, представляющий диалоговое окно;
- `JList` — класс, представляющий компонент, который позволяет пользователю выбрать какой-либо элемент списка;
- `JTable` — класс, представляющий компонент, служащий для построения двумерной таблицы;
- JTextArea — класс, представляющий многострочную область текста без разметки с возможностью его редактирования;
- `JTextField` — то же, что и JTextArea, только область текста является однострочной;
- `JToolBar` — класс, реализующий панель управления, т. е. панель, с помощью которой можно выполнить наиболее часто встречающиеся действия;
- `JTree` — класс, позволяющий оперировать иерархически организованными данными, представленными в виде дерева, каждый узел которого может быть свернут и развернут (аналогия — левое окно проводника Windows).

Кроме большого количества компонент GUI, проект Swing предоставляет возможность манипулировать слоями, прозрачными и непрозрачными панелями, виртуальными экранами и т. п.

Заканчивая обзор графических возможностей Java, отметим, что в том случае, когда в качестве клиента Web-приложения используется браузер MSIE, для того чтобы можно было использовать все преимущества Swing, необходимо на соответствующее клиентское рабочее место установить J2SE1.3. Если же используется AWT, такой установки не потребуется.

Операции ввода/вывода

За операции ввода/вывода в Java отвечает пакет `java.io`. Все они могут производиться не только из автономного приложения, но и из Java-апплетов и Java-сервлетов.

Базовым классом для операций над файлами и папками является `java.io.File`, который позволяет их создавать и удалять. Кроме того, с помощью этого класса можно создавать временные файлы, а также такие, которые будут удаляться сразу после их закрытия. Для организации входного и выходного буферизованных потоков служат соответственно классы

java.io.FileInputStream И java.io.FileOutputStream. А ДЛЯ чтения И записи В НИХ, соответственно — FileReader И FileWriter.

Но в Java подобные действия можно проводить не только над файлами. Существует множество локальных и сетевых сервисов Java, которые обрабатывают данные, полученные из какого-либо потока (stream) и отправляют результат в другой поток. Например, далее мы покажем, как с помощью этого механизма можно обработать данные, которые пользователь ввел в поля какой-либо Web-формы. В данном случае эти данные будут доставлены на сервер в заголовке HTTP-запроса, для чтения которого мы откроем буферизованный поток.

С помощью потоков можно обрабатывать не только "сырые" данные, но и обладающие конкретным типом. Причем эти типы могут быть не только базовыми (строки, символы, байтовые массивы...), но и достаточно сложными. Используя, например, класс ObjectInputStream, можно читать из потока объекты произвольного типа.

Помимо вышеописанных возможностей, пакет java.io позволяет обрабатывать ограничения доступа к файлам и папкам (класс FilePermission), взаимодействовать С каналами (классы PipedInputStream И PipedOutputStream), производить синтаксический анализ (parsing) данных, полученных из потока (класс StreamTokenizer), а также выполнять множество других функций.

Интернационализация

По мере роста Internet возникает все большая потребность в поддержке приложениями культурной, политической и географической специфики того региона, в котором эти приложения потенциально могут быть запущены. Эта поддержка включает в себя не только возможность обрабатывать текст на любом языке мира, но и его формат, единицы измерения и пр., соответствующие стране этого языка. Разработчики Java создали для этих целей единый механизм, позволяющий решить проблему интернационализации приложений один единственный раз, а не для каждого разрабатываемого с нуля или модифицируемого под какой-либо дополнительный регион приложения.

Для поддержки национальных особенностей среды, в которой выполняются приложения, в Java определен класс java.util.Locale, который мы условно назовем локализацией. С помощью этого класса можно воспользоваться существующими типами локализаций в зависимости от выбранного региона или создать свою собственную. Для создания экземпляра класса Locale нужно применить один из двух конструкторов:

```
Locale(String language, String country)
```

или

```
Locale(String language, String country, String variant)
```

Изак. 582

Как можно заметить, первые два аргумента обоих конструкторов задают соответственно язык и страну. Второй конструктор требует задания третьего аргумента — variant. Этот аргумент зависит от опционально используемого браузера, а также от платформы, в которой выполняется приложение. Значениями аргумента variant могут быть, например, WIN — для Windows, MAC — для Macintosh, POSIX — для POSIX и т. д.

Для использования определенных в Java типов локализации можно воспользоваться ПОЛЯМИ класса Locale, например, `Locale.US`, `Locale.UK` И Т. П. Методы этого класса служат в основном для манипулирования аргументами конструктора, а также различных значений по умолчанию.

При интернационализации класс Locale используется в качестве аргумента для методов и конструкторов тех классов, которые зависят от национальных особенностей региона, в котором выполняется приложение. Таким образом, единожды создав свой класс Locale, программист может использовать его где-либо в своих приложениях.

Примерами классов, ИСПОЛЬЗУЮЩИХ Locale, ЯВЛЯЮТСЯ `java.util.` Calendar и `java.util.TimeZone`, которые, как следует из их названия, служат для управления соответственно временными характеристиками (эра, год, месяц, день недели, число, час, минута, секунда, миллисекунда) и временной зоной (смещение, зависящее от часового пояса). Класс calendar является единственным и полнофункциональным средством для управления временем. Используя созданные ранее экземпляры классов Locale и `TimeZone`, с его помощью можно модифицировать класс Date, который служит только для хранения какой-либо даты.

Для того чтобы задать формат отображения даты, служит класс `java.text.DateFormat`. Родителем этого класса является абстрактный класс `java.text.Format`, у которого помимо DateFormat есть еще два непосредственных потомка — `java.text.NumberFormat` И `java.text.MessageFormat`, которые, как следует из их названий, применяются для форматирования (представления), соответственно, чисел и текстовых сообщений. Классы DateFormat и NumberFormat также являются абстрактными, и непосредственным форматированием занимаются их потомки.

Здесь мы не будем полностью приводить описание механизма локализации Java-приложения, в силу его объемности. Отметим только те возможности Java, которыми воспользуемся в наших последующих разработках. При работе **С потоками (классы** `InputStreamWriter`, `OutputStreamWriter`) И строками (класс string) у нас возникнет потребность в обработке русскоязычного текста, которую мы не сможем реализовать с помощью Locale по умолчанию. Такие операции, как, например, сравнение строк, чтение или запись в поток, будут генерировать исключения, если при этом используется русскоязычный текст.

Для решения этой проблемы нужно воспользоваться строковым аргументом enc конструкторов вышеприведенных классов. Мы не будем здесь приводить формы записи этих конструкторов, т. к. описываемые классы имеют по несколько таких конструкторов. Кроме того, существует еще несколько классов, для корректной работы которых с русскоязычным текстом необходимо правильно указать кодировку с помощью аргумента enc. Далее мы приводим допустимые значения этого аргумента для русскоязычного текста:

- `UTF8` — 8-битный Unicode;
- `UTF-16` — 16-битный Unicode;
- `Cp866` — русскоязычный текст MS-DOS;
- `Cp1251` — русскоязычный текст Windows;
- `ISO8859_5` — русскоязычный текст по стандарту ISO;
- `KOI8_R` — русскоязычный текст Unix/Linux;
- `MacCyrillic` — русскоязычный текст Mac OS.

Заметим, что значение кодировки должно быть задано в том регистре символов, который использован в вышеприведенном списке.

Сетевые возможности

Для осуществления связи между несколькими компьютерами Java предлагает два механизма. Первый реализует связь через т. н. сокеты (socket — гнездо). С помощью сокетов можно установить соединение с любым сервером Internet или построить свой собственный сервер. Для создания клиент-серверного приложения, работающего по этому принципу, служит абстрактный класс `java.net.SocketImpl`, который используется для обработки как клиентских, так и серверных сокетов. Для обработки же только клиентских сокетов (или просто сокетов) служит класс `java.net.Socket`, а для серверных, соответственно, `java.net.ServerSocket`.

Для своих разработок мы будем использовать более высокоуровневые средства. А именно тот набор классов Java, который для организации запросов к серверу использует URL (Universal Resource Locator — уникальное местоположение ресурса). Этот набор классов и реализует второй механизм связи компьютеров в Internet.

Как мы уже говорили ранее, нами будет предъявлен Java-апплет, поддерживающий связь с сервером. Для его создания мы воспользуемся двумя классами: `java.net.URL` И `java.net.URLConnection`, первый ИЗ которых ПОЗВОЛИТ нам управлять адресом URL, а именно, разбивать его на подстроки, соответствующие названию протокола, доменному имени сервера, пути к файлу ресурса, а также строки запроса (query string). Второй — `URLConnection` — предназначен для установки связи, а также для чтения и

записи в файл ресурса, заданного конкретным URL. Подробнее об этом можно прочитать в *главе 8*.

Помимо вышеперечисленных возможностей пакет java.net содержит также различные средства защиты информации (классы Authenticator, NetPermission и SocketPermission), а также инструменты, обеспечивающие дополнительную функциональность механизмов связи между компьютерами.

Безопасность

Изначально модель обеспечения безопасности Java была довольно примитивной. В JDK 1.0 (Java Development Kit version 1.0) ее принцип заключался в том, что локальный код имел неограниченные права доступа к ресурсам системы, в которой он выполнялся. Тот же код, который загружался из удаленной системы, т. е. удаленный код, исполнялся с некоторыми ограничениями. Например, Java-апплет не мог осуществлять операции чтения/записи над локальной файловой системой. В JDK 1.1 было введено понятие подписи. Подписанный удаленный код рассматривался как локальный, если его цифровая подпись была допустимой или доверяемой.

Начиная с версии 1.2 (платформа Java 2), структура механизма безопасности становится гораздо более функциональной и гибкой. Она предусматривает комплексный контроль уровня доступа и политик безопасности, которые пользователь может как настраивать, так и добавлять свои собственные, а также различные способы шифрования данных и многое другое. Причем каждый отдельный аспект обеспечения безопасности является легко расширяемым под нужды конкретного сетевого приложения. Далее вкратце рассмотрим некоторые принципы защиты информации, предлагаемые платформой Java 2.

Ограничения (permissions) прав доступа какого-либо кода не исчерпываются лишь разрешением или запрещением операций над файловой системой. Механизм обеспечения безопасности Java 2 позволяет управлять уровнем доступа к самым разным системным ресурсам, таким, например, как сокет. Предком всех классов, отвечающих за какие-либо ограничения, является java.security.Permission. Мы не будем здесь полностью описывать механизм управления ограничениями, а вместо этого приведем список некоторых классов, из названий которых можно получить представление о том, к каким системным ресурсам могут быть обеспечены различные уровни доступа.

Подклассы java.security.Permission:

- java.security.BasicPermission;
- java.io.FilePermission;
- java.net.SocketPermission.

Подклассы java.security.BasicPermission:

- javax.sound.sampled.AudioPermission;
- java.awt.AWTPermission;
- java.net.NetPermission;
- java.util.PropertyPermission;
- java.lang.RuntimePermission;
- java.security.SecurityPermission;
- java.io.SerializablePermission;
- java.sql.SqlPermission.

Самым простым способом определения ограничений будет создание специального файла политик (Policy file), который устанавливает соответствие между подписантами исполняемого кода, источником этого кода, представленным URL, именем класса, описывающего нужное ограничение, подписантами этого класса, идентификатором объекта, над которым данный код выполняет действие, и самим этим действием. Иными словами, этот файл сообщает о том, может ли какой-либо код, подписанный некоторыми лицами, производить то или иное действие, также кем-либо подписанное, над определенным объектом. Заметим, что все вышеперечисленные компоненты, кроме самого ограничения, в некоторых случаях могут быть опциональными.

Для того чтобы проверить, имеет ли право тот или иной код на какое-либо действие, служит метод checkPermission Класса java.security. AccessController, аргументом которого должен быть класс, описывающий это право или ограничение. Помимо этой функции, класс AccessControiier может помечать код как привилегированный, т. е. способный выполнять некоторые дополнительные действия, выходящие за рамки определенного для этого уровня доступа.

Как мы уже сказали, политики управления правами доступа для каждого пользователя могут храниться в отдельных файлах. Причем эти файлы могут иметь как текстовый формат, так и какой-либо другой. В первом случае такой файл можно отредактировать либо с помощью любого текстового редактора, либо с помощью специальной утилиты policytool, входящей в Комплект поставки J2SE1.3. Если же политики хранятся в файле, формат которого отличен от текстового, например, он может представлять собой набор объектов Policy (класс java.security.Policy), то в этом случае некоторые из вышеперечисленных классов нуждаются в переопределении.

Помимо этой может возникнуть множество различных ситуаций, когда стандартные классы Java, отвечающие за безопасность, нужно переопределить, т. е. расширить их функциональность в зависимости от специфики решаемой задачи. За проверку допустимости всех элементарных операций, таких, например, как чтение, запись, удаление, установка соединения, отве-

чает так называемый менеджер безопасности, представленный классом java.lang.SecurityManager. Этот класс отвечает за весь механизм безопасности Java, и следовательно, переопределяя его, разработчик может полностью изменить принципы работы этого механизма. Заметим, что утилита appletviewer, которая, как следует из ее названия, служит для просмотра апплетов, а также большинство браузеров, в том числе от фирм Netscape и Microsoft, содержат свой собственный менеджер безопасности.

Еще одним из главнейших классов, отвечающих за безопасность, является java.security.ProtectionDomain, **который** СЛУЖИТ ДЛЯ **разбиения на** группы различных источников кода, представленных каким-либо URL, и сопоставления каждой из этих групп отдельного набора ограничений. Этот класс применяется в основном, когда существует множество различных как внешних, так и внутренних источников кода, обладающих различными уровнями доступа. Причем количество этих источников настолько велико, что нуждается в разбиении на группы, сформированные по каким-либо критериям. Кроме того, этот класс можно использовать для определения своих собственных способов манипулирования как источниками кода, так и соответствующими ограничениями.

Для проверки степени доверия того или иного кода служит механизм цифровых подписей. Те подписи, которые предназначены для проверки степени доверия подписантов, описанных в файлах политик, как правило, хранятся в отдельном файле. Этот файл обозначается как keystore, т. е. хранилище ключей. Для редактирования этого хранилища предусмотрена утилита keytool, входящая в комплект поставки J2SE1.3.

Понятия цифровой подписи и ключей относится к криптографии, возможности которой в Java представлены очень широко. В основе криптографической структуры платформы Java 2 лежит понятие "провайдер", которое, как правило, обозначает пакет или набор пакетов, объединяющее те классы Java, которые реализуют самые разные алгоритмы шифрования, генерирования ключей, цифровых подписей и т. п. Для работы с этими алгоритмами служит класс java.security.Provider, который позволяет использовать сразу несколько провайдеров от различных разработчиков. В J2SE1.3 имеется встроенный провайдер, который носит имя "SUN".

Далее приведем список алгоритмов различного назначения, предлагаемый провайдером "SUN".

- SHA-1, MD2, MD5 — алгоритмы шифрования сообщений (message digest algorithms);
- DSA — алгоритм цифрового подписания (Digital Signature Algorithm), данный алгоритм используется для генерирования пар ключей используемых цифровой подписью;
- SHA1withDSA — алгоритм цифрового подписания, использующий соответственно SHA1 для шифрования и DSA для генерирования ключей;

- RSA — алгоритм шифрования, предназначенный в данном случае для генерирования ключей; название представляет собой аббревиатуру, образованную именами его создания — Rivest, Shamir и Adlemen;
- MD2withRSA, MD5withRSA, SHA1withRSA — алгоритмы цифрового подписания, основанные на RSA (генерирования ключей) и использующие соответственно MD2, MD5 и SHA1 для шифрования.

О завершение описания криптографической структуры платформы Java 2 приведем список основных классов, обеспечивающих ее функциональность:

- `java.security.Provider` — позволяет работать сразу с несколькими провайдерами;
- `java.security.MessageDigest` — **реализует алгоритм шифрования СО-общений, поставляемый провайдером;**
- `java.security.Signature` — обеспечивает функциональность цифровых подписей;
- интерфейс `java.security.Key` — описывает ключи, обладающие тремя характеристиками: алгоритм, способ шифрования и формат, является базовым **ПО отношению К интерфейсам** DSAPrivateKey, DSAPublicKey, PrivateKey, PublicKey, RSAPrivateCrtKey, RSAPrivateKey И `RSAPublicKey`, **служащим для описания ключей различных типов;**
- интерфейс `java.security.spec.KeySpec` — описывает спецификации для ключей (key specification), является базовым для многих классов и интерфейсов, служащих для описания различных спецификаций;
- `java.security.KeyFactory` — служит для манипулирования ключами, заданными с помощью разных спецификаций;
- пакет `java.security.cert` — обеспечивает функциональность сертификатов;
- **Классы пакета** java.security **С именами** KeyPair, KeyPairGenerator, KeypairGeneratorSpi — отвечают соответственно за хранение в памяти, генерирование и способы генерирования ключей;
- классы пакета `java.security` с именами Keystore и KeystoreSpi — отвечают за представление в памяти наборов ключей и сертификатов.

Поддержка архивов

Начиная с версии 1.1, в JDK встроена поддержка работы с архивами формата ZIP и GZIP. Эту функциональность обеспечивает пакет `java.util.zip`. Далее с пояснениями следует список некоторых классов этого пакета:

- `ZipFile` — обеспечивает такие операции над файлом архива, как открытие и закрытие этого файла, получение какого-либо компонента архива и т. д.;
- zipEntry — позволяет манипулировать каким-либо компонентом архива;

- Inflater — управляет распаковкой (разархивацией);
- Deflater — управляет упаковкой (архивацией).

С помощью пакета java.util.zip можно также выполнять поточные операции чтения/записи с файлами в ZIP-формате.

На базе формата ZIP в Java *2* реализован еще один специализированный формат архивирования, который называется JAR (Java Archive). В архиве этого формата могут находиться файлы самых разных типов — классы Java (class), рисунки, звуковые файлы и многое другое, что позволяет браузеру загружать некоторую их совокупность за одну транзакцию. А учитывая еще и то, что эти файлы могут быть сжаты, можно сказать, что скорость загрузки данных с сервера при использовании JAR-архива может быть значительно увеличена.

Ранее мы говорили о возможности электронной подписи для Java-апплетов. Файл, содержащий архив формата JAR, также можно пописать (с помощью утилиты jarsigner), что обеспечивает безопасное соединение между клиентом (например, Java-апплет) и сервером (JAR-файл). Отметим здесь, что для манипулирования Java-архивами можно воспользоваться как специальной утилитой jar, так и пакетом java.util.jar.

Взаимодействие с базами данных

При создании клиент-серверного приложения часто возникает потребность в хранении на сервере больших объемов данных. Этими данными может быть информация о пользователях приложения, статистика их обращений к серверу, или в случае с Internet-приложением, статистика посещений сайта, а также какие-либо специфические данные, для обработки которых создавалось данное приложение. Как правило, для решения подобных задач используются самые разные СУБД. И здесь нас будет интересовать один вопрос: как "подружить" выбранную СУБД с тем инструментарием, которым пользуется разработчик для создания приложения?

Суть этой проблемы состоит в том, что существует великое множество разного рода средств программирования, а средств манипулирования базами данных еще больше. Таким образом, для ее решения требуется некий универсальный интерфейс доступа к базам данных, посредством которого можно было бы из приложения, написанного если не на любом, то по крайней мере, на одном из самых распространенных языков программирования, управлять как структурой, так и контентом используемой базы данных.

Таким интерфейсом является ODBC (Open DataBase Connectivity — открытый [механизм] соединений с базами данных), который позволяет программам, написанным на C, взаимодействовать с теми базами данных, для которых используемая реализация ODBC содержит механизм этого взаимодействия. В общих чертах использование ODBC можно описать следующим

образом: разработчик какой-либо СУБД включает в состав своего продукта ODBC-драйвер, который может взаимодействовать с С-приложением для манипулирования этой СУБД.

Для Java интерфейсом доступа к базам данных является JDBC (Java DataBase Connectivity), названный по аналогии с ODBC. На самом деле JDBC не является самостоятельным интерфейсом. Принято говорить, что в Java для обеспечения доступа к базам данных служит мост JDBC-ODBC. Это означает, что для обеспечения этого доступа из Java-приложения разработчику этого приложения понадобятся два драйвера: ODBC-драйвер и его использующий JDBC-драйвер. Здесь возникает вопрос: почему нельзя воспользоваться только лишь одним ODBC-драйвером, ведь язык Java допускает вставки на языке С?

Во-первых, использование С-кода в Java-приложениях вызывает проблемы с переносимостью, безопасностью и скоростью выполнения этих приложений.

Во-вторых, перенос некоторого С-кода ODBC в Java нежелателен вследствие того, что в С существуют такие конструкции, которые не определены в Java. Самым ярким примером подобной конструкции могут послужить указатели.

В-третьих, ODBC гораздо более сложен в изучении, чем JDBC, т. к. язык С по отношению к Java является низкоуровневым языком.

За реализацию интерфейса JDBC отвечает пакет `java.sql`, в частности в его составе имеется класс `DriverManager`, который, как следует из его названия, служит для управления драйверами для различных баз данных. Аббревиатура sql в названии пакета говорит о том, что с его помощью можно выполнять запросы к подключенным базам данных с использованием языка SQL (Standard Query Language — Стандартный Язык Запросов).

Главная проблема, связанная с SQL, заключается в том, что стандартизация этого языка идет слишком медленно. Как следствие, разработчики баз данных, ориентируясь на потребности пользователей своего продукта, расширяют возможности этого языка по своему усмотрению. Для решения этой проблемы Java предлагает следующие средства.

- Класс java.sql.Types описывает общие типы данных, используемые в запросах SQL.
- Реализация любых дополнительных конструкций SQL возлагается на разработчика JDBC-драйвера. Язык Java допускает применение таких конструкций на страх и риск разработчика соответствующего Java-приложения.
- Интерфейс JDBC позволяет работать с так называемыми escape-операторами в стиле ODBC.
- Для того чтобы Java-приложение могло получить информацию о требованиях и возможностях конкретной базы данных, в пакете `java.sql` определен Интерфейс `DatabaseMetaData`.

Кроме того, разработчики JDBC-драйверов должны обеспечивать т. н. ANSI SQL-92 Entry Level. Здесь ANSI SQL-92 обозначает набор стандартов для SQL, разработанных ANSI (American National Standards Institute — Американский Национальный Институт Стандартизации) в 1992 году, а Entry Level — список возможностей SQL. Заметим, что хотя интерфейс JDBC версии 2.0 включает поддержку SQL3 и SQLJ, однако, разработчики JDBC-драйверов не обязаны обеспечивать эту поддержку.

В заключение приведем полный список задач, за выполнение которых отвечает интерфейс JDBC.

- Обеспечение связи с источником данных.
- Отправка SQL-запросов к базе данных.
- Получение и обновление результатов запроса.
- Преобразование типов, поддерживаемых в SQL, к типам данных Java.
- Получение информации о базе данных, а также о сгенерированном в ответ на запрос результате, который может иметь формат, специфический для данной базы данных.
- Обработка ошибок (исключений).
- Обеспечение безопасности.

Поддержка служб каталогов

При создании клиент-серверного приложения всегда возникает задача конфигурирования различных серверных служб под нужды этого приложения. Настройка политик пользователей, имеющих необходимый уровень доступа к этому серверу, прав доступа рабочих станций, обслуживаемых сервером, различных системных процессов и многих других серверных компонент представляет собой задачу, которая, как правило, является гораздо более сложной, чем создание клиент-серверного приложения.

При программировании и последующем администрировании серверов одной из глобальных проблем является унификация обслуживания различных служб и приложений, работающих под управлением этого сервера, а также данных, которые используют эти службы. Говоря о сложности решения этой проблемы, следует учитывать тот факт, что форматы конфигурационных файлов рассматриваемых служб, а также форматы ими обрабатываемы" файлов чаще всего отличаются друг от друга.

Для решения этой проблемы и предназначены так называемые службы каталогов (Directory Services). Благодаря характерной для этих служб иерархической структуре данных, обеспечивается возможность упорядочить или каталогизировать если не каждую "атомарную" единицу информации, то хотя бы некоторые наборы данных, сгруппированных по некоторому принципу.

Примерами таких наборов могут послужить характеристики компьютеров, описываемые службой DNS (Domain Name System — Система Доменных Имен), информация о пользователях как отдельно взятого приложения, так и всей системы в целом, описания компонентов файлового хранилища и многое другое. Заметим, что наличие единого сервиса, такого, как служба каталогов, позволяет избежать дублирования информации, которое возникает при децентрализованном управлении ресурсами.

Служба каталогов не обязана сама хранить какие-либо данные, вместо этого она может предоставлять ссылку на них, что способствует организации не только единого способа хранения данных, но также и единого доступа к ним. А такие достоинства, как единообразие запросов к данным, организованным самыми разными способами, возможность внесения каких-либо изменений только в одно место (например, смена пароля какого-либо пользователя не влечет за собой изменение этого пароля для каждого приложения, с которым он взаимодействует) не могут быть не оценены по достоинству администраторами серверов.

Принцип работы службы каталогов достаточно прост. Каждый узел дерева, представляющего иерархически упорядоченные данные, является набором некоторых атрибутов, описывающих этот узел. Легче всего понять это на следующем примере. Рассмотрим совокупность описаний сотрудников некоторой организации, которая помимо головного офиса имеет филиалы, каждый из которых в свою очередь подразделяется на отделы, а отделы, может быть, на еще более мелкие подразделения. В данном случае каждый узел дерева будет представлять собой описание одного сотрудника. Атрибутами узла будут какие-либо характеристики соответствующего сотрудника, возможно зависящие от того, в насколько крупном отделении организации он работает.

Учитывая вышесказанное, легко заметить аналогию между способами хранения информации службой каталога и языка XML. Точно так же эта аналогия прослеживается и между способами представления и управления данными этими двумя средствами. Естественно, такое сходство обеспечивает возможность их тесной интеграции друг с другом, что довольно часто наблюдается у появляющихся сегодня продуктов.

Для обеспечения взаимодействия со службами каталогов в Java предлагается интерфейс JNDI (Java Naming and Directory Interface — Java-интерфейс доступа к службе имен и каталога), не зависящий от какой-либо конкретной реализации службы каталогов. Его реализация представлена пакетом `javax.naming`. Он отвечает за такие базовые операции, как создание и удаление привязки имя-объект (name-to-object), поиск объектов по критерию, просмотр списка привязок имя-объект и т. д. Здесь под именем понимается ссылка на объект, а объект — описание соответствующего объекта, представленное экземпляром класса object.

Далее приведем список пакетов, входящих в состав javax.naming:

- javax.naming.directory — обеспечивает манипулирование именами объектами и атрибутами (установка привязки, создание и удаление атрибутов, поиск узлов и т. д.);
- javax.naming.event — служит для обработки событий, которые генерируются в результате осуществления элементарных операций, произведенных методами некоторых классов предыдущего пакета;
- javax.naming.ldap — позволяет приложению использовать протокол LDAP v3 (LightWeight Directory Access Protocol — Облегченный (легковесный) Протокол Доступа к Каталогу);
- javax.naming.spi — служит для переопределения некоторых, как правило, низкоуровневых операций над каталогом или для добавления своих собственных, специфических операций.

Кроме этих пакетов, используются также и те, которые поставляются вместе с конкретной реализацией службы каталогов. Как правило, эти дополнительные пакеты входят в состав продукта, если он является коммерческим, или поставляются отдельно, если продукт распространяется свободно.

6.4.3. Java и Internet: дополнительные возможности

В предыдущем разделе мы описали только основные области применения Java. И разумеется, функциональность платформы Java 2 ими не исчерпывается. В частности, мы не упомянули поддержку такого механизма, как RMI (Remote Method Invocation, что можно приблизительно перевести, как вызов удаленного кода), инструменты для обработки звука, различные математические конструкции и многое другое. Но все это можно найти в документации, свежую версию которой так же, как и J2SE1.3, можно найти на сайте компании Sun — **www.sun.com.**

Здесь же речь пойдет о дополнительных возможностях платформы Java 2, предлагаемых продуктом, который называется Java 2 Enterprise Edition, т. е. почти точно так же, как и описанный в предыдущем разделе. Единственное отличие в названиях составляет слово "Enterprise", которое говорит о том, что данный продукт содержит в себе не только стандартные возможности Java, но и дополнительные, необходимые для построения приложения уровня предприятия. Далее мы для краткости будем называть этот продукт J2EE.

Сервлеты

Первое, что мы рассмотрим в этом разделе, — механизм, позволяющий создавать Java-код, выполняющийся на стороне сервера. Сервлеты представля-

ют собой небольшие, кроссплатформенные Web-компоненты, которые динамически загружаются и выполняются сервером. Управление сервлетами осуществляется с помощью так называемого Servlet Container (далее просто контейнер), который работает как встроенный либо дополнительный компонент Web-сервера.

Контейнер реализует механизм взаимодействия сервлетов с Web-клиентами посредством парадигмы запрос-ответ (request-response), основывающейся на протоколе HTTP. Принцип работы этого механизма заключается в следующем. Web-клиент, например, браузер, посылает HTTP-запрос Web-серверу, далее этот запрос обрабатывается Web-сервером и перенаправляется контейнеру. Далее контейнер определяет, какому сервлету соответствует запрос, и после этого выполняет данный сервлет, передавая ему объекты, соответствующие запросу (request) И ответу (response).

Как правило, сервлеты служат для выполнения двух типов задач. В задачах первого типа они используются для внутренних целей сервера. В этом случае клиент не взаимодействует с сервлетом такого типа напрямую. Примером задачи, в которой могут быть применены подобные сервлеты, является проведение каких-либо математических расчетов сервером. Это может выглядеть примерно так: пользователь, взаимодействуя с некоей Web-страницей, представленной файлом в формате HTML, дает указание серверу выполнить некоторые расчеты. После подтверждения ввода, Web-страница посредством сервера вызывает сервлет с соответствующими параметрами. Далее сервлет выполняет требуемые расчеты и возвращает результаты пользователю. В качестве реального примера сервлета такого типа можно привести инструментарий публикаций Cocoon, который в качестве параметра принимает имя XML-файла и возвращает пользователю результат его преобразования, т. е. HTML-код.

Второй тип сервлетов служит для простого генерирования HTML-кода. Рассмотрим тривиальный пример.

Листинг 6.1. ExampleServlet.java

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

public class ExampleServlet extends HttpServlet {

 Public void doGet (HttpServletRequest request,
                    HttpServletResponse  response)
```

```
        throws IOException, ServletException   {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head>");
      out.println ("<title>Title </title>");
      out.println("</head>");
      out.println("<body>");
      out.println ("<h1>First level title</h1>") ;
      out.println ("<p>Some text</p>");
      out.println("<p>");
      out.println("Parametertext:"+request.getParameter("param")) ;
      out.println("</p>");
      out.println("</body>");
      out.println("</html>");
  }

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

Как видно из листинга 6.1, данный сервлет умеет делать две вещи: генерировать HTML-код и обрабатывать параметры HTTP-запроса. Для того чтобы продемонстрировать работу этого сервлета, нужно откомпилировать его Java-код с помощью команды javac ExampleServet.java, поместить получившийся в результате байт-код (в данном случае ExampleServet.class) в папку, содержимое которой Web-сервер способен интерпретировать как сервлеты, и затем, запустив браузер, набрать в его адресной строке

```
http://your_address.your_domain/path_to_servlet/
ExampleClass?param=param_value
```

Можно заметить, что принцип работы сервлетов достаточно прост, но вместе с тем использование всей полноты возможностей Java при динамическом генерировании Web-страниц позволяет создавать многофункциональные приложения с Web-интерфейсом. Поэтому сочетание простоты разработки и сложности разрабатываемых Web-приложений является неоспоримым достоинством Java-сервлетов.

Далее в книге мы еще вернемся к теме сервлетов с тем, чтобы подробно описать те их особенности, которые мы будем учитывать в наших разработках.

Java Server Pages (JSP)

В отличие от сервлетов, JSP предназначены только для динамического генерирования контента Web-документов в форматах HTML, XHTML, XML, DHTML и т. п. Можно сказать, что JSP представляют собой тот же самый механизм, что и сервлеты (второго) типа, только более высокого уровня. Давайте рассмотрим, как будет выглядеть JSP-страница (листинг 6.2), которая делает то же самое, что и сервлет, представленный листингом 6.1.

Листинг *6.2.* JSPExample.jsp

```
<html>
  <head>"
    <title> Title </title>
  </head>
  <body>
    <hl> First level title</hl>
    <p> Some text</p>
    <p>Parameter text:
       <% request.getParameter("param") %>
    </p>
  </body>
</html>
```

Как видно из листинга 6.2, совместное использование Java- и HTML-кода гораздо более просто, чем в сервлетах. Но вместе с тем, страницы JSP сохраняют всю функциональность сервлетов, за исключением того, что их выполнение происходит только по HTTP-запросу. С другой стороны, механизм JSP предоставляет возможность создавать библиотеки тегов (и даже пространства имен для них), которые служат для применения синтаксиса языков разметки при формировании логического слоя документа. Иными словами, вместо часто повторяющегося кода Java мы можем использовать ранее нами созданный тег, который при компиляции JSP-страницы будет заменен этим кодом.

Главное отличие JSP-страниц от сервлетов заключается в том, что сервлеты Представляют собой Java-код, который содержит разметку и текст документа, а JSP-страница сама является этим документом, содержащим небольшое количество Java-кода. Заметим, что количество Java-кода в странице JSP будет именно небольшим, т. к. механизм JSP позволяет загружать и выполнять внешний по отношению к этой странице Java-код. Этим обстоятельством заодно обеспечивается отделение представления данных от логического слоя документа.

Завершая описание JSP, отметим, что принцип работы JSP-страниц точно такой же, как и у сервлетов. То есть используется JSP-контейнер, базирующийся на парадигме запрос-ответ, взаимодействие клиента и JSP-страницы происходит по протоколу HTTP. Но, разумеется, две рассмотренные технологии имеют некоторые различия в их реализовывающих API.

Дополнительное ПО

Как мы уже говорили ранее, для обеспечения функциональности сервлетов и JSP можно использовать J2EE. Заметим сразу, что мы этого делать не будем из-за избыточности этого продукта. Здесь под избыточностью мы подразумеваем наличие у J2EE таких функций, которые выходят далеко за рамки сервлетов и JSP. Но, тем не менее, мы кратко опишем те возможности, которые нам предоставляет J2EE для разработки Web-приложений перед тем, как перейти к используемому нами программному обеспечению.

При разработке приложений уровня предприятия как правило задействуют так называемую трехуровневую модель приложения. Здесь первый уровень или слой представляют пользователи (клиенты) приложения, второй — обработчик запросов пользователей, и третий — хранилище информации. При этом каждый из этих слоев может быть распределен в сети. Рассмотрим схему на рис. 6.6.

Рис. 6.6. Трехуровневая модель приложения уровня предприятия

Как видно из рисунка, на первом уровне представлены разнообразные клиенты приложения, на втором — сервер, посредством которого клиенты взаимодействуют с разного рода компонентами, находящимися на третьем уровне.

Второй уровень рассматриваемой модели часто называют средним уровнем (middle tier), а сервер, его представляющий, соответственно, среднеуровневым сервером (Middle-Tier Server). Для создания такого сервера, как правило, и используется J2EE, хотя этот продукт может быть применен для создания как клиента (редко), так и организации хранилища информации.

J2EE служит для создания трехуровневых приложений в сети Web, т. е. выступает в качестве инструментария для построения среднего уровня, в то время как первый уровень представлен браузером, а третий — базами данных, доступ к которым осуществляется посредством моста JDBC-ODBC. Сказанное проиллюстрировано на рис. 6.7.

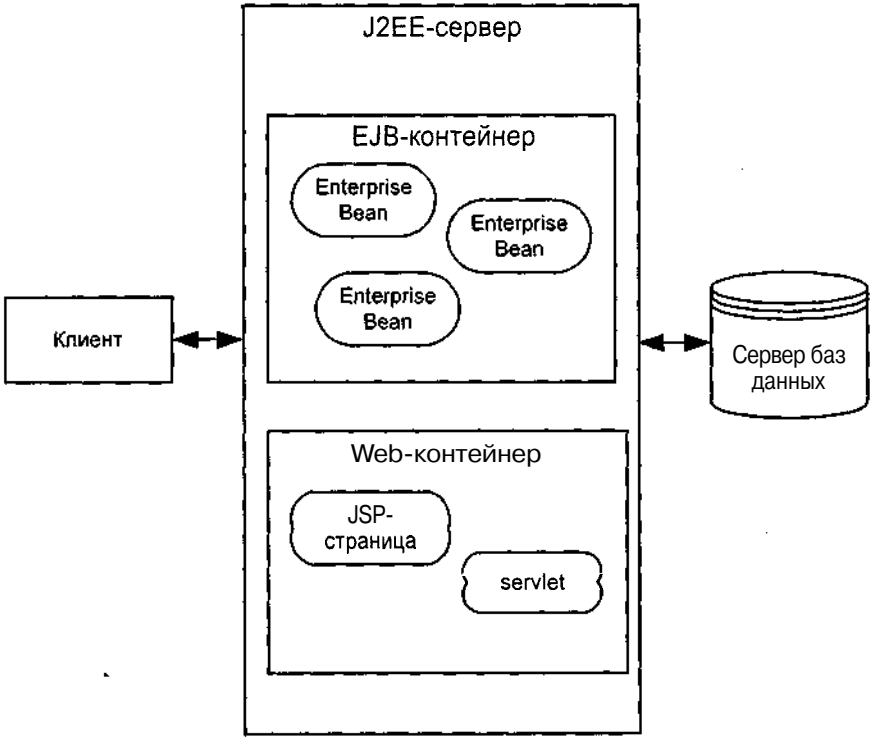


Рис. 6.7. Структура приложения на базе J2EE

Здесь, как мы видим, сервер J2EE включает в себя два компонента: Web-контейнер и EJB-контейнер. Первый состоит из сервлет-контейнера и JSP-контейнера, о которых мы говорили ранее. При помощи этого компонента осуществляется взаимодействие между браузером, пославшим HTTP-запрос, и вторым компонентом — EJB-контейнером. Скажем о нем несколько слов.

Аббревиатура EJB расшифровывается как Enterprise Java Beans. Перевод этого выражения на русский язык представляется весьма трудным делом.

Поэтому мы дадим этой аббревиатуре следующее определение: EJB представляют собой Web-компоненты (естественно, написанные на Java), выполняющиеся на сервере и представляющие собой реализацию различных сервисов системного уровня. Контейнер EJB отвечает за выполнение следующих функций:

- управление транзакциями;
- обеспечение безопасности;
- обеспечение соединений с удаленными клиентами;
- управление жизненным циклом EJB;
- организация пула баз данных.

Здесь под управлением жизненным циклом EJB понимается следующее. EJB делятся на два типа: session EJB, которые создаются только при соответствующем запросе клиента (причем session EJB может иметь только одного клиента) и уничтожаются, когда клиент закрывает сессию. Второй тип называется entity EJB. EJB этого типа продолжают существовать даже при прекращении работы EJB-контейнера и могут иметь несколько клиентов. При сравнении этих двух типов напрашивается аналогия между оперативной и постоянной памятью. Так как различные EJB могут существовать (жить) разное время, то возникает вопрос, сколько времени должен жить каждый из них. Работа контейнера, связанная с определением срока жизни каждого EJB, и называется управлением жизненного цикла EJB.

Помимо EJB- и Web-контейнеров J2EE содержит также расширения некоторых стандартных механизмов, которые входят в состав J2SE. В частности. значительно расширены механизм обеспечения безопасности и интерфейс доступа к базам данных (JDBC). Проанализировав инструментарий, предлагаемый J2EE, можно прийти к выводу, что при построении Web-интерфейса с акцентом на язык XML, а это основная задача книги, использование данного продукта себя не оправдывает, т. к. из всей функциональности J2EE нам понадобится лишь Web-контейнер.

Более того, нам пригодится только та его часть, которая позволяет применять сервлеты, т. е. сервлет-контейнер. Он понадобится нам для двух целей: запустить Cocoon, который является сервлетом, а также обеспечить возможность вставки Java-кода в XML-документы, как мы это уже демонстрировали в *главе 4.* Фактически XML-документ, содержащий Java-код, с помощью Cocoon можно превратить в сервлет как в одну из промежуточных стадий серверного преобразования XML-документа в HTML-документ. Но к этому вопросу мы вернемся чуть позднее.

Сервлет-контейнер может быть реализован одним из трех способов: как некая встроенная часть Web-сервера, как дополнение к Web-серверу или как автономное приложение. Существует множество как коммерческих, так и

бесплатных продуктов, обеспечивающих эту функциональность. Заметим, что, как правило, наряду с сервлет-контейнером эти продукты предоставляют также и JSP-контейнер, хотя последний нам и не понадобится. Заметим также, что приложения, реализующие сервлет-контейнер, часто называют сервлет-движками (servlet engine). Перечислим некоторые из них.

- Apache Tomcat;
- BEA WebLogic;
- Dynamo;
- IBM WebSphere;
- Inprise Application Server;
- iPlanet Web Server.

Для своих разработок мы воспользуемся первым из них. Apache Tomcat (версии 3.2.4) представляет собой сервлет- и JSP-контейнер. Он может быть использован и как автономное приложение и в связке с некоторыми популярными Web-серверами. Этими серверами являются:

- Apache версии 1.3 и выше;
- Microsoft Internet Information Server, версии 4.0 и выше;
- Microsoft Personal Web Server, версии 4.0 и выше;
- Netscape Enterprise Server, версии 3.0 и выше.

Так как эта книга посвящена языку XML, мы не будем здесь обсуждать особенности тонкой настройки Apache Tomcat, а также способы его инсталляции. Также не имеет значения, будет ли он работать как автономное приложение или в связке с Web-сервером — в любом случае наши цели будут достигнуты.

Заканчивая раздел, отметим, что в наши задачи будет входить только установка этого продукта, а также минимальное его конфигурирование с целью обеспечения необходимых условий для применения средств, предлагаемых вышеописанными технологиями.

6.5. Распределение задач между сервером и клиентом

Итак, мы рассмотрели те функции, которые должны выполнять различные слои клиент-серверного приложения. Или другими словами, слои, на которые разбивается документ при его формировании, причем каждый слой соответствует отдельной группе разработчиков, обладающей знаниями в соответствующей узкой области. Напомним, что при нашем подходе этими слоями являются:

- Слой контента. В обязанности рабочей группы, отвечающей за этот слой, входит информационное наполнение документа, т. е. простой набор текста.
- Слой структуры информации. Отвечает за создание такой структуры информации, при которой обеспечивается уникальная идентификация блоков текста, в зависимости от их смысловой нагрузки. Иными словами, создаются условия для эффективного манипулирования и навигации по всем имеющимся данным.
- Слой представления информации. Его можно условно разбить на четыре подуровня.
  - Слой, отвечающий за графическое и звуковое сопровождение документа. Над этим слоем трудятся художники, создающие статические и динамические изображения, отдельные символы и шрифты целиком, а также композиторы, предоставляющие как специфические звуки, так и музыкальные произведения.
  - Дизайнерский слой. Можно сказать, что на этом подслое происходит верстка документа. Определяется взаиморасположение объектов в пространстве (на Web-странице) и времени (сопоставление звуковых эффектов и им соответствующих событий), созданных на предыдущем подслое, а также оформление различных блоков текста.
  - Слой, отвечающий за взаимодействие пользователя и документа. Создатели этого слоя обеспечивают пользовательский интерфейс для манипулирования контентом и структурой (очень редко) документа.
  - Логический слой. Базируясь на структуре информации, отвечает за выполнение над ней операций манипулирования и навигации.

В предыдущем разделе мы рассмотрели трехуровневую модель клиент-серверного приложения. Здесь же мы сопоставим программные компоненты как клиента, так и сервера с вышеописанными слоями архитектуры создания и управления документами.

При этом мы не ставим перед собой цель полностью описать философию построения документооборота. Совокупность различных программных средств и технологий, описанных ниже, представляет собой только нашу точку зрения на тот инструментарий, который может быть использован для практического применения языка XML в рамках демонстрационного приложения.

Первый слой разработки документа представляет собой накопление информации, которое должно проходить с учетом ее последующей структуризации. Если в качестве структуры мы выберем, например, реляционную модель данных, то для ввода информации нам предварительно потребуется обеспечить интерфейс доступа к базам данных. Вообще говоря, перед вво-

ом информации необходимо организовать доступ к ее хранилищу, но т. к. мы выбираем в качестве этого хранилища XML-файлы, то для этих целей нам понадобится лишь обычный текстовый редактор.

Заметим здесь, что последовательная реализация каждого слоя документа невозможна. Поэтому далее мы будем предполагать, что разработка всех слоев происходит параллельно, т. е. каждый слой разрабатывается с учетом специфики остальных. Например, информационное наполнение осуществляется с учетом структуры вводимой информации.

реализация слоя структуры заключается в создании словаря имен элементов, а также других конструкций языка XML, которые затем вставляются в документ, тем самым образуя его структуру. Практически это означает, что на данном этапе мы (вернее, соответствующая рабочая группа) взаимодействуем только с XML-файлом. Обработка сценария XSLT входит в сферу следующего слоя — представления информации.

На этом слое происходит формирование внешнего вида документа и его аудио-сопровождения. При этом в основном используется язык HTML, а именно на дизайнерском подслое используются элементы этого языка, а также каскадные стилевые таблицы CSS, на следующем — главная роль во взаимодействии пользователя и документа отводится языку JavaScript. Здесь довольно трудно разделить реализацию каждого подслоя между отдельными рабочими группами. Тем не менее, заметим, что при большом количестве описаний стилей CSS их можно вынести в отдельный файл, то же самое можно проделать и с кодом JavaScript. Ну и разумеется, графические, звуковые и им подобные объекты разрабатываются и хранятся отдельно, в соответствии со своей спецификой.

Последний по списку — логический — слой отвечает за реализацию всех операций над хранилищем информации, а также за решение некоторых внутренних задач приложения, например, за обработку параметров HTTP-запроса, установку дополнительных соединений или какие-либо вычисления. Главная задача, решаемая при разработке этого слоя, заключается в том, чтобы обеспечить возможность программным средствам взаимодействовать с применяемыми форматами данных.

Общая схема работы Web-приложения, состоящего из вышеописанных компонентов, выглядит примерно так. По запросу клиента на стороне сервера Происходит преобразование соответствующих XML- и XSL-файлов, которые возможно содержат Java-код. Как результат этого преобразования клиент получает совокупность файлов HTML и, возможно, отдельных файлов CSS, JavaScript, Java-апплетов, а также графических и звуковых объектов. Далее на стороне клиента в результате взаимодействия пользователя с полученной Web-страницей инициируется следующий запрос серверу, после чего весь вышеописанный процесс повторяется. Напомним, что это взаимодействие осуществляется с помощью браузера Microsoft Internet Explorer версии 5.5.

Теперь попытаемся сопоставить уровни разработки приложения и используемые нами программные компоненты. При установке только одного Web-сервера, в нашем случае Apache, у нас будет реализован лишь слой представления информации, т. е. в нашем распоряжении будут возможности, предлагаемые языком HTML. Хотя с помощью соответствующих модулей этого Web-сервера (например, модуля подключения PHP) можно реализовать и логический слой, мы, как уже говорили ранее, не станем этого делать.

Возможность реализовать этот слой мы получим далее при установке на сервер виртуальной машины Java. Но эта возможность будет выглядеть весьма бедно, т. к. мы не собираемся разрабатывать больших и тем более автономных приложений на этом языке, в нашем случае он играет вспомогательную роль. Поэтому с установкой J2SE мы получим в свое распоряжение только апплеты, с помощью которых клиент сможет осуществлять ограниченный круг действий по взаимодействию с сервером и хранящейся на нем информацией.

Для обеспечения полной функциональности логического слоя нам понадобится Apache Tomcat. Возможность работы с небольшими серверными Java-приложениями (сервлетами), а также встраивания их в документ (через механизм JSP) обеспечивает нам простоту разработки, относительно невысокую загруженность сервера, а также решение большинства встающих перед нами задач. Заметим, что Tomcat может работать и автономно, без участия Web-сервера, выполняя функции последнего. Но, тем не менее, мы будем использовать оба продукта, т. к. сервер может обслуживать не только какие-либо Web-приложения, но и предоставлять доступ к статическим Web-страницам, служащим только для предоставления информации. Кроме того, Apache предоставляет несколько большую свободу в конфигурировании серверной части информационной системы, чем та часть Tomcat, которая выполняет функции Web-сервера.

И наконец, последним программным компонентом, которым мы воспользуемся, будет инструментарий публикаций Cocoon, отвечающий за структуру данных, находящихся в информационном хранилище. Мы уже достаточно говорили о возможностях языка XML в этой области, и поэтому нет нужды возвращаться здесь к этому вопросу. Вместо этого в следующем разделе мы подробно опишем сам этот продукт.

6.6. Инструментарий публикаций Cocoon

Cocoon представляет собой мощное, кроссплатформенное (полностью написанное на Java) средство для построения Web-приложений, основой которых является язык XML. Помимо поддержки основной совокупности технологий, объединенных этой аббревиатурой, он также предоставляет широ-

кие возможности для интеграции с различными альтернативными инструментами обработки данных.

Cocoon реализует трехслойную модель разработки Web-приложений. Эта модель подразумевает отдельную разработку структуры документа (XML-составляющая), представления данных (XSL-составляющая) и логики (язык программирования Java). Дополнительную гибкость этому методу придает использование дополнительных программных продуктов — сервлет- и JSP-контейнера Tomcat, а также Web-сервера Apache.

Можно было бы ожидать, что пользователям Cocoon приходится разрабатывать не только три вышеперечисленных слоя Web-приложения, но и механизм интеграции этих слоев, т. е. саму модель. На самом деле этого не происходит, т. к. управление небольшим количеством технологий разработки документов, а также способов взаимодействия документов между собой может быть легко реализовано внутри самого парсера.

Но это касается только используемой нами версии Cocoon 1.8.2. На момент написания книги последней версией является 2.0.1. Можно сказать, что продукты этих версий отличаются друг от друга, как калькулятор от компьютера. Разработчики последней версии Cocoon уделили внимание великому множеству всевозможных действий, которые могут быть проделаны как над содержимым какого-либо конкретного документа, так и над их совокупностью. Иными словами, в этом парсере заложены средства не только разработки документа, но и хранилища этих документов, причем эти средства настолько разнообразны, что сами нуждаются в структурировании и представлении.

В частности, для разработки даже небольшого документа с помощью Cocoon 2.0.1 "с нуля", разработчик должен будет создать также файл с расширением xtap, который служит для описания всех этих средств. А для того чтобы описать каждое из них или даже исключить его применение, необходимо иметь представление о том, для чего это средство предназначено и как работает. Учитывая то, что подобных механизмов рассматриваемый продукт содержит несколько десятков, можно сказать, что хотя Cocoon 2.0.1 и является гораздо более мощным средством разработки, но тем не менее, использование его начинающими Web-мастерами представляется весьма затруднительным, т. к. требует изучения множества дополнительных технологий, а также синтаксиса их описания для своего приложения.

Так как большинство из предлагаемых технологий, поддерживаемых Cocoon 2.0.1, далеки от "чистого" XML, а также из-за большого объема работ, предшествующих непосредственной разработке приложения, мы не будем использовать этот продукт. Кроме того, для иллюстрации возможностей XML и других вышеописанных средств Web-публикаций функциональности Cocoon 1.8.2 будет вполне достаточно.

Обратимся к документации, входящей в дистрибутив данного продукта, и рассмотрим, как внутри него происходит формирование ответа на запрос

клиента. В основе системы публикаций Cocoon лежит механизм, базой которого является т. н. *реактор* (reactor). На рис. 6.8 приведено графическое представление этого механизма.

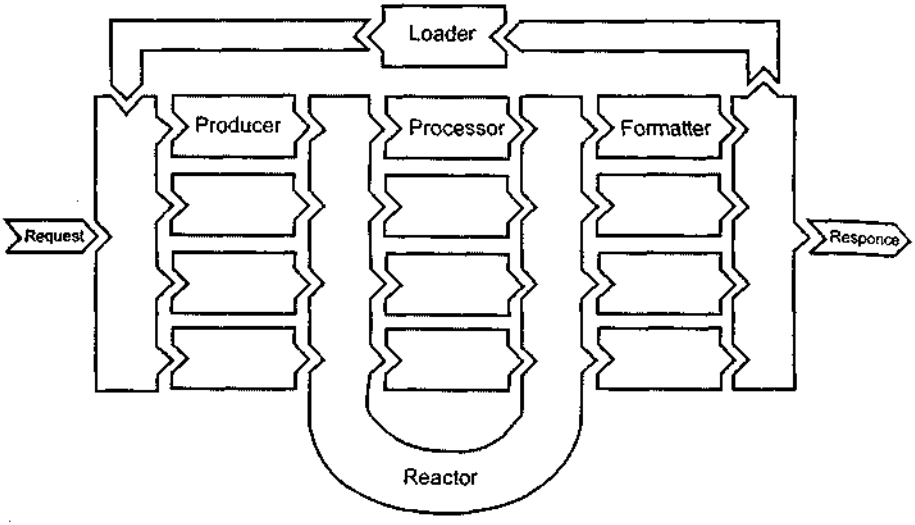


Рис. 6.8. Схема работы Cocoon: взгляд изнутри

Рассмотрим компоненты схемы, представленные на рис. 6.8.

- Request (запрос) — обеспечивает информацию о запросе клиента, необходимую для его обработки процессором. Запрос должен сообщать о том, кто из клиентов сгенерировал запрос, какой URI запрашивается и какой из генераторов должен обрабатывать этот запрос.
- Producer (генератор) — обрабатывает запрошенный URI и генерирует XML-документ. Так как генератор является встраиваемым модулем, он работает как подсервлет (subservlet) среди других компонентов Cocoon, что позволяет пользователям применять свои собственные генераторы. Генератор отвечает за создание XML-документа, который далее передается на обработку реактору. Реализация генератора должна определять функцию, которая генерирует документ, опираясь на объект, представляющий запрос.
- LI Reactor (реактор) — отвечает за выбор процессора, который должен работать над документом, в зависимости от инструкций процессору (PI — Processing Instruction). Такое название компонента обуславливается тем, что он "реагирует" на PI, встретившиеся в документе. Схема реактора отличается от механизма последовательного применения каждого имеющегося в наличии процессора. Она позволяет динамически конфигурировать способ обработки документа, что увеличивает производи-

тельность, т. к. используются только необходимые процессоры. Реактор отвечает также за перенаправление документа надлежащему форматеру.

Processor (процессор) — служит для специфической обработки XML-кода. Специфика обработки указывается с помощью PI. Дистрибутив Cocoon 1.8.2 содержит следующие процессоры:

- LDAP (Light weight Directory Access Protocol) Processor;
- ESQL (Extended SQL) Processor;
- XSP (extensible Server Pages) Processor;
- XInclude Processor (довольно слабо реализует соответствующий стандарт W3C);
- XSLT Processor.

I Formatter (форматер) — трансформирует содержащееся в памяти представление XML-документа в поток (stream), который далее интерпретируется клиентом, сделавшим запрос. После того как документ покидает реактор, он форматируется в зависимости от PI для своего потребителя. MIME-тип сгенерированного документа зависит от реализации применяемого форматера.

I Response (ответ) — инкапсулирует документ вместе с его свойствами (такими, например, как длина, MIME-тип и т. п.).

I Loader (загрузчик) — отвечает за загрузку отформатированного документа в том случае, когда этот документ представляет собой исполняемый код. Этот компонент используется для откомпилированных серверных страниц (XSP), при этом ранее разделенные логика и контент смешиваются и компилируются, и результат этой компиляции представляет собой генератор (producer). Иными словами, когда выводом форматера является исполняемый код, он не посылается непосредственно клиенту, а вместо этого он загружается и выполняется как генератор.

Как уже говорилось ранее, пользователь Cocoon 1.8.2 не должен заботиться о конфигурировании вышеописанной модели. Этот процесс автоматизирован и выполняется парсером самостоятельно. Единственным исключением является необходимость использования PI для указания тех процессоров и форматеров, которые должны быть применены при обработке документа. Этими инструкциями процессору являются:

- `<?cocoon-process type="xxx"?>`

и

`<?cocoon-format type="yyy"?>`

Мы уже указали те типы процессоров, которые реализованы в Cocoon. Далее приведем список форматеров.

- HTML Formatter;
- XHTML Formatter;
- Text Formatter (имеется в виду плоский текст — plain text);
- XML Formatter;
- FO2PDF Formatter.

Каждый из этих форматеров отвечает за формат конечного документа. Этими форматами соответственно являются HTML, XHTML, текст, XML и PDF. Последний может быть просмотрен с помощью Acrobat Viewer/Reader.

Для того чтобы документ был корректно обработан Cocoon, он должен содержать объявление XML:

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

что полностью согласуется со спецификацией W3C. Кроме заголовка документ должен содержать как минимум корневой элемент. Все остальные конструкции для формирования документа являются опциональными.

После объявления XML может следовать одна или несколько конструкций, обеспечивающих подключение к документу сценариев XSLT. Для этих целей служит инструкция обработки следующего формата:

```
<?xml-stylesheet href = "файл сценария XSLT"
                 media = "media-тип клиента"?>
```

Необязательный атрибут media содержит информацию о том, каким media-типом клиента будет прочитан конечный документ. Его значениями могут быть:

- explorer — браузер Microsoft Internet Explorer;
- opera — браузер Opera;
- lynx — текстовый браузер Lynx;
- java — любой Java-код, использующий стандартные классы URL;
- wap — браузер Nokia WAP Toolkit;
- netscape — один из браузеров Netscape Navigator или Mozilla.

Кроме этой PI в документе могут также находиться и те, которые служат для указания процессоров и форматеров. Здесь мы опишем только использование XSP-процессора, отвечающего за логический слой документа.

Синтаксис статических и серверных или динамических страниц должен быть одинаков. Это значит, что динамически генерируемый документ должен содержать объявление XML и как минимум корневой элемент. Мы будем использовать XSP-процессор для вставок Java-кода в XML-составляющую документа, хотя существует возможность вставлять этот код

и в сценарии XSLT, но такое его нестандартное применение редко себя оправдывает.

Для того чтобы документ был обработан XSP-процессором, он должен содержать соответствующую инструкцию обработки:

```
<?cocoon-process type = "xsp"?>
```

Следующим необходимым условием корректности XSP-страницы является наличие в документе корневого элемента с именем xsp:page:

```
<xsp:page language = "java"
          xmlns:xsp = "http://www.apache.org/1999/XSP/Core">

...
</xsp:page>
```

Обязательный атрибут xmlns:xsp отвечает за декларацию пространства имен с префиксом xsp. Этому пространству принадлежат все элементы, которые обеспечивают функциональность XSP-процессора. Далее приведем их полный список:

- xsp:page — корневой элемент всех XSP-страниц, определяющий язык программирования, на котором выполнены вставки программного кода в документ, а также пространство имен xsp;
- xsp:structure — элемент верхнего уровня, служащий для указания блока, в котором будут перечислены внешние импортируемые модули;
- xsp:include — применяется в контенте элемента xsp:structure для указания одного внешнего импортируемого модуля способом, зависящим от синтаксиса выбранного языка программирования;
- xsp:logic — контентом этого элемента является вставляемый в документ программный код;
- xsp:content — служит для вставки XML-кода в программный код вставки. Используется только в контенте элемента xsp:logic;
- xsp:expr — генерирует текстовый узел, значением которого является результат вычисления выражения, представляющего собой программный код, являющийся контентом данного элемента. Используется вне контента xsp:logic, если же есть необходимость применять xsp:expr в контенте xsp:logic, то xsp:expr должен быть заключен в контент элемента xsp:content;
- xsp:element — служит для динамического создания узла-элемента в исходном дереве. Так же, как xsp:expr, используется вне xsp: logic;
- xsp:attribute — служит для динамического создания узла-атрибута в исходном дереве. Так же, как xsp:expr, используется вне xsp: logic;

- xsp:pi — применяется для динамического создания узла-Pi в исходном дереве. Так же, как xsp:expr, используется вне xsp:logic;
- xsp:comment — служит для динамического создания узла-комментария в исходном дереве. Так же, как xsp:expr, используется вне xsp:logic

Ранее в своих примерах мы работали почти со всеми этими элементами. Обратим внимание на следующие два — xsp:structure и xsp:include. Принцип их применения довольно прост. Так как вставки программного кода мы пишем на языке Java, то эти элементы будут служить для импорта некоторых пакетов, входящих в дистрибутив J2SE1.3. Это может выглядеть примерно так:

```
<xsp:structure>
  <xsp:include>java.net.*</xsp:include>
  <xsp:include>my_utils.*</xsp:include>
</xsp:structure>
```

Далее представим список тех пакетов Java, которые импортируются по умолчанию, т. е. не нуждаются в явном импортировании с помощью xsp:structure И xsp:include:

- java.io.*
- java.util.*
- org.w3c.dom.*
- org.xml.sax.*
- javax.servlet.*
- javax.servlet.http.*
- org.apache.cocoon.parser.*
- org.apache.cocoon.producer.*
- org.apache.cocoon.framework.*
- org.apache.cocoon.processor.xsp.*

Помимо XSP-процессора в наших разработках нам понадобится библиотека тегов (taglib), называющаяся FP (Form Processing — обработка форм). Как следует из этого названия, библиотека служит для обработки данных, полученных в результате взаимодействия пользователя с HTML-формой. Главной особенностью предоставленной в этой библиотеке является возможность чтения и записи во внешний XML-файл непосредственно из текущего документа. Поэтому мы будем ее рассматривать не как средство обработки форм, а как некий примитивный язык запросов к данным в формате XML, уделяя внимание главным образом возможности записи данных в XML-файл.

Для того чтобы можно было работать с библиотекой тегов FP, необходимо объявить пространство имен с префиксом fp. Это объявление осуществля-

ется точно так же, как и при применении XSP-процессора. И точно так же, пространству имен fp будут принадлежать все элементы данной библиотеки. Как правило, эта библиотека используется вместе с XSP-процессором, поэтому чаще всего соответствующие пространства имен объявляются с помощью элемента xsp:page:

```
<xsp:page language = "java"
        xmlns:xsp = "http://www.apache.org/1999/XSP/Core"
        xmlns:fp = "http://apache.org/cocoon/XSP/FP/1.0">
...
</xsp:page>
```

рассмотрим здесь те элементы пространства имен fp, с помощью которых производятся операции чтения и записи над XML-файлом. Описание остальных элементов данного пространства имен можно найти в документации, входящей в дистрибутив Cocoon 1.8.2, здесь мы его приводить не будем, т. к. далее рассмотрим более эффективные средства обработки HTML-форм.

Перед тем, как проводить операции чтения и записи необходимо указать ют файл, над которым они будут осуществляться. В этом контексте данный файл называется ресурсом, мы же будем его иногда называть целевым. Для этих целей используются четыре элемента:

- fp:resource — служит контейнером для следующих трех элементов. Имеет атрибут id, с помощью которого ресурс уникально идентифицируется, в том случае, когда этих ресурсов несколько;
- fp: resource-file — служит для указания пути к целевому файлу;
- fp:resource-node — с помощью XPath адресует целевой узел в целевом файле;
- fp:default-mode — служит для указания режима записи в целевой файл. Может принимать одно из трех значений:
  - replace — в этом режиме целевой узел будет заменен на записываемый;
  - insert-before — в этом режиме будет создан новый эквивалентный записываемому узел, который будет вставлен перед целевым;
  - insert-after — в этом режиме будет создан новый эквивалентный записываемому узел, который будет вставлен после целевого.

Элементы fp:resource-file И fp:resource-node ЯВЛЯЮТСЯ обязательными, т. е. должны быть частью контента fp:resource. Элемент fp:default-mode не обязателен, его значением по умолчанию является replace. Рассмотрим пример.

```
<fp:resource id = "item1">
  <fp:resource-file>items.xml</fp:resource-file>
  <fp:resource-node>items/item[position()=1]</fp:resource-node>
  <fp:default-mode>replace</fp:default-mode>
</fp:resource>
```

Как видно, здесь мы собираемся работать с файлом items.xml, целевым узлом items/item[position()=1], при этом если мы будем что-либо записывать в этот файл, то тот узел, в который мы будем записывать данные, будет замещен записываемым. Отметим здесь два момента. Во-первых, в силу особенностей описываемой библиотеки тегов элемент items в нашем случае необходимо будет сделать непосредственным потомком не корня, а корневого элемента документа, представленного файлом items.xml. Во-вторых, путь к целевому файлу может быть записан с учетом синтаксиса, характерного для используемой операционной системы. В Windows примером такого пути может быть c:\work\items.xml, а в UNIX-системах — /work/items.xml.

Для чтения из целевого файла используется элемент

```
<fp:read
   from = "id ресурса"
   select = "целевой узел"
   as = "способ чтения"/>
```

Здесь значением атрибута from является уникальный идентификатор ресурса, атрибута select — выражение XPath, адресующее тот узел, из которого должно быть произведено чтение относительно целевого узла, указанного В fp:resource-node, И as — ОДНО ИЗ значений string ИЛИ node. Если значением as является node, то прочитанная информация будет представлять собой текст с разметкой XML, причем к этой разметке могут быть применены соответствующие правила шаблона. Если же значением атрибута as является string, то будет прочитано строковое значение целевого узла так, как при вызове функции string(), аргументом которой является адрес данного целевого узла.

Атрибуты select и from являются обязательными, а as — нет, его значением по умолчанию является string. Контент элемента fp:read должен быть пустым. Значением атрибута select может быть выражение, адресующее *несколько* узлов.

Соответственно, для записи данных в целевой файл используется элемент

```
<fp:write
   to = "id ресурса"
   select = "целевой узел"
   as = "способ записи"/>
```

Здесь атрибут to содержит уникальный идентификатор ресурса, в который должна производиться запись. Атрибуты select и as эквивалентны одноименным атрибутам `fp:read`. Также атрибуты to и select обязательны, as — нет, и его значение по умолчанию — string. Контент `fp:write` может содержать элементы, принадлежащие пространству имен fp. Если этими элементами являются `fp:read`, то прочитанные данные будут записаны в целевой файл. Значением атрибута select должно быть выражение, адресующее единственный узел, в противном случае будет либо сгенерирована ошибка, либо возникнет клинч, т. е. парсер будет теоретически бесконечно долго обрабатывать такую конструкцию.

После того как документ, содержащий `fp:read` и/или `fp:write`, будет обработан, эти элементы будут заменены результатом проведенной операции, т. е. чтения или записи. Только после этого документ будет обработан процессором XSLT.

Описанием двух механизмов — XSP-процессора и библиотеки тегов FP — мы и закончим эту главу. О поддержке других технологий, реализованных в Cocoon 1.8.2, можно прочитать в документации, поставляемой вместе с дистрибутивом этого парсера.

ГЛАВА 7

Каскадные стилевые таблицы

В этой главе мы опишем некоторые средства, в совокупности с которыми язык XML образует полноценную среду разработки Web-приложений. Здесь мы кратко опишем способы применения каскадных стилевых таблиц, обработку HTML-форм, а также некоторые полезные приемы использования Java как со стороны сервера, так и со стороны клиента.

7.1. Что такое CSS?

Первый стандарт каскадных стилевых таблиц (CSS — Cascading Style Sheets) был предложен консорциумом W3C в 1996 году. Его целью было создание простого языка описания способов оформления (стилей) документов, использующих разметку. На данный момент законченным является стандарт CSS 2, вышедший в 1998 году, и ведется разработка CSS 3.

Несмотря на то, что каскадные стилевые таблицы являются мощным средством разработки дизайна документа, их применение тормозится производителями Web-клиентов, которые не спешат встраивать в свои продукты поддержку последних стандартов. Поэтому в настоящее время можно говорить о полной поддержке некоторыми браузерами только самого первого стандарта — CSS 1, более поздние версии поддерживаются лишь частично.

Для нас это не будет большой проблемой, т. к. в задачи этой книги входит, прежде всего, описание технологии XML, а оформление документов является второстепенной задачей. Поэтому, кратко описав основные конструкции и принципы использования CSS, будем считать, что необходимый объем знаний о каскадных стилевых таблицах мы предоставили.

7.2. Для чего используются CSS?

После того, как закончено информационное наполнение документа, с помощью XML и XSLT задана и описана его структура, начинается разработка его дизайна. В это понятие входит оформление блоков текста, их взаиморасположение, а также подбор иллюстраций и звукового сопровождения документа. Как уже говорилось ранее, описание внешнего вида документа может быть вынесено в отдельный файл или набор файлов. При этом стиль каждого компонента, а также документа в целом, привязывается к этому компоненту с помощью соответствующего идентификатора стиля. Рассмотрим преимущества этого механизма.

В том случае, когда дизайн документа описан отдельно от самого документа, во-первых, обеспечивается модульная разработка всех аспектов внешнего вида документа в зависимости от устройства, на котором будет просматриваться этот документ. Это означает, что, создав несколько стилей CSS для различных устройств, эти стили можно подключать к документу, опираясь на информацию о типе данного устройства.

Кроме того, связывание стиля CSS и документа может происходить еще и в зависимости от пожеланий пользователя, для которого этот документ предназначен. Ведь оформление документа тесным образом зависит от национальных особенностей региона, в котором находится пользователь, от того, насколько детальную информацию хочет получить этот пользователь, от его способности полноценно воспринимать ту или иную форму подачи этой информации и от много другого. С учетом всех вышеперечисленных факторов можно создать набор стилей CSS, который будет удовлетворять самым разным требованиям, возникающим в связи со стремлением максимально расширить целевую группу пользователей.

Во-вторых, стили, однажды описанные для таких элементов текста, как определения, сноски, примечания, названия глав, частей, разделов и т. п., могут быть использованы многократно как внутри одного документа, так и для их совокупности. Таким образом, если разработать хорошо структурированный набор файлов, содержащих описания стилей CSS, то сопровождение всего объема документов значительно упрощается, т. к. отпадает необходимость переписывать для однотипных компонентов документа код, служащий для их представления. И кроме того, для изменения дизайна какого-либо одного компонента достаточно внести исправления только в одно описание стиля, которое к тому же легко найти в силу структурированности всего набора стилей.

Немаловажным достоинством языка CSS является еще и то, что он обладает единым стандартом, а это позволяет пользователям различных браузеров более менее одинаково просматривать один и тот же документ. С другой стороны, разработчики документов, благодаря CSS, могут отойти от использования разметки в целях придания документу должного внешнего вида.

Широко известным примером этого является применение таблиц (тег <table>) в языке HTML для обеспечения строго определенного взаиморасположения компонентов Web-страницы. Недостаток этого метода — необходимость полной загрузки страницы браузером для вычисления размеров каждой ячейки таблицы. А если таблица служит для форматирования всей Web-страницы целиком, то пользователю придется ждать ее полной загрузки вместе со всеми картинками, кнопками и т. п. При большой насыщенности Web-страницы нетекстовыми объектами этот процесс может занимать весьма долгое время.

В начале главы мы сказали, что язык CSS является простым. На самом деле было бы более точным назвать синтаксис этого языка примитивным. Из всех инструментов построения Web-приложений, описанных в этой книге, язык CSS наиболее легок в изучении. Далее следует весьма короткий список правил синтаксиса этого языка, а также описание его некоторых наиболее часто используемых дефиниций.

7.3. Основы синтаксиса CSS

Язык описания стилей не содержит таких относительно сложных конструкций, как операторы условного или безусловного перехода, циклов, а также любых других, присущих даже самому простому языку программирования. В нем, главным образом, используется только одна конструкция, представляющая собой описание стиля, привязанного к какому-либо объекту, который должен в соответствии с этим стилем отображаться.

Эта конструкция называется правилом. Правило состоит из двух частей: селектора (selector), служащего для описания объектов, к которым будет привязан описываемый стиль, и декларации (declaration), представляющей собой само описание стиля. Декларация, в свою очередь, состоит из списка различных свойств (property) отображения объекта и их значений. Далее приведем синтаксис правила:

```
object {
property1 : value1;
property2 : value2;
...
propertyN : valueN;
}
```

Здесь object является селектором, а то, что находится в фигурных скобках, — декларацией, которая содержит пары свойство — значение, отделенные друг от друга точкой с запятой (;). Рассмотрим пример:

```
H1 { color : red;
     text-align : center; }
```

Это правило говорит о том, что заголовок первого уровня, представленный тегом H1, должен иметь красный цвет шрифта и быть отцентрированным по горизонтали. Селектор может состоять не только из имени одного объекта. Но прежде, чем переходить к его синтаксису, рассмотрим способы привязки стилей к документу.

7.4. Привязка таблиц CSS к документу

Как мы уже говорили ранее, стиль CSS может находиться во внешнем по отношению к документу файле. Для того чтобы подключить такой стиль, в языке HTML служит элемент

```
<link
   rel= "stylesheet"
   type = "text/ess"
   href = "URL файла CSS"/>
```

Атрибут rel, представляющий собой сокращение от relationship (отношение), устанавливает тип отношения между объектом link и привязываемым объектом. В данном случае значение этого атрибута — stylesheet — говорит о том, что с помощью link реализуется привязка стилевой таблицы. Значением атрибута type является MIME-тип привязываемого файла, в данном случае значение text/ess говорит о том, что этот файл содержит описания стилей CSS в текстовом формате. И, наконец, последний атрибут содержит относительный или абсолютный URL файла описаний стилей.

Элемент link может находиться только в контенте элемента head.

Примечание

Все элементы и атрибуты языка HTML здесь и далее описаны так, как они поддерживаются браузером Microsoft Internet Explorer 5.5, что может незначительно расходиться тем, как они определены соответствующей спецификацией W3C. Кроме того, рассматривая какой-либо элемент языка HTML, мы будем описывать только те его атрибуты, которые представляют для нас интерес в соответствующем контексте.

Для подключения внешней таблицы стилей в языке XML используется инструкция обработки

```
<?xml:stylesheet type = "text/ess" href = "URL файла CSS"?>
```

Но о том, как применять CSS вместе с XML-документами без посредничества HTML, мы поговорим чуть позже.

Еще одним способом связывания таблиц CSS и HTML-документа является непосредственная вставка этой таблицы в его заголовок. Для этих целей в языке HTML определен элемент style, который может находиться только в контенте элемента head. Рассмотрим пример:

```
<html>
  <head>
    <style>
      h1 { color : red; text-align : center }
      h2 ( color : blue }
    </style>
  </head>
  <body>
    <h1> Заголовок </h1>
    Некоторый текст
  </body>
</html>
```

Элемент style имеет два атрибута. Первый — media — указывает, для какого устройства предназначен данный стиль оформления. Он может принимать три значения:

- screen — оформленный документ предназначен для просмотра на экране компьютера;
- print — документ предназначен для печати на принтере;
- all — значение по умолчанию. Документ может быть использован и для просмотра на экране и для печати.

Второй атрибут — type — указывает язык, на котором описаны стили CSS. Для него значением по умолчанию является text/ess.

Третий способ связывания объекта и соответствующего ему стиля заключается в использовании атрибута style, которым обладает элемент языка HTML, представляющий данный объект.

```
<h1 style="color:red; text-align:center"/>
```

Здесь имеются в виду только те элементы, которые могут быть отображены на каком-либо устройстве. Примером неотображаемого элемента может послужить param, необходимый для установки начальных значений свойств **у таких элементов, как** object, embed И applet.

И последним способом связывания таблиц CSS и HTML- документа является импортирование этих таблиц, для чего служит так называемый at-оператор @import. Этот оператор может находиться только в начале блока. образованного фигурными скобками. Блоком при этом может быть как дек-

ларация целиком, так и некоторая ее часть. Кроме того, @import должен предшествовать всем правилам, содержащимся в данном блоке. Таким образом, следующие два примера являются верными:

```
@import "my_style.css"
h1 {color:red}
```

```
@import "style1.css"
@import "style2.css"
h1 {color:red}
```

д этот — нет:

```
@import "style1.css"
h1 {color:red}
@import "style2.css"
```

Здесь второй at-оператор будет проигнорирован, т. к. ему предшествует правило для элемента h1.

7.5. Синтаксис селекторов

Теперь перейдем к рассмотрению различных форм записи селектора. Как следует из его названия, он предназначен для отбора тех объектов, к которым будет применен соответствующий стиль оформления, представленный декларацией.

Сначала рассмотрим две самые тривиальные формы. Селекторы, записанные в соответствии с этими формами, называются универсальным (universal) и типизированным (typed). Универсальный селектор записывается с использованием звездочки (*) и по аналогии с подобной конструкцией из XPath служит для отбора элементов с произвольными именами:

```
* {text-indent : 2em}
```

С помощью типизированного селектора отбираются элементы с конкретным заданным именем. Все селекторы из предыдущих примеров являются типизированными. С их помощью выполняется отбор элементов с именами h1 и h2. В том случае, когда типизированные селекторы обладают одинаковыми декларациями, их можно группировать, перечисляя их имена через запятую, после чего следует соответствующая декларация. Например, вместо

```
h1 {color : red}
h2 {color : red}
h2 (color : red}
```

можно записать

```
h1, h2, h3 (color : red}
```

Все остальные чуть более сложные формы записи селекторов можно условно разбить на три категории, в зависимости от того, по какому критерию отбираются элементы.

7.5.1. Отбор по атрибутам

По критерию, соответствующему первой категории, элементы выбираются в зависимости от того, какими атрибутами они обладают, а также в зависимости от значений этих атрибутов. Синтаксис селектора данного типа выглядит очень похоже на синтаксис предикатов языка XPath:

some_element [условие для attribute1] [условие для attribute2]...

Существует четыре способа задания условий для отбора элементов в зависимости от их атрибутов:

- [attribute] — отбирает элементы, имеющие атрибут attribute;
- [attribute = value] — отбирает элементы, имеющие атрибут attribute, значением которого является строка или идентификатор value (в качестве примера идентификатора можно привести red, соответствующий красному цвету);
- [attribute ~= value] — в том случае, когдэ value предстзвляет собой список слов, разделенных пробелами, указывает на те элементы, которые обладают атрибутом attribute, и значением этого атрибута является хотя бы одно из вышеуказанного списка слов. При этом ни одно из слов не должно сзмо содержэть пробелы;
- [attribute |= value] — этот способ отбора элементов работает точно так же, как и предыдущий, но здесь разделителем в списке слов является дефис.

Помимо этих четырех, есть еще три специфических способа отбора элементов по их атрибутам. Очень часто для связи элемента с соответствующим ему стилем оформления используется атрибут class, значением которого является имя данного стиля. Вместо полной записи some_element[class = "style_name"], возможна Сокращенная (содержащзя точку) some_element.style_name, с помощью которой будут отобраны те же самые элементы.

Для отбора элементов, обладающих специальным атрибутом id, употребляется следующая формз зписи #id_value. Тэким образом будут отобраны все те элементы, значением атрибута id которых является id_value. Заметим, что как в HTML-, тзк и в XML-документах может нзходиться только один элемент с заданным значением атрибута id, т. к. предназначением этого атрибута является уникальная идентификация элементов, поэтому по-

лезность данного способа отбора элементов весьма сомнительна. Кроме того, при обработке XML-документа т. н. непроверяющим (not validating) процессором атрибуты типа ID не будут рассмотрены вовсе.

7.5.2. Отбор по родственным связям

Селекторы второй категории служат для отбора элементов в зависимости от их родственных связей с другими элементами.

Чтобы отобрать потомков (descendant) заданного элемента, используется список типизированных или универсальных селекторов, разделенных пробелами. При этом крайний правый селектор будет отбирать потомков тех элементов, которые отбираются вторым справа селектором, второй справа — потомков элементов, отбираемых третьим справа селектором, и т. д. до крайнего левого селектора. Рассмотрим несколько примеров.

- `h1 em {...}` — будет отбирать элементы с именем em, которые являются потомками элемента ы. Далее следует код, в котором полужирным шрифтом выделены отбираемые элементы.

```
<h1>
    <em>...</em>
    <span>
        <em>...</em>
    </span>
</h1>
```

- `ы * em {...}` — будет отбирать элементы с именем em, которые являются потомками (но не непосредственными) элемента ы.

```
<h1>
    <em>...</em>
    <span>
        <em>...</em>
    </span>
</h1>
```

- `h1 div em {...}` — будет отбирать элементы с именем em, являющиеся потомками элемента div, который в свою очередь является потомком ы.

```
<h1>
    <em>...</em>
    <div>
        <em>...</em>
    </div>
```

```
        <span>
            <em>...</em>
        </span>
        <span>
            <div>
                <span>
                    <em>...</em>
                </span>
            </div>
        </span>
    </h1>
```

Для отбора элементов, являющихся непосредственными потомками (child) заданного элемента, по аналогии с отбором потомков используется список селекторов, разделителем в котором уже является знак "больше" (>). Например, с помощью следующих правил будут отбираться:

- `h1 > em {...}` — все элементы em, являющиеся непосредственными потомками элемента h1;
- `ы > * > em {...}` — все элементы em, являющиеся внуками элемента ы.

Существует также возможность выбора братьев (sibling) заданного элемента, т. е. таких элементов, которые имеют общего родителя (parent) (но не произвольного предка). Эта форма записи селекторов несколько отличается от двух предыдущих. В ней могут присутствовать только два типизированных селектора, причем элемент, отбираемый правым селектором, должен немедленно следовать за элементом, отбираемым левым. Разделителем для этих двух селекторов служит знак "плюс" (+). Таким образом, правило `span + div {...}` будет отбирать все элементы span и div, имеющие общего родителя с произвольным именем, причем div сразу следует за span.

Селекторы всех вышеописанных типов, кроме последнего, можно комбинировать друг с другом. Например, правило `div em > span p {...}` будет отбирать все элементы p, являющиеся как угодно далекими потомками элемента span, который является непосредственным потомком элемента em, который, в свою очередь, как и p, является не обязательно непосредственным потомком элемента div. К каждому селектору этого правила можно также добавить условия отбора по атрибутам.

В языке CSS имеется еще одна возможность отбора элемента по его родственным связям. С ее помощью можно выбрать первого непосредственного потомка заданного элемента. Но для описания этой возможности нам понадобится ввести понятие псевдокласса.

7.5.3. Отбор с использованием псевдоклассов

В спецификации CSS понятие псевдокласса вводится для классификации элементов по характеристикам отличным от их имени, атрибутов и взаиморасположения в документе. Здесь имеются в виду такие характеристики элементов, которые никак не связаны с древовидной моделью документа. Псевдоклассы могут быть динамическими в том смысле, что элемент может становиться или переставать быть псевдоклассом по мере взаимодействия пользователя с документом. Единственным исключением является `:first-child` — псевдокласс, смысл которого неразрывно связан с понятием дерева документа.

Некоторые псевдоклассы могут применяться совместно для одного и того же элемента, в то время как другие являются взаимоисключающими. Псевдоклассы употребляются только вместе с элементами, но не с узлами другого типа (такими, например, как текстовые узлы, комментарии и т. п.).

Псевдокласс `:first-child` служит для выбора первого непосредственного потомка того элемента, вместе с которым используется данный псевдокласс. Рассмотрим несколько примеров:

- `p:first-child {...}` — данное правило будет описывать стиль для любого элемента p, являющегося первым непосредственным потомком какого-либо элемента в документе. Эта запись будет эквивалента с `* > p:first-child {...}`;
- `div > p:first-child {...}` — отбирает элементы p, являющиеся первыми непосредственными потомками для каждого элемента div в документе.

Следующие два псевдокласса служат для описания представлений ссылок (link). Псевдокласс `:link` служит для отбора ссылок на те ресурсы, которые еще не были загружены пользовательским агентом. Проще говоря, `:link` необходим для описаний стиля ссылок на те Web-страницы, которые пользователь еще не посетил. Второй псевдокласс — `:visited`, как следует из его названия, применяется для описания ссылок на посещенные Web-страницы.

Для создания ссылки в языке HTML используется элемент A, поэтому следующие два правила, используемые для оформления HTML-документа, будут эквивалентны:

```
A:link { color : red }
:link { color : red }
```

Псевдоклассы `:link` и `:visited` являются взаимоисключающими.

Следующая группа псевдоклассов, которую мы рассмотрим, называется динамической. Ее образуют псевдоклассы `:hover`, `:active` и `:focus`. Рассмотрим их по порядку.

- :hover — служит для отбора тех элементов, на которые пользователь указал, но не активизировал. Этот псевдокласс может быть, например, применен к элементу, который генерирует некоторую область в окне браузера, причем над этой областью находится курсор мыши или подобного ей координатно-указательного устройства;
- :active — служит для отбора активизированных пользователем элементов. Таким элементом может быть тот, который отбирается с помощью :hover, но с той разницей что пользователь должен не только установить над соответствующей областью курсор мыши, но и щелкнуть по ней. При этом элемент будет активизирован в то время, когда кнопка мыши уже нажата, но еще не отпущена;
- :focus — применяется к тем элементам, которые находятся в фокусе (в результате обработки событий от клавиатуры или еще какой-либо формы ввода текста).

Динамические псевдоклассы не являются взаимоисключающими, т. е. можно использовать любую их последовательность, и даже в сочетании с :link И :visited. Правило A:focus:hover:visited { color:red } утверждает, что ссылки, находящиеся в фокусе, (над которыми находится курсор мыши), и ресурс, на который они ссылаются (он уже был затребован пользователем), будут отображаться красным цветом.

Спецификация CSS не определяет, какие из элементов могут находиться в трех вышеперечисленных состояниях (указанные, активизированные и находящиеся в фокусе), а также каким образом данные элементы входят в эти состояния или выходят из них. Различные пользовательские агенты могут разрешать эти вопросы по-разному.

Последним из всех определенных в CSS псевдоклассов является :lang, служащий для оформления элементов в зависимости от языка, заданного для контента этого элемента. Основными способами указания языка для контента элемента служат: в языке HTML — атрибут lang, в XML — xml:lang. С другой стороны, для сопоставления правила, использующего :lang, с языком контента элемента служит двухбуквенное обозначение этого языка, которое указывается в скобках сразу после имени этого псевдокласса. Далее приведем два примера:

- div:lang(de) {} — это правило будет применено к тем элементам, которые имеют либо атрибут lang, если документ имеет формат HTML, либо — xml:lang, если форматом документа является XML, причем значением этого атрибута будет de (deutsch — немецкий);
- div:lang(ru) {} — выбираются те же элементы, что и в предыдущем примере, только контент элемента div является русскоязычным (ru — russian).

7.5.4. Отбор с использованием псевдоэлементов

Псевдоэлементы служат для отбора тех частей документа, которые не могут быть обозначены с помощью языка разметки, использованного для создания документа. Например, как в HTML, так и в XML не существует элемента для обозначения первой буквы или первой строки контента какого-либо узла дерева документа. Кроме того с помощью псевдоэлементов в CSS может быть описан стиль контента, который не существует в исходном документе (этот контент генерируется с помощью псевдоэлементов `:before` и `:after`).

Всего в спецификации CSS определено четыре псевдоэлемента. Первым из них, который мы рассмотрим, является `:first-line`. Он предназначен для описания стиля первой строки контента сопоставленного с ним элемента. рассмотрим пример.

```
p:first-line {text-transform : uppercase }
```

Это правило указывает, что все буквы параграфа, представленного элементом p, должны быть преобразованы в верхний регистр. Здесь селектор `p:first-line` не обозначает какой-либо присутствующий в документе элемент. На самом деле он служит для отбора псевдоэлементов, представляющих собой первую строку контента реальных элементов p языка HTML. Таким образом, следующий код

```
<p>Здесь представлен некоторый достаточно длинный
(в смысле HTML) параграф, который разбивается на несколько
линий. Первая строка идентифицируется с помощью псевдоэлемента
:first-line. Остальные строки интерпретируются как обычные
строки параграфа.</p>
```

будет отформатирован как

ЗДЕСЬ ПРЕДСТАВЛЕН НЕКОТОРЫЙ ДОСТАТОЧНО ДЛИННЫЙ
(в смысле HTML) параграф, который разбивается на несколько
линий. Первая строка идентифицируется с помощью псевдоэлемента
:first-line. Остальные строки интерпретируются как обычные
строки параграфа.

Заметим, что длина первой строки зависит от того, как именно данный пользовательский агент форматирует текст.

Псевдокласс `:first-line` может употребляться только совместно с блочными элементами. Отличия между блочными и внутритекстовыми элементами мы опишем несколько позднее. Далее следует список свойств (property), которые могут быть использованы вместе с данным псевдоклассом: все свойства **шрифтов, Цвета И фона, а также** word-spacing, letter-spacing,

text-decoration, text-transform, text-shadow, vertical-align, line-height и clear. Краткое описание свойств CSS, поддерживаемых MSIE 5.5, приведено в *приложении 1.*

Следующий псевдоэлемент — :first-letter — по аналогии с :first-line служит для оформления первой буквы контента сопоставленного с ним элемента. Главным образом, данный псевдокласс нужен для создания так называемой буквицы. Таким образом, с использованием правила

```
p:first-letter { font-size : 200%;
                 font-weight : bold;
                 float : left;}
```

следующий код

```
<p>Здесь представлен
достаточно длинный
параграф...</p>
```

будет отформатирован как

Здесь представлен
достаточно длинный
параграф...

Точно так же, как и :first-line, псевдоэлемент :first-letter допустим исключительно с блочными элементами. Кроме свойств, рассмотренных для :first-line, в декларации селектора, содержащего псевдоэлемент :first-letter, могут указываться свойства, отвечающие за разного рода отступы: внутренние (padding), внешние (margin) и пограничные (border), которые будут описаны в *разд. 7.1.3.* При этом свойство vertical-align может быть использовано только, если значением float является попе.

В некоторых случаях у разработчиков документов возникает необходимость вставки контента, который не представлен в исходном дереве. Если, например, заголовки первого уровня оформлены с помощью элемента h1 и каждый из разделов, представленных этими заголовками, является отдельной главой, то иногда может потребоваться, чтобы перед текстом заголовка появлялось слово "Глава", после которого следует номер этой главы. Причем было бы очень удобно, если бы такой контент генерировался автоматически.

Для того чтобы генерировать и вставлять контент перед или после контента заданного элемента, как раз и необходимы псевдоклассы :before и :after. Иными словами, с помощью этих псевдоклассов можно вставлять контент соответственно сразу после открывающего тега заданного элемента и непосредственно перед закрывающим тегом.

Существуют три способа вставки контента:

- с помощью свойства content вместе с псевдоэлементами :before и :after;
- с помощью свойств cue-before и cue-after, определенных для акустических (aural) устройств (см. ниже). Когда свойство content применяется вместе с акустическими свойствами, порядок их обработки следующий: :before, cue-before, [pause-before], контент Элемента, [pause-after], cue-after, :after. Здесь в квадратных скобках приведены опционально используемые акустические свойства;
- с помощью свойства display, значением которого должно быть list-item.

Здесь мы рассмотрим только первый способ. Приведем пример

```
h1.chapter {content : "Глава"}
```

С помощью этого правила перед контентом каждого элемента h1 будет вставляться слово "Глава". Для того чтобы вставить ее номер, нужно воспользоваться счетчиком, который может быть представлен одной из функций counter() или counters(). Здесь мы не будем описывать механизм нумерации CSS, т. к. эта тема слишком объемна и, поэтому, не вписывается в рамки нашей книги.

Значениями свойства content могут быть:

- строка, представляющая собой текстовый контент для вставки;
- URI внешнего ресурса, содержащего вставляемый объект;
- счетчик — см. выше;
- open-quote и close-quote — специальные значения для вставки соответственно открывающих и закрывающих кавычек;
- no-open-quote и no-close-quote — специальные значения, с помощью которых не вставляется ничего, но увеличивается (или уменьшается) уровень вложенности кавычек;
- attr(X) — функция, возвращающая строковое значение атрибута с именем x, принадлежащего тому элементу, для которого описан селектор.

Способы формирования списков и выделения текста многоуровневыми кавычками также не будут здесь описаны. Вместо этого мы приведем несколько примеров.

```
q:before { content: open-quote }
q:after { content: close-quote }

h1:before {
    content: "Глава " counter(chapter) ". ";
    counter-increment: chapter;
    counter-reset: section; }
```

```
h2:before (
     content: counter(chapter) "." counter(section);
     counter-increment: section; }
```

В первом примере псевдоэлементы :before и :after служат для автоматической расстановки кавычек вокруг контента элемента q.

Второй пример чуть сложнее. В нем псевдоэлемент :before используется в двух целях: для нумерации глав (chapter) и их подразделов (section). При этом при нумерации глав генерируется слово "Глава", за которым следует позиция в документе элемента chapter и далее идет заголовок, представленный контентом элемента h1. Здесь counter-increment говорит о том, что счетчик глав должен увеличиваться на единицу при обработке каждого следующего элемента chapter. Свойство counter-reset обеспечивает сброс счетчика подразделов до нуля с началом каждой главы.

Заголовки второго уровня представлены элементом h2. Его оформление также осуществляется счетчиком для двухуровневой нумерации подразделов. Здесь первый уровень представлен номером главы, второй — номером подраздела, причем эти номера разделены точкой. Следовательно, форматом нумерации подразделов будет следующий: 1.1, 1.2, ..., 2.1, 2.2 и т. д. Счетчик увеличивается на единицу с каждым встреченным элементом section, а нумерация разделов каждый раз будет начинаться с единицы с началом новой главы.

Как можно заметить, схема работы псевдоэлементов :after и :before одна и та же, но с тем исключением, что для :before сгенерированный контент вставляется непосредственно перед закрывающим тегом элемента, для которого описан селектор.

7.6. Разрешение конфликтов между правилами

Здесь мы опишем механизм выбора наиболее предпочтительной комбинации "элемент/свойство" из некоторого их набора, находящегося в одном файле. Разрешение конфликтов между правилами, находящимися в разных источниках, мы не станем рассматривать в силу его сложности. Заметим только, что для импортированных стилей обеспечивается выбор правил в зависимости от порядка импортирования.

Так же, как и в случае стилей XSLT, в CSS для каждой комбинации элемент/свойство вычисляется ее приоритет, который в терминологии CSS называется спецификой (specificity). Значение специфики складывается из трех составляющих:

- величины а, равной количеству атрибутов ID в селекторе;
- величины ь, равной количеству остальных атрибутов, а также псевдоклассов в селекторе;
- величины с, равной количеству имен элементов в селекторе.

При этом количество псевдоэлементов игнорируется. После того, как величины а, ь и с подсчитаны, вычисляется значение специфики, которое представляет собой результат склейки строковых значений этих величин. Рассмотрим примеры:

*	{}	/*	a=0	b=0	c=0	Специфика=о	*/
LI	{}	/*	a=0	b=0	c=1	Специфика=1	*/
UL LI	{}	/*	a=0	b=0	c=2	Специфика=2	*/
UL LI + OL	{}	/*	a=0	b=0	c=3	Специфика=з	*/
H1 + *[rel=up]	{}	/*	a=0	b=1	c=1	Специфика=11	*/
UL OL LI.red	{}	/*	a=0	b=1	c=3	Специфика=13	*/
LI.red.level	{}	/*	a=0	b=2	c=1	Специфика=21	*/
#34y	{}	/*	a=1	b=0	c=0	Специфика=100	*/

Здесь пояснения, заключенные между /* */ представляют собой комментарии, о которых мы расскажем чуть позже.

7.7. Наследование свойств

Используя древовидную структуру документа, CSS реализует механизм наследования свойств. Предположим, что мы описали два правила для двух разных элементов, и каждое из этих правил содержит свойства, которые в другом правиле не определены:

```
div {color : red}
em {font-weight : bold}
```

Теперь, если мы вставим элемент em в контент элемента div:

```
<div> Некоторый <em>текст</em> </div>
```

то элементом em будут унаследованы свойства его родителя. Для нас это будет означать, что контент элемента em будет отображаться красным цветом.

Таким образом, можно сказать, что наследование — это способность элемента-потомка использовать для своего оформления те свойства элементов-предков, которые не определены в соответствующем ему правиле.

7.8. Комментарии

В целях отладки, а также для пояснения к коду в CSS применяются комментарии со следующими правилами синтаксиса. Комментарий начинается с /* и заканчивается на */. Комментарий может присутствовать где угодно в таблице CSS, и его контент не будет обрабатываться процессором.

Контент комментария может состоять из произвольного числа строк. Комментарии не могут быть вложенными.

7.9. Типы устройств

Одной из важнейших особенностей стилевых таблиц является возможность указать, как документ должен быть представлен на том или ином типе устройств. Некоторые из свойств CSS разработаны и поддерживаются только специфическим для них типом устройств. Применение других может давать различные результаты в зависимости от этого типа.

Для того чтобы указать, для какого типа устройства предназначено данное правило или набор правил, служит at-оператор @media. Синтаксис его такой: после @media следует список типов устройств, разделенных запятой, и далее, в фигурных скобках перечисляются правила отображения элементов для этого типа.

```
@media print {
  body { font-size : 10pt }
@media screen {
  body { font-size : 12pt }
@media screen, print {
  body { line-height : 1.2 }
}
```

В списке устройств могут быть перечислены следующие значения:

- all — правило действительно для всех устройств;
- aural — акустические устройства (синтезаторы речи);
- braille — тактильные устройства Брайля;
- embossed — принтеры для распечатки страниц с использованием азбуки Брайля;
- handheld — портативные устройства (с характерным для них маленьким, монохромным экраном);
- print — принтеры или программы для постраничного предварительного просмотра текста на экране;

- projection — проекторы;
- screen — цветной экран компьютера;
- tty — устройства ввода/вывода, имеющие моноширинную символьную таблицу, такие как телетайпы, терминалы и им подобные устройства без графических возможностей;
- tv — телевизионные устройства (для которых характерно низкое разрешение, поддержка цвета и звука, ограниченная прокрутка рабочей области).

Все вышеперечисленные устройства, согласно спецификации CSS, разбиваются на следующие группы:

- непрерывные (continuous) или постраничные (paged);
- визуальные (visual), акустические (aural) или тактильные (tactile);
- табличные (grid) или графические (bitmap);
- интерактивные (interactive) или статические (static);
- все (all) устройства.

7.10. Поддерживаемые типы значений

7.10.1. Числа

В языке CSS обеспечена поддержка целых (integer) и действительных чисел (real). Целые числа состоят из цифр от 0 до 9, действительные, помимо цифр, могут содержать точку (.). И целые и действительные числа могут начинаться со знаков "плюс" (+) или "минус" (-).

7.10.2. Единицы длины

К единицам длины относятся единицы измерения, с помощью которых задаются длина и ширина прямоугольной области в двумерном пространстве. Их значения указываются с помощью чисел, записанных в кавычках. Некоторые свойства могут содержать отрицательные значения длины и Ширины области.

Согласно спецификации CSS, единицы длины делятся на относительные (relative) и абсолютные (absolute). В относительных единицах измерения выражаются значения, вычисленные относительно какого-либо объекта, в абсолютных — величины, зависящие от физических характеристик используемого устройства.

К относительным единицам измерения относятся:

- em — величина 1em равна размеру шрифта или, более точно, значению свойства font-size, т. о. например, 0.5em и 1.5em будут соответственно равны половине и полутора размерам используемого шрифта. Если em непосредственно является единицей длины значения font-size, то за 1em принимается величина font-size родительского элемента;
- ex — величина 1ex равна значению свойства x-height;
- px — 1px равен размеру одного пиксела, зависящего от устройства.

Абсолютными единицами измерения являются:

- cm — сантиметры;
- mm — миллиметры;
- in — ДЮЙМЫ (1in = 2 . 54cm);
- pt — **пункты** (1pt = 1/72in);
- pc — ПИКИ (1pc = 12pt).

7.10.3. Проценты

Запись этого свойства представляет собой число, заканчивающееся символом процента (%). Значение свойства, указанное в процентах, может быть вычислено относительно свойства того же элемента, либо какого-либо предка, либо исходя из контекста (относительно длины или ширины блочного элемента, содержащего данный элемент). Если значение в процентах указано для корневого элемента, то оно вычисляется относительно значения по умолчанию данного свойства. Рассмотрим пример:

```
P { font-size : 10pt }
P { line-height : 120%}
```

После обработки этих правил значением свойства line-height будет 12pt.

7.10.4. URL + URN = URI

URL (Uniform Resource Locator — уникальное местоположение ресурса) указывает адрес ресурса в сети Web. URN (Uniform Resource Name — уникальное имя ресурса) в совокупности с URL образует URI (Uniform Resource Identifier — уникальный идентификатор ресурса).

В CSS URI необходимы для обеспечения связи с внешними источниками данных. Для этих целей имеется функция url(), аргументом которой является абсолютный или относительный URI ресурса, указанный в кавычках. Например, правило

```
body {background : url(http://www.my_web.com/res/picture.gif) }
```

говорит о том, что в качестве фона для элемента body должен быть взят рисунок, находящийся по адресу (URL) **http://www.my_web.com/res/** и имеющий имя (URN) picture.gif.

7.10.5. Счетчики

В качестве значений некоторых свойств могут быть использованы счетчики, служащие, как правило, для подсчета элементов в документе. Для обеспечения функциональности счетчиков в CSS определены свойства counter-increment И counter-reset, а также функции counter О И `counters()`. Говоря о счетчиках как о значениях некоторых свойств, мы и имели в виду эти функции. Ранее мы уже приводили пример применения счетчиков, когда описывали вставку сгенерированного контента с помощью псевдоэлементов `:before` И `:after`.

7.10.6. Цвета

В спецификации CSS определены три способа задания цвета. Первый — с помощью предопределенных ключевых слов, служащих для обозначения основных цветов. Всего ключевых слов 16, и ими являются следующие: aqua **(цвет МОрСКОЙ ВОЛНЫ)**, black **(черный),** blue **(СИНИЙ),** fuchsia **(фуКСИЯ),** gray (серый), green (зеленый), lime (лайм), maroon (насыщенный красно-коричневый), navy (темно-синий), olive (оливковый), purple (пурпурный), ted (красный), silver (серебряный), teal (цвет морской волны), white (белый) и yellow (желтый). Ключевые слова в CSS записываются без кавычек.

Второй и третий способы реализованы на основе цветовой RGB-модели.

При записи по второму способу, значением свойства, отвечающего за цвет, будет начинающееся с решетки (#) шестнадцатеричное число, в свою очередь, состоящее из трех двухбайтовых величин для каждой из составляющих: красной (Red), зеленой (Green) и синей (Blue). Схематично синтаксис такого идентификатора цвета можно записать как `#rrggbb`. Для этого способа существует краткая форма записи, состоящая не из шести, а из трех символов — `#rgb`. При обработке значение цвета, заданное таким образом, конвертируется в шестисимвольный формат. Причем это преобразование происходит дублированием каждого из трех символов, а не путем дополнения нулями до шестисимвольного значения. Например, значение #f30 будет Преобразовано к `#ff3300`, а не к `#f30000`.

Третий способ задания значения цвета заключается в использовании функции `rgb()`, имеющей три аргумента, каждый из которых отвечает за соответствующую составляющую RGB-представления цвета. В качестве значения каждого аргумента этой функции могут выступать десятичные числа от

О до 255 или процентные величины. В последнем случае 0% соответствует числу 0, 100% — числу 255.

Следующие правила являются эквивалентными.

```
div { color : #f00 }
div { color :ttffOOOO}
div { color : rgb(255, 0, 0) }
div { color : rgb(100%, 0, 0) }
```

7.10.7. Строки

Строковое значение для свойства устанавливается с помощью двойных или одинарных кавычек. Если требуется, чтобы строка содержала одинарную кавычку, в то время как эта строка сама заключена в одинарные кавычки, то перед содержащейся в ней одинарной кавычкой должен стоять обратный слеш (\). Точно так же строка, заключенная в двойные кавычки, не может содержать внутри себя двойные кавычки. В этом случае также необходим обратный слеш.

Строковое значение свойства не может состоять из нескольких строк. Но строка может содержать специальный символ \A, обозначающий разрыв. Далее следует несколько примеров строк:

```
"Эта строка содержит 'одинарные' кавычки"
'Эта строка содержит "двойные" кавычки'
"Строка с \"кавычками\""
'Строка с \'кавычками\''
```

Помимо вышеописанных, в стандарте CSS определены также единицы измерения для отсчета углов, времени и частоты. Но они применимы только к специфическим типам устройств и не могут быть задействованы при разработке Web-приложения, клиентом которого является MSIE. Поэтому мы исключаем их из нашего обзора.

7.11. CSS и размещение объектов на Web-странице

В этом разделе мы рассмотрим т. н. модель форматирования документа для визуального просмотра (visual formatting model). В соответствии с этой моделью пользовательский агент обрабатывает дерево документа для визуального устройства. В результате такой обработки каждый элемент исходного дерева генерирует ноль, один или несколько т. н. боксов (box) согласно box-модели, которая также будет рассмотрена в рамках этого раздела.

Схема расположения сгенерированных боксов зависит от трех факторов:

- типа бокса;
- характеристик, определяемых box-моделью;
- характеристик среды размещения (принцип размещения: позиционирование или выравнивание, а также размерность пространства, в котором располагаются области).

рассмотрим каждый из этих факторов по порядку.

7.11.1. Блочные, внутритекстовые и невидимые элементы

В стандарте CSS определено несколько типов элементов, в зависимости от того, как располагаются сгенерированные ими боксы относительно остального контента. Здесь мы рассмотрим только три из них, а именно те, которые поддерживаются MSIE 5.5. Если для какого-либо элемента необходимо задать требуемый тип бокса, т. е. если для него этот тип не задан по умолчанию, то можно воспользоваться свойством display, которое определено для каждого элемента языка HTML.

Первый тип элементов — блочный — характеризуется тем, что генерируемые этими элементами боксы при позиционировании по умолчанию располагаются, начиная с новой строки. При этом следующий за блочным элемент или текстовый узел генерирует бокс, который также располагается с новой строки вне зависимости от своего типа. Таким образом, блочные боксы находятся как бы вне остального контента, т. е. образуют отдельный блок. Значением свойства display для блочных элементов является block.

Второй тип элементов называется внутритекстовым, и, как следует от его названия, боксы этого типа не разрывают контент, а располагаются прямо в нем. Для внутритекстовых элементов значением свойства display является inline. Главным различием между блочными и внутритекстовыми элементами выступает их взаиморасположение в родительском, т. е. их содержащем блоке. Блочные боксы выстраиваются вертикально и могут быть выровнены по горизонтали. Соответственно, внутритекстовые боксы выстраиваются горизонтально и могут быть выровнены по вертикали.

Одним из наиболее часто используемых блочных элементов HTML является div, а внутритекстовым — span. Другими словами, для этих элементов значениями по умолчанию свойства display будут соответственно block и inline. Проиллюстрируем на примерах разницу в их форматировании.

```
<div style="align:center">
 Некоторый текст
    <div> Первый блочный элемент </div>
    <div> Второй блочный элемент </div>
    <div> Третий блочный элемент </div>
 Некоторый текст
</div>
```

Результатом обработки этого кода будет визуальное представление, изображенное на рис. 7.1.

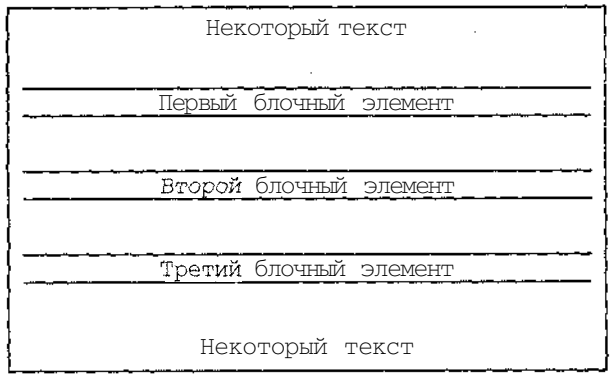


Рис. 7.1. Представление блочных элементов

Как видно, он содержит три вертикально расположенных блочных бокса, отцентрированные по горизонтали. При этом текстовые узлы образуют т. н. анонимные боксы, которые находятся в отдельных строках.

Если же элемент div заменить элементом span

```
<div align = "center">
 Некоторый текст
    <span style = "vertical-align:super"> Первый внутритекстовый элемент
</span>
    <span style = "vertical-align:middle"> Второй внутритекстовый элемент
</span>
    <span style = "vertical-align:sub"> Третий внутритекстовый элемент
</span>
 Некоторый текст
</div>
```

то в качестве представления документа получим вариант, изображенный на рис. 7.2.

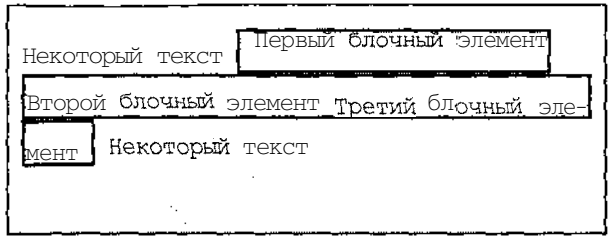


Рис. 7.2. Форматирование внутритекстовых элементов

Как видно из рис. 7.2, здесь боксы уже не располагаются один после другого, а равномерно заполняют родительский бокс, разрывая строки по мере его заполнения в ширину. При этом значение атрибута align родительского элемента div не играет никакой роли, а свойства vertical-align по вполне понятным причинам нет вовсе.

Третий тип элементов представляет собой средство для указания процессору не отображать те элементы, которые не предназначены для оформления документа. Невидимые элементы служат некоторым специфическим целям разработчика. По своим функциям они напоминают инструкции обработки из языка XML. Для того чтобы элемент не имел представления, нужно в качестве значения свойства display указать попе.

При создании HTML-документа блочные элементы главным образом служат для построения структуры отображения контента, т. е. взаиморасположения соответствующих им боксов. Наиболее часто используемым в этих целях элементом HTML является p, служащий для представления некоторой части текста в виде параграфа. Блочными также являются элементы, служащие для представления пунктов списков — li, а также элементы заголовков различного уровня h1, h2 и т. д.

Внутритекстовые элементы служат в основном для оформления некоторых частей текста, выделяющихся по смыслу на общем фоне. Ими могут быть ключевые слова, определения, сноски, пометки, небольшие примечания и т. п. Для этих целей хорошо подходят различные свойства шрифтов: размер, цвет, подчеркивание, наклон (курсив) и т. д. Примерами внутритекстовых элементов являются:

- i (italic) — курсив;
- b (bold) — полужирный шрифт;
- strike(strikethrough) — зачеркнутый текст.

Среди примеров невидимых элементов можно выделить meta, служащий для передачи клиенту метаданных, а с атрибутом name, представляющий собой якорь для ссылок, head, содержащий заголовок документа, и title, содержащий заголовок некоторой части документа.

7.11.2. Box-модель и таблицы

Эта модель описывает свойства отображения прямоугольной области вне зависимости от ее местоположения как абсолютного, так и относительно соседних боксов. Согласно этой модели каждая прямоугольная область имеет четкую внутреннюю структуру, определяющую характеристики подобда, стей, образующих данную прямоугольную область. Рассмотрим рис. 7.3.

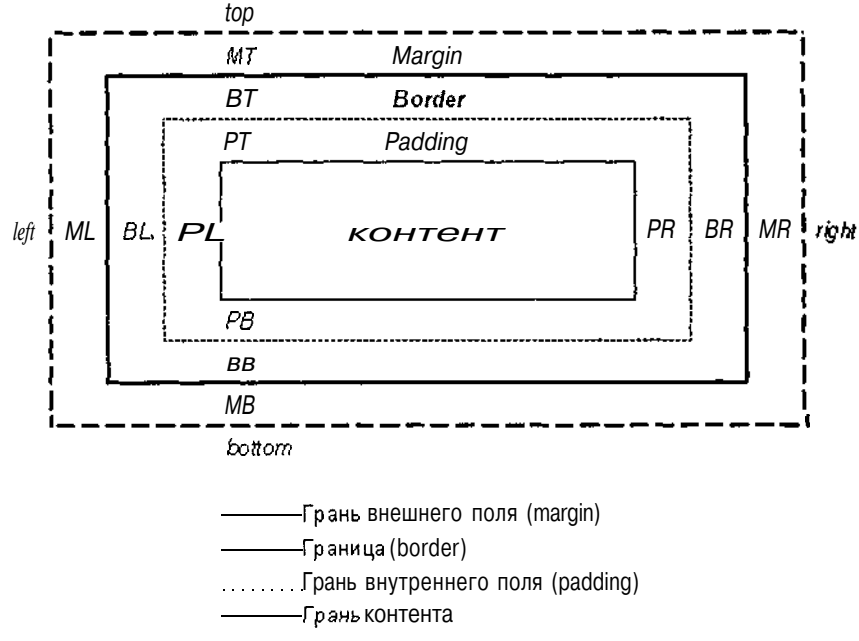


Рис. 7.3. Структура прямоугольной области

Здесь изображены внутренние подобласти, из которых в соответствии с box-моделью состоит каждая прямоугольная область. Рассмотрим их по порядку.

- margin (обрамление) — представляет собой внешнее поле, обрамляющее прямоугольную область. Для задания характеристик внешнего поля служат следующие свойства:
  - `margin-left` — значением этого свойства является ширина левого внешнего поля (ML на рис. 7.3); margin-right — ширина правого внешнего поля (MR на рис. 7.3);
  - margin-top — высота верхнего внешнего поля (MT на рис. 7.3);
  - `margin-bottom` — высота нижнего внешнего поля (MB на рис. 7.3);

- margin — это значение присваивается всем остальным margin-свойствам следующим образом: если задано одно или четыре значения, разделенные запятыми, то оно присваивается всем остальным свойствам; если два, ТО первое присваивается свойствам margin-top И margin-bottom, второе — свойствам margin-left и margin-right; если три, то первое **Присваивается СВОЙСТВУ** margin-top, **второе — Свойствам** margin-left И margin-right, **И третье — свойству** margin-bottom.

  Если два блочных бокса находятся строго один под другим (или один в другом), то соответствующие соприкасающиеся внешние поля могут быть сжаты в некоторое единое внешнее поле. При этом размер этого результирующего поля равен максимальному из них или нулю, если все эти размеры отрицательны. Сжиматься могут только вертикальные внешние поля (MT и MB на рис. 7.3).

- border (граница) — граница прямоугольной области. За стиль ее отображения **отвечают свойства** border-top-style, border-bottom-style, border-left-style, border-right-style, border-style, **Значением КОТОРЫХ** является стиль внешнего вида границы. Этими значениями могут быть:

  - none — значение по умолчанию, при котором граница не отображается;
  - dotted — граница представляет собой линию, состоящую из точек;
  - dashed — граница представляет собой линию, состоящую из штрихов;
  - solid — непрерывная линия;
  - double — двойная непрерывная линия;
  - groove — трехмерная линия, напоминающая бороздку;
  - ridge — трехмерная линия, напоминающая гребень;
  - inset — трехмерная линия, при которой контент области выглядит вогнутым;
  - outset — трехмерная линия, при которой контент выглядит выпуклым.

Для задания цвета границы служат свойства border-top-color, border-bottom-color, border-left-color, border-right-color И border-color. Значениями этих свойств может быть идентификатор цвета, заданный так, как это было описано в *разд. 7.1.2.*

Свойства, отвечающие за размеры границы, определены по аналогии со свойствами, указывающими размеры внешнего поля. Этими свойствами являются: border-top-width, border-bottom-width, border-left-width, border-right-width И border-width. **Значениями ЭТИХ СВОЙСТВ СООТВеТСТ**венно являются: высота верхней границы (BT на рис. 7.3), высота нижней границы (BB на рис. 7.3), ширина левой границы (BL на рис. 7.3), ширина правой границы (BR на рис. 7.3) и значение, присваивающееся четырем предыдущим свойствам.

Внимание

Если для какой-либо границы установлен стиль none (например, `border-top-style: none`), то свойство, отвечающее за ее ширину, игнорируется, т. е. считается равным нулю (здесь `border-top-width: 0`).

При задании характеристик границ можно воспользоваться сокращенным синтаксисом для всех вышеперечисленных свойств. В рамках этого синтаксиса определены свойства border-top, border-bottom, `border-left`, border-right и border. Значениями этих свойств могут быть размеры, стиль или цвет соответственно верхней, нижней, левой, правой, а также всех четырех границ области.

- padding (заполнение) — представляет собой внутреннее по отношению к границе (border) поле области. Для задания его характеристик служат следующие свойства:
  - padding-top — высота верхнего внутреннего поля (PT на рис. 7.3);
  - padding-bottom — высота нижнего внутреннего поля (PB на рис. 7.3);
  - `padding-left` — ширина левого внутреннего поля (PL на рис. 7.3);
  - padding-right — ширина правого внутреннего поля (PR на рис. 7.3);
  - padding — служит для одновременного задания значений четырех вышеперечисленных полей по аналогии с margin.
- Контент — подобласть, содержащая контент элемента. Ее границы вычисляются процессором в зависимости от размеров этого контента. Для управления этим вычислением служит свойство overflow, которое может принимать одно из четырех значений:
  - visible — если задано это значение, то размер области контента равен размеру самого контента. Иными словами, если размер контента превышает размеры области, то размеры области будут увеличены процессором так, чтобы контент помещался в нее целиком, т. е. был полностью виден. Является значением по умолчанию;
  - hidden — при этом размеры области являются такими, какими их определил пользователь, и не изменяются в зависимости от размера контента;
  - scroll — у области присутствует полоса (полосы) прокрутки вне зависимости от того, помещается контент в область или нет. Если размер контента больше размера области, то полоса прокрутки становится активной, т. е. появляется возможность прокручивать контент;
  - auto — если установлено это значение, то полоса (полосы) прокрутки появляется у области, если размер контента больше размера области, и исчезает, если — меньше.

Если У области присутствуют полосы прокрутки, то они располагаются ме-жду ее границей (border) и внутренним полем (padding).

Примечание

Процессор браузера MSIE 5.5 помимо `overflow` поддерживает еще два свойства `overflow-x` и `overflow-y`. Значениями этих свойств могут быть те же значения, что и для `overflow`. Эти свойства отвечают за вычисление соответственно ширины и высоты области контента в зависимости от его размеров.

На рис. 7.3 присутствуют также слова top (верх), right (правый), bottom (низ) и left (левый). Одноименные свойства отвечают за позиционирование прямоугольной области в окне пользовательского агента. О том, как процессор вычисляет позицию бокса, речь пойдет в следующем подразделе.

Для ячеек таблицы схема расположения соответствующих подобластей несколько расходится с Box-моделью. Прямоугольная область, представляющая собой ячейку таблицы, не содержит внешнего поля (margin), а значит, для генерирующего ее элемента td не применимы margin-свойства. Но для элемента table, генерирующего таблицу, определены свойства cellspacing и `cellpadding`, отвечающие соответственно за расстояние между границами (border) ячеек и за расстояние между границей (border) ячейки и ее внутренним полем (padding).

7.11.3. Позиционирование областей

В спецификации CSS определены два способа расположения областей в родительском блоке. Первый представляет собой непосредственное позиционирование областей с помощью свойства position, второй — схему размещения плавающих областей с использованием свойства float. Отметив, что Данные свойства не всегда являются взаимоисключающими, рассмотрим каждый из этих способов.

При непосредственном позиционировании допустимо применение следующих свойств:

- top — расстояние от верхней грани родительской области до верхней грани данной области;
- left — расстояние от левой грани родительской области до левой грани данной области;
- bottom — расстояние от нижней грани родительской области до нижней грани данной области;
- right — расстояние от правой грани родительской области до правой грани данной области.

Эти свойства представляют собой координаты прямоугольной области, в Родительской области. При этом центр системы координат находится

в левом верхнем углу родительской области, ось абсцисс *(X)* направлена вправо, а ось ординат *(Y)* — вниз.

Значением свойств top, bottom, left и right могут быть любые величины, представляющие собой единицы длины или проценты, а также специальное значение auto, которое является значением по умолчанию. Если установлено это значение, то соответствующая координата вычисляется процессором автоматически.

Наряду с данными четырьмя свойствами иногда применяются width (ширина контента) и height (высота контента), при этом значение width может **быть вычислено как** right-left, a height — как bottom-top.

Значения вышеперечисленных свойств могут по-разному использоваться процессором в зависимости от значения свойства position, с помощью которого можно указать требуемую схему позиционирования области. При этом каждому значению этого свойства соответствует своя схема позиционирования. Всего таких схем четыре.

Нормальная (normal) схема позиционирования

Если выбрана эта схема, то области в родительском боксе располагаются аналогично блочным и внутритекстовым областям. Иными словами, блочные элементы генерируют области, начинающиеся с новой строки, и следующий за данной областью бокс любого типа также начинается с новой строки. Внутритекстовые элементы генерируют области, не разрывающие контент родительского бокса.

При данной схеме позиционирования родительской по отношению к позиционируемой будет область, генерируемая элементом, являющимся родителем того элемента, который генерирует позиционируемую область. Значением свойства position для этой схемы является static. Оно является значением по умолчанию. При нормальной схеме позиционирования значения свойств top, left, right и bottom процессором игнорируются.

Схема относительного (relative) позиционирования

Принцип расчета координат области согласно схеме относительного позиционирования является наиболее сложным по сравнению с остальными. При данной схеме позиционирования родительская область определяется так же, как и при нормальной. Соответствующим ей значением свойства position является relative.

Согласно этой схеме вычисление координат области происходит в два этапа. Сначала процессор вычисляет координаты области согласно нормальной схеме. После этого к вычисленным значениям прибавляются значения свойств top, left, right и bottom. При некоторых значениях этих свойств возможно наложение областей друг на друга. Рассмотрим несколько примеров.

Вначале проиллюстрируем схему относительного позиционирования на областях, соответствующих внутритекстовым элементам, взяв в качестве такого span. Для этого где-либо в документе или вне его определим правило:

```
span ( position : relative }
```

Пусть у нас имеется следующий код:

```
<div>
    Некоторый текст
    <span style="top:-10px"> Внутритекстовый элемент 1.0
        <span style="top:10px"> Внутритекстовый элемент 1.1 </span>
    </span>
    <span style="top:10px"> Внутритекстовый элемент 2.0 </span>
    Некоторый текст
</div>
```

После обработки этого кода мы получим следующее представление документа

Некоторый текст Внутритекстовый элемент 1.0

Внутритекстовый элемент 1.1

Внутритекстовый элемент 2.0

Некоторый текст

Как видно, каждый из представленных здесь боксов располагается выше или ниже остальных. Рассмотрим эти боксы в том порядке, как представлены в дереве документа соответствующие им узлы.

Первый из боксов генерируется текстовым узлом с контентом некоторый текст. Координаты соответствующей области вычислены так, как это определено нормальной схемой позиционирования областей.

Следующая область (внутритекстовый элемент 1.0) располагается на 10 пикселов выше предыдущей. Иными словами, координата top этой области Равняется значению этой координаты, вычисленной согласно нормальной схеме, минус 10 пикселов.

Область, представленная контентом внутритекстовый элемент 1.1, соответствует второму элементу span, который является непосредственным потомком первого одноименного элемента. Расчет координаты top этой области выполняется следующим образом. Сначала вычисляется значение top согласно нормальной схеме позиционирования. Но здесь родительским боксом является не та область, которая соответствует div, как раньше, а та, которая генерируется первым элементом span. Таким образом, координата top, согласно нормальной схеме позиционирования и относительно рассматриваемого элемента span, будет иметь значение 10px.

После того, как координата top вычислена согласно нормальной схеме позиционирования, к ней прибавляется значение свойства top рассматриваемого элемента span, равное 10px. Таким образом, результирующим значением top будет Юрх для рассматриваемого элемента span, и -10px+10px=0px — относительно области, генерируемой элементом div. В итоге текст внутритекстовый элемент 1.1 будет располагаться на том же самом уровне (на той же самой высоте), что и текстовый узел с контентом некоторый текст.

Месторасположение области, представленной контентом Внутритекстовый элемент 2.0 определяется точно так же, как и области с контентом внутритекстовый элемент 1.0. Отличие заключается только в том, что ее положение на 10 пикселов ниже, а не выше нормальной позиции.

И наконец, последний узел текстового типа с контентом Некоторый текст располагается, как и первый текстовый узел, в соответствии с нормальной схемой позиционирования.

Для того чтобы проиллюстрировать относительную схему позиционирования областей, генерируемых блочными элементами, предположим, что нам требуется расположить на одной высоте три текстовых бокса. Для этой цели в качестве генератора внутритекстовых боксов естественно воспользоваться элементом span. Но мы будем работать с div, т. к. этот элемент генерирует блочные боксы.

Для каждого из этих элементов определим правило:

```
div.my_class { position : relative; height : Юрх }
```

Далее следует код, представляющий собой решение нашей задачи:

```
<div>
    <div class="my_class" style="top:0px; left:0px;">Текст 1 </div>
    <div class="my_class" style="top:-10px; left:50px; ">Текст 2 </div>
    <div class="my_class" style="top:-20px; left:100px; ">Текст 3 </div>
</div>
```

В результате его обработки мы получим следующее представление

```
Текст 1     Текст 2     Текст 3
```

Так как высота каждой области равна Юрх, то в соответствии с особенностями блочных боксов согласно нормальной модели позиционирования данные области будут иметь в качестве значений координаты top соответственно Орх, Юрх и 20px, и располагаться друг под другом. Поэтому для того, чтобы данные области располагались на одной высоте, нам необходимо присвоить значениям свойств элементов div соответственно Орх, -Юрх и -20px. Эти значения, будучи прибавлены к вычисленным на первом этапе позиционирования, обеспечат для всех координат top значение, равное 0px.

Но если мы ограничимся только заданием значений для свойств top, то все три области будут наложены друг на друга. Поэтому для каждого из элементов div мы задали еще и значение свойства left, благодаря которому области располагаются рядом друг с другом с интервалом 50px.

Относительная схема позиционирования применяется, как правило, для внутритекстовых боксов. Примерами здесь могут послужить те из них, которые служат для оформления сносок, а также верхних и нижних индексов.

Схема абсолютного (absolute) позиционирования

Согласно схеме абсолютного позиционирования координатами областей являются значения свойств top, bottom, left и right, вне зависимости от типа позиционируемых боксов, их взаимного расположения, а также любых других факторов. Родительской областью при абсолютной схеме позиционирования будет область, генерируемая элементом, являющимся ближайшим предком того элемента, который генерирует позиционируемую область, и имеющим значение свойства position отличное от static. Значением свойства position, соответствующим данной схеме, будет absolute. Рассмотрим пример.

Вначале определим правило:

```
div { position : absolute;
      width : 100; height : 100;
      border-style : solid;
      border-width : 1px; }
```

В результате обработки следующего кода:

```
<div style="top:10;left:10"/>
<div style="top:30;left:30"/>
<div style="top:50;left:50"/>
```

получим представление, изображенное на рис. 7.4.

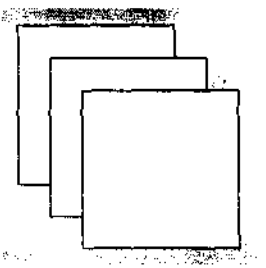


Рис 7.4. Абсолютное позиционирование областей

Схема фиксированного (fixed) позиционирования

Данная схема основывается на схеме абсолютного позиционирования областей. Другими словами, координаты области согласно данной схеме вычисляются так же, как и в случае абсолютной. Отличие этих двух типов областей состоит в том, что при фиксированном позиционировании область всегда имеет одни и те же координаты в рабочем пространстве устройства, на котором выводится документ. Это пространство является родительской областью при данной схеме позиционирования.

Для устройств, поддерживающих непрерывный вывод, таких, например, как экран компьютера, область с фиксированно заданными координатами всегда отображается в одном и том же месте экрана, вне зависимости от того, какая часть документа просматривается в данный момент. Данная область не меняет своего местоположения и при прокрутке (scrolling) документа.

Для устройств с постраничным выводом позиции области фиксируются в пределах страницы.

Значением свойства position для фиксированной схемы позиционирования является `fixed`.

Внимание

Фиксированная схема позиционирования областей не поддерживается браузером MSIE 5.5.

При выборе относительной, абсолютной или фиксированной схемы позиционирования процессором могут быть проигнорированы значения следующих свойств:

- right, если контент в родительской области направлен слева направо и какое-либо из свойств left или left-margin имеет значение, отличное OT auto;
- left, если контент в родительской области направлен справа налево и какое-либо из свойств right или `right-margin` имеет значение, отличное от auto;
- bottom, если какое-либо из свойств top или top-margin имеет значение, отличное auto.

Примечание

Направление контента в родительской области указывается с помощью свойства direction, которое может принимать какое-либо из двух значений: `ltr` (left-to-right — слева направо) и `rtl` (right-to-left — справа налево). Направление контента справа налево характерно, например, для арабского языка.

7.11.4. Схема размещения плавающих (float) областей

Второй способ размещения областей в родительском боксе в отличие от схем позиционирования основывается не на непосредственном указании координат области, а на расположении элемента, генерирующего эту область, в дереве документа, а также на различных вариантах обтекания контентом данной области.

Для того чтобы сделать область плавающей, необходимо для генерирующего ее элемента установить свойству float какое-либо из двух значений: left или right. Значение left означает, что область будет соприкасаться с левой гранью родительской области, если же установлено значение right — с правой. При этом верхняя грань плавающей области выравнивается либо относительно верхней грани ее содержащей строки (строчный бокс), либо, если такая строка отсутствует, относительно верхней грани блочного бокса, содержащего данную область.

Рассмотрим следующий код

```
<html>
 <head>
  <style>
   div, p { border-style: solid;
            border-width: 1px }
  </style>
 </head>
 <body>
  <div style="position:absolute;width:200px">
   <img src="image.bmp" alt="Рисунок"
        style="float:right;width:70px;height:130px"/>
   <p>Параграф, представленный первым элементом p, содержит
      некоторый текст, обтекающий рисунок.</p>
   <p>Параграф, представленный вторым элементом p, содержит
      некоторый текст, обтекающий рисунок.</p>
  </div>
 </body>
</html>
```

Результате его обработки получим следующее представление, изображенное на рис. 7.5.

Для элемента, соответствующего плавающей области, должно быть задано свойство width, кроме тех случаев, когда этот элемент служит для вставки

внешних объектов, таких, например, как рисунок или Java-апплет. При вставке внешних объектов механизм вычисления ширины области не регламентируется спецификацией CSS, за него отвечает вставляемый объект.

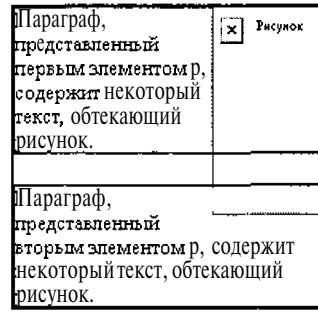


Рис. 7.5. Плавающая область

Если необходимо, чтобы контент какой-либо из областей не обтекал плавающую область, то следует воспользоваться свойством clear, которое может принимать одно из четырех значений:

- left — верхнее внешнее поле данной области увеличивается настолько, что ее верхняя граница находится ниже, чем внешняя нижняя грань каждой плавающей со смещением влево области, если элемент, генерирующий эту плавающую область, предшествует элементу, генерирующему данную область;
- right — верхнее внешнее поле данной области увеличивается настолько, что ее верхняя граница находится ниже, чем внешняя нижняя грань каждой плавающей со смещением вправо области, если элемент, генерирующий эту плавающую область, предшествует элементу, генерирующему данную область;
- both — область ведет себя так, как если бы свойству clear одновременно были присвоены оба значения: left и right;
- none — значение по умолчанию, при котором контент области обтекает плавающую область.

Свойство float также может принимать значение none, при котором область не является или перестает быть плавающей.

Далее следует список правил, в соответствии с которыми должны отображаться плавающие области.

1. Граница левого внешнего поля плавающей с выравниванием по левому краю области не может находиться левее левой границы родительского

бокса. Аналогичное правило существует и для плавающих с выравниванием по правому краю областей.

2. Если перед элементом, генерирующим плавающую с выравниванием по левому краю область, находятся элементы, генерирующие аналогичные области, то либо левая внешняя граница данной области должна быть справа от внешней границы каждой из предшествующих областей, либо ее верхняя грань должна располагаться под нижней гранью любой предшествующей области. Аналогичное правило существует и для плавающих с выравниванием по правому краю областей.
3. Правая внешняя граница плавающей с выравниванием по левому краю области не может быть справа от левой внешней границы любой плавающей с выравниванием по правому краю области. Аналогичное правило существует и для плавающих с выравниванием по правому краю областей.
4. Верхняя внешняя грань плавающей области не может находиться выше, чем верхняя грань родительской области.
5. Верхняя внешняя грань плавающей области не может находиться выше, чем верхняя внешняя грань любого блочного или плавающего бокса, сгенерированного ранее данной области.
6. Плавающая с выравниванием по левому краю область не может иметь правой внешней грани, которая находится правее ее содержащего бокса, если данная область не максимально смещена к левому краю родительского бокса. Существует аналогичное правило и для плавающих с выравниванием по правому краю областей.
7. Плавающая область должна располагаться так высоко, насколько это возможно.
8. Плавающая с выравниванием по левому краю область должна быть максимально смещена к левому краю родительского бокса, плавающая с выравниванием по правому краю — к правому.

7.11.5. Третье измерение

В спецификации CSS обеспечена возможность размещения областей не только на плоскости, но и в трехмерном пространстве. Ранее, описывая, например, схему абсолютного позиционирования областей, мы показали, как области могут перекрывать друг друга. Но в той ситуации область перекрывала другие области только в том случае, если сгенерировавший ее элемент был обработан Позже тех элементов, которые сгенерировали остальные области.

Иногда может возникнуть ситуация, при которой разработчику понадобится ˚рганизовать перекрытие областей не по этому принципу, а как-либо иначе.

внешних объектов, таких, например, как рисунок или Java-апплет. При вставке внешних объектов механизм вычисления ширины области не регламентируется спецификацией CSS, за него отвечает вставляемый объект.

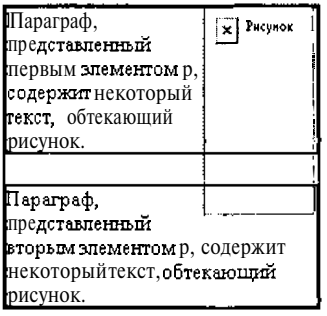


Рис. 7.5. Плавающая область

Если необходимо, чтобы контент какой-либо из областей не обтекал плавающую область, то следует воспользоваться свойством clear, которое может принимать одно из четырех значений:

- left — верхнее внешнее поле данной области увеличивается настолько, что ее верхняя граница находится ниже, чем внешняя нижняя грань каждой плавающей со смещением влево области, если элемент, генерирующий эту плавающую область, предшествует элементу, генерирующему данную область;
- right — верхнее внешнее поле данной области увеличивается настолько, что ее верхняя граница находится ниже, чем внешняя нижняя грань каждой плавающей со смещением вправо области, если элемент, генерирующий эту плавающую область, предшествует элементу, генерирующему данную область;
- both — область ведет себя так, как если бы свойству clear одновременно были присвоены оба значения: left и right;
- none — значение по умолчанию, при котором контент области обтекает плавающую область.

Свойство float также может принимать значение попе, при котором область не является или перестает быть плавающей.

Далее следует список правил, в соответствии с которыми должны отображаться плавающие области.

1. Граница левого внешнего поля плавающей с выравниванием по левому краю области не может находиться левее левой границы родительского

бокса. Аналогичное правило существует и для плавающих с выравниванием по правому краю областей.

2. Если перед элементом, генерирующим плавающую с выравниванием по левому краю область, находятся элементы, генерирующие аналогичные области, то либо левая внешняя граница данной области должна быть справа от внешней границы каждой из предшествующих областей, либо ее верхняя грань должна располагаться под нижней гранью любой предшествующей области. Аналогичное правило существует и для плавающих с выравниванием по правому краю областей.
3. Правая внешняя граница плавающей с выравниванием по левому краю области не может быть справа от левой внешней границы любой плавающей с выравниванием по правому краю области. Аналогичное правило существует и для плавающих с выравниванием по правому краю областей.
4. Верхняя внешняя грань плавающей области не может находиться выше, чем верхняя грань родительской области.
5. Верхняя внешняя грань плавающей области не может находиться выше, чем верхняя внешняя грань любого блочного или плавающего бокса, сгенерированного ранее данной области.
6. Плавающая с выравниванием по левому краю область не может иметь правой внешней грани, которая находится правее ее содержащего бокса, если данная область не максимально смещена к левому краю родительского бокса. Существует аналогичное правило и для плавающих с выравниванием по правому краю областей.
7. Плавающая область должна располагаться так высоко, насколько это возможно.
8. Плавающая с выравниванием по левому краю область должна быть максимально смещена к левому краю родительского бокса, плавающая с выравниванием по правому краю — к правому.

7.11.5. Третье измерение

В спецификации CSS обеспечена возможность размещения областей не только на плоскости, но и в трехмерном пространстве. Ранее, описывая, например, схему абсолютного позиционирования областей, мы показали, как области могут перекрывать друг друга. Но в той ситуации область перекрывала другие области только в том случае, если сгенерировавший ее элемент был обработан Позже тех элементов, которые сгенерировали остальные области.

Иногда может возникнуть ситуация, при которой разработчику понадобится организовать перекрытие областей не по этому принципу, а как-либо иначе.

Для этих целей в CSS определен механизм, называющийся многослойным представлением. Согласно этому механизму каждой области присвоен номер слоя, который соответствует координате на оси Z, направленной из глубины рабочего пространства к наблюдателю. Таким образом, чем больше номер слоя, тем ближе к пользователю находится область, перекрывающая пересекающиеся с ней области, которые обладают более низким номером.

Для того чтобы задать номер слоя области, в CSS определено свойство z-order, значением которого может быть любое целое число или специальное значение auto. По умолчанию установлено значение, унаследованное от ближайшего предка. Если же ни у одного предка не задано целочисленное значение z-order, то значением по умолчанию является auto и область располагается на нулевом уровне.

Существуют понятия локальной и глобальной системы отсчета номеров слоя. В глобальной системе рассчитываются номера слоев областей, сгенерированных элементами, являющимися непосредственными потомками корневого элемента. Локальную систему создает каждая область для отсчета номеров слоев своих дочерних областей.

Рассмотрим следующий код.

Листинг 7.1. Пример использования слоев

```
<html>
 <head>
  <style>
   div {position:absolute;
        border-width:2px; border-style:solid; padding-left:5px;
        background-color:white;
        font-weight:bold}
  </style>
 </head>
<body>
 <div style="top:100px;left:100px;width:210px;height:210px">
  <div style="top:10px;left:10px;width:100px;height:100px;z-index:-1">
    z-order:-1
  </div>
  <div style="top:40px;left:40px;width:100px;height:100px;z-index:1">
    z-order:1
    <div style="top:30px;left:30px;width:80px;height:50px;z-index:3">
      z-order:3
    </div>
```

```
  </div>
  <divstyle="top:95px;left:95px;width:100px;height:100px;z-index:2">
    z-order:2
  </div>
 </div>
</body>
</html>
```

На рис. *7.6* представлен результат обработки этого кода.

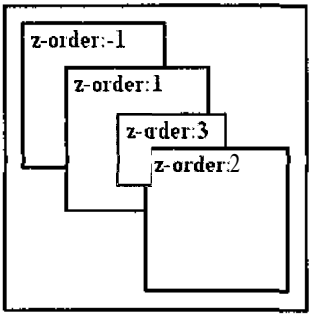


РИС. 7.6. Пример ИСПОЛЬЗОВанИЯ z-order

Внимание

Как видно из листинга 7.1, мы использовали свойство с именем `z-index`, которое в MSIE 5.5 является синонимом для z-order.

Обратите внимание на то, что, как видно из рис. 7.6, область с номером слоя 3 перекрывается областью с номером слоя 2. Это обуславливается тем, что в глобальной системе отсчета номер перекрываемой области равен 1, т. е. тому же значению z-order, что и у ее родителя. Но в локальной системе отсчета номер данной области равен 3, и поэтому данная область перекрывает родительскую, т. к. номер слоя родительской области в рассматриваемой локальной системе отсчета равен 0.

7.12. Поддержка CSS 1 и 2 в MSIE 5.5

Стандарты, связанные с языками разметки, непрерывно развиваются консорциумом W3C. И язык CSS не является исключением. С другой стороны, Производители пользовательских агентов не спешат включать в свои продукты поддержку самых последних версий этих стандартов. Например, не-

давно вышедший браузер Microsoft Internet Explorer 6.0 только в этой версии объявил о стопроцентной поддержке CSS 1.

Поэтому необходимо помнить, что то или иное свойство CSS может быть определено в самой последней версии спецификации этого языка и поэтому может не поддерживаться ни одним пользовательским агентом. Но даже если это свойство определено в относительно ранней версии спецификации, то отсюда еще не следует факт его поддержки используемым пользовательским агентом.

Далее мы приводим краткий обзор того, в какой мере язык CSS поддерживается браузером MSIE 5.5. Более подробную информацию о сопоставлении стандартов обеих версий CSS, а также об их поддержке в MSIE 5.5 можно получить из *приложения 1.*

7.12.1. Определение стиля для элементов

Ранее мы описали три способа подключения стилей к HTML-документу CSS. Все они поддерживаются MSIE 5.5. Напомним, что первый заключается в подключении внешней стилевой таблицы с помощью элемента link, второй — в описании встроенного стиля в заголовке документа (в контенте элемента head) с помощью элемента style, и третий — в непосредственном описании стиля для конкретного отображаемого элемента с помощью атрибута style этого элемента.

Можно упомянуть также еще один способ, при котором внешние стилевые таблицы импортируются во внешние или внутренние таблицы с помощью at-оператора CSS @import. Но вообще говоря, применение @import не является отдельным способом подключения стилевых таблиц, т. к. этот оператор используется в коде CSS, который уже должен быть подключен к HTML-документу с помощью одного из трех вышеперечисленных способов.

При описании стиля элемента следует учитывать, что некоторые свойства данного элемента могут быть для него уже определены и для них задано значение по умолчанию. Например, не имеет смысла для элемента div определять свойство display со значением block. Кроме того, MSIE 5.5 предоставляет возможность указывать значения некоторых свойств с помощью специальных атрибутов. Например, вместо `<h1 style="align:center">...</h1>` можно записать `<h1 align="center">... </h1>`.

7.12.2. Динамические стили

При разработке дизайна Web-страницы не стоит забывать о существовании JavaScript, который позволяет организовать динамическое изменение ее внешнего вида в зависимости от действий пользователя. С помощью встро-

енных в MSIE 5.5 средств могут быть определены, например, цвет фона некоторой области или нескольких областей, ее размер и координаты, и вообще все те свойства CSS, которые поддерживает этот браузер.

Существуют два способа взаимодействия со стилем, приписанным к некоторому элементу. Первый заключается в обращении к объекту style, который является полем или свойством любого объекта, служащего для представления каждого отображаемого элемента языка HTML. Рассмотрим пример.

```
<div id = "div1" style = "position:absolute; left:10px; width:100px"/>
```

Для того чтобы обратиться к объекту style, нужно воспользоваться конструкцией `div1.style`. Здесь элемент div представлен объектом divl, а style, сам будучи объектом, является его свойством.

Объект style сам имеет свойства, которые являются свойствами и в смысле CSS, и в смысле языка JavaScript. Здесь термин "объект" следует понимать в смысле объектно-ориентированного программирования, т. е. как переменную такого типа данных, которому свойственна инкапсуляция — объединение данных (свойства — properties) и методов (methods) их обработки, а также наследование и полиморфизм. Более подробно об объектной модели документа (Document Object Model — DOM), реализованной в MSIE 5.5, мы поговорим в следующей главе.

Свойствами объекта style являются все те свойства CSS, которые определены для элемента HTML, к которому приписан данный объект. Например, свойству left элемента div из вышеприведенного примера будет соответствовать свойство `style.left` объекта divl.

Для именования свойств объекта style в MSIE 5.5 действует следующее соглашение: если имя некоторого свойства CSS содержит дефис (например, background-color), то соответствующее ему свойство объекта style будет иметь такое же имя, но в нем дефис будет отсутствовать, и вторая часть имени свойства CSS, которая следует сразу после дефиса, будет начинаться с прописной буквы (в нашем случае — `backgroundColor`). Аналогично строится имя для свойства CSS, название которого содержит несколько дефисов.

Если же имя свойства CSS не содержит дефиса, то имя свойства style будет с ним полностью совпадать, хотя встречаются и исключения. Напомним, что язык JavaScript является чувствительным к регистру символов, и поэтому нужно быть внимательным при использовании имен различных конструкций этого языка.

Теперь если мы захотим, например, изменить ширину области, представленной объектом divl, то нам нужно будет написать следующий сценарий JavaScript:

```
<script>
    div1.style.width = новое значение ширины области;
    alert('Ширина области = '+ div1.style.width);
</script>
```

Здесь во второй строке сценария используется функция alert о, которая служит для вывода сообщения, представленного ее аргументом, на экран.

Вторым способом взаимодействия со стилем элемента средствами JavaScript является организация доступа к атрибуту class данного элемента. Напомним, что этот атрибут содержит имя класса (стиля), описанного либо во внешней таблице CSS, либо в контенте элемента style. Для этих целей служит свойство className, приписанное каждому объекту, представляющему любой отображаемый элемент HTML. Предположим, что у нас имеется следующее описание стиля для элемента div:

```
div.class1 { background-color : green }
div.class2 { background-color : blue }
```

А где-либо в документе есть код:

```
<div id = "div1" class = "class1">... </div>
```

Теперь для смены цвета фона этого элемента с зеленого на синий мы можем написать следующий код JavaScript:

```
...
div1.className = 'class2';
...
```

Кроме свойств style и className, иногда можно воспользоваться именами атрибутов элемента, отвечающих за его отображение. Если элемент имеет такие атрибуты (как в вышеприведенном примере для h1), то соответствующий ему объект будет иметь одноименные свойства. Рассмотрим следующий код:

```
<h1 align = "center"
    onmouseover = "this.align='right'"
    onmouseout = "this.align='center'"> Заголовок </h1>
```

Здесь представление элемента h1 зависит от действий пользователя, просматривающего документ. Изначально текст заголовок выровнен по центру родительской по отношению к h1 области. Если пользователь установит указатель мыши над областью, соответствующей элементу h1, то его контент будет выровнен по правому краю, а после того как пользователь переместит указатель мыши за пределы данной области, текст снова будет выровнен по центру.

Благодаря тому, что существует возможность взаимодействия со стилями посредством JavaScript, и при этом внешний вид элементов меняется "на лету", т. е. без перезагрузки страницы, механизм стилей в MSIE 5.5 назван динамическим. В этой связи следует обратить внимание на т. н. выражения (expressions), которые служат для присваивания значений свойствам в тех случаях, когда эти значения не известны во время разработки документа.

Предположим, что мы хотим отобразить некоторую блочную область, высота которой должна быть в два раза больше ее ширины, а ширина, в свою очередь, определена как некоторый процент от ширины родительской области (например, `width:60%`), т. е. не является постоянной величиной. Если мы укажем высоту, как `height:30%`, то это будет неверным, т. к. здесь 30% процентов будет вычисляться относительно высоты родительской области, а не ее ширины.

формально механизм выражений представляет собой способ присвоить некоторому свойству значение, являющееся формулой, переменными в которой выступают свойства каких-либо других элементов. При этом объекты, соответствующие этим элементам, должны находиться в области видимости данного объекта. В нашем случае результатом вычисления выражения должна быть величина, равная width/2. Для того чтобы присвоить это значение СВОЙСТВУ height, нужно воспользоваться функцией `expression()`. Это присвоение будет записано следующим образом:

```
height:expression(this.width/2)
```

Аргументом функции `expression()` является код JavaScript, в результате обработки которого должно быть получено значение, корректное для определяемого свойства.

Наряду с expression o можно воспользоваться следующими функциями:

- `object.setExpression(sPropertyName, sExpression, sLanguage)` — служит для присвоения свойству sPropertyName, принадлежащему объекту object, значения, являющегося выражением sExpression. Аргумент sLanguage представляет собой строку, в которой содержится имя языка, на котором написано выражение, и может принимать одно из трех значений VBScript, JScript и JavaScript. Обязательными являются только первые два аргумента функции;
- string `object.getExpression(sPropertyName)` — возвращает выражение, являющееся значением свойства sPropertyName, принадлежащего объекту object. Аргумент sPropertyName обязателен;
- boolean `object.removeExpression(sPropertyName)` — **возвращает** true, если выражение, являющееся значением свойства sPropertyName, принадлежащего объекту object, было успешно удалено. В противном случае возвращает false. Аргумент sPropertyName обязателен;

- document.recalc(bForceAll) — служит для пересчета выражений. Если булево значение необязательного аргумента bForceAii равно false (значение по умолчанию), то происходит пересчет только тех выражений, которые были изменены со времени последнего пересчета. Если значением аргумента является true, то пересчитываются все выражения в документе.

7.13. Совместное использование CSS и XML

Ранее в наших примерах для представления данных мы уже употребляли в шаблонах XSLT элементы языка HTML. Но существует возможность применения для этой цели только каскадных стилевых таблиц. Для этого достаточно вставить в XML-файл следующую инструкцию обработки:

```
<?xml-stylesheet type="text/css" href="URI файла CSS"?>
```

В первой и второй главах книги мы приводили в качестве примера простого XML-документа обычное заявление (см. листинги 1.1 и 2.1). Предположим, что мы вставили в XML-файл, представленный листингом 1.1, аналогичную приведенной выше инструкцию обработки:

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

Для того чтобы данные, содержащиеся в этом файле, имели представление такое, как на рис. 2.1, вполне подойдет таблица стилей, код которой дан в листинге 7.2.

Листинг 7.2. Каскадная таблица стилей application.css

```
application {display : block}

description {display : block;
             width : 200px;
             float : right}

for, from {display : block;
           font-weight : bold;}

title {display : block;
       clear : right;
       text-align : center;
       font-size : 200%
       font-weight : bold;}

body {display : block;
      text-indent : 1.5em}
```

Внимание

В данной таблице стилей отсутствует правило для элемента date, т. к. MSIE 5.5 не поддерживает ни один из описанных ранее способов, определенных в спецификации CSS для генерирования контента. В листинге 2.1 для этой цели используется шаблон XSLT.

Заметим, что стилевые таблицы могут находиться как во внешнем, так и во внутреннем источнике. Для этого в XML файл необходимо вставить следующую инструкцию обработки:

```
<?xml-stylesheet type="text/css" href="#style1"?>
```

А стилевую таблицу поместить в контент специально созданного элемента с произвольным именем, например, style:

```
<style id="style1" style="display : none">
...
</style>
```

Этот элемент может находиться где угодно в документе среди потомков корневого элемента. Заметим, что style должен иметь два атрибута: id — идентификатор элемента, по которому его находит процессор, основываясь на атрибуте href инструкции обработки, а также атрибут style со значением display : none для того, чтобы код CSS не отображался на Web-странице.

Заметим также, что совместное использование XML и CSS без посредничества HTML не исключает подключения к документу XSLT. Но в этом случае результатом обработки будет документ в формате XML, а не HTML.

Браузер MSIE 5.5 обеспечивает некоторые, хотя и несколько ограниченные, возможности обработки XML-документов. Поэтому вышеописанные документы формата XML+CSS, а также XML+CSS+XSLT вполне могут быть им обработаны без помощи какого-либо внешнего синтаксического анализатора (в нашем случае Cocoon). Но здесь следует учитывать, что MSIE 5.5 не полностью поддерживает язык XSLT, а значит может обрабатывать только относительно простые XML-документы.

Для решения тех задач, которым посвящена книга, не имеет никакого значения, в какой мере браузер MSIE 5.5 реализует поддержку языка XML. Большинство документов, которые мы описали или опишем далее, имеют Логический слой, что подразумевает активное взаимодействие как документов различного формата, так и различных приложений на стороне сервера. При этом некоторые из таких серверных операций гораздо эффективнее Производить именно на стороне сервера или клиента.

Предположим, например, что наше Web-приложение выполняет некоторые сложные математические расчеты. Если бы обработку XML-документа брал на себя клиент, то он должен был бы, во-первых, обладать достаточной

производительностью для того, чтобы выполнить эти расчеты (сравнимой с производительностью сервера). Во-вторых, этот клиент должен был бы поддерживать все те технологии, которые использует данное Web-приложение, такие, например, как один или несколько языков программирования, реализующих его логический слой, интерфейс доступа к базам данных и т. п. И в-третьих, такому клиенту понадобилось бы поддерживать постоянное соединение с сервером как с хранилищем информации, причем это соединение было бы загружено весьма приличным трафиком.

Короче говоря, если бы существовал клиент, удовлетворяющий всем вышеперечисленным требованиям, то его смело можно было назвать сервером, работающим в паре с другим сервером, т. е. с тем, к которому подключен данный клиент. Поэтому поддержка браузером или каким-либо еще клиентом языка XML актуальна только при разработке отдельных, т. е. слабо взаимодействующих друг с другом, и относительно простых XML-документов. Для полнофункционального же Web-приложения такая поддержка не играет какой-либо значительной роли.

7.14. Практический пример: создание кнопки

В этом подразделе мы поэтапно рассмотрим процесс разработки дизайна простого объекта с использованием CSS. Несмотря на возможность создания XML-документа без посредничества HTML, мы все-таки воспользуемся этим языком, т. к. для многих его элементов требуемый стиль оформления уже определен по умолчанию соответствующей спецификацией. Например, в предыдущем подразделе для оформления элемента `title` мы применяли свойства display, font-weight и font-size, хотя вместо этого проще было бы воспользоваться элементом `h1` так, как это было проделано в листинге 2.1.

Как правило, кнопка представляет собой прямоугольную область с надписью. Поэтому, прежде всего, мы должны выбрать ее местоположение относительно остального контента, а также ее размеры. Если кнопка располагается внутри этого контента, не разрывая его, то проще всего воспользоваться элементом span, в противном случае, т. е. если соответствующая область является блочной, для ее представления можно выбрать `div`.

Предположим, что кнопка располагается вне контента, т. е. среди других подобных ей элементов управления. Тогда нам следует воспользоваться элементом div, а для того, чтобы она была расположена в точном соответствии с остальными элементами управления, мы применим схему абсолютного позиционирования областей. Итак, первой частью правила для div будет:

```
position : absolute; /* схема позиционирования абсолютная */
left : 20px; top : 20px; /* координаты левого верхнего угла области */
width : 60px; height : 20px; /* ширина и высота области */
```

Выберем в качестве надписи для кнопки слово "Текст" и пропишем его в контент элемента div. Выбрав в качестве фона серый цвет

```
background-color : gray;
```

мы уже можем посмотреть на то, что у нас получилось (рис. 7.7).

Рис. 7.7. Создание кнопки: первый этап

Отметим, что на рис. 7.7 текст надписи плохо виден и не отцентрирован. Определив следующие свойства

```
text-align : center;
color : white;
```

в результате получим то, что показано на рис. 7.8.

Рис. 7.8. Создание кнопки: второй этап

Теперь для того, чтобы кнопка, изображенная на рис. 7.8, выглядела объемно, необходимо определить свойства, отвечающие за отображение границы области:

```
border-width : 2px;
border-style : outset;
```

В итоге получим изображение кнопки, представленное на рис. 7.9, весьма приближенное к реальному.

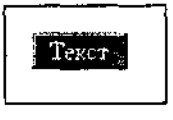


Рис. 7.9. Создание кнопки: третий этап

Заметим, что указатель мыши имеет форму курсора, которая соответствует положению мыши над текстом. Между тем он должен иметь форму стрелки, которая появляется при наведении указателя мыши, например, на полосу Прокрутки или меню. Для того чтобы исправить положение, добавим последнее правило:

```
cursor : default
```

Если немного поработать с цветами шрифта и фона, то можно получить такую кнопку, которую содержит любое диалоговое окно, т. е. ничем не отличающуюся от аналога, предоставляемого графическим ядром операционной системы.

Созданную нами кнопку можно превратить в объект, обладающий весьма изощренным дизайном. В отличие от кнопок, которые можно создать с помощью специальных элементов языка HTML (этими элементами являются button и input, атрибут type которого имеет значение button), контент соответствующего нашей кнопке элемента div может содержать произвольный HTML-код, а также любые внешние объекты (рисунки, анимацию, Flash-объекты и т. п.).

О том, как заставить кнопку реагировать на нажатие клавиши мыши, мы обсудим в *главе 9,* в которой более подробно опишем механизм динамического взаимодействия со стилями CSS.

ГЛАВА 8

Методика обработки данных

Самым распространенным средством взаимодействия пользователя с хранилищем информации на сервере являются Web-формы (или HTML-формы). В этой главе мы спроектируем форму, которая содержит поля ввода различных данных, а также опишем механизм, с помощью которого серверная часть Web-приложения обрабатывает эти данные.

8.1. Проектирование HTML-формы

В предыдущей главе мы, используя каскадные стилевые таблицы, создали кнопку, представляющую собой некоторую прямоугольную область. HTML-форма также отображается как прямоугольная область, поэтому нет нужды описывать этот аспект ее представления. Вместо этого мы рассмотрим те элементы языка HTML, которые влияют на способы обработки и ввода Данных, передаваемых с помощью этой формы.

8.1.1. Элемент *form*

Итак, для того чтобы создать Web-форму, в языке HTML служит элемент form. Далее приведем синтаксис этого элемента с описанием его наиболее важных атрибутов:

```
<form
    action = "URI приложения, обрабатывающего данные формы"
    target = "окно, в котором будет выведен результат обработки формы"
    method = "метод передачи данных"
```

```
    enctype = "MIME-тип данных">
<!-- контент формы -->
</form>
```

Первый атрибут action содержит адрес приложения, которое получит данные, введенные пользователем, обработает их и вернет результаты пользователю. Значением этого атрибута может быть, например, приложение CGI, скрипт PHP и любой другой инструмент серверной обработки данных.

Так как для реализации логического слоя Web-приложения мы выбрали язык Java, то самым логичным будет указать в качестве значения этого атрибута адрес созданного нами сервлета. Но, во-первых, непосредственное создание сервлета представляет собой достаточно непростую задачу и выходит за рамки этой книги; во-вторых, напомним, что XML-документ посредством Cocoon сам может быть превращен в сервлет. Поэтому в качестве значения action мы будем указывать URL некоторого XML-документа, при этом сервер фактически будет обращаться не к самому этому XML-документу, а к его представляющему байт-коду. Далее мы проиллюстрируем этот механизм на примерах.

Второй атрибут target служит для указания того места, в котором должны быть выведены результаты обработки данных, поступившие от сервера. MSIE 5.5 определяет следующие значения для этого атрибута:

- пале — имя окна или фрейма (для его представления служит элемент frame);
- _blank — результат будет выведен в новом окне браузера, специально открытом для этих целей;
- _parent — результат будет загружен в окно, являющееся родительским по отношению к содержащему форму;
- _search — результат будет загружен в панель поиска MSIE 5.5;
- _self — значение по умолчанию, результат будет загружен в окно, ранее содержащее форму;
- _top — результат загружается в окно, являющееся самым отдаленным предком окна, содержащего форму.

Атрибут method служит для указания способа, с помощью которого данные будут отправлены на сервер. Всего существует два допустимых значения для данного атрибута:

- get — значение по умолчанию. Данные передаются в строке запроса (query string), которая добавляется к URI, указанному в значении атрибута action. Этому способу присущи два недостатка: во-первых, объем данных, передаваемых в строке запроса, ограничен 256 байтами минус длина строки URI, и, во-вторых, передаваемые данные отображаются в адресной строке, а также в строке состояния браузера, т. е. видны пользователю, а это не всегда приемлемо.

- post — данные передаются с помощью специально создаваемого HTTP-запроса (HTTP post transaction), при этом они невидимы для пользователя, и их объем можно считать неограниченным. Недостатком этого способа является то, что данные передаются в совокупности с некоторой служебной информацией, служащей для их форматирования, поэтому их серверная обработка является более сложной по сравнению с обработкой данных, переданных с использованием метода get. Далее мы опишем тонкости обработки данных, переданных с помощью обоих способов.

Атрибут enctype служит для указания формата передаваемых данных. Его значением ПО умолчанию является `application/x-www-form-urlencoded`. Описание различных MIME-типов не входит в наши задачи, но о них можно узнать из стандарта протокола HTTP, описание которого нетрудно найти в Internet. Здесь мы укажем только еще один из этих типов — `multipart/form-data`, который используется при передаче данных методом post. Напомним, что ранее мы уже упоминали несколько простейших MIME-типов, которые также могут быть использованы в качестве значения атрибута enctype. Примером такого значения может послужить `text/plain`.

Внимание

В JavaScript свойством объекта `form`, **соответствующим атрибуту** enctype, **является** encoding (`form_object.encoding = "`*MIME-тип*`"`).

Теперь перейдем к контенту элемента form. В его качестве может выступать любой HTML-код, который, как правило, содержит элементы, обеспечивающие различные формы ввода информации пользователем. Этими элементами являются:

- input — служит для создания полей ввода нескольких типов данных;
- textarea — предназначен для ввода неформатированного текста, состоящего из нескольких строк;
- select — селектор, служащий для выбора одного из предопределенных текстовых значений.

Рассмотрим все эти элементы по порядку.

8.1.2. Элемент *input*

Синтаксис элемента следующий:

```
<input
  name = "имя поля ввода"
  type = "тип поля ввода"
  value = "начальное значение поля ввода" />
```

Первый атрибут — name — служит для идентификации поля ввода среди других полей формы. При передаче данных серверу значение поля name будет взято в качестве имени соответствующего параметра HTTP-запроса как для метода get, так и для post. Следует отметить, что атрибут name по своему предназначению отличается от атрибута id, который служит для идентификации объектов JavaScript.

Атрибут value предназначен для взаимодействия Web-страницы с введенным пользователем значением. С помощью этого атрибута можно указать начальное значение соответствующего поля ввода, а после того, как пользователь ввел данные и подтвердил ввод, существует возможность обработать эти данные с помощью языка JavaScript до отправки их на сервер. Можно, например, проверить корректность ввода, и в том случае, если введенные данные не удовлетворяют каким-либо требованиям, то не отправлять их на сервер, а вместо этого попросить пользователя посредством соответствующего сообщения повторить ввод:

```
if (input_object.value > max_value)
    {alert("Введенное значение больше максимально возможного!
           Повторите ввод.");
    return 0;} else
    form_object.submit();
```

Значением атрибута type может быть одно из следующих, определенных браузером MSIE 5.5, ключевых слов:

- text — элемент ввода представляет собой однострочное поле для ввода текста. Если выбрано это значение, то для указания максимальной длины строки, которую может ввести пользователь, может быть задействован атрибут `maxlength`, принадлежащий данному элементу input. Кроме того, с помощью атрибута size, принадлежащего тому же элементу можно указать длину в символах области, представляющей соответствующее поле ввода.
- hidden — элемент ввода не отображается на экране, но его значение будет передано на сервер после подтверждения ввода. Обычно это поле используется для передачи служебной информации приложению, обрабатывающему введенные пользователем данные. В качестве примера можно привести следующую ситуацию. Предположим, что форма используется в нескольких целях: для создания новой записи в информационном хранилище (например, регистрационной карточки нового пользователя), а также редактирования уже имеющихся. В данном случае значением (value) поля типа hidden может выступить заданный разработчиком формы идентификатор режима (редактирования или создания), в котором используется данная форма.

- password — элемент ввода, аналогичный полю ввода типа text, но с двумя отличиями. Каждый символ набираемого пользователем текста отображается в виде звездочки (*), как при наборе пароля. Второе отличие заключается в том, что начальным значением этого поля всегда является пустая строка.

- checkbox — элемент ввода, использующийся при организации выбора одного или нескольких вариантов из предлагаемого набора. При этом каждому варианту соответствует отдельный элемент input с атрибутом type со значением checkbox. Для поля этого типа значением по умолчанию атрибута value всегда будет on. Значением этого поля является одно из двух состояний: помечен (вариант выбран) и не помечен (вариант не выбран). Для того чтобы указать, является вариант изначально помеченным или непомеченным, служит атрибут checked, принадлежащий соответствующему элементу input. Если значением атрибута checked является true, то вариант считается выбранным, если false, то нет.

- radio — элемент ввода, использующийся для организации выбора только одного из имеющихся вариантов. При этом каждому варианту соответствует отдельный элемент input с атрибутом type со значением radio. Для того чтобы пользователь имел возможность выбрать только один из данной группы вариантов, необходимо, чтобы все их представляющие поля ввода имели одно и то же имя. Или, иными словами, все элементы input, представляющие данную группу вариантов, имели одно и то же значение атрибута name. Также, как и поле типа checkbox, поле рассматриваемого типа может принимать одно из двух значений: помечен/не помечен. Начальное состояние элемента также определяется с помощью атрибута checked, который может принимать значения true ИЛИ false.

- file — элемент ввода, служащий для отправки файла от клиента серверу. Поле этого типа в MSIE 5.5 представляет собой строку, сразу после которой (справа) располагается кнопка **Обзор.** При нажатии на эту кнопку появляется стандартное диалоговое окно Windows, служащее для открытия файлов. После того как файл выбран и в диалоговом окне нажата кнопка **Открыть,** в строке рассматриваемого поля ввода появляется полный путь и имя файла, который подлежит отправке на сервер. Атрибут value в этом случае служит только для получения заданного значения поля ввода, т. е. пути и имени файла, но не для установки начального значения. При использовании поля типа file важную роль играют значения атрибутов method и enctype элемента form.

- button — элемент ввода, служащий для создания кнопки, к которой по умолчанию не приписано никакого действия. Иными словами, эта кнопка не будет реагировать ни на какие действия пользователя, кроме

тех случаев, когда ее поведение было специально описано разработчиком Web-страницы.

- reset — элемент ввода, который также отображается как кнопка, при нажатии на которую все остальные элементы ввода формы принимают свои начальные значения.
- submit — элемент ввода, который также отображается как кнопка, при нажатии на которую происходит подтверждение ввода, т. е. пользователь сообщает о том, что он закончил ввод данных и они могут быть отправлены на сервер.
- image — элемент ввода, служащий для вставки в форму изображения. При нажатии на него кнопкой мыши происходит подтверждение ввода, и на сервер передаются две пары имя/значение, соответствующие этому элементу. Первая пара представляет собой имя данного поля ввода, к которому добавлена строка x=, и значение x-координаты указателя мыши, относительно левой стороны изображения в момент нажатия. Если, например, именем (значением атрибута name) поля типа image было my_img, а x-координатой указателя мыши — 20, то на сервер будет отправлена строка `my_img.x=20`. Вторая пара имя/значение служит для отправки на сервер y-координаты указателя мыши по аналогии с первой парой. Для указания URI фафического объекта используется обязательный атрибут src, принадлежащий соответствующему элементу input. Атрибут value всегда игнорируется.

8.1.3. Элемент *textarea*

Этот элемент предназначен для ввода многострочного неформатированного текста и, соответственно, представляет собой многострочное поле ввода. То обстоятельство, что текст не форматирован, означает, что любая находящаяся в нем разметка будет интерпретироваться процессором HTML как обычный текст. Если где-либо в тексте встретится, например, элемент i, то в результате вместо выделения части текста курсивом, этот элемент будет присутствовать в тексте явно, т. е. в виде тега <i>.

Рассмотрим синтаксис этого элемента:

```
<textarea
    name = "имя элемента ввода"
    value = "значение элемента ввода"
    rows = "количество столбцов"
    cols = "количество строк"
    wrap = "режим переноса строк">
<!-- текст для редактирования -->
</textarea>
```

Здесь атрибуты name и value полностью эквивалентны своим аналогам, принадлежащим элементу input, атрибут type которого имеет значение text. Атрибуты rows и cols указывают количество соответственно столбцов и строк редактируемого текста.

Атрибут wrap заслуживает более серьезного внимания. Он предназначен для того, чтобы определить, как должны быть представлены строки, которые по ширине не вмещаются в область редактируемого текста. Этот атрибут может принимать одно из трех значений:

- *soft* — значение по умолчанию, при котором текст переносится по словам и при подтверждении ввода в местах разрыва строк символы конца строки и возврата каретки процессором не устанавливаются;
- hard — текст переносится по словам, но при подтверждении ввода в местах разрыва строк процессором устанавливаются символы конца строки и возврата каретки;
- off — перенос строки не производится, но вместо этого у области редактирования присутствует горизонтальная полоса прокрутки.

8.1.4. Элемент *select*

Селектор позволяет организовать выбор одного или нескольких вариантов из предопределенного списка. Приведем его синтаксис.

```
<select
    name = "имя селектора"
    size = "количество отображаемых вариантов"
    multiple = "индикатор многовариантности выбора">
<!-- список вариантов -->
</select>
```

Здесь атрибут name, так же, как и у других элементов ввода, служит для задания имени параметра в отправляемой серверу паре имя/значение.

Атрибут size указывает количество элементов списка, отображаемых одновременно. Если значением size является 1, то селектор отображается в виде строки, представляющей собой текст одного из элементов списка, справа от которой располагается кнопка, служащая для отображения остальных элементов списка. Если же значением size является число, большее чем единица, то отображается столько элементов списка, сколько указано в значении этого атрибута. Если при этом значение size меньше общего количества элементов списка, то с правой стороны прямоугольной области, представляющей этот список, будет присутствовать полоса прокрутки.

Атрибут multiple может принимать одно из двух значений: true, если пользователю разрешается выбирать одновременно несколько вариантов (аналогия — поле типа checkbox), и false, если разрешается выбрать только один вариант (аналогия — поле типа radio).

Для описания каждого элемента списка, представленного селектором, служит элемент

```
<option
    value = "значение выбранного варианта"
    selected = "индикатор выбора варианта">
<!-- Текст элемента списка -->
</option>
```

Здесь атрибут value предназначен для задания того значения, которое будет отправлено серверу в случае выбора этого варианта. Если значение value не задано, то будет отправлен текст элемента списка, т. е. контент элемента option.

Атрибут selected эквивалентен атрибуту checked, принадлежащему элементу input и обрабатываемому в том случае, когда значением атрибута type этого элемента является checkbox или radio. Если значением selected является true, то вариант является изначально помеченным, если false, то — нет.

Примечание

Область, генерируемая селектором, является оконным элементом (window element) и не может быть перекрыта другими областями. Иными словами, эта область всегда будет полностью видна при любом значении свойства z-index пересекающихся с ней областей. Сам элемент select свойством z-index не обладает.

Далее мы более подробно объясним, как передаются и обрабатываются данные, введенные пользователем в поля ввода различных типов.

8.1.5. Законченная форма

Здесь мы приведем пример формы, которая содержит поля ввода почти всех вышеописанных типов. Исключение составляют элемент ввода textarea, т. к. его обработка эквивалентна обработке поля типа text, представленного элементом input, а также селектор с возможностью одновременного выбора нескольких альтернатив, т. к. обработка отправляемых с помощью этого элемента данных почти ничем не отличается для его варианта с возможностью выбора одной альтернативы, но значительно увеличивает объем кода.

Листинг 8.1. Файл form.xml. Исходный код формы.

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-format type="text/html"?>

<html>
 <head>
  <style>
   div, form, fieldset {position :absolute}
   form {left:10px;top:10px;width:390px;height:290px;
         border-width:2px; border-style: outset;
         background-color:#8399B1}
   input {width:180px}
  </style>
 </head>
 <body>
  <form action="form_handler.xml">
   <div style="top:10px;left:10px">
    Текстовое поле 1 <br/>
    <input name="text1" type="text"/>
   </div>
   <div style="top:55px;left:10px">
    Текстовое поле 2 <br/>
    <input name="text2" type="text" value="Значение по умолчанию"/>
   </div>
   <div style="top:100px;left:10px">
    Поле типа password <br/>
    <input name="text3" type="password"/>
   </div>
   <div style="top:145px;left:10px">
    Селектор <br/>
    <select style="width:180px" name="selector" size="3" multiple="true">
     <option>Первое значение</option>
     <option>ВТОрое значение</option>
     <option>Третье значение</option>
    </select>
   </div>
```

```
    <div style="top:190px;left:10px">
     Поле типа file <br/>
     <input style="width:300px" name="file" type="file"/>
    </div>
    <fieldset style="top:10px;left:210px;width:160px">
     <legend>Три поля типа checkbox</legend>
     <input style="width:20px" name="ch1" type="checkbox"/>
       checkbox 1 <br/>
     <input style="width:20px" name="ch2" type="checkbox" checked="true"/>
     checkbox 2 <br/>
     <input style="width:20px" name="ch3" type="checkbox"/>
     checkbox 3 <br/>
    </fieldset>
    <fieldset style="top:100px;left:210px;width:160px">
     <legend>Три поля типа radio</legend>
     <input style="width:20px" name="r1" type="radio" value="Значение1"/>
     radio 1 <br/>
     <input style="width:20px" name="r1" type="radio" checked="true"
            value="Значение2"/> radio 2 <br/>
     <input style="width:20px" name="r1" type="radio" value="Значение3"/>
     radio 3 <br/>
    </fieldset>
    <div style="top:250px;left:10px">
     <input name="hValue" type="hidden"
            value="Скрытое от пользователя значение"/>
     <input style="width:80px" type="reset"/>
     <input style="width:120px" type="submit"/>
    </div>
   </form>
  </body>
</html>
```

Как видно из этого листинга, мы не воспользовались возможностями языка XSLT, т. к. данный XML-документ не требует какой-либо обработки содержащейся в нем информации в зависимости от ее структуры. Хотя, разумеется, можно было бы при необходимости заключить некоторые блоки в логические скобки тегов XML и создать для них соответствующие правила шаблона.

о листинге 8.1 нами было использовано несколько не описывавшихся ранее элементов, которые служат для оформления полей формы. Этими элементами являются fieldset и legend. Как следует из названия первого элемента (field set — множество полей), он предназначен для представления некоторой совокупности полей формы. Элемент legend при этом служит для представления названия этой совокупности. Заметим, что элемент legend должен быть первым непосредственным потомком fieidset.

Далее следует рис. 8.1, на котором изображена получившаяся в результате обработки файла form.xml форма.

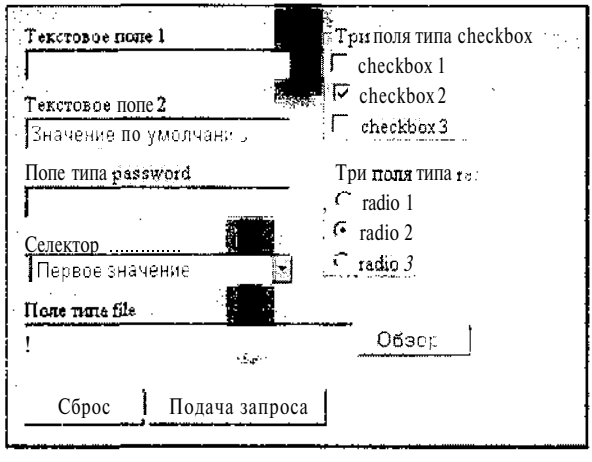


Рис. 8.1. Законченная форма

Рассмотрим подробно, что же здесь изображено.

Первое поле с подписью текстовое поле 1 представляет собой результат обработки элемента input со значением text атрибута type. Это поле сформировано контентом первого элемента div в листинге 8.1. Контент второго элемента div описывает второе текстовое поле с надписью текстовое поле *2.* Оно отличается от первого тем, что для него с помощью атрибута value задано начальное значение значение по умолчанию. Последнее поле, которое служит для ввода текста, описывается контентом третьего элемента div. Типом этого поля является password, и подписано оно соответственно Поле типа password.

Ниже текстовых полей располагается селектор, описанный с помощью элементов select и option. Список его значений состоит из трех строк: первое значение, Второе значение И Третье значение. Атрибуты size И multiple элемента select имеют значения по умолчанию, т. е. соответственно 1 (если список не развернут, отображается только один его элемент) и false (одно подтверждение ввода соответствует выбору одной альтернативы).

Ниже селектора располагается поле типа file с соответствующей подписью. Как видно, оно представляет собой строку, в которой после выбора файла будет отображаться его полный путь и имя, а также кнопку выбора файла (с надписью **Обзор),** располагающуюся справа от строки.

В правой части формы находятся, снизу вверх: три поля типа checkbox и три поля типа radio. Для второго поля типа checkbox (с именем ch1 — значение атрибута name) с помощью атрибута checked задано значение по умолчанию помечено. С помощью одноименного атрибута задано значение по умолчанию и для второго поля типа radio. Отметим, что в соответствии со спецификой элементов ввода типа radio, все три соответствующие этим полям элемента input, присутствующие в листинге 8.1, имеют одно и то же имя — r1.

Поля ввода типа checked находятся в рамке, на верхней стороне которой присутствует надпись три поля типа checkbox. Такое представление достигается с помощью элементов оформления полей ввода fieldset и legend. Для того чтобы некоторая группа полей ввода находилась в рамке, необходимо соответствующие элементы ввода поместить в контент элемента fieidset. А для того чтобы надписать данную группу полей, нужно воспользоваться элементом legend, контентом которого и будет соответствующая надпись. Три элемента типа radio были нами сгруппированы по тому же самому принципу, что и поля типа checkbox.

Отметим, что в языке HTML определен еще один элемент оформления полей ввода — label. Он служит для создания надписей для этих полей, причем надпись является контентом элемента label, а привязка этого элемента к соответствующему элементу ввода осуществляется с помощью атрибута for этого элемента. Значением for должно быть имя (значение атрибута name) надписываемого элемента ввода. Как видно из листинга 8.1, у нас не возникло необходимости в использовании элемента label.

8.2. Сервлеты и обработка HTML-формы

Вернемся немного назад и еще раз рассмотрим листинг 6.1. Как можно заметить, принцип работы сервлетов довольно прост и заключается в формировании вывода на основе запроса, сделанного клиентом. При этом вывод представлен объектом out, являющимся экземпляром класса PrintWriter, его характеристики — объектом response (экземпляр класса HttpServletResponse), а запрос — объектом request (экземпляр класса HttpServletRequest).

Заметим, что в листинге 6.1 существует только один объект (out), создаваемый в рамках этой схемы. Объекты же response и request не создаются

разработчиком сервлета, а формируются средой, в которой запускается данный сервлет, а именно предоставляются для обработки в виде аргументов какой-либо из функций `doGet()` или `doPost()` в зависимости от использования метода запроса (get или post).

Если мы в качестве сервлета рассмотрим какой-либо XML-документ, преобразованный парсером в Java-код, а затем откомпилированный в байт-код, то можно заметить, что нет необходимости создавать даже и тот единственный объект, с помощью которого осуществляется управление выводом сервлета. Это объясняется тем, что Cocoon, преобразуя XML-документ в Java-сервлет, автоматически генерирует не только данный объект, но и некоторые другие, необходимые для корректной работы этого XML-документа.

Поэтому при создании XML-документа, который в результате обработки будет преобразован в сервлет, для манипулирования некоторыми слоями этого XML-документа нам достаточно будет объектов request и response.

Но здесь возникает несколько вопросов. Во-первых, для того чтобы получить доступ к объектам request и response, нужно ли нам описывать класс, расширяющий `HttpServlet`, а также функции `doGet()` и `doPost()`? Во-вторых, какие еще возможности предоставляет сервлет-контейнер помимо работы с этими двумя объектами?

Ответ на первый вопрос будет коротким. Инструментарий публикаций Cocoon 1.8.2 разрешает использовать объекты request и response в контенте документа, корневым элементом которого является `xsp:page`. После того, как подобный документ сформирован, Cocoon генерирует Java-код, который представляет собой код полноценного сервлета, т. е. с уже описанными всеми необходимыми составляющими. Иными словами, при условии корректности исходного документа, код сервлета генерируется парсером автоматически.

В качестве ответа на второй вопрос мы приведем список некоторых объектов, которые могут быть использованы в контенте XSP-страницы, а также несколько методов объекта request, полезных при разработке достаточно сложных серверных приложений.

Наряду с request и response в контенте XSP-страницы могут присутствовать следующие объекты:

- session — экземпляр класса Httpsession, который служит, как правило, для хранения информации, соответствующей текущей сессии пользователя;
- `servletContext` — экземпляр класса `ServletContext`, который нужен Для хранения и манипулирования информации о среде выполнения (контекста) сервлета.

Далее следует список остальных объектов с краткими комментариями:

- document — экземпляр класса org.w3c.Document, является Java-представлением документа, содержащего xsp:page;
- xspGlobal — объект, встроенный для совместимости с ранними версиями (номер версии меньше 2.2) стандарта, описывающего механизм сервлетов;
- xspNodeStack — экземпляр класса java.util.Stack, служащий для управления вложенностью элементов XML-документа;
- xspCurrentNode — экземпляр класса org.w3c.Node, служащий для Java-представления текущего узла XML-документа;
- xspParentNode — экземпляр класса org.w3c.Node, служащий для Java-представления родителя текущего узла XML-документа;
- xspParser — объект, представляющий собой DOM-парсер, реализованный в Cocoon и служащий для создания новых, а также синтаксического анализа внешних XML-документов.

Помимо этих объектов в API Cocoon определен класс XSPUtil, методы которого представляют собой некоторые полезные утилиты для обработки размеченного текста, URL-адресов, файлов и путей к ним, а также простых строк без разметки.

Ранее мы уже применяли один из методов объекта request, служащий для получения значения параметра HTTP-запроса, отправленного с использованием метода get. Этот метод называется getParameter(), и его аргументом является строка, представляющая собой имя того параметра, значение которого мы хотим получить. Разумеется, у объекта request этот метод не единственный. Но мы здесь не будем перечислять и описывать все методы этого объекта, а вместо этого приведем небольшой пример, с помощью которого можно получить представление о возможностях механизма обработки запросов, реализованного в сервлетах.

Листинг 8.2. Файл request.xml. Некоторые методы объекта request

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="request.xsl" type="text/xsl"?>

<xsp:page language="java"
  xmlns:xsp="http://www.apache.org/1999/XSP/Core">

<request>
```

```
<title>Некоторые методы объекта <code>request</codex/title>
 <list>
  <item>
   <method>request.getMethod()</method>
   <value><xsp:expr>request.getMethod()</xsp:expr></value>
   <note>Метод запроса</note>
  </item>
  <item>
   <method>request.getRequestURI()</method>
   <value><xsp:expr>request.getRequestURI()</xsp:expr></value>
   <note>Запрашиваемый URI</note>
  </item>
  <item>
   <method>request.getContextPath()</method>
   <value><xsp:expr>request.getContextPath()</xsp:exprX/value>
   <note>Путь к контексту сервлета</note>
  </item>
  <item>
   <method>request.getServletPath()</method>
   <value><xsp:expr>request.getServletPath()</xsp:expr></value>
   <note>Путь к сервлету</note>
  </item>
  <item>
   <method>request.getLocale()</method>
   <value><xsp:expr>request.getLocale()</xsp:expr></value>
   <note>Схема интернационализации</note>
  </item>
  <item>
   <method>request.getProtocol()</method>
   <value><xsp:expr>request.getProtocol()</xsp:expr></value>
   <note>Протокол запроса</note>
  </item>
  <item>
   <method>request.getQueryString()</method>
   <value><xsp:expr>request.getQuerystring()</xsp:exprX/value>
   <note>Строка параметров</note>
  </item>
```

```
    <item>
     <method>request.getContentLength()</method>
     <value><xsp:expr>request.getContentLength()</xsp:expr></value>
     <note>Размер содержимого запроса</note>
    </item>
    <item>
     <method>request.getServerName()</method>
     <value><xsp:expr>request.getServerName()</xsp:expr></value>
     <note>Имя сервера</note>
    </item>
    <item>
     <method>request.getServerPort()</method>
     <value><xsp:expr>request.getServerPort()</xsp:expr></value>
     <note>Используемый порт</note>
    </item>
    <item>
     <method>request.getRemoteAddr()</method>
     <value><xsp:expr>request.getRemoteAddr()</xsp:expr></value>
     <note>Адрес клиента</note>
    </item>
  </list>
</request>
</xsp:page>
```

Далее следует сценарий XSLT, с помощью которого методы, демонстрируемые листингом 8.2, а также результаты их работы и краткие комментарии располагаются в ячейках таблицы.

Листинг 8.3. Файл request.xsl. XSLT-составляющая демонстрации объекта request

```
<?xml version="1.0" encoding="koi8-r"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="request">
<xsl:processing-instruction name="cocoon-
format">type="text/html"</xsl:processing-instruction>
  <html>
```

```
  <head>
   <style>
    table, td {border-width:1px;border-style:solid;border-color:black}
    td {padding:5px}
   </style>
  </head>
  <body>
   <xsl:apply-templates/>
  </body>
 </html>
</xsl:template>

<xsl:template match="title">
 <h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match="list">
 <table cellspacing="0" cellpadding="0">
  <trxtd style="font-weight : bold; text-align : center">Метод</td>
      <td style="font-weight : bold; text-align : center">
         Возвращенное значение</td>
      <td style="font-weight :bold; text-align : center">
         Комментарий</td></tr>
   <xsl:apply-templates/>
 </table>
</xsl:template>

<xsl:template match="item">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="value | method | note">
  <td><xsl:apply-templates/></td>
</xsl:template>

</xsl:stylesheet>
```

В результате обработки листингов 8.2 и 8.3 получим представление, изображенное на рис. 8.2.

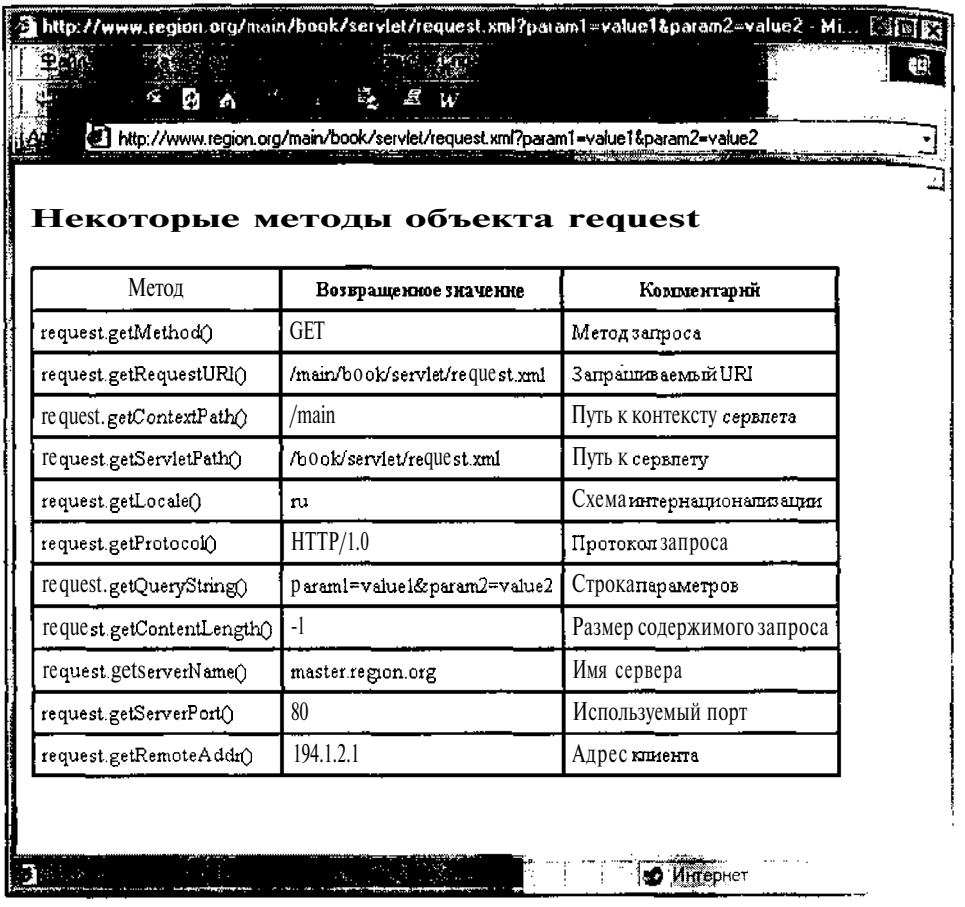


Рис. 8.2. Некоторые методы объекта request

8.3. Обработка параметров: метод *GET*

При отправке данных Web-формы методом get параметры передаются в строке URL, которая соответствует следующим правилам синтаксиса:

```
http://URL/URN?param1=value1&param2=value2
```

Здесь URL — местонахождение ресурса, URN — имя файла ресурса, после которых следует строка запроса (query string). Строка запроса отделяется от Internet-адреса знаком вопроса (?), а для параметров разделителем служи

амперсант (&). Каждый параметр состоит из имени параметра и его значения, отделенных друг от друга знаком равенства (=).

Обработка параметров, отправленных методом get, осуществляется с помощью метода getParameter о объекта request. Аргументом этого метода является имя параметра, а возвращаемая строка — его значением. Далее следует листинг, с помощью которого обрабатываются данные формы, представленной листингом 8.1.

Листинг 8.4. Файл form_handler.xml. Обработчик параметров

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type="xsp"?>

<xsp:page language="java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

<html>
 <field_text1>
   <b>Значение первого текстового поля:</b>
   <xsp:expr>request.getParameter("text1")</xsp:expr><br/>
 </field_text1>
 <field_text2>
   <b>Значение второго текстового поля:</b>
   <xsp:expr>request.getParameter("text2")</xsp:expr><br/>
 </field_text2>
 <field_text3>
   <b>Значение поля типа <code>password</code>:</b>
   <xsp:expr>request.getParameter("text3")</xsp:expr><br/>
 </field_text3>
 <field_select>
   <b>Значение селектора:</b>
   <xsp:expr>request.getParameter("selector")</xsp:expr><br/>
 </field_select>
 <field_file>
   <b>Значение поля типа <code>file</code>:</b>
   <xsp:expr>request.getParameter("file")</xsp:expr><br/>
 </field_file>
 <field_checkbox>
```

```
    <b>Три поля типа <code>checkbox</code>:</b><br/>
    <div style="padding-left:1em">
      <b>checkbox1 = </b>
       <xsp:expr>request.getParameter("ch1")</xsp:expr><br/>
      <b>checkbox2 = </b>
       <xsp:expr>request.getParameter("ch2")</xsp:expr><br/>
      <b>checkbox3 = </b>
       <xsp:expr>request.getParameter("ch3")</xsp:expr>
    </div>
  </field_checkbox>
  <field_radio>
   <b>Значение поля типа <code>radio</code>:</b>
   <xsp:expr>request.getParameter("r1")</xsp:expr><br/>
  </field_radio>
  <field_hidden>
   <b>Значение поля типа <code>hidden</code>:</b>
   <xsp:expr>request.getParameter("hValue")</xsp:expr><br/>
  </field_hidden>
</html>
</xsp:page>
```

В этом листинге мы также не воспользовались языком XSLT, но поместили представление каждого обработанного параметра в контент отдельного элемента, не принадлежащего языку HTML. Таким образом, при необходимости дополнительной обработки документа можно создать и подключить к нему соответствующий сценарий XSLT, содержащий правила шаблона для каждого из этих элементов.

Как видно из листинга 8.4, обработка параметров, отправленных методом get, не представляет сложной задачи. Для получения значения поля ввода, достаточно написать `request.getParameter("name_field")`, где *`name_field`* представляет собой значение атрибута name соответствующего элемента ввода.

Рассмотрим последовательно особенности обработки каждого параметра, в зависимости от типа поля ввода, посредством которого было отправлено его значение.

Первые три поля формы являются текстовыми, и для получения соответствующих им данных не требуется какой-либо специальной настройки формы и ее обработчика. Именами этих полей являются соответственно `text1`, `text2` и `text3`, и для получения значений одноименных параметров доста-

точно просто последовательно для каждого из них вызвать метод getParameter() с соответствующим аргументом. Значение, полученное посредством поля типа hidden, обрабатывается аналогично.

Для получения значения селектора, представленного в форме элементом select, также будет достаточным ПРОСТОЙ ВЫЗОВ Метода getParameter() С аргументом "selector". Но здесь следует иметь в виду, что если атрибут multiple этого элемента будет иметь значение true и пользователь выберет несколько альтернатив из списка, то строка запроса будет содержать несколько параметров с именем selector, значениями которых будут выбранные пользователем альтернативы.

Поэтому для обработки параметров в данной ситуации лучше всего воспользоваться методом request.getQueryString() И С помощью функций обработки строк получить все значения параметров с одинаковыми именами из строки запроса, полученной в результате вызова этого метода. Еще одним способом обработки данных селектора с возможностью одновременного выбора нескольких альтернатив является применение JavaScript. С помощью этого языка можно выяснить, какие именно альтернативы выбрал пользователь, затем сформировать дополнительный параметр, значением которого и будет информация о выбранных альтернативах. Заметим, что для отправки такого параметра на сервер лучше всего воспользоваться полем ВВОДа ТИПа hidden.

При отправке данных методом get значением поля типа file является полный путь и имя этого файла, сам файл при этом на сервер не посылается. Следовательно, для обработки значения, соответствующего полю этого типа, будет достаточно такого Же вызова request.getParameter(), как И ДЛЯ текстовых полей. Аргументом этого метода также будет значение атрибута name элемента input, соответствующего данному полю.

Параметр, соответствующий полю типа checkbox, формируется следующим образом. Если пользователь пометил (выбрал) это поле, то на сервер отправляется параметр с именем, эквивалентным имени данного поля, и значением on. В противном случае (если поле не выбрано) параметр на сервер Не Отправляется. ПОЭТОМУ результатом вызова request.getParameter() ДЛЯ Получения значения данного поля будет либо on, либо пустая строка (результат преобразования в строку null).

Как мы уже говорили ранее, для реализации возможности выбора пользователем одной из нескольких альтернатив, представленных набором полей типа radio, необходимо, чтобы каждый элемент ввода из этой группы имел одно и то же имя, заданное атрибутом name. Поэтому при отправке на сервер информации о выбранном значении, данной группой генерируется только один параметр с именем, эквивалентным имени ка-

ждого из элементов ввода из этой группы (r1 в листинге 8.4), и значением, эквивалентным значению атрибута value того элемента, который соответствует выбранному полю. Значением по умолчанию для value является on.

8.3.1. Обработка русскоязычных параметров

При обработке браузером MSIE 5.5 русскоязычных параметров, переданных Web-формой, иногда могут возникать некоторые проблемы, связанные с кодировками передаваемого текста. В общем, браузер справляется с передачей русскоязычных параметров в строке запроса, если они заданы в кодировке отличной от Unicode, и форма, а также ее обработчик настроены соответствующим образом.

Но проблема заключается в том, что при отправке данных посредством Web-формы браузер сам формирует строку запроса. И даже в том случае, когда отключена его опция "Всегда отправлять адреса URL как UTF-8", русскоязычные параметры могут быть отправлены (даже всегда отправляются) именно в этой кодировке.

Мы можем предложить два способа разрешения этой ситуации. Первый заключается в самостоятельном формировании разработчиком строки запроса с помощью языка JavaScript. Этот способ, на наш взгляд, является достаточно неэффективным, поэтому мы сразу перейдем ко второму, который предполагает передачу параметров в кодировке Unicode.

К сожалению, нам не известен диалект Unicode, который применяется MSIE 5.5 для кодировки параметров, поэтому мы можем предложить свой собственный перекодировщик, выполняющийся на стороне сервера. В его основе лежит предположение о том, что браузером используется некий 16-битный диалект Unicode.

Далее следует листинг, содержащий Java-код данного перекодировщика.

Листинг 8.5. Файлform_handler.xml. Обработка русскоязычных параметров.

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type="xsp"?>

<xsp:page language="java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

<html>
```

```
<xsp:logic>
class UTFParam (
  public String getParameter(String param) {
  String res="";
  try {
   StringTokenizer st =
     new StringTokenizer(new String(param.getBytes("UTF-16")),"\0") ;
   st.nextToken();
   while (st.hasMoreTokens()) res+=st.nextToken();
   } catch (Exception e) ( System.out.println(e); }
   return res;
 }
};
UTFParam utfp = new UTFParam();
</xsp:logic>
<field_text1>
  <b>Значение первого текстового поля:</b>
  <xsp:expr>utfp.getParameter(request.getParameter("text1"))</xsp:expr>
  <br/>
</field_textl>
<field_text2>
  <b>Значение второго текстового поля:</b>
  <xsp:expr>utfp.getParameter(request.getParameter("text2"))</xsp:expr>
  <br/>
</field_text2>
<!-- Остальные параметры обрабатываются аналогично двум первым, т. е.
вместо request.getParameter(имя параметра) используется
utfp.getParameter(request.getParameter(имя параметра)) -->
</html>
</xsp:page>
```

Принцип работы перекодировщика заключается в том, что полученное значение параметра после его преобразования в UTF-16 представляет собой набор лексем, разделенных нулем (символом с кодом 0), без первой лексемы. Таким образом, для получения значения русскоязычного параметра нам достаточно преобразовать его исходный текст в UTF-16 и затем выбрать из преобразованного значения все лексемы, разделенные нулями, начиная со второй. Значением параметра при этом будет результат склейки всех этих лексем.

На рис. 8.3 проиллюстрирована работа перекодировщика form_handler.xml, представленного листингом 8.5.

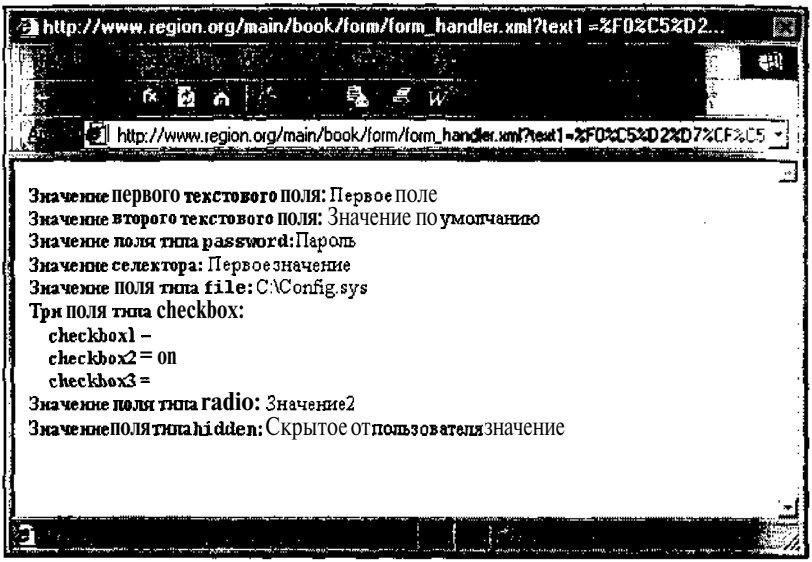


Рис. 8.3. Результат обработки русскоязычных параметров

8.4. Обработка параметров: метод *POST*

Параметры, переданные методом post, обрабатываются несколько сложнее, чем при методе get. Сложность заключается в том, что они передаются отдельным HTTP-запросом, тело которого помимо самих параметров содержит еще и некоторую служебную информацию. Принцип обработки параметров в данном случае заключается в получении текста HTTP-запроса, из которого затем извлекаются их значения.

Java-код, служащий для обработки данного HTTP-запроса, слишком велик, чтобы помещать его непосредственно в документ. Кроме того, если Web-приложение состоит из нескольких XML-документов, обрабатывающих параметры, отправленные методом post, то дублирование этого кода в каждом таком документе будет избыточным. Поэтому мы создадим внешний по отношению к XML-документам код Java, отвечающий за обработку параметров, а затем подключим его к нашему документу с использованием элемента `xsp:structure`, определенном в Cocoon.

Далее следуют три листинга, первые два из которых представляют собой классы Java, отвечающие за обработку параметров, а третий — сам обработчик формы.

Листинг 8.6. Файл name_value.java. Вспомогательный класс

```
package param_utils;

public class name_value {
 public String name; /* имя параметра */
 public String value; /* значение параметра */
 public name_value(String n, String v)
        { name=n; value=v; }}
```

В этом листинге описан вспомогательный класс name_value, который используется классом Params. Он служит для описания абстрактной пары имя/значение, которая в следующем листинге будет интерпретироваться как имя и значение параметра.

Листинг 8.7. Params.java. Обработчик текста HTTP-запроса

```
package param_utils;

public class Params {
    public name_value[] params;
    int sz;

    public Params(StringBuffer sb, String b) {
     String data = new String(sb); /* текст HTTP-запроса */
     int size = 0;
     /* bIndex — начальная позиция 1-го параметра */
     int bIndex=data.indexOf(b, 0)+5;
     /* eIndex — конечная позиция 1-го параметра */
     int eIndex=data.indexOf(b, bIndex);
     /* вычисляем size — количество параметров */
     while (elndex != -1 && elndex > blndex)
       {bIndex = eIndex+5; elndex = data.indexOf(b, bIndex); size++;}
     sz = size;
     params = new name_value[size];/* создаем массив параметров */
     int length = b.length(); /* вычисляем длину границы */
     /*далее вычисляются:
       blndex — начальная позиция n-го параметра
       elndex — конечная позиция n-го параметра
       bInd — начальная позиция имени параметра
```

```
        eInd — конечная позиция имени параметра */
      bIndex = data.indexOf(b, 0) + length;
      eIndex = data.indexOf(b, bIndex);
      int bInd = data.indexOf('"', blndex) + 1;
      int eInd = data.indexOf('"', bInd);
      /*далее в цикле заполняется массив параметров с использованием
        blndex, eIndex, bInd, eInd для вырезания из текста HTTP-запроса
        (data) имен и значений параметров */
      for (int i = 0; i < size; i++)
          { params[i] = new name_value(data.substring(bInd, eInd),
                                       data.substring(eInd+1, eIndex));
            blndex = data.indexOf(b, elndex) + length;
            elndex = data.indexOf(b, bIndex);
            bInd = data.indexOf('"', blndex) +1;
            eInd = data.indexOf('"', bInd); }}

    public String getValueByName(String name) {
      /* поиск параметра по имени */
      for (int i = 0; i < sz; i++)
          if (params[i].name.equals(name)) return params[i].value;
      return "";}
}
```

Далее следует обработчик формы, использующий методы класса Params для получения значения параметров запроса.

Листинг 8.8. Файл form_handler.xml. Обработчик формы

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type="xsp"?>

<xsp:page language="java"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

  <xsp:structure>
   <xsp:include>param_utils.*</xsp:include>
  </xsp:structure>

<html>
 <xsp:logic>
```

```
/* получаем входной поток сервлета */
ServletInputStream is = request.getInputStream();
/* создаем экземпляр класса BufferedReader для чтения
   из этого потока */
BufferedReader br =
  new BufferedReader(new InputStreamReader(is,"UTF8"));
/* в переменной sBuf будет храниться текст HTTP-запроса */
StringBuffer sBuf = new StringBuffer();
String 1;
try {
  1 = br.readLine(); /* читаем одну строку из потока */
  while (l!=null)    (
    sBuf.append(l); /* добавляем прочитанную строку к sBuf */
    1 = br.readLine();/* читаем следующую строку */}
    } catch (IOException e) {System.out.println(e);
<xsp:content>
 <xsp:expr>e</xsp:expr>
</xsp:content>};
/* считываем в header заголовок HTTP-запроса */
String header = request.getHeader("Content-Type");
/* вычисляем index — начальная позиция последней
   встретившейся в header подстроки boundary= */
int index = header.lastIndexOf("boundary=");
/* вырезаем значение границы из header */
String boundary = header.substring(index + 9);
/* формируем реальное значение границы */
boundary = "--" + boundary;

I* создаем экземпляр Params — класса обработки параметров */
  Params p = new Params(sBuf, boundary);
</xsp:logic>
<field_text1>
  <b>Значение первого текстового поля:</b>
 <xsp:expr>p.getValueByName("text1")</xsp:expr><br/>
</field_text1>
<field_text2>
  <b>Значение второго текстового поля:</b>
  <xsp:expr>p.getValueByName("text2")</xsp:expr><br/>
</field_text2>
```

```
<field_text3>
  <b>Значение поля типа <code>password</code>:</b>
  <xsp:expr>p.getValueByName("text3")</xsp:expr><br/>
</field_text3>
<field_select>
  <b>Значение селектора:</b>
  <xsp:expr>p.getValueByName("selector")</xsp:expr><br/>
</field_select>
<field_file>
  <b>Значение поля типа <code>file</code>:</b><br/>
  <xsp:logic>
   String field_file = p.getValueByName("file");
   int bIndex = field_file.indexOf("filename=")+10;
   int eIndex= field_file.indexOf(' " ' ,bIndex);
   String file_name = field_file.substring(bIndex, eIndex);
   String file_content = field_file.substring(eIndex+24);
  </xsp:logic>
  <div style="padding-left:1em">
    <b>Текст, находящийся между <code>"file="</code>
       и следующим параметром:</b><br/>
       <xsp:expr>p.getValueByName("file")</xsp:expr><br/>
    <b>Файл:</b> <xsp:expr>file_name</xsp:expr><br/>
    <b>Содержимое файла:</b><br/>
    <xsp:expr>file_content</xsp:expr>
  </div>
</field_file>
<field_checkbox>
  <b>Три поля типа <code>checkbox</code>:</b><br/>
  <div style="padding-left:1em">
    <b>checkbox1 = </b><xsp:expr>p.getValueByName("ch1")</xsp:expr><br/>
    <b>checkbox2 = </b><xsp:expr>p.getValueByName("ch2")</xsp:expr><br/>
    <b>checkbox3 = </b><xsp:expr>p.getValueByName("ch3")</xsp:expr>
  </div>
</field_checkbox>
<field_radio>
 <b>Значение поля типа <code>radio</code>:</b>
 <xsp:expr>p.getValueByName("r1")</xsp:expr><br/>
</field_radio>
<field_hidden>
```

```
  <b>Значение поля типа <code>hidden</code>:</b>
  <xsp:expr>p.getValueByName("hValue"</xsp:expr><br/>
</field_hidden>
  <b>Содержимое HTTP-запроса (<code>sBuf</code>):</b><br/>
  <xsp:expr>sBuf.toString()</xsp:expr>
</html>
</xsp:page>
```

Листинги 8.6 и 8.7 образуют пакет классов Java с именем `param_utils`. Этот пакет мы и подключаем в листинге 8.8 с помощью элемента `xsp:structure`. Отметим, что путь к файлам, содержащим Java-классы, должен быть определен с помощью переменной среды CLASSPATH. После этого данные классы можно использовать в XML-документах, а также в других Java-классах без их предварительной компиляции.

После обработки файла form_handler.xml мы получим результат, изображенный на рис. 8.4.

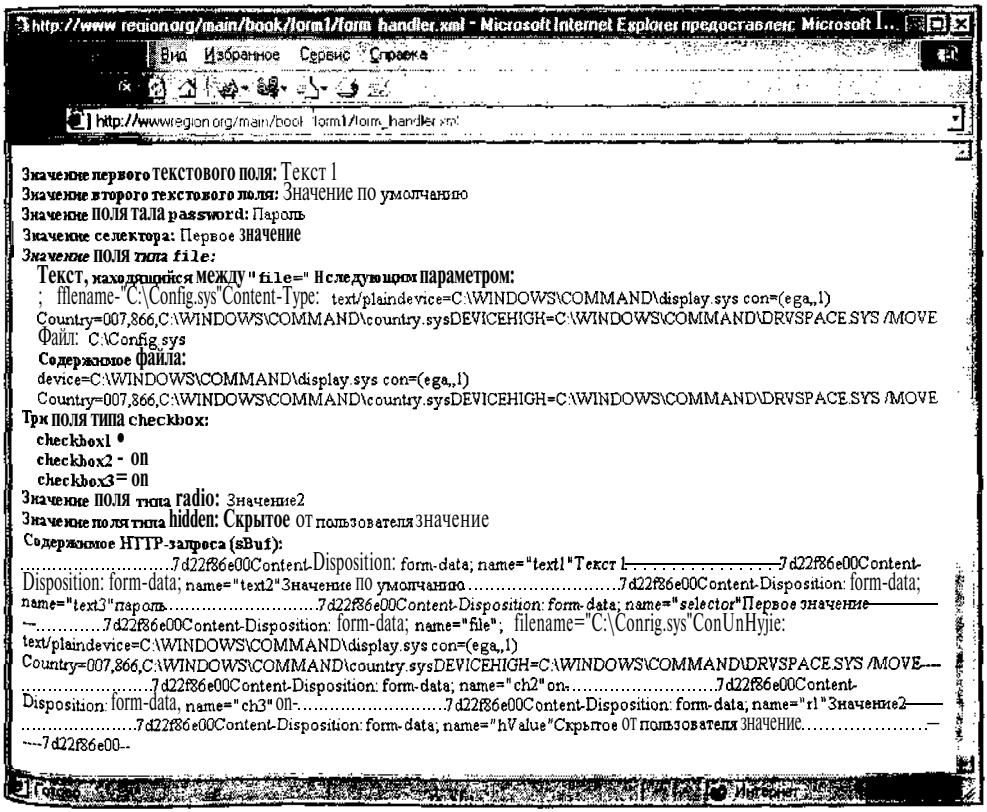


Рис. 8.4. Результат обработки параметров

Далее мы приводим алгоритм обработки параметров, реализованный в листингах 8.6, 8.7 и 8.8.

1. В контенте элемента `xsp:logic`, который является первым непосредственным потомком элемента html, мы с помощью класса `ServletInputStream` получаем входной поток сервлета, т. е. поток данных от клиента, представленный тем HTTP-запросом, который нам предстоит обработать.

2. После того, как входные данные помещены в переменную `sBuf` (ее содержимое отображается в нижней части рис. 8.4), мы получаем значение границы, которая представляет собой строку, служащую разделителем между параметрами запроса. Значение границы вырезается из заголовка HTTP-запроса, получаемого с помощью вызова `request.getHeader("Content-Type")`. Его результатом, помещенным в переменную header, в данном случае будет:

   ```
   multipart/form-data; boundary=-----------------------------7d22f86e00
   ```

   Если сравнить значения границы в заголовке и в содержимом переменной `sBuf`, то можно заметить, что последнее отличается от первого тем, что к нему добавлено слева два символа "-" (без кавычек). Поэтому для получения значения границы мы вырезаем из переменной header все то, что следует сразу после `boundary=` и добавляем слева `--`. Отметим, что полученное значение границы уникально для каждого HTTP-запроса, а ее величина практически исключает вероятность совпадения значения этой границы со значением или его частью какого-либо из параметров.

3. Поместив значения реального разделителя параметров в переменную boundary, мы создаем объект p, являющийся экземпляром класса `Params`, используя в качестве аргументов его конструктора переменные sBuf и boundary. Перейдем к рассмотрению листинга 8.7.

4. При создании экземпляра класса Params вызывается его конструктор, в котором мы создаем массив параметров, представляющих собой пары имя/значение. Для описания этих пар служит вспомогательный класс `name_value`. Последовательно обрабатывая находящийся между разделителями текст, мы сначала подсчитываем общее количество параметров и создаем массив с равной этому количеству длиной. После этого, опираясь на местоположение кавычек как разделителей имени и значения параметра, мы заполняем этот массив вырезанными парами имя/значение. Формат параметров при их передаче методом post точно такой же, как и для метода get (пале = *"value")* с тем отличием, что значение заключено в кавычки. Исключение составляет параметр, отправляемый посредством элемента ввода типа `file`, но об этом позднее.

5. После того, как массив параметров создан, к нему можно обращаться для получения значения того или иного параметра. Для этих целей в листинге 8.7 описан метод `getValueByName()`, который, принимая имя параметра в качестве своего аргумента, отыскивает в массиве пару имя/значение, в которой имя соответствует этому аргументу, и возвращает значение из этой пары. Возвращаясь к листингу 8.8, мы можем увидеть вызовы этого метода для каждого из параметров.

При отправке методом post параметра, порожденного элементом ввода типа `file`, на сервер передается не только путь и имя этого файла, но и его содержимое. Обработка этого параметра представлена контентом элемента `field_file`. Здесь мы, опираясь на строку `filename=` и символ кавычки (") как на разделители, вырезаем из текста, соответствующего данному параметру, путь/имя и содержимое файла. Для наглядности мы на рис. 8.4 представили текст всего параметра целиком, а также вырезанные из него значения.

Внимание

При отправке файла на сервер в начале значения параметра типа **file** содержится MIME-тип содержимого файла. Его значение можно увидеть после строки `Content-Type:`. В нашем примере мы предполагали, что на сервер отсылаются HTML-файлы, и соответствующим им значением MIME-типа было `text/html`. В случае текстового файла без разметки MIME-типом будет **text/plain** (см. рис. 8.4), если же файл не текстовый, то наиболее вероятным для него значением MIME-типа будет application/octet-stream. Поэтому при вырезании из значения параметра типа **file** содержимого этого файла, нужно определить ту позицию, с которой начинается это содержимое. В листинге 8.8 содержимое файла (`file_content`) начинается с 24-й позиции, отсчитанной от закрывающей кавычки имени файла (`file_name`), что соответствует MIME-типу `text/html`. Для вырезания контента типа **text/plain** следует вместо 24 использовать 25, а при MIME-типе `application/octet-stream` — 39.

Отметим, что при получении входного потока сервлета у нас была кодировка UTF8 (см. листинг 8.7, создание экземпляра класса `InputStreamReader`), поэтому не имеет значения, являются значения параметров русскоязычными или нет. При отправке параметров методом post MSIE 5.5 формирует HTTP-запрос вполне корректно. Но это не касается содержимого файла, отправляемого на сервер. Необходимо точно знать его кодировку и при вызове конструктора входного потока указать ее значение в качестве значения аргумента enc этого конструктора. Если кодировка будет указана неверно, то будет сгенерировано исключение `MalformedURLException` и данные, содержащиеся в файле, сервером обработаны не будут.

8.5. Обработка данных на стороне клиента

В этом подразделе мы опишем сетевой апплет, который будет нами использоваться практически в каждом XML-документе при разработке элементов управления. Единственной функцией этого апплета является загрузка заданного ресурса с удаленного узла. При этом ресурс задается с помощью URI и в наших разработках представляет собой некоторый HTML-код, который после загрузки с сервера встраивается в загрузивший его документ. Хотя, разумеется, загружаемым ресурсом может быть файл с любым содержимым.

Предположим, что нам необходимо, чтобы в некоторой части Web-страницы отображался текст, соответствующий каким-либо действиям пользователя. Пусть, например, пользователь, выбирая день недели, должен получать телевизионную программу на этот день. Проще всего такую задачу решить с помощью фреймов, но фреймы обладают рядом недостатков, которые мы обсудим в следующей главе, поэтому здесь мы предлагаем другое решение вышеописанной задачи.

Представим себе внешний вид будущей Web-страницы, содержащей программу телепередач на неделю. Она будет состоять, как минимум, из двух областей: области управления, в которой располагаются объекты, служащие для выбора дня недели (это могут гиперссылки, кнопки и т. п.), а также той области, в которой находится собственно программа телепередач. Мы не будем здесь проектировать дизайн этой Web-страницы. Предположим, что мы это уже сделали, и теперь займемся логическим слоем соответствующего ей Web-приложения, т. е. обработкой действий пользователя и загрузкой информации с удаленного узла.

Для пользователя было бы очень удобно, если бы он, щелкая кнопкой мыши на элементе управления, соответствующем какому-либо дню недели, сразу бы получал информацию о телепередачах на этот день. Точно так же, и для разработчика будет естественным написать код, суть которого сводится к простому правилу: "пользователь нажал кнопку мыши — соответствующий код загружается". Это может быть что-нибудь вроде:

```
<button onclick = "load(день_недели, область-приемник)">
  День недели
</button>
```

Здесь элемент button служит для представления кнопки с надписью, соответствующей какому-либо дню недели, аргумент область-приемник — той области (или боксу), в которой появится текст телепрограммы, и аргумент день_недели — дню недели, для которого требуется загрузить телепрограмму.

Мы не будем останавливаться на обработке события, представленного здесь атрибутом onclick, этот механизм мы опишем в следующей главе

и сразу перейдем к обшим характеристикам функции load(), которая должна загружать информацию с удаленного узла. Так как в JavaScript отсутствует инструментарий для установки связи с удаленным узлом и загрузки с него данных, то в теле функции load() должен присутствовать вызов функции, написанной на Java, потому что в Java такой инструментарий определен.

На стороне клиента код Java может присутствовать только в виде апплета, если, конечно, на клиентской машине не установлена виртуальная машина этого языка. А апплет, в свою очередь, представляет собой класс, определенный разработчиком и расширяющий класс `java.Applet`.

Для того чтобы вызвать метод класса апплета из сценария JavaScript, нужно воспользоваться следующим синтаксисом:

```
class_name.method_name(arg1, arg2,...)
```

где `class_name` — имя класса апплета, method_name — имя вызываемого метода и `arg1`, arg2 и т. д. — список его аргументов. Далее следует код сетевого апплета, с помощью которого производится установка соединения с удаленным узлом, а также загрузка с него данных.

Листинг 8.9. UIDispatcher.java. Сетевой апплет

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

public class UIDispatcher extends Applet {

 public String getHTML(String url, String query) {
  StringBuffer sBuf = new StringBuffer();
  try {
    URL GetMe = new URL(url+"?"+query);
    URLConnection URLc = GetMe.openConnection();
    URLc.setUseCaches(false);
    InputStream is  = GetMe.openStream();
    BufferedReader br = new
                  BufferedReader(new InputStreamReader(is,"UTF8"));
    String 1 = br.readLine();
    while (1!=null)       {
      sBuf.append(1);
```

8.5. Обработка данных на стороне клиента

В этом подразделе мы опишем сетевой апплет, который будет нами использоваться практически в каждом XML-документе при разработке элементов управления. Единственной функцией этого апплета является загрузка заданного ресурса с удаленного узла. При этом ресурс задается с помощью URI и в наших разработках представляет собой некоторый HTML-код, который после загрузки с сервера встраивается в загрузивший его документ. Хотя, разумеется, загружаемым ресурсом может быть файл с любым содержимым.

Предположим, что нам необходимо, чтобы в некоторой части Web-страницы отображался текст, соответствующий каким-либо действиям пользователя. Пусть, например, пользователь, выбирая день недели, должен получать телевизионную программу на этот день. Проще всего такую задачу решить с помощью фреймов, но фреймы обладают рядом недостатков, которые мы обсудим в следующей главе, поэтому здесь мы предлагаем другое решение вышеописанной задачи.

Представим себе внешний вид будущей Web-страницы, содержащей программу телепередач на неделю. Она будет состоять, как минимум, из двух областей: области управления, в которой располагаются объекты, служащие для выбора дня недели (это могут гиперссылки, кнопки и т. п.), а также той области, в которой находится собственно программа телепередач. Мы не будем здесь проектировать дизайн этой Web-страницы. Предположим, что мы это уже сделали, и теперь займемся логическим слоем соответствующего ей Web-приложения, т. е. обработкой действий пользователя и загрузкой информации с удаленного узла.

Для пользователя было бы очень удобно, если бы он, щелкая кнопкой мыши на элементе управления, соответствующем какому-либо дню недели, сразу бы получал информацию о телепередачах на этот день. Точно так же. и для разработчика будет естественным написать код, суть которого сводится к простому правилу: "пользователь нажал кнопку мыши — соответствующий код загружается". Это может быть что-нибудь вроде:

```
<button onclick = "load(день_недели, область-приемник)">
  День недели
</button>
```

Здесь элемент button служит для представления кнопки с надписью, соответствующей какому-либо дню недели, аргумент область-приемник — той области (или боксу), в которой появится текст телепрограммы, и аргумент день_недели — дню недели, для которого требуется загрузить телепрограмму.

Мы не будем останавливаться на обработке события, представленного здесь атрибутом onclick, этот механизм мы опишем в следующей главе

и сразу перейдем к общим характеристикам функции load(), которая должна загружать информацию с удаленного узла. Так как в JavaScript отсутствует инструментарий для установки связи с удаленным узлом и загрузки с него данных, то в теле функции load() должен присутствовать вызов функции, написанной на Java, потому что в Java такой инструментарий определен.

fla стороне клиента код Java может присутствовать только в виде апплета, если, конечно, на клиентской машине не установлена виртуальная машина этого языка. А апплет, в свою очередь, представляет собой класс, определенный разработчиком и расширяющий класс java.Applet.

Для того чтобы вызвать метод класса апплета из сценария JavaScript, нужно воспользоваться следующим синтаксисом:

```
class_name.method_name(arg1, arg2,...)
```

где class_name — имя класса апплета, method_name — имя вызываемого метода и argl, arg2 и т. д. — список его аргументов. Далее следует код сетевого апплета, с помощью которого производится установка соединения с удаленным узлом, а также загрузка с него данных.

Листинг 8.9. UIDispatcher.java. Сетевой апплет

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

public class UIDispatcher extends Applet {

 public String getHTML(String url, String query) {
  StringBuffer sBuf = new StringBuffer();
  try {
    URL GetMe = new URL(url+"?"+query);
    URLConnection URLc = GetMe.openConnection();
    URLc.setUseCaches(false);
    InputStream is  = GetMe.openStream();
    BufferedReader br = new
                  BufferedReader(new InputStreamReader(is,"UTF8"));
    String 1 = br.readLine();
    while (l!=null)      {
      sBuf.append(l);
```

```
    1 = br.readLine(); }
    } catch (IOException e) { System.out.println(e); };
  return sBuf.toString();
  }
}
```

Как видно из листинга 8.9, методом класса UIDispatcher (класс апплета), выполняющим эти действия, является `getHTML()`, аргументами которого являются URL ресурса и строка запроса (querystring), служащая для передачи удаленному узлу параметров загрузки данных.

Рассмотрим принцип работы этого метода. Сначала с помощью класса `java.net.URL` мы создаем Java-представление URL-адреса. Далее, с помощью класса `java.net.URLConnection` мы устанавливаем соединение с удаленным узлом. Затем, с помощью этого же класса открываем входной поток (класс `InputStream`) для чтения файла ресурса и считываем этот файл по той же самой схеме, которая была реализована нами при чтении текста HTTP-запроса в листинге 8.8.

Теперь нам осталось только написать HTML-код, использующий вызов этого метода для загрузки программы телепередач. Этот код может выглядеть, например, так:

```
<html>

  <head>
    <script>
     function load(day, obj) {
      switch (day) {
       case 'Понедельник' : obj.innerHTML =
            UIDispatcher.getHTML('http://www.some_host.org',
                                 'day=Monday');
            break;
     ...
       case 'Воскресенье' : obj.innerHTML =
            UIDispatcher.getHTML('http://www.some_host.org',
                                 'day=Sunday');
        }
     }
    </script>
  </head>
<body>
```

```
<applet id="UIDispatcher" code="UIDispatcher.class"
        style="height:0px;width:0px;visibility:hidden">
</applet>
<div id="text_container">
  <!-- Здесь будет располагаться текст телепрограммы -->
</div>
<button onclick = "load('Понедельник', text_container)">
  Понедельник
</button>
...
<button onclick = "load('Воскресенье', 'text_container')">
  Воскресенье
</button>
</body>
</html>
```

Теперь, если в код загружаемого файла вставить сценарий JavaScript, позволяющий в зависимости от параметров строки запроса предоставлять ту или иную часть этого файла, то при каждом вызове функции load о может возвращаться та его часть, которая соответствует заданному дню недели. Но в этом случае возникает следующая проблема: если телепрограммы для всех дней недели хранятся в одном файле, то этот файл сначала будет целиком загружен клиенту, и только после этого выполнится сценарий JavaScript, который выберет текст программы, соответствующий заданному дню.

Если переформулировать задачу так, чтобы телепрограммы предоставлялись в зависимости не от дня недели, а, скажем, от дня месяца или года, то избыточность загружаемых данных становится очевидной. Поэтому будет естественным загружать файл, форматом которого является не HTML, а XML.

Действительно, применяя различные директивы XSLT и основываясь на значениях параметров строки запроса, можно трансформировать исходный XML-документ так, что в результате мы получим требуемый код, который в точности соответствует запросу клиента. Этого можно добиться, например, используя в XML-файле элементы с именами, соответствующими дням недели (Sunday, Monday и т. д.), а в сценарии XSLT в правиле шаблона для корневого элемента прописать директиву

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

чем и будет достигнуто решение нашей задачи.

Далее в кодах элементов управления мы продемонстрируем те возможности, которые могут быть реализованы с помощью вышеописанного апплета. А пока скажем только, что он является одной из важнейших составляющих инструментария построения Web-интерфейса.

8.6. Операции над файловой системой на стороне клиента

Иногда возникает необходимость в обработке относительно большого объема данных на стороне клиента. Если этих данных не слишком много, то можно воспользоваться механизмом Cookies, но что делать в том случае, когда не только на сервере, но и на машине клиента хранятся целые документы?

Примером может послужить ситуация, когда некоторая часть документов изготавливается пользователем локально и на сервер пересылается уже в почти готовом виде, т. е. после некоторой незначительной обработки. Предположим, например, что пользователь в каком-либо визуальном HTML-редакторе создает документы, структура которых жестко не определена. После того, как такой документ сформирован, пользователь помечает в его тексте некоторые слова как ключевые, причем список этих ключевых слов находится на сервере. Таким образом, должна быть обеспечена возможность загрузки локально файла на Web-страницу, загруженную с сервера и содержащую инструментарий пометки ключевых слов.

Именно для разрешения такой ситуации мы предлагаем встроенный в MSIE 5.5 компонент ActiveX, под названием `FileSystemObject`. С его помощью можно производить операции как над объектами файловой системы в целом (файлы, папки, диски), так и над содержимым отдельно взятого файла.

Однако следует иметь в виду, что для работоспособности данного компонента необходимо соответствующим образом настроить режим безопасности MSIE 5.5. А именно, отредактировать секцию **Элементы ActiveX и модули подключения,** для доступа к которой нужно выбрать команду **Сервис | Свойства обозревателя** и далее в появившемся диалоговом окне выбрать вкладку **Безопасность.** Следует быть очень осторожным при редактировании настроек этой секции, т. к. разрешение доступа к файловой системе клиентской машины влечет за собой значительное увеличение вероятности успеха ее взлома.

Итак, перед тем, как приступить к операциям над файловой системой клиента, нам нужно создать объект типа FileSystemObject. Делается это с помощью следующей конструкции:

```
var fso = new ActiveXObject("Scripting.FileSystemObject")
```

после чего можно приступать к использованию методов и свойств этого объекта. Заметим, что инициализация объекта FileSystemObject происходит только один раз, вне зависимости от количества попыток разработчика создать объект этого типа.

Объектную модель FileSystemObject образуют следующие объекты и коллекции.

- Непосредственно сам FileSystemObject как основной объект.
- Объект Drive, служащий для получения различной информации о дисках.
- Коллекция объектов Drives, являющаяся представлением всех определенных в системе дисков.
- Объект File, реализующий операции над файлами такие, как создание, удаление и перемещение файла, а также служащий для обработки различных характеристик файла (путь, имя, размер и т. п.).
- Коллекция Files, служащая для представления списка файлов, содержащихся в файле.
- Объект Folder, содержащий методы и свойства для операций над папками, а также для получения их характеристик.
- Коллекция Folders, служащая для представления списка папок, содержащихся в другой папке.
- Объект TextStream, реализующий операции чтения и записи над текстовыми файлами.

8.6.1. Операции над дисками

Для получения информации о диске необходимо воспользоваться методом getDrive() объекта FileSystemObject, который, принимая в качестве аргумента путь к диску, возвращает объект типа Drive. В табл. 8.1 содержится описание свойств этого объекта, каждое из которых соответствует какой-либо характеристике диска.

Таблица 8.1. *Свойства объекта FileSystemObject*

Свойство	Характеристика диска
TotalSize	Общий размер в байтах
AvailableSpace	Размер свободного места в байтах
DriveLetter	Буква диска
DriveType	Тип диска: 1 — переносной (removable), 2 — непереносной (fixed), 3 — сетевой (network), 4 - CDROM, 5 - RAM-диск

Таблица 8.1 (окончание)

Свойство	Характеристика диска
SerialNumber	Серийный номер
FileSystem	Тип файловой системы (FAT, FAT32, NTFS и т. п.)
IsReady	Доступен ли для использования
VolumeName	Метка тома
ShareName	Сетевое имя
Path	Путь
RootFolder	Корневая папка

Для проверки, существует ли диск с заданным именем, предназначен метод DriveExists () объекта Filesystemobject, принимающий в качестве аргумента имя диска и возвращающий true, если такой диск существует, и false, если — нет.

Далее следует небольшой пример, иллюстрирующий получение некоторых характеристик диска с помощью объектов Filesystemobject и Drive.

```
<html>
 <body>
  <script>
  var fso = new ActiveXObject("Scripting.FileSystemObject");
  var drv = fso.getDrive("c:\\");
  document.write("<b> Всего байтов: </b>");
  document.write(drv.TotalSize);
  document.write("<br><b> Свободных байтов: </b>");
  document.write(drv.AvailableSpace);
  document.write("<br><b> Тип диска: </b>");
  document.write(drv.DriveType);
  document.write("<br><b> Серийный номер: </b>");
  document.write(drv.SerialNumber);
  document.write("<br><b> Тип файловой системы: </b>");
  document.write(drv.FileSystem);
  </script>
 </body>
</html>
```

8.6.2. Операции над папками

Для получения объекта, представляющего какую-либо папку, нужно воспользоваться методом `getFolder()` объекта FileSystemObject, по аналогии с `getDrive()`, описанным выше. Для манипулирования папками и их характеристиками можно воспользоваться либо тем значением (объект типа `Folder`), которое будет возвращено в результате вызова `getFolder()`, либо специальными методами объекта FileSystemObject. В табл. 8.2 содержится описание тех возможностей, которые предоставляют Folder и `FileSystemObject` для управления папками и их характеристиками.

Таблица 8.2. *Методы, реализующие операции над папками*

Операция над папкой	Метод
Создание	`FileSystemObject.CreateFolder`
Удаление	`Folder.Delete` или `FileSystemObject.DeleteFolder`
Перемещение	`Folder.Move` или `FileSystemObject.MoveFolder`
Копирование	`Folder.Copy` или `FileSystemObject.CopyFolder`
Получение имени	`Folder.Name`
Выяснение существования	`FileSystemObject.FolderExists`
Получение имени Родительской папки	`FileSystemObject.GetParentFolderName`
Получение Folder-представления системной папки	`FileSystemObject.GetSpecialFolder`

Здесь аргументом каждого метода из перечисленных в правой колонке таблицы будет значение пути, соответствующее той операции, которую выполняет данный метод. Под системной понимается одна из трех папок:

1. Папка Windows. Соответствующим ей возвращаемым значением является 0.
2. Системная папка (то есть та, в которой хранятся библиотеки, шрифты и Драйверы устройств). Соответствующим ей возвращаемым значением является 1.
3. Временная папка (то есть предназначенная для хранения временных файлов). Соответствующим ей возвращаемым значением является 2.

Далее приводится табл. 8.3, содержащая описание свойств объекта Folder, за исключением тех из них, которые определены также и для объекта File. Последние мы опишем в следующем подразделе.

Таблица 8.3. *Некоторые свойства объекта* Folder

Свойство	Значение
Files	Коллекция файлов (объект Files), содержащихся в данной папке
IsRootFolder	Является ли данная папка корневой (true, если да, и false, если нет)
SubFolders	Коллекция папок (объект Folders), вложенных в данную

8.6.3. Операции над файлами

Как уже говорилось ранее, для представления файлов служит объект File, который можно получить с помощью вызова метода getFile() объекта FileSystemObject. Далее мы опишем по отдельности свойства и методы этого объекта.

Существуют три способа создания текстового файла. Каждый из этих способов подразумевает использование отдельного метода какого-либо из объектов FileSystemObject ИЛИ File. Рассмотрим ИХ ПО порядку.

- FileSystemObject.CreateTextFile(*name, overwrite, Unicode)* — создает текстовый файл с именем name. Аргумент overwrite при этом указывает, должен ли быть перезаписан файл в том случае, если файл с таким именем уже существует (true — файл может быть перезаписан, false — нет). Необязательный аргумент Unicode указывает, символы какого набора используются для содержимого файла (true — Unicode, false —ASCII).
- FileSystemObject.OpenTextFile(*name, iomode, create, format)* — открывает или создает файл с именем name. Необязательный аргумент iomode может принимать одно из трех значений: 1 — файл предназначен только для чтения, 2 — файл предназначен для записи и 8 — файл предназначен для добавления данных в его конец. Необязательный аргумент create указывает, должен ли быть создан новый файл, если файл с именем name отсутствует (true — файл будет создан, false — нет). И последний, также необязательный, аргумент format служит для задания формата содержимого файла. Он может принимать одно из трех специальных значений: TristateTrue — содержимым файла является Текст Unicode, TristateFalse — ASCII И TristateUseDefault — используется набор символов, по умолчанию заданный в системе.

- File.OpenAsTextStream(*iomode, format)* — **открывает файл, Представленный объектом** File. **Аргументы этого метода полностью эквивалентны аргументам метода** FileSystemObject.OpenTextFile().

Последний метод является одним из четырех методов, определенных для объекта File. Остальные три метода предназначены для проведения операций перемещения, копирования и удаления файлов. В табл. 8.4 представлены эти методы, а также их аналоги, определенные для объекта FileSystemObject.

Таблица 8.4. *Базовые методы объекта* ***File***

Операция	Метод
Перемещение	File.Move **или** FileSystemObject.MoveFile
Копирование	File.Copy или FileSystemObject.CopyFile
Удаление	File.Delete **или** FileSystemObject.DeleteFile

Методы Move и Copy обладают двумя аргументами: destination (обязательный) — путь, куда будет перемещен или скопирован файл, и overwrite (необязательный) — следует ли переписать файл, если он уже присутствует в приемнике копирования, заданном первым аргументом. Методы MoveFile И CopyFile, ПОМИМО destination И overwrite, **Обладают еще** ОДНИМ обязательным аргументом source, значением которого является путь и имя перемещаемого или копируемого файла.

Метод Delete обладает одним необязательным аргументом force, значением которого может быть true (удалить файл, даже если он обладает атрибутом "только для чтения") и false (не удалять файл, если он имеет атрибут "только для чтения"). Метод DeleteFile, помимо force, обладает еще одним, обязательным аргументом source, значением которого является путь и имя удаляемого файла.

Для проверки, существует ли файл с заданным именем, может быть применен метод FileExists() объекта FileSystemObject, принимающий в качестве аргумента путь и имя файла и возвращающий true, если такой файл существует, и false, если — нет.

Рассмотрим несколько примеров, иллюстрирующих работу этих методов.

1. var fso = new ActiveXObject("Scripting.FileSystemObject") — **инициализация** FileSystemObject;
2. var f = fso.getFile("c:\\temp\\test.txt") — **возвращает объект ТИ**па File, являющийся представлением файла c:\temp\test.txt, и помещает полученное значение в переменную f;

- var f = fso.OpenTextFile("c:\\temp\\test.txt",1) — **открывает** файл c:\temp\test.txt для чтения;
- var f = fso.OpenTextFile("c:\\temp\\test.txt", 2, true) — открывает файл c:\temp\test.txt для записи, причем если файл с этим именем отсутствует, то будет создан новый файл;
- f.Copy("c:\\temp\\test1.txt") — копирует файл, представленный объектом f, в папку c:\temp и изменяет его имя на test1.txt;
- fso.DeleteFile("c:\\temp\\test1.txt") — **удаляет файл** testi.txt, находящийся в папке c:\temp, даже если он имеет атрибут "только для чтения".

Для чтения данных из текстового файла у объекта File определены следующие три метода, представленные в табл. 8.5.

Таблица 8.5. *Методы объекта File, реализующие операцию чтения*

Метод	Описание
Read(chars)	Читает из файла chars байт
ReadLine()	Читает из файла одну строку
ReadAll()	Читает все содержимое файла

Если при чтении из файла необходимо пропустить некоторое количество данных, то для этих целей можно воспользоваться одним из двух методов: Skip(chars) — пропускает chars символов, или SkipLine(), который пропускает одну строку.

Следующие три метода объекта File, приведенные в табл. 8.6, предназначены для записи данных в файл.

Таблица 8.6. *Методы объекта File, реализующие операцию записи*

Метод	Описание
Write(text)	Записывает в файл text
WriteLine *(line)*	Записывает в файл строку line
WriteBlankLines(*lines*)	Записывает в файл lines пустых строк

После того, как работа над содержимым файлом закончена, файл нужно закрыть. Для этих целей служит метод close о объекта File, не имеющий аргументов.

Рассмотрим пример, демонстрирующий операции чтения и записи над файлами:

```
' var fso = new ActiveXObject("Scripting.FileSystemObject");
var f1 = fso.OpenTextFile("c:\\temp\\file1.txt",1);
var f2 = fso.OpenTextFile("c:\\temp\\file2.txt",2,true);
var file_content = f1.ReadAll();
f2.Write(file_content);
f1.Close();
f2.Close();
```

Результатом этого кода будет копирование файла file1.txt в файл file2.txt. При этом копирование выполняется не с помощью методов Copyo или CopyFile(), а прочтением всей информации из файла-источника и ее последующей записи в файл-приемник. Это сделано для того, чтобы обеспечить возможность обработки прочитанных данных перед их записью в другой файл, т. е. до выполнения метода f2.Write(file_content);.

Внимание

При чтении или записи данных из файла с помощью FileSystemObject корректно обрабатываются только те данные, которые представлены в кодировке cp1251. Поэтому попытка загрузить в какую-либо область Web-страницы неанглоязычные данные в кодировке, отличной от указанной, приведет к тому, что эти данные будут отображаться так, как если бы для этой Web-страницы была задана кодировка cp1251, вне зависимости от ее действительной кодировки, и, следовательно, вместо данных мы увидим бессмысленный набор символов. Тем не менее, эти данные можно отправить на сервер и, указав соответствующую им кодировку в качестве аргумента конструктора объекта, читающего информацию из входного потока (например, InputStreamReader), корректно эти данные обработать.

Теперь рассмотрим табл. 8.7, содержащую свойства объекта File, а также те характеристики файла, которым они соответствуют.

Таблица 8.7. *Свойства объекта File*

Свойство	Значение
Attributes	**Атрибуты файла**
DateCreated	**Дата создания**
DateLastAccessed	**Дата последнего обращения**
DateLastModified	**Дата последней модификации**
Drive	**Буква диска**

Таблица 8.7 (окончание)

Свойство	Значение
Name	Полное имя
ParentFolder	Folder-представление родительской папки
Path	Полный путь
ShortName	Имя в формате 8.3
ShortPath	Путь в формате 8.3
Size	Размер в байтах
Type	Тип файла из списка зарегистрированных в Windows типов

Всеми вышеперечисленными свойствами обладает и объект Folder.

Свойство Attributes может принимать следующие значения:

- 0 — атрибуты файла или папки не установлены;
- 1 — файл или папка предназначены только для чтения;
- 2 — скрытый;
- 4 — системный;
- 8 — объект является меткой тома;
- 16 — объект является папкой;
- 32 — архивный;
- 64 — объект является мягкой или жесткой ссылкой;
- 128 — сжатый файл.

Значения 8, 16, 64 и 128 являются значениями только для чтения, и соответствующие им атрибуты, будучи заданными, снять невозможно.

8.6.4. Манипулирование путями и именами файлов

Следующие методы объекта FileSystemObject образуют инструментарий, соответствующий заголовку подраздела.

- BuildPath(*path*, name) — **формирует полный путь с помощью конкатенации** своих аргументов. Первый аргумент (path) задает путь к файлу.

второй (name) — его имя. Оба аргумента обязательны. При необходимости добавляется разделитель \.

- GetAbsolutePathName *(path)* — **Возвращает ПОЛНЫЙ путь К объекту,** заданному обязательным аргументом path. При этом path рассматривается как путь, указанный относительно текущей папки. Формирование пути эквивалентно формированию абсолютного URL на основе базового и относительного Internet-адресов.
- GetBaseName(*path*) — возвращает имя, представленное последним компонентом пути, заданного обязательным аргументом path. Если объект обладает расширением, то в возвращаемом значении оно отсутствует.
- GetExtensionName *(path)* — возвращает расширение (расширенное имя) последнего компонента пути, заданного обязательным аргументом path.

8.6.5. Коллекции дисков, папок и файлов

Объектная модель FileSystemObject определяет три типа коллекций.

- Drives — свойство FileSystemObject, значением которого является список всех дисков, имеющихся на той машине, где был инициализирован этот объект. Каждый диск коллекции представлен объектом Drive.
- Folders — свойство объекта Folder, значением которого является список всех папок, содержащихся в представленной данным Folder. Каждая папка коллекции представлена объектом Folder.
- Files — свойство объекта Folder, значением которого является список всех файлов, содержащихся в папке, представленной данным Folder. Каждый файл коллекции представлен объектом File.

Объекты, представляющие каждую из вышеперечисленных коллекций, обладают только двумя свойствами: Count, значением которого является количество элементов в данной коллекции, и Item(*key*), значением которого является элемент, заданный с помощью своего обязательного аргумента key. Как правило, последнее свойство не используется, но вместо него для последовательной обработки коллекций применяется объект Enumerator, определенный в языке JScript.

Для того чтобы проиллюстрировать принципы подобной обработки, рассмотрим пример, включенный в документацию по этому языку.

```
function ShowDriveList() {
var fso, s, n, e, x;    /* Объявляем переменные */
fso = new ActiveXObject("Scripting.FileSystemObject");
e = new Enumerator(fso.Drives);    /* Создаем Enumerator для Drives */
s="",
```

```
for (; !e.atEnd(); e.moveNext()){ /* Цикл по всем элементам коллекции *,
  x = e.item();
  s += x.DriveLetter;    /* Добавляем к s букву диска */
  s += " - ";            /* Добавляем к s символ дефиса */
  if (x.DriveType == 3) /* Если диск сетевой, */
    n = x.ShareName;     /* то присвоить n его сетевое имя, */
  else if (x.IsReady)    /* иначе, и если диск готов к использованию, */
       n = x.VolumeName;   /* то присвоить n значение метки тома. */
      else n = "  [Диск не доступен]"; /* во всех остальных случаях */
  s += n + "<br>";/* Добавляем к s значение n и элемент новой строки */
  return(s); /* Возвращается список доступных дисков */  }
```

Как видно, представленная функция формирует и возвращает список доступных для использования дисков. Обработка коллекций, отличных от Drives, производится аналогично.

Далее следует список методов объекта Enumerator:

- atEnd() — возвращает true, если был достигнут конец коллекции и false в противном случае;
- item() — возвращает текущий элемент коллекции;
- moveFirst() — делает текущим первый элемент коллекции;
- moveNext() — делает текущим следующий элемент коллекции.

Заметим, что объект Enumerator не имеет свойств, а объекты, представляющие коллекции, не имеют методов.

ГЛАВА 9

Представление данных

В этой главе мы расскажем о том, как реализовать слой Web-приложения, отвечающий за представление данных, т. е. за удобный для пользователя механизм их обработки. Понятие представления данных, как мы уже говорили ранее, включает в себя не только внешний вид информационных компонентов Web-страницы, но и способы взаимодействия пользователя с этими компонентами. Кроме того, реализация слоя представления данных не должна исключать возможность связи с сервером и наносить ущерб остальным слоям Web-приложения в целом.

Таким образом, можно сказать, что в этой главе мы займемся в основном разработкой внешнего вида блоков информации, зависящего от их структуры, а также его изменением под воздействием различных факторов. Но, занимаясь этими разработками, мы будем учитывать тот факт, что они ведутся в рамках Web-приложения и, следовательно, подчиняются требованиям, предъявляемым Web-средой.

Одно из таких требований мы уже назвали — возможность эффективного функционирования всех слоев Web-приложения, но не эффективная реализация слоя представления данных в ущерб эффективности реализаций других слоев. Вторым примером может послужить ограничение на размер в байтах той информации, которая перемещается между клиентом и сервером.

Разработка слоя представления данных представляет собой достаточно сложный, опирающийся на множество технологий процесс, но, тем не менее, в этой главе мы попробуем в той или иной степени охватить каждый аспект этого процесса.

9.1. Графический пользовательский интерфейс

Любая операционная система, обладающая графическим ядром, для своих пользователей обеспечивает возможность гибкого и удобного проведения

различных операций с помощью встроенных в нее графических компонентов. Совокупность этих компонентов принято называть GUI (Graphics User Interface — Графический Пользовательский Интерфейс). При этом само слово "интерфейс" обозначает набор средств коммуникации между компьютером и его пользователем.

В основе GUI лежит понятие окна, которое представляет собой как главный графический компонент, являющийся контейнером для остальных, дочерних графических компонентов, так и сами эти дочерние компоненты. Любое приложение, обладающее GUI, взаимодействует со своим пользователем с помощью много- или однооконного интерфейса. В первом случае GUI представляет собой главное окно, в котором могут создаваться дочерние окна, во втором — одно единственное окно.

Каждое из окон, главное или дочернее, может содержать в себе графические компоненты, которые называются элементами управления (controls). Примерами элементов управления могут послужить деревья, селекторы, кнопки, а также все те элементы, которые были перечислены в *главе 8* как поля ввода Web-формы. Но прежде чем переходить к элементам управления, рассмотрим те компоненты, из которых может состоять любое из главных или дочерних окон, образующих оконный интерфейс.

В качестве примера возьмем окно браузера MSIE 5.5 (рис. 9.1).

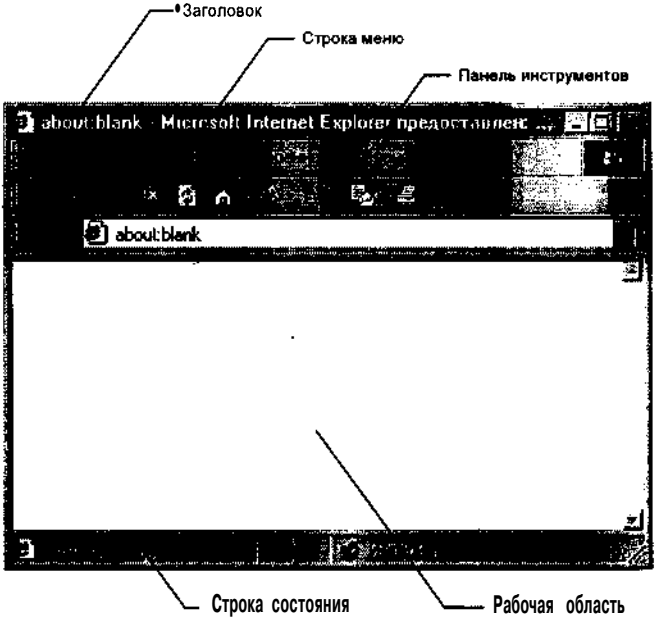


Рис. 9.1. Окно браузера MSIE 5.5

Как видно из рис. 9.1, окно браузера состоит из пяти компонентов, образующих следующий список.

- Заголовок — содержит слева направо: системное меню, служащее для управления размерами окна и для его закрытия, информацию о том, что содержится в рабочей области окна, а также кнопки свертывания, развертывания/восстановления и закрытия окна.
- Строка меню — содержит графические компоненты, являющиеся визуальным представлением операций, которые может осуществить пользователь над содержимым рабочей области окна или, если это главное окно, над его дочерними окнами.
- Панель инструментов — состоит из кнопок, каждая из которых, как правило, соответствует какому-либо пункту меню. Панель инструментов предназначена для быстрого выполнения наиболее часто использующихся команд.
- Рабочая область — содержит либо дочерние окна главного окна, либо представление той информации, для обработки которой предназначено приложение.
- Строка состояния — показывает дополнительную информацию о содержимом рабочей области, а также о состоянии выполнения той или иной операции.

Помимо этих пяти компонентов можно назвать адресную строку, но на самом деле она является частью панели инструментов, а также границу окна, с помощью которой можно изменять его размеры.

В том случае, когда приложение обладает однооконным интерфейсом, рабочая область окна этого приложения может содержать помимо информации, Для обработки которой было создано данное приложение, еще и некоторый инструментарий, необходимый для этой обработки. В терминологии GUI каждый из таких инструментов представлен некоторым элементом управления.

Например, рабочая область Проводника Windows разбита на две части, в левой находится дерево, каждым узлом которого является графическое представление какого-либо устройства или объекта файловой системы, а в правой — информация, сопоставленная с этим объектом. Как правило, этой информацией является содержимое папки, соответствующей помеченному узлу дерева в левой части окна Проводника.

Если рассмотреть в качестве примера браузер MSIE 5.5, то инструментами обработки информации, находящейся в рабочей области соответствующего окна (см. рис. 9.1), будут все те средства манипулирования данными, которые представлены на просматриваемой Web-странице. Иными словами, эти инструменты обеспечиваются не приложением (браузером), а самим документом (Web-страницей), представляющим собой обрабатываемую этим приложением информацию.

Каждый компонент окна или элемент управления обладают характерным для них внешних видом, наиболее приспособленным к выполнению тех функций, для которых они предназначены. Но кроме соответствующих ему операций каждый оконный компонент или элемент управления должен обеспечивать еще и их выполнение пользователем. Таким образом, можно сказать, что каждый элемент интерфейса обладает как видимой составляющей — внешним видом, так и невидимой — механизмом, обеспечивающим выполнение тех действий, которые сопоставлены с этим элементом.

Как же устроена невидимая составляющая (везде далее НВ) GUI? Фактически перед ней стоят две задачи. Во-первых, НВ должна обеспечивать механизм реагирования на действия пользователя, которые тот совершил над данным элементом интерфейса. Этот механизм должен формально описывать полную классификацию всевозможных действий пользователя и в соответствии с ней относить каждое свершенное действие к какому-либо классу. Во-вторых, НВ должна содержать инструкции для реагирования на каждое действие пользователя из тех, которые должны быть обработаны.

В последние годы для решения этих двух задач все более и более активно применяются средства объектно-ориентированного программирования (ООП). И разработанный с их помощью GUI является также объектно-ориентированным (хотя и не на 100%). Согласно парадигме ООП каждому окну приложения сопоставляется отдельный объект, часть методов которого реализует механизм реагирования на действия пользователя и в соответствии с типом действия передает управление тому компоненту приложения, который уже не является частью интерфейса.

Классификация действий пользователя в приложении, реализующем объектно-ориентированный интерфейс, происходит на основе т. н. событий и сообщений. При каждом щелчке мыши или нажатии кнопки клавиатуры генерируется аппаратное событие, в ответ на которое специальный метод объекта окна посылает другому специальному методу соответствующее сообщение. Таким образом, разработчик имеет возможность для каждого типа принимаемого сообщения написать код, выполняющий требуемые действия.

Если рассмотреть всевозможные объектно-ориентированные языки программирования, то можно заметить, что вышеописанная модель инфраструктуры объектно-ориентированного интерфейса реализуется каждым из этих языков в принципе одинаково, но с небольшими отличиями. В некоторых языках обработка событий скрыта от разработчика, и ему доступна только обработка сообщений, в других — наоборот. Но в общем схема работы объектно-ориентированного интерфейса будет выглядеть примерно так:

1. В результате изменения состояния какого-либо устройства генерируется аппаратное событие. Чаще всего это событие генерируется в ответ на поступление данных от какого-либо устройства ввода (нажатие кнопки мыши или клавиатуры, перемещение указателя мыши и т. п.).

2. Управление передается обработчику событий, который выполняет одну или обе из следующих двух функций: обрабатывает событие и генерирует одно или несколько сообщений, соответствующих этому событию.
3. По определенному разработчику принципу сгенерированные сообщения отсылаются нужным объектам и обрабатываются этими объектами. В этом процессе могут генерироваться новые сообщения, которые также будут отправлены каким-либо объектам и ими обработаны и т. д.
4. В процессе обработки сообщений управление может быть передано коду, который не относится к той части приложения, которая отвечает за функциональность интерфейса (не интерфейсная часть приложения). Этот код может отвечать, например, за обработку данных, для которой было создано данное приложение, за установку связи с другими локальными или удаленными приложениями и т. п.

Заметим еще раз, что некоторые действия, описанные в данной схеме, являются опциональными.

9.2. Архитектура GUI в терминах технологий XML и HTML

В *главе 6* мы рассматривали многослойную архитектуру Web-приложения. Если представить себе Web-приложение, в котором реализован каждый из этих слоев, то принцип его работы можно будет описать следующим образом:

1. Получение доступа к данным. При этом если данные хранятся непосредственно в XML-документе, то на обработку поступает этот документ. Если же они представлены в каком-либо другом виде, то на обработку поступает XML-документ, контент которого динамически генерируется логическим слоем Web-приложения. В последнем случае логический слой не только формирует структуру нового XML-документа, но и отвечает за получение данных от внешнего источника информации, например, из базы данных.
2. Структурированные данные (текст + разметка) с помощью правил шаблона трансформируются в HTML-документ, который затем отправляется клиенту.

Общепринятое разбиение Web-приложения на слои порождает только три слоя: слой структуры, логической обработки и представления данных. Все остальные виды слоев, рассмотренные в *главе 6,* введены нами для удобства описания архитектуры Web-приложения. Поэтому, формально говоря, все многообразие инструментария, служащего для разработки интерфейса (HTML, JavaScript, CSS), описывается вторым пунктом вышеприведенного

списка и образует лишь один слой Web-приложения — слой представлени информации.

Но если вспомнить, что за корректную работу интерфейсной части Web. приложения отвечает клиентская часть этого приложения, то, рассматривая интерфейсную часть Web-приложения отдельно, в рамках ее реализующи технологий, эту часть также можно разбить на составляющие, аналогичны слоям Web-приложения. Фактически мы и проделали это поверхностно в *главе 6*.

Исходя из того, что используемыми нами технологиями являются HTML, CSS и JavaScript, слоями, образующими HTML-ориентированное Web-приложение, будут:

- Слой структуры данных. Этот слой представлен тегами языка HTML, а в основе принципа структурирования данных лежит взаиморасположение контента этих тегов. Иными словами, данные структурируются в зависимости не от своего содержания или смысла, а от своего месторасположения на Web-странице.
- Слой представления данных. Этот слой реализуется с помощью каскадных стилевых таблиц. Контент каждого тега HTML оформляется в соответствии с заданными разработчиком, либо установленными по умолчанию правилами CSS.
- Логический слой. Здесь обеспечивается реакция структуры и оформления данных на те или иные действия пользователя.

Таким образом, схема работы Web-приложения, обладающего такой архитектурой, будет складываться из следующих этапов:

1. На основе исходного HTML-документа формируется его конечное дерево. На этом этапе в документ интегрируются различные внешние данные, такие, как стили CSS и сценарии JavaScript. Некоторые ветви конечного дерева, отсутствующие в исходном документе, генерируются сценариями JavaScript.
2. На основе взаиморасположения тегов в исходном документе формируется примерная схема расположения на Web-странице текстового контента этих тегов.
3. Оформление документа завершается уточнением месторасположения текстовых блоков на Web-странице с использованием информации о размере этих текстовых блоков, а также некоторых правил CSS, влияющих на размер и позиционирование областей, соответствующих элементам HTML исходного документа. На этом этапе также раскрашиваются те объекты, для которых разработчиком были заданы соответствующие правила CSS.

В следующем разделе мы опишем реализацию как Web-приложения в целом. так и его HTML-составляющей, на которой мы остановимся более подробно.

9.3. Многоуровневая разработка компонентов GUI

Итак, как было сказано в предыдущем разделе, выполнение Web-приложения начинается с обработки исходного XML-документа или с его формирования в том случае, когда некоторая часть данных, подлежащих обработке, находится во внешнем источнике. В последнем случае получение внешних данных реализуется на логическом слое Web-приложения.

Существует множество различных способов хранения данных, но мы рассмотрим только тот, при котором внешние данные хранятся в формате XML. Средством для встраивания таких данных в XML-документ, которое мы выберем, являются элементы языка XML, принадлежащие пространству имен с префиксом `FP:`. Эти элементы были описаны нами в *разд. 6.6,* поэтому мы не будем здесь на них останавливаться, но проиллюстрируем их применение *в разд. 9.6* при описании комплексного примера.

После того как XML-документ сформирован, наступает этап его обработки, на котором генерируется HTML-документ, отправляемый клиенту. Генерирование этого документа в целом было описано нами в предыдущем разделе, а различные подробности были приведены почти во всех предыдущих главах. В частности, мы показали, как в сценарии XSLT встраиваются коды JavaScript *(см. разд. 4.4),* а также, как происходит оформление данных с помощью стилей CSS *(см. разд. 7.4).* Эти этапы мы также не будем здесь повторно описывать, а продемонстрируем результаты их применения в *разд. 9.6.*

Здесь мы подробно остановимся на реализации интерфейсной части Web-приложения, а именно на некоторых возможностях языка JavaScript как технологии создания клиентской части Web-приложения. Мы будем рассматривать те конструкции этого языка, которые обеспечивают решение задач, стоящих перед невидимой частью интерфейса, описанной нами в *разд. 9.1.* Так как выбранным нами пользовательским агентом является браузер MSIE 5.5, мы приведем здесь описания событий, определенных в этом браузере, а также механизма их обработки.

9.4. События

Обработать событие в языке HTML можно двумя способами. Первый из них заключается в следующем. Для элемента, который должен отреагировать на событие, описывается атрибут, именем которого является обрабатываемое событие, а значением либо код обработки события, либо функция JavaScript, являющаяся обработчиком события. После того как событие

произойдет, управление будет передано либо коду, являющемуся значением соответствующего атрибута, либо функции-обработчику. Рассмотрим несколько примеров.

```
<div id = "obj" onclick = "alert('событие onclick произошло')">
    Нажми меня
</div>
```

В этом коде мы определили, как должен вести себя элемент div (или соответствующий ему объект obj) в том случае, если пользователь щелкнет по нему левой кнопкой мыши. Как видно из листинга, управление будет передано коду, являющемуся значением атрибута onclick, причем в этом значении содержится функция alert(). Таким образом, после того как пользователь кликнет по элементу div, на экран будет выведено предупреждающее сообщение, представленное аргументом этой функции.

Вторым способом обработки события является определение одноименного с этим событием свойства объекта, который должен отреагировать на данное событие. Рассмотрим следующий код:

```
<div id = "obj">
    Нажми меня
</div>
<script>
    obj.onclick = function() {
alert ('событие onclick произошло'); }
</script>
```

Результатом обработки этого кода будет то же самое, что и в предыдущем примере. Заметим, что рассматриваемый способ обработки событий полезен, когда создается много однотипных объектов, т. е. одинаково реагирующих на одно и тоже событие. Такими объектами могут, например, быть пункты меню или элементы какого-либо списка. В этом случае вместо многократного переписывания одних и тех же атрибутов и их значений для разных элементов достаточно будет один раз описать функцию, содержащую код обработки всех подлежащих обработке событий, которую затем вызвать для каждого из этих элементов.

Далее следует перечень основных событий, определенных в MSIE 5.5.

Примечание

Везде далее, кроме листингов, мы будем писать имена элементов HTML большими буквами для того, чтобы отличать их от соответствующих им объектов JavaScript, а также остальных конструкций этого языка, являющихся чувствительными к регистру символов.

9.4.1. onabort

Генерируется, когда пользователь прерывает загрузку рисунка.

Поддерживается элементом: IMG.

9.4.2. onbeforecopy

Генерируется перед тем, как выделенный контент будет скопирован в системный буфер (клипборд).

Поддерживается следующими элементами (объектами): A, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, DT, EM, FIELDSET, FORM, Hn, I, IMG, LABEL, LEGEND, LI, LISTING, MENU, NOBR, OL, P, PLAINTEXT, PRE, S, SAMP, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TD, TEXTAREA, TH, TR, TT, U, UL.

9.4.3. onbeforecut

Генерируется перед тем, как выделенный контент будет удален из документа и помещен в буфер обмена.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.4. onbeforeeditfocus

Генерируется перед тем, как элемент окажется в фокусе, т. е. сопоставленные с этим объектом данные станут доступны для редактирования.

Поддерживается следующими элементами (объектами): A, APPLET, AREA, BUTTON, custom, DIV, document, FIELDSET, INPUT, MARQUEE, OBJECT, SELECT, SPAN, TABLE, TD, TEXTAREA, TR.

9.4.5. onbeforepaste

Генерируется перед тем, как данные из буфера обмена будут вставлены в Документ.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.6. *onbeforeunload*

Генерируется перед тем, как страница будет выгружена.

Поддерживается следующими элементами (объектами): body, FRAMESET, window.

9.4.7. *onblur*

Генерируется, когда объект теряет фокус.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, BUTTON, CAPTION, CENTER, CITE, custom, DD, DEL, DFN, DIR, DIV, DL, DT, EM, EMBED, FIELDSET, FONT, FORM, FRAME, FRAMESET, Hn, HR, I, IFRAME, IMG, INPUT (**кроме** type = "hidden"), INS, ISINDEX, KBD, LABEL, LEGEND, LI, LISTING, MARQUEE, MENU, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, window, XMP.

9.4.8. *onchange*

Генерируется при изменении контента объекта.

Поддерживается следующими элементами: INPUT type = "text", SELECT, TEXTAREA.

9.4.9. *onclick*

Генерируется, когда пользователь щелкнет по объекту левой кнопкой мыши.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI,

LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.10. oncontextmenu

Генерируется, когда пользователь щелкнет по объекту правой кнопкой мыши, открывающей контекстное меню.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.11. oncopy

Генерируется при копировании контента в системный буфер.

Поддерживается следующими элементами (объектами): A, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, DT, EM, FIELDSET, FORM, Hn, HR, I, IMG, LEGEND, LI, LISTING, MENU, NOBR, OL, P, PLAINTEXT, PRE, S, SAMP, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TD, TH, TR, TT, U, UL.

9.4.12. oncut

Генерируется при вырезании контента в буфер обмена.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, тт, U, UL, VAR, XMP.

9.4.13. ondblclick

Генерируется, когда пользователь дважды щелкнет левой кнопкой мыши по объекту.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.14. *ondrag*

Генерируется во время перетаскивания объекта при операции drag-and-drop.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, S, SAMP, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TR, TT, U, UL, VAR, XMP.

9.4.15. *ondragend*

Генерируется при завершении перетаскивания объекта (когда пользователь отпустит левую кнопку мыши) при операции drag-and-drop.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, S, SAMP, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TR, TT, U, UL, VAR, XMP.

9.4.16. *ondragenter*

Генерируется элементом-приемником при перетаскивании пользователем объекта в допустимое место при операции drag-and-drop.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH. THEAD, TR, TT, U, UL, VAR, XMP.

9.4.17. *ondragleave*

Генерируется, когда пользователь перетаскивает объект за пределы области, соответствующей элементу-приемнику, при операции drag-and-drop.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, FIELDSET, FONT, FORM, до, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TR, TT, U, UL, VAR, XMP.

9.4.18. *ondragover*

Генерируется при перемещении пользователем перетаскиваемого объекта над областью, соответствующей элементу-приемнику, при операции drag-and-drop. Генерируется ПОСЛе ondragenter.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, *I*, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TR, TT, U, UL, VAR, XMP.

9.4.19. *ondragstart*

Генерируется в начале операции drag-and-drop.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.20. *ondrop*

Генерируется элементом-приемником, когда пользователь завершил операцию drag-and-drop и отпустил кнопку мыши.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TR, TT U, UL, VAR, XMP.

9.4.21. *onerror*

Генерируется при возникновении ошибки во время загрузки объекта.

Поддерживается следующими элементами (объектами): IMG, OBJECT, STYLE, window.

9.4.22. *onfocus*

Генерируется при получении фокуса объектом.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, BUTTON, CAPTION, CENTER, CITE, custom, DD, DEL, DFN, DIR, DIV, DL, DT, EM, EMBED, FIELDSET, FONT, FORM, FRAME, FRAMESET, Hn, HR, I, IFRAME, IMG, INPUT, INS, ISINDEX, KBD, LABEL, LEGEND, LI, LISTING, MARQUEE, MENU, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, window, XMP.

9.4.23. *onhelp*

Генерируется при нажатии клавиши <F1>, если окно браузера активно.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, window, XMP.

9.4.24. *onkeydown*

Генерируется при нажатии клавиши клавиатуры.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE,

CODE, custom, DD, DEL, DFN, DIR, DIV, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE. TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.25. *onkeypress*

Генерируется при нажатии клавиши клавиатуры, соответствующей букве или цифре.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.26. *onkeyup*

Генерируется, когда пользователь отпускает клавишу клавиатуры.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, document, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.27. *onload*

Генерируется сразу после того, как объект был загружен.

Поддерживается следующими элементами (объектами): APPLET, body, EMBED, FRAME, FRAMESET, IFRAME, IMG, LINK, SCRIPT, window.

9.4.28. *onlosecapture*

Генерируется, когда объект перестает перехватывать все события от мыши в текущем документе.

Примечание

Для того чтобы объект начал перехватывать все события от мыши в текущем документе, нужно вызвать метод setCapture() этого объекта. Для того чтобы объект перестал перехватывать эти события, нужно вызвать метод releaseCapture () этого объекта.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BR, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT, KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, OPTION, P, PLAINTEXT, PRE, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.29. onmousedown

Генерируется, когда нажата любая кнопка мыши.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.30. onmouseenter

Генерируется, когда пользователь перемещает указатель мыши над объектом. Генерируется только данным объектом, но не объектами, в нем содержащимися.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BASE, BASEFONT, BDO, BGSOUND, BIG, BLOCKQUOTE, body, BUTTON. CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, DT, EM, EMBED. FIELDSET, FONT, FORM, Hn, HR, HTML, I, IMG, INPUT (**кроме** type = "hidden"). KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID. NOBR, OL, P. PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SOB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.31. onmouseleave

Генерируется объектом, когда указатель мыши выходит за пределы области, соответствующей этому объекту.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS. APPLET, AREA, B, BASE, BASEFONT, BDO, BGSOUND, BIG, BLOCKQUOTE, body. BUTTON.

CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, DT, EM, EMBED, FIELDSET, FONT, FORM, *Hn*, HR, HTML, *I*, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.32. *onmousemove*

Генерируется, когда пользователь перемещает указатель мыши над объектом. Генерируется и данным объектом, и содержащимися в нем объектами.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.33. *onmouseout*

Генерируется объектом, когда указатель мыши находится за пределами области, соответствующей этому объекту.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.34. *onmouseover*

Генерируется объектом, когда указатель мыши находится в пределах области, соответствующей этому объекту.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.35. onmouseup

Генерируется, когда пользователь отпускает кнопку мыши в то время, как указатель мыши находится над объектом.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.36. onpaste

Генерируется объектом, когда пользователь вставляет в него данные из буфера обмена.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OL, P, PLAINTEXT, PRE, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.37. onpropertychange

Генерируется, когда изменяется значение какого-либо свойства объекта.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, COMMENT, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET. FONT, FORM, Hn, HR, *I*, IMG, INPUT, KBD, LABEL, LEGEND, LI, LISTING, MAP, MARQUEE, MENU, NOBR, OBJECT, OL, OPTION, P, PLAINTEXT, PRE, S, SAMP, SCRIPT. SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.38. onreadystatechange

Генерируется, когда изменяется состояние загрузки объекта.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BASE, BASEFONT, BDO, BGSOUND, BIG, BLOCKQUOTE, body, BR.

BUTTON, CAPTION, CENTER, CITE, CODE, COL, COLGROUP, COMMENT, custom, DD, DEL, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, HEAD, Hn, HR, HTML, I, IFRAME, IMG, INPUT, INS, ISINDEX, KBD, LABEL, LEGEND, LI, LINK, LISTING, MAP, MARQUEE, MENU, namespace, NEXTID, NOBR, NOFRAMES, NOSCRIPT, OBJECT, OL, OPTION, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SCRIPT, SELECT, SMALL, SPAN, STRIKE, STRONG, STYLE, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, title, TR, TT, U, UL, VAR, XML, XMP.

9.4.39. *onreset*

Генерируется, когда пользователь устанавливает для полей формы исходные значения.

Поддерживается элементом FORM.

9.4.40. *onresize*

Генерируется при изменении размеров объекта.

Поддерживается следующими элементами (объектами): A, ADDRESS, APPLET, B, BIG, BLOCKQUOTE, BUTTON, CENTER, CITE, CODE, custom, DD, DFN, DIR, DIV, DL, DT, EM, EMBED, FIELDSET, FORM, FRAME, Hn, HR, I, IMG, INPUT (**кроме** type = "hidden"), ISINDEX, KBD, LABEL, LEGEND, LI, LISTING, MARQUEE, MENU, OBJECT, OL, P, PRE, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TEXTAREA, TT, U, UL, VAR, window, XMP.

9.4.41. *onresizeend*

Генерируется, когда изменение размеров объекта завершено.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, FRAME, FRAMESET, Hn, HR, I, IFRAME, IMG, INPUT, INS, ISINDEX, KBD, LABEL, LEGEND, LI, LISTING, MARQUEE, MENU, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, window, XMP.

9.4.42. *onresizestart*

Генерируется, когда начинается изменение размеров объекта.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, APPLET, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE,

custom, DD, DFN, DIR, DIV, DL, document, DT, EM, EMBED, FIELDSET, FONT, FORM, FRAME, FRAMESET, Hn, HR, I, IFRAME, IMG, INPUT, INS, ISINDEX, KBD, LABEL, LEGEND, LI, LISTING, MARQUEE, MENU, OBJECT, OL, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, window, XMP.

9.4.43. *onscroll*

Генерируется при прокрутке содержимого объекта.

Поддерживается следующими элементами (объектами): APPLET, BDO, body, custom, DIV, EMBED, MAP, MARQUEE, OBJECT, TABLE, TEXTAREA, window.

9.4.44. *onselect*

Генерируется, когда изменяется выделенная область.

Поддерживается следующими элементами (объектами): BODY, INPUT type = "text", TEXTAREA.

9.4.45. *onselectionchange*

Генерируется при изменении выделенной области, соответствующей объекту select.

Поддерживается объектом document.

9.4.46. *onselectstart*

Генерируется, когда пользователь начинает выделять данные.

Поддерживается следующими элементами (объектами): A, ACRONYM, ADDRESS, AREA, B, BDO, BIG, BLOCKQUOTE, body, BUTTON, CAPTION, CENTER, CITE, CODE, custom, DD, DEL, DFN, DIR, DIV, DL, DT, EM, FIELDSET, FONT, FORM, Hn, HR, I, IMG, INPUT (кроме type = "hidden"), KBD, LABEL, LI, LISTING, MAP, MARQUEE, MENU, NEXTID, NOBR, OBJECT, OL, OPTION, P, PLAINTEXT, PRE, Q, RT, RUBY, S, SAMP, SELECT, SMALL, SPAN, STRIKE, STRONG, SUB, SUP, TABLE, TBODY, TD, TEXTAREA, TFOOT, TH, THEAD, TR, TT, U, UL, VAR, XMP.

9.4.47. *onsubmit*

Генерируется, когда пользователь подтверждает ввод данных в форму.

Поддерживается элементом FORM.

9.4.48. onunload

Генерируется непосредственно перед тем, как объект был выгружен из документа.

Поддерживается следующими элементами (объектами): BODY, FRAMESET, window.

В конце *главы 7* мы обещали рассказать о том, как сделать так, чтобы нарисованная нами кнопка реагировала на нажатие кнопки мыши, а также, чтобы курсор мыши имел форму стрелки при прохождении над нашей кнопкой. Рассмотрим следующий код:

```
<div
 style="position:absolute; left:20px; top:20px; width:90px; height:20px;
        background-color:#8399B1; text-align:center; color:white;
        border-width:2px; border-style:outset; cursor:default"
 onmousedown="this.setCapture();this.style.borderStyle='inset'"
 onmouseup="this.releaseCapture();this.style.borderStyle='outset'"
 onclick = "alert('Кнопка нажата')"
 onselectstart = "return false">
 Нажми меня
</div>
```

Здесь мы обрабатываем четыре события от мыши: onmousedown *(см. разд. 9.4.30),* onmouseup *(см. разд. 9.4.36),* onclick *(см. разд. 9.4.9)* и onselectstart *(см. разд. 9.4.47).* Первое событие генерируется, когда пользователь нажимает кнопку мыши, курсор которой находится над кнопкой. При этом для объекта, соответствующего элементу div с помощью метода setcapture о , включается режим перехвата всех событий от мыши. Это сделано для того, чтобы в случае, если пользователь отпустит кнопку мыши за пределами области, соответствующей элементу div, кнопка приобрела такой внешний вид, которым она обладала до нажатия.

Внешний вид кнопки регулируется с помощью изменения значения свойства borderStyle. Если этим значением является inset, то кнопка выглядит Нажатой, если outset, то — как до своего нажатия. Как видно из кода, кнопка приобретает нажатый вид, когда генерируется событие onmousedown, И обычный, когда генерируется onmouseup, при обработке которого объект Перестает перехватывать все события от мыши. Отключение режима их пе-Рехвата осуществляется с помощью метода releaseCapture о .

Действие, которое должно осуществиться, когда пользователь нажимает Кнопку, задано кодом обработчика события onclick. В нашем случае этим Действием является вывод на экран сообщения "Кнопка нажата". Заметим, Что это действие не будет выполнено, если пользователь отпустит кнопку Мыши за пределами области, соответствующей элементу div.

И, наконец, событие onselectstart обрабатывается для того, чтобы пользователь не смог выделить текст надписи кнопки. Если бы это событие не было обработано, то, нажав кнопку мыши над областью элемента div и перемещая указатель мыши, пользователь выделял бы этот текст целиком или частично.

9.5. Обработка структуры HTML-документа

При обработке HTML-документа MSIE 5.5 строит для него DOM-модель, в основе которой, как мы уже говорили, лежит иерархическая или древовидная модель данных. Таким образом, если графический компонент в результате каких-либо действий пользователя изменяет свой (или другого компонента) внешний вид, то эти изменения представляют собой набор операций над деревом документа. При этом каждым узлом этого дерева является какая-либо конструкция языка HTML (элемент, атрибут, комментарий и т. п.), следовательно, нас здесь будут интересовать те возможности JavaScript, с помощью которых мы сможем обрабатывать структуру дерева HTML-документа. Проводя параллель между форматами XML и HTML, можно заметить, что реализация последнего в MSIE 5.5 предоставляет разработчику те же самые наборы инструментов, что и язык XML. Среди этих инструментов имеются средства для абсолютной и относительной адресации узлов дерева HTML-документа, которые представлены, разумеется, не осями, как в XML, а свойствами некоторых объектов в JavaScript. С помощью отдельных средств JavaScript можно проводить не только примитивную адресацию узлов, но и поиск по различным критериям, что, в общем, также является адресацией, только более сложной.

Ну и, конечно же, среди рассматриваемых ниже конструкций JavaScript описываются средства непосредственной модификации дерева документа. Эти средства включают в себя не только создание и удаление узлов, но и такие нетривиальные операции, как клонирование, взаимозамена узлов, копирование всего множества атрибутов одного элемента другому, а также несколько способов удаления узлов и ветвей дерева.

Как и в предыдущих разделах, мы не будем слишком подробно останавливаться на описании каждого свойства или метода. Вместо этого мы разобьем всю их совокупность на семейства и для каждого такого семейства представим подробное описание одного или нескольких его типичных представителей.

9.5.1. Абсолютная адресация узлов

После того как в соответствии с DOM дерево HTML-документа построено, этому документу ставится в соответствие объект document, который фактически является JavaScript-представлением дерева документа. И используя

свойства и методы объекта document, можно получить как общую информацию о документе в целом, так и о каждой его части, т. е. узле или ветви дерева. В последнем случае используются т. н. коллекции, которые можно обрабатывать, оперируя индексами их членов.

В табл. 9.1 представлены все коллекции, являющиеся полями объекта document.

Таблица 9.1. *Коллекции объектов document*

Коллекция	Содержимое
all	Все элементы, содержащиеся в документе
anchors	Объекты A, для которых определено какое-либо из свойств name или id
applets	Все объекты APPLET в документе
childNodes	Элементы и узлы текстового типа, являющиеся непосредственными потомками текущего узла
children	Объекты, являющиеся непосредственными потомками текущего объекта
embeds	Все объекты EMBED в документе
forms	Все объекты FORM в документе
frames	Все объекты FRAME в документе
images	Все объекты IMG в документе
links	Объекты A, для которых определено свойство href, а также все объекты AREA
namespaces	Все объекты namespace в документе
scripts	Все объекты SCRIPT в документе
stylesheets	Все объекты stylesheet в документе, представленные элементами LINK и STYLE

Примечание

При использовании некоторых конструкций JavaScript следует иметь в виду, что иногда разница между объектом, соответствующим элементу и обозначаемому, как oElement, и объектом, обозначаемым oNode, представляющим собой узел дерева документа, является принципиальной, т. к. узел помимо элемента может быть еще и текстовым узлом. В таких случаях мы будем конкретизировать, какой из типов объектов имеется в виду.

Рассмотрим основные приемы работы с коллекциями. Каждый объект, представляющий собой коллекцию, обладает свойством length, значением которого является количество элементов коллекции. С помощью этого свойства можно последовательно перебрать все члены коллекции от первого до последнего. Для каждого узла документа номер его позиции в коллекции all можно получить с помощью свойства `sourceIndex` объекта, соответствующего этому элементу.

Помимо свойства length, каждая коллекция обладает методом item(*index*), с помощью которого можно обратиться к любому члену коллекции, используя его индекс, представленный аргументом метода. Причем в качестве индекса может выступать не только порядковый номер члена коллекции, но значение его свойства id *(см. разд. 9,5.3),* а иногда и свойства name.

К нужному члену коллекции можно получить доступ и с помощью следующей конструкции:

```
collectionName(index1, index2)
```

Здесь collectionName — имя коллекции, indexl — обязательный аргумент, представляющий собой первичный индекс члена коллекции, описанный выше, index2 — необязательный аргумент, значением которого является номер члена коллекции в списке, который был получен отбором из коллекции всех ее членов, свойство id которых совпадает со значением indexl. Иными словами, если значением первого аргумента является id искомых членов коллекции и таких членов несколько, то второй аргумент указывает, какой из них является искомым — первый, второй и т. д. из найденных.

Опираясь на все вышесказанное, рассмотрим механизм абсолютной адресации. Суть его довольно проста и заключается в том, что, адресуя какой-либо узел в документе, мы должны последовательно, начиная с объекта document, перечислить всех его предков, являющихся членами какой-либо из коллекций. Далее следует несколько примеров:

```
document.forms(0).elements(3)
document.all('my_div').item('name1').children(3)
```

Мы не будем здесь останавливаться на объяснениях того, как работают эти примеры. Мы привели их только для того, чтобы показать всю громоздкость такого способа адресации. Хотя в практических задачах абсолютные адреса объектов являются, как правило, гораздо более длинными, но с другой стороны, иногда и их применение бывает оправданным. Более удобным способом адресации объектов является их относительная адресация, но об этом чуть позже.

Кроме возможности абсолютной адресации узлов объект document предоставляет пару методов для модификации документа. Этими методами являются write () и `writeln()`, которые записывают в документ свой строковый

аргумент. Последний метод отличается от первого тем, что добавляет после строки своего аргумента разрыв строки.

Недостатками этих методов является то, что весь контент документа, созданный без их помощи при первом к ним успешном обращении полностью уничтожается. Иными словами, если где-либо в документе находится код `document.write('nothing')` и этот код сработает, то после этого весь документ будет состоять лишь из одного слова nothing. От первоначального документа не останется ничего, даже корневого элемента.

Исходя из вышесказанного можно сделать вывод, что методы `write()` и `writeln()` могут быть использованы либо в самом начале документа (в контенте элемента SCRIPT, который является первым непосредственным потомком элемента BODY), либо весь документ должен быть сгенерирован с помощью этих методов. В остальных случаях частичная или полная потеря контента неизбежна.

Примечание

Для получения доступа к объекту, представляющему корневой элемент документа (как правило, HTML), служит свойство `documentElement` объекта document.

9.5.2. Относительная адресация узлов

Принцип работы относительной адресации, предлагаемый JavaScript, во многом аналогичен соответствующему принципу адресации элементов документа в формате XML. Относительная адресация элементов HTML реализуется в MSIE 5.5 с помощью шести свойств, которые перечислены в табл. 9.2.

Таблица 9.2. *Свойства, реализующие относительную адресацию узлов*

Свойство	Адресуемый узел
firstchild	Первый непосредственный потомок текущего узла
lastchild	Последний непосредственный потомок текущего узла
nextSibling	Следующий сразу после текущего и имеющий с ним одного и того же родителя
parentElement	Родитель текущего узла; если текущий узел является корневым, имеет значение null; свойство эквивалентно с parentNode
parentNode	Родитель текущего узла; если текущий узел является корневым, имеет значение null; свойство эквивалентно с parentElement
previousSibling	Находящийся непосредственно перед текущим и имеющий с ним одного и того же родителя

В качестве примера рассмотрим следующий код:

```
<div id="o1">
    <div id="o2"/>
    <div id="o3">
        <div id="o4"/>
    </div>
    <div id="o5"/>
</div>
```

Следующие булевы выражения будут истинными:

```
o1.firstChild == o2
o1.lastChild == oS
o3.previousSibling == o2
o3.nextSibling == o5
o4.parentNode == o3
o4.parentNode.parentElement.lastChild.previousSibling == o3
```

9.5.3. Идентификация и характеристики узла

Почти для каждого элемента HTML определен атрибут ID, служащий для уникальной идентификации этого элемента. Значением данного атрибута должна быть последовательность из букв, цифр и символа подчеркивания (_), первый символ которой не является цифрой. Если для элемента определен атрибут ID, то значение этого атрибута можно использовать в сценариях JavaScript как имя объекта, соответствующего этому элементу. В качестве примера можно рассмотреть последние строки предыдущего раздела.

MSIE 5.5 допускает дублирование одного и того же значения для ID разных элементов, но в этом случае будет возникать ошибка при попытке обратиться к такому объекту по имени. Если, например, в документе находятся два элемента, атрибут ID которых имеет значение `o1`, то в результате обработки операнда `o1.lastChild == o5` возникнет ошибочная ситуация, и соответствующий сценарий JavaScript выполнен не будет. Отметим, что описываемый атрибут доступен как для чтения, так и для записи.

Уникальный идентификатор для объекта можно сгенерировать с помощью свойства `uniqueID()`. Это свойство возвращает для одного и того же узла дерева одинаковое значение, т. е. многократный вызов `uniqueID()` для одного и того же элемента будет давать идентичный результат. Но результаты вызова `uniqueID()` для одного и того же элемента будут разными при каждой новой загрузке документа браузером.

Помимо уникального идентификатора узлам принадлежат свойства, значениями которых являются характеристики, описанные в табл. 9.3.

Таблица 9.3. *Характеристики узлов и соответствующие им свойства*

Свойство	Значение
`nodeName`	Имя узла, которым является имя элемента или атрибута, либо значение `#text`, если узел текстовый; только для чтения
`nodeType`	Тип узла: 1 — элемент, 3 — текстовый узел
nodeValue	Значение узла, которым является текст узла, если узел текстовый, значение атрибута, если узел является атрибутом, **null**, если узел является элементом
tagName	Имя тега

Все перечисленные в табл. 9.3 свойства, кроме nodeValue, доступны только для чтения.

Значение атрибута можно также получить с помощью метода `getAtribute()`, аргументом которого должно быть имя этого атрибута. Этот метод должен быть вызван для объекта, соответствующего тому элементу, значение атрибута которого требуется получить.

9.5.4. Поиск узла

Иногда бывает необходимо провести поиск элемента в документе или некоторой его части по какому-либо критерию. MSIE 5.5 определяет три таких критерия: уникальный идентификатор (ID), значение атрибута name и имя тега. Далее следует список методов, реализующих поиск по этим критериям:

- `document.getElementById(`*`sID`*`)` — во всем документе находит и возвращает первый объект, соответствующий элементу, атрибут ID которого имеет значение аргумента метода;
- `document.getElementsByName(`*`sName`*`)` — во всем документе находит все объекты, соответствующие элементам, у которых значение либо атрибута ID, либо атрибута name совпадает со значением аргумента sName метода и возвращает коллекцию, состоящую из найденных объектов;
- *`obj`*`.getElementsByTagName(`*`sTagName`*`)` — во всем контенте элемента, представленного объектом `obj`, отыскивает все объекты, соответствующие элементам, имя которых совпадает со значением аргумента `sTagName` метода, возвращает коллекцию, состоящую из найденных объектов.

9.5.5. Проверка на наличие контента и атрибутов

В качестве еще одного инструмента поиска элементов можно назвать метод `contains()`, который возвращает true, если объект, заданный аргументом

метода, находится в контенте элемента, соответствующего объекту, для которого был вызван метод. В противном случае возвращается false. Рассмотрим пример:

```
<div id="o1">
    <div id="o2"/>
    <div id="o3">
        <div id="o4"/>
    </div>
    <div id="o5"/>
</div>
```

Следующие булевы выражения будут истинными:

```
o1.contains(o2)
o1.contains(o4)
o3.contains(o4)
```

а все остальные, представленные всевозможными комбинациями пар объектов, ложными, кроме тех случаев, когда объекты в одной паре имеют одно и то же имя. Например, выражение `o1.contains(o1)` также будет истинным.

Далее следует описание инструментов, помогающих определить, имеются ли у элемента атрибуты и контент, а также может ли элемент обладать контентом:

- *a ttr.* specified — содержит true, если атрибут, соответствующий объекту attr, определен в документе, и false — в противном случае; свойство доступно только для чтения; атрибут может быть определен непосредственно в документе или создан с помощью JavaScript *(см. разд. 9.5.6);*
- *obj*.hasChildNodes() — возвращает true, если элемент, соответствующий объекту obj, содержит элементы или текстовые узлы, и false — в противном случае;
- *obj*.canHaveChildren — содержит true, если элемент, соответствующий объекту obj, может иметь дочерние узлы, и false — в противном случае; свойство доступно только для чтения;
- *obj*.canHaveHTML — содержит true, если элемент, соответствующий объекту obj, может содержать разметку HTML, и false — в противном случае; свойство доступно только для чтения.

9.5.6. Создание, удаление и модификация узлов

Этот раздел содержит описание методов, предназначенных для модификации дерева документа, а также являющихся вспомогательными для этого процесса. Если Web-страница генерируется на основе структурированных данных, то при этом каждая логическая единица информации естественным образом трансформируется в узел или ветвь HTML-документа. Примером

может послужить представление линейного списка с возможностью создания, удаления и редактирования его элементов. В такой ситуации средства JavaScript, соответствующие этим операциям, становятся незаменимыми.

Создание и копирование узлов

Средствами JavaScript можно создавать узлы только двух типов — элемент и текстовый узел. Для создания первого служит метод объекта document *(см. разд. 9.5.1)*

```
createElement(sTagName)
```

Значением обязательного аргумента sTagName этого метода является имя элемента, а возвращаемым значением — объект, являющийся JavaScript-представлением этого элемента.

Текстовый узел создается с помощью метода

```
createTextNode(sText)
```

который также принадлежит объекту document. Значением необязательного аргумента sText при этом является текст создаваемого узла, а возвращаемым значением — объект, являющийся JavaScript-представлением создаваемого узла.

Помимо операций непосредственного создания узлов можно воспользоваться методом, клонирующим узел любого типа

```
obj.cloneNode(bCloneChildren)
```

Будучи вызванным для объекта obj, этот метод создает копию (клон) данного объекта со всеми его потомками, если значением необязательного аргумента bCloneNode является true, и без потомков — в противном случае. Значением по умолчанию для аргумента является false.

Существует возможность копирования не только элементов и текстовых узлов, но и всей совокупности атрибутов, принадлежащих какому-либо элементу. Для этих целей служит метод

```
obj.mergeAttributes(oSource, bID)
```

Который копирует все атрибуты, принадлежащие элементу, представленному объектом bSource, вместе с их значениями в элемент, представленный объектом obj. При этом необязательный аргумент bID служит для того, чтобы указать интерпретатору JavaScript, нужно ли копировать значение атрибута ID. Если значением аргумента является true (по умолчанию), то значение атрибута ID скопировано не будет.

В некоторых ситуациях может возникнуть потребность поменять местами Два узла дерева. Для этого предназначен метод

```
obj.swapNode(oNode)
```

который меняет местами узлы obj и oNode. Возвращаемым значением является объект obj.

Примечание

Использование метода cloneNode() иллюстрируется в *разд. 9.6*.

Встраивание узлов в дерево документа

Основным инструментом встраивания элемента или текстового узла в дерево HTML-документа является метод

obj.appendChild(*oNode*)

который делает узел oNode последним непосредственным потомком элемента, представленного объектом obj, т. е. добавляет его в конец коллекции childNodes этого объекта. Возвращаемым значением при этом **Является** oNode.

Более специфическим является метод

obj.applyElement(*oElement, sWhere)*

добавляющий элемент, представленный объектом oElement, в дерево и делающий его родителем элемента, представленного объектом obj, если значением необязательного аргумента sWhere является строка outside (значение по умолчанию), и непосредственным потомком, если этим значением является inside. В последнем случае элемент, представленный объектом oElement, будет единственным непосредственным потомком элемента, представленного объектом obj, и будет содержать всех его потомков. Возвращаемым значением является встроенный в дерево элемент, контент которого после встраивания в любом случае будет отличаться от контента элемента, представленного объектом oElement.

Для создания или изменения значения атрибута в JavaScript определен метод

obj.setAttribute(*sName, vValue)*

создающий атрибут с именем sName и значением vValue для элемента, представленного объектом obj. Тип значения vValue может быть произвольным (строка, число, булево значение, объект). Если при создании атрибута оказывается, что атрибут с именем sName уже существует, то происходит смена старого его значения на vValue.

Основным средством вставки элемента в конкретное место дерева является метод

obj.insertAdjacentElement(*sWhere,oElement)*

С помощью этого метода реализуется вставка элемента, представленного объектом oElement, в такое место дерева, которое определяется относительно элемента, представленного объектом obj, в зависимости от значения аргумента sWhere. Этими значениями могут быть следующие строки:

- 0 beforeBegin — элемент вставляется непосредственно перед открывающим тегом элемента, представленного объектом obj;
- О afterBegin — элемент вставляется сразу после открывающего тега элемента, представленного объектом obj;
- beforeEnd — элемент вставляется непосредственно перед закрывающим тегом элемента, представленного объектом obj;
- afterEnd — элемент вставляется сразу после закрывающего тега элемента, представленного объектом obj.

При попытке вставить элемент, представленный уже существующим объектом, вставка произведена не будет. В этом и остальных случаях возвращаемым значением будет вставленный объект.

Еще одним, довольно специфическим методом, служащим для встраивания узла в дерево документа, является

```
obj.insertBefore(oNode, oChild)
```

Если не задано значение необязательного аргумента oChild, то этот метод является полностью эквивалентным с методом appendChild(). Если же это значение задано, то узел oNode будет вставлен перед узлом oChild, отсюда и название метода. Возвращаемым значением является вставленный объект.

Примечание

Использование метода `insertAdjacentElement()` **иллюстрируется** ***в разд. 9.6.***

Удаление и замена узлов из дерева документа

Основным методом, служащим для удаления узлов, является метод

```
obj.removeNode(bRemoveChildren)
```

С помощью этого метода элемент, представленный объектом obj, удаляется из Дерева. Если при этом значением необязательного аргумента bRemoveChildren является true, то вместе с этим элементом удаляются и все его потомки. Если же аргумент имеет значение по умолчанию false, то потомки элемента не будут удалены из дерева, а станут потомками его родителя. Возвращаемым значением является объект, соответствующий удаляемому узлу.

Следующий метод эквивалентен предыдущему:

```
obj.removeChild(oNode)
```

Он удаляет первого непосредственного потомка элемента, представленного объектом obj. Непосредственный потомок при этом представлен объектом oNode, являющимся значением аргумента метода. Возвращаемым значением является объект, соответствующий удаляемому элементу. Если узел является элементом, то весь его контент удаляется вместе с ним.

Для удаления одного атрибута служит метод

```
obj.removeAttribute(sName)
```

который удаляет атрибут с именем sName, являющимся аргументом метода, у элемента, представленного объектом obj. При этом возвращаемым значением будет булево значение true, если удаление прошло успешно, и false — в противном случае.

Для удаления всей совокупности атрибутов, принадлежащих какому-либо одному элементу, используется метод

```
obj.clearAttributes()
```

где элемент представлен объектом obj.

Для замены узлов типов "элемент" и "текстовый узел" используются следующие два метода:

- obj.replaceNode(oNewNode) — заменяет узел, представленный объектом obj, на узел, представленный объектом oNewNode, и возвращает при этом объект obj;
- obj.replaceChild(oNewNode, oChildNode) — заменяет непосредственного потомка, представленного объектом oChildNode, принадлежащего элементу, представленному объектом obj, на узел, представленный объектом oNewNode. Возвращаемым значением при этом является объект oChiidNode.

Примечание

Использование метода removeNode() иллюстрируется в разд. 9.6.

9.5.7. Обработка контента элемента

В этом разделе мы будем разделять контент элемента на два вида: внутренний контент, представляющий собой содержимое элемента в обычном смысле, и внешний, который состоит из внутреннего контента, а также открывающего и закрывающего тега элемента. В табл. 9.4 описываются свойства, содержащие как внутренний, так и внешний контент.

Таблица 9.4. *Свойства, значением которых является контент элемента*

Свойство	Описание
innerHTML	Содержит внутренний контент элемента с разметкой HTML
innerText	Содержит внутренний контент элемента без разметки HTML. Совпадает со значением innerHTML, из которого удалена разметка
outerHTML	Содержит внутренний контент элемента с разметкой HTML. При присвоении свойству нового значения элемент, которому принадлежит это свойство, удаляется из документа и заменяется HTML-кодом, являющимся этим значением
outerText	Содержит внешний контент элемента с разметкой HTML. Совпадает со значением outerHTML, из которого удалена разметка. При присвоении свойству нового значения элемент, которому принадлежит это свойство, удаляется из документа и заменяется текстом, являющимся этим значением

Не всегда бывает удобно редактировать весь контент элемента целиком. Действительно, если предположить, что размер этого контента достаточно велик, то легко представить себе объем кода, с помощью которого осуществляется, например, поиск и замена какой-либо части текста в этом контенте. Далее мы опишем методы, служащие для более тонкой обработки контента элемента, а также контента, окружающего этот элемент (не путать с внешним контентом).

Для того чтобы получить значение текстового узла, находящегося в непосредственной близости от закрывающего и открывающего тегов элемента, служит метод

```
obj.getAdjacentText(sWhere)
```

который возвращает текст, местоположение которого относительно элемента, представленного объектом obj, определяется в зависимости от значения текстового аргумента sWhere этого метода. Этот аргумент может принимать одно из четырех значений:

- beforeBegin — метод возвращает значение текстового узла, находящегося непосредственно перед открывающим тегом элемента;
- afterBegin — метод возвращает значение текстового узла, находящегося сразу после открывающего тега элемента;
- beforeEnd — метод возвращает значение текстового узла, находящегося непосредственно перед закрывающим тегом элемента;
- afterEnd — метод возвращает значение текстового узла, находящегося сразу после закрывающего тега элемента.

Следующие два метода служат для вставки соответственно HTML-кода и текста непосредственно перед открывающим тегом элемента или сразу после закрывающего.

```
obj.insertAdjacentHTML(sWhere, sHTML)
obj.insertAdjacentText(sWhere, sText)
```

При этом элемент представлен объектом obj, вставляемый контент — значением второго аргумента соответствующего метода, а место вставки — значением первого аргумента, который может принимать те же значения, что и метод getAdjacentText(). Зависимость места вставки контента от значения аргумента sWhere во всех трех вышеперечисленных методах эквивалентна.

Помимо вставки существует возможность заменить значение текстового узла на новое. Для этих целей используется метод

```
obj.replaceAdjacentText(sWhere, sNewText)
```

Здесь элемент представлен объектом obj, местоположение текстового узла, чье значение требуется изменить, определяется значением первого аргумента swhere, а текст для замены — значением второго аргумента sNewText. При этом аргумент sWhere может принимать те же значения, что и его аналог в методе getAdjacentText(), и зависимость места вставки контента от значения этого аргумента является той же самой.

Последним методом, который мы опишем в этом разделе, является

```
oTextNode.splitText(iIndex)
```

использующийся для разбиения текстового узла, представленного объектом oTextNode, на две части. Разбиение при этом осуществляется следующим образом:

1. Если необязательный аргумент iIndex не задан, метод возвращает объект, соответствующий текстовому узлу без текста.
2. Если iIndex задан, то метод возвращает объект, соответствующий текстовому узлу, содержимым которого является результат вырезания из текстового значения oTextNode текста, начиная с позиции iindex и до его конца. При этом значением oTextNode после разбиения является оставшаяся часть текста, т. е. результат вырезания из текстового значения этого объекта текста, начиная с позиции о, и до позиции, заданной iindex.

Напомним, что значение текстового узла можно получить с помощью свойства nodeValue *(см. разд. 9.5.3)*.

9.5.8. Обработка таблиц

В этом разделе мы опишем методы и свойства, применяющиеся для обработки таблиц, созданных с помощью элемента TABLE. И методы, и свойства принадлежат объекту, соответствующему какому-либо из элементов: TABLE, TBODY, THEAD, TFOOT, TD и TR. Эти свойства (все они доступны только для чтения) представлены в табл. 9.5.

Таблица 9.5. Свойства для обработки таблиц

Свойство	Значение
caption	Объект CAPTION, представляющий заголовок таблицы
cellIndex	Позиция ячейки в строке (TR) таблицы
rowIndex	Позиция строки в таблице
sectionRowIndex	Позиция строки в таблице или в какой-либо из трех ее секций: TBODY, TFOOT, THEAD
tFoot	Объект, соответствующий секции TFOOT таблицы
tHead	Объект, соответствующий секции THEAD таблицы

В табл. 9.6 описаны методы, применяющиеся для обработки таблиц.

Таблица 9.6. Методы для обработки таблиц

Метод	Описание
createCaption()	Создает заголовок (CAPTION) таблицы
createTFoot()	Создает секцию TFOOT таблицы
createTHead ()	Создает секцию THEAD таблицы
deleteCaption()	Удаляет заголовок (CAPTION) таблицы
deleteCell()	Удаляет ячейку таблицы
deleteRow()	Удаляет строку таблицы
deleteTFoot ()	Удаляет секцию TFOOT таблицы
deleteTHead()	Удаляет секцию THEAD таблицы
insertCell()	Вставляет ячейку в таблицу
insertRow()	Вставляет строку в таблицу

9.5.9. Обработка URL

В этом разделе приводится табл. 9.7, содержащая перечень свойств с описаниями, служащих для получения URL документа и его составляющих, а также для обработки различных частей URL-адреса.

Таблица 9.7. *Свойства для обработки URL*

Свойство	Значение
hash	Часть URL-адреса, следующая после символа #
host	Часть URL-адреса, соответствующая имени компьютера и номеру порта
hostname	Часть URL-адреса, соответствующая имени компьютера
href	Для элемента BASE — базовый URL, для элементов A, AREA, LINK — URL адресуемого ресурса, для объекта location — URL соответствующего этому объекту документа, для объекта stylesheet — URL стилевой таблицы
nameProp	Имя файла, содержащегося в значениях свойств href или src элементов A или IMG
port	Часть URL-адреса, соответствующая номеру порта
protocol	Часть URL-адреса, соответствующая протоколу
referrer	URL компьютера, с которого был сделан запрос текущего документа; только для чтения
search	Часть URL-адреса, следующая после символа ?
URL	URL текущего документа
URLUnencoded	Декодированный URL текущего документа (коды символов, недопустимых в URL-адресе, например пробелов, заменены самими символами); только для чтения
A.urn	URN текущего документа

Свойствами hash, host, hostname, port, protocol, search **обладают** элементы A И AREA, **а также объект** location, **а Свойствами** referrer, URL И URLUnencoded — **объект** document.

9.5.10. Обработка таблиц стилей

В этом разделе приведена табл. 9.8, содержащая описания методов, служащих для обработки стилевых таблиц.

Таблица 9.8. *Методы для обработки таблиц CSS*

Метод	Описание
addImport(*URL*)	Добавляет к списку импортируемых стилевых таблиц еще одну таблицу, URL которой представлен аргументом метода

Таблица 9.8 *(окончание)*

Метод	Описание
`addRule` (Selector, *property_value*)	Для стилевой таблицы создает правило, селектор которого представлен первым, а пара свойство-значение — вторым аргументом метода
`createStyleSheet(`*URL*`)`	Создает стилевую таблицу для документа, подключенную к нему с помощью элемента LINK или встроенную в него с помощью элемента STYLE, в зависимости от значения аргумента метода
`removeRule(`*index*`)`	Удаляет правило с номером, представленным аргументом метода

Методы `addImport()`, `addRule()` И `removeRule()` **принадлежат объекту** `styleSheet`, **а метод** `createStyleSheet()` — **объекту** `document`.

9.6. Законченный пример

Суммируя все сказанное ранее об объектной модели HTML-документа, можно сделать вывод, что формирование конечного документа происходит не только на стороне сервера, но и на стороне клиента. И если серверная часть Web-приложения среди прочих задач выполняет обработку структуры контента документа, т. е. информации, которую представляет данный документ, то клиентская помимо простого отображения данных в том или ином виде также способна обрабатывать структуру, но уже не контента, а тех данных, которые отвечают за его представление.

С тем, как происходит обработка структуры данных на стороне сервера, мы Уже познакомились в предыдущих главах книги. То же самое можно сказать и про остальные фазы формирования HTML-документа, протекающие на Различных слоях Web-приложения. Поэтому в данном разделе мы представим пример, который иллюстрирует оба этапа формирования конечного Документа, а именно, формирование HTML-документа на стороне сервера, а также трансформация HTML-документа на стороне клиента в конечный документ, который можно увидеть в окне браузера.

В этом примере мы сосредоточимся на последнем, втором этапе обработки документа, т. к. первый этап был проиллюстрирован, как мы надеемся, достаточно подробно в *главе 4*, *в* которой мы на примерах описали несколько видов поиска информации в XML-документе и предоставление результатов этого поиска пользователю.

Наряду с двумя вышеупомянутыми этапами обработки документа, можно было бы ввести еще один — тот, который соответствует логическому слою Web-приложения. Но, как мы уже говорили ранее, основной задачей логического

слоя Web-приложения является организация совместной работы приложений, обрабатывающих XML-документы, и остальных компонентов серверного программного обеспечения. Примером такой работы может послужить динамическая вставка в XML-документ информации из какой-либо базы данных. Но реализация подобной задачи выходит за рамки этой книги.

Как выходит за ее рамки и описание Java API, предназначенного для динамического генерирования XML-кода, т. е. создание узлов исходного дерева XML-документа. Соответствующие этому API пакеты Java и импортируемые Cocoon перечислены в конце *главы 6,* а некоторые встроенные в Cocoon объекты, являющиеся экземплярами классов из этих пакетов, описаны в *разд. 8.2.* Описание этого API мы не приводим здесь потому, что оно заслуживает отдельной книги, объем которой был бы гораздо больше той, которую вы сейчас держите в руках.

Главная цель рассмотренного ниже примера — продемонстрировать как можно больше возможностей языка JavaScript по обработке дерева HTML-документа, хотя, разумеется, реализация слоев структуры информации, а также логического слоя будет также нами описана. Находясь в рамках поставленной задачи, мы попытаемся предоставить основной набор инструментов, предлагаемых JavaScript, а также методы их использования для разрешения множества типовых ситуаций. Поэтому не стоит рассматривать пример этого раздела как руководство по стилю программирования клиентской части Web-приложения.

С этой стороны, данный пример обладает целым рядом существенных недостатков таких, например, как избыточность кода, неэффективное решение некоторых внутренних задач приложения и т. п. Но, с другой стороны, пример представляет собой лишь демонстрацию возможностей JavaScript по созданию пользовательского интерфейса. Поэтому при его изучении следует обращать внимание, прежде всего, на обработку событий и структуры HTML-документа, которая была описана ранее в этой главе.

9.6.1. Постановка задачи

Описывая архитектуру объектно-ориентированного интерфейса, мы использовали как базовое понятие окна. Как уже упоминалось, приложение, обладающее таким интерфейсом, является либо однооконным, либо многооконным, т. е. его интерфейс представлен главным окном, в котором по мере необходимости создаются дочерние окна, каждое из которых служит для обработки некоторой автономной порции информации. При этом каждая такая порция не обязательно соответствует отдельному документу, и структура всех порций, как правило, идентична.

Web-приложение, обладающее именно таким, многооконным интерфейсом, мы и попытаемся здесь построить. Оно должно быть представлено главным окном, в котором пользователь имеет возможность создавать и удалять до-

черние окна, а также перемещаться между ними. Так как многослойное Web-приложение способно не только представлять информацию, но и ее обрабатывать (главным образом, модифицировать), мы помимо создания пользовательского интерфейса реализуем также возможность сохранения и загрузки информации о дочерних окнах со стороны сервера. В качестве этой информации мы возьмем координаты дочернего окна в трехмерном пространстве *(см. разд. 7.11)*, а также данные, содержащиеся в рабочей области окна, которые пользователь может отредактировать.

Отметим, что на практике вряд ли может возникнуть такая постановка задачи, но, как мы уже говорили ранее, наше Web-приложение является демонстрационным и предназначено, главным образом, для иллюстрации обработки объектной модели HTML-документа. Немного забежим вперед и с помощью рис. 9.2 опишем схему его работы.

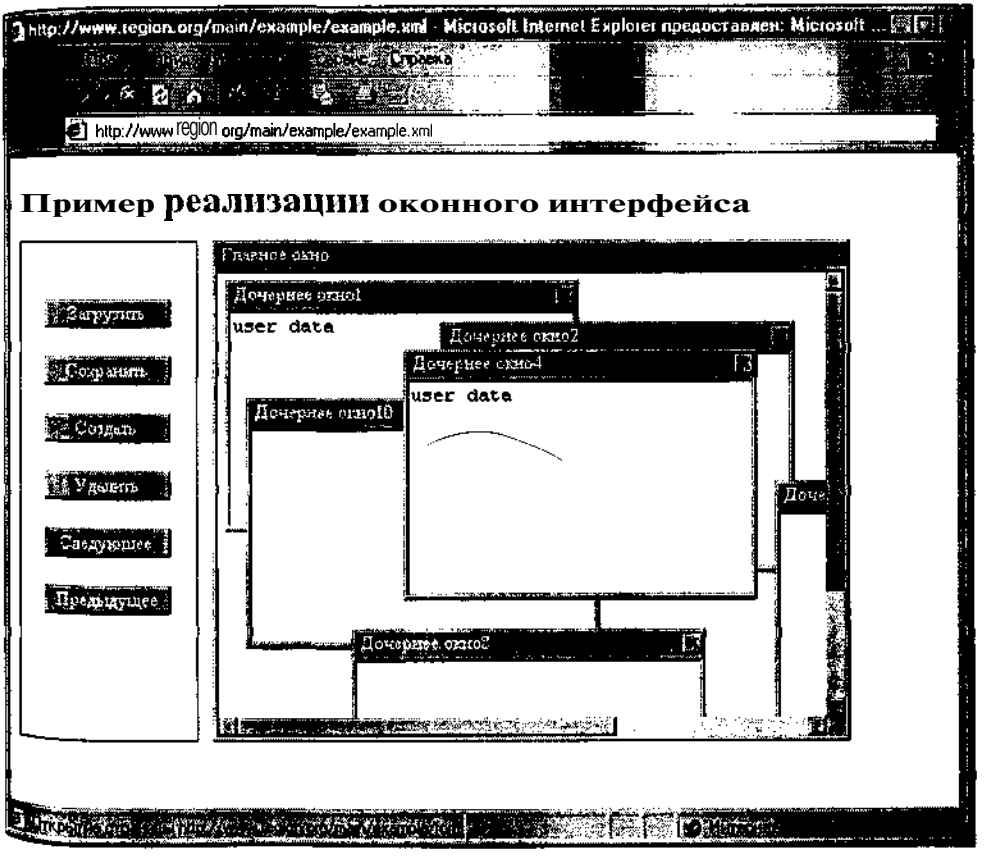


Рис. 9.2. Пример реализации оконного интерфейса

Как видно из рис. 9.2, слева на Web-странице располагается ряд кнопок, к каждой из которых привязана одна из следующих шести операций (сверху вниз):

1. Загрузка ранее сохраненных дочерних окон (здесь имеются в виду не сами окна, а соответствующие им данные) с сервера.
2. Сохранение дочерних окон на сервере.
3. Создание дочернего окна.
4. Удаление дочернего окна.
5. Переход к следующему дочернему окну.
6. Переход к предыдущему окну.

В правой части Web-страницы располагается главное окно, в котором и будут отображаться результаты всех вышеперечисленных операций, кроме сохранения окон.

Помимо операций, соответствующих кнопкам, пользователю будут доступны также такие операции, как перемещение окон внутри главного окна, причем при выходе дочернего окна за пределы главного (вниз или вправо) у последнего будут появляться полосы прокрутки, удаление дочернего окна с помощью кнопки, расположенной в его правом верхнем углу, а также ввод и редактирование информации, содержащейся в дочернем окне. Теперь перейдем к рассмотрению кода Web-приложения.

9.6.2. Реализация приложения

Рассматриваемое Web-приложение имеет следующую файловую структуру:

- example.xml — главный файл, к которому происходит обращение пользователя. При этом его Internet-адресом должно быть `http://www.region.org/main/example/example.xml` **(об установке примеров** *см. приложение 3*);
- form.xml — файл, реализующий все слои рассматриваемого Web-приложения. Результат его совместной с form.xsl обработки представлен на рис. 9.2;
- G form.xsl — файл, содержащий сценарий XSLT для form.xml;
- UIDispatcher.class — сетевой апплет для связи с сервером (о принципе его работы *см. разд. 8.5);*
- example.js — файл, в котором содержатся функции JavaScript, выполняющие те операции, которые сопоставлены с кнопками, а также функция `WndProc()`, реализующая это сопоставление;
- form_handler.xml — файл, являющийся серверным обработчиком формы, т. е. данных, отправленных пользователем со стороны клиента. Представляет собой большую часть логического слоя Web-приложения. Оставшаяся, очень небольшая часть этого слоя реализуется в файле form.xml;

- windows.xml — файл, содержащий информацию о дочерних окнах;
- windows.xsl — файл, содержащий сценарий XSLT для windows.xml.

рассмотрим все эти файлы (кроме файла апплета) по порядку.

Файл example.xml

Сам по себе этот файл не содержит значимого в смысле поставленной задачи кода. Он предназначен только для того, чтобы Web-страница не перезагружалась при подтверждении ввода данных пользователем.

Листинг 9.1. Файл example.xml. Главный файл

```
<?xml version="1.0" encoding="koi8-r"?>
<?xml-stylesheet href="#style1" type="text/xsl"?>
<?cocoon-process type="xslt"?>

<example>
 <head>
  <xsl:stylesheet id="style1"
                  version="1.0"
                  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:template match="example">
    <html>
     <frameset  cols="100%, 0%">
      <frame name="frame1" src="form.xml"/>
      <frame name="frame2" src="about:blank"/>
     </frameset>
    </html>
   </xsl:template>
  </xsl:stylesheet>
 </head>
</example>
```

Как видно из листинга 9.1, мы воспользовались здесь встроенным в XML-Документ стилем XSLT *(см. разд. 5.2)*. Так как, example.xml не содержит Данных в формате XML, а только примитивный HTML-код, нет необходимости создавать два отдельных файла, один из которых содержал бы код XML, а другой сценарий XSLT для трансформации этого кода.

В результате обработки этого файла мы получим HTML-документ, содержимым которого будет содержимое правила шаблона для элемента example.

Данный HTML-код служит для создания двух фреймов, первый из которых имеет имя frame1 и предназначен для вывода результатов обработки остальных файлов Web-приложения за исключением form_handler.xml. Второй, с именем frame2, предназначен для вывода результатов обработки данных, введенных пользователем в форму *(см. разд. 8.1)*.

Первый фрейм имеет размеры, равные размерам окна браузера, а второй не отображается, т. к. имеет нулевую ширину. Поэтому, когда пользователь подтверждает ввод данных в форму, происходит перезагрузка не всей Web-страницы, видимая часть которой представлена фреймом frame1, а только той, в которую выводятся результаты подтверждения пользователем ввода данных в форму. Таким образом, перезагрузки содержимого окна браузера фактически не происходит.

Заметим, что использование фреймов влечет за собой появление границы (border) вокруг рабочей области окна браузера. Но с этим можно справиться. Как именно, мы покажем в *разд. П2.6.1*, а здесь заметим, что фрейм frame2 можно использовать в отладочных целях. Если потянуть мышью за правую границу рабочей области окна браузера, которая является одновременно и правой границей фрейма frame1, то мы увидим содержимое frame2. Таким образом, если при обработке файла form_handler.xml возникнет критическая ошибка, то сообщение о ней появится в окне, соответствующем этому фрейму. В этом же окне можно выводить отладочную информацию (например, содержимое каких-либо переменных с помощью директивы xsp:expr) в том случае, когда связка из формы и ее обработчика работает не так, как требуется.

Файл form.xml

В этом файле формируется структура Web-страницы. В нем примечательным является один из двух представленных в Web-приложении способов загрузки данных с сервера.

Листинг 9.2. Файл form.xml, реализующий многооконный интерфейс

```
<?xml version="1.0" encoding="koi8-r"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="form.xsl" type="text/xsl"?>

<xsp:page language="java"
          xmlns:fp="http://apache.org/cocoon/XSP/FP/1.0"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">
```

```
<form>
<template_wnd/>
 <main_wnd>
  <fp:resource id="windows">
   <fp:resource-file>windows.xml</fp:resource-file>
   <fp:resource-node>items</fp:resource-node>
  </fp:resource>
  <fp:read from="windows" select="./*" as="node"/>
 </main_wnd>
 <buttons>
  <button command="load">Загрузить</button>
  <button command="save">Сохранить</button>
  <button command="create">Создать</button>
  <button command="delete">Удалить</button>
  <button command="next">Следующее</button>
  <button command="prev">Предыдущее</button>
 </buttons>
</form>
</xsp:page>
```

Рассмотрим последовательно все представленные в листинге 9.2 узлы исходного дерева XML-документа. Первый элемент — `template_wnd` — служит для представления окна, которое является шаблоном для создания дочерних окон. Далее мы покажем, как с его помощью создаются эти окна.

После `template_wnd` следует элемент `main_wnd`, служащий для представления главного окна, в котором будут содержаться остальные окна. В контенте этого элемента происходит чтение данных из файла windows.xml. Как мы Уже говорили при описании листинга 9.1, эти данные представляют собой информацию о каждом, ранее созданном дочернем окне. Операция чтения осуществляется с помощью элементов, принадлежащих пространству имен с префиксом `fp:`, которое было описано нами в *разд. 6.6.*

Мы использовали данные элементы потому, что их совокупность напоминает язык запросов к содержимому XML-документа. Было бы гораздо более удобным применить соответствующий API, написанный на Java, стандарт которого утвержден консорциумом W3C. Но поскольку темой книги является язык XML, а не программирование на Java, то мы будем стараться везде, где это возможно, пользоваться конструкциями этого языка, хотя и не будем избегать применять средства, предлагаемые языком Java, но для других целей.

Здесь важно подчеркнуть, что употребление элементов из пространства имен fp на практике должно быть сведено к минимуму, т. к. реализованные

с их помощью операции чтения и записи внешних XML-данных выполняются весьма своеобразно, а иногда и просто некорректно. По нашему мнению, лучше всего исключить применение этих элементов вовсе. Но в чисто демонстрационных целях мы решили показать и этот способ обработки внешних XML-данных в качестве иллюстрации (хотя и бедной) общих принципов работы языка запросов.

В листинге 9.2, равно как и в тех, которые поставляются на CD-ROM, XML-код, отвечающий за вышеупомянутые операции, работает без ошибок. Мы не будем подробно описывать результаты его обработки в силу вышеуказанных его недостатков, а ограничимся здесь и далее только ссылками на ту информацию, которая представлена в *разд. 6.6.*

За элементом `main_wnd` следует элемент `buttons`, служащий для представления окна, содержащего кнопки, описанные в *разд. 9.6.1.* При этом каждая из этих кнопок представлена элементом `button`, контентом которого является надпись для соответствующей кнопки. Каждому из этих элементов принадлежит атрибут `command`, по значению которого определяется, какую из команд требуется выполнить, когда пользователь нажмет кнопку, соответствующую данному элементу `button`. Как видно из листинга 9.2, таких команд шесть, как и кнопок, соответствующих этим командам: `load` (загрузить дочерние окна с сервера), `save` (сохранить окна на сервере), `create` (создать дочернее окно), `delete` (удалить дочернее окно), `next` (сделать активным следующее дочернее окно) и `prev` (сделать активным предыдущее дочернее окно).

Файл form.xsl

Этим файлом представлен один из двух содержащихся в примере сценарий XSLT. Вторым является example.xsl, но содержащиеся в нем правила трансформации узлов исходного дерева несравненно беднее и играют гораздо менее важную роль, чем те, которые представлены в данном листинге.

Листинг 9.3. Файл form.xsl. Сценарий XSLT для form.xml

```
<?xml version="1.0" encoding="koi8-r"?>

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

<xsl:template match="form">
 <xsl:processing-instruction name="cocoon-format">↳
  type="text/html"↳
```

```
</xsl:processing-instruction>
<html>
 <head>
  <script src="example.js"/>
  <script>
   var counter = 1;
   var current_wnd = null
  </script>
  <applet height="0" width="0" id="UIDispatcher"
          code="UIDispatcher.class" style="display:none"/>
 </head>
<body onmouseup="if (current_wnd != null)
                 current_wnd.firstChild.status='fixed'"
      onselectstart="if (event.srcElement.tagName != 'TEXTAREA') return
false">
 <h1>Пример реализации оконного интерфейса</h1>
 <form id="saver" method="post" enctype="multipart/form-data"
       action="form_handler.xml" target="frame2"/>
 <input type="hidden" value=""/>

 <xsl:call-template name="template_wnd"/>
 <xsl:apply-templates/>
</body>
</html>
</xsl:template>

<xsl:template name="template_wnd">
 <xsl:param name="top" select="5"/>
 <xsl:param name="left" select="5"/>
 <xsl:param name="number" select="0"/>
 <xsl:param name="zIndex" select="0"/>
 <xsl:param name="data"/>
 <xsl:param name="visibility" select="'hidden'"/>
 <div id="_window{$number}" number="{$number}"
      style="position:absolute;top:{$top};left:{$left};
             width:250;height:175;
             border-style:outset;border-width:2;
             background-color:#8399B1;visibility:{$visibility};
             z-index:{$zIndex}"
```

```
      oncontextmenu= "activateWnd(this)"
      onclick = "activateWnd(this)">
   <div style="width:100%;height:17;background-color:gray;
               color :white;padding-left:5;cursor:default"
      currPosX="0" currPosY="0" status="fixed"
      onmousedown="activateWnd(this.parentNode);this.status='moved';
                   this.currPosX=event.clientX;
                   this.currPosY=event.clientY"
      onmouseup="this.status='fixed'"
      onmousemove="move_wnd(this,event.clientX,event.clientY)">
    Дочернее окно<xsl:if test="not($number=0)">
                   <xsl:value-of select="$number"/>
                 </xsl:if>
   <button
     style="position: absolute; right: 0 ; top: 2 ; height : 14; font-weight: bold"
     onclick="deleteWnd()">
      <div style="position:absolute;top:-6;cursor:default">x</div>
   </button>
   </div>
   <divstyle= "position:relative; top:1;width:100%;height:151 ">
    <textarea style="width:100%;height : 100%;overflow:auto">
     <xsl:value-of select="$data"/>
    </textarea>
   </div>
 </div>
</xsl:template>

<xsl:template match="main_wnd">
 <div id="{name()}"
      style="position:absolute;top:60;left:150;width:400;height:350;
             border-style:outset;border-width:2;
             background-color:#8399B1">
   <div style="width:100%;height:17;background-color:#506880;
               color:white;padding-left:5;cursor:default">
    Главное окно
   </div>
   <div style="position:relative;top:2;width:446;height:327;
               border-style:inset;border-width:2;
```

```
            background-color:white;overflow:auto"
      onmousemove="if (current_wnd != null)
                  move_wnd(current_wnd.firstChild,
                           event.clientX,event.clientY)">
   <xsl:apply-templates/>
   <xsl:if test="not(count(window)=0)">
    <script>
     var zIndex;
     var max=main_wnd.lastChild.childNodes[0].style.zIndex;
     current_wnd=main_wnd.lastChild.childNodes[0];
     for (var i=1; i &lt; <xsl:value-of select="count(window)"/>; i++) {
         zIndex=main_wnd.lastChild.childNodes[i].style.zIndex;
         if (max &lt; zIndex) { max=zIndex;
            current_wnd=main_wnd.lastChild.childNodes(i) }}
     counter=max+1;
     current_wnd.firstChild.style.backgroundColor='#506880';
     main_wnd.lastChild.lastChild.removeNode(true)
    </script>
   </xsl:if>
  </div>
 </div>
</xsl:template>

<xsl:template match="window">
 <xsl:call-templatename="template_wnd">
  <xsl:with-param name="top" select="top"/>
  <xsl:with-param name="left" select="left"/>
  <xsl:with-param name="data" select="data"/>
  <xsl:with-param name="number" select="number"/>
  <xsl:with-param name="zIndex" select="zIndex"/>
  <xsl:with-param name="visibility" select="'visible'"/>
 </xsl:call-template>
</xsl:template>

<xsl:template match="buttons">
 <divstyle="position:absolute;top:60;;left:10;width:130;;height:350;
            border-style:solid;border-width:1">
  <xsl:apply-templates/>
```

```
 </div>
</xsl:template>

<xsl:template match="button">
<div style="position:absolute;left:20;top:{position()*20};
            width:90;height:20;
            background-color:#8399B1;text-align: center;color:white;
            border-width:2;border-style:outset;cursor:default"
  onmousedown="this.setCapture();this.style.borderStyle='inset'"
  onmouseup="this.releaseCapture();this.style.borderStyle='outset'"
  onclick="WndProc('{@command}')">
  <xsl:apply-templates/>
</div>
</xsl:template>

</xsl:stylesheet>
```

Первым в листинге 9.3 следует правило шаблона для элемента документа, которым является элемент form (см. листинг 9.2). В нем определяется элемент HTML-документа с именем html, а также его непосредственные потомки — элементы head и body. Первый служит для подключения к Web-странице внешних сценариев JavaScript и сетевого апплета, представленного файлом UIDispatcher.class.

В контент элемента head, помимо всего прочего, входит короткий сценарий JavaScript, назначение которого мы опишем позднее. В следующих разделах будут описаны также обработчики событий onmouseup и onselectstart, перехватываемых объектом, соответствующим элементу body, и назначение пустых элементов form и input, которые используются при отправке данных на сервер (команда save). Вообще, все то, что касается операций над окнами, реализованных на JavaScript, мы опишем в *разд. 9.6.3—9.6.8.*

Здесь мы последовательно рассмотрим правила шаблонов для элементов `template_wnd` (именованное правило шаблона, которое применяется к документу с помощью директивы `xsl:` call-template, об использовании которой *См. разд. 2.6.7),* main_wnd, window, buttons И button.

Правило шаблона для элемента `template_wnd` служит для описания представления дочернего окна. Это окно в HTML-дереве представлено элементом div, с идентификатором (атрибут id), начинающимся со слова `_window` и оканчивающимся номером этого окна. Как видно из листинга 9.3, этот номер определяется параметром с именем number. Помимо number в рассматриваемом правиле шаблона имеются следующие параметры его вызова:

- top — содержит *y*-координату верхней границы дочернего окна;
- left — содержит x-координату левой границы дочернего окна;
- data — содержит данные для рабочей области дочернего окна;
- zIndex — содержит *z*-координату дочернего окна *(см. разд. 7.11.5)*;
- visibility — содержит индикатор видимости дочернего окна, по умолчанию окно невидимо.

При просмотре всего правила шаблона легко заметить, как значения вышеперечисленных параметров встраиваются в HTML-представление, являющееся результатом вызова именованного шаблона template_wnd. Это встраивание происходит с помощью конструкции {$имя_параметра}, о которой можно прочитать в *разд. 2.8* и *3.3*. Например, значение параметра с именем top присваивается одноименному свойству CSS с помощью конструкции top:{$top}.

Примечание

Везде далее при описании стилей CSS, где не указаны единицы длины, по умолчанию подразумеваются пикселы (px, *см. разд. 7.10.2)*. Например, если где-либо встречается определение top: 10;left:20, то его следует читать, как top: 10px;left: 20px. Такая сокращенная форма записи обеспечивается браузером Microsoft Internet Explorer, и хотя она является не совсем корректной, мы рекомендуем ее использовать, т. к. с ее помощью объем описаний стилей становится меньше, что влияет на скорость загрузки Web-страниц.

В начале книги мы описали все те компоненты, из которых может состоять окна (см, рис. 9.1). Рассматриваемое правило реализует только некоторые из этих компонентов, а именно, трехмерные границы окна, заголовок и одну из кнопок управления окном, т. к. создание окна, содержащего все представленные на рис. 9.1 компоненты, является слишком громоздкой задачей.

В данном примере границы окна не используются для изменения его размеров, реализацию этого механизма мы включили в один из примеров, находящихся на CD-ROM *(см. разд. П2.6.1)*. Прорисовка границ окна происходит с использованием стиля CSS, описанного для элемента div, представляющего дочернее окно и являющегося в рассматриваемом шаблоне элементом верхнего уровня. Объект, соответствующий этому элементу, отвечает за обработку событий oncontextmenu и onclick, которые активизируют окно, если пользователь щелкнет по нему какой-либо из кнопок мыши.

Заголовок дочернего окна представлен элементом div, являющимся первым непосредственным потомком элемента div, соответствующего всей области дочернего окна и описанного в предыдущем абзаце. Объект, являющийся JavaScript-представлением заголовка, отвечает за обработку трех событий от мыши: onmousedown, onmousemove И onmouseup, ИСПОЛЬЗУЮЩИХСЯ ДЛЯ пере-

мещения данного дочернего окна в пределах главного. Первое из этих со, бытии служит еще и для активизации дочернего окна, если пользователь над его заголовком нажмет левую кнопку мыши.

В элементе div, соответствующем заголовку, содержится текст этого заголовка, формируемый склейкой строки "Дочернее окно" и номером этого окна, если этот номер не нулевой. Кроме текста заголовка, в его области находится кнопка закрытия окна, представленная элементом button.

Последним элементом в контенте div, соответствующим дочернему окну в целом, является элемент также с именем div, содержащий элемент textarea, который представляет рабочую область рассматриваемого дочернего окна и служит для ввода и редактирования сопоставленных с этим окном данных. При вводе данных в области, соответствующей textarea, допускаются символы русского языка.

Правило шаблона, служащее для трансформации элемента `main_wnd`, во многом похоже на правило шаблона для `template_wnd`. Действительно, если посмотреть на рис. 9.2, то можно заметить, что внешний вид главного и дочернего окон почти ничем не отличается. Но HTML-представление этих окон обуславливает различия в их функциональности. Так, главное окно, в отличие от дочерних, нельзя удалить или переместить, потянув его мышью за заголовок. Поэтому в контенте элемента div (в листинге 9.3 ему присваивается идентификатор main_wnd), представляющего главное окно, обрабатывается только одно событие — onmousemove (объектом, соответствующим его заголовку), обработчик которого осуществляет перемещение активного дочернего окна. В контенте этого div содержится сценарий JavaScript, который выполняется при загрузке данных из файла windows.xml, описанного ранее, при описании файла form.xml. Принцип работы этого сценария мы подробно рассмотрим *в разд. 9.6.8.*

Примечание

Везде далее, когда мы будем говорить, что дочерние окна содержатся в родительском окне, мы будем иметь в виду, что элементы, соответствующие дочерним окнам, являются непосредственными потомками того элемента, который представляет рабочую область родительского окна, а не все это окно целиком, т. к. помимо рабочей области оно содержит еще и заголовок. Рабочую область родительского окна представляют второй и последний непосредственный потомок элемента с идентификатором main_wnd. Таким образом, объектом, соответствующим рабочей области родительского окна, будет либо `main_wnd.firstChild.nextSibling`, либо `main_wnd.lastChild`.

Далее в листинге 9.3 следует правило шаблона для элемента window. Но если просмотреть листинг 9.2, то можно заметить, что элементы с таким именем в нем отсутствуют. Для чего же тогда предназначено это правило шаблона? Ответ на этот вопрос несложен. Это правило шаблона применяется

для трансформации тех элементов window, которые были загружены в файле form.xml с помощью элементов, принадлежащих пространству имен fp.

рассматриваемое правило шаблона выполняет только одну операцию: вызывает правило шаблона с именем `template_wnd` и параметрами, значениями которых является контент потомков элемента window. О том, что является этим контентом, мы расскажем далее при описании файла windows.xml, в котором и содержатся загружаемые элементы window. Этот файл представлен листингом 9.5.

Заметим, что применение правила шаблона для элементов window происходит в правиле шаблона для элемента `main_wnd`. Таким образом, процесс загрузки данных с сервера происходит в четыре этапа, описываемых далее.

1. Загрузка элементов window вместе с их контентом в контент элемента `main_wnd`.
2. Применение к каждому из этих window соответствующего правила шаблона.
3. Вызов для каждого из window правила шаблона с именем `template_wnd`, что влечет за собой создание дочернего окна для каждого из этих window.
4. Некоторые дополнительные действия (активизирование окна, которое было активным в момент сохранения данных обо всех дочерних окнах), выполняющиеся в сценарии JavaScript, который содержится в правиле **шаблона** ДЛЯ `main_wnd`.

Последним правилом, о котором мы здесь расскажем, является правило шаблона для элемента buttons. В результате применения этого правила в HTML-документе создается прямоугольная область, содержащая кнопки, каждая из которых представлена элементом button. О том, как создать кнопку, т. е. придать прямоугольной области внешний вид кнопки и заставить ее реагировать на действия пользователя, мы рассказывали в конце *разд. 9.4,* поэтому не будем здесь останавливаться на этом подробно. Заметим только, что при нажатии на кнопку, будет сгенерировано событие onclick, обработчиком которого является функция `WndProc()`. Эта функция представляет собой классификатор действий пользователя, рассмотренный нами в *разд. 9.1.* Подробнее о принципе ее работы мы расскажем при описании листинга 9.7, в котором представлены все функции JavaScript, используемые в Web-приложении.

Файл form_handler.xml

Обработчик формы, представленный в файле form_handler.xml, выполняет только одну задачу — записывает в файл windows.xml данные, сопоставленные с теми дочерними окнами, которые создал пользователь на стороне клиента.

Листинг 9.4. Файл form_handler.xml. Обработчик формы

```
<?xml version="1.0"?>
<?cocoon-process type="xsp"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="#style1" type="text/xsl"?>

<xsp:page language="java"
          xmlns:fp="http://apache.org/cocoon/XSP/FP/1.0"
          xmlns:xsp="http://www.apache.org/1999/XSP/Core">

  <xsp:structure>
   <xsp:include>param_utils.*</xsp:include>
  </xsp:structure>

<doc>
<head>
<xsl:stylesheet id="style1"
                version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="doc">
 <html>
  <xsl:apply-templates select="handler"/>
 </html>
</xsl:template>

<xsl:template match="handler">
  <html>
   <xsl:apply-templates/>
   <script defer="true">
    location.href="about:blank";
   </script>
  </html>
</xsl:template>

</xsl:stylesheet>
</head>
 <handler>
  <xsp:logic>
```

```
ServletInputStream is = request.getInputStream();
BufferedReader br = new BufferedReader(new ↳
InputStreamReader(is,"UTF8"));
StringBuffer sBuf = new StringBuffer();
String 1;
try {
  1=br.readLine();
  while (1!=null)     {
    sBuf.append(l);
    1=br.readLine(); }
    } catch (IOException e) { System.out.println(e); };
String header = request.getHeader("Content-Type");
int index = header.lastIndexOf("boundary=");
String boundary = header.substring(index + 9);
boundary = "--" + boundary;

Params p = new Params(sBuf,boundary);

int number=Byte.decode(p.getValueByName("number")).intValue();
String wnd_info="";
int ind=0;

<![CDATA[
String data="<?xml version=\"1.0\"?>\r\n";
data += "<?cocoon-process type=\"xslt\"?>\r\n";
data += "<?xml-stylesheet href=\"windows.xsl\" ↳
type=\"text/xsl\"?>\r\n";
data += "\r\n";
data += "<data>\r\n";
data += "<items>\r\n";
data += "<empty/>\r\n";
for (int i=0; i < number; i++) {
    wnd_info = p.getValueByName("window"+String.valueOf(i));
    ind=wnd_info.indexOf(";");
    data += " <window>\r\n";
    data += "  <top>"+wnd_info.substring(0,ind)+"</top>\r\n";
   . data += " ↳
<left>"+wnd_info.substring(ind+1,wnd_info.indexOf(";",ind+1))+↳
"</left>\r\n";
    ind=wnd_info.indexOf(";",ind+1);
```

```
        data += " 
<zIndex>"+wnd_info.substring(ind+1,wnd_info.indexOf(";",ind+1))+
"</zIndex>\r\n";
        data += " 
<number>"+wnd_info.substring(wnd_info.lastIndexOf(";")+1)+
"</number>\r\n";
        data += " 
<data>"+p.getValueByName("data"+String.valueOf(i))+"</data>\r\n";
        data += " </window>\r\n"; }
    data += "</items>\r\n";
    data += "</data>"; ]]>

    String abspath = XSPUtil.relativeFilename("",request,servletContext);
    File f = new File(abspath + File.separator + "windows.xml");

    try {
      OutputStreamWriter osw=new OutputStreamWriter(new 
FileOutputStream(f), "UTF8");
      osw.write(data,0,data.length());
      osw.flush();
      osw.close(); }
    catch(Exception ex) { System.out.println(ex.toString()); }
  </xsp:logic>
 </handler>
</doc>
</xsp:page>
```

Как видно из листинга 9.4, файл form_handler.xml так же, как и example.xml, содержит встроенный стиль XSLT. Так как рассматриваемый файл реализует логический слой Web-приложения, и, следовательно, содержащиеся в нем данные не нуждаются в представлении, то нет необходимости создавать отдельный файл, отвечающий за это представление.

Но, тем не менее, в результате трансформации form_handler.xml получается некоторый HTML-код, который содержит сценарий JavaScript, представленный в правиле шаблона для элемента handler. Назначение этого сценария заключается в том, чтобы очищать окно, соответствующее фрейму `frame2`. Это сделано для того, чтобы при перезагрузке Web-страницы браузер не спрашивал у пользователя, повторить ли отправку данных формы, чтобы заново сгенерировать тот HTML-документ, который содержится В окне `frame2`.

Далее после описания стиля следует процесс формирования содержимого файла windows.xml. Он основывается на получении значений параметров

HTTP-запроса, который был сгенерирован пользователем при нажатии кнопки **Сохранить.** Причем получение значений этих параметров происходит так, как это было нами описано в *разд. 8.4.* А подключение (импорт) отвечающих за эту операцию Java-классов описано в *приложении 3.*

Контент файла windows.xml создается последовательным накоплением данных в переменной data таким образом, что в результате мы получаем XML-документ, контент которого структурирован так, как это можно увидеть в листинге 9.5. Заранее мы не знаем, сколько окон создаст пользователь, поэтому нам потребуется отправить их количество в качестве значения отдельного параметра HTTP-запроса. Мы назвали этот параметр number. После того, как обработчиком прочитано значение этого параметра, организуется цикл от о до number, в теле которого формируется информация о каждом дочернем окне.

Отметим, что данные, сопоставленные с дочерним окном, отправляются в качестве значений двух параметров HTTP-запроса. Первый содержит значения top, left, zIndex и номер дочернего окна, разделенные точкой с запятой (;), а второй — данные из его рабочей области. Поэтому бульшая часть тела цикла содержит Java-код поиска точки с запятой в значении параметра, а также вырезки из него тех частей, которые соответствуют вышеперечисленным характеристикам. Именем первого из этих параметров является window + *номер окна,* а второго — data + *номер окна.*

Файл windows.xml

Этот файл представляет собой весьма облегченную версию единого хранилища информации. Данные, содержащиеся в этом файле, служат для описания внешнего вида окна, а именно его координат, от которых зависит, будет ли окно активным и, если нет, то какие окна будут его перекрывать.

Листинг 9.5. Файл windows.xml, описывающий дочерние окна

```
<?xml version="1.0"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="windows.xsl" type="text/xsl"?>

<data>
<items>
<empty/>
 <window>
  <top>5</top>
  <left>5</left>
```

```
  <zIndex>1</zIndex>
  <number>1</number>
  <data>user data</data>
</window>
<window>
  <top>34</top>
  <left>157</left>
  <zIndex>8</zIndex>
  <number>2</number>
  <datax/data>
</window>
<window>
  <top>53</top>
  <left>131</left>
  <zIndex>10</zIndex>
  <number>4</number>
  <data>user data</data>
</window>
<window>
  <top>145</top>
  <left>393</left>
  <zIndex>9</zIndex>
  <number>6</number>
  <datax/data>
</window>
<window>
  <top>248</top>
  <left>94</left>
  <zIndex>7</zIndex>
  <number>8</number>
  <datax/data>
</window>
<window>
  <top>87</top>
  <left>20</left>
  <zIndex>6</zIndex>
  <number>10</number>
```

```
  <data></data>
 </window>
</items>
</data>
```

Как видно из листинга *9.5,* каждое окно описывается элементом window, в контенте которого присутствуют следующие элементы-потомки: top, left, zIndex, data и number, соответствующие координатам, данным рабочей области и номеру дочернего окна.

Файл windows.xsl

В функции документа, представленного файлом windows.xml. входит не только хранение информации, но и ее представление, которое осуществляется с помощью сценария XSLT, находящегося в файле windows.xsl, листинг которого следует далее.

Листинг 9.6. Файл windows.xsl. Сценарий XSLT для windows.xml

```
<?xml version="1.0" encoding="koi8-r"?>

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

<xsl:template match="data">
 <div>
  <xsl:apply-templates/>
 </div>
</xsl:template>

<xsl:templatematch="window|top|left|number|zIndex">
 <div>
  <xsl:apply-templates/>
 </div>
</xsl:template>

</xsl:stylesheet>
```

Представление информации, содержащейся в windows.xml, понадобится нам При загрузке данных с сервера. Поэтому содержимое правил шаблона листинга 9.6 будет объяснено нами при описании этой операции *(см. Разд. 9.6.8).*

Файл example.js

В этом файле содержатся все функции JavaScript, реализующие все те операции, которые может осуществить пользователь над дочерними окнами.

Листинг 9.7. Файл example.js.
Сценарий JavaScript, описывающий функции обработки окон

```
function WndProc(command) {
 switch (command) {
  case 'load'   : loadData(); break;
  case 'save'   : saveData(); break;
  case 'create' : createWnd(); break;
  case 'delete' : deleteWnd(); break;
  default : goToWnd(command) }}

function activateWnd(obj) {
 if (current_wnd != null)
    current_wnd.firstChild.style.backgroundColor='gray';
 current_wnd = obj;
 current_wnd.firstChild.style.backgroundColor='#506880';
 var wnd = main_wnd.lastChild.firstChild;
 if (obj.style.zIndex!=0)
    while (wnd != null) {
     if (wnd.style.zIndex > obj.style.zIndex) wnd.style.zIndex--;
     wnd = wnd.nextSibling }
 current_wnd.style.zIndex=counter-1 }

function createWnd() {... }
function deleteWnd() {... }
function move_wnd(obj, nX, nY) {... }
function goToWnd(command) {... }
function loadData() {... }
function saveData() {... }
```

В листинге 9.7 приведены тела только двух функций — WndProc() и activateWnd(), остальные функции представлены только своими объявлениями. Каждую из них мы опишем в том разделе, название которого соответствует операции, выполняемой этой функцией.

функция wndProc(*command*), принимая *в* качестве своего значения название команды, которую требуется выполнить, осуществляет вызов функции, соответствующей этой команде. Напомним, что пользователь выполняет операции «ад дочерними окнами, нажимая кнопки, находящиеся в левой части Web-страницы. Каждая из этих кнопок представлена элементом button исходного дерева, имеющим атрибут command, значением которого является команда, сопоставленная с данной кнопкой (см. листинг 9.2). Таким образом, в зависимости от значения своего аргумента функция wndProc(), используя конструкцию switch-case, вызывает одну из функций, перечисленных в листинге 9.7.

функция activateWnd(*obj*) используется для активирования какого-либо одного дочернего окна. Она может быть вызвана в двух случаях: когда пользователь щелкнет любой кнопкой мыши по области какого-либо дочернего окна, а также после того, как будет выполнена операция загрузки. В последнем случае активизируется то окно, которое было активным в момент сохранения. В обоих случаях активизируемое окно представлено объектом obj, являющимся аргументом функции.

Операция активизирования окна заключается в следующем. Если активизируемое окно не является единственным дочерним окном, то цвет заголовка того окна, которое было активным, меняется на серый. Цвет заголовка активизируемого окна меняется, становится синим (#506880). *z*-координате (свойство zIndex) присваивается значение, равное общему количеству всех дочерних окон. При этом количество всех дочерних окон плюс единица содержится в переменной counter (см. листинг 9.3). Для тех дочерних окон, у которых величина zIndex больше *z*-координаты активизируемого окна, значение этого свойства уменьшается на единицу. Это делается для того, чтобы активизируемое окно имело максимальную *z*-координату, т. е. располагалось поверх остальных дочерних окон, а порядок расположения последних вдоль оси *z* оставался прежним, а также, чтобы в главном окне не появлялось двух дочерних окон с одинаковым значением свойства zindex.

В качестве объекта, соответствующего активизированному окну, выступает current_wnd, который служит для представления текущего окна (см. листинг 9.3). Если значением этого объекта является null, то это означает, что главное окно не содержит дочерних окон. Если же дочерние окна имеются, то текущим элементом является активное окно.

9.6.3. Создание окна

За создание дочернего окна отвечает следующая функция:

```
function createWnd() {
 var obj = _window0.cloneNode(true);
 obj.style.visibility = 'visible';
```

```
obj.id = '_window'+counter;
obj.number = counter;
obj.firstChild.firstChild.nodeValue+=counter;
obj.style.pixelTop += 30*((counter-1)%8);
obj.style.pixelLeft = obj.style.pixelTop;
main_wnd.lastChild.insertAdjacentElement('beforeEnd',obj);
counter++;
activateWnd(obj) }
```

Операция создания окна реализуется клонированием узла, соответствующего элементу div, представляющему шаблонное окно. Напомним, что шаблонное окно создается вызовом именованного правила шаблона с именем `template_wnd` (см. листинг 9.3). В результате клонирования создается объект obj, который так же, как шаблонное окно, является JavaScript-представлением элемента div.

После того, как копия шаблонного окна создана, для объекта obj определяются следующие характеристики:

1. Окно становится видимым.
2. Объекту присваивается уникальный идентификатор.
3. Окну присваивается номер (свойство number, которому соответствует атрибут, созданный нами в правиле шаблона `template_wnd`, см. листинг 9.3), равный значению переменной counter.
4. К заголовку окна добавляется его номер. Теперь текстом заголовка является Дочернее `окно`N, где N — номер окна.
5. Вычисляются координаты окна. Свойства pixelTop и pixelLeft служат для получения или присваивания значений соответственно свойствам top и left, но координаты, заданные значениями первых, выражаются в пикселах *(см. приложение 1).*

Далее дочернее окно с вышеописанными характеристиками встраивается в главное (объект `main_wnd`), активизируется, и значение счетчика окон увеличивается на единицу.

9.6.4. Удаление окна

За удаление дочернего окна отвечает следующая функция:

```
function deleteWnd() {
 if (current_wnd == null) return false;
 var obj = current_wnd;
 if (current_wnd.previousSibling != null)
```

```
    activateWnd(current_wnd.previousSibling);  else
    if (current_wnd.nextSibling != null)
       activateWnd(current_wnd.nextSibling); else couneter=1;
obj.removeNode(true) }
```

В теле этой функции выполняются следующие действия:

1. Если активного окна не существует, а это означает, что в главном окне не содержится дочерних окон, то выполнение функции завершается. Текущему окну соответствует объект current_wnd (см. листинг 9.3).
2. В значении obj запоминается объект, соответствующий текущему окну.
3. Проверяется, есть ли непосредственно перед элементом, соответствующим удаляемому окну, еще один такой же элемент. Если такой элемент существует, то соответствующее ему окно становится активным, если нет, то происходит проверка, следует ли за удаляемым элементом еще один такой элемент. Если есть, то активизируется окно, располагающееся сразу после удаляемого. Если условия обеих проверок не выполнены, то это означает, что удаляемое окно является единственным, и активизирование какого-либо другого окна не происходит, а счетчику дочерних окон присваивается значение 1.
4. Объект, соответствующий удаляемому окну, удаляется из дерева HTML-документа вместе со всеми своими потомками.

9.6.5. Перемещение окна

За перемещение дочернего окна отвечает следующая функция:

```
function move_wnd(obj,nX,nY) {
 if (obj.status == 'fixed') return false;
 obj.parentNode.style.pixelLeft += (nX-obj.currPosX);
 obj.parentNode.style.pixelTop += (nY-obj.currPosY);
 if (obj.parentNode.style.pixelTop < 0)
    obj.parentNode.style.pixelTop=0;
 if (obj.parentNode.style.pixelLeft < 0)
    obj.parentNode.style.pixelLeft = 0;
 obj.currPosX=nX;
 obj.currPosY=nY }
```

Операция перемещения окна осуществляется следующим образом. Когда пользователь нажимает левую кнопку мыши над областью заголовка, объект, соответствующий этой области, генерирует событие onmousedown. Обработчик этого события после активизирования перемещаемого окна устанавливает значение свойства status объекта равным moved, что говорит

о том, что объект является перемещаемым. Далее при обработке этого события в значениях свойств currPosx и currPosY запоминаются координаты перемещаемого окна такими, какими они были до начала перемещения. При этом значения данных координат мы получаем из свойств объекта event, который создается заново каждый раз, когда генерируется любое событие, в том числе и onmousedown.

После того, как пользователь нажал над заголовком кнопку мыши, он для перемещения окна передвигает указатель мыши в то место, где должно будет находиться перемещаемое окно. В это время генерируется событие onmousemove, обработчиком которого является приведенная выше функция перемещения окна. И наконец, после того, как окно было перемещено в желаемое место, пользователь отпускает кнопку мыши, в результате чего генерируется событие onmouseup. Обработка этого события заключается только в присвоении свойству status объекта, соответствующего заголовку перемещенного окна, значения fixed, что говорит о том, что окно не перемещается.

Для корректного перемещения окна нам понадобилось обработать событие onmousemove, генерируемое объектом, соответствующим рабочей области главного окна. Вообще говоря, событие от мыши генерируется для всех объектов, в область которых попадает точка, где находился курсор мыши во время возникновения события. При этом событие генерируется сначала для объекта, являющегося предком всех остальных, для которых это событие генерируется тем позже, чем длиннее цепочка их предков.

Иногда при быстром перемещении окна курсор мыши может "обогнать" это окно, т. е. выйти за пределы его заголовка. Это происходит потому, что обрабатываются далеко не все события от мыши, т. к. их слишком много, а только некоторая их часть. В такой ситуации для объекта, соответствующего заголовку, событие onmousemove генерироваться перестанет и окно прекратит перемещаться, хотя пользователь еще не отпустил кнопку мыши. Чтобы этого не случилось, данное событие обрабатывается объектом, соответствующим рабочей области главного окна, представленного объектом main_wnd.

Обработчик этого события проверяет, содержатся ли в главном окне дочерние окна, и если да, то вызывает для текущего (активного) окна функцию перемещения. Рассмотрим принцип работы этой функции.

В первой строке тела функции проверяется, является ли окно перемещаемым, и если нет, то выполнение функции завершается. Если же окно является перемещаемым, то выполняется расчет его новых координат. Всего функцией move_wnd() обрабатываются четыре переменных (помимо obj.status и собственно obj):

- obj.currPosx и obj.currPosY — координаты окна до перемещения;
- nx и nY — координаты того места, в которое должно быть перемещено окно.

При этом заголовок перемещаемого окна представлен объектом obj, а значения координат nx и nY мы получаем из значений свойств clientx и clientY объекта event. Значения новых координат окна вычисляются с помощью прибавления к старым координатам разницы между вторыми и первыми координатами вышеприведенного списка. Отметим, что здесь вычисляются новые координаты для всего дочернего окна, а не только для его заголовка. И поскольку элемент div, соответствующий заголовку, является непосредственным потомком элемента, соответствующего всему дочернему окну, новые значения координат присваиваются объекту obj.parentNode.

Примечание

Такой способ вычисления координат опирается на тот факт, что координаты event.clientx и event.clientY являются глобальными, т. е. отсчитываются от левого верхнего угла рабочей области окна браузера, в то время как style.pixelLeft и style.pixelTop — относительно левого верхнего угла области, генерируемой родительским по отношению к обладающему этими свойствами элементом. Использование глобальных координат обусловлено тем, что они не зависят от положения объекта, сгенерировавшего событие, в иерархии объектов документа. В то время как его координаты в системах координат областей, соответствующих каким-либо его предкам (локальные), будут отличаться друг от друга, и их расчет системе координат родительского окна может оказаться весьма сложным, в особенности при усложнении условий задачи, выполняемой рассматриваемым Web-приложением.

После того как новые координаты окна вычислены, следует проверка, не было ли окно перемещено выше верхней границы родительского окна, и если это так, то верхняя граница перемещаемого окна выравнивается по верхней границе родительского окна. Далее точно так же обеспечивается невозможность пересечения перемещаемым окном левой границы родительскогоокна.

После того как перемещение окна выполнено, его новые координаты запоминаются, как текущие координаты, которые будут использоваться при следующем перемещении окна как координаты до перемещения. Заметим, что вызов функции move_wnd() происходит множество раз за тот интервал времени, который ограничен слева временем, когда было сгенерировано событие onmousedown, а справа — когда произошло onmouseup. Иными словами, вышеописанный процесс перемещения окна будет повторяться до тех пор, Пока пользователь не отпустит левую кнопку мыши.

9.6.6. Переход к соседнему окну

Здесь под переходом к соседнему окну имеется в виду операция активизирования соседнего дочернего окна. За эту операцию отвечает функция:

```
function goToWnd(command) {
 if (main_wnd.lastChild.firstChild==null) return false;
 switch (command) {
   case 'next' :
    if (current_wnd.nextSibling != null)
       activateWnd(current_wnd.nextSibling); else
       activateWnd(main_wnd.lastChild.firstChild); break;
   default :
    if (current_wnd.previousSibling != null)
       activateWnd(current_wnd.previousSibling); else
       activateWnd(main_wnd.lastChild.lastChild) }}
```

В качестве аргумента command эта функция принимает строку, которая указывает, какое именно дочернее окно должно быть активизировано: следующее (значение next) или предыдущее (значение prev).

В первой строке тела функции проверяется, содержит ли родительское окно дочерние окна. Если нет, то переход осуществлять некуда и функция прекращает свою работу. В противном случае в зависимости от значения аргумента функции происходит выбор того окна, которое должно быть активизировано, после чего оно и становится активным.

Так как все дочерние окна представлены элементами, являющимися непосредственными потомками элемента, соответствующего родительскому окну, то в качестве следующего дочернего окна выбирается то, которому соответствует элемент, располагающийся сразу после текущего (`nextSibling`), а в качестве предыдущего то, которому соответствует элемент, располагающийся непосредственно перед текущим (`previousSibling`). При этом в качестве текущего берется окно, представленное объектом current_wnd.

При таком способе выбора следующего или предыдущего окна могут возникнуть две нестандартные ситуации. Если текущим окном является последнее (lastChild) из всех дочерних окон, а пользователь хочет осуществить переход к следующему окну, то переход происходит на первое (firstChild) из всех дочерних окон. И наоборот, если текущим окном является первое окно, то при переходе на предыдущее активизируется последнее окно.

В силу особенностей функции `activate_wnd()` после активизирования следующего или предыдущего окна то окно, которое было текущим до перехода, деактивизируется, т. е. перестает быть активным (см. листинг 9.7 и описание функции `activate_wnd()`).

9.6.7. Сохранение данных

За операцию сохранения данных, сопоставленных с дочерними окнами, на сервере отвечает функция:

```
function saveData() {
var inp_obj=saver.nextSibling.cloneNode();
inp_obj.setAttribute('name','number');
inp_obj.value=main_wnd.lastChild.childNodes.length;
saver.innerHTML='';
saver.insertAdjacentElement('afterBegin',inp_obj);
for (var i=0; i < main_wnd.lastChild.childNodes.length; i++) {
    inp_obj=saver.nextSibling.cloneNode();
    inp_obj.setAttribute('name','window'+i);
    var obj = main_wnd.lastChild.childNodes(i);
    inp_obj.value=obj.style.pixelTop+';'+obj.style.pixelLeft+';
                '+obj.style.zIndex+';'+obj.number;
    saver.insertAdjacentElement('afterBegin',inp_obj);
    inp_obj=saver.nextSibling.cloneNode();
    inp_obj.setAttribute('name','data'+i);
    inp_obj.value=obj.lastChild.firstChild.value;
    saver.insertAdjacentElement('afterBegin',inp_obj) }
saver.submit() }
```

Как отправлять на сервер данные с помощью Web-формы, мы рассказали в *главе 8.* В частности, в *разд. 8.1* было сказано о том, как такую форму спроектировать. Но если просмотреть листинг 9.3, то можно заметить, что нигде в нем такой формы не описывается. Вместо этого в правиле шаблона присутствует пустой элемент form с идентификатором saver. Контент этого элемента является пустым потому, что мы не можем знать заранее, сколько дочерних окон создаст пользователь и, следовательно, сколько полей ввода Должна содержать форма. Ведь количество этих полей должно быть прямо пропорционально количеству окон, т. к. для того, чтобы передать данные, сопоставленные с одним дочерним окном, Web-форма должна содержать набор полей ввода, с помощью каждого из которых на сервер отправлялась бы какая-либо одна характеристика этого окна. И всего таких наборов Должно быть столько, сколько дочерних окон содержится в главном окне.

Для разрешения сложившейся ситуации мы можем предложить два выхода. Первый из них заключается в том, чтобы описать некоторое фиксированное количество полей ввода в контенте элемента form, а отправляемые данные каким-либо образом отформатировать и присвоить в качестве значений для этих полей ввода. Но этот путь является слишком сложным, т. к. нам пришлось бы изобретать не очень простой формат для отправляемых данных, а также сочинять код JavaScript, большую часть которого занимали бы функции обработки строк (склейка строк, вырезка подстроки, поиск подстроки в строке и т. д.).

Поэтому мы воспользуемся вторым способом, который заключается в динамическом создании контента элемента form. Это означает, что при каждом выполнении операции сохранения мы будем заново создавать все те элементы ввода для формы, которые нам понадобятся для того, чтобы отправить на сервер все данные, сопоставленные с дочерними окнами. Но при этом мы также частично воспользуемся идеей, заложенной в первом способе формирования данных, подлежащих отправке.

Сразу после элемента form в листинге 9.3 следует элемент input. Мы воспользуемся этим элементом как шаблоном для создания всех необходимых элементов ввода в контенте элемента form. Создание этих элементов во многом напоминает создание дочерних окон на основе окна-шаблона, представление которого обеспечивается правилом шаблона с именем `template_wnd`. Заметим, что т. к. пользователь не должен непосредственно вводить данные в поля ввода формы, мы выбрали поле типа hidden.

В первой строке тела функции происходит клонирование шаблонного элемента input с целью создания поля ввода, с помощью которого на сервер будет отправлено общее количество дочерних окон. При этом клонировании аргумент функции `cloneNode()` отсутствует, т. к. мы не собираемся создавать копию узла, содержащую всех его потомков потому, что у шаблонного элемента input эти потомки отсутствуют. Напомним, что значением по умолчанию аргумента данной функции является `false`.

После того как копия элемента ввода создана, нужно определить ее свойства name И value. Здесь МЫ воспользовались функцией `setAttribute()`, С ПОМОЩЬЮ которой установили значение свойства name так, как если бы у копируемого элемента input этот атрибут отсутствовал вовсе. А свойству value мы присвоили значение непосредственно, т. е. с помощью оператора присваивания (=). Этим значением является длина коллекции, содержащей всех непосредственных потомков того элемента, который соответствует рабочей области родительского окна. Напомним, что объектом, соответствующим этой области, является `main_wnd.lastChild` (см. описание листинга 9.3).

После того как первое поле ввода создано, мы очищаем контент элемента form, т. к. он может быть не пустым вследствие предыдущих операций сохранения. Далее мы вставляем объект, соответствующий данному полю ввода, сразу после открывающего тега элемента form. После этого можно приступать к формированию остальных элементов ввода, с помощью которых на сервер отправятся данные, сопоставленные с дочерними окнами.

Для этих целей мы организуем цикл по всем объектам, представляющим дочерние окна. И в теле этого цикла мы опять занимаемся клонированием элемента, располагающимся сразу после элемента form в листинге 9.3. Как мы уже говорили при описании обработчика формы (см. листинг 9.4) для представления данных, сопоставленных с одним дочерним

окном, мы используем две переменные. Первая содержит значения свойств style.pixelTop, style.pixelLeft, style.zIndex И number, **принадлежащих** объекту, соответствующему данному дочернему окну, а вторая — данные из рабочей области этого окна.

При описании обработчика формы эти две переменные трактовались как параметры HTTP-запроса, здесь же они представляют собой значения свойств name и value созданного в начале цикла объекта, соответствующего элементу input. Точно так же, как и при создании первого поля ввода, мы здесь применяем функцию setAttribute о для создания свойства name, и оператор присваивания для задания значения свойства value.

При этом именем (значением свойства name) элемента ввода, с помощью которого на сервер отправляются характеристики окна, является window + счетчик цикла (i), а именем элемента ввода, служащего для отправки данных из рабочей области окна, — data + счетчик цикла. Значением свойства value первого элемента ввода является строка, содержащая значения свойств, соответствующих характеристикам окна, разделенные точкой с запятой, а последнего — значение текстового узла, соответствующего данным из рабочей области данного дочернего окна.

После формирования каждого из вышеописанных элементов ввода следует добавление соответствующих им объектов в контент элемента form. И по завершении работы цикла, т. е. когда все необходимые элементы ввода созданы, происходит подтверждение окончания ввода данных в форму. Для этого используется метод submit о, которым обладает объект saver, представляющий элемент form.

Для отправки на сервер характеристик дочерних окон мы воспользовались одним элементом ввода вместо четырех. Мы это сделали, во-первых, для того, чтобы уменьшить объем отправляемых на сервер данных, и, во-вторых, чтобы продемонстрировать достоинства первого из описанных выше способов их отправки. Действительно, легко видеть, что формат данных, содержащихся в свойстве value рассматриваемого элемента ввода, очень прост, так же, как и очень проста их серверная обработка (см. листинг 9.4).

9.6.8. Загрузка данных

При описании файла form.xml (см. листинг 9.2) мы продемонстрировали один из двух предлагаемых нами способов загрузки данных, находящихся во внешнем источнике. И если при использовании первого способа, мы загружали данные на стороне сервера, т. е. динамически генерировали некоторую часть XML-документа, то теперь покажем, как это сделать со стороны клиента, т. е. займемся генерированием данных для HTML-документа.

За загрузку данных с сервера отвечает функция:

```
function loadData() {
 var obj = document.createElement('div');
 var str =
    UIDispatcher.getHtml('http://www.region.org/main/example/windows.xml',
   ,'');
 varbIndex=str.indexOf('<div>')+5;
 vareIndex=str.lastIndexOf('</div>');
 str = str.substring(bIndex,eIndex);
 obj.innerHTML=str;
 counter=0;
 current_wnd=null;
 main_wnd.lastChild.innerHTML='';
 var objl;
 for (var i=0; i < obj.childNodes.length; i++) {
     obj1 = _window0.cloneNode(true);
     obj1.style.visibility = 'visible';
     obj1.number = obj.childNodes(i).childNodes(3).innerText;
     obj1.id = '_window'+obj1.number;
     obj1.firstChild.firstChild.nodeValue+=obj1.number;
     obj1.style.pixelTop = obj.childNodes(i).childNodes(0).innerText;
     obj1.style.pixelLeft = obj.childNodes(i).childNodes(1).innerText;
     obj1.style.zIndex = obj.childNodes(i).childNodes(2).innerText;
     if (obj1.style.zIndex > counter) {
        counter=obj1.style.zIndex;
        current_wnd=obj1}
     obj1.lastChild.firstChild.value =
          obj.childNodes(i).childNodes(4).innerText;
     main_wnd.lastChild.insertAdjacentElement('beforeEnd',obj1);}
  if (current_wnd != null)
     current_wnd.firstChild.style.backgroundColor='#506880';
  counter++ }
```

В первой строке тела этой функции мы создаем объект, который является JavaScript-представлением элемента div. Этот элемент в дальнейшем послужит нам в качестве контейнера для загруженной с сервера информации.

Как видно из кода функции, мы для загрузки серверных данных применили сетевой апплет, описанный нами в *разд. 8.5.* Методом, которым мы при этом воспользовались, является `getHtml()`, а объектом, которому он принадлежит, — `UIDispatcher`, класс которого описан в листинге 8.9. Для того чтобы работать с этим апплетом, мы подключили его в контенте элемента head, находящегося в листинге 9.3.

Вышеуказанный метод действует следующим образом. Принимая в качестве своих аргументов Internet-адрес загружаемого ресурса, а также строку запроса (query string), он открывает поток для чтения этого ресурса, считывает все данные из этого потока и затем возвращает их в виде значения строкового типа. При обращении к XML-документу (в нашем случае windows.xml) происходит его трансформация в HTML-документ, который и является результатом загрузки. Действительно, ведь для сервера не имеет значения, откуда пришел запрос к XML-документу: от браузера или от сетевого апплета.

рассмотрев листинги 9.5 и 9.6, можно сделать вывод, что загружаемой информацией будет HTML-код, в котором каждая характеристика окна, а также данные из его рабочей области будут содержаться в контенте отдельного элемента div. Кроме того, все эти элементы также будут содержаться в контенте элементов div, каждый из которых соответствует всей информации, сопоставленной с окном в целом. И наконец, вся загружаемая информация также будет содержаться в контенте элемента div, который является элементом загружаемого HTML-документа.

После того как данные с сервера загружены, мы "вырезаем" из содержащей их строки все то, что находится между первым встретившимся открывающим тегом и последним встретившимся закрывающим тегом . Поскольку корневым элементом загружаемого документа является div, то в результате мы получим набор узлов-братьев, каждый из которых содержит информацию об одном дочернем окне. Необходимость "вырезать" эти данные возникает потому, что загружаемый HTML-документ в зависимости от настроек серверного программного обеспечения может содержать бесполезную для нас информацию — объявление HTML-документа, элементы meta, уведомление о том, что страница была сгенерирована Cocoon 1.8.2 за определенное количество времени и т. п.

Для файла windows.xml с таким содержимым, которое представлено в листинге 9.5 после загрузки и вышеописанной операции "вырезания", мы получим следующий HTML-код:

```
<div>
 <div>5</div>
 <div>5</div>
 <div>1</div>
 <div>1</div>
 <div>user data</div>
</div>
<div>
 <div>34</div>
 <div>157</div>
 <div>8</div>
```

```
<div>2</div>
<divx/div>
</div>
<div>
<div>53</div>
<div>131</div>
<div>10</div>
<div>4</div>
<div>user data</div>
</div>
<div>
<div>145</div>
<div>393</div>
<div>9</div>
<div>6</div>
<div></div>
</div>
<div>
<div>248</div>
<div>94</div>
<div>7</div>
<div>8</div>
<divx/div>
</div>
<div>
<div>87</div>
<div>20</div>
<div>6</div>
<div>10</div>
<divx/div>
</div>
```

Далее мы записываем этот HTML-код в свойство `innerHTML` созданного нами ранее объекта, соответствующего элементу div. В результате мы получили элемент div, который в качестве своих непосредственных потомков содержит элементы с таким же именем, каждый из которых содержит информацию, сопоставленную с одним дочерним окном, но не само HTML-представление этого окна.

Теперь, когда в нашем распоряжении имеется вся информация о дочерних окнах, можно приступить к их созданию и встраиванию в родительское окно. Перед этим мы устанавливаем значение счетчика окон равным 1 (на-

помним, что переменная counter содержит величину, равную количеству окон плюс единица), очищаем рабочую область родительского окна и указываем, что текущее окно отсутствует, т. к. мы пока не создали ни одного дочернего окна.

Далее следует цикл по всем непосредственным потомкам элемента, представленного объектом obj и содержащего вышеприведенный HTML-код. В теле этого цикла находится код, который во многом напоминает тело функции `create_wnd()`. Это естественно, т. к. и этот код, и функция `create_wnd()` предназначены для создания дочернего окна. Задачи этого кода сводятся к созданию копии шаблонного окна, определению для данного окна загруженных характеристик и его вставке в рабочую область родительского окна.

Рассмотрим особенности кода, представленного телом цикла. Здесь мы обращаемся к непосредственным потомкам элемента, представленного объектом `obj`, как членам коллекции непосредственных потомков этого элемента. Индексом при этом служит счетчик цикла. Таким образом, мы последовательно перебираем все элементы, контент которых описывает все те дочерние окна, которые должны быть созданы.

Заметим, что каждый из перебираемых элементов сам содержит непосредственные потомки, к которым также можно обращаться как к членам коллекции childNodes. Как следует из вышеприведенного HTML-кода, в каждой из таких коллекций будет ровно пять членов, каждый из которых будет содержать в качестве своего контента:

- childNodes (0) — **координату** `pixelTop` **дочернего окна;**
- childNodes (1) — координату `pixelLeft`;
- childNodes (2) — *z*-координату дочернего окна (свойство `zIndex`);
- childNodes (3) — номер (СВОЙСТВО number);
- childNodes (4) — данные для рабочей области дочернего окна.

С помощью этой информации мы выставляем для клона значения соответствующих свойств.

Кроме того, мы находим элемент, область которого располагается поверх областей всех остальных элементов, т. е. значение свойства zindex для этого элемента максимально. Во время этого поиска текущим окном (которое, напомним, всегда совпадает с активным) становится как раз то, которое располагается поверх остальных окон. Здесь мы также присваиваем переменной counter значение максимального `zIndex`. Это нужно сделать для того, чтобы такие операции пользователя (связанные с изменением значения переменной counter, а также со значением zindex активного окна), как создание, удаление и активизирование окна после загрузки выполнялись корректно.

После того как объект, соответствующий дочернему окну, сформирован, мы вставляем это окно в родительское. Далее делаем цвет фона заголовка окна синим и увеличиваем счетчик дочерних окон на единицу с тем, чтобы его значением было общее количество всех окон плюс один.

При описании первого способа загрузки, мы не упомянули несколько важных вещей. При использовании этого способа данные загружаются в XML-документ с помощью элементов, принадлежащих пространству имен fp. Но просто загрузить данные недостаточно, нужно еще выставить корректное значение счетчика дочерних окон, сделать окно с максимальной z-координатой активным и т. д. Именно для этих целей и служит сценарий JavaScript, представленный в правиле шаблона для элемента дерева XML-документа с именем `main_wnd` (см. листинг 9.3).

В этом сценарии мы находим дочернее окно, которому соответствует максимальное значение `zIndex`, запоминаем это значение и делаем это окно текущим. Здесь мы также последовательно перебираем дочерние окна в теле цикла по всем непосредственным потомкам элемента, соответствующего рабочей области родительского окна. Далее, как уже было сказано, определяем значение счетчика дочерних окон и делаем цвет фона заголовка активного окна синим.

В конце рассматриваемого сценария JavaScript мы с помощью метода `removeNode()` удаляем последнего непосредственного потомка того объекта, который соответствует рабочей области родительского окна. Заметим, что удаляемый узел не является представлением дочернего окна, он соответствует элементу script, содержащему рассматриваемый сценарий JavaScript. Мы удаляем этот узел для того, чтобы в дальнейшем избежать путаницы между ним и объектами, соответствующими дочерним окнам. Если этот узел не удалить, то может возникнуть ошибка, например, при попытке пользователя осуществить операцию перехода к следующему дочернему окну, в то время как текущим является последнее окно, а следующим сразу после соответствующего этому окну элементом является элемент script.

Сравнивая оба способа загрузки данных с сервера, отметим, что достоинством первого является простота его реализации. Достаточно с помощью элементов из пространства имен fp прочитать эти данные из файла windows.xml, а потом применить к сгенерированным элементам соответствующие правила шаблона. Для второго же способа необходимо описывать достаточно сложную функцию. Но с другой стороны, при использовании второго способа с сервера загружается гораздо меньше данных. Этими данными являются лишь данные, сопоставленные с дочерними окнами, в то время как при первом способе загружаются HTML-представления окон, что при большой величине файла windows.xml может сильно замедлить загрузку.

Таким образом, можно сказать только, что выбор того или иного способа загрузки целиком зависит от условий поставленной задачи. И мы не станем

настаивать на одном или другом из них, а заметим только, что в примерах, находящихся на CD-ROM, используются оба эти способа, взаимно дополняя друг друга как средство формирования документа на стороне сервера, с одной стороны, и как аналогичное средство на стороне клиента, с другой.

9.7. Заключение

В этой главе мы рассмотрели разработку последнего слоя Web-приложения — слоя представления данных. Напомним, что пример, демонстрирующий преимущества слоя структуры XML-документа, мы предоставили и описали в *главе 4*.

Мы надеемся, что разработка каждого слоя Web-приложения была раскрыта настолько подробно, чтобы самостоятельно решать не очень сложные задачи с помощью описанных технологий. Данная книга не ограничивается только описаниями, т. е. теорией, ее практическая сторона представлена двумя комплексными примерами *(см. главы 4* и *9), а* также шестью демонстрационными Web-приложениями на CD-ROM. Поэтому разработка собственных Web-приложений может быть основана на кодах этих программ, а текст книги можно рассматривать как развернутый комментарий к ним.

Мы надеемся, что читатель по достоинству оценит предлагаемый ему этой книгой инструментарий, а также приемы его использования, и хотим пожелать ему успехов в применении и развитии Web-программирования.

Часть III

Приложения

Приложение 1. Некоторые свойства объекта *Style* (о поддержке **CSS-браузером** MSIE 5.5)

Приложение 2. Описание демонстрационных Web-приложений

Приложение 3. Установка программного обеспечения и файлов примеров

Приложение 4. Описание компакт-диска

ПРИЛОЖЕНИЕ 1

Некоторые свойства объекта *style* (о поддержке CSS-браузером MSIE 5.5)

При разработке дизайна Web-приложения на основе технологии CSS зачастую возникает проблема с совместимостью различных версий спецификаций CSS, а также их реализациями. Прочитав в документации о существовании какого-либо свойства CSS, не стоит спешить тут же его использовать, т. к. это свойство может не поддерживаться пользовательским агентом, а если и поддерживается, то совсем не обязательно так, как это описано в соответствующей спецификации.

Исходя из того, что выбранным нами пользоьательским агентом является браузер Microsoft Internet Explorer версии 5.5, и согласно практической направленности книги, мы приводим далее те свойства CSS, которые реализованы именно в этом браузере и именно этой версии. Теоретическое описание технологии CSS (то есть ее спецификации версий 1 и 2) можно найти в Internet по адресу **http://www.w3.org.** В табл. П1.1 перечислены свойства CSS, соответствующие им свойства объекта style, а также краткие описания каждого из них. Отметим, что описанная ниже реализация CSS отличается от соответствующей спецификации (любой версии) тем, что как в спецификации, так и в нижеследующем описании присутствуют свойства, определенные только в одном из этих документов.

Таблица П1.1. *Некоторые свойства CSS и соответствующие им свойства объекта* `style`

Свойство CSS	Свойство style	Описание
background	background	**Устанавливает или возвращает до пяти значений следующих свойств фона объекта:** background-attachment, background-color, background-image, background-position, background-repeat
background-attachment	Background-Attachment	Устанавливает или возвращает, как фоновый рисунок, соответствующий объекту, отображается документом. Может принимать два значения: scroll — рисунок прокручивается вместе с документом; fixed— фоновый рисунок неподвижен при прокрутке
background-color	backgroundColor	Устанавливает или возвращает цвет фона под объектом *(см. разд. 7.10.6)*
background-image	backgroundImage	Устанавливает или возвращает фоновый рисунок
background-position	Background-Position	Устанавливает или возвращает позицию фона объекта
background-position-x	Background-PositionX	Устанавливает или возвращает *x*-координату свойства backgroundPosition
background-position-y	background-PositionY	Устанавливает или возвращает *y*-координату свойства backgroundPosition
background-repeat	Background-Repeat	Устанавливает или возвращает, как фоновый рисунок заполняет область объекта. Может принимать четыре значения repeat (по умолчанию) — рисунок повторяется многократно по горизонтали вертикали; no-repeat— рисунок не повторяется; `repeat-x` — рисунок повторяется по горизонтали; `repeat-y` — рисунок повторяется по вертикали
behavior	behavior	Предназначено для поддержки технологии, отвечающей за поведение (behavior) объектов и поддерживаемой исключительно браузером MSIE5.5. Так как мы нигде не использовали эту технологию, здесь мы ее описывать не будем
border	border	Устанавливает или возвращает свойства границы прямоугольной области, представленной объектом *(см. разд. 7.11.2)*

Таблица П1.1 (продолжение)

Свойство CSS	Свойство style	Описание
Border-bottom	BorderBottom	Устанавливает или возвращает свойства нижней границы области (см. разд. 7.11.2)
border-bottom-color	BorderBottom-Color	Устанавливает или возвращает цвет нижней границы области (см. разд. 7.10.6 и 7.11.2)
border-bottom-style	BorderBottom-Style	Устанавливает или возвращает стиль нижней границы области (см. разд. 7.11.2)
border-bottom-width	BorderBottom-Width	Устанавливает или возвращает высоту нижней границы области (см. разд. 7.11.2)
border-collapse	borderCollapse	Устанавливает или возвращает значение, которое показывает, должны ли границы строк или столбцов таблицы быть объединены в единую границу (значение — collapse) или быть отдельными так, как это определяет стандарт HTML (значение по умолчанию — separate)
border-color	borderColor	Устанавливает или возвращает цвет границы области (см. разд. 7.10.6 и 7.11.2)
border-left	borderLeft	Устанавливает или возвращает свойства левой границы области (см. разд. 7.11.2)
border-left-color	BorderLeft-Color	Устанавливает или возвращает цвет левой границы области (см. разд. 7.10.6 и 7.11.2)
border-left-style	BorderLeft-Style	Устанавливает или возвращает стиль левой границы области (см. разд. 7.11.2)
border-left-width	BorderLeft-Width	Устанавливает или возвращает ширину левой границы области (см. разд. 7.11.2)
border-right	borderRight	Устанавливает или возвращает свойства правой границы области (см. разд. 7.11.2)
border-right-color	BorderRight-Color	Устанавливает или возвращает цвет правой границы области (см. разд. 7.10.6 и 7.11.2)
border-right-style	BorderRight-Style	Устанавливает или возвращает стиль правой границы области (см. разд. 7.11.2)
border-right-width	BorderRight-Width	Устанавливает или возвращает ширину правой границы области (см. разд. 7.11.2)
border-style	borderStyle	Устанавливает или возвращает стиль границы области (см. разд. 7.11.2)

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
border-top	borderTop	Устанавливает или возвращает свойства верхней границы области *(см. разд. 7.11.2)*
border-top-color	borderTopColor	Устанавливает или возвращает цвет верхней границы области *(см. разд. 7.10.6 и 7.11.2)*
border-top-style	borderTopStyle	Устанавливает или возвращает стиль верхней границы области *(см. разд. 7.11.2)*
border-top-width	borderTopWidth	Устанавливает или возвращает высоту верхней границы области *(см. разд. 7.11.2)*
border-width	borderWidth	Устанавливает или возвращает ширину границы области (см. *разд. 7.11.2)*
bottom	bottom	Устанавливает или возвращает *y*-координату нижней грани области *(см. разд. 7.11.3)*
clear	clear	Устанавливает или возвращает, должен ли объект быть обтекаемым плавающим объектом. Может принимать одно из трех значений: left, right и none *(см. разд. 7.11.4)*
clip	clip	Устанавливает или возвращает, какая часть абсолютно позиционируемой области должна быть видимой. Может принимать два значения: auto — значение по умолчанию, при котором координаты видимой части рассчитываются автоматически, и четверка чисел, являющихся какими-либо единицами длины *(см. разд. 7.10.2)* и представляющими собой соответственно координаты верхней, правой, нижней и левой граней видимой части области
color	color	Устанавливает или возвращает цвет текста в области *(см. разд. 7.10.6)*
Нет	cssText	Устанавливает или возвращает текстовое представление стиля объекта
cursor	cursor	Устанавливает или возвращает внешний вид курсора, отображающего указатель мыши
direction	direction	Устанавливает или возвращает направление текста в области. Может принимать два значения: ltr — значение по умолчанию, при котором текст располагается слева направо; rtl — направление текста справа налево

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
Display	display	Устанавливает или возвращает, должен ли объект иметь представление или нет *(см. разд. 7.11.1)*. Может принимать одно из семи значений: block, none, inline, inline-block, list-item, table-header-group, table-footer-group
font	font	Устанавливает или возвращает характеристики шрифта
font-family	fontFamily	Устанавливает или возвращает имя семейства шрифта области
font-size	fontSize	Устанавливает или возвращает размер шрифта области
font-style	fontStyle	Устанавливает или возвращает стиль шрифта области
font-variant	fontVariant	Устанавливает или возвращает, являются ли символы текста малыми прописными буквами. Может принимать одно из двух значений: normal — не являются (значение по умолчанию), small-caps — являются
font-weight	fontWeight	Устанавливает или возвращает "вес" шрифта. Может принимать одно из пяти значений: normal — значение по умолчанию, шрифт обычный; bold — шрифт полужирный; bolder — полужирный шрифт с большей насыщенностью; lighter — шрифт светлее ("легче"), чем при normal. Пятым значением является какое-либо из чисел: 100, 200, 300, 400, 500, 600, 700, 800 и 900, указывающих вес шрифта (т. е. чем больше число, тем насыщеннее шрифт)
height	height	Устанавливает или возвращает у-координату верхней грани области *(см. разд. 7.11.3)*
ime-mode	imeMode	Устанавливает или возвращает состояние редактора метода ввода. Предназначено для ввода и редактирования китайского, японского и корейского текстов

Таблица П1.1 (продолжение)

Свойство CSS	Свойство style	Описание
layout-grid	layoutGrid	Используется, как правило, для обработки текстов на иероглифическом языке. Устанавливает или возвращает до четырех значений следующих свойств: layoutGridChar, layoutGridLine, layoutGridMode, layoutGridType
layout-flow	layoutFlow	Устанавливает или возвращает направление расположения контента области. Может принимать одно из двух значений: horizontal — значение по умолчанию, при котором контент располагается слева направо, и следующая горизонтальная линия текста располагается под предыдущей; vertical-ideographic — контент располагается сверху вниз, и следующая вертикальная линия располагается слева от предыдущей
layout-grid-char	layoutGridChar	Устанавливает или возвращает ширину ячейки сетки, в ячейках которой находятся иероглифы китайского, корейского или японского текстов
layout-grid-line	layoutGridLine	Устанавливает или возвращает высоту ячейки сетки, в ячейках которой находятся иероглифы китайского, корейского или японского текстов
layout-grid-mode	layoutGridMode	Устанавливает или возвращает режим форматирования китайского, японского или корейского текстов. Может принимать одно из четырех значений: char — область текста разбивается на вертикальные линии, в которых располагается текст; line— область текста разбивается на горизонтальные линии; both (значение по умолчанию) — область текста представляет собой сетку; none — форматирование отсутствует
layout-grid-type	layoutGridType	Устанавливает или возвращает тип сетки, в которой располагаются иероглифы китайского, корейского или японского текстов. Может принимать одно из трех значений: loose, strict И fixed
left	left	Устанавливает или возвращает x-координату левой грани области *(см. разд. 7.11.3)*

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
letter-spacing	letterSpacing	Устанавливает или возвращает размер дополнительного пространства между буквами текста области
line-break	lineBreak	Устанавливает или возвращает правило переноса строки для японского текста
line-height	lineHeight	Устанавливает или возвращает расстояние между строками текста области (между их нижними или верхними гранями)
list-style	listStyle	Устанавливает или возвращает до трех значений следующих свойств списка: listStyleImage, listStylePosition, listStyleType
list-style-image	listStyleImage	Устанавливает или возвращает рисунок, представляющий собой маркер элемента списка. Может принимать одно из двух значений: none и URL рисунка
list-style-position	ListStyle-Position	Устанавливает или возвращает, как именно располагается маркер относительно элемента списка. Может принимать одно из двух значений: outside— значение по умолчанию, при котором маркер располагается снаружи текста элемента списка; inline— маркер располагается внутри списка
list-style-type	listStyleType	Устанавливает или возвращает внешний вид маркера списка. Может принимать одно из девяти значений: disk (по умолчанию, диск), circle (круг), square (квадрат), decimal (десятичные цифры), lower-roman (малые римские цифры), upper-roman (большие римские цифры), lower-alpha (малые буквы английского алфавита), upper-alpha (большие буквы английского алфавита), none (маркер отсутствует)
margin	margin	Устанавливает или возвращает размеры внешнего поля области, оперируя каким-либо из четырех свойств: marginBottom, marginLeft, marginRight, marginTop *(см. разд. 7.11.2)*
margin-bottom	marginBottom	Устанавливает или возвращает высоту нижнего внешнего поля *(см. разд. 7.11.2)*

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
margin-left	marginLeft	Устанавливает или возвращает ширину левого внешнего поля *(см. разд. 7.11.2)*
margin-right	marginRight	Устанавливает или возвращает ширину правого внешнего поля *(см. разд. 7.11.2)*
margin-top	marginTop	**Устанавливает или возвращает высоту** верхнего внешнего поля *(см. разд. 7.11.2)*
overflow	overflow	Устанавливает или возвращает величину, показывающую, что происходит с областью, если высота или ширина ее контента больше, чем высота или ширина самой области. Оперирует двумя свойствами: overflowX, overflowY
overflow-x	overflowX	Управляет шириной области в зависимости от ширины контента этой области. Может принимать одно из четырех значений: visible (значение по умолчанию) — ширина области увеличивается так, чтобы был виден весь ее контент; scroll — ширина области не зависит от ширины ее контента, и у области всегда присутствует горизонтальная полоса прокрутки; hidden — ширина области не зависит от ширины ее контента, и у области всегда отсутствует горизонтальная полоса прокрутки; auto — ширина области не зависит от ширины ее контента, и у области появляется горизонтальная полоса прокрутки, если ширина ее контента больше ширины области, и исчезает в противном случае
overflow-y	overflowY	Управляет высотой области в зависимости от высоты контента этой области. Может принимать одно из четырех значений: visible (значение по умолчанию) — высота области увеличивается так, чтобы был виден весь ее контент; scroll — высота области не зависит от высоты ее контента, и у области всегда присутствует вертикальная полоса прокрутки; hidden — высота области не зависит от высоты ее контента, и у области всегда отсутствует вертикальная полоса прокрутки; auto — высота области не зависит от высоты ее контента, и у области появляется вертикальная полоса прокрутки, если высота ее контента больше высоты области, и исчезает в противном случае

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
padding	padding	Устанавливает или возвращает размеры внутреннего поля области, оперируя каким-либо из четырех свойств: paddingBottom, paddingLeft, padding Right, paddingTop *(см. разд. 7.11.2)*
padding-bottom	paddingBottom	Устанавливает или возвращает высоту нижнего внутреннего поля *(см. разд. 7.11.2)*
padding-left	paddingLeft	Устанавливает или возвращает ширину левого внутреннего поля *(см. разд. 7.11.2)*
padding-right	paddingRight	Устанавливает или возвращает ширину правого внутреннего поля *(см. разд. 7.11.2)*
padding-top	paddingTop	Устанавливает или возвращает высоту верхнего внутреннего поля *(см. разд. 7.11.2)*
page-break-after	pageBreakAfter	Устанавливает или возвращает величину, указывающую, должен ли быть вставлен разрыв страницы сразу после объекта. Может принимать одно из трех значений: always — всегда вставлять после объекта разрыв страницы; auto (значение по умолчанию) — разрыв страницы вставляется исходя из ее размеров и размеров объекта; пустая строка ('') — не вставлять разрыв страницы после объекта
page-break-before	pageBreakBefore	Устанавливает или возвращает величину, указывающую, должен ли быть вставлен разрыв страницы непосредственно до объекта. Может принимать одно из трех значений: always — всегда вставлять перед объектом разрыв страницы; auto (значение по умолчанию) — разрыв страницы вставляется исходя из размеров объекта и страницы; пустая строка ('') — не вставлять разрыв страницы перед объектом
Нет	pixelBottom	Устанавливает или возвращает значение свойства bottom в пикселах, вне зависимости от типа единиц длины *(см. разд. 7.10.2),* которые были использованы для задания значения свойства bottom

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
Нет	pixelHeight	Устанавливает или возвращает значение свойства height в пикселях, вне зависимости от типа единиц длины *(см. разд. 7.10.2)*, которые были использованы для задания значения свойства height
Нет	pixelLeft	Устанавливает или возвращает значение свойства left в пикселах, вне зависимости от типа единиц длины *(см. разд. 7.10.2)*, которые были использованы для задания значения свойства left
Нет	pixelRight	Устанавливает или возвращает значение свойства right в пикселах, вне зависимости от типа единиц длины *(см. разд. 7.10.2)*, которые были использованы для задания значения свойства right
Нет	pixelTop	Устанавливает или возвращает значение свойства top в пикселах, вне зависимости от типа единиц длины *(см. разд. 7.10.2)*, которые были использованы для задания значения свойства top
Нет	pixelWidth	Устанавливает или возвращает значение свойства width в пикселах, вне зависимости от типа единиц длины *(см. разд. 7.10.2)*, которые были использованы для задания значения свойства width
Нет	posBottom	Устанавливает или возвращает значение свойства bottom в тех единицах длины *(см. разд. 7.10.2)*, в которых было задано значение свойства bottom
Нет	posHeight	Устанавливает или возвращает значение свойства height в тех единицах длины *(см. разд. 7.10.2)*, в которых было задано значение свойства height
position	position	Устанавливает или возвращает значение, определяющее схему позиционирования областей *(см. разд. 7.11.3)*. Может принимать одно из трех значений: static, absolute или relative
Нет	posLeft	Устанавливает или возвращает значение свойства left в тех единицах длины *(см. разд. 7.10.2)*, в которых было задано значение свойства left

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
Нет	posRight	Устанавливает или возвращает значение свойства right в тех единицах длины *(см. разд. 7.10.2)*, в которых было задано значение свойства right
Нет	posTop	Устанавливает или возвращает значение свойства top в тех единицах длины *(см. разд. 7.10.2)*, в которых было задано значение свойства top
Нет	posWidth	Устанавливает или возвращает значение свойства width в тех единицах длины *(см. разд. 7.10.2)*, в которых было задано значение свойства width
scrollbar-3dlight-color	scrollbar3d-LightColor	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* внешних верхней и левой граней кнопок прокрутки и бегунка полосы прокрутки
scrollbar-arrow-color	ScrollbarArrow-Color	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* стрелок на кнопках полосы прокрутки
scrollbar-base-color	ScrollbarBase-Color	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* базовых элементов (вместе с их гранями) полосы прокрутки: кнопок, бегунка и рабочей области, в которой перемещается бегунок
scrollbar-darkshadow-color	ScrollbarDark-ShadowColor	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* внешних нижней и правой граней кнопок прокрутки и бегунка полосы прокрутки
scrollbar-face-color	ScrollbarFace-Color	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* кнопок и бегунка (без их внутренних и внешних граней) полосы прокрутки
scrollbar-highlight-color	ScrollbarHigh-lightColor	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* "освещенной" части кнопок и бегунка (т. е. их внутренних левой и верхней граней), а также "освещенной" составляющей рабочей области полосы прокрутки
Scrollbar-shadow-color	ScrollbarShadow-Color	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* "затененной" части кнопок и бегунка (т. е. их внутренних правой и нижней граней)

Таблица П1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
scrollbar-track-color	ScrollbarTrackColor	Устанавливает или возвращает цвет *(см. разд. 7.10.6)* рабочей области полосы прокрутки, т. е. той области, в которой перемещается бегунок
float	styleFloat	Устанавливает или возвращает значение, указывающее сторону родительской области, с которой соприкасается плавающая область, представленная объектом *(см. разд. 7.11.4)*. Может принимать одно из трех значений: left, right и none
table-layout	tableLayout	Устанавливает или возвращает способ вычисления ширины ячеек таблицы. Может принимать одно из двух значений: auto (значение по умолчанию) — ширина ячейки равна максимальной длине контента ячейки, находящейся в этой же колонке; fixed— ширина ячейки равна значению свойства width соответствующего объекта col (определен в HTML) либо, если col.width не определено, значению ширины первой ячейки данной колонки
text-align	textAlign	Устанавливает или возвращает способ выравнивания текста в области. Может принимать одно из четырех значений: left (значение по умолчанию) — текст выровнен по левому краю; right — текст выровнен по правому краю; center — текст отцентрирован; justify— текст выровнен по левому и правому краям (растянут)
text-align-last	textAlignLast	Устанавливает или возвращает способ выравнивания последней или единственной строки текста в области. Может принимать одно из шести значений: left — текст выровнен по левому краю; right — текст выровнен по правому краю; center — текст отцентрирован; justify — текст выровнен по левому и правому краям (растянут); auto (значение по умолчанию) — последняя строка выровнена так же, как и остальной текст (так, как это задано свойством textAlign); inherit — текст выровнен так же, как в родительской по отношению к данной области

ТаблицаП1.1 *(продолжение)*

Свойство CSS	Свойство style	Описание
text-autospace	textAutospace	Устанавливает или возвращает фактор добавления дополнительного пространства между разноязычными фрагментами текста
text-decoration	TextDecoration	Устанавливает или возвращает различные способы декорирования текста. Может принимать одно из следующих пяти значений: none (значение по умолчанию) — декорирование отсутствует; underline — текст подчеркнут; line-through — текст перечеркнут; overline — линия располагается над текстом; blink — не поддерживается
Нет	TextDecorationBlink	Устанавливает или возвращает булево значение, указывающее, имеет ли свойство textDecoration значение blink. True — имеет, false — нет
Нет	TextDecorationLineThrough	Устанавливает или возвращает булево значение, указывающее, имеет ли свойство textDecoration значение line-through. True — имеет, false — нет
Нет	TextDecorationNone	Устанавливает или возвращает булево значение, указывающее, имеет ли свойство textDecoration значение none. True — имеет, false — нет
Нет	TextDecorationOverline	Устанавливает или возвращает булево значение, указывающее, имеет ли свойство textDecoration значение overline. True — имеет, false — нет
Нет	TextDecorationUnderline	Устанавливает или возвращает булево значение, указывающее, имеет ли свойство textDecoration значение underline. True — имеет, false — нет
text-indent	textIndent	Устанавливает или возвращает величину сдвига первой строки текста относительно левого края (величина красной строки)
text-justify	textJustify	Устанавливает или возвращает режим выравнивания текста по ширине (значением textAlign должно быть justify). Может принимать одно из следующих девяти значений: auto (значение по умолчанию) — браузер сам выбирает режим выравнивания;

Таблица П1.1 (продолжение)

Свойство CSS	Свойство style	Описание
		newspaper — выравнивание происходит за счет уменьшения или увеличения промежутков между буквами и словами; distribute — то же, что и newspaper, но выравнивание оптимизировано под азиатские языки; distribute-all-lines — то же, что и distribute, но последняя строка текста также выравнивается по ширине; distribute-center-last — не поддерживается; inter-word — выравнивание происходит только за счет уменьшения или увеличения промежутков между словами; inter-cluster — то же, что и inter-word, но выравнивание оптимизировано под азиатские языки; inter-ideograph — выравнивает строки иероглифического текста за счет изменения промежутков между словами и иероглифами; kashida — выравнивание происходит за счет увеличения ширины букв (предназначено для арабских языков)
text-kashida-space	textKashidaSpace	Устанавливает или возвращает степень растягивания по ширине арабских символов. Указывается в процентах. При этом значением textJustify должно быть какое-либо из следующих значений: auto, distribute, kashida, newspaper
text-transform	textTransform	Указывает или возвращает способ преобразования регистра символов текста. Может принимать одно из четырех значений: none (значение по умолчанию) — преобразование не производится; capitalize — первые буквы слов преобразуются в верхний регистр; uppercase — все буквы преобразуются в верхний регистр; lowercase — все буквы преобразуются в нижний регистр
text-underline-position	TextUnderline-Position	Устанавливает или возвращает позицию декоративной линии, если свойство textDecoration имеет какое-либо из двух значений: underline или overline. Может принимать одно из двух значений: below (значение по умолчанию) — линия располагается под текстом; above — линия располагается над текстом

Таблица П1.1 (продолжение)

Свойство CSS	Свойство style	Описание
top	top	Устанавливает или возвращает y-координату верхней грани области *(см. разд. 7.11.3)*
vertical-align	verticalAlign	Устанавливает или возвращает режим вертикального выравнивания текста. Может принимать одно из девяти значений: auto — текст выравнивается в соответствии со значением свойства `layoutFlow`; baseline (значение по умолчанию) — контент объекта выравнивается по базовой линии встроенных боксов *(см. разд. 7.11.1),* располагающихся с данным на одной строке: sub — текст становится нижним индексом; super — текст становится верхним индексом; `top` — то же, что и baseline, но контент выравнивается по верхнему краю боксов; middle — то же, что и baseline, но контент выравнивается по средней линии боксов; `bottom` — то же, что и baseline, но контент выравнивается по нижнему краю боксов; text-`top` — то же, что и baseline, но текст выравнивается по верхнему краю текста бокса; text-bottom— то же, что и baseline, но текст выравнивается по нижнему краю текста бокса
visibility	visibility	Устанавливает или возвращает статус видимости объекта. Может принимать одно из трех значений: inherit (значение по умолчанию) — статус видимости наследуется от родительского объекта; visible— объект видим; hidden — объект невидим
white-space	whiteSpace	Устанавливает или возвращает режим переноса строк, не помещающихся в область. Может принимать одно из трех значений: normal (значение по умолчанию) — строки разрываются автоматически, если их длина превышает длину содержащей их области, остаток строки переносится на следующую строку; nowrap — строки не переносятся автоматически; `pre` — не поддерживается

Таблица П1.1 *(окончание)*

Свойство CSS	Свойство style	Описание
word-break	wordBreak	Устанавливает или возвращает режим переноса слов на следующую строку. Предназначено в основном для текстов, содержащих фрагменты на китайском, японском или корейском языках. Может принимать одно из трех значений: normal (значение по умолчанию) — перенос слов разрешен; break-all — перенос слов разрешен как для азиатского, так и для неазиатского фрагментов текста; keep-all — перенос слов запрещен для азиатских фрагментов текста
word-spacing	wordSpacing	Устанавливает или возвращает величину дополнительного пространства между словами в тексте области
word-wrap	wordwrap	Устанавливает или возвращает величину, указывающую, могут ли разрываться слова при переносе строк или нет. Может принимать два значения: normal (значение по умолчанию) — слова не разрываются; break-word — разрыв слов разрешен
writing-mode	writingMode	Указывает или возвращает направление заполнения контентом области. Может принимать одно из двух значений: lr-tb (значение по умолчанию) — контент заполняет область слева направо, а горизонтальные строки выстраиваются сверху вниз; tb-rl — контент заполняет область сверху вниз, а вертикальные строки выстраиваются справа налево
z-index	zIndex	Устанавливает или возвращает номер слоя на оси *Z* позиционируемой области *(см. разд. 7.11.5)*
zoom	zoom	Устанавливает или возвращает масштаб отображения объекта. Может принимать в качестве значения процентную величину, число с плавающей точкой или значение по умолчанию normal, при котором масштаб равен либо 1.0, либо 100%

ПРИЛОЖЕНИЕ 2

Описание демонстрационных Web-приложений

На CD-ROM находятся шесть демонстрационных Web-приложений, иллюстрирующих следующие компоненты графического пользовательского интерфейса:

- элемент управления Selector;
- элемент управления Chooser;
- меню;
- дерево;
- Web-приложение, реализующее обработку документов на стороне клиента;
- Web-приложение, демонстрирующее совместную работу пяти предыдущих компонентов.

Об их установке, а также об установке программного обеспечения можно прочитать в *приложении 3.* Далее следует краткое описание каждого из вышеперечисленных Web-приложений. В этих описаниях имена файлов, образующих каждое Web-приложение, выделены полужирным шрифтом.

П2.1. Элемент управления Selector

Данный элемент управления во многом напоминает свой аналог из HTML, которому соответствует элемент select этого языка. Отличие заключается в том, что Selector не обеспечивает возможности выбора нескольких альтернатив. Этот элемент управления является не только демонстрационным, но и может послужить при разработке реального Web-приложения, т. к. компенсирует один недостаток селектора, реализованного в HTML.

Этот недостаток заключается в том, что представление элемента select является т. н. оконным элементом управления (windowed control), и вследствие этого соответствующий ему объект не поддерживает свойство `zIndex`. Это означает, что данный элемент управления всегда будет виден на Web-странице, даже в том случае, когда в соответствующей ему области располагаются другие области со сколь угодно большой *z*-координатой.

На рис. П2.1 изображен внешний вид элемента Selector.

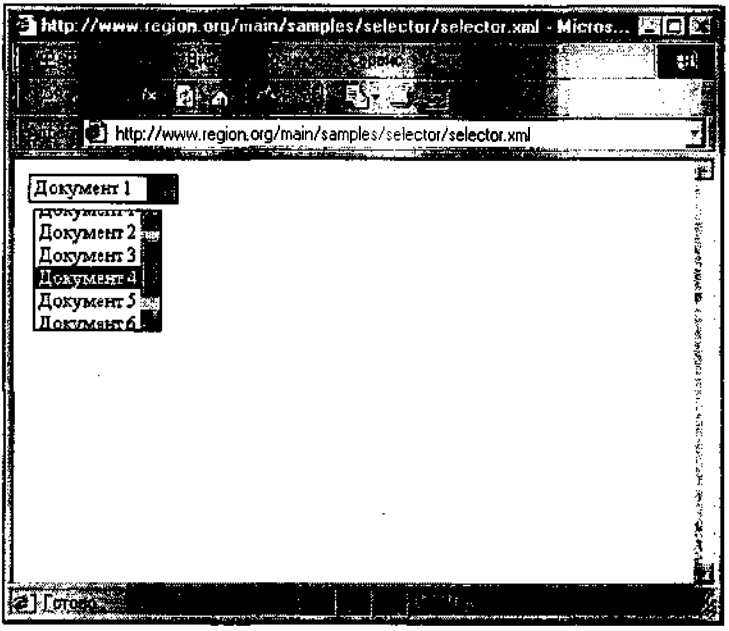


Рис. П2.1. Элемент управления Selector

Принцип работы рассматриваемого элемента управления ничем не отличается от принципа работы HTML-элемента select, значением атрибута multiple является false. Как видно из рис. П2.1, этот элемент управления отображается в виде прямоугольной области, в которой находится текст выбранной альтернативы, а также кнопка, служащая для показа всех имеющихся альтернатив. Как только пользователь нажимает на эту кнопку, перед ним появляется окно, содержащее альтернативы, каждую из которых он может выбрать с помощью мыши. Если при этом список альтернатив слишком велик, то у содержащего его окна появляется полоса прокрутки. После того, как пользователь щелкнет на какой-либо из альтернатив кнопкой мыши, область, содержащая их список, исчезает, а выбранная альтернатива появляется в той области, которая содержит кнопку.

Рассматриваемое Web-приложение состоит только из трех файлов:

- selector.xml — главный файл Web-приложения;
- selector.xsl — сценарий XSLT для трансформации главного файла;
- dist.xml — файл, в котором содержатся описания альтернатив.

файл selector.xml содержит всего три элемента: selector — элемент документа; inp_line, служащий для представления строки, в которой содержится выбранная альтернатива, и container, описывающий область, содержащую список всех имеющихся альтернатив.

В контенте элемента `inp_line` описывается файл ресурса dist.xml, из которого в контенте container происходит чтение списка альтернатив. Здесь мы описали файл ресурса в контенте одного элемента, а операцию чтения из него в другом для того, чтобы показать, что их взаиморасположение не является жестко определенным. Единственное условие, которое должно быть выполнено, заключается в том, что описание файла ресурса должно предшествовать операциям над ним.

Рассмотрим файл selector.xsl. В контенте элемента head правила шаблона для элемента документа содержится описание стилей CSS для наиболее значимых элементов HTML-документа, а также сценарий JavaScript, отвечающий за обработку действий пользователя над элементом управления. При этом описание стиля содержит следующие правила:

- table — стиль таблицы, которая используется для представления области, содержащей выбранную альтернативу и кнопку;
- `td.inp_line` — стиль области, содержащей выбранную альтернативу;
- `td.button` — стиль для кнопки;
- `div.container` — стиль области, содержащей список альтернатив;
- div.item — стиль каждого элемента из списка альтернатив.

Сценарий JavaScript содержит следующие функции:

- `coHandler()` — отвечает за появление и исчезновение области, содержащей список альтернатив, когда пользователь нажимает кнопку;
- `setSize()` — отвечает за появление и исчезновение полосы прокрутки у области, содержащей список альтернатив, в зависимости от того, умещается ли этот список в область или нет.

Помимо правила шаблона для элемента selector рассматриваемый сценарий XSLT содержит правила для элементов `inp_line`, container и document. Элементы с именем document загружаются из файла dist.xml и служат для представления каждой альтернативы из списка. Элемент div, являющийся этим представлением, генерирует события:

- onclick — в результате его обработки альтернатива становится выбранной и помещается в область, соответствующую объекту `inp_line`, являющемуся результатом трансформации элемента `inp_line` XML-документа;
- onmousemove и `onmouseout` — при обработке этих событий изменяется стиль отображения элементов, соответствующих альтернативам из списка в зависимости от того, находится ли над областью, соответствующей элементу, курсор мыши или нет.

Выбор пользователя на сервере не сохраняется, т. е. вне зависимости от того, какую альтернативу выбрал пользователь, при следующем запросе к Web-приложению выбранной будет та альтернатива, которой соответствует первый элемент с именем document в файле dist.xml.

П2.2. Элемент управления Chooser

Данный элемент управления предназначен для реализации выбора нескольких альтернатив из многих. Рассмотрим рис. П2.2.

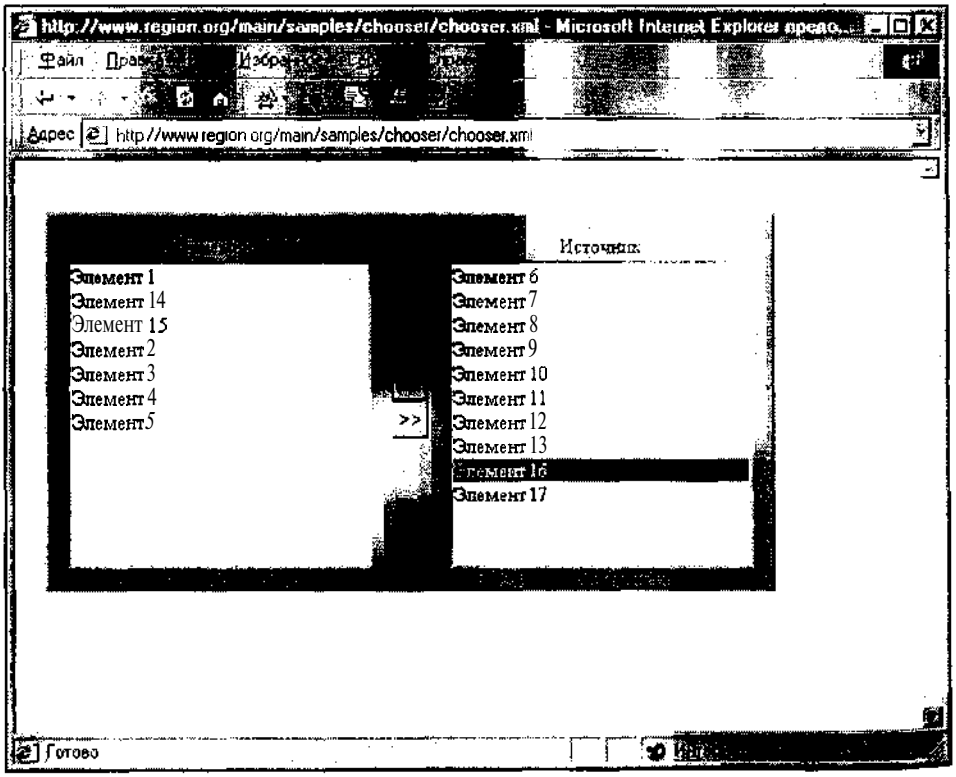


Рис. П2.2. Элемент управления Chooser

Здесь изображены две области. Правая содержит список всех имеющихся альтернатив, а левая — тех из них, которые выбрал пользователь. Выбор при этом осуществляется с помощью двух кнопок, располагающихся между этими двумя областями.

Для того чтобы выбрать какую-либо альтернативу, пользователь должен в правой области щелкнуть по ней левой кнопкой мыши, а затем нажать кнопку, на которой изображена стрелка "влево". Для того чтобы удалить альтернативу из списка выбранных, пользователь должен в левой области щелкнуть по ней левой кнопкой мыши, а затем нажать кнопку "вправо". Если помеченной является альтернатива из правой области и пользователь нажмет стрелку "вправо", то ничего не произойдет, и наоборот.

Web-приложение, реализующее элемент управления Chooser, состоит из пяти файлов:

- chooser.xml — главный файл Web-приложения;
- chooser.xsl — сценарий XSLT для трансформации главного файла;
- chooser.js — сценарий JavaScript, отвечающий за обработку действий пользователя над элементом управления;
- list1.xml — файл, содержащий выбранные альтернативы;
- list2.xml — файл, содержащий список всех имеющихся альтернатив.

Файл chooser.xml содержит следующие элементы:

- chooser — элемент документа;
- `read_all` — элемент, в контенте которого происходит чтение из файлов list1.xml и list2.xml;
- `dist_all` — элемент, в контенте которого происходит чтение всех выбранных альтернатив из файла list1.xml;
- `sourse_all` — элемент, в контенте которого происходит чтение всех имеющихся альтернатив из файла list2.xml;
- `dist` — элемент, соответствующий левой области элемента управления;
- source — элемент, соответствующий правой области элемента управления;
- controls — элемент, соответствующий кнопкам выбора и отмены выбора альтернатив.

Рассмотрим сценарий XSLT, содержащийся в файле chooser.xsl. В правиле шаблона для элемента документа содержится описание следующих стилей:

- `table.container` — содержит стиль для области, соответствующей элементу управления целиком;
- td. items — содержит стиль для левой и правой областей элемента управления.

В нем также содержится сценарий JavaScript, в котором создается объект Chooser, определенный в файле chooser.js.

Рассмотрим правила шаблона для остальных элементов:

- dist — отвечает за представление левой области. В правиле шаблона для этого элемента с помощью директивы xsl:apply-templates обрабатываются только те элементы document, которые содержатся в контенте элемента dist_all и не содержатся в контенте source_all. Это делается для того, чтобы в левой области отображались только те выбранные пользователем альтернативы, которые имеются в общем списке альтернатив.
- source — в правиле шаблона для этого элемента используется директива xsl:apply-templates, значение атрибута select которой аналогично описанному в предыдущем пункте. Оно обеспечивает отсутствие в правой области тех альтернатив, которые были выбраны пользователем.
- controls — отвечает за представление кнопок выбора и отмены выбора альтернатив.
- document — в правиле шаблона для этого элемента создается представление каждой альтернативы. В качестве этого представления выступает элемент div с атрибутом pos, значением которого является позиция элемента document в общем списке одноименных элементов, содержащихся в list2.xml. Мы не смогли воспользоваться здесь функцией position о, т. к. данные элементы при трансформации обрабатываются не в том порядке, как они располагаются в файле list2.xml, а в том, как они размещены в том наборе узлов, описанном значением атрибута select **соответствующей Директивы** xsl:apply-templates. **Поэтому** функция position() будет возвращать номер позиции узла в данном наборе узлов, а не номер позиции элемента document в общем списке.

Рассмотрим файл chooser.js, в котором содержится сценарий JavaScript, обеспечивающий взаимодействие пользователя с элементом управления. В этом сценарии определяется объект Chooser, обладающий одним свойством uitem, значением которого является текущая альтернатива, т. е. тот объект, который соответствует данной альтернативе, и над которым пользователь щелкнул левой кнопкой мыши.

Для объекта Chooser определены следующие методы:

- selectItem() — служит для пометки (активизирования) альтернативы пользователем;
- onTextClick() — обработчик события onclick, генерируемого объектом, соответствующим альтернативе;
- moveItem() — служит для перемещения альтернативы из одной области в другую (операции выбора и отмены выбора);
- getSel() — возвращает объект, соответствующий объекту, являющемуся представлением той альтернативы, которая должна стать текущей после операции выбора или отмены выбора;

- `moveItemLR()` — вспомогательный метод для перемещения объектов из одной области в другую.

Выбранные пользователем альтернативы на сервере не сохраняются, т. е. файл list1.xml остается неизменным. Файл list2.xml не должен изменяться ни при каких условиях, т. к. содержит полный список альтернатив.

П2.3. Меню

Данное Web-приложение реализует пользовательское меню, мало чем отличающееся по своей функциональности от стандартного меню, которое создается средствами операционной системы. Примером такого стандартного меню может послужить меню Internet Explorer. Рассмотрим рис. П2.3.

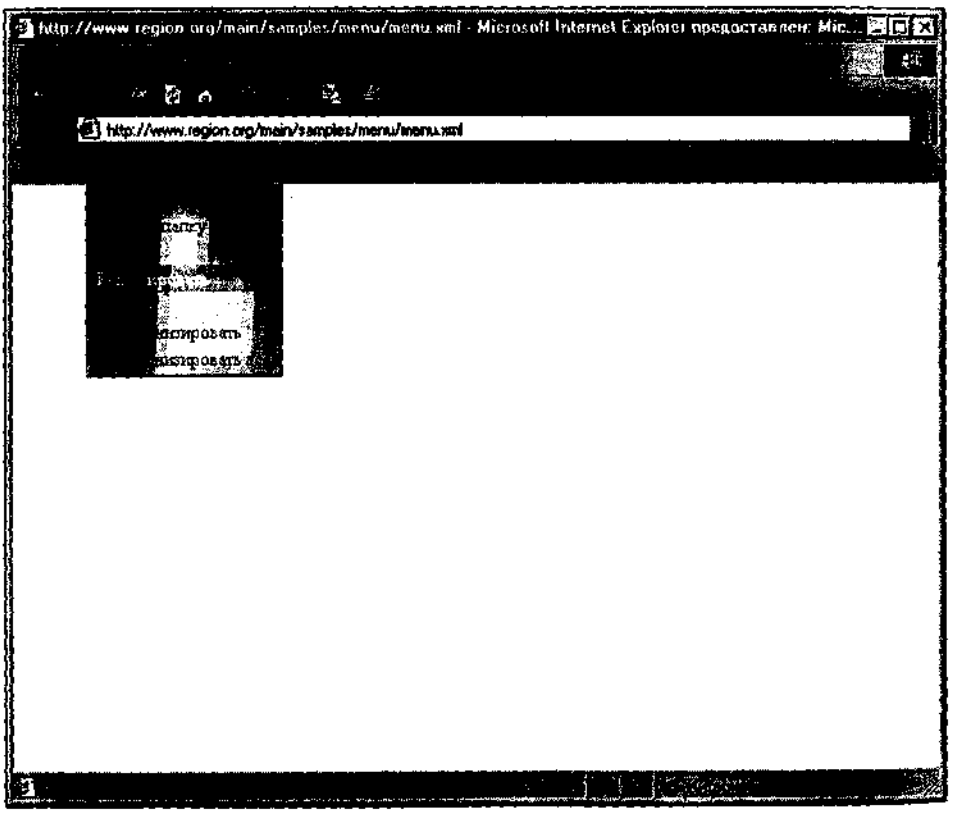


Рис. П2.3. Выпадающее меню

Для данного примера мы цвета подобрали так, чтобы меню выглядело в ви де части окна Internet Explorer. Выбранной цветовой схемой Windows пр этом является "Дождливый день".

Все составляющие данного меню можно условно разделить на четыре группы:

- полоса меню, в которой содержатся пункты меню верхнего уровня;
- пункты меню, при нажатии на которые левой кнопкой мыши появляются соответствующие им выпадающие меню;
- выпадающие меню, содержащие обычные (не верхнего уровня) пункты меню;
- пункты меню, при нажатии на которые левой кнопкой мыши выполняются команды, соответствующие этим пунктам.

В рассматриваемом Web-приложении главными событиями, для которых определены ИХ обработчики, ЯВЛЯЮТСЯ onclick, onmousemove И onmouseout. С их помощью обеспечиваются следующие функции меню:

- При нажатии пользователем левой кнопки мыши над пунктом меню верхнего уровня появляется выпадающее меню, соответствующее этому пункту. Если при этом какое-либо из выпадающих меню было видимым, то оно становится невидимым.
- При нажатии пользователем левой кнопки мыши где-либо вне меню скрывается то выпадающее меню, которое было видимым.
- При перемещении курсора мыши над пунктами меню верхнего уровня данные пункты активизируются (изменяется стиль соответствующих им элементов). Если при этом было видимым какое-либо из выпадающих меню, то оно становится скрытым, но становится видимым то, которое соответствует тому пункту меню верхнего уровня, над которым в данный момент перемещается курсор мыши.
- При нажатии пользователем левой кнопки мыши над пунктом выпадающего меню выполняется команда, соответствующая данному пункту. При этом выпадающее меню становится скрытым.

Web-приложение, реализующее пользовательское меню, состоит из следующих четырех файлов:

- menu.xml — главный файл Web-приложения;
- menu.xsl — содержит сценарий XSLT для трансформации главного файла;
- menu.css — содержит описания стилей компонентов меню;
- menu.js — содержит сценарий JavaScript, отвечающий за реагирование Web-приложения на действия пользователя.

Файл menu.xml содержит следующие элементы:

- page — элемент документа;
- menu — соответствует всему пользовательскому меню в целом;

- `menubar` — соответствует полосе меню;
- `menuext` — соответствует выпадающему меню;
- `menuitem` — соответствует либо пункту меню верхнего уровня, если находится в контенте элемента menubar, либо пункту выпадающего меню, если находится в контенте элемента menuext.

Каждый из элементов menuitem обладает атрибутом name, в котором содержится значение, интерпретируемое либо как идентификатор объекта, соответствующего данному пункту меню, либо как соответствующая этому пункту команда. Значением атрибута text, принадлежащего элементу menuitem, является текст надписи пункта меню, соответствующего этому элементу.

Рассмотрим сценарий XSLT, содержащийся в файле menu.xsl. В правиле шаблона для элемента документа, в контенте элемента head в HTML-документ включаются файл, содержащий описания стилей menu.css, а также файл, содержащий сценарий JavaScript menu.js. В файле menu.css описываются следующие стили:

- `div.menu_layer` — описывает стиль области, располагающейся под областью, содержащей полосу меню;
- `div.menubar` — описывает стиль полосы меню;
- `div.other` — описывает стиль области, не содержащей области, которой соответствует стиль `menu_layer`, а также области, соответствующей полосе меню. Область, соответствующая стилю other, занимает остальное пространство Web-страницы;
- `div.TopMenuItemOut` — описывает стиль элемента, соответствующего пункту меню верхнего уровня, который применяется к этому элементу, если над ним не находится курсор мыши;
- `div.TopMenuItemOn` — описывает стиль элемента, соответствующего пункту меню верхнего уровня, который применяется к этому элементу, если над ним находится курсор мыши;
- `div.MenuContainer` — описывает стиль выпадающего меню;
- `div.MenuItem` — описывает стиль пункта меню, находящегося в выпадающем меню.

Помимо элемента head правило шаблона для элемента документа содержит элементы div, находящиеся в контенте элемента body и соответствующие стилям `menu_layer` и other. Мы не будем здесь приводить описания правил шаблона для остальных элементов, имеющихся в файле menu.xsl, но дадим к ним краткие пояснения при описании файла menu.js. В этом файле содержатся определения следующих объектов и функций:

- MenuHandler — объект, отвечающий за появление и скрытие выпадающих меню. Данный объект создается один раз для всего меню в контенте элемента head. Ему принадлежат следующие свойства и методы:
  - expandedItem — свойство, содержащее объект, соответствующий видимому в данный момент выпадающему меню;
  - turn(TopMenuItem) — функция, показывающая и скрывающая выпадающее меню, соответствующее элементу верхнего уровня, представленному объектом TopMenuItem. Выполняется при обработке события onclick, генерируемого для элементов, соответствующих каждой компоненте (составляющей) меню.
- TopMenu(TopMenu, sMenuHandler) — функция, в которой устанавливаются некоторые характеристики пункта меню верхнего уровня, заданного объектом TopMenu, а также обработчики событий onclick, onselectstart, onmousemove И onmouseout ДЛЯ ЭТОГО объекта. Вызов функции происходит в правиле шаблона для пункта меню верхнего УРОВНЯ. Образцом ЭТОГО правила является //menubar/menuitem.
- Menu(MenuItem, father) — функция, в которой устанавливаются некоторые характеристики пункта, заданного объектом MenuItem, выпадающего меню, а также обработчики событий onclick, onselectstart, onmousemove И onmouseout ДЛЯ ЭТОГО объекта. Вызов функции ПрОИСХОдит в правиле шаблона пункта выпадающего меню. Образцом этого правила является //menext/menuitem. Значением второго аргумента является объект, соответствующий пункту меню верхнего уровня, которому соответствует выпадающее меню, содержащее пункт меню, представленный объектом Menuitem.

Web-приложение, реализующее пользовательское меню, наглядно демонстрирует преимущества разделения данных и их структуры. При необходимости как-либо изменить меню (добавить или удалить пункты, поменять их надписи) достаточно отредактировать файл menu.xml, вместо того, чтобы многократно переписывать HTML-код, соответствующий какому-либо из компонентов этого меню. От этого было бы невозможно уйти при создании меню без применения XML, т. е. только средствами HTML.

П2.4. Дерево

В результате обработки этого приложения в окне браузера появляется дерево, содержащее узлы, которые можно интерпретировать как документы и папки документов. При этом пользователь имеет возможность открывать и закрывать папки, а также с помощью мыши помечать и папки, и документы. Рассмотрим рис. П2.4.

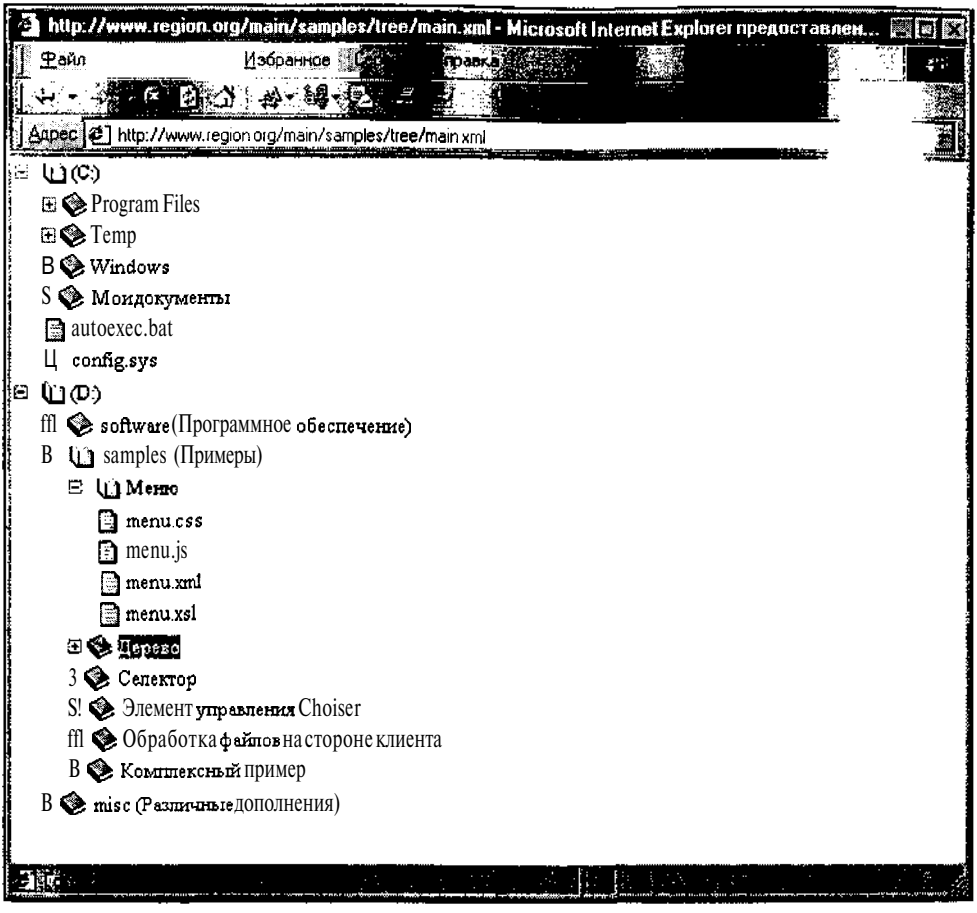


Рис. П2.4. Дерево

Как видно из рис. П2.4, узлами дерева в данном случае являются диски, папки и файлы некоторого компьютера. При этом для обозначения открытой и закрытой папки выбран рисунок, изображающий открытую и закрытую книгу соответственно. Обозначением же файла служит рисунок, изображающий лист бумаги. Диски обозначаются так же, как и папки.

Файлы, составляющие данное Web-приложение, мы поместили в несколько папок, описание которых, а также их содержимого следует далее:

- applet — папка, в которой содержится сетевой апплет, представленный файлом UIDispatcher.class и служащий для загрузки данных с сервера *(см. разд. 8.5 и 9.6.8);*
- docs — папка, в которой содержатся файлы и папки, служащие для описания узлов дерева;

- gif — папка, в которой содержатся следующие рисунки, отображающиеся в узлах дерева:
  - bo.gif — рисунок, изображающий открытую книгу;
  - bs.gif — рисунок, изображающий закрытую книгу;
  - dc.gif — рисунок, изображающий лист бумаги;
  - minus.gif — рисунок, на котором изображен знак "минус";
  - plus.gif — рисунок, на котором изображен знак "плюс";
- js — папка, в которой содержатся следующие сценарии JavaScript:
  - main.js — файл, содержащий сценарий JavaScript для обработки действий пользователя над главным документом Web-приложения;
  - tree.js — файл, содержащий сценарий JavaScript для обработки действий пользователя над узлами дерева;
- xsl — папка, содержащая следующие сценарии XSLT:
  - main.xsl — файл, содержащий сценарий XSLT для трансформации главного файла Web-приложения (main.xml);
  - tree.xsl — файл, содержащий сценарий XSLT для трансформации XML-файла, описывающего дерево (tree.xml);
- main.xml — главный файл Web-приложения;
- tree.xml — файл, описывающий дерево.

Так как информация о структуре дерева содержится в файлах, находящихся в папке docs, файл main.xml содержит только один элемент раде — элемент документа. Правило шаблона для этого элемента, находящееся в файле main.xsl, помимо подключения файлов со сценариями JavaScript и апплета в контенте элемента body содержит элемент script, в сценарии которого вызывается функция load(), которую мы опишем позднее.

Более серьезного внимания заслуживают файлы tree.xml и tree.xsl. В первом из них в контенте элемента документа с именем folders происходит чтение той части дерева, которая описана в файле items.xml. Этот файл находится в папке docs или одной из ее подпапок, и путь к нему передается с помощью параметров HTTP-запроса path и val, переданных методом GET. Прежде чем описывать файл tree.xsl, рассмотрим типичный файл items.xml.

В каждой папке, находящейся где-либо в docs, имеется папка с именем serv. И в каждой папке serv находится файл с именем items.xml. В этом файле описывается содержимое той папки, в которой размещается содержащая данный файл items.xml папка serv, причем это описание представляет собой информацию о той ветке дерева, которой соответствует данная папка. Файл

items.xml описывает содержимое только данной папки, но не ее подпапок, в каждой из которых находится свой собственный, т. е. описывающий содержимое этой подпапки, файл items.xml.

Корневым элементом документа, представленного файлом items.xml, является элемент с именем data. Помимо data этот файл может содержать следующие элементы:

- folder — описывает папку;
- document — описывает файл;
- empty — описывает узел дерева, который отображается при открытии пустой папки.

При этом контент каждого из этих элементов образуют следующие два элемента:

- `file_name` — его контентом является имя файла или папки на диске;
- `document_name` — его контентом является имя узла дерева, соответствующего данному файлу или папке.

Прежде чем перейти к описанию соответствующих этим элементам правил шаблона, рассмотрим объекты и функции JavaScript, содержащиеся в файлах main.js и tree.js. В первом из них описываются следующие три функции:

- `load(targ, dist, arg)` — принимая в качестве своих аргументов объект, соответствующий области, в которую должны быть загружены данные, URL ресурса, содержащего данные и строку запроса (query string), загружает данные с сервера. Иногда может сложиться такая ситуация, когда происходит обращение к апплету (в нашем случае к методу `getHtml()` объекта `UIDispatcher`), в то время как данный апплет еще не успел загрузиться. Для этого мы проверяем состояние загрузки апплета, используя свойство readystate, и если апплет загрузился не полностью, то через интервал времени, равный 300 миллисекундам, рекурсивно снова вызываем функцию `load()`. Так продолжается до тех пор, пока апплет не будет полностью загружен, и только тогда будет вызван метод `getHtml()`.
- `getContent(tag_name, source)` — ИЗ строки _source вырезает подстроку, содержащуюся между первым встретившимся в _source открывающим тегом с именем `tag_name` и последним встретившимся закрывающим тегом с тем же именем. Эта функция предназначена для получения из всех загруженных данных только тех, которые подлежат обработке. С ее помощью отсекается некоторая служебная информация, загружаемая вместе с данными, а также некоторые лишние элементы верхнего уровня *(см. разд. 9.6.8)*.

- norm(str) — принимая в качестве аргумента строку str, возвращает эту же строку, из начала и конца которой удалены все пробелы.

Далее следует описание объектов и функций JavaScript, содержащихся в файле tree.js:

- expand(pm, oc) — функция, которая при открытии папки заменяет рисунок, изображающий знак "минус" на рисунок, изображающий знак "плюс", а рисунок, изображающий открытую книгу, — на рисунок, изображающий закрытую книгу.
- collapse(pm, oc) — функция, которая при закрытии папки выполняет действия, обратные действиям expand().
- Tree — объект, отвечающий за выполнение пользователем операций над деревом. Обладает свойством uitem, в котором содержится объект, соответствующий текущему (помеченному) узлу дерева, а также следующие методы:
  - getPath(obj) — для файла или папки, представленной объектом obj, возвращает путь от папки, содержащей Web-приложение, т. е. путь всегда начинается с docs. Объект obj служит также для представления узла дерева, и его идентификатором является имя файла или папки. Таким образом, перебирая последовательно всех предков данного узла, формируется строка, содержащая возвращаемый путь. Объект, являющийся предком узла, которому соответствует объект obj, содержится в значении свойства parent этого объекта;
  - onImgClick(pm, oc, _self, _child) — является обработчиком события onclick, генерируемого объектами, соответствующими рисункам, отображающимся в узле, соответствующем папке. В теле этой функции С ПОМОЩЬЮ КакОЙ-ЛИбо ИЗ функций expand О ИЛИ collapse О происходит замена этих рисунков, а также открытие или закрытие папки. При открытии папки у дерева появляется новая ветка, соответствующая содержимому этой папки. Загрузка содержимого осуществляется с помощью сетевого алплета и функции getcontent();
  - selectItem(name) — делает помеченным (текущим) узел дерева, которому соответствует объект name, а также снимает пометку узла, если этот узел уже был помечен;
  - onTextClick(name) — представляет собой обработчик события, генерируемого объектом, соответствующим текстовой части узла дерева, представленного объектом name. В этом Web-приложении служит ТОЛЬКО ДЛЯ вызова фуНКЦИИ selectItem().

Рассмотрим сценарий XSLT, содержащийся в файле tree.xsl. В правиле шаблона для элемента документа проверяется, является ли та папка, данные о содер-

жимом которой были прочитаны в файле tree.xml, пустой, и если это так, то применяется правило шаблона для элемента empty. В противном случае применяются правила шаблона для элементов с именами folder и document.

Правило шаблона для элемента folder служит для представления узла дерева, который является папкой. При этом узел представлен элементом table, идентификатор которого совпадает с именем папки. Для этого элемента определяются значения следующих атрибутов (не считая определяемых языком HTML):

- parent — содержит объект, являющийся непосредственным предком данного узла;
- child — содержит объект, соответствующий элементу, содержащему непосредственных потомков данного узла;
- text — содержит объект, соответствующий элементу, контентом которого является текстовая часть узла.

Рассматриваемый элемент table содержит еще два элемента с таким же именем. Первый из них служит для корректного размещения рисунков и имени папки относительно друг друга, а второй служит для описания области, в которой будут располагаться потомки этой папки. Мы использовали элементы table для форматирования потому, что MSIE 5.5 не всегда корректно отображает элементы блочного типа. В частности, пустой элемент (то есть состоящий из одного тега — тега пустого элемента) и элемент с пустым контентом, но имеющий закрывающий тег, этим браузером будут отображаться по-разному.

Правило шаблона для элемента с именем document мало чем отличается от аналогичного правила для элемента folder. Действительно, как видно из рис. П2.4, узлы, соответствующие папке и файлу, отображаются почти одинаково. Разница заключается в том, что узел, соответствующий файлу, содержит только один рисунок и не имеет той области, в которой должны отображаться его потомки, т. к. файл не может содержать других файлов или папок.

То же самое будет справедливым и для элемента с именем empty. Соответствующее этому элементу правило шаблона является еще более простым, чем аналогичное правило для элемента document. Узел, соответствующий рассматриваемому элементу, вовсе не содержит рисунков.

Теперь, для того чтобы принцип работы Web-приложения был более понятен, мы опишем последовательность действий, которые выполняются в том случае, когда пользователь открывает папку:

1. Для того чтобы открыть папку, пользователь должен щелкнуть левой кнопкой мыши по одному из рисунков, отображающихся в соответст-

вующем ей узле. В результате этого объектом, соответствующим данному рисунку, будет сгенерировано событие onclick.

2. Затем в качестве обработчика этого события будет вызван метод onImgClick() объекта Tree. При этом первым аргументом будет объект, соответствующий рисунку, изображающему знак "плюс", вторым — объект, соответствующий рисунку, изображающему закрытую книгу, третьим — объект, соответствующий данному узлу дерева, и четвертым — объект, соответствующий области, в которой будут содержаться узлы, являющиеся потомками данного.

3. При выполнении тела метода с помощью метода expand() будут заменены рисунки, соответствующие папке, с помощью метода getPath() сгенерирован путь к данной папке, и в область, заданную четвертым аргументом, будет помещен результат загрузки данных, осуществленной С ПОМОЩЬЮ Метода getHtml() **объекта** UIDispatcher.

4. Аргументами метода getHtml() при этом будут URL ресурса http://www.region.org/main/samples/tree/tree.xml **И строка запроса,** в которой будут переданы путь к данной папке (в качестве значения параметра path) и имя данной папки (в качестве значения параметра val HTTP-запроса).

5. При обработке HTTP-запроса к файлу tree.xml, в контенте этого файла будут прочитаны данные из файла items.xml, который находится в папке serv, которая, в свою очередь, помещена в папке с именем, являющимся значением параметра val. При этом путь к последней будет содержаться в значении параметра path.

6. После того, как данные из файла items.xml загружены в контент файла tree.xml, к ним будут применены правила шаблона, содержащиеся в файле tree.xsl. HTML-код, являющийся результатом этой трансформации, будет значением, возвращаемым методом getHtml().

7. Далее из возвращенного значения с помощью функции getContent() будут вырезаны "полезные" данные, которые затем будут вставлены в область, представленную объектом, являющимся четвертым аргументом метода onImgClick().

П2.5. Web-приложение, реализующее обработку документов на стороне клиента

Обработка документов на стороне клиента, реализуемая этим Web-приложением, заключается в загрузке файла, находящегося на стороне кли-

ента, пометки некоторых слов из его контента как ключевых и последующем его сохранении. Выбор ключевого слова при этом реализован с помощью элемента управления Selector. Рассмотрим рис. П2.5.

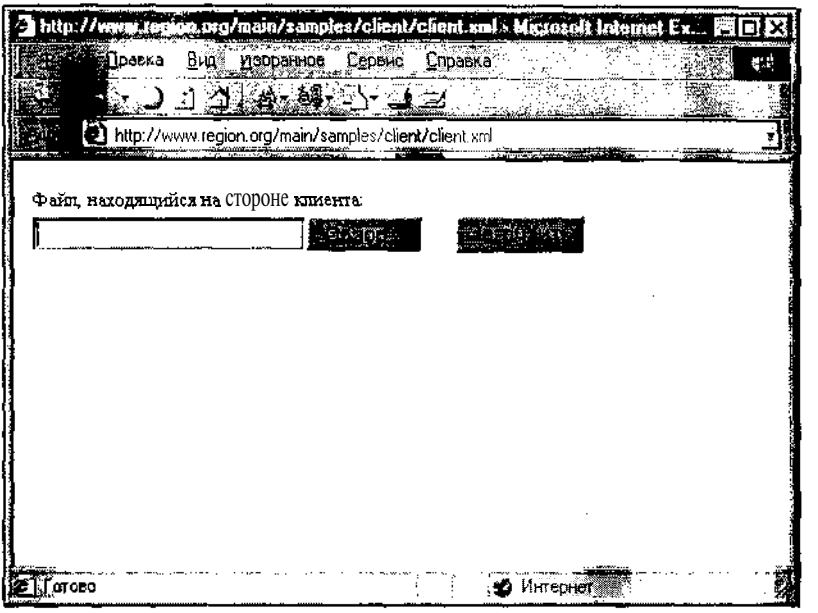


Рис. П2.5. Web-страница, предлагающая загрузку файла

Как видно из рис. П2.5, на Web-странице располагается строка для выбора того файла, который требуется загрузить, а также кнопка, с помощью которой осуществляется загрузка этого файла. При этом остальные компоненты Web-приложения невидимы. После загрузки файла Web-страница будет выглядеть так, как это изображено на рис. П2.6.

Теперь в окне браузера отображается кнопка **Сохранить,** служащая, как это следует из ее названия для сохранения файла, элемент управления Selector, служащий для выбора ключевого слова, кнопка, с помощью которой выделенное мышью слово помечается как ключевое, либо, если это слово уже помечено, пометка снимается, кнопка очистки, удаляющая содержимое файла с Web-страницы, а также само содержимое загруженного файла.

Кнопка **Загрузить** используется не только для загрузки файла, в результате выполнения сопоставленного с ней сценария JavaScript становятся видимыми те компоненты Web-приложения, которые присутствуют на рис. П2.6 и отсутствуют на рис. П2.5. Кнопка **Очистить** выполняет действия обратные действиям, выполняющимся при нажатии кнопки **Загрузить,** т. е. при ее нажатии из окна не только удаляется содержимое файла, но и те компоненты Web-Приложения, которые становятся видимыми при нажатии кнопки **Загрузить.**

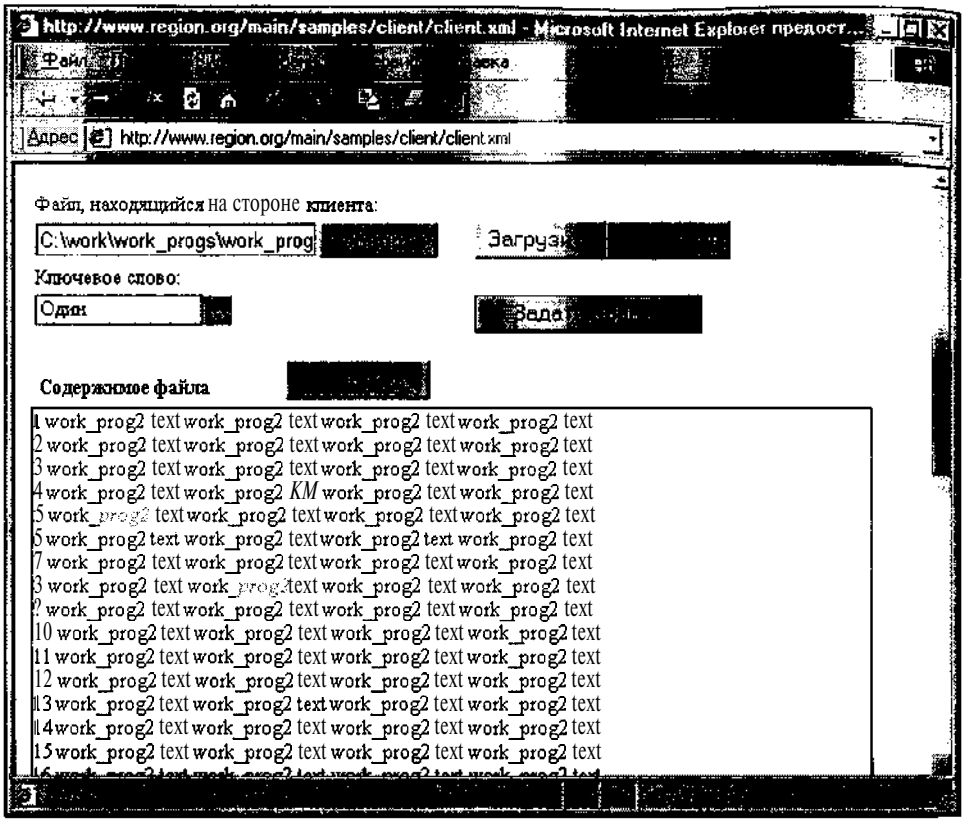


Рис. П2.6. Web-страница с загруженным файлом

Далее описывается список файлов и папок, составляющих рассматриваемое Web-приложение:

- applet — папка, содержащая сетевой апплет, представленный файлом UIDispatcher.class *(см. разд. 8.5);*
- selector — папка, содержащая файлы, описывающие элемент управления Selector *(см. разд. П2.1)*;
- client.xml — главный файл Web-приложения;
- client.xsl — сценарий XSLT для трансформации главного файла;
- client.js — содержит сценарий JavaScript для обработки действий пользователя;
- keywords.xml — содержит список ключевых слов.

Так как обработка файла ведется на стороне клиента, а также в силу того, что Web-страница не содержит какой-либо многократно в ней представленной компоненты интерфейса, контент файла client.xml имеет очень неболь-

шой объем. И как следствие, файл client.xsl содержит небольшое количество правил шаблона. В правиле шаблона для элемента документа client, как и в остальных рассматриваемых в этом приложении примерах, подключаются стилевые таблицы CSS, сценарии JavaScript, а также сетевой апплет.

Далее следует описание компонент, за представление которых отвечает рассматриваемый сценарий XSLT:

- компонент, реализующий выбор файла, представлен элементом input, значением атрибута type которого является file, а идентификатором — iFile;
- кнопки **Загрузить, Задать/Убрать и Очистить** представлены элементами button, идентификатором второй из них является bSave;
- компонент, реализующий выбор ключевого слова, представлен элементом управления Selector;
- прямоугольная область с границей, в которой располагается содержимое загруженного файла, представлена элементом div с идентификатором wp_container.

Взаиморасположение всех вышеперечисленных компонент обеспечивается с помощью таблицы (элемент table). В этой таблице строка, содержащая текст "Ключевое слово:", представлена элементом tr, который имеет идентификатор firLine. Следующая сразу после нее строка, содержащая элемент управления Selector и кнопку **Задать/Убрать**, также представлена элементом tr. Его идентификатором является secLine. Сразу после таблицы следует область, представленная элементом div с идентификатором wp_cont_note и содержащая текст "Содержимое файла".

Назначение всех этих объектов будет объяснено при описании файла client.js. Далее следует описание содержащихся в нем функций:

- load(targ, dist, arg) — служит для загрузки данных (элемента управления Selector) с сервера с помощью сетевого апплета, описана в *разд. П2.4*;
- norm(str) — служит для нормализации строки, заданной аргументом str, описана в *разд. П2.4*;
- showFile() — после загрузки файла делает видимыми объекты wp_cont_note, firLine и secLine, а также строит границу объекта wp_container;
- hideFile() — проверяет, был ли изменен загруженный файл, и если да, то сохраняет его, делает невидимыми объекты wp_cont_note, firLine и secLine, а также удаляет границу объекта wp_container и очищает его контент;

- loadFile(obj1, obj2) — **С помощью объекта** FileSystemObject *(см. разд. 8.6.3)* загружает файл, путь и имя которого заданы значением свойства value объекта iFile, переданного функции в качестве значения аргумента objl, а также делает видимой кнопку **Сохранить,** представленную объектом bsave, передаваемым функции в качестве значения аргумента obj2;
- saveFile(obj) — **С помощью объекта** FileSystemObject *(см. разд. 8.6.3)* сохраняет файл, путь и имя которого заданы значением свойства value объекта iFile, переданного функции в качестве значения аргумента obj;
- setKeyword(kword) — с помощью объекта TextRange (в книге не описывается) помечает выделенное мышью слово как ключевое, помещая это слово в контент элемента i, для которого задан атрибут keyword со значением ключевого слова, а также некоторые другие атрибуты.

Функция loado вызывается в правиле шаблона для элемента документа. С ее помощью в Web-страницу встраивается код, загруженный с сервера и служащий для представления элемента управления Selector.

Функция showFile() вызывается при нажатии на кнопку **Загрузить,** т. е. когда объект, соответствующий этой кнопке, сгенерирует событие onclick. При нажатии на эту же кнопку вызывается функция loadFile().

Функция saveFile o вызывается при нажатии на кнопку **Сохранить,** а также из тела функции hideFile(), т. е. в том случае, когда пользователь хочет убрать с Web-страницы загруженный файл, но данный файл при этом был изменен.

Функция hideFile o вызывается при нажатии на кнопку **Очистить,** а также при нажатии на кнопку Обзор, являющуюся частью объекта, соответствующего элементу input, атрибут type которого имеет значение file. В последнем случае данную функцию вызывает обработчик события onclick, генерируемого для этого объекта в том случае, когда пользователь хочет выбрать новый файл для загрузки, в то время как на Web-страницу уже был загружен и после этого изменен другой файл.

П2.6. Web-приложение, демонстрирующее совместную работу пяти предыдущих компонентов

В отличие от пяти предыдущих, данное Web-приложение способно сохранять ту информацию, которая является результатом действий пользователя. В нем реализована обработка трех типов документов, которые объединены

в информационное хранилище, находящееся на стороне сервера. Эта обработка включает в себя операции создания, удаления и редактирования документов. Первоначальный вид приложения показан на рис. П2.7.

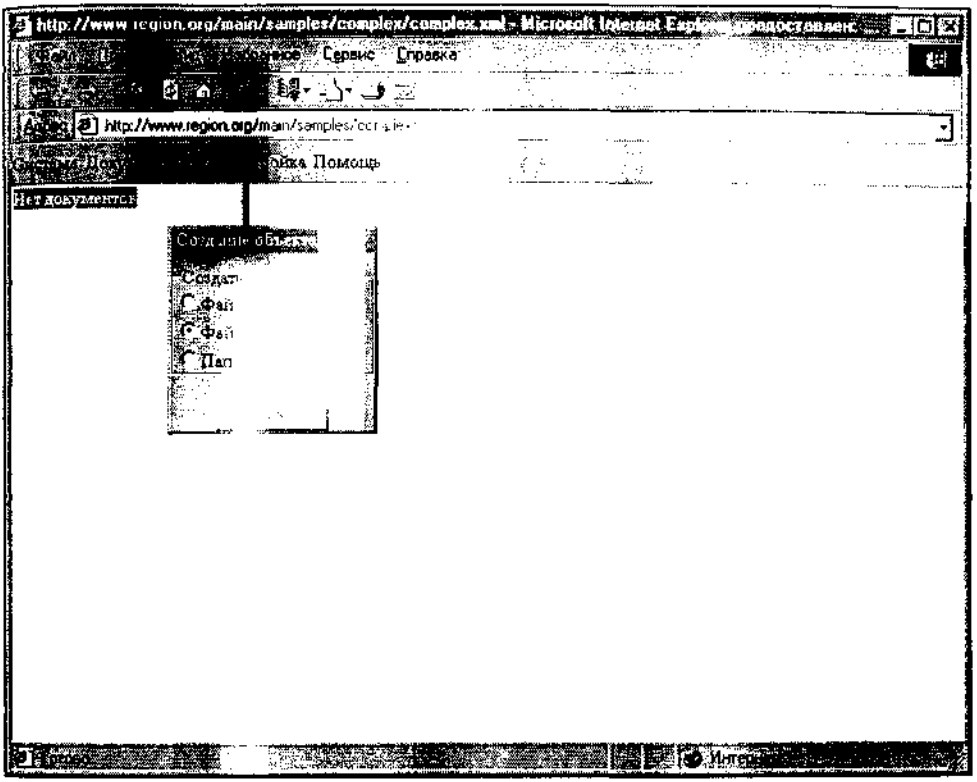


Рис. П2.7. Исходный вид Web-приложения

Как видно из рис. П2.7, в окне браузера находится меню, а рабочая область Web-приложения разделена вертикальной чертой (везде далее сепаратор) на две части. Здесь, как и в *разд. П2.3,* цвета подобраны так, чтобы главное окно Web-приложения, т. е. вся Web-страница целиком выглядела встроенной в окно браузера. Рассмотрим по отдельности каждую компоненту пользовательского интерфейса Web-приложения.

В левой части главного окна в виде дерева *(см. разд. П2.4)* представлено все множество документов, хранящихся на сервере. На рис. П2.7 в левой части Документы отсутствуют, мы создадим их несколько позднее и предоставим соответствующий рисунок. Документы, содержащиеся в хранилище, могут быть одного из следующих трех типов:

- Документ 1-го типа — в нем демонстрируется обработка файлов, находящихся как на стороне клиента, так и на стороне сервера. Этот доку-

мент представляет собой усложненный вариант Web-приложения, описанного в *разд. П2.5.*

- Документ 2-го типа — служит для демонстрации элемента управления, реализующего выбор нескольких альтернатив из списка. Он представляет собой усложненный вариант Web-приложения, описанного в *разд. П2.2.*

- Папка — узел дерева, содержащий другие его узлы.

Операции над деревом реализуются с помощью меню, реализация которого описана в *разд. П2.3.* При этом доступны следующие команды:

- Создать. Этой команде соответствует пункт меню **Создать,** находящийся в выпадающем меню, соответствующем пункту меню верхнего уровня **Документы.** На рис. П2.7 изображено окно **Создание объекта,** которое появляется при выполнении пользователем этой команды. С помощью данной команды создается узел дерева такого типа, который пользователь может выбрать с помощью этого окна. Все остальные доступные команды находятся в этом же выпадающем меню.

- Удалить. Этой команде соответствует пункт **Удалить.** С ее помощью удаляется помеченный узел дерева.

- Редактировать. Этой команде соответствует пункт **Редактировать.** С ее помощью в зависимости от типа узла осуществляется его редактирование. Например, если узлом является папка, то с помощью этой команды реализуется переименование этой папки.

При выполнении команд, соответствующих остальным пунктам меню, не происходит ничего.

Размер левой и правой частей главного окна Web-приложения может быть изменен перемещением сепаратора. Для этого пользователь должен, нажав над соответствующей сепаратору областью левую кнопку мыши, перемещать ее курсор влево или вправо.

Рассматриваемое Web-приложение составляют следующие файлы и папки:

- applet — папка, содержащая сетевой апплет, служащий для загрузки данных с сервера *(см. разд. 8.5);*

- css — папка, содержащая файлы CSS, описывающие стили компонентов главного окна, меню, дерева, а также элементов управления Chooser и Selector;

- docs — папка, в которой хранятся документы, образующие информационное хранилище;

- form1 — папка, содержащая файлы, отвечающие за создание и редактирование документов 1-го типа;

- form2 — папка, содержащая файлы, отвечающие за создание и редактирование документов 2-го типа;
- form3 — папка, содержащая файлы, отвечающие за создание и переименование папок;
- gif— папка, содержащая файлы рисунков, использующиеся для представления узлов дерева (их список приводится в *разд. П2.4*);
- js — папка, содержащая файлы JavaScript, отвечающие за пользовательские операции над главным окном Web-приложения (main.js), деревом (tree.js), меню (menu.js), а также элементами управления Chooser (chooser.js) и Selector (selector.js);
- complex.xml — главный файл Web-приложения, отвечающий за разбиение окна браузера (не главного окна Web-приложения) на две части с помощью фреймов;
- main.xml — файл, соответствующий главному окну Web-приложения;
- main.xsl — файл, содержащий сценарий XSLT для трансформации main.xml;
- command.xml — файл, отвечающий за выполнение команд над документами всех трех типов, а также служащий для выполнения серверного кода в зависимости от типа обрабатываемого документа;
- command.xsl — файл, содержащий сценарий XSLT для трансформации command.xml;
- tree.xml — файл, отвечающий за обработку дерева документов;
- tree.xsl — файл, содержащий сценарий XSLT для трансформации tree.xml;
- menu.xml — файл, содержащий структуру меню;
- menu.xsl — файл, отвечающий за представление меню.

Далее мы по отдельности опишем файлы, находящиеся в папке complex, содержащей рассматриваемое Web-приложение, а также файлы, размещенные в ее подпапках form1, form2 и form3.

П2.6.1. Файлы, содержащиеся в папке complex

В этом подразделе мы опишем также некоторые файлы из папок css и js, которые подключаются к рассматриваемым в данном подразделе файлам.

Файл complex.xml содержит описание окон, представленных элементами frame, которые являются результатом разбиения окна браузера. Целесообразность этого разбиения была нами описана в *разд. 9.6.2.* Но в отличие от своего аналога, представление файла complex.xml выглядит более эстетично, т. к. для фреймов не отображаются границы и размер этих фреймов пользо-

ватель не может изменить. Этот результат достигается с помощью манипуляции значениями атрибутов frameborder И framespacing элемента frameset, а также С ПОМОЩЬЮ атрибута noresize элемента frame.

Как видно из рис. П2.7, структура расположения компонентов Web-приложения является достаточно сложной. Поэтому для их представления в файле main.xml находятся несколько элементов, каждый из которых соответствует одному такому компоненту. Этими элементами являются:

- regions — элемент документа, соответствующий главному окну Web-приложения;
- dialog — соответствует окну, в котором пользователь выбирает тип создаваемого документа;
- region — в зависимости от значения атрибута name, соответствует либо окну, над которым располагается меню, либо левой части главного окна, либо его правой части, либо сепаратору;
- menu — представлением этого элемента является пользовательское меню.

Каждый из элементов с именем region обладает атрибутом class, значением которого является имя стиля, описание которого содержится в файле main.css, размещенном в папке css. Далее следует описание этих стилей:

- div.up — описывает стиль отображения области, располагающейся под полосой меню;
- div.left — описывает стиль области, соответствующей левой части главного окна Web-приложения;
- div.right — описывает стиль области, соответствующей правой части главного окна Web-приложения;
- div. separator — описывает стиль сепаратора;
- div.tree_text — описывает стиль области, в которой находится текстовая часть узла дерева.

Хотя последний стиль имеет отношение скорее к представлению дерева, нежели главного окна, мы включили его в main.css, т. к. бессмысленно иметь отдельный файл для описания единственного стиля, а также создавать для этого описания заголовок (с помощью элемента head), содержащий еще и элемент style, который включал бы в себя данное описание.

Далее следует список образцов, соответствующих правилам шаблона, содержащимся в файле main.xsl, а также описание этих правил:

- regions — правило шаблона для элемента документа. В контенте элемента head, содержащемся в этом шаблоне, подключаются стили CSS, сценарии JavaScript, создается набор обычных переменных и объектов, необходимых для корректной обработки пользовательских операций над пунктами меню,

узлами дерева, а также элементами управления Chooser и Selector. Кроме того, в контенте head представлен сценарий JavaScript, содержащий функцию `cmDispatch(command)`, отвечающую за обработку команд, соответствующих пунктам меню. В зависимости от значения своего аргумента данная функция выполняет создание, редактирование или удаление документа. Вызов этой функции осуществляется в теле метода onclick, принадлежащего объекту Menu, который описан в файле menu.js.

- menu — в этом шаблоне с помощью функции `load()` загружается пользовательское меню. Описание этой функции приводится *в разд. П2.4.*
- dialog — в этом шаблоне создается окно, с помощью которого пользователь может выбрать тип создаваемого документа. О принципах построения таких окон можно прочитать в *разд. 9.6.2.*
- `region[@name='__up']` — служит для создания области, располагающейся под полосой меню.
- `region[@name='__separator']` — служит для создания области, соответствующей сепаратору. Здесь при обработке события `onmousedown` переменным, использующимся при перемещении сепаратора, присваиваются значения, необходимые для выполнения этой операции.
- `region[@name='__right']` — необходима для создания области, соответствующей правой части главного окна Web-приложения.
- `region[@name='__left']` — применяется для создания области, соответствующей левой части главного окна Web-приложения. Кроме того, в этом шаблоне с помощью функции `load` осуществляется загрузка дерева документов в данную область.

Для каждого из элементов, служащих для представления области, соответствующей какому-либо из элементов region, описывается обработчик события onclick. Этот обработчик отвечает за исчезновение выпадающего меню, если оно было видимо в тот момент, когда пользователь где-либо над одной из данных областей нажал левую кнопку мыши.

Далее следует описание функций, содержащихся в файле main.js, который находится в папке js:

- `load(targ, dist, arg)` — служит для загрузки данных с сервера с помощью сетевого апплета, описана *в разд. П2.4.*
- `norm(str)` — служит для нормализации строки, заданной аргументом str, описана в *разд. П2.4.*
- `getContent(tag_name, source)` — **служит ДЛЯ вырезания "полезных"** данных из тех, которые были получены в результате зафузки, описана *в разд. П2.4.*
- `rep(select)` — используется для зафузки с сервера данных, соответствующих представлению одного узла дерева. Узел выбирается в зависи-

мости от значения аргумента select. Значение этого аргумента передается на сервер с помощью параметра HTTP-запроса с именем item. Данная функция употребляется при создании нового узла дерева. Когда новый документ создан и в хранилище информации сделаны соответствующие изменения, ветка дерева, в которой он содержится, должна быть обновлена, т. е. в ней должен появится узел, соответствующий этому документу. Вместо того чтобы обновлять все дерево целиком или даже его ветку, мы загружаем с сервера код, соответствующий данному узлу и вставляем этот узел в дерево.

- `move_wnd(obj,nX,nY)` — служит для перемещения окна, которому соответствует элемент dialog исходного дерева. Принцип работы этой функции описан в *разд. 9.6.5.*
- Resize o — служит для изменения размеров левой и правой частей главного окна в том случае, когда пользователь перемещает сепаратор. Принцип ее работы аналогичен принципу работы функции `move_wnd()`. Важным отличием является то, что перемещение сепаратора происходит только по *x*-координате.

Реализация дерева в данном Web-приложении отличается от своего аналога, описанного в *разд. 4.4,* тем, что в файлах tree.xml и tree.xsl обеспечивается загрузка данных, являющихся представлением одного узла дерева. Если значением параметра HTTP-запроса с именем item является few, то в контент tree.xml загружается ветка дерева, соответствующая папке, путь к которой задан значением параметра path, а ее имя — параметром `val`. В противном случае из файла items.xml, находящегося в этой папке (не в папке serv), загружается только один элемент, имя которого задано значением параметра item.

В правиле шаблона для элемента документа folders, содержащемся в файле tree.xsl, в зависимости от значения параметра item происходит выбор тех элементов, к которым должны быть применены остальные правила шаблона. Если значением этого параметра является `few`, то трансформируются все элементы document, загруженные из файла items.xml. В противном случае трансформируется только один такой элемент — тот, который имеет своим непосредственным потомком элемент `file_name`, контент которого совпадает со значением параметра `item`.

Еще одним отличием рассматриваемой реализации дерева является отсутствие в файле tree.xml элемента с именем folder. В этом файле все узлы дерева представлены элементом с именем document, при этом тип узла определяется значением атрибута type этого элемента. Значениями этого атрибута могут быть: `type1` — документ 1-го типа, type2 — документ 2-го типа и folder — папка. В файлах items.xml тип узла определяется контентом элемента type, являющегося непосредственным потомком каждого элемента с именем document. Этим контентом может быть текст, совпадающий с одним из допустимых значений вышеописанного атрибута type.

реализация пользовательского меню в данном Web-приложении отличается от своего аналога, описанного в *разд. П2.3,* наличием вызова функции `cmDispatch(command)` в теле метода onclick объекта Menu, который описан в файле menu.js. При этом в качестве команды, заданной аргументом `command`, используется идентификатор объекта, соответствующего тому пункту меню, над которым пользователь щелкнул левой кнопкой мыши.

файл command.xml выполняет действия, во многом схожие с действиями функции cmDispatch(), только на стороне сервера. В теле этой функции на основании значения ее аргумента определяется, какая именно команда должна быть выполнена, и далее как на стороне клиента, так и на стороне сервера выполняются действия, соответствующие этой команде. При этом действиями на стороне клиента являются различные подтверждения действий пользователя, а также редактирование узлов дерева. Эти действия выполняются В теле функции cmDispatch O .

За обработку же команды на стороне сервера отвечает файл command.xml. Этот файл обрабатывается в результате вызова функции load(), которая находится либо в теле функции `cmDispatch()`, либо (при создании нового документа) в теле обработчика события onclick, генерируемого объектом, соответствующим кнопке **OK,** находящейся в окне **Создание объекта.** При этом одним из параметров HTTP-запроса, который осуществляет функция load(), является параметр с именем command, содержащий строку, соответствующую одной из допустимых команд: create, Edit и Delete. В зависимости от того, какая команда должна быть выполнена, файлу command.xml могут быть переданы дополнительные параметры HTTP-запроса. В контенте command.xml содержатся следующие элементы:

- commands — элемент документа. В нем создается атрибут command, значением которого является команда, которую требуется выполнить. В правиле шаблона, соответствующем этому элементу, в зависимости от значения атрибута command выбирается, к какому из остальных элементов исходного дерева должно быть применено одно из остальных правил шаблона, содержащихся в файле command.xsl.
- create — элемент, для которого применяется соответствующее правило шаблона только в том случае, когда значением атрибута command, принадлежащего элементу commands, является create. В его контенте создается атрибут с именем type, значением которого является тип создаваемого документа. В правиле шаблона для элемента create в правую часть главного окна Web-приложения с помощью функции `load` загружается HTML-код, являющийся результатом обработки файлов form.xml и form.xsl. Файлы с этими именами находятся в папках form1, form2 и form3. При загрузке данного HTML-кода местоположение файлов form.xml и form.xsl определяется на основании значения атрибута type. Если этим значением является `type1`, загрузка осуществляется из папки form1, если type2 — form2 и если

folder — form3. При вызове функции load() файлу form.xml передается параметр *c* именем command, значением которого является create.

- *edit* — пустой элемент, для которого применяется соответствующее правило шаблона только в том случае, когда значением атрибута command, принадлежащего элементу commands, является edit. Соответствующее ему правило шаблона во многом похоже на то, которое соответствует элементу create. Отличие заключается в том, что при HTTP-запросе к файлу form.xml, этому файлу передаются дополнительные параметры, описывающие редактируемый элемент, а именно: путь к файлу, содержащему документ, и имя этого файла. Значением параметра command является edit.
- delete — элемент, для которого применяется соответствующее правило шаблона только в том случае, когда значением атрибута command, принадлежащего элементу commands, является delete. Правило шаблона для этого элемента содержит только директиву xsl:apply-templates, т. к. все действия, соответствующие команде Delete, выполняются на стороне сервера, а именно в контенте рассматриваемого элемента. В нем описывается Java-класс Deleter, с помощью которого производится рекурсивное удаление папки, и далее, с помощью метода fdelete(), принадлежащего этому классу, удаляется файл или папка, соответствующая удаляемому документу. При этом имя удаляемого объекта содержится в значении параметра item HTTP-запроса. Помимо удаления файла или папки из файла items.xml, путь к которому содержится в значении параметра path, удаляется элемент document, соответствующий удаляемому документу, т. е. тот, который имеет своим непосредственным потомком элемент file_name, контент которого совпадает со значением параметра item.

Прежде чем переходить к обработке документов, перечислим несколько ограничений, которые мы решили не преодолевать в связи с тем, что рассматриваемое Web-приложение является демонстрационным. При создании документа 2-го типа или папки нужно указать имя создаваемого объекта, которым он будет представлен в дереве, а также имя соответствующего ему объекта файловой системы. При задании последнего должны быть выполнены следующие условия:

1. Имя объекта не должно начинаться с двойного подчеркивания (__), т. к. с него начинаются имена всех объектов, определенных в Web-приложении. Это сделано для того, чтобы идентификатор какого-либо узла дерева не совпадал с именами других объектов JavaScript. Так может произойти потому, что в качестве имени объекта, соответствующего узлу дерева, ему присваивается имя соответствующего этому узлу файла или папки.
2. По тем же причинам для всех узлов дерева должны быть заданы разные имена, соответствующие объектам файловой системы, даже если эти узлы находятся в разных папках или сами являются папками.

При описании тех частей Web-приложения, файлы которых содержатся в папках form1, form2 и form3, мы не станем уделять внимание тому, как было достигнуто представление этих частей, а лишь опишем их внешний вид и принципы работы. Основу последующих описаний составляет логический слой данных частей Web-приложения.

П2.6.2. Файлы, содержащиеся в папке form1

Данная часть Web-приложения служит для демонстрации обработки файла как на стороне клиента, так и на стороне сервера. С ее помощью создаются и редактируются документы первого типа. Для создания этого документа (рис. П2.8) следует в меню выбрать пункт **Создать** и в появившемся окне **Создание объекта** выбрать опцию **Файл 1-го типа,** после чего нажать кнопку OK. После этого в правую часть главного окна Web-приложения будет загружен HTML-код, являющийся результатом обработки файлов form.xml и form.xsl, которые находятся в папке form1.

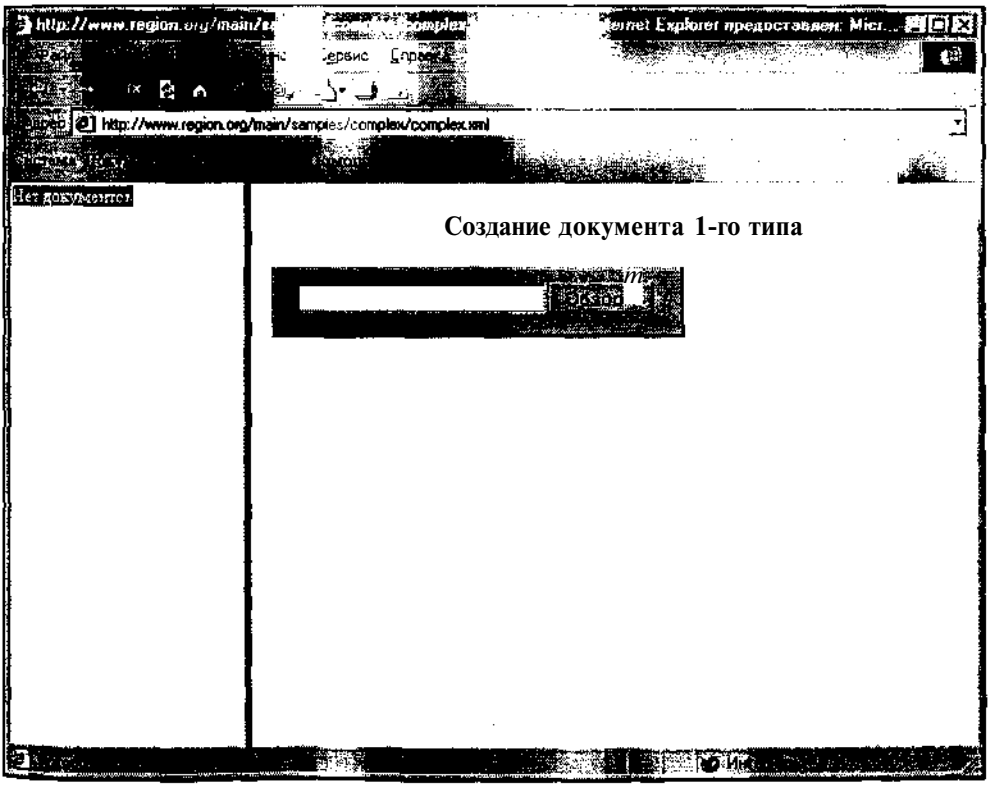


Рис. П2.8. Создание документа 1-го типа

Как видно из рис. П2.8, в правой части главного окна Web-приложения содержится элемент ввода, позволяющий пользователю выбрать файл для отправки на сервер. После того как файл выбран, в правой части окна появляются кнопки **Загрузить,** с помощью которой можно загрузить файл на Web-страницу, и **Отправить,** с помощью которой пользователь может отправить файл на сервер. Если он нажмет первую, то результат будет выглядеть приблизительно так, как на рис. П2.9.

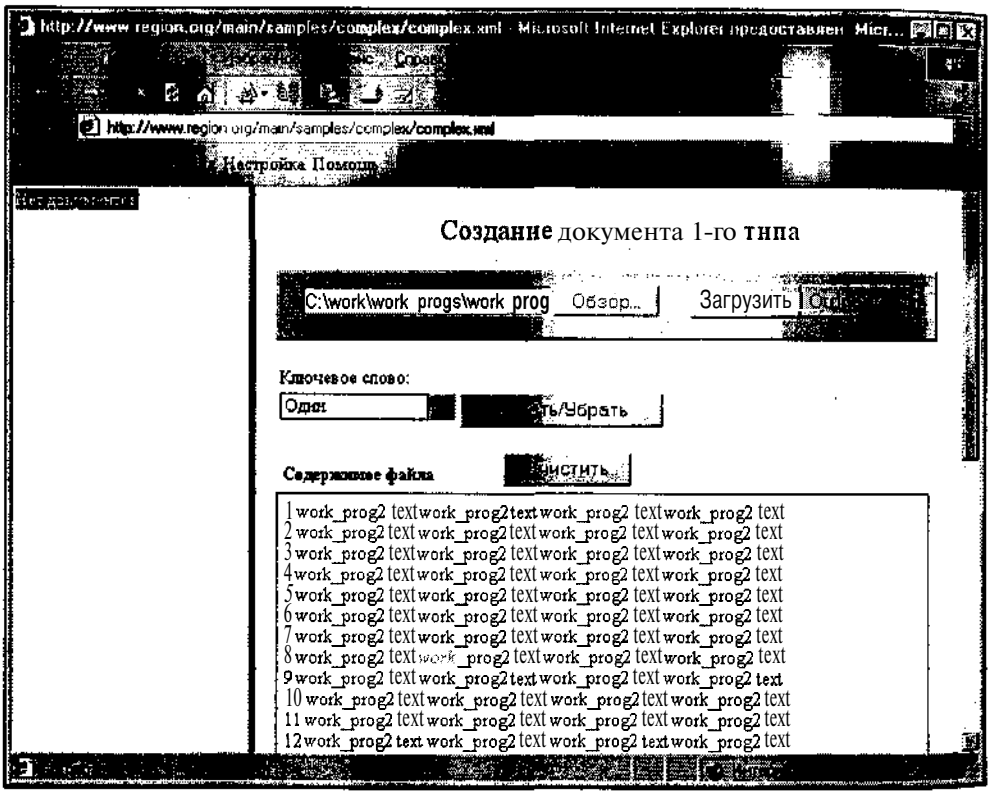


Рис. П2.9. Редактирование загруженного файла

Как видно из рис. П2.9, содержимое правой части окна мало чем отличается от того представления, которое является результатом выполнения Web-приложения, описанного в *разд. П2.5.* Хотя принцип его работы является чуть более сложным.

Документ 1-го типа создается в том случае, когда значением параметра command HTTP-запроса является create. Операции редактирования соответствует значение edit этого параметра. При редактировании внешний вид Web-приложения будет примерно таким, как на рис. П2.10.

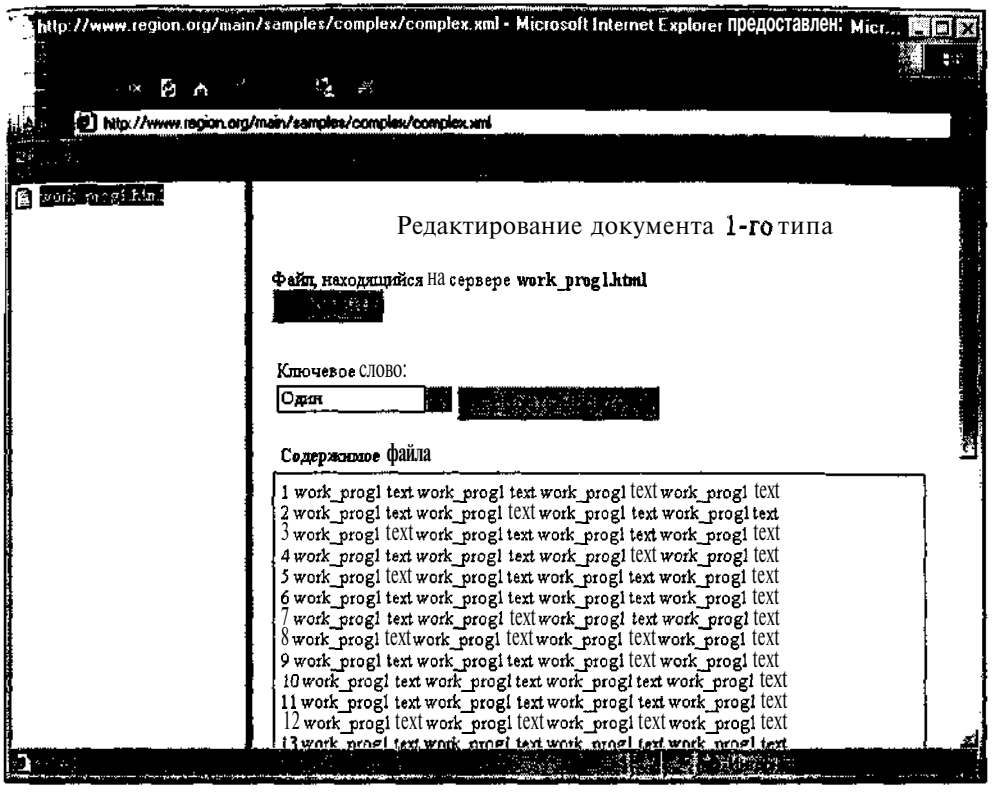


Рис. П2.10. Редактирование документа 1-го типа

Как видно из рис. П2.10, при редактировании в правой части отсутствует элемент ввода для выбора файла, но представлено содержимое хранящегося на сервере файла, который требуется отредактировать. При необходимости заменить документ на сервере, нужно его сначала удалить, а потом создать новый.

Рассмотрим принцип работы обработчика формы, с помощью которой выбранный и возможно отредактированный файл отправляется на сервер. Этот обработчик содержится в файле form_handler.xml. Данный файл отличается от своего аналога, описанного в *разд. 9.6.2,* во-первых, тем, что стиль XSLT, служащий для его трансформации, вынесен в отдельный файл с именем form_handler.xsl, а не является встроенным. Это сделано только для того, чтобы не загромождать код, соответствующий обработчику формы.

Далее поэтапно описываются основные действия, выполняющиеся во время обработки form_handler.xml:

1. Создается объект p, являющийся экземпляром класса `Params` и описывающий параметры HTTP-запроса. Далее следует список всех обрабатываемых в этом файле параметров:

- command — содержит команду для выполнения;
- path — содержит путь к файлу___items.xml, заданный относительно папки complex;
- file — содержит значение, переданное с помощью элемента ввода input, атрибут type которого имеет значение file;
- file_name — содержит имя файла, отправленного на сервер;
- file_content — содержит контент файла, отправленного на сервер

2. Переменной file_content присваивается содержимое файла. Если файл был передан с помощью элемента ввода input, значением атрибута type которого является file, то это содержимое вырезается из значения параметра file. В противном случае содержимое файла будет содержаться в значении параметра с именем file_content.
3. С помощью метода relativeFilename(), принадлежащего классу XSPUtil, переменной abspath присваивается значение абсолютного пути к файлу___items.xml, заданного относительно корневого каталога.
4. Переменной relpath присваивается значение пути к файлу___items.xml, заданное относительно папки complex. Причем reipath содержит и само имя этого файла.
5. На диске создается файл с именем, содержащимся в переменной file_name, при этом путь к этому файлу содержится в переменной abspath.
6. В созданный файл записывается содержимое переменной file_content.
7. ЕСЛИ КОМаНДОЙ, КОТОРУЮ Необходимо ВЫПОЛНИТЬ, ЯВЛЯеТСЯ **create, ТО** в файле___items.xml создается элемент с именем document. При этом контент данного элемента образуют следующие элементы:
   - type — содержит тип документа, в данном случае type1;
   - file_name — содержит имя файла на диске, соответствующего документу;
   - document_name — содержит имя документа, которым он будет представлен в дереве.

При создании и редактировании документов 1-го типа должны быть выполнены следующие условия:

1. На сервере не должны содержаться файлы с одинаковыми именами.
2. Загрузке на сервер подлежат файлы, содержимым которых является HTML-код.
3. Расширением загружаемого файла должно быть html.

4 Загружаемый файл не должен содержать символов, не принадлежащих английскому языку. В противном случае нужно задать значение используемой кодировки в качестве аргумента конструктора потока `InputStreamReader()`.

П2.6.3. Файлы, содержащиеся в папке form2

Данная часть Web-приложения демонстрирует интеграцию элемента управления Chooser в комплексное Web-приложение. В результате ее выполнения создается или редактируется документ 2-го типа. Для создания этого документа следует в окне с заголовком **Создание объекта** выбрать опцию **Документ 2-го типа** и нажать кнопку OK. На рис. П2.11 изображен результат этой операции.

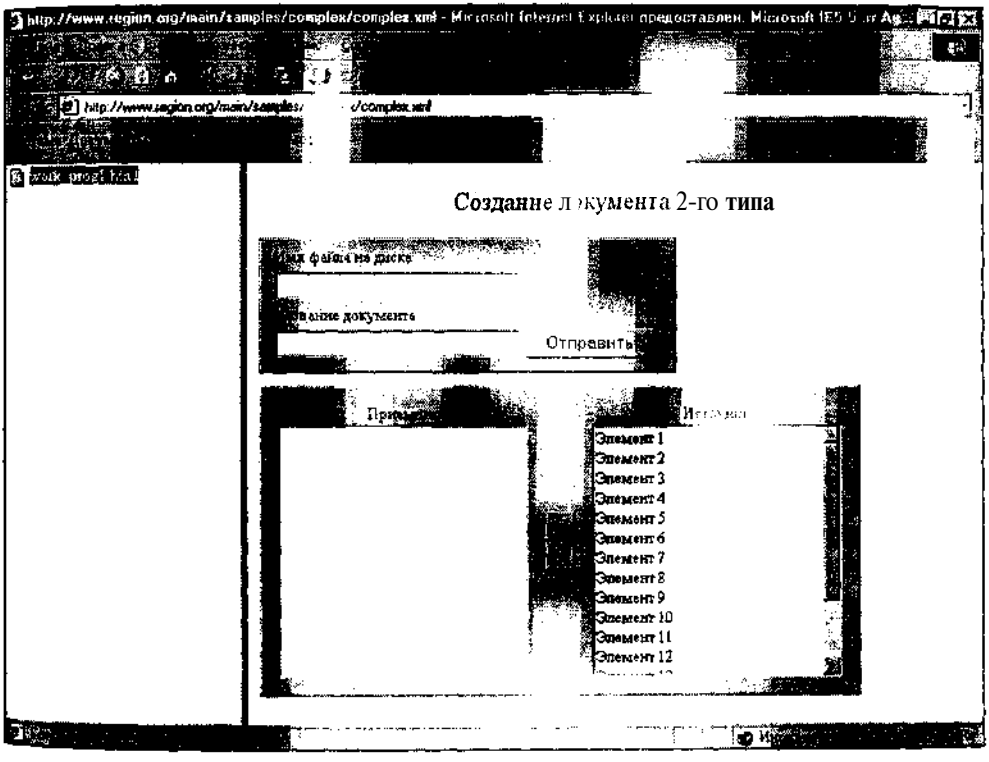


Рис. П2.11. Создание документа 2-го типа

Как видно из рис. П2.11, представление, содержащееся в правой части главного окна Web-приложения, отличается от своего аналога, описанного в *разд. П2.5*, только тем, что в нем присутствуют элементы ввода для задания имени файла, соответствующего создаваемому документу, а также тому

имени, которым будет представлен создаваемый документ в дереве. Список альтернатив, содержащийся в правой части элемента управления, находится в файле source.xml, который располагается в папке form2.

При редактировании внешний вид правой части главного окна будет отличаться от изображенного на рис. П2.11 тем, что в полях ввода будут стоять оба имени редактируемого документа.

Рассмотрим обработчик формы, с помощью которой на сервер отправляются данные, соответствующие выбранным пользователем альтернативам. Этот обработчик содержится в файле form_handler.xml, находящемся в папке form2. Далее следует описание действий, выполняющихся при обработке этого файла:

1. Для получения значений параметров HTTP-запроса создается экземпляр класса `Params`. Далее следует список всех параметров, передающихся обработчику формы:
   - command — содержит команду для выполнения;
   - path — содержит путь к файлу items.xml, заданный относительно папки complex;
   - `file_name` — содержит имя файла, соответствующего создаваемому или редактируемому документу;
   - items — строка, содержащая значения выбранных пользователем альтернатив в специальном формате (процесс формирования такой строки на стороне клиента, а также ее обработка на стороне клиента описаны *в разд. 9.6.2* и *9.6.7)*.
2. Переменной data присваивается значение, которым является содержимое файла, соответствующего создаваемому документу.
3. Переменным abspath и `relpath` присваиваются значения соответственно абсолютного и относительного пути *(см. разд. П2.6.2)*.
4. Создается файл с именем, содержащимся в значении параметра file_name, с расширением `xml`. Путь к этому файлу содержится в значении переменной abspath.
5. В созданный файл записывается значение переменной data.
6. В зависимости от того, какую команду требуется выполнить, либо (при создании документа) в файл items.xml, путь к которому содержится в значении переменной reipath, добавляется новый элемент с именем document, либо (при редактировании документа) редактируется тот элемент, у которого контентом его непосредственного потомка с именем `file_name` является значение переменной `old_value`. В значении этой переменной содержится имя файла, которое он имел до редактирования.

При редактировании также удаляется файл с именем, совпадающим со значением `old_value`, к которому добавлена строка `.xml`. Этот файл должен быть удален, т. к. и при создании и при редактировании создается новый файл, включающий описание выбранных пользователем альтернатив.

Отметим, что контентом элемента type, являющегося непосредственным потомком каждого элемента с именем document, содержащегося в файле __items.xml, будет type2.

П2.6.4. Файлы, содержащиеся в папке form3

Рассматриваемая в этом разделе часть является наиболее простой. Она используется при создании папки. Для того чтобы создать папку, нужно в окне **Создание объекта** выбрать опцию **Папка** и нажать кнопку **OK.** После этого в окне браузера появится представление примерно такое, как на рис. П2.12.

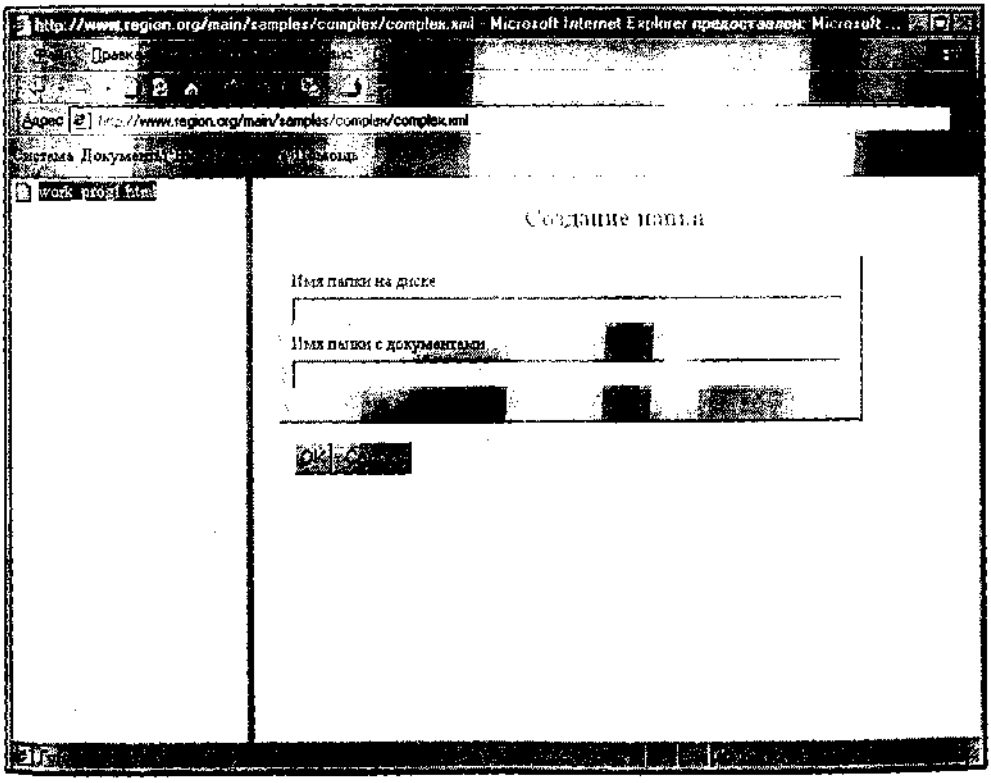


Рис. П2.12. Создание папки

Как видно из рис. П2.12, в правой части главного окна Web-приложения содержатся только два элемента ввода. Первый используется для задания имени папки на диске, второй — для задания того имени, которым данная папка будет представлена в дереве. При редактировании эти элементы ввода содержат оба имени редактируемой (переименовываемой) папки.

Рассмотрим обработчик формы, содержащийся в файле form_handler.xml, который находится в папке form3.

1. Для получения значений параметров HTTP-запроса создается экземпляр класса Params. Далее следует список всех параметров, передающихся обработчику формы:
   - command — содержит команду для выполнения;
   - path — содержит путь к файлу items.xml, заданный относительно папки complex;
   - folder_name1 — содержит имя папки на диске;
   - folder_name2 — содержит имя папки в дереве;
2. Переменным abspath и relpath присваиваются значения соответственно абсолютного и относительного пути *(см. разд. П2.6.2)*.
3. Создается папка, именем которой является значение переменной folder_name1. Путь к создаваемой папке содержится в значении переменной abspath.
4. В созданной папке создается файл items.xml, для получения пути к которому используется значение переменной abspath. Этот файл не содержит элементов с именем document, а только элемент empty, имеющий двух непосредственных потомков:
   - file_name — контентом этого элемента является строка, полученная склейкой строкового значения empty и имени папки, в которой находится созданный items.xml;
   - document_name — контентом этого элемента является строка Нет документов.
5. В той папке, в которой была создана новая папка, находится файл items.xml, и в зависимости от выполняемой команды над ним выполняются действия, аналогичные описанным в последнем пункте списка, приведенного в *разд. П2.6.3*. При редактировании папка, существующая на диске, переименовывается.

При создании или редактировании элемента с именем документ контент его непосредственных потомков будет содержать:

1. file_name – **значение параметра** folder_name1;
2. document_name – **значение параметра** folder_name2;
3. type — строковое значение folder.

ПРИЛОЖЕНИЕ 3

Установка программного обеспечения и файлов примеров

Установка ПО и примеров почти одинакова для обеих операционных систем — Windows и Linux. Поэтому мы не будем описывать процесс установки дважды, но отдельно оговорим специфику той или иной из них. Далее следует пошаговое описание настройки простейшей среды разработки, в которой могут быть запущены наши примеры. О расположении упоминаемых ниже файлов можно узнать из *приложения 4.*

П3.1. Установка Tomcat

Tomcat может быть установлен одним из как минимум двух способов. При первом из них Tomcat устанавливается как автономное приложение, при втором — интегрируясь с Web-сервером Apache. В последнем случае Apache обрабатывает все HTTP-запросы пользователей, за тем исключением, когда запрашивается файл JSP или XML. Такие запросы перенаправляются сервлет-контейнеру Tomcat. Автономный вариант установки Tomcat обеспечивается меньшим временем конфигурирования Web-среды, но из круга его возможностей следует исключить все то, что предлагает Web-сервер Apache.

Рассмотрим по порядку различные варианты установки Tomcat.

П3.1.1. Tomcat как автономное приложение

Распакуем архив jakarta-tomcat-3.2.4.tar.gz. Для этого под Windows можно воспользоваться архиватором WinRAR, а под Linux выполнить команду:

```
tar -xzf jakarta-tomcat-3.2.4.tar.gz
```

После того, как файлы распакованы, переименуем содержащую их папку в tomcat и перепишем ее в корневой каталог диска C под Windows и в каталог /usr/java под Linux.

Для последующего конфигурирования среды необходимо создать переменную среды TOMCAT_HOME, которая должна содержать путь к файлам виртуальной машины Java:

- под Windows

```
set TOMCAT_HOME=C:\tomcat
```

- под Linux

```
TOMCAT_HOME = /usr/java/tomcat
export TOMCAT_HOME
```

П3.1.2. Tomcat+Apache

Для того чтобы Tomcat и Apache работали вместе, нужно проделать все то, что было описано в *разд. П3.2.1*, а также установить Web-сервер Apache. Для установки Apache под Windows нужно запустить находящийся на CD-ROM файл его дистрибутива:

```
apache_1.3.26-win32-x86-no_src.exe
```

Под Linux нужно выполнить команду:

```
tar -xzf apache_1.3.20.tar.gz
```

где apache_1.3.20.tar.gz — файл дистрибутива Apache для Linux. В результате последней команды Web-сервер Apache будет распакован и готов к запуску.

После этого нужно в случае ОС Linux переписать модуль mod_jserv.so в папку, в которой содержатся модули Web-сервера Apache. В случае ОС Windows этот модуль будет называться ApacheModuleJServ.dll. Он входит в состав дистрибутива JServ, которому соответствует файл инсталлятора ApacheJServ-1.1.2-2.exe.

В папке $TOMCAT_HOME/conf (или $TOMCAT_HOME\conf в случае ОС Windows) находится файл tomcat-apache.conf, генерируемый при каждом запуске Tomcat и являющийся конфигурационным файлом объединенной пары приложений Tomcat+Apache. Необходимо сообщить Web-серверу о существовании этого файла. Это можно сделать тремя способами:

1. В конец конфигурационного файла Web-сервера Apache httpd.conf добавить строку
   - include `C:\tomcat\conf\tomcat-apache.conf` **В случае ОС Windows;**
   - include `/usr/java/tomcat/conf/tomcat-apache.conf` **В случае ОС Linux.**

2. Переписать файл tomcat-apache.conf в папку, в которой содержатся конфигурационные файлы Web-сервера Apache, и в конец файла httpd.conf добавить строку
   - include conf\tomcat-apache. conf **В случае ОС Windows;**
   - include conf/tomcat-apache. conf **В случае ОС Linux.**
3. Скопировать содержимое файла tomcat-apache.conf в конец файла httpd.conf.

Внимание

В последних двух случаях при каждом изменении файла server.xml, который находится в той же папке, что и tomcat-apache.conf, необходимо заново проделывать действия, описанные одном из пунктов (2 или 3) вышеприведенного списка.

Остальные различия в конфигурировании Tomcat в зависимости от способа его установки мы опишем далее при установке и настройке инструментария публикаций Cocoon.

П3.2. Установка Cocoon

1. Распаковать архив, содержащий файлы Cocoon так же, как и Tomcat. Пусть папка, содержащая Cocoon, называется cocoon.
2. Переписать файлы, содержащиеся в cocoon/lib и cocoon/bin, в папку tomcat/lib (в случае ОС Windows в путях слеш обратный).
3. Зайти в папку tomcat/lib.
4. Переименовать файл xerces_1_2.jar в 1xerces_1_2.jar.
5. Переименовать файл xml.jar в zxml.jar.
6. Запустить Tomcat.
7. Сконфигурировать Web-приложение, использующее Cocoon *(см. разд. П3.3)*.

П3.3. Конфигурирование примеров

1. В папке tomcat/webapps (tomcat\webapps под Windows) создать папку main.
2. В эту папку переписать с CD-ROM все папки примеров (chooser, client, complex, selector, menu, tree).

3. В файле server.xml, который находится в папке tomcat/conf (tomcat\conf в случае ОС Windows), находим строки

```
<Context path="/examples"
docBase="webapps/examples"
crossContext="false"
debug="0"
reloadable="true" >
</Context>
```

Эти строки описывают папку examples, находящуюся в папке tomcat/webapps (tomcat\webapps в случае ОС Windows). О сути этого описания можно прочитать в документации по Tomcat, а также в комментариях, содержащихся в файле server.xml.

4. Опишем точно так же созданную нами папку main:

```
<Context path="/main"
  docBase="webapps/main"
  crossContext="false"
  debug="0"
  reloadable="true" >
</Context>
```

и для удобства вставим это описание сразу после описания папки examples.

В папке tomcat/webapps/samples (tomcat\webapps\samples в случае ОС Windows) создать папку WEB-IMF и переписать в нее находящиеся на CD-ROM файлы web.xml и cocoon.properties.

После всех вышеперечисленных действий должны заработать все примеры, кроме дерева (папка tree) и комплексного примера (папка complex). Эти два примера не будут работать, т. к. они динамически подгружают код с сервера *(см. разд. 8.5),* обладающего доменным именем www.region.org, а комплексный пример нуждается в Java-классах, отвечающих за обработку параметров, переданных методом POST *(см. разд. 8.4).* Поэтому для работоспособности этих примеров необходимо сделать две вещи:

1. Дать серверу имя `www.region.org`. Это можно сделать, настроив соответствующим образом DNS-сервер или добавив в конец файла C:\Windows\hosts, находящегося на клиентской машине, строку

```
194.1,2.7       www.region.org
```

Здесь вместо `194.1.2.7` должен стоять IP-адрес вашего сервера.

2. Создать в папке WEB-INF папку classes и переписать в нее папку param_utils, содержащую файлы name_value.class и Params.class, после

чего добавить в переменную CLASSPATH путь к этим файлам. Это можно сделать, например, в файле startup.bat добавив в его начало строку:

```
set CLASSPATH = C:\tomcat\webapps\main\WEB-INF\classes
```

Под Linux изменять значение CLASSPATH не нужно.

Внимание

При запуске Tomcat как автономного приложения для корректного распознавания адреса `http://www.region.org` нужно изменить номер порта, на котором работает Tomcat со значения 8080 на значение 80. Для этого следует открыть файл `server.xml`, найти в нем число 8080 и заменить его на 80. Номер порта содержится В строке `<Parameter name="port" value="8080"/>`.

После всех вышеперечисленных действий работоспособными должны быть все примеры. Напомним, что при редактировании, добавлении или удалении описаний папок в файле server.xml, содержимое файла tomcat-apache.conf, генерируемого при каждом запуске Tomcat, будет другим. Поэтому, если этот файл не подключается непосредственно к httpd.conf, его нужно будет либо снова переписать в папку, содержащую конфигурационные файлы Web-сервера Apache, либо заново добавить его содержимое в файл httpd.conf *(см. разд. П3.1.2)*.

П3.4. Запуск серверных приложений

1. Запустить Web-сервер Apache:

   Пуск | Программы | Apache HTTP Server | Start Apache in Console.

2. Запустить Tomcat:

   - под Linux

     `tomcat/bin/startup.sh`

   - под Windows

     `tomcat\bin\startup.bat`

Примечание

При использовании на сервере операционной системы Win9x непосредственно запустить startup.bat не удастся, т. к. не хватит памяти для всех переменных среды. Но его можно запустить следующим образом: активировать пункт **Выполнить** в главном меню Windows и в появившейся строке набрать команду:

```
command.com /E:4096 /C c:\tomcat\bin\startup.bat
```

а для завершения работы в этой строке нужно набрать соответственно:

```
corranand.com /E:4096 /C c:\tomcat\bin\shutdown.bat
```

Теперь, установив на клиентскую машину браузер MSIE5.5, получим полностью сконфигурированную для наших целей среду выполнения Web-приложений.

П3.5. Установка Cocoon2

В этом разделе мы опишем, как установить Cocoon2, если Tomcat установлен как автономное приложение.

1. Распакуйте архив cocoon-2.0.1-bin.tar.gz так, как это было описано в *разд. П3.1.1.* Для удобства обозначим папку, содержащую файлы Cocoon2, как cocoon2.
2. Выполните действия, описанные в *разд. П3. 1.*
3. Перепишите в папку tomcat/webapps (tomcat\webapps в случае ОС Windows) файл cocoon.war из папки cocoon2.
4. Если Tomcat запущен, остановите его, после чего запустите, в результате чего файл cocoon.war будет распакован и в папке webapps появится папка cocoon.
5. Остановите Tomcat.
6. В папке tomcat/lib (tomcat\lib в случае ОС Windows) удалите файл jaxp.jar.
7. Из папки tomcat/webapps/cocoon/WEB-INF/lib (tomcat\webapps\cocoon\WEB-INF\lib в случае ОС Windows) в папку tomcat/lib (tomcat\lib в случае ОС Windows) перепишите файлы xml-apis.jar и xerces-1.4.4.jar.
8. Запустите Tomcat.
9. Если сервер имеет имя `www.server.org`, то набрав в строке браузера `http://www.server.org:8080/cocoon`, через некоторое время в окне браузера можно увидеть главную страницу Cocoon2.

П3.5.1. Особенности установки Cocoon2 под Linux

При использовании JDK от фирмы SUN его реализация библиотеки AWT постоянно пытается соединиться с X-сервером, даже в том случае, когда не требуется ничего отображать. Для решения этой проблемы проще всего воспользоваться реализацией AWT от других разработчиков, а именно от eTecks.

Данная реализация представлена продуктом, который носит имя PJA (Pure Java AWT — AWT на чистом Java) и номер версии 2.4. Для того чтобы воспользоваться возможностями, предлагаемыми PJA, нужно:

1. Распаковать находящийся на CD-ROM файл pja_2.4.zip, содержащий дистрибутив PJA, и из папки lib, которая находится в папке, содержащей

все файлы этого дистрибутива, в папку tomcat/lib переписать два файла — pja.jar и pjatools.jar.

2. В файл startup.sh, находящийся в tomcat/bin, перед строкой

```
$BASEDIR/tomcat.sh start "$@"
```

вставить следующие четыре строки

```
export TOMCAT_OPTS="-Xbootclasspath/a:../lib/pja.jar \
-Dawt.toolkit=com.eteks.awt.PJAToolkit \
-Djava.awt.graphicsenv=com.eteks.java2d.PJAGraphicsEnvironment \
-Djava2d.font.usePlatformFont=false"
```

После этого, как и при Windows-установке, можно обращаться к главной странице Cocoon2.

ПРИЛОЖЕНИЕ 4

Описание компакт-диска

Далее следует дерево каталогов CD-ROM.

\	
software	
Windows	Программное обеспечение для Windows
apache_1.3.26-win32-x86-no_src.exe	Web-server Apache
ApacheJServ-1.1.2-2.exe	Необходим для интеграции Apache и Tomcat
Linux	Программное обеспечение для Linux
apache_1.3.20.tar.gz	Web-server Apache
mod_jserv.so	Необходим для интеграции Apache и Tomcat
Both	Кроссплатформенное программное обеспечение
cocoon-1.8.2.tar.gz	Инструментарий публикаций Cocoon
jakarta-tomcat-3.2.4.tar.gz	Сервлет-контейнер Tomcat
Additional	Дополнительное программное обеспечение
cocoon-2.0.1-bin.tar.gz	Cocoon2
Pja_2.4.zip	Улучшенная версия библиотеки AWT
samples	Примеры
chooser	Элемент управления chooser

Client	Обработка файлов на стороне клиента
complex	Комплексный пример
menu	Меню
selector	Селектор
tree	Дерево
misc	Различные дополнения
classes	Классы для обработки параметров Web-формы
name_value.java	
name_value.class	
Params.java	
Params.class	
work_progs	Типовые текстовые файлы для загрузки на сервер
work_prog1.html	
work_prog2.html	
work_prog3.html	
work_prog4.html	
conf	Конфигурационные файлы для Cocoon
cocoon.properties	
web.xml	

Предметный указатель

B

Box-модель 372

C

CDATA 59
child 54
Cocoon 70, 343, 411
comment() 99
CSS 352

D

DTD 49, 68, 283

E

encoding 48, 75
ENTITY 62

N

node() 99
notation 65

P

priority 86
processing-instruction() 99

Q

qualified name 56

R

request 411
response 411

S

sibling 54
standalone 49
stylesheet 73

T

text() 99

V

version 48

X

xml:lang 58
xml:space 58
xsl:apply-imports 90
xsl:apply-templates 88

xsl:attribute 125
xsl:attribute-set 128
xsl:call-template 95
xsl:choose 109
xsl:comment 132
xsl:copy 132
xsl:copy-of 134
xsl:decimal-format 206
xsl:element 123
xsl:for-each 106
xsl:if 108
xsl:import 80
xsl:include 80
xsl:key 201
xsl:message 120
xsl:number 110
xsl:otherwise 109
xsl:output 135
xsl:param 101, 105
xsl:preserve-space 135
xsl:processing-instruction 131
xsl:sort 119
xsl:strip-space 135
xsl:stylesheet 77, 280
xsl:template 84
xsl:text 130
xsl:transform 77, 280
xsl:value-of 97
xsl:variable 101
xsl:when 109
xsl:with-param 105
xsp:attribute 127, 347
xsp:comment 132, 348
xsp:content 347
xsp:element 124, 347
xsp:expr 347
xsp:include 347
xsp:logic 347
xsp:page 347
xsp:pi 348
xsp:structure 347

А

Адресация 83, 141
Атрибут 57, 82, 165

Б

Бокс:
- блочный 373
- внутритекстовый 373
- невидимый 375

В

Версия 48
Выражение 141

Г

Граница прямоугольной области 377
Группа рабочая:
- администраторов 295
- дизайнеров 310
- пользователей 295
- программистов 310
- разработчиков 295
- редакторов 312
- художников 310

Д

Декларация:
- стиля объекта 354
- типа документа 49

Дерево 46
- импорта 80
- исходное 73, 84
- конечное 73, 84

Документ:
- действительный 70
- корректный 45, 70

Документооборот 295

И

Идентификатор:
- O публичный 66
- o системный 66

Имя, полное элемента 56
Инструкция обработки 49, 70, 82

К

Ключ 201
Кодировка 48, 75, 174
Комментарий 60, 82
Контент 46
Координаты прямоугольной области 379
Корень 82

М

Маршрут прохождения документа 295
Метод:
- o вывода конечного дерева 136
- O отправки параметров HTTP-запроса 400, 416, 422

Модель, трехуровневая Web-приложения 336
Модель данных 46, 299
- o иерархическая 299
- o реляционная 299

Н

Нотация 65

О

Образец 84, 174
Объявление:
- o XML 67
- O сущности 62
- O типа документа 67

Ось 147
- O обратная 161
- o по умолчанию 164
- o прямая 161

Отбор узлов 82, 83, 145

П

Параметр 101, 105
Переменная 101
Подключение таблиц CSS к документу 355
Поле:
- O внешнее 376
- O внутреннее 378

Потомок 54
Правило:
- O проверки узлов 159
- O стиля CSS 354
- o шаблона встроенные 91
- O шаблона именованное 94

Правило шаблона 84
Предикат 160
Предок 54
Представление 73
Приоритет импорта таблиц XSLT 80
Пролог 67
Пространство имен 56, 76, 82, 284
Псевдокласс 361
Псевдоэлемент 363
Путь адресации 141
- O абсолютный 146
- o относительный 144

Р

Разметка 46, 59
Расширения:
- o функции 285
- O элементы 285

Режим 93
Русскоязычный 216, 322, 420

С

Селектор объектов CSS 354, 357
Сервлет 411
Символ пробельный 134
Слой:
- O логический 313
- O представления 313
- O структуры 313

Ссылка:
- O на переменную 102
- O на сущность 62

Структура 45
Сущность 61
  o внешняя 66
  O внутренняя 66
  o именованная 64
  o неанализируемая 65
  o нумерованная 63
  O параметрическая 69
  O предопределенная 64
  O символьная 63
  O со смешанным контентом 65
Схема позиционирования
  o абсолютная 383
  O нормальная 380
  o относительная 380
  O фиксированная 384
Сценарий XSLT 55

Т

Таблица каскадная стилевая 310, 352
Таблица стилевая
  для трансформации 73
Текст 59
Текстовый узел 82
Текущий узел 144, 165
Тип:
  O данных 166
  o документа декларация 283
  O переменной 101
  O узла 82
  O узлов основной 159
Тип данных, набор узлов 168

У

Узел 53

Ф

Форма 309, 399, 406

Ш

Шаблон 84
Шаг адресации 145, 146

Э

Элемент 50, 82
  O верхнего уровня 78
  O корневой 46
  O пустой 50, 54
  o стиля 77

Я

Язык русский 55, 75