

Ильдар Хабибуллин

САМОУЧИТЕЛЬ

XML

Санкт-Петербург

«БХВ-Петербург»

2003

УДК 681.3.068+800.92XML
ББК 32.973.26-018.1
X12

Хабибуллин И. Ш.

X12 Самоучитель XML. — СПб.: БХВ-Петербург, 2003. — 336 с.: ил.
ISBN 5-94157-339-1

Книга предназначена для самостоятельного изучения новой компьютерной технологии — XML (eXtensible Markup Language, расширяемый язык разметки), быстро проникающей буквально во все области обмена информацией. Представлены новейшие аспекты технологии XML — язык описания документов XSD, язык создания запросов XQuery, форматирование документов на языке XSL-FO. Начав с основ технологии XML, автор выводит читателя на уровень самостоятельного создания программ-обработчиков документов XML. Изложение основано на авторском курсе лекций, его отличает краткость и простота. Большое количество примеров и упражнений позволяет глубоко освоить материал.

Для программистов

УДК 681.3.068+800.92XML
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Наталья Сержантова</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн обложки	<i>Игоря Цырульников</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 11.09.03.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 27,1.

Тираж 4000 экз. Заказ №

"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-339-1

© Хабибуллин И. Ш., 2003
© Оформление, издательство "БХВ-Петербург", 2003

Содержание

Предисловие	11
Введение	13
Выделение фрагментов текста	14
Стандарт SGML.....	16
Язык разметок XML.....	17
Структура книги	18
 ЧАСТЬ I. КОНСТРУКЦИИ ЯЗЫКА XML И ЕГО РЕАЛИЗАЦИЙ	21
 Глава 1. Структура документа XML	23
Пролог документа XML.....	24
Пример: разметка адресной книжки	25
Упражнения	26
Корневой элемент	26
Верный документ	27
Элементы документа XML.....	28
Ссылки на сущности.....	30
Секция CDATA.....	30
Комментарии	31
Атрибуты.....	31
Имена.....	33
Пространства имен XML.....	34
Упражнения	38
Инструкции по обработке	38
Информационное множество XML	39
Единица информации документа	39
Единица информации элемента	40
Единица информации атрибута	40
Единица информации инструкции по обработке	40
Единица информации символа.....	41
Единица информации комментария	41
Единица информации пространства имен	41

Единица информации DTD	41
Единица информации объявления DTD	41
Единица информации ссылки на сущность	42
Единица информации необрабатываемой секции	42
Примеры	42
Разметка книги	42
Упражнение	45
Таблицы	45
Упражнения	47
Вопросы для самопроверки	47
Глава 2. Описание структуры документа средствами DTD	49
Конструкции DTD	49
Объявление типа элемента	49
Упражнения	51
Объявление атрибутов	52
Упражнения	54
Объявление сущности	54
Упражнения	56
Объявление обозначения	56
Пример: описание DTD записной книжки	57
Размещение описания DTD	58
Программы-анализаторы XML	60
Заключение	61
Вопросы для самопроверки	61
Глава 3. Описание схемы документа на языке XSD	62
Встроенные простые типы XSD	63
Вещественные числа	63
Целые числа	63
Строки символов	64
Дата и время	64
Двоичные типы	65
Прочие встроенные простые типы	65
Определение простых типов	65
Сужение	65
Список	67
Объединение	68
Упражнения	69
Объявление элементов и их атрибутов	69
Упражнение	70
Определение сложных типов	70
Определение типа пустого элемента	71
Упражнения	71
Определение типа элемента с простым телом	71
Упражнение	72
Определение типа вложенных элементов	72
Упражнения	74
Определение типа со сложным телом	75

Пример: схема адресной книги	77
Безымянные типы	79
Пространства имен языка XSD	81
Включение файлов схемы в другую схему.....	84
Связь документа XML со своей схемой	85
Другие языки описания схем	86
Вопросы для самопроверки.....	87

Глава 4. Создание ссылок на языке XLink 88

Пространство имен языка XLink	89
Атрибут <i>title</i>	90
Атрибут <i>label</i>	91
Атрибут <i>href</i>	92
Атрибут <i>type</i>	92
Типы ссылок.....	93
Тип <i>none</i>	93
Тип <i>locator</i>	94
Тип <i>simple</i>	95
Упражнение.....	95
Тип <i>extended</i>	96
Упражнение.....	96
Тип <i>title</i>	96
Тип <i>resource</i>	97
Тип <i>arc</i>	97
Атрибуты <i>from</i> и <i>to</i>	98
Упражнение.....	99
Атрибут <i>show</i>	99
Упражнение.....	99
Атрибут <i>actuate</i>	99
Атрибут <i>role</i>	100
Атрибут <i>arcrole</i>	101
Создание банка ссылок	101
Программы-обработчики атрибутов XLink.....	102
Вопросы для самопроверки.....	103

Глава 5. Уточненные ссылки XPointer 104

Простые указатели	106
Использование простых указателей в ссылках.....	107
Упражнение.....	107
Указатели, основанные на схеме	107
Использование указателей в ссылках.....	108
Понятие схемы в языке XPointer.....	109
Схема <i>element()</i>	109
Упражнения	111
Схема <i>xpointer()</i>	111
Примеры.....	112
Дерево документа	115
Упражнения	116

Дополнения языка XPointer	116
Функции языка XPointer	117
Функция <i>string-range()</i>	117
Функция <i>start-point()</i>	119
Функция <i>end-point()</i>	119
Функция <i>range-to()</i>	119
Функция <i>range()</i>	120
Функция <i>range-inside()</i>	120
Функция <i>here()</i>	120
Функция <i>origin()</i>	120
Упражнения	120
Схема <i>xmlns()</i>	120
Программы-обработчики XPointer	122
Вопросы для самопроверки	122
Глава 6. Адресация на языке XPath	123
Дерево документа	123
Узлы дерева	124
Атомарные значения	125
Последовательности	126
Выражения, определяющие путь	126
Шаг, направляемый осью поиска	126
Оси поиска	127
Тесты узла	130
Тест по имени узла	130
Тест по виду узла	131
Упражнения	134
Предикаты	135
Упражнения	136
Шаг, направляемый фильтром	136
Выражения	136
Переменные	137
Арифметические операции	137
Сравнения	138
Логические операции	138
Условные выражения	139
Циклы	139
Кванторы	140
Операции с множествами	141
Функции	142
Числовые функции	142
Строковые функции	143
Функции даты и времени	144
Функции узлов	145
Функции последовательности	145
Функции, создающие последовательности	146
Упражнения	147
Вопросы для самопроверки	147

Глава 7. Язык запросов XQuery	148
Конструкторы	149
Прямой конструктор элемента.....	149
Выражения в содержимом конструктора	149
Выражения в атрибутах конструктора	150
Вычисляемый конструктор.....	151
Вычисляемые конструкторы элемента и атрибута	151
Конструктор корневого узла документа	152
Конструктор текстового узла.....	152
Выражение запроса FLWOR	152
Примеры запросов.....	155
Упражнения	161
Оператор варианта.....	162
Упражнение.....	162
Функции пользователя	163
Упражнение.....	164
Пролог	164
Определение пространств имен	165
Импорт схемы.....	166
Определение переменных.....	166
Импорт модуля	166
Пример главного модуля	167
Реализации XQuery	168
Вопросы для самопроверки.....	169
 ЧАСТЬ II. ОБРАБОТКА ДОКУМЕНТОВ XML	 171
Глава 8. Преобразование документов средствами XSLT	173
Таблицы стилей CSS в языке XML.....	176
Язык описания стилей XSL	177
Язык записи преобразований XSLT.....	178
Несложное форматирование вывода.....	181
Включение таблицы стилей в документ XML.....	184
Преобразование документа XML в документ HTML.....	185
Ресурсы языка XSLT.....	187
Образцы (patterns)	188
Функции <i>id()</i> и <i>key()</i>	188
Элементы, объявленные в XSLT	190
Декларация <i>xsl:import</i>	190
Декларация <i>xsl:include</i>	191
Декларация <i>xsl:import-schema</i>	191
Декларация <i>xsl:variable</i>	192
Декларация <i>xsl:param</i>	193
Элемент <i>xsl:with-param</i>	194
Инструкция <i>xsl:value-of</i>	194
Инструкции управления <i>xsl:if</i> , <i>xsl:for-each</i> , <i>xsl:choose</i>	195
Упражнения	196

Декларация <i>xsl:function</i>	197
Упражнение.....	198
Декларация <i>xsl:template</i>	198
Упражнения.....	200
Инструкция <i>xsl:apply-templates</i>	200
Инструкция <i>xsl:call-template</i>	200
Инструкции <i>xsl:attribute</i> , <i>xsl:element</i> , декларация <i>xsl:attribute-set</i>	201
Инструкции <i>xsl:copy</i> , <i>xsl:copy-of</i>	202
Элемент <i>xsl:sort</i> , декларация <i>xsl:sort-key</i>	203
Декларация <i>xsl:key</i>	205
Декларация <i>xsl:output</i>	206
Инструкция <i>xsl:result-document</i>	207
Последовательность преобразований.....	207
Применение правил преобразования.....	208
Создание преобразованных узлов.....	209
Тождественное преобразование.....	210
Отбор отдельных узлов.....	212
Группировка элементов.....	213
Инструкция <i>xsl:for-each-group</i>	214
Решение задачи.....	215
Вывод нескольких документов.....	216
Процессоры XSLT.....	218
Вопросы для самопроверки.....	219
Глава 9. Форматирование объектов XSL-FO.....	220
Язык XSL.....	221
Единицы измерения.....	224
Цвет.....	225
Форматирование блока.....	226
Стенографические свойства.....	226
Составные атрибуты.....	226
Расстояние между блоками.....	227
Форматирование абзаца.....	228
Упражнения.....	230
Форматирование текста.....	230
Упражнение.....	231
Вставка изображения.....	231
Упражнение.....	231
Горизонтальные линии.....	231
Упражнение.....	233
Форматирование страницы.....	233
Граница.....	236
Упражнение.....	238
Промежуточная область.....	238
Сноски.....	239
Колонтитулы, номера страниц и другое оформление.....	239
Упражнение.....	241
Фон.....	241
Упражнение.....	243

Списки.....	243
Упражнение.....	247
Таблицы.....	247
Форматеры XSL.....	250
Вопросы для самопроверки.....	250
Глава 10. Обработка документов XML при помощи событий.....	252
Стандартные средства Java для обработки XML.....	252
Анализ документа XML.....	253
Принципы анализа с помощью SAX2 API.....	255
Извлечение содержимого документа XML.....	259
Инициализация SAX2-анализатора.....	261
Упражнение.....	262
Поиск элемента в документе XML.....	262
Упражнение.....	264
Извлечение различных сведений.....	264
Упражнения.....	266
Дополнительные события SAX.....	266
Упражнение.....	270
Цепочка анализаторов.....	270
Преобразование элементов XML в объекты Java.....	273
Связывание данных XML с объектами Java.....	279
Объекты данных JDO.....	288
Вопросы для самопроверки.....	289
Глава 11. Обработка документов при помощи DOM.....	291
DOM-анализатор фирмы Sun.....	292
Основные интерфейсы DOM API.....	294
Интерфейс <i>Node</i>	295
Интерфейс <i>Document</i>	296
Интерфейс <i>Element</i>	298
Обход дерева DOM.....	299
Упражнение.....	303
Обход дерева методами DOM API.....	304
Интерфейс <i>NodeIterator</i>	304
Интерфейс <i>TreeWalker</i>	304
Интерфейс <i>DocumentTraversal</i>	305
Интерфейс <i>NodeFilter</i>	306
Прочие узлы дерева DOM API.....	310
Интерфейс <i>Attr</i>	310
Интерфейс <i>ProcessingInstruction</i>	310
Интерфейс <i>CharacterData</i>	311
Интерфейс <i>Text</i>	311
Интерфейс <i>Comment</i>	312
Интерфейс <i>CDATASection</i>	312
Интерфейсы <i>Entity</i> , <i>EntityReference</i> , <i>Notation</i>	312
Интерфейс <i>NodeList</i>	312
Конструирование нового дерева.....	312
Упражнение.....	313

Добавление элемента в дерево DOM	313
Прочие модули DOM API	316
Модуль HTML	317
Модуль Style	317
Модуль Views	319
Модуль Events	320
Интерфейс <i>Event</i>	320
Интерфейс <i>MutationEvent</i>	321
Интерфейс <i>UIEvent</i>	322
Интерфейс <i>MouseEvent</i>	322
Интерфейс <i>EventTarget</i>	322
Интерфейс <i>EventListener</i>	323
Обработка события	323
Модуль Range	324
Другие DOM-анализаторы	325
Вопросы для самопроверки	326
 Список использованной литературы	327
 Предметный указатель	329

Предисловие

Еще несколько десятков лет назад придуман удобный способ хранения представленной в виде таблиц информации — реляционные базы данных. Всевозможные списки, реестры, счета, квитанции сохраняются в таблицах баз данных и легко извлекаются оттуда в удобной для анализа форме. Гораздо хуже обстоит дело с информацией, которую неудобно представить в виде таблицы. Статьи, книги, отчеты, приказы, инструкции приходится хранить в виде файлов или "складывать" в одну ячейку таблицы реляционной базы данных. Это значительно затрудняет решение очень важной и актуальной задачи — поиска и извлечения информации.

Есть много подходов к решению этой задачи. Из документов выбираются ключевые слова, строятся полные индексы и словари, применяются методы нечеткого поиска. Еще один подход, приведший к появлению XML (eXtensible Markup Language, расширяемый язык разметок), заключается во внесении структуры в документ. Раз уж не удастся разбить документы на строки и столбцы, внося в них структуру таблицы, то, может быть, удастся создать другую структуру? С незапамятных времен документы разбивают на главы, разделы, параграфы, абзацы. Это значительно облегчает поиск информации по оглавлению документа. Если развить и уточнить правила разбиения документов на отдельные элементы, то можно будет добиться быстрого поиска и выделения нужных сведений.

Эта идея разметки и разбиения документов оказалась чрезвычайно плодотворной. В сущности, весь Интернет в той его части, которая называется World Wide Web, пользуется ею. Технология XML бурно развивает ее уже более пяти лет, охватывая все новые и новые области хранения и обработки данных. Разбиение документа на структурные элементы облегчает не только поиск, но и анализ данных, получение из них новых выводов, того, что называется "data mining". Поэтому все больше и больше информации хранится в виде документов XML. Многие системы управления базами данных, такие как Oracle, предоставляют средства удобного хранения документов XML.

Разрабатываются базы данных, специально предназначенные для хранения информации в виде XML. Самые популярные браузеры, такие как Internet Explorer, Mozilla, Opera, "учатся" показывать документы XML. Многие текстовые процессоры, даже Microsoft Office, начинают переходить на формат XML.

Технология XML сейчас находится на подъеме. Она еще сравнительно невелика, ее описание уместилось в одну книгу, которую вы сейчас держите в руках. Будущее предвещает ее дальнейшее развитие, значительное усложнение и укрупнение. Поэтому если уж изучать XML, то начинать надо прямо сейчас!

Введение

Как известно, компьютер "понимает" только язык электромагнитных сигналов, имеющих два уровня. Один уровень обозначается нулем, другой — единицей. Поэтому самый простой способ записи текста в компьютере заключается в том, чтобы закодировать каждый символ текста какой-нибудь последовательностью нулей и единиц. Имея в распоряжении один электромагнитный импульс, можно закодировать два символа, обозначив один из них единицей, а другой нулем. С помощью двух импульсов можно закодировать уже четыре символа. Например, можно обозначить букву А комбинацией 00, букву Б — 01, букву В — 10, а букву Г — двумя единицами 11. Три импульса дают возможность закодировать восемь символов и т. д.

Обычные английские тексты используют около сотни символов: заглавные и строчные буквы латиницы, цифры, знаки препинания, некоторые математические символы. Для их кодирования применяют комбинации из семи символов, набор которых давно стандартизирован. Этот стандарт называется ASCII (American Standard Code for Information Interchange). Число семь неудобно для компьютера, в машине лучше оперировать степенями двойки. Поэтому в коде ASCII обычно применяют комбинации не из семи, а из восьми нулей и единиц — *байты*, начиная каждую комбинацию с нуля. В стандарте ASCII с использованием восьми разрядов первая буква английского алфавита А кодируется комбинацией 01000001, вторая буква В — комбинацией 01000010, а, например, знак процента % — комбинацией 00100101.

Многие языки, в том числе и русский, требуют записи дополнительных символов — букв кириллицы, символов с диакритическими знаками. Такие символы просто добавляются к набору ASCII. Для их записи используют комбинации из восьми нулей и единиц, начинающиеся с единицы. Разработаны международные стандарты для кодирования дополнительных символов. Символы европейских алфавитов с диакритическими знаками образуют кодировку ISO8859-1, часто называемую Latin 1. Буквы греческого алфавита попали в кодировку ISO8859-2, обычно определяемую как Latin 2.

Для кодирования кириллицы разработана международная кодировка ISO8859-5, но она редко применяется в России. Гораздо чаще используются другие кодировки. На русифицированных компьютерах, управляемых операционными системами MS Windows, применяется кодировка CP1251, разработанная корпорацией Microsoft. На машинах, работающих под управлением операционных систем семейства UNIX, используется кодировка KOI8-R. Со времен MS DOS осталась кодировка CP866. Машины Apple Macintosh применяют кодировку MacCyrillic.

Это многообразие кириллических кодировок приводит к дополнительной нагрузке на пользователя. Для того чтобы правильно прочесть кириллический текст, он должен знать, в какой кодировке текст был записан, или перебрать несколько кодировок в своем текстовом редакторе.

Итак, для записи простого, как говорят, "плоского" (flat) текста, применяются байтовые кодировки: один символ — один байт. Это удобно для передачи текста. На любом компьютере есть средства для чтения плоского текста. Если знать его кодировку, то текст будет правильно прочитан. Мировое компьютерное сообщество решило освободиться и от этого условия и перейти на двухбайтовую шестнадцатирядную кодировку, позволяющую закодировать 65 536 символов. Такая кодировка называется *Unicode*. В ней перед каждым кодом Latin 1 стоит нулевой байт, перед каждым кодом кириллицы — байт 00000100 и т. д. Для текстов, записанных в Unicode, нет необходимости указывать кодировку, но сам текст занимает ровно в два раза больше места в файле.

Компромиссное решение предоставляет кодировка *UTF-8* (Unicode Transfer Format). В ней символы, входящие в ASCII, кодируются одним байтом, национальные символы большинства языков — двумя байтами, зато менее распространенные символы кодируются тремя байтами.

Выделение фрагментов текста

Часто возникает необходимость выделения каких-то фрагментов текста курсивом, подчеркиванием, жирным шрифтом или каким-то другим способом. Иногда надо увеличить размер шрифта, изменить его начертание, вставить в текст буквицу, формулу, какое-то изображение. У байтовых кодировок и у Unicode нет таких возможностей, ведь для курсива или подчеркнутых символов нужно задавать дополнительные коды. Для выделения фрагментов текста надо вводить новые средства. Развитие текстовых редакторов пошло двумя путями.

Первый путь — применение графических текстовых редакторов, таких как MS Word. Они используют не кодировку символа, а описание его графического изображения. Полученное описание записывается в файл специального формата, прочитать который может только такой же текстовый редактор.

Второй путь — внести в текст пометки, указывающие на особенности того или иного фрагмента текста. Этот способ давно применяется в полиграфии. При подготовке документа к печати его рукописный вариант испещряется корректурными знаками и цветными пометками, указывающими наборщику, какой шрифт выбрать при наборе текста. Именно по этому пути пошла Web-технология, успешно воплотив его в гипертекстовом языке разметок HTML (Hypertext Markup Language). В этом языке пометки, называемые *тегами* (tags), записываются в угловых скобках, чтобы отличить их от символов самого текста. Некоторые теги языка HTML показаны в листинге В1.

Листинг В1. Текст с тегами языка HTML

```
<html>
<body>
<h2 align=center>Заголовок</h2>
Обычный текст, <i>курсив</i>, снова обычный текст.<p> Теперь
<u>подчеркнутый текст</u>. <p>Текст <font size=+3> увеличенного
размера. </font>
</body>
</html>
```

В листинге В1 тег `<h2>` (header) показывает, что дальше идет заголовок второго уровня. Параметр `align=center` тега `<h2>` предписывает размещение заголовка посередине окна браузера. Тег `<i>` (italic) означает, что дальше пойдет курсив, а тег `<u>` (underline) — что дальше надо выводить текст с подчеркиванием. Тег `<p>` (paragraph) начинает новый абзац. Тег `` изменяет шрифт текста, в частности параметр `size=+3` увеличивает размер шрифта на три единицы.

Как видите, многим тегам языка HTML соответствуют закрывающие теги, отмеченные наклонной чертой. Они отменяют действие открывающих тегов, возвращая текст к первоначальному виду. Сам текст записывается между тегами `<body>` и `</body>`, а весь документ — между тегами `<html>` и `</html>`.

Текст с тегами языка HTML "понимают" все браузеры. На рис. В1 показано, как выглядит текст листинга В1 в окне браузера Internet Explorer.

Язык разметок HTML прост и нагляден, но у него есть два существенных ограничения. Во-первых, набор тегов HTML строго фиксирован, его нельзя расширить или изменить. Все браузеры должны таким образом интерпретировать одинаковые теги, чтобы пользователь, написавший текст с разметками HTML, был уверен, что этот текст будет одинаково выглядеть во всех браузерах. Во-вторых, теги языка HTML показывают только визуальную разметку, внешний вид документа, но ничего не говорят о его структуре.

Например, заголовки следует помечать одинаковым тегом, например тегом `<h2>`, даже если все они одного уровня.

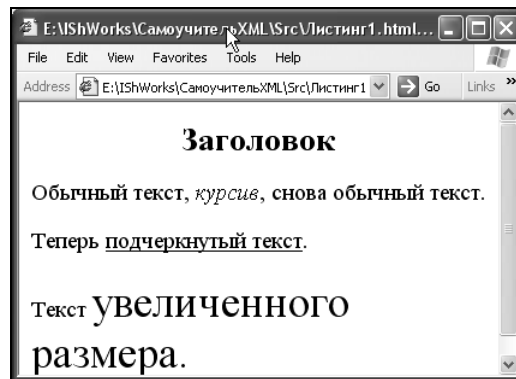


Рис. В1. Текст HTML в окне браузера

Простота языка HTML перевесила все его недостатки. Она привела к взрывному росту числа Web-сайтов и авторов многочисленных Web-страничек. Обычные пользователи компьютеров ощутили себя творцами, получили возможность заявить о себе, высказать свои мысли и чувства, найти в Интернете своих единомышленников.

Ограниченные возможности языка HTML быстро перестали удовлетворять поднаторевших разработчиков, почувствовавших себя профессионалами. Введение таблиц стилей CSS (Cascading Style Sheet) и включений на стороне сервера SSI (Server Side Includes) лишь ненадолго уменьшило недовольство разработчиков. Профессионалу всегда не хватает средств разработки, он постоянно испытывает потребность добавить к ним какое-то свое средство, позволяющее воплотить все его фантазии.

Стандарт SGML

Такая возможность есть. Еще в 1986 году стал стандартом язык создания языков разметки SGML (Standard Generalized Markup Language), с помощью которого и был создан язык HTML. Основная особенность языка SGML заключается в том, что он позволяет создать новый язык разметок, определив набор тегов создаваемого языка. Каждый конкретный набор тегов, разработанный по правилам SGML, снабжается описанием DTD (Document Type Definition) — *определением типа документа*, разъясняющим связь тегов между собой и правила их применения. Специальная программа — драйвер принтера или SGML-браузер — руководствуется этим описанием для печати или отображения документа на экране дисплея.

В это же время выявилась еще одна, самая важная область применения языков разметки — поиск и выборка информации. В настоящее время подавляющее большинство информации хранится в реляционных базах данных. Они удобны для хранения и поиска информации, представляемой в виде таблиц: анкет, ведомостей, списков и т. п., — но неудобны для хранения различных документов, планов, отчетов, статей, книг, которые не могут быть представлены в виде таблиц. Между тем, человечество накопило несметное количество таких документов. Большинство из них уже переведено в электронную форму, как правило, в байтовых кодировках. Очень часто ставится задача выбрать какие-то определенные сведения из таких документов: адреса, фамилии, даты, решения, резюме.

Тегами языка разметки можно задать структурную, а не визуальную разметку документа, разбить документ на главы, параграфы и абзацы или на какие-то другие элементы, выделить важные для поиска участки документа. Например, можно определить тег `<address>` и помечать им все адреса, встречающиеся в тексте. Легко написать программу, анализирующую размеченный такими тегами документ и извлекающую из него нужную информацию. В нашем примере анализирующая программа просто извлечет весь текст, заключенный между тегами `<address>` и `</address>`.

Язык SGML оказался слишком сложным, требующим тщательного и объемистого описания элементов создаваемого с его помощью языка. Только одна его спецификация содержит более пятисот страниц. Он применяется лишь в крупных проектах, например, для создания единой системы документооборота крупной фирмы. Скажем, map-страницы операционной системы Solaris Operational Environment написаны на специально сделанной реализации языка SGML.

Язык разметок XML

Золотой серединой между языками SGML и HTML стал язык XML (eXtensible Markup Language) — расширяемый язык разметок. Это подмножество языка SGML, избавленное от излишней сложности, но позволяющее разработчику Web-страниц создавать свои собственные теги. Язык XML достаточно широк, чтобы можно было создать все нужные теги, и достаточно прост, чтобы можно было быстро их описать.

Разработка XML началась в 1996 году. Ею занимается общественная организация W3C (World Wide Web Consortium), основной сайт которой находится по адресу <http://www.w3c.org/>. Сведения об XML собраны на странице <http://www.w3c.org/XML/>. В 1998 году консорциум выпустил спецификацию XML версии 1.0. Она постоянно совершенствуется, последний вариант спецификации всегда находится по адресу <http://www.w3c.org/TR/rec-xml>.

Появление новой версии Unicode 3.0 вызвало необходимость перевода на нее XML. Консорциум W3C начал разработку второй версии XML 1.1. Во

время написания этих строк она находилась в начальной стадии, появилась только рекомендация. Ее текущее состояние можно посмотреть по адресу <http://www.w3c.org/TR/xml11>. Уже создана версия Unicode 4.0, поэтому, видимо, выпуск XML 1.1 опять будет отложен.

Правила написания документа XML не сложны. Мы подробно рассмотрим их в *главе 1*. Но с каждой разновидностью документов XML надо связать ее описание DTD (Document Type Definition) или какое-то другое описание структуры и способа разметки документов. Такое описание часто бывает сложнее самого документа. Оно выполняется на специальном языке описаний, значит, надо изучить и этот язык. Мы посвятим тщательному разбору таких языков описаний *главы 2* и *3*.

Одного описания документа недостаточно для успешной работы с ним. Нужны еще средства поиска информации в документе, перекрестные ссылки одних частей документа на другие. Этому посвящены следующие главы книги.

Но и этого мало для работы с документами. Необходимо уметь быстро извлекать из документа нужную информацию, уметь преобразовывать документ и представлять его в различных видах. Предпоследние главы книги подробно объясняют способы решения этих задач, предложенные технологией XML.

Наконец, в *главах 10* и *11* раскрывается "кухня" технологии XML, показываются методы работы с документами XML, объясняется, как создаются программы-обработчики этих документов.

Структура книги

Книга состоит из одиннадцати глав, разбитых на две части.

В *части 1* рассматриваются конструкции языка XML и его реализаций, непосредственно связанных с технологией XML.

Глава 1 описывает конструкцию самого документа XML. Из нее вы узнаете, из чего состоит документ XML, как правильно записать элементы XML и научитесь корректно составлять хорошо оформленные документы XML.

К каждому хорошо оформленному документу XML должно быть приложено его описание. Его можно сделать по-разному. Из *главы 2* вы узнаете, как делать описание документа на языке DTD (Document Type Definition). Этот язык, пришедший в XML из стандарта SGML, оказался не совсем подходящим для описания документов XML. Он недостаточно подробен, у него не хватает средств для полного описания всех конструкций XML и сейчас он постепенно заменяется другими языками. Тем не менее его надо знать, потому что уже тысячи документов снабжены такими описаниями.

В *главе 3* описан наиболее мощный язык описания документов XML — язык XSD (XML Schema Definition Language). Этот язык сам является реализацией XML. Описание документа, сделанное на нем, само будет документом XML. Эта особенность языка XSD позволяет автоматически обрабатывать описания как обычный документ XML. Прочитав *главу 3*, вы научитесь точно и правильно описывать любой документ на языке XSD.

Успех языка HTML во многом определен возможностью легко создавать гиперссылки на другие документы или на иные точки того же самого документа. Такая возможность есть и в технологии XML. Ее предоставляет язык XLink (XML Linking Language). Возможности языка XLink гораздо шире, чем возможности тега <a> языка HTML. Вы изучите их, прочитав *главу 4*.

Иногда надо сделать ссылку не на определенный элемент документа, а на что-нибудь вроде "третьего абзаца пятого параграфа четвертого раздела договора". Это легко сделать на языке XPointer, также входящем в технологию XML. Изучению этого языка и способам его практического применения посвящена *глава 5*.

Глава 6 также познакомит вас со ссылками, но не на одно конкретное место документа, а, например, на некоторое множество элементов или на определенное слово, где бы оно ни находилось. Вы не удивитесь, узнав, что такие ссылки делаются на специальном языке адресации XPath. Это очень мощный язык, используемый во многих других языках, входящих в технологию XML. Область его применения все время расширяется и уже сейчас его просто необходимо знать для понимания особенностей большого числа конструкций XML.

Всем известно, какую большую роль играет язык SQL в работе с базами данных. Поскольку сейчас XML все чаще применяется для хранения документов в базах данных, нужны какие-то средства для быстрого и точного извлечения данных самого разного вида из документов XML. Такие средства предоставляет новый язык XQuery, только что прошедший стадию предварительной разработки. Его подробному изложению посвящена *глава 7*.

На этом первая часть книги, посвященная описанию базовых средств технологии XML, заканчивается. Прочитав ее, проделав упражнения и ответив на вопросы, вы можете быть уверены, что понимаете суть технологии XML и способны написать хорошо оформленные документы XML, содержащие все средства, облегчающие работу с ними.

Часть II книги описывает средства обработки готовых документов XML.

В *главе 8* подробно рассмотрены конструкции языка преобразования документов XML, называемого XSLT (eXtensible Stylesheet Language for Transformations). Этот язык позволяет на базе одного документа XML автоматически построить другой, может быть, имеющий совершенно иную структу-

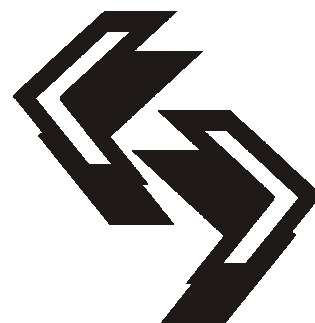
ру, или выделить какие-то части документа и составить на их основе новый документ или даже несколько документов. Если вам надо собирать и размножать сведения, содержащиеся в документах XML, то язык XSLT поможет вам в этом.

Каждый документ XML, в конечном счете, должен быть распечатан или выведен на экран дисплея, сотового телефона, пейджера или даже на какое-то голосовое устройство. Для этого он должен быть предварительно отформатирован в расчете на конкретное представление в устройстве вывода. Такое представление можно сделать на языке XSL-FO (eXtensible Stylesheet Language Formatting Objects), описанию которого посвящена *глава 9*.

Две последние главы книги — *10* и *11* — предназначены для любителей программирования, которые хотят понять алгоритмы обработки документов XML. В них изложены методы разбора, анализа и обработки документов XML. В *главе 10* рассматривается обработка документов, основанная на событиях, а в *главе 11* — обработка, основанная на построении дерева документа в оперативной памяти. Освоив материал этих глав, вы сможете написать свои собственные программы-обработчики документов XML.

Технология XML развивается очень быстро и бурно. В настоящее время она еще сравнительно проста и обозрима в пределах одной небольшой книги. Через несколько лет XML обрстет сложными и громоздкими конструкциями, овладеть которыми будет нелегко. Чем раньше вы приступите к изучению конструкций XML, тем легче вам будет следить за развитием данной технологии и быть в курсе всех событий, связанных с ней. Давайте начнем!

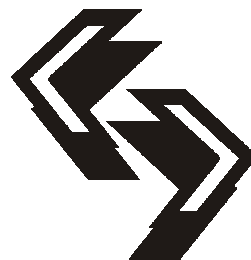
ЧАСТЬ I



Конструкции языка XML и его реализаций

- Глава 1. Структура документа XML
- Глава 2. Описание структуры документа средствами DTD
- Глава 3. Описание схемы документа на языке XSD
- Глава 4. Создание ссылок на языке XLink
- Глава 5. Уточненные ссылки XPointer
- Глава 6. Адресация на языке XPath
- Глава 7. Язык запросов XQuery

ГЛАВА 1



Структура документа XML

Вы, наверное, уже поняли из введения, что язык XML не содержит готового набора тегов, подобно HTML, а описывает правила создания своих собственных тегов. Эти теги будут размечать ваши документы так, как это нужно. После создания тегов вы получаете одну из *реализаций* языка XML и используете ее для разметки документов. Поэтому точнее говорить не "документ XML", а "документ, размеченный тегами, созданными по правилам XML". Мы не будем употреблять такую длинную фразу, но полезно иметь ее в виду. В тех случаях, когда надо будет подчеркнуть, что речь идет о конкретном наборе тегов, мы будем говорить о "реализации XML".

Уже создано много общепринятых реализаций XML. Наиболее известны такие языки:

- XHTML — язык гипертекстовой разметки HTML, приведенный в соответствие с правилами XML;
- MathML — язык записи математических формул;
- CML — язык записи химических формул;
- VoxML — язык записи звуков;
- WML — язык, применяемый в беспроводной технологии.

Есть еще множество других языков, их число растет с каждым днем.

Сам язык XML задает только общие правила, по которым создаются теги и оформляется документ XML. Эти правила изложены в спецификации XML, расположенной по адресу <http://www.w3.org/TR/REC-xml>. Документ XML, написанный с учетом всех правил спецификации, называется *хорошо оформленным* (well-formed) документом. Хорошо оформленный документ будет гарантированно читаться и анализироваться любым XML-редактором, хотя результат анализа может оказаться совершенно неожиданным. Для того что-

бы XML-редактор правильно проанализировал прочитанный им документ и "понял" смысл тегов, необходимо, чтобы документ был не только хорошо оформленным, но и *верным* (valid).

Правила оформления документа XML, изложенные в спецификации XML, не очень обременительны. В этой главе мы их подробно изучим. По спецификации XML документ начинается с необязательного пролога.

Пролог документа XML

Пролог (prolog), начинающий документ XML, состоит из двух частей.

В первой части пролога, занимающей одну строку, записывается *объявление XML* (XML Declaration). Оно заключено между символами `<?...?>`, содержит пометку `xml` и номер версии спецификации XML. Все это вместе выглядит так:

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

Кроме этого, объявление XML может содержать указание на кодировку символов, в которой написан документ. По умолчанию считается, что документ записан в кодировке UTF-8. Написанное выше объявление XML эквивалентно такому:

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

Большинство текстовых редакторов в России работает в кодировке CP1251, поэтому объявление XML часто выглядит так:

```
<?xml version="1.0" encoding="windows-1251"?>
```

В объявление можно поместить еще параметр `standalone` со значениями `"yes"` или `"no"`. Значение `"no"` показывает, что документ использует определения элементов, сделанные в другом, внешнем, документе. По умолчанию принимается значение `"yes"`. Итак, полное объявление XML может выглядеть следующим образом:

```
<?xml version="1.0" encoding="windows-1251" standalone="yes"?>
```

Вторая часть пролога — *объявление типа документа*, DTD (Document Type Declaration) — может занимать одну или несколько строк. В этой части объявляются теги, использованные в документе, или приводится ссылка на файл, в котором записаны такие объявления. Объявление типа документа начинается с символов `<!DOCTYPE`, а заканчивается угловой скобкой — знаком "больше" `>`. Его содержимое зависит от способа объявления тегов. Способам объявления типа документа посвящены следующие две главы, а сейчас перейдем к примерам.

Пример: разметка адресной книжки

Создавая описание документа на языке XML, надо, прежде всего, продумать структуру документа. Пусть мы решили, наконец, упорядочить свою записную книжку с адресами и телефонами. В ней записаны фамилии, имена и отчества родственников, сослуживцев и знакомых, их дни рождения, адреса, состоящие из почтового индекса, названия города, улицы, номера дома и квартиры, а также телефоны, если они есть: рабочие и домашние. Мы определим теги для выделения каждого из этих элементов, продумаем связь между тегами и получим структуру, показанную в листинге 1.1.

Листинг 1.1. Пример XML-документа

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook SYSTEM "ntb.dtd">

<notebook>

  <person>

    <name>
      <first-name>Иван</first-name>
      <second-name>Петрович</second-name>
      <surname>Сидоров</surname>
    </name>

    <birthday>25.03.1977</birthday>

    <address>
      <street>Садовая, 23-15</street>
      <city>Урюпинск</city>
      <zip>123456</zip>
    </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
    </phone-list>

  </person>

  <person>

    <name>
      <first-name>Мария</first-name>
```

```
<second-name>Петровна</second-name>
<surname>Сидорова</surname>
</name>

<birthday>17.05.1969</birthday>

<address>
  <street>Ягодная, 17</street>
  <city>Жмеринка</city>
  <zip>234561</zip>
</address>

<phone-list>
  <home-phone>2334455</home-phone>
</phone-list>

</person>

</notebook>
```

Разумеется, можно назвать теги по-другому, создать другой набор тегов и расположить их иначе. Например, выделить номер дома из адреса в отдельный элемент. Дата рождения может быть разбита на день, месяц и год рождения. Все это определяется теми целями, ради которых документ разбивается тегами на отдельные элементы. Если мы захотим в начале месяца выбрать имена всех своих знакомых, родившихся в этом месяце, то лучше выделить его в отдельный элемент. Это облегчит работу программе поиска.

Упражнения

1. Продумайте еще раз структуру записной книжки. Стоит ли разбивать адрес на более мелкие составляющие? Может быть, следует объединить имя, отчество и фамилию в один элемент? Нужно ли вводить в книжку дополнительные сведения? Как это сделать?
2. У вас большая библиотека. Продумайте структуру ее библиографического описания и создайте теги для оформления этой структуры.
3. Как бы вы записали структуру этой книги?

Корневой элемент

У хорошо оформленного документа все, что находится после пролога, обязательно должно содержаться в *корневом элементе* (root element). В листинге 1.1 это элемент, начинающийся с тега `<notebook>` и заканчивающийся тегом `</notebook>`. Такое требование возникло потому, что один документ

XML можно вложить в другой. При этом корневой элемент вложенного документа станет просто одним из элементов документа, в который он вложен. Такое вложение не нарушит структуру документа.

Имя корневого элемента считается именем всего документа и указывается во второй части пролога — объявлении типа документа (Document Type Declaration) — после слова `DOCTYPE`. Объявление типа документа записано во второй строке листинга 1.1. В ней после слова `DOCTYPE` и имени документа, в квадратных скобках должно идти *определение типа документа* — DTD (Document Type Definition):

```
<!DOCTYPE notebook [ Сюда заносится описание DTD ]>
```

Очень часто определение DTD составляется сразу для нескольких документов XML. В таком случае его удобно записать отдельно от документа. Если определение DTD отделено от документа, то во второй части пролога вместо квадратных скобок записывается одно из слов `SYSTEM` или `PUBLIC`. За словом `SYSTEM` идет адрес в форме URI файла с определением DTD, а за словом `PUBLIC`, кроме того, можно записать дополнительную информацию. Определение DTD дает возможность убедиться в *верности* документа.

Внимание!

Не путайте понятие "объявление типа документа" — DTD (Document Type Declaration) — вторую часть пролога, с понятием "определение типа документа" — DTD (Document Type Definition), которое вставляется во вторую часть пролога `<!DOCTYPE>`. К сожалению, оба понятия обозначаются аббревиатурой DTD.

Верный документ

Для описания адресной книжки в листинге 1.1 нам понадобились открывающие теги `<notebook>`, `<person>`, `<name>`, `<address>`, `<street>`, `<city>`, `<zip>`, `<phone-list>`, `<work-phone>`, `<home-phone>` и соответствующие им закрывающие теги, помеченные наклонной чертой. Теперь необходимо дать им объявление. В объявлении указываются только самые общие признаки логической взаимосвязи элементов и их тип. Документ, выдерживающий такую взаимосвязь, будет *верным* документом. Программа, анализирующая документ, сможет проверить правильность его разметки, сверив ее с объявлениями тегов.

Вот как может выглядеть объявление тегов листинга 1.1:

- элемент `<notebook>` может содержать в себе только нуль или больше элементов `<person>` и больше ничего;

- ❑ элемент `<person>` содержит ровно один элемент `<name>`, нуль или несколько элементов `<address>`, а также нуль или один элемент `<phone-list>`;
- ❑ элемент `<name>` имеет не более чем по одному элементу `<first>`, `<second>` и `<surname>`, содержимое которых — строки символов;
- ❑ элемент `<address>` содержит по одному элементу `<street>`, `<city>` и `<zip>`;
- ❑ элементы `<street>` и `<city>` содержат по одной текстовой строке;
- ❑ элемент `<zip>` содержит одно целое число;
- ❑ необязательный элемент `<phone-list>` содержит нуль или более элементов `<work-phone>` и `<home-phone>`;
- ❑ элементы `<work-phone>` и `<home-phone>` содержат по одной строке, состоящей только из цифр.

Это словесное описание, называемое *схемой* документа XML, формализуется несколькими способами. Наиболее распространены два способа: можно сделать определение DTD (Document Type Definition), пришедшее в XML из SGML, или описать схему на языке XSD (XML Schema Definition Language). Этим описаниям посвящены следующие две главы, а пока ограничимся словесным описанием.

Прочитав формализованное описание и узнав из него схему документа, программа-анализатор может проверить соответствие каждого документа его схеме и сделать вывод, верен этот документ или нет.

Элементы документа XML

Документ XML состоит из *элементов* (elements). Элемент начинается *открывающим тегом* (start-tag) в угловых скобках, затем идет необязательное *содержимое* (content) элемента, после него записывается *закрывающий тег* (end-tag) в угловых скобках. Например:

```
<surname>Сидорова</surname>
```

Закрывающий тег содержит наклонную черту, после которой повторяется имя открывающего тега.

Язык XML, в отличие от языка HTML, требует обязательно записывать закрывающие теги. Правда, из этого правила есть одно исключение. Если в содержимом элемента нет ни одного символа, даже пробела, то закрывающий тег можно не записывать. В этом случае открывающий тег должен заканчиваться символами `</>`, например, в языке XHTML перевод строки записывается тегом

```
<br />
```

Эта запись равносильна записи

```
<br></br>
```

Элемент без всякого содержимого называется *пустым* (empty) элементом.

В записи `
` перед наклонной чертой оставлен пробел. Это не обязательно, но многие браузеры пропускают пустой элемент без такого пробела, искажая вид документа.

Сразу надо сказать, что язык XML, в отличие от HTML, различает регистры букв. Теги `<surname>`, `<Surname>` `<SURNAME>` — совершенно различные теги, не имеющие друг к другу никакого отношения.

Внимание!

Все реализации языка XML различают регистры букв!

Из листинга 1.1 видно, что содержимым элемента может быть другой элемент или несколько элементов, т. е. элементы документа XML могут быть вложены друг в друга. Надо следить за тем, чтобы элементы не пересекались, а полностью вкладывались друг в друга. Как уже говорилось выше, все элементы, составляющие хорошо оформленный документ, вложены в корневой элемент этого документа. Тем самым, хорошо оформленный документ наделяется структурой дерева, ветви которого состоят из вложенных элементов. На рис. 1.1 показана структура адресной книжки, описанной в листинге 1.1.

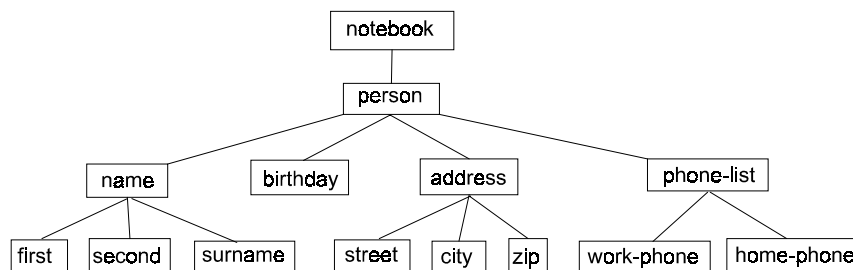


Рис. 1.1. Дерево элементов документа XML

Что делать, если в содержимом элемента есть знаки "больше" или "меньше"? Ведь их появление будет понято как начало или конец вложенного тега. В таком случае знак "меньше" заменяют строкой `<`, а знак "больше" — строкой `>`. "Увидев" символ амперсанда, программа-анализатор XML "поймет", что дальше идет условное обозначение знака "меньше" или "больше", завершающееся точкой с запятой. Но что делать, если в содержимом элемента надо вставить символ амперсанда? Правило таково: в содержимом элемента амперсанд заменяется строкой `&`.

Например, следующий фрагмент программного кода

```
if (x < MAX_VALUE && x > MIN_VALUE) proc1();
```

можно записать в документе XML так:

```
<progcode>
  if (x &lt; MAX_VALUE &amp;&amp; x &gt; MIN_VALUE) proc1();
</progcode>
```

Такие наборы символов необычного вида представляют собой примеры ссылок на сущности. Речь о них пойдет в следующем разделе, а пока надо заметить, что вместо символов "меньше", "больше" и амперсанда можно использовать их числовые коды. Они тоже записываются обозначениями специального вида — `<`, `>` и `&`; соответственно, — чтобы анализатор не принял их за обыкновенные числа. Коды можно записать в шестнадцатеричной форме, добавив букву "икс" — `<`, `>`, `&`. Обратите внимание на то, что коды завершаются точкой с запятой. С использованием приведенных выше обозначений предыдущий фрагмент выглядит так:

```
<progcode>
  if (x &#60; MAX_VALUE &#38;&#38; x &#62; MIN_VALUE) proc1();
</progcode>
```

Ссылки на сущности

Ссылки на сущности (entity references), начинающиеся с амперсанда и заканчивающиеся точкой с запятой, — это указание программе-анализатору подставить вместо них заранее определенную строку символов — *сущность*. Например, я могу записать ссылку на сущность `&author;`. Программный анализатор подставит вместо нее фамилию автора. Почему? Как анализатор узнает, что именно надо подставить вместо ссылки? Ответ прост: сущности задаются в определении типа документа (Document Type Definition), которое мы рассмотрим в следующей главе.

Если текст содержит много знаков "больше", "меньше" и амперсандов, например, требуется весь вложенный элемент понимать как простую строку символов, то удобнее организовать так называемую секцию CDATA.

Секция CDATA

Секция CDATA (character data) начинается со строки `<![CDATA[`. Далее записывается все то, что следует считать не разметкой, а простой строкой символов. Секция завершается двумя закрывающими квадратными скобками и знаком "больше". Например:

```
<![CDATA[<surname>Сидорова</surname>]]>
```

Все знаки "больше", "меньше" и амперсанды, расположенные в секции CDATA, понимаются как обычные символы, не определяющие никакой разметки.

Секции CDATA можно создать в содержимом любого элемента XML. Внутри секции CDATA нельзя создать новую секцию, секции CDATA нельзя вкладывать друг в друга.

Приведем еще раз пример, содержащий фрагмент программного кода:

```
<progcode>
  <![CDATA[if (x < MAX_VALUE && x > MIN_VALUE) proc1(x);]]>
</progcode>
```

Программа-анализатор не разбивает секцию CDATA на элементы, а считает ее просто набором символов, которые надо передать без изменений на выход программы.

Комментарии

Если надо сделать какой-то фрагмент документа XML вообще "невидимым" для программы-анализатора, то его можно оформить как комментарий, записав перед ним символы <!--, а после него — символы --> с двумя дефисами подряд. Например:

```
<!-- Это комментарий -->
```

Программа-анализатор пропустит всю эту конструкцию, даже не "заглянув" в нее.

Такой синтаксис комментария накладывает на него два ограничения:

- ☐ в комментарии нельзя записывать два дефиса подряд;
- ☐ комментарий нельзя завершить дефисом.

Атрибуты

У открывающих тегов XML могут быть *атрибуты* (attributes), состоящие из имени и значения, связанных знаком равенства. Например, имя, отчество и фамилию можно записать как атрибуты first, second и surname тега <name>:

```
<name first="Иван" second="Петрович" surname="Сидоров" />
```

В отличие от языка HTML, в языке XML значения атрибутов обязательно надо заключать в кавычки или в апострофы. Если значение атрибута содержит в себе кавычки, то его нужно заключить в апострофы, если значение содержит апострофы, то его заключают в кавычки. Если же кавычки и апострофы уже использованы для ограничения значений атрибутов, то внутри

значения кавычку можно заменить ссылкой на сущность `"`, а апостроф — ссылкой `'`. Вместо ссылок на сущности можно записать коды кавычки и апострофа — `"` и `'`. В шестнадцатеричной форме коды кавычки и апострофа выглядят так: `"` и `'`.

Во время обдумывания разметки документа и создания тегов всегда возникает вопрос о том, куда поместить ту или иную информацию: в содержимое элемента или в его атрибуты? Ответ неоднозначен, он зависит и от самой информации, и от целей ее разметки, а часто это просто дело вкуса.

Атрибуты удобны для описания простых значений. Практически у каждого гражданина России обязательно есть одно имя, одно отчество и одна фамилия. Их удобно записывать атрибутами. Но у гражданина России может быть несколько телефонов, поэтому их номера удобнее оформить как элементы `<work-phone>` и `<home-phone>`, вложенные в элемент `<phone-list>`, а не атрибуты открывающего тега `<phone-list>`. Заметьте, что элемент `<name>` с атрибутами пустой, у него нет содержимого, следовательно, не нужен закрывающий тег. Поэтому тег `<name>` с атрибутами завершается символами `</>`. В листинге 1.2 приведена измененная адресная книжка.

Листинг 1.2. Пример XML-документа с атрибутами в открывающем теге

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook SYSTEM "ntb.dtd">

<notebook>

  <person>

    <name first="Иван" second="Петрович" surname="Сидоров" />

    <birthday>25.03.1977</birthday>

    <address>
      <street>Садовая, 23-15</street>
      <city>Урюпинск</city>
      <zip>123456</zip>
    </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
    </phone-list>

  </person>
```



```
<person>

  <name first="Мария" second="Петровна" surname="Сидорова" />

  <birthday>17.05.1969</birthday>

  <address>
    <street>Ягодная, 17</street>
    <city>Жмеринка</city>
    <zip>234561</zip>
  </address>

  <phone-list>
    <home-phone>2334455</home-phone>
  </phone-list>

</person>

</notebook>
```

Атрибуты открывающего тега удобны и для указания типа элемента. Например, мы не уточняем, в городе живет наш родственник, в поселке или в деревне. Можно ввести в открывающий тег `<city>` атрибут `type`, принимающий одно из значений `город`, `поселок`, `деревня`. Например:

```
<city type="город">Москва</city>
```

Как видите, значения атрибутов можно записывать не только латинскими, но и русскими буквами. Спецификация XML допускает для записи значений атрибутов и содержимого элементов практически все символы Unicode.

Как уже говорилось ранее, вместо любого символа можно ввести его код в десятичной форме, предварив запись символами `&#` и закончив точкой с запятой, например, символ "меньше" можно записать строкой `<`. Можно сделать шестнадцатеричную запись кода символа, предварив ее строкой `&#x` и закончив точкой с запятой. Код того же символа "меньше" в шестнадцатеричной форме записывается строкой `<`. Амперсанд можно записать как `&` или `&`. Такая форма записи удобна для символов, которых нет на клавиатуре.

При записи имен элементов и атрибутов правила гораздо строже.

Имена

Правила записи имен довольно просты и заключаются в следующем:

- ❑ имена элементов и атрибутов могут содержать только буквы, входящие в секцию букв кодировки Unicode, арабские цифры, дефисы, знаки подчеркивания, точки и двоеточия;

- ❑ имя может начинаться с буквы, знака подчеркивания или двоеточия;
- ❑ имя не может начинаться со строки `xml` в любом регистре, поскольку такие имена зарезервированы для самой спецификации XML.

С введением понятия пространства имен, о котором речь пойдет в следующем разделе главы, двоеточие получило особое значение. Теперь оно используется только для отделения префикса от локального имени.

Заметьте, что в имени не может быть пробелов. Запись

```
<my cool element>
```

неверна!

Примечание

Напомню еще раз, что в именах различается регистр букв.

Каждое имя должно быть уникально в пределах документа. Это правило не сложно выполнить, но в документ XML можно включить иной документ, даже относящийся к другой реализации XML. Во включенном документе не должны встречаться такие же имена, что и во включающем, поскольку у совпадающих имен может быть совершенно разный смысл. Для решения проблемы совпадающих имен введено понятие пространства имен.

Пространства имен XML

Поскольку в разных языках разметок — реализациях XML — могут встретиться одни и те же имена тегов и их атрибутов, имеющие совершенно разный смысл, надо иметь возможность их как-то различать. Для этого имена тегов и атрибутов снабжают кратким префиксом, который отделяется от имени двоеточием. Префикс имени связывается с идентификатором, определяющим *пространство имен* (namespace). Все имена тегов и атрибутов, префиксы которых связаны с одним и тем же идентификатором, образуют одно пространство имен, в котором имена должны быть уникальны. Префикс и идентификатор пространства имен определяются атрибутом `xmlns` следующим образом:

```
<ntb:notebook xmlns:ntb = "http://some.firm.com/2003/ntbml">
```

Как видите, префикс `ntb` только что определен, но его уже можно использовать в имени `ntb:notebook`. В дальнейшем имена тегов и атрибутов, которые мы хотим отнести к пространству имен `http://some.firm.com/2003/ntbml`, снабжаются префиксом `ntb`, например:

```
<ntb:city ntb:type="поселок">Горелово</ntb:city>
```

Имя вместе с префиксом, например `ntb:notebook` или `ntb:city`, называется *расширенным* или *уточненным именем* (QName, Qualified Name). Часть име-

ни, записанная после двоеточия, называется *локальной частью* (local part) имени.

Идентификатор пространства имен должен иметь форму URI. Адрес URI, такой, например, как `http://some.firm.com/2003/ntbml`, не имеет никакого значения и может не соответствовать никакому действительному адресу Интернета. Программы, использующие документ, не будут обращаться по этому адресу, поскольку там нет никакой Web-странички. Просто идентификатор пространства имен должен быть уникальным во всем Интернете, и разработчики рекомендации по применению пространства имен "Namespaces in XML", которую можно посмотреть по адресу <http://www.w3.org/TR/REC-xml-names>, справедливо решили, что будет удобно использовать для него DNS-имя сайта, на котором размещено определение пространства имен. Смотрите на URI просто как на уникальную строку символов, идентифицирующую пространство имен. Обычно указывается URL фирмы, создавшей данную реализацию XML, или имя файла с описанием схемы XML.

Примечание

В этой книге адреса Интернета выделяются полужирным шрифтом, а идентификаторы пространств имен никак не выделяются.

По правилам SGML и XML, двоеточие может применяться в именах как обычный символ, поэтому имя с префиксом — это просто фокус, всякая программа, "не знающая" пространства имен, анализируя документ, рассматривает уточненное имя как обычное имя. Отсюда следует, в частности, что в объявлении типа документа (Document Type Declaration) нельзя опускать префиксы имен.

Атрибут `xmlns` может появиться в любом элементе XML, а не только в корневом. Определенный им префикс можно применять в том элементе, в котором записан атрибут `xmlns`, и во всех вложенных в него элементах. Более того, в одном элементе можно определить несколько пространств имен:

```
<ntb:notebook
  xmlns:ntb = "http://some.firm.com/2003/ntbml"
  xmlns:bk  = "http://some.firm.com/2003/bookxml">
```

Во вложенных элементах пространство имен можно переопределить, связав префикс с другим идентификатором.

Появление имени тега без префикса в документе, использующем пространство имен, означает, что имя принадлежит *пространству имен по умолчанию* (default namespace). Например, язык XHTML допускает применение тегов HTML и тегов XML в одном документе. Допустим, мы определили тег с именем `title`. Чтобы не принять его за один из тегов HTML, поступаем следующим образом:

```
<html xmlns = "http://www.w3.org/1999/xhtml"
      xmlns:ntb = "http://some.firm.com/2003/ntbml">

  <head>
    <title>Моя библиотека</title>
  </head>

  <body>
    <ntb:book>
      <ntb:title>Самоучитель XML</ntb:title>
    </ntb:book>
  </body>

</html>
```

В этом примере пространством имен по умолчанию становится пространство имен XHTML, имеющее общеизвестный идентификатор `http://www.w3.org/1999/xhtml`, и теги, относящиеся к этому пространству имен, записываются без префикса.

Атрибуты никогда не входят в пространство имен по умолчанию. Если имя атрибута записано без префикса, то это означает, что атрибут не относится ни к одному пространству имен. Программы, анализирующие документ, не будут искать такое имя ни в одном пространстве имен.

Обратимся еще раз к нашей адресной книжке и перепишем листинг 1.2 с использованием пространства имен (листинг 1.3).

Листинг 1.3. Пример XML-документа с пространством имен

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook SYSTEM "ntb.dtd">

<ntb:notebook xmlns:ntb = "http://some.firm.com/2003/ntbml">

  <ntb:person>

    <ntb:name ntb:first="Иван" ntb:second="Петрович"
              ntb:surname="Сидоров" />

    <ntb:birthday>25.03.1977</ntb:birthday>

    <ntb:address>
      <ntb:street>Садовая, 23-15</ntb:street>
      <ntb:city>Урюпинск</ntb:city>
      <ntb:zip>123456</ntb:zip>
    </ntb:address>
```

```
<ntb:phone-list>
  <ntb:work-phone>2654321</ntb:work-phone>
  <ntb:work-phone>2654023</ntb:work-phone>
  <ntb:home-phone>3456781</ntb:home-phone>
</ntb:phone-list>

</ntb:person>

<ntb:person>

  <ntb:name ntb:first="Мария" ntb:second="Петровна"
    ntb:surname="Сидорова" />

  <ntb:birthday>17.05.1969</ntb:birthday>

  <ntb:address>
    <ntb:street>Ягодная, 17</ntb:street>
    <ntb:city>Жмеринка</ntb:city>
    <ntb:zip>234561</ntb:zip>
  </ntb:address>

  <ntb:phone-list>
    <ntb:home-phone>2334455</ntb:home-phone>
  </ntb:phone-list>

</ntb:person>

</ntb:notebook>
```

Хорошо оформленный документ должен использовать пространства имен для всех своих элементов.

Префиксы, начинающиеся с символов `xml` с любым регистром букв, зарезервированы за самим языком XML. Мы уже видели префикс `xmlns` (XML namespace). Он используется для связи другого, определяемого, префикса, с идентификатором его пространства имен. Префикс `xmlns` не нужно определять, он введен уже упоминавшейся рекомендацией "Namespaces in XML" и связан там с идентификатором пространства имен <http://www.w3.org/2000/xmlns/>.

Еще один префикс, `xml`, связан в той же рекомендации с идентификатором <http://www.w3.org/XML/1998/namespace>. Его тоже не надо определять в документе XML. Никакой другой префикс не может быть связан с этими идентификаторами.

Пока есть только два атрибута с префиксом `xml`. Для каждого элемента верного документа, в котором используются эти атрибуты, они должны быть объявлены в описании DTD (Document Type Definition).

Атрибут `xml:lang` определяет язык, используемый в содержимом элемента. Например, `xml:lang="us"`, `xml:lang="en-US"`, `xml:lang="ru-RU"`. Атрибут можно использовать так:

```
<p xml:lang="ru_RU" >Запись на русском языке</p>
```

Атрибут `xml:space` указывает программе-обработчику, как следует обрабатывать пробельные символы (пробелы, знаки горизонтальной табуляции и символы перевода строки), записанные в содержимом элемента. У атрибута есть всего два значения. Значение `preserve` предписывает сохранять пробельные символы в неприкосновенности. Это важно для некоторых текстов, например, программных кодов. Значение `default` оставляет пробельные символы на усмотрение программы-обработчика.

Упражнения

1. Определите пространство имен по умолчанию для листингов 1.1 и 1.2.
2. Определите пространства имен для элементов, размечающих документы, созданные в упражнениях 2 и 3 к листингу 1.1.

Инструкции по обработке

Еще одна конструкция, которая может присутствовать в документах XML наряду с элементами, — это *инструкции по обработке* (processing instructions). Они, как следует из их названия, содержат указания программе-обработчику документа XML. В отличие от элементов, инструкции по обработке заключаются между символами `<?` и `?>`. Сразу за начальным вопросительным знаком записывается имя программного модуля, которому предназначена инструкция. Затем, через пробел, идет сама инструкция, передаваемая программному модулю. Она записывается произвольно, просто как строка символов, которая, конечно, не должна содержать пару символов `?>`, означающую конец инструкции. Смысл инструкции определяется программным модулем, которому она предназначена.

Примером инструкции по обработке может служить первая строка пролога документа XML — объявление XML

```
<?xml version="1.0" encoding="windows-1251"?>
```

Эта инструкция предназначена программе, обрабатывающей документ XML. Инструкция передает ей номер версии и кодировку, в которой записан документ.

Этот пример объясняет одно ограничение, накладываемое на имя программного модуля — им не может служить слово `xml`, записанное в любом регистре.

Инструкции по обработке могут располагаться в любом месте документа, но целиком в пределах одного элемента.

Информационное множество XML

Итак, документ XML состоит из нескольких конструкций: пролога, элементов со своими атрибутами и содержимым, инструкций по обработке, секций CDATA, ссылок на сущности, комментариев. Для создания программ-анализаторов и обработчиков необходимо точное описание всех возможных составляющих документа XML.

Точные правила записи всех конструкций и вообще всего, что есть в документе XML, описаны в рекомендации "Информационное множество XML" (XML Information Set), выпущенной консорциумом W3C в 2001 году. Ее можно посмотреть по адресу <http://www.w3.org/TR/xml-infoset>.

Информационное множество документа XML, коротко называемое *инфосет* (infoset), — это набор всех *единиц информации* (information items), входящих в документ. В хорошо оформленном документе обязательно есть единица информации документа, другие единицы информации необязательны.

Единица информации документа

Единица информации документа (document information item) — это документ в целом, все его содержимое. Для того чтобы единица информации документа четко выделялась среди остальной информации, в документе следует соблюдать два условия:

- ☐ документ должен быть хорошо оформленным;
- ☐ все элементы документа должны принадлежать какому-либо пространству имен.

Более того, только при выполнении этих условий создается информационное множество документа.

Каждая единица информации обладает некоторыми *свойствами* (properties). Для единицы информации документа это:

- ☐ упорядоченный список вложенных элементов;
- ☐ единица информации элемента;
- ☐ множество объявлений DTD элементов;
- ☐ множество секций CDATA;
- ☐ базовый адрес URI;
- ☐ кодировка символов документа;

- ☐ значения параметров `version` и `standalone` объявления XML;
- ☐ все инструкции по обработке документа.

Кроме единицы информации документа, спецификация "Информационное множество XML" определяет еще десять типов единиц информации.

Единица информации элемента

Единица информации элемента (element information item) — это элемент в целом. Ее свойства составляют:

- ☐ идентификатор пространства имен;
- ☐ префикс и локальная часть имени элемента;
- ☐ упорядоченный список вложенных элементов;
- ☐ атрибуты элемента;
- ☐ атрибуты пространства имен;
- ☐ множество единиц информации пространства имен;
- ☐ базовый адрес URI;
- ☐ объемлющий (родительский) элемент, в который вложен данный элемент.

Единица информации атрибута

Единица информации атрибута (attribute information item) — это каждый отдельный атрибут элемента. К ее свойствам относятся:

- ☐ идентификатор пространства имен;
- ☐ префикс и локальная часть имени атрибута;
- ☐ нормализованное значение атрибута;
- ☐ способ объявления атрибута в схеме документа;
- ☐ тип атрибута;
- ☐ ссылки на атрибут;
- ☐ элемент, которому принадлежит атрибут.

Единица информации инструкции по обработке

Единица информации инструкции по обработке (processing instruction information item) — это каждая инструкция по обработке документа. Ее свойствами являются:

- ☐ программный модуль, выполняющий инструкцию;
- ☐ содержимое инструкции;

- ☐ базовый адрес URI;
- ☐ объявление инструкции в схеме документа;
- ☐ элемент, в котором расположена инструкция.

Единица информации символа

Каждый символ документа составляет отдельную *единицу информации символа* (character information item). Она обладает тремя свойствами:

- ☐ кодировкой символа;
- ☐ признаком, отличающим пробельные символы;
- ☐ элементом, в котором встретился символ.

Единица информации комментария

Каждый комментарий, вставленный в документ, — это отдельная *единица информации комментария* (comment information item). У комментария всего два свойства:

- ☐ содержимое комментария;
- ☐ элемент, снабженный комментарием.

Единица информации пространства имен

Единица информации пространства имен (namespace information item) обладает двумя свойствами:

- ☐ префиксом имен элементов и атрибутов;
- ☐ идентификатором пространства имен.

Единица информации DTD

Схема документа, описанная посредством DTD, — это *единица информации DTD* (document type declaration information item). У нее такие свойства:

- ☐ системный и открытый идентификаторы;
- ☐ упорядоченный список вложенных инструкций по обработке документа;
- ☐ единица информации описываемого документа.

Единица информации объявления DTD

Каждое объявление, сделанное в схеме документа, образует одну *единицу информации объявления* (notation information item).

Ее свойства:

- ☐ имя объявления;
- ☐ системный и открытый идентификаторы;
- ☐ базовый адрес URI объявления.

Единица информации ссылки на сущность

Единица информации ссылки на сущность (unexpanded entity reference information item) обладает следующими свойствами:

- ☐ именем ссылки;
- ☐ системным и открытым идентификаторами;
- ☐ базовым адресом URI;
- ☐ элементом, в котором встретилась ссылка.

Единица информации необрабатываемой секции

Каждая необрабатываемая часть документа, такая как секция CDATA, составляет отдельную *единицу информации необрабатываемой секции* (unparsed entity information item). Ее свойства:

- ☐ имя секции;
- ☐ системный и открытый идентификаторы;
- ☐ базовый адрес URI;
- ☐ объявление секции.

Примеры

Итак, мы ознакомились со структурой документа, размеченного тегами по правилам языка XML. Мы изучили правила этой разметки, рассмотрели все конструкции языка XML, выяснили, какой документ будет хорошо оформленным, а какой — верным. Приведем несколько примеров документов XML, размеченных по этим правилам.

Разметка книги

Вооружившись знаниями, полученными в этой главе, сделаем разметку какой-либо книги.

Предположим, структуру книги составляют титульный лист, оглавление, предисловие, введение, несколько частей, разбитых на главы, заключение, список литературы и предметный указатель. Титульный лист содержит фамилию автора, название книги, издательство, место и время издания. Главы

разбиты на разделы, подразделы, секции. Некоторые части текста, например определения, примечания, рисунки, листинги, также достойны разметки.

Приступая к созданию тегов, определим пространство имен с идентификатором `http://habib.bhv.ru/2003/bookml`. Снабдим теги префиксом `bk`, связанным с этим пространством имен. Набор тегов и элементы полученного документа XML легко понять из листинга 1.4. В листинге 1.4, разумеется, приведен не весь документ.

Листинг 1.4. Разметка книги

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE book SYSTEM "book.dtd">

<bk:book xmlns:bk = "http://habib.bhv.ru/2003/bookml">

  <bk:title-page>

    <bk:authors>
      <bk:author bk:name="Ильдар" bk:surname="Хабибуллин" />
    </bk:authors>

    <bk:book-title>Самоучитель XML</bk:book-title>

    <bk:publ-house>БХВ-Петербург</bk:publ-house>

    <bk:publ-place>Санкт-Петербург</bk:publ-place>

    <bk:publ-date>2003</bk:publ-date>

  </bk:title-page>

  <bk:toc>

    <bk:toc-elem>

      <bk:topic>Предисловие</bk:topic>
      <bk:page>10</page>

    </bk:toc-elem>

    <bk:toc-elem>

      <bk:topic>Введение</bk:topic>
      <bk:page>14</page>

    </bk:toc-elem>

    <!-- и т. д. -->

  </bk:toc>
```

```

<bk:chapter>

  <bk:title bk:level="1">Предисловие</bk:title>

  <bk:section>
    Эта книга посвящена... (далее текст предисловия)
  </bk:section>

</bk:chapter>


<bk:chapter>

  <bk:title bk:level="1">Введение</bk:title>

  <bk:section>
    Как известно,... (далее начало введения)
  </bk:section>

  <bk:listing bk:number="B.1"
    bk:head="Текст с тегами языка HTML">

    <![CDATA[
      <html><body>
        <!--и т. д. -->
      ]]>

  </bk:listing>

  <bk:section>
    В листинге B.1...
  </bk:section>

  <bk:picture bk:number="B.1"
    bk:head="Текст HTML в окне браузера" />

  <bk:section>
    Язык разметок...
  </bk:section>

  <bk:title level="2">Описание структуры документа</bk:title>

</bk:chapter>

<bk:chapter>

  <bk:title bk:level="1">
    Глава 1 Структура документа XML
  </bk:title>

```

```
<bk:section>
    Вы, наверное, ... (далее начало главы)
</bk:section>

<bk:title bk:level="2">
    Пролог документа XML
</bk:title>

<!-- и т. д. и т. п. ... -->

<bk:conclusion>
    В заключение этой главы отметим...
</bk:conclusion>

</bk:chapter>

<!-- описание следующих глав... -->

</bk:book>
```

Конечно, это только один из вариантов определения структуры книги, вы можете предложить другие способы.

Упражнение

Подумайте, как можно описать структуру нумерованных и ненумерованных списков, часто встречающихся в документах, а также вложенных списков.

Таблицы

Документы XML очень часто используются для оформления табличных сведений, таких как списки сотрудников или контрагентов, различные каталоги, библиографические или регистрационные карточки. Подобные записи следует создавать так, чтобы из них можно было выбирать самые разнообразные сведения по всевозможным критериям. Как правило, такая информация хранится в реляционных базах данных в виде одной или нескольких таблиц. У каждой таблицы есть свое имя, столбцы таблицы тоже снабжены именами. Некоторые таблицы связаны между собой внешними ключами.

Нужные сведения выбираются из базы данных операторами SQL, при этом сохраняется структура таблиц — выборка состоит из строк, содержащих значения одного или нескольких столбцов. К каждому значению можно обратиться по имени столбца.

Такая структура выборки естественно переходит в документ XML. Имя базы данных становится именем корневого элемента, имя таблицы служит именем элемента, содержащего строку выборки, имена столбцов размечают значения строки. Этот процесс легко автоматизировать. Уже написано мно-

го программ, представляющих выборку из базы данных в виде документа XML. Они могут сгенерировать другую структуру документа XML, создать атрибуты элементов, изменить имена элементов.

Возьмем классический пример образцово спроектированной базы данных "Поставщик-Детали", состоящей из трех таблиц.

Таблица SUPPLIER (табл. 1.1) состоит из строк, содержащих уникальный номер поставщика SNUM, его имя SNAME и место его проживания LOC.

Таблица 1.1. SUPPLIER

SNUM	SNAME	LOC
10123	Иванов	Санкт-Петербург
10212	Петров	Москва
10034	Гейтс	Редмонд

Таблица PRODUCT (табл. 1.2) содержит уникальный номер PNUM каждой детали, ее название PNAME, цвет COLOR и вес WEIGHT.

Таблица 1.2. PRODUCT

PNUM	PNAME	COLOR	WEIGHT
012	Болт	Синий	25
013	Гайка	Синий	20
026	Шайба	Белый	5
029	Шуруп	Черный	12

Третья таблица SP (табл. 1.3) связывает первые две таблицы. Она содержит номера поставщиков и деталей SNAME и PNAME, а также количество деталей QTY, поступивших от каждого поставщика.

Таблица 1.3. SP

SNAME	PNAME	QTY
10123	Гайка	500
10123	Болт	800
10212	Гайка	300
10034	Шайба	800
10034	Шуруп	10 000

В листинге 1.5 приведена выборка из базы данных "Поставщик-Детали", записанная как документ XML. В ней представлены сведения обо всех поставках гаек.

Листинг 1.5. Представление выборки из базы данных на языке XML

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE supplprod SYSTEM "suppl-prod.dtd">

<sp:supplprod xmlns:sp = "http://habib.bhv.ru/2003/supplprodml">

  <sp:suppl-prod>

    <sp:snum>10123</sp:snum>
    <sp:sname>Иванов</sp:sname>
    <sp:loc>Санкт-Петербург</sp:loc>
    <sp:pname>гайка</sp:pname>
    <sp:qty>500</sp:qty>

  </sp:suppl-prod>

  <sp:suppl-prod>

    <sp:snum>10212</sp:snum>
    <sp:sname>Петров</sp:sname>
    <sp:loc>Москва</sp:loc>
    <sp:pname>гайка</sp:pname>
    <sp:qty>300</sp:qty>

  </sp:suppl-prod>

</sp:supplprod>
```

Упражнения

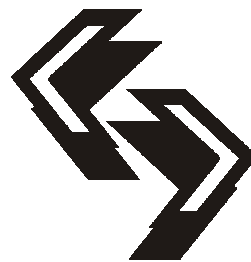
1. Преобразуйте документ XML, показанный в листинге 1.5, введя атрибуты в элемент `sp:suppl-prod`.
2. Запишите в виде документа XML другие выборки из базы данных "Поставщик-Детали".

Вопросы для самопроверки

1. Какая организация создала и развивает язык XML и на каком сайте расположены ее материалы?
2. На каких сайтах расположена полезная информация о создании документов XML?

3. Какими документами следует руководствоваться при создании реализации XML?
4. Что значит "хорошо оформленный документ"?
5. Что значит "верный" документ?
6. Обязателен ли пролог документа XML?
7. Обязателен ли корневой элемент?
8. Найдите ошибки в записи элемента `<Support id=1234>SD>12xz</support>`.
9. Верна ли запись `<item selected>Спички</item>`?
10. Какими способами можно записать символ "меньше" в содержимом элемента?
11. Можно ли использовать одинаковые имена во вложенных элементах, если данные имена вложены в разные элементы?
12. Можно ли записать атрибуты в закрывающем теге?
13. Как можно предотвратить разбор элемента программой-анализатором?
14. Почему идентификатор пространства имен записывается как адрес URI?
15. Можно ли использовать в одном документе XML несколько пространств имен?
16. Что такое "пространство имен по умолчанию"?

ГЛАВА 2



Описание структуры документа средствами DTD

Каждая реализация XML вводит новые теги. Программа, просматривающая документ XML, размеченный этими тегами, должна "понимать" вложенность, порядок следования и взаимосвязь тегов, число и тип атрибутов, а также другие характеристики тегов, образующие *схему* документа. Для этого к документу XML следует приложить формализованное описание, сделанное на языке, понятном для программы-анализатора. Такое описание можно сделать разными способами. Создано несколько языков описания схемы документа XML. В этой главе мы рассмотрим язык DTD (Document Type Definition), а в следующей главе — язык XSD.

Конструкции DTD

Описание схемы документа XML, сделанное на языке определения типа документа DTD, выглядит очень непривычно. Оно состоит из *объявлений разметки* (markup declarations), начинающихся с пары символов `<!`. За этими символами идет одно из слов `ELEMENT`, `ATTLIST`, `ENTITY`, `NOTATION`, указывающих, что именно объявляется: элемент, список атрибутов, сущность или обозначение. Объявление разметки заканчивается символом "больше". Каждый элемент или атрибут объявляется отдельно, поэтому все описание DTD часто бывает очень большим, иногда превышая объем описываемого документа. Рассмотрим подробнее объявление разметки каждого типа.

Объявление типа элемента

Каждый элемент документа XML должен быть описан в объявлении типа элемента (element type declaration). Объявление типа элемента начинается с символов `<!ELEMENT.`, после которых через пробел идет имя элемента и объявляется его содержимое.

Проще всего объявляется пустой элемент. Его содержимое, точнее, отсутствие содержимого, отмечается одним словом `EMPTY`. Например:

```
<!ELEMENT br EMPTY>
```

После такого объявления в документе XML будет верна запись пустого элемента `br` в виде `
` или в виде

```
<br></br>
```

Следующее объявление можно считать полной противоположностью объявлению пустого элемента:

```
<!ELEMENT something ANY>
```

Оно означает, что элемент с именем `something` может содержать что угодно. Такое объявление не несет никакой информации об элементе и поэтому встречается очень редко.

Элемент с обычным текстовым содержимым и без вложенных элементов объявляется так:

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

Слово `#PCDATA` (`Parsed Character DATA`) означает строку символов, просматриваемую программой-анализатором документа XML.

Если объявляемый элемент содержит вложенные элементы, то объявление должно содержать список их имен, перечисленных через запятую, в скобках. Вложенные элементы должны следовать в документе XML в том порядке, в каком они перечислены в объявлении. Например, объявление элемента `sp:suppl-prod` из листинга 1.5 может выглядеть так:

```
<!ELEMENT sp:suppl-prod (sp:snum, sp:sname, sp:loc, sp:pname, sp:qty)>
```

Обратите внимание на то, что объявляется только один уровень вложенности. Следующий уровень будет объявлен при объявлении вложенного элемента.

Кроме вложенных элементов, внутри элемента может встретиться обычный текст. В таком случае объявление элемента будет выглядеть следующим образом:

```
<!ELEMENT someelement (#PCDATA, inelem, anoelem)>
```

В приведенных выше объявлениях каждый вложенный элемент может встретиться ровно один раз. То обстоятельство, что вложенный элемент можно записать в объявляемом элементе несколько раз, отмечается звездочкой, плюсом или вопросительным знаком.

Например, в корневой элемент `sp:supplprod` листинга 1.5 можно вложить сколько угодно элементов `sp:suppl-prod`. Поэтому объявление корневого элемента будет выглядеть так:

```
<!ELEMENT sp:supplprod (sp:suppl-prod)* >
```

Звездочка после закрывающей круглой скобки означает, что список вложенных элементов может повторяться в документе XML несколько раз или не встречаться вовсе. Звездочку можно поставить и после отдельного имени вложенного элемента в списке. Это будет означать повторяемость или отсутствие данного элемента в документе XML. Наконец, можно выделить скобками группу повторяющихся элементов и поставить звездочку после нее.

Есть еще два обозначения такого же рода: знак "плюс", означающий, что элемент или список элементов может повторяться один или несколько раз, и вопросительный знак, показывающий, что элемент или список может войти в документ не более одного раза.

Например, объявление элемента `bk:chapter` из листинга 1.4 может выглядеть так:

```
<!ELEMENT bk:chapter (bk:title, (bk:section+, bk:listing*,
    bk:picture*)*, bk:conclusion?)>
```

Это объявление показывает, что в элемент `bk:chapter` сначала вкладывается один элемент `bk:title`, потом один или несколько элементов `bk:section`. Между элементами `bk:section` могут встречаться элементы `bk:listing` и `bk:picture`. Последним вложенным элементом может оказаться элемент `bk:conclusion`.

Итак:

- ☐ ? — элемент или список может встретиться нуль или один раз;
- ☐ * — элемент или список может встретиться нуль или несколько раз;
- ☐ + — элемент или список может встретиться один или несколько раз.

Иногда из нескольких вложенных элементов или списков может встретиться только один. В таком случае их имена перечисляются через вертикальную черту. Например, при описании книги, кроме имени автора и названия, указывается издательство и место издания, а при описании журнальной статьи — номер журнала. Объявление элемента `edition`, общее для книги и журнала, может выглядеть так:

```
<!ELEMENT edition (author+, title, name,
    (publ-house, publ-place) | issue)) >
```

Упражнения

1. Сделайте объявления элементов, встретившихся в листингах *главы 1*.
2. Список вложенных элементов в объявлении элемента строго задает порядок их следования в документе XML. Как сделать объявление вложенных элементов, которые могут идти в любом порядке?

Объявление атрибутов

Атрибуты элемента объявляются уже после объявления самого элемента. Все атрибуты одного элемента объявляются сразу, одним списком. Список начинается с символов `<!ATTLIST`, после них через пробел следует имя элемента, к которому относятся атрибуты. Затем идут объявления атрибутов. Список, как всегда, заканчивается символом "больше".

Для каждого атрибута записывается его имя, тип и признак обязательности.

Тип атрибута записывается приведенными далее обозначениями.

- ☐ `CDATA` — строка символов.
- ☐ `ID` — уникальный идентификатор, однозначно определяющий элемент, в котором встретился этот атрибут; значения такого атрибута не должны повторяться в документе.
- ☐ `IDREF` — идентификатор, содержащий одно из значений атрибутов типа `ID`, используется в качестве ссылки на другие элементы.
- ☐ `IDREFS` — идентификатор, содержащий набор значений атрибутов типа `ID`, перечисленных через пробелы; тоже используется в качестве ссылки сразу на несколько элементов.
- ☐ `ENTITY` — имя непроверяемой анализатором сущности.
- ☐ `ENTITIES` — имена непроверяемых сущностей.
- ☐ `NMTOKEN` — слово, содержащее только символы, применяемые в именах.
- ☐ `NMTOKENS` — слова, перечисленные через пробелы.
- ☐ `NOTATION` — обозначение.
- ☐ Список значений атрибута, перечисляемых через вертикальную черту, в скобках.

Атрибуты типа `ID` играют ту же роль, что и первичные ключи в таблицах баз данных. Они позволяют различать элементы и индексировать их. Это облегчает и ускоряет поиск элементов, а также позволяет ссылаться на них. Ссылки выполняются атрибутами типа `IDREF` или типа `IDREFS`. Например, в одном месте документа могут встретиться элементы:

```
<name reg-num="123045">Иванов</name>
<name reg-num="123052">Петров</name>
```

с атрибутом `reg-num` типа `ID`, а в другом месте — элемент:

```
<post ref="123045">Заведующий отделом</post>
```

с атрибутом `ref` типа `IDREF`. Такая связь означает, что должность Иванова — "Заведующий отделом". Объявления DTD этих элементов могут выглядеть так:

```
<!ELEMENT name (#PCDATA)>
<!ATTLIST name
  reg-num ID #REQUIRED>
<!ELEMENT post (#PCDATA)>
<!ATTLIST post
  ref IDREF #IMPLIED>
```

Значения атрибутов типа `ENTITY` — это имена сущностей, объявленных в этом же описании DTD.

Атрибуты типа `NMTOKEN` могут содержать имена других элементов или атрибутов, например, для того чтобы ссылаться на них.

Значения атрибутов типа `NOTATION` — это обозначения, расшифрованные в описании DTD.

Объявление атрибута заканчивается признаком обязательности присутствия атрибута в элементе или подразумеваемым значением атрибута. Такое значение, называемое значением по умолчанию, будет использовано, если атрибут не записан явно в документе XML. Признак обязательности — это одно из трех слов, показывающих, всегда ли нужно записывать атрибут и что будет, если его нет в элементе:

- `#REQUIRED` — атрибут надо обязательно записывать в элементе;
- `#IMPLIED` — атрибут необязателен, у него нет значения по умолчанию;
- `#FIXED` — у атрибута есть только одно значение, которое записывается тут же, через пробел.

Если в конце объявления атрибута вместо одного из этих слов стоит значение атрибута, принимаемое по умолчанию, то понятно, что такой атрибут необязателен.

Например, атрибуты `first`, `second` и `surname` элемента `name` из листинга 1.2 можно объявить следующим списком:

```
<!ATTLIST name
  first    CDATA #REQUIRED
  second   CDATA #IMPLIED
  surname  CDATA #REQUIRED>
```

Второй пример. У каждого листинга в книге, которую вы читаете, есть свой уникальный номер, поэтому атрибуты элемента `bk:listing` из листинга 1.4 объявляются так:

```
<!ATTLIST bk:listing
  bk:number ID #REQUIRED
  bk:head   CDATA #REQUIRED>
```

Внимание!

Обратите внимание на то, что при использовании пространства имен надо всегда указывать уточненное (QName), а не локальное имя.

Еще пример. В *главе 1* в элемент `city`, встречавшийся в листингах 1.1—1.3, мы хотели внести атрибут `type`, принимающий одно из трех значений: `город`, `поселок`, `деревня`. Значение `город` принималось по умолчанию. Вот описание этого атрибута:

```
<!ATTLIST city
  type (город | поселок | деревня) "город">
```

Следующий пример взят из спецификации XML. В прошлой главе нам встречались атрибуты `xml:lang` и `xml:space` с предопределенным префиксом `xml`. Там же было сказано, что их надо объявлять в описании DTD, иначе документ не будет считаться верным. Вот как это можно сделать:

```
<!ATTLIST p xml:lang NMTOKEN "ru_RU">
<!ATTLIST pre xml:space (default | preserve) "preserve">
```

Упражнения

1. Объявите все атрибуты листингов 1.3 и 1.4.
2. Сделайте описание документов из упражнений *главы 1*.

Объявление сущности

В *главе 1* мы встречали ссылки на сущности `<`, `>`, `&` и т. д. Они начинаются с амперсанда, а заканчиваются точкой с запятой. Ссылки на сущности используются как краткие обозначения для громоздких или часто повторяющихся фрагментов документа XML. Сами сущности (entity), подставляемые в документ вместо ссылок, объявляются в описании DTD.

Все сущности можно разделить на три группы:

- ❑ *внутренние сущности* — задаются при объявлении сущности;
- ❑ *внешние сущности* — содержатся в отдельных файлах или встроены в программу-анализатор;
- ❑ *параметризованные сущности* — используются только внутри описания DTD.

Объявление внутренней сущности (entity declaration) начинается с символов `<!ENTITY`, после которых идет имя сущности. Через пробел записывается сама сущность — ее значение в кавычках. Например:

```
<!ENTITY author "Александр Сергеевич Пушкин">
```

После такого объявления программа-анализатор, увидев в документе ссылку на сущность `&author;`, подставит вместо нее имя, отчество и фамилию великого поэта. Ссылку на сущность можно применять тут же, в описании DTD, уже в следующем объявлении:

```
<!ENTITY cover "Автор &author;. Название &title;.">
```

Конечно, надо следить за тем, чтобы сущности `author`, `title` и `cover` не определялись друг через друга, образуя бесконечную рекурсию.

Такие сущности называются *внутренними* (internal entities), потому что для их объявления не нужен никакой внешний объект. Сущность представлена тут же, в этом же объявлении. Сущности могут быть и внешними. Для *внешних* сущностей (external entities) указывается только место их расположения в виде адреса URI. Перед указанием адреса записывается одно из слов `SYSTEM` или `PUBLIC`. Вот примеры объявления внешних сущностей:

```
<!ENTITY tel
  SYSTEM "http://www.tty.com/TelDef.xml">
<!ENTITY tel
  PUBLIC "-//DTD/Teletanomy//TEXT The new teletanomy definition//EN"
  "http://www.tty.com/TelDef.xml">
```

Как видно из этих примеров, разница между пометками `SYSTEM` и `PUBLIC` заключается в том, что после слова `PUBLIC` идет какое-то общеизвестное объявление. Обычно здесь записывается известная ссылка, введенная консорциумом W3C или другой организацией. Часто значение такой ссылки встраивается прямо в программу-анализатор. Если программа-анализатор не найдет эту ссылку, то она воспользуется адресом URI, идущим за ссылкой.

Объявление *параметризованных* сущностей (parametric entities), используемых только внутри описания DTD, выполняется точно так же, как объявление внутренних и внешних сущностей, только между началом объявления `<!ENTITY` и именем сущности вставляется знак процента, отделенный пробелами:

```
<!ENTITY % lang "ru_RU">
```

Ссылка на параметризованную сущность начинается не с амперсанда, а со знака процента, в примере — `%lang;`. Ссылка `%lang;` будет использована в описании DTD для обозначения языка и территории. Введение этой ссылки удобно тем, что при смене языка надо будет поменять значение `ru_RU` только в одном месте описания.

Все сущности можно разделить и по другому признаку — разбираемые (parsed) и не разбираемые (unparsed) программой-анализатором.

Разбираемые сущности представляют собой фрагмент документа XML или целый документ и подлежат обработке программой-анализатором после

подстановки их вместо ссылки на сущность. После подстановки разбираемая сущность становится просто частью документа XML.

Иногда сущность не надо обрабатывать средствами XML, например, это двоичный программный код, чертеж или другое изображение. Для того чтобы предотвратить обработку такой информации, нужно указать в объявлении, что это не разбираемая сущность. Для этого в конце объявления не разбираемой сущности делается пометка `NDATA` и указывается обозначение (notation) вставляемого объекта. Пусть, например, в документ надо вставить изображение в формате GIF. Объявляем не разбираемую сущность `tel-logo`:

```
<!ENTITY tel-logo
  SYSTEM "/images/TelDefLogo32x16.gif" NDATA image-gif >
```

Изображение, находящееся в файле `/images/TelDefLogo32x16.gif`, каким-то образом должно быть обработано, например, выведено на экран дисплея или распечатано на принтере. Это может сделать специализированная программа обработки изображений. Обозначение `image-gif`, указанное после слова `NDATA`, задает такую программу.

Речь об обозначениях пойдет в следующем разделе, а сейчас, заканчивая этот раздел, напомним, что в языке XML имеется пять предопределенных сущностей, ссылающихся на угловые скобки, обозначаемые символами "меньше" (`<`) и "больше" (`>`), на амперсанд (`&`), на кавычки (`"`) и на апостроф (`'`). Вот их объявления, взятые из спецификации XML:

```
<!ENTITY lt      "&#38;#60;">
<!ENTITY gt      "&#62;">
<!ENTITY amp     "&#38;#38;">
<!ENTITY apos    "&#39;">
<!ENTITY quot    "&#34;">
```

Как видите, символы записываются своими десятичными кодами в кодировке Unicode, чтобы предотвратить их обработку.

Упражнения

1. Отметьте часто повторяемые значения в ваших документах, оформите их в виде сущностей и объявите в описании DTD.
2. Используйте внешние сущности для вставки одних документов XML в другие документы.

Объявление обозначения

Обозначение (notation) объявляется подобно сущности. Обозначения тоже могут быть внутренними и внешними. При объявлении внутренних обозна-

чений после символов `<!NOTATION` записывается имя обозначения, затем в кавычках — его расшифровка.

При объявлении внешних обозначений используются пометки `SYSTEM` или `PUBLIC`, причем за словом `PUBLIC` не обязательно должна следовать общеизвестная ссылка, так что слова `SYSTEM` и `PUBLIC` здесь равнозначны. Продолжая пример предыдущего раздела, можно написать

```
<!NOTATION image-gif SYSTEM "viewer.exe">
```

Это объявление связывает обозначение `image-gif` с программой обработки изображений, находящейся в файле `viewer.exe`.

Пример: описание DTD записной книжки

Теперь, после того как мы подробно разобрали конструкции описания DTD, можно полностью формализовать описание записной книжки, приведенное в *главе 1*. Это описание DTD, соответствующее листингу 1.2, записано в листинге 2.1.

Листинг 2.1. Описание DTD записной книжки из главы 1

```
<!ELEMENT notebook (person)*>
<!ELEMENT person (name, birthday?, address*, phone-list?)>
<!ELEMENT name EMPTY>
<!ATTLIST name
    first    CDATA #IMPLIED
    second   CDATA #IMPLIED
    surname  CDATA #REQUIRED>
<!ELEMENT birthday (#PCDATA)>
<!ELEMENT address (street, city, zip)?>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ATTLIST city
    type (город | поселок | деревня) "город">
<!ELEMENT zip (#PCDATA)>
<!ELEMENT phone-list (work-phone*, home-phone*)>
<!ELEMENT work-phone (#PCDATA)>
<!ELEMENT home-phone (#PCDATA)>
```

Как видите, описание DTD записной книжки почти очевидно. Оно повторяет приведенное в *главе 1* словесное описание. В первой строке слово `ELEMENT` означает, что элемент `notebook` может содержать вложенные элементы `person`, которые перечисляются в круглых скобках. Звездочка после закрывающей скобки означает, что их может быть сколько угодно.

Во второй строке объявляется элемент `person`. Он может содержать элементы `name`, `birthday`, `address`, `phone-list`. Порядок перечисления этих вложенных элементов в скобках должен соответствовать порядку их появления в документе.

Элемент `name` обязателен и может встретиться внутри элемента `person` только один раз.

Элементы `birthday` и `phone-list` необязательны, но могут встретиться только один раз.

Элемент `address` необязателен, его можно записать несколько раз.

Слово `EMPTY` в третьей строке листинга 2.1 означает, что элемент `name` — пустой.

Далее, слово `ATTLIST` начинает описание списка атрибутов элемента `name`. Для каждого атрибута указывается имя, тип и обязательность указания атрибута. Существует всего девять типов атрибута, они приведены выше, но чаще всего употребляется тип `CDATA`, означающий произвольную строку символов Unicode, или перечисляются значения типа. Так сделано в описании атрибута `type` элемента `city`, принимающего одно из трех значений: город, поселок или деревня. В кавычках показано значение по умолчанию `город`.

Объявление атрибута `surname` заканчивается пометкой `#REQUIRED`, означающей, что атрибут надо записывать обязательно. Атрибуты `first` и `second` помечены `#IMPLIED`. Эти атрибуты необязательны. У атрибута `type` вместо этих пометок задано значение по умолчанию `город`. Такое объявление делает атрибут `type` необязательным.

При объявлении элемента `phone-list` звездочка поставлена после имени каждого вложенного элемента. После такого объявления вложенные элементы `work-phone` и `home-phone` можно записывать в документе XML в любом порядке.

Размещение описания DTD

Описание DTD можно занести в отдельный файл, например `ntb.dtd`, указав его имя во второй части пролога `DOCTYPE`, как показано во второй строке каждого листинга главы I. Можно включить описание непосредственно во вторую часть пролога XML-файла, заключив его в квадратные скобки, как это сделано в листинге 2.2.

Листинг 2.2. Описание DTD в документе XML

```
<?xml version="1.0" encoding="Windows-1251"?>
<!DOCTYPE notebook [
```

```

<!ELEMENT notebook (person)*>
<!ELEMENT person (name, birthday?, address*, phone-list?)>
<!ELEMENT name EMPTY>
<!ATTLIST name
    first    CDATA #IMPLIED
    second   CDATA #IMPLIED
    surname  CDATA #REQUIRED>
<!ELEMENT birthday (#PCDATA)>
<!ELEMENT address (street, city, zip)?>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ATTLIST city
    type (город | поселок | деревня) "город">
<!ELEMENT zip (#PCDATA)>
<!ELEMENT phone-list (work-phone*, home-phone*)>
<!ELEMENT work-phone (#PCDATA)>
<!ELEMENT home-phone (#PCDATA)>
]>

<notebook>

  <person>

    <name first="Иван" second="Петрович" surname="Сидоров" />

    <birthday>25.03.1977</birthday>

    <address>
      <street>Садовая, 23-15</street>
      <city>Урюпинск</city>
      <zip>123456</zip>
    </address>

    <phone-list>
      <work-phone>2654321</work-phone>
      <work-phone>2654023</work-phone>
      <home-phone>3456781</home-phone>
    </phone-list>

  </person>

  <person>

    <name first="Мария" second="Петровна" surname="Сидорова" />

    <birthday>17.05.1969</birthday>

```

```
<address>
  <street>Ягодная, 17</street>
  <city>Жмеринка</city>
  <zip>234561</zip>
</address>

<phone-list>
  <home-phone>2334455</home-phone>
</phone-list>

</person>

</notebook>
```

Программы-анализаторы XML

После того как создано описание DTD нашей реализации XML и написан документ, размеченный тегами этой реализации, следует проверить правильность их написания и соответствие их друг другу. Для этого есть специальные программы — *проверяющие анализаторы* (validating parsers). Все фирмы, разрабатывающие средства для работы с XML, выпускают бесплатные или коммерческие проверяющие анализаторы. Например, фирма IBM выпускает анализатор xml4j, написанный на языке Java. Он входит в состав сервера приложений WebSphere, но его можно использовать и отдельно, свободно загрузив с адреса <http://www.alphaworks.ibm.com/> архивы xml4j.jar, xerces.jar и xercesSamples.jar.

Проверяющий анализатор фирмы Sun Microsystems, написанный тоже на языке Java, содержится в пакете Java-классов JAXP (Java API for XML Processing), входящем в состав и J2SDK Standard Edition, и J2SDK Enterprise Edition. Кроме того, этот пакет можно загрузить отдельно или в составе пакета Java XML Pack с адреса <http://java.sun.com/xml/>.

Корпорация Microsoft предоставляет проверяющий анализатор MSXML (Microsoft XML Parser), доступный по адресу <http://msdn.microsoft.com/xml/>.

У корпорации Oracle есть полный набор средств для работы с XML, в том числе и анализатор Oracle XML Parser. Его можно получить по адресу <http://otn.oracle.com/tech/xml/>.

Фирма Mind Electric занимается разработкой программных продуктов, использующих XML. Ее анализатор Electric XML и другие продукты находятся по адресу <http://www.themindelectric.com/products/xml/>.

Большую популярность получил анализатор XP Джеймса Кларка, который можно найти по адресу <http://www.jclark.com/xml/xp/>.

Есть еще множество проверяющих анализаторов, но лидером среди них является, пожалуй, Apache Xerces 2, входящий во многие средства обработки

документов XML, выпускаемые другими фирмами. Он создан сообществом Apache Software Foundation и свободно доступен по адресу <http://xml.apache.org/xerces2-j/>.

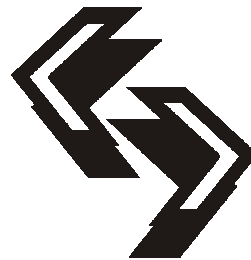
Заключение

Ограниченные средства DTD не позволяют полностью описать структуру документа XML. В частности, описание DTD не указывает точное количество повторений вложенных элементов, оно не задает точный тип содержимого элемента. Например, в листинге 2.2 из описания DTD не видно, что в элементе `birthday` содержится дата рождения. Эти недостатки DTD привели к появлению других схем описания документов XML. Наиболее развитое описание дает язык XSD (XML Schema Definition Language), который мы рассмотрим в следующей главе. Мы будем называть описание на этом языке просто *схемой XML* (XML Schema).

Вопросы для самопроверки

1. Для чего нужно описание документа XML?
2. Какие конструкции описывает DTD?
3. Какие сведения об элементе можно указать при объявлении элемента XML?
4. Можно ли объявить атрибуты элемента вместе с его объявлением?
5. Как указать порядок следования вложенных элементов?
6. Как объявляются атрибуты элемента?
7. Что такое сущность в языке XML?
8. Чем отличаются внутренние и внешние сущности?
9. Для чего нужны параметризованные сущности?
10. Где можно поместить описание DTD?

ГЛАВА 3



Описание схемы документа на языке XSD

Описание структуры документа XML, выполненное средствами DTD, очень скоро перестало удовлетворять разработчиков. Потребовалось более точное описание схемы документа, учитывающее тип содержимого элемента, количество повторов вложенного элемента и другие подробности.

В мае 2001 года консорциум W3C рекомендовал описывать структуру документов XML на новом языке описания схем XSD (XML Schema Definition Language). На этом языке записывается *схема XML* (XML Schema), описывающая конструкции, использованные в документе XML.

Язык XSD создан как реализация XML. Это значит, что схема XML сама записывается в виде документа XML. Ее элементы называют *компонентами* (components), чтобы отличить их от элементов описываемого документа XML. Корневой компонент схемы носит имя `schema`. Компоненты схемы описывают элементы XML и определяют различные типы элементов. Рекомендация схемы XML, которую можно посмотреть по адресу <http://www.w3.org/XML/Schema/>, перечисляет 13 типов компонентов, но наиболее важны компоненты, определяющие простые и сложные типы элементов, сами элементы и их атрибуты.

Язык XSD различает простые и сложные элементы XML. *Простыми* (simple) элементами описываемого документа XML считаются элементы, не содержащие атрибутов и вложенных элементов. Соответственно, *сложные* (complex) элементы содержат атрибуты и/или вложенные элементы. Схема XML определяет *простые типы* — типы простых элементов, и *сложные типы* — типы сложных элементов.

Язык описания схем содержит множество встроенных простых типов, перечисленных в следующем разделе.

Встроенные простые типы XSD

Встроенные типы языка описания схем XSD позволяют записывать двоичные и десятичные целые числа, вещественные числа, дату и время, строки символов, логические значения, адреса URI. Рассмотрим их по порядку.

Вещественные числа

Вещественные числа в языке XSD разделены на три типа: `decimal`, `float` и `double`.

Тип `decimal` составляют вещественные числа, записанные с фиксированной точкой: 123.45, -0.1234567689345 и т. д. Фактически хранятся два целых числа. Одно число представляет мантиссу, другое — порядок вещественного числа. Спецификация языка XSD не ограничивает количество цифр в мантиссе, но требует, чтобы можно было записать не менее 18 цифр. При обработке документа средствами технологии Java этот тип легко реализуется, например, классом `java.math.BigDecimal`, входящим в стандарт Java API.

Типы `float` и `double` соответствуют стандарту IEEE754–85 и одноименным типам Java. Они записываются с фиксированной или с плавающей десятичной точкой. Например, 34.567, -45.67 , $1e-5$, $34.58e14$.

Целые числа

Основной целый тип `integer` понимается как подтип типа `decimal`, содержащий числа с нулевым порядком. Это целые числа с любым количеством десятичных цифр: -34567 , 123456789012345 и т. д. При использовании средств Java для обработки документа этот тип легко реализуется классом `java.math.BigInteger`.

Типы `long`, `int`, `short` и `byte` полностью соответствуют одноименным типам Java. Они понимаются как подтипы типа `integer`, типы более коротких чисел считаются подтипами более длинных чисел, например тип `byte` — это подтип типа `short`, оба они подтипы типа `int` и т. д.

Значения типа `byte`, как следует из его названия, занимают один байт и изменяются от -128 до 127. Тип `short` занимает два байта, его значения лежат в диапазоне от -32768 до $+32767$. Числа типа `int` хранятся в четырех байтах и меняются от -2147483648 до $+2147483647$. Наконец, тип `long` располагается в восьми байтах, его значения от -9223372036854775808 до $+9223372036854775807$.

Типы `nonPositiveInteger` и `negativeInteger` — подтипы типа `integer` — составлены из неположительных и отрицательных чисел соответственно с любым количеством цифр.

Типы `nonNegativeInteger` и `positiveInteger` — подтипы типа `integer` — составлены из неотрицательных и положительных чисел соответственно с любым количеством цифр.

У типа `nonNegativeInteger` есть подтипы целых чисел без знака `unsignedLong`, `unsignedInt`, `unsignedShort` и `unsignedByte`.

Строки символов

Основной символьный тип `string` описывает произвольную строку символов Unicode. Его можно реализовать средствами Java, используя класс `java.lang.String`.

Тип `normalizedString` — подтип типа `string` — это строки, не содержащие символов перевода строки `'\n'`, возврата каретки `'\r'` и горизонтальной табуляции `'\t'`.

В строках типа `token` — подтипа типа `normalizedString` — нет, кроме того, начальных и завершающих пробелов и нескольких подряд идущих пробелов.

В типе `token` выделены три подтипа. Подтип `language` определен для записи названия языка согласно рекомендации RFC 1766, например, `ru`, `en`, `de`, `fr`. Подтип `NMTOKEN` используется только в атрибутах для записи их перечисляемых значений. Подтип `name` составляют *имена XML* — последовательности букв, цифр, дефисов, точек, двоеточий, знаков подчеркивания, начинающиеся с буквы (кроме зарезервированной последовательности букв `X`, `x`, `M`, `m`, `L`, `l` в любом сочетании регистров) или знака подчеркивания. Мы видели в предыдущих главах, что имена, начинающиеся со строки `xml`, используются самой спецификацией XML, например, имя атрибута `xmlns`. Двоеточие в значениях типа `name` применяется для выделения префикса в уточненных именах при использовании пространства имен.

Из типа `name` выделен подтип `NCName` (Non-Colonized Name) имен, не содержащих двоеточия, в котором, в свою очередь, определены три подтипа: `ID`, `ENTITY`, `IDREF`, — описывающие идентификаторы XML, сущности и перекрестные ссылки на идентификаторы.

Дата и время

Тип `duration` описывает промежуток времени, например, запись `P1Y2M3DT10H30M45S` означает один год (1Y), два месяца (2M), три дня (3D), десять часов (10H), тридцать минут (30M) и сорок пять секунд (45S). Запись может быть сокращенной, например, `P120M` означает 120 месяцев, а `T120M` — 120 минут.

Тип `dateTime` содержит дату и время в формате `CCYY-MM-DDThh:mm:ss`, например, `2003-04-25T09:30:05`. Остальные типы выделяют какую-либо часть даты или времени.

Тип `time` содержит время в обычном формате `hh:mm:ss`.

Тип `date` содержит дату в формате `CCYY-MM-DD`.

Тип `gYearMonth` выделяет год и месяц в формате `CCYY-MM`.

Тип `gMonthDay` содержит месяц и день месяца в формате `-MM-DD`.

Тип `gYear` означает год в формате `CCYY`, тип `gMonth` — месяц в формате `-MM-`, тип `gDay` — день месяца в формате `-DD`.

Двоичные типы

Двоичные целые числа записываются либо в шестнадцатеричной форме без всяких дополнительных символов: `0B2F`, `356C0A` и т. д., это тип `hexBinary`, либо в кодировке `Base64`, это тип `base64Binary`.

Прочие встроенные простые типы

Еще три встроенных простых типа описывают значения, часто используемые в документах XML.

Адреса URI относятся к типу `anyURI`.

Расширенное имя тега или атрибута (*qualified name*), т. е. имя вместе с префиксом, отделенным от имени двоеточием, — это тип `QName`.

Обозначение `NOTATION` описания DTD выделено как отдельный простой тип схемы XML. Его используют для записи математических, химических и других символов, нот, азбуки Брайля и прочих обозначений.

Определение простых типов

В схемах XML с помощью встроенных типов можно тремя способами определить новые типы простых элементов. Они вводятся как *сужение* (*restriction*) встроенного или ранее определенного простого типа, *список* (*list*) или *объединение* (*union*) простых типов.

Простой тип определяется компонентом схемы `simpleType`, имеющим вид

```
<xsd:simpleType name="имя типа">Определение типа</xsd:simpleType>
```

Сужение

Сужение простого типа определяется компонентом `restriction`, в котором атрибут `base` указывает сужаемый простой тип, а в содержимом задаются ограничения, выделяющие определяемый простой тип. Например, почтовый индекс `zip` можно определить как шесть арабских цифр следующим образом:

```
<xsd:simpleType name="zip">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{6}" />
  </xsd:restriction>
</xsd:simpleType>
```

Можно дать другое определение простого типа `zip` как целого положительного числа, находящегося в диапазоне от 100000 до 999999:

```
<xsd:simpleType name="zip">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:minInclusive value="100000" />
    <xsd:maxInclusive value="999999" />
  </xsd:restriction>
</xsd:simpleType>
```

Теги `<pattern>`, `<maxInclusive>` и др., задающие ограничения, называются *фасетками* (facets). Вот их список:

- ☐ `<maxExclusive>` — наибольшее значение, которое уже не входит в определяемый тип;
- ☐ `<maxInclusive>` — наибольшее значение определяемого типа;
- ☐ `<minExclusive>` — наименьшее значение, уже не входящее в определяемый тип;
- ☐ `<minInclusive>` — наименьшее значение определяемого типа;
- ☐ `<totalDigits>` — общее количество цифр в определяемом числовом типе — сужении типа `decimal`;
- ☐ `<fractionDigits>` — количество цифр в дробной части числа;
- ☐ `<length>` — длина значений определяемого типа;
- ☐ `<maxLength>` — наибольшая длина значений определяемого типа;
- ☐ `<minLength>` — наименьшая длина значений определяемого типа;
- ☐ `<enumeration>` — одно из перечислимых значений;
- ☐ `<pattern>` — регулярное выражение;
- ☐ `<whiteSpace>` — применяется при сужении типа `string` и определяет способ преобразования пробельных символов `'\n'`, `'\r'`, `'\t'`. Атрибут `value` этого тега принимает одно из трех значений:
 - `preserve` — не убирать пробельные символы;
 - `replace` — заменить пробельные символы пробелами;

- `collapse` — после замены пробельных символов пробелами убрать начальные и конечные пробелы, а из нескольких подряд идущих пробелов оставить только один.

В тегах-фасетках можно записывать следующие атрибуты, называемые *базисными фасетками* (fundamental facets):

- ☐ `ordered` — задает упорядоченность определяемого типа, принимает одно из трех значений:
 - `false` — тип неупорядочен;
 - `partial` — тип частично упорядочен;
 - `total` — тип полностью упорядочен;
- ☐ `bounded` — задает ограниченность или неограниченность типа значением `true` или `false`;
- ☐ `cardinality` — задает конечность или бесконечность типа значением `finite` или `countably infinite`;
- ☐ `numeric` — показывает, числовой этот тип или нет, значением `true` или `false`.

Как видно из приведенных выше и ниже примеров, в одном сужении может быть несколько ограничений-фасеток. При этом фасетки `<pattern>` и `<enumeration>` задают независимые друг от друга ограничения, их можно мысленно объединить союзом "или". Остальные фасетки задают общие, совместно накладываемые ограничения, их можно мысленно объединить союзом "и".

Список

Простой тип-список — это тип элементов, в теле которых записываются через пробел несколько значений одного и того же простого типа. Например, в документе XML может встретиться такой элемент, содержащий список целых чисел:

```
<days>21 34 55 46</days>
```

Список определяется компонентом `list`, в котором атрибутом `itemType` указывается тип элементов определяемого списка. Тип элементов списка можно определить и в содержимом элемента `list`. Например, показанный выше элемент документа XML `days` можно определить в схеме так:

```
<xsd:element name="days" type="listOfInteger" />
```

а использованный при его определении тип `listOfInteger` задать как список не более чем из пяти целых чисел следующим образом:

```

<xsd:simpleType name="listOfInteger">

  <xsd:restriction>

    <xsd:simpleType>
      <xsd:list itemType="xsd:integer" />
    </xsd:simpleType>

    <xsd:maxLength value="5" />

  </xsd:restriction>

</xsd:simpleType>

```

При определении списка можно применять фасетки `<length>`, `<minLength>`, `<maxLength>`, `<enumeration>`, `<pattern>`. В приведенном выше примере список — тело элемента `days` — не может содержать более пяти чисел.

Объединение

Простой тип-объединение определяется компонентом `union`, в котором атрибутом `memberTypes` можно указать имена объединяемых типов. Например:

```

<xsd:union memberTypes="xsd:string xsd:integer listOfInteger" />

```

Другой способ — записать в содержимом компонента `union` определения простых типов, входящих в объединение. Например:

```

<xsd:attribute name="size">

  <xsd:simpleType>
    <xsd:union>

      <xsd:simpleType>

        <xsd:restriction base="xsd:positiveInteger">
          <xsd:minInclusive value="8"/>
          <xsd:maxInclusive value="72"/>
        </xsd:restriction>

      </xsd:simpleType>

      <xsd:simpleType>

        <xsd:restriction base="xsd:NMTOKEN">
          <xsd:enumeration value="small"/>
          <xsd:enumeration value="medium"/>
          <xsd:enumeration value="large"/>
        </xsd:restriction>

      </xsd:simpleType>

    </xsd:union>

  </xsd:simpleType>

```

```
        </xsd:simpleType>

        </xsd:union>
    </xsd:simpleType>

</xsd:attribute>
```

После этого атрибут `size` можно использовать, например, так:

```
<font size='large'>Глава 1</font>
<font size='12'>Простой текст</font>
```

Упражнения

1. Подберите простой тип для записи фамилии.
2. Определите тип телефонного номера.
3. Определите тип температуры, измеряемой в градусах Кельвина.
4. Определите тип угла, измеряемого в градусах.
5. Определите тип, состоящий всего из нескольких значений.

Объявление элементов и их атрибутов

Элементы, из которых будет состоять документ XML, объявляются в схеме компонентом `element`:

```
<xsd:element name="имя элемента" type="тип элемента"
  minOccurs="наименьшее число появлений элемента в документе"
  maxOccurs="наибольшее число появлений" />
```

Значение по умолчанию необязательных атрибутов `minOccurs` и `maxOccurs` равно 1. Это означает, что если эти атрибуты отсутствуют, то элемент должен появиться в документе XML ровно один раз. Например:

```
<xsd:element name="degree" type="xsd:nonPositiveInteger" />
```

Указание типа элемента в атрибуте `type` удобно, если это встроенный простой тип или тип, определенный заранее. Тогда в атрибуте `type` можно записать только имя типа.

Если же тип элемента определяется здесь же, то определение типа элемента лучше вынести в содержимое компонента `element`:

```
<xsd:element name="имя элемента" >
  Определение типа элемента
</xsd:element>
```

Объявление атрибута элемента тоже несложно:

```
<xsd:attribute name=" имя атрибута" type="тип атрибута"
  use="обязательность атрибута" default="значение по умолчанию" />
```

Необязательный атрибут `use` принимает три значения:

- ☐ `optional` — описываемый атрибут необязателен (это значение по умолчанию);
- ☐ `required` — описываемый атрибут обязателен;
- ☐ `prohibited` — описываемый атрибут неприменим. Это значение полезно при определении подтипа, чтобы отменить некоторые атрибуты базового типа.

Например:

```
<xsd:attribute name="id" type="positiveInteger" use="required" />
```

Если описываемый атрибут необязателен, то атрибутом `default` можно задать его значение по умолчанию:

```
<xsd:attribute name="name" type="NCName" default="anonymous" />
```

Определение типа атрибута, — а это должен быть простой тип, — можно вынести в содержимое элемента `attribute`:

```
<xsd:attribute name="имя атрибута">
  Тип атрибута
</xsd:attribute>
```

Упражнение

Объявите элементы, встретившиеся в листингах *главы 1*, и их атрибуты.

Определение сложных типов

Напомним, что тип элемента называется *сложным*, если в элемент вложены другие элементы и/или в открывающем теге элемента есть атрибуты.

Сложный тип определяется компонентом `complexType`, имеющим вид:

```
<xsd:complexType name="имя типа" >
  Определение типа
</xsd:complexType>
```

Необязательный атрибут `name` задает имя типа, а в содержимом компонента `complexType` описываются элементы, входящие в сложный тип, и/или атрибуты открывающего тега.

Определение сложного типа можно разделить на три группы:

- определение типа пустого элемента;
- определение типа элемента с простым телом;
- определение типа элемента, содержащего вложенные элементы.

Рассмотрим эти определения подробнее.

Определение типа пустого элемента

Проще всего определяется тип пустого элемента — элемента, не содержащего тела, а содержащего только атрибуты в открывающем теге. Таков, например, элемент `name` листинга 1.2. Каждый атрибут объявляется одним компонентом `attribute`, как в предыдущем разделе, например:

```
<xsd:complexType name="imageType">
  <xsd:attribute name="href" type="xsd:anyURI" />
</xsd:complexType>
```

После этого определения можно в схеме объявить элемент `image` типа `imageType`:

```
<xsd:element name="image" type="imageType" />
```

а в документе XML использовать это объявление:

```
<image href="http://some.com/images/myface.gif" />
```

Упражнения

1. Определите тип пустого элемента `
`.
2. Определите тип элемента `name` из листинга 1.2.

Определение типа элемента с простым телом

Немного сложнее описание элемента, содержащего тело простого типа и атрибуты в открывающем теге. Этот тип отличается от простого типа только наличием атрибутов и определяется компонентом `simpleContent`. В теле этого компонента должен быть либо компонент `restriction`, либо компонент `extension`, атрибутом `base` задающий тип (простой) тела описываемого элемента.

В компоненте `extension` указываются атрибуты открывающего тега описываемого элемента. Все вместе выглядит так, как в следующем примере:

```
<xsd:complexType name="calcResultType">
  <xsd:simpleContent>
```

```

    <xsd:extension base="xsd:decimal">
      <xsd:attribute name="unit" type="xsd:string" />
      <xsd:attribute name="precision"
        type="xsd:nonNegativeInteger" />
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

```

Эту конструкцию можно описать словами так: "Определяется тип `calcResultType` элемента, тело которого содержит значения встроенного простого типа `xsd:decimal`. Простой тип расширяется тем, что к нему добавляются атрибуты `unit` и `precision`".

Если в схеме объявить элемент `result` этого типа следующим образом:

```
<xsd:element name="result" type="calcResultType" />
```

то в документе XML можно написать:

```
<result unit="см" precision="2">123.25</result>
```

В компоненте `restriction`, кроме атрибутов, описывается простой тип содержимого элемента и/или фасетки, ограничивающие тип, заданный атрибутом `base`. Например:

```

<xsd:complexType name="calcResultType">
  <xsd:simpleContent>
    <xsd:restriction base="xsd:decimal">
      <xsd:totalDigits value="8" />
      <xsd:attribute name="unit" type="xsd:string" />
      <xsd:attribute name="precision"
        type="xsd:nonNegativeInteger" />
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

```

Упражнение

Определите тип элемента `city` из листинга 1.2.

Определение типа вложенных элементов

Если значениями определяемого сложного типа будут элементы, содержащие вложенные элементы, как, например, элементы `address`, `phone-list`

листинга 1.2, то перед тем, как перечислять их описания, надо выбрать *модель группы* (model group) вложенных элементов. Дело в том, что вложенные элементы, составляющие определяемый тип, могут появляться или в определенном порядке, или в произвольном порядке, кроме того, можно выбирать только один из перечисленных элементов. Эта возможность и называется *моделью группы* элементов. Она определяется одним из трех компонентов: sequence, all или choice.

Компонент sequence применяется в том случае, когда перечисляемые элементы должны записываться в документе в определенном порядке. Пусть, например, мы описываем книгу. Сначала определяем тип:

```
<xsd:complexType name="bookType">
  <xsd:sequence maxOccurs="unbounded">
    <xsd:element name="author" type="xsd:normalizedString"
      minOccurs="0" />
    <xsd:element name="title" type="xsd:normalizedString" />
    <xsd:element name="pages" type="xsd:positiveInteger"
      minOccurs="0" />
    <xsd:element name="publisher" type="xsd:normalizedString"
      minOccurs="0" />
  </xsd:sequence>
</xsd:complexType>
```

Потом описываем элемент:

```
<xsd:element name="book" type="bookType" />
```

Элементы author, title, pages и publisher должны входить в элемент book именно в таком порядке. В документе XML надо писать:

```
<book>
  <author>И. Ильф, Е. Петров</author>
  <title>Золотой теленок</title>
  <publisher>Детская литература</publisher>
</book>
```

Если же вместо компонента xsd:sequence записать компонент xsd:all, то элементы author, title, pages и publisher можно перечислять в любом порядке.

Компонент choice применяется в том случае, когда надо выбрать один из нескольких элементов. Например, при описании журнала вместо издатель-

ства, описываемого элементом `publisher`, надо записать название журнала. Это можно определить так:

```
<xsd:complexType name="bookType">

  <xsd:sequence maxOccurs="unbounded">

    <xsd:element name="author" type="xsd:normalizedString"
      minOccurs="0" />

    <xsd:element name="title" type="xsd:normalizedString" />

    <xsd:element name="pages" type="xsd:positiveInteger"
      minOccurs="0" />

    <xsd:choice>

      <xsd:element name="publisher" type="xsd:normalizedString"
        minOccurs="0" />

      <xsd:element name="magazine" type="xsd:normalizedString"
        minOccurs="0" />

    </xsd:choice>

  </xsd:sequence>

</xsd:complexType>
```

Как видно из этого примера, компонент `choice` можно вложить в компонент `sequence` или, наоборот, вложить компонент `sequence` в компонент `choice`. Такие вложения можно проделать сколько угодно раз. Кроме того, каждая группа в этих моделях может появиться сколько угодно раз, т. е. в компоненте `choice` тоже можно записать атрибут `maxOccurs="unbounded"`.

Модель группы `all` отличается в этом от моделей `sequence` и `choice`. В компоненте `all` не допускается использование компонентов `sequence` и `choice`. Аналогично, в компонентах `sequence` и `choice` нельзя применять компонент `all`. Каждый элемент, входящий в группу модели `all`, может появиться не более одного раза, т. е. атрибут `maxOccurs` этого элемента может равняться только единице.

Упражнения

1. Определите тип элемента `address` из листинга 1.2.
2. Определите тип элемента `phone-list` из того же листинга.

Определение типа со сложным телом

При определении сложного типа можно воспользоваться уже определенным, *базовым*, сложным типом, расширив его дополнительными элементами, или, наоборот, удалив из него некоторые элементы. Для этого необходимо применить компонент `complexContent`. В этом компоненте, так же как и в компоненте `simpleContent`, записывается либо компонент `extension`, если надо расширить базовый тип, либо компонент `restriction`, если нужно его сузить. Базовый тип указывается атрибутом `base`, так же как и при записи компонента `simpleContent`, но теперь это должен быть сложный, а не простой тип!

Расширим, например, определенный выше тип `bookType`, добавив год издания — элемент `year`:

```
<xsd:complexType name="newBookType">

  <xsd:complexContent>

    <xsd:extension base="bookType">

      <xsd:sequence>
        <xsd:element name="year" type="xsd:gYear">
        </xsd:sequence>

      </xsd:extension>

    </xsd:complexContent>

  </xsd:complexType>
```

При сужении базового типа компонентом `restriction` надо перечислить те элементы, которые останутся после сужения. Например, оставим в типе `newbookType` только автора и название книги из типа `bookType`:

```
<xsd:complexType name="newBookType">

  <xsd:complexContent>

    <xsd:restriction base="bookType">

      <xsd:sequence>
        <xsd:element name="author" type="xsd:normalizedString"
          minOccurs="0" />
        <xsd:element name="title" type="xsd:normalizedString" />
      </xsd:sequence>

    </xsd:restriction>

  </xsd:complexType>
```

```

    </xsd:restriction>

    </xsd:complexContent>

</xsd:complexType>

```

Это описание выглядит странно. Почему надо заново объявлять все элементы, остающиеся после сужения? Не проще ли определить новый тип?

Дело в том, что в язык XSD внесены элементы объектно-ориентированного программирования, которых мы не будем касаться. Расширенный и суженный типы связаны со своим базовым типом отношением наследования, и к ним можно применить операцию подстановки. У всех типов языка XSD есть общий предок — базовый тип `anyType`. От него наследуются все сложные типы. Это подобно тому, как у всех классов Java есть общий предок — класс `Object`, а все массивы наследуются от него. От базового типа `anyType` наследуется и тип `anySimpleType` — общий предок всех простых типов.

Таким образом, сложные типы определяются как сужение типа `anyType`. Если строго подходить к определению сложного типа, то определение типа `bookType`, сделанное в начале предыдущего раздела, надо записать так:

```

<xsd:complexType name="bookType">

    <xsd:complexContent>

        <xsd:restriction base="xsd:anyType">

            <xsd:sequence maxOccurs="unbounded">

                <xsd:element name="author" type="xsd:normalizedString"
                    minOccurs="0" />

                <xsd:element name="title" type="xsd:normalizedString" />

                <xsd:element name="pages" type="xsd:positiveInteger"
                    minOccurs="0" />

                <xsd:element name="publisher" type="xsd:normalizedString"
                    minOccurs="0" />

            </xsd:sequence>

        </xsd:restriction>

    </xsd:complexContent>

</xsd:complexType>

```

Рекомендация языка XSD позволяет сократить эту запись, что мы и сделали в предыдущем разделе. Это подобно тому, как в Java мы опускаем слова "extends Object" в заголовке описания класса.

Закончим на этом описание языка XSD и перейдем к примерам.

Пример: схема адресной книги

В листинге 3.1 записана схема документа, приведенного в листинге 1.2.

Листинг 3.1. Схема XSD записной книжки

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="notebook" type="notebookType" />

  <xsd:complexType name="notebookType">

    <xsd:element name="person" type="personType"
      minOccurs="0" maxOccurs="unbounded" />

  </xsd:complexType>

  <xsd:complexType name="personType">
    <xsd:sequence>

      <xsd:element name="name">

        <xsd:complexType>
          <xsd:attribute name="first" type="xsd:string"
            use="optional" />
          <xsd:attribute name="second" type="xsd:string"
            use="optional" />
          <xsd:attribute name="surname" type="xsd:string"
            use="required" />
        </xsd:complexType>

      </xsd:element>

      <xsd:element name="birthday" type="xsd:date" minOccurs="0" />

      <xsd:element name="address" type="addressType"
        minOccurs="0" maxOccurs="unbounded" />

      <xsd:element name="phone-list" type="phone-listType"
        minOccurs="0" />

    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

```

</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="addressType" >
<xsd:sequence>

  <xsd:element name="street" type="xsd:string" />
  <xsd:element name="city" type="cityType" />
  <xsd:element name="zip" type="xsd:positiveInteger" />

</xsd:sequence>
</xsd:complexType>

<xsd:complexType name='cityType'>

  <xsd:simpleContent>
    <xsd:extension base='xsd:string' >
      <xsd:attribute name='type' type='placeType'
        default='город' />
    </xsd:extension>
  </xsd:simpleContent>

</xsd:complexType>

<xsd:simpleType name="placeType">

  <xsd:restriction base = "xsd:string">
    <xsd:enumeration value="город" />
    <xsd:enumeration value="поселок" />
    <xsd:enumeration value="деревня" />
  </xsd:restriction>

</xsd:simpleType>

<xsd:complexType name="phone-listType">

  <xsd:element name="work-phone" type="xsd:string"
    minOccurs="0" maxOccurs="unbounded" />
  <xsd:element name="home-phone" type="xsd:string"
    minOccurs="0" maxOccurs="unbounded" />

</xsd:complexType>

<xsd:simpleType name="ruDate">

  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{2}.[0-9]{2}.[0-9]{4}" />
  </xsd:restriction>

</xsd:simpleType>

</xsd:schema>

```

Листинг 3.1, как обычный документ XML, начинается с пролога, показывающего версию XML и определяющего стандартное пространство имен схемы XML с идентификатором `http://www.w3.org/2001/XMLSchema`. С этим идентификатором связан префикс `xsd`. Конечно, префикс может быть другим, часто пишут префикс `xs`.

Все описание схемы нашей адресной книжки заключено в третьей строке, в которой указано, что адресная книга состоит из одного элемента с именем `notebook`, имеющего сложный тип `notebookType`. Этот элемент должен появиться в документе ровно один раз. Остальная часть листинга 3.1 посвящена описанию типа этого элемента и других типов.

Определение сложного типа `notebookType` несложно (простите за каламбур). Оно занимает три строки листинга, не считая открывающего и закрывающего тега, и просто говорит о том, что данный тип составляют несколько элементов `person` типа `personType`.

Определение типа `personType` немногим сложнее. Оно означает, что этот тип составляют четыре элемента: `name`, `birthday`, `address` и `phone-list`. Для элемента `name` сразу же объявлены необязательные атрибуты `first` и `second` простого типа `string`, определенного в пространстве имен `xsd`. Тип обязательного атрибута `surname` — тоже `string`.

Далее в листинге 3.1 определяются оставшиеся типы `addressType`, `phone-listType` и `ruDate`. Необходимость определения простого типа `ruDate` возникает потому, что встроенный в схему XML тип `date` предписывает записывать дату в виде 2003-02-22, а в России принят формат 22.02.2003. Тип `ruDate` определяется как сужение (`restriction`) типа `string` с помощью шаблона. Шаблон (`pattern`) для записи даты в виде ДД.ММ.ГГГГ задается регулярным выражением.

Безымянные типы

Все описанные в листинге 3.1 типы используются только один раз. Поэтому необязательно давать типу имя. Схема XML, как говорилось выше, позволяет определять безымянные типы. Такое определение дается внутри описания элемента. Именно так в листинге 3.1 описаны атрибуты элемента `name`. В листинге 3.2 показано упрощенное описание схемы адресной книги.

Листинг 3.2. Схема документа XML с безымянными типами

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <xsd:element name='notebook'>
    <xsd:complexType>
      <xsd:sequence>
```

```
<xsd:element name='person' maxOccurs='unbounded'>
<xsd:complexType>
<xsd:sequence>

  <xsd:element name='name'>
    <xsd:complexType>

      <xsd:attribute name='first'
        type='xsd:string' use='optional' />

      <xsd:attribute name='second'
        type='xsd:string' use='optional' />

      <xsd:attribute name='surname'
        type='xsd:string' use='required' />

    </xsd:complexType>
  </xsd:element>

  <xsd:element name='birthday'>
    <xsd:simpleType>

      <xsd:restriction base='xsd:string'>
        <xsd:pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
      </xsd:restriction>

    </xsd:simpleType>
  </xsd:element>

  <xsd:element name='address' maxOccurs='unbounded'>
    <xsd:complexType>
    <xsd:sequence>

      <xsd:element name='street' type='xsd:string' />
      <xsd:element name='city'>
        <xsd:complexType>
        <xsd:simpleContent>

          <xsd:extension base='xsd:string'>
            <xsd:attribute name='type' type='xsd:string'
              use='optional' default='gorod' />
          </xsd:extension>

        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
```



```
<xsd:element name='zip' type='xsd:positiveInteger' />

</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name='phone-list'>
  <xsd:complexType>
    <xsd:sequence>

      <xsd:element name='work-phone' type='xsd:string'
        minOccurs='0' maxOccurs='unbounded' />

      <xsd:element name='home-phone' type='xsd:string'
        minOccurs='0' maxOccurs='unbounded' />

    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
```

Еще одно упрощение можно сделать, используя пространство имен по умолчанию. Посмотрим, какие пространства имен применяются в схемах XML.

Пространства имен языка XSD

Имена элементов и атрибутов, используемые при записи схем, определены в пространстве имен с идентификатором <http://www.w3.org/2001/XMLSchema>. Префикс имен, относящихся к этому пространству, часто называют *xs* или *xsd*, как в листингах 3.1 и 3.2. Каждый анализатор "знает" это пространство имен и "понимает" имена из этого пространства.

Можно сделать это пространство имен пространством по умолчанию, но тогда надо задать пространство имен для определяемых в схеме типов и элементов. Для удобства такого определения введено понятие *целевого про-*

пространства имен (target namespace). Идентификатор целевого пространства имен определяется атрибутом targetNamespace, например:

```
<xsd:schema targetNamespace="http://some.firm.com/2003/ntbNames">
```

После такого определения имени, определяемые в этой схеме, будут относиться к новому пространству имен с идентификатором `http://some.firm.com/2003/ntbNames`. Так сделано в листинге 3.3. Для упрощения записи в нем стандартное пространство имен схемы XML с идентификатором `http://www.w3.org/2001/XMLSchema` сделано пространством имен по умолчанию. Имена, относящиеся к целевому пространству, снабжены префиксом `ntb`, чтобы они не попали в пространство имен по умолчанию.

Листинг 3.3. Схема документа XML с целевым пространством имен

```
<?xml version='1.0'?>
<schema xmlns='http://www.w3.org/2001/XMLSchema'
  targetNamespace='http://some.firm.com/2003/ntbNames'
  xmlns:ntb='http://some.firm.com/2003/ntbNames'>

<element name='ntb:notebook'>

  <complexType>
    <sequence>

      <element name='person' maxOccurs='unbounded'>
        <complexType>
          <sequence>

            <element name='name'>
              <complexType>

                <attribute name='first'
                  type='string' use='optional' />

                <attribute name='second'
                  type='string' use='optional' />

                <attribute name='surname'
                  type='string' use='required' />

              </complexType>
            </element>

            <element name='birthday'>
              <simpleType>

                <restriction base='string'>
                  <pattern value='[0-9]{2}.[0-9]{2}.[0-9]{4}' />
                </restriction>
              </simpleType>
            </element>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
```

```

        </simpleType>
      </element>

      <element name='address' maxOccurs='unbounded'>
        <complexType>
          <sequence>

            <element name='street' type='string' />
            <element name='city' type='string' />
            <element name='zip' type='positiveInteger' />

          </sequence>
        </complexType>
      </element>

      <element name='phone-list'>
        <complexType>
          <sequence>

            <element name='work-phone' type='string'
              minOccurs='0' maxOccurs='unbounded' />

            <element name='home-phone' type='string'
              minOccurs='0' maxOccurs='unbounded' />

          </sequence>
        </complexType>
      </element>

    </sequence>
  </complexType>
</element>

</sequence>
</complexType>
</element>

</sequence>
</complexType>
</element>

</schema>

```

Поскольку в листинге 3.3 пространством имен по умолчанию сделано пространство `http://www.w3.org/2001/XMLSchema`, префикс `xsd` не нужен.

Следует заметить, что в целевое пространство имен попадают только *глобальные имена*, чьи описания непосредственно вложены в корневой элемент `schema`. Это естественно, потому что только глобальными именами можно воспользоваться далее в этой или другой схеме. В листинге 3.3 лишь одно глобальное имя `notebook`. Вложенные имена `name`, `address` и др. только *ассоциированы* с глобальными именами.

В схемах и документах XML часто применяется еще одно стандартное пространство имен. Рекомендация языка XSD определяет несколько атрибутов: `type`, `nil`, `schemaLocation`, `noNamespaceSchemaLocation`, которые применяются не только в схемах, а и непосредственно в описываемых этими схемами документах XML, называемых *экземплярами схем* (XML schema instance). Имена этих атрибутов относятся к пространству имен `http://www.w3.org/2001/XMLSchema-instance`. Этому пространству имен чаще всего приписывают префикс `xsi`, например:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Включение файлов схемы в другую схему

В создаваемую схему можно включить файлы, содержащие другие схемы. Для этого есть два элемента схемы: `include` и `import`. Например:

```
<xsd:include xsi:schemaLocation="names.xsd" />
```

Включаемый файл задается атрибутом `xsi:schemaLocation`. В примере он использован для того, чтобы включить в создаваемую схему содержимое файла `names.xsd`. Файл должен содержать схему с описаниями и определениями из того же пространства имен, что и в создаваемой схеме, или без пространства имен, т. е. в нем не использован атрибут `targetNamespace`. Это удобно, если мы хотим добавить к создаваемой схеме определения схемы `names.xsd` или просто разбить большую схему на два файла. Можно представить себе результат включения так, как будто содержимое файла `names.xsd` всего лишь записано на месте элемента `include`.

Перед включением файла можно изменить некоторые определения, приведенные в нем. Для этого используется элемент `redefine`, например:

```
<xsd:redefine schemaLocation="names.xsd">

  <xsd:simpleType name="nameType">

    <xsd:restriction base="xsd:string">
      <xsd:maxLength value="40"/>
    </xsd:restriction>

  </xsd:simpleType>

</xsd:redefine>
```

Если же включаемый файл содержит имена из другого пространства имен, то надо воспользоваться элементом схемы `import`. Например, пусть файл `A.xsd` начинается со следующих определений:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://some.firm.com/someNames">
```

а файл B.xsd начинается с определений:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://some.firm.com/anotherNames">
```

Мы решили включить эти файлы в новый файл C.xsd. Это делается так:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://some.firm.com/yetAnotherNames"
  xmlns:pr1="http://some.firm.com/someNames"
  xmlns:pr2="http://some.firm.com/anotherNames">

<xsd:import namespace="http://some.firm.com/someNames"
  xsi:schemaLocation="A.xsd" />

<xsd:import namespace="http://some.firm.com/anotherNames"
  xsi:schemaLocation="B.xsd" />
```

После этого в файле C.xsd можно использовать имена, определенные в файлах A.xsd и B.xsd, снабжая их префиксами pr1 и pr2 соответственно.

Элементы include и import следует располагать перед всеми определениями схемы.

Значение атрибута xsi:schemaLocation — строка URI, поэтому файл с включаемой схемой может располагаться в любом месте Интернета.

Связь документа XML со своей схемой

Программе-анализатору, проверяющей соответствие документа XML его схеме, надо как-то указать файлы (один или несколько), содержащие схему документа. Это можно сделать разными способами. Во-первых, можно подать эти файлы на вход анализатора. Так делает, например, проверяющий анализатор XSV (XML Schema Validator) (<ftp://ftp.cogsci.ed.ac.uk/pub/XSV/>):

```
$ xsv ntb.xml ntb1.xsd ntb2.xsd
```

Во-вторых, можно задать файлы со схемой как свойство анализатора, устанавливаемое методом `setProperty()`, или значение переменной окружения анализатора. Так делает, например, проверяющий анализатор Xerces.

Эти способы удобны тогда, когда документ в разных случаях нужно связать с различными схемами. Если же схема документа фиксирована, то ее

удобнее указать прямо в документе XML. Это делается одним из двух способов:

- ❑ Если элементы документа не принадлежат никакому пространству имен и записаны без префикса, то в корневом элементе документа записывается атрибут `noNamespaceSchemaLocation`, указывающий расположение файла со схемой в форме URI:

```
<notebook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ntb.xsd">
```

В этом случае в схеме не должно быть целевого пространства имен, т. е. не следует использовать атрибут `targetNamespace`.

- ❑ Если же элементы документа относятся к некоторому пространству имен, то применяется атрибут `schemaLocation`, в котором через пробел перечисляются пространство имен и расположение файла со схемой, описывающей это пространство имен. Продолжая пример предыдущего раздела, можно написать:

```
<notebook xmlns="http://some.firm.com/2003/ntbNames"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://some.firm.com/someNames A.xsd
    http://some.firm.com/anotherNames B.xsd"
  xmlns:pr1="http://some.firm.com/someNames"
  xmlns:pr2="http://some.firm.com/anotherNames">
```

После этого в документе можно использовать имена, определенные в схемах A.xsd и B.xsd, снабжая их префиксами `pr1` и `pr2` соответственно.

Другие языки описания схем

Даже из приведенного выше краткого описания языка XSD видно, что он получился весьма сложным и запутанным. Есть уже несколько книг, полностью посвященных этому языку. Их объем ничуть не меньше объема всей этой книги.

Есть и другие, более простые языки описания схемы документа XML. Наибольшее распространение получили следующие из них:

- ❑ Schematron — <http://www.ascc.net/xml/resource/schematron/>;
- ❑ RELAX NG — Regular Language Description for XML, New Generation, <http://www.oasis-open.org/committees/relax-ng/>, этот язык возник как слияние языков Relax и TREX;
- ❑ Relax — <http://www.xml.gr.jp/relax/>;

- ❑ TREX — Tree Regular Expressions for XML,
<http://www.thaiopensource.com/trex/>;
- ❑ DDML — Document Definition Markup Language, известный еще как XSchema, <http://purl.oclc.org/NET/ddml/>.

Менее распространены языки DCD (Document Content Description), SOX (One's Schema for Object-Oriented XML), XDR (XML-Data Reduced).

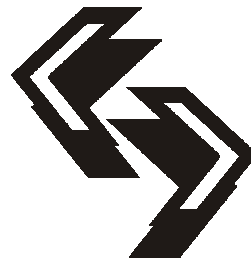
Хороший обзор языков описания схем представлен на странице <http://www.oasis-open.org/cover/schemas.html>.

Все эти языки позволяют более или менее полно описывать схему документа. Возможно, они вытеснят язык XSD или будут существовать совместно.

Вопросы для самопроверки

1. Что такое схема документа?
2. Зачем надо описывать схему документа?
3. Чем не устраивает разработчиков описание DTD?
4. Чем вызвано введение такого большого числа типов?
5. Как можно определить свои собственные простые типы?
6. Есть ли в языке XSD предопределенные сложные типы?
7. Что такое "фасетка"?
8. Можно ли определить типы в одном файле, а использовать их в другом?
9. Как можно включить содержимое одного файла с описанием схемы документа в другой файл?
10. Какие пространства имен использует язык XSD?

ГЛАВА 4



Создание ссылок на языке XLink

Основная особенность языка HTML, благодаря которой он завоевал огромную популярность, — это возможность создавать *гиперссылки* (hyper references). Простым щелчком мыши по тексту, выделенному голубым цветом, мы переносимся в другой документ, может быть, находящийся на другом конце света. Еще щелчок, и мы уже связаны с иным хостом, расположенным совсем в другом месте нашей планеты. Мы не знаем точно, где находится этот хост, да и знать не хотим. Нам важно лишь то, что мы можем за несколько минут собрать информацию из самых разных источников.

Все эти богатейшие возможности обеспечиваются одним тегом `<a>` вида:

```
<a href="http://host.domain/newdoc.html">Новый документ</a>
```

Мы только указываем атрибутом `href` адрес нового документа, записанный в форме URI. Все остальное берет на себя система World Wide Web. Она отыскивает указанный в адресе хост, устанавливает связь с Web-сервером, работающим на этом хосте, передает ему запрос, дожидается ответа и отображает в окне нашего браузера полученный от сервера документ `newdoc.html`.

Этот документ появится на экране вместо того, с которым мы работали. Если мы хотим увидеть документ в новом окне, то добавляем атрибут `target`, содержащий имя окна или константы `_BLANK`, `_PARENT`, `_SELF`, `_TOP`, указывающие, куда поместить новый документ — в пустое окно, в родительское, в то же самое окно или в основное окно браузера.

В языке HTML есть возможность загружать не только файлы, но и изображения. Для этого применяется тег ``. В нем атрибутом `src` записывается адрес URI файла с изображением:

```

```

После этого изображение появляется на том месте, где располагался тег ``.

Язык XML, разумеется, не мог не предоставить возможность создавать подобные ссылки на другие документы, изображения или на какие-то иные места того же самого документа. В 1999 году консорциумом W3C был придуман специальный язык создания ссылок XLink — одна из реализаций XML. Язык быстро развивался, и в 2001 году была выпущена его первая версия. Она оформлена в виде рекомендации "XML Linking Language (XLink)", последнюю версию которой всегда можно посмотреть по адресу <http://www.w3.org/TR/xlink/>.

Гиперссылки языка HTML скованы многими условиями, к числу которых относятся следующие ограничения:

- ☐ гиперссылку можно создать только в текущем документе;
- ☐ сослаться можно только на один документ;
- ☐ гиперссылка создается только одним тегом `<a>`;
- ☐ у тега `<a>` фиксированный набор атрибутов, его нельзя изменить;
- ☐ гиперссылку может активизировать только пользователь, но не программа, обрабатывающая документ.

Разработчики языка XLink постарались снять эти и другие ограничения. Язык XLink позволяет создать ссылку в одном, а использовать в других документах. Ссылка может указывать сразу на несколько документов. Сослаться можно не только на документ XML, но и на любой информационный ресурс: изображение, чертеж, программу. Отпадает необходимость в специализированных тегах, таких как тег ``. Можно организовать ссылку, связывающую другие документы, например, ссылка, записанная в документе `doc1.xml`, может установить связь между документом `doc2.xml` и документом `doc3.xml`. Кроме того, язык XLink отмечает направление ссылки и позволяет организовать обратные ссылки. Эти возможности делают язык XLink чрезвычайно мощным, способным удовлетворить нужды самого привередливого разработчика.

Перейдем к рассмотрению структуры языка XLink.

Пространство имен языка XLink

Интересная особенность языка XLink заключается в том, что он не вводит новые элементы, а определяет только атрибуты, которые можно использовать в любых определяемых вами элементах. Каждый элемент в документе XML, использующий атрибуты языка XLink, становится ссылкой. Атрибуты, введенные языком XLink, находятся в пространстве имен <http://www.w3.org/1999/xlink>. Как обычно, перед использованием атрибутов надо связать это пространство имен с каким-либо префиксом. Очень часто этот префикс называют `xlink:`

```
<someElement xmlns:xlink="http://www.w3.org/1999/xlink">
    Содержимое элемента
</someElement>
```

Всего в языке XLink объявлено десять атрибутов:

- ❑ атрибут `type` задает тип ссылки;
- ❑ атрибут `href` описывает адрес ресурса, с которым связана ссылка;
- ❑ атрибут `show` определяет способ показа полученного по ссылке ресурса;
- ❑ атрибут `actuate` устанавливает момент активизации ссылки;
- ❑ атрибуты `label`, `from`, `to` отмечают и указывают начальные и конечные пункты ссылки;
- ❑ атрибуты `role`, `arcrole`, `title` объясняют смысл ссылки.

Разумеется, кроме атрибутов языка XLink в объявляемых вами элементах-ссылках можно объявлять и любые другие атрибуты.

Рассмотрим подробнее каждый из атрибутов языка XLink.

Атрибут *title*

Атрибут `title` может встретиться в ссылке любого типа, за исключением ссылки типа `title`, в которой он просто будет лишним. Значением атрибута `title` может быть любая строка символов. Она предназначена для человека и может быть выведена на экран дисплея, записана в журнальный файл сообщений или распечатана на принтере. Обычно эта строка играет роль комментария или информационного сообщения, поясняя действие ссылки или отмечая способ ее активизации. В отличие от обычного комментария атрибут `title` обрабатывается программой-обработчиком документа XML. Например, значение атрибута `title` можно использовать в подсказке, всплывающей при наведении на ссылку курсора мыши.

Листинг 4.1 показывает пример ссылки с атрибутом `title`.

Листинг 4.1. Пример ссылки типа *simple*

```
<someLnk xmlns:xlink="http://www.w3.org/1999/xlink"
    xlink:type="simple"
    xlink:title="Загружается классификатор"
    xlink:href="http://some.domain.com/pub/ind/kldr012.xml">
    Классификатор адресов
</someLnk>
```

В этом примере показана ссылка на документ `kldr012.xml`, по своему действию аналогичная гиперссылке языка HTML. Первый атрибут элемента `someLink` определяет префикс `xlink` пространства имен языка XLink. Вторым атрибут `xlink:type` показывает тип ссылки. В третьем атрибуте `xlink:title` записано информационное сообщение. Четвертый атрибут `xlink:href` задает адрес документа `kldr012.xml`.

Атрибут *label*

Кроме атрибута `title` в языке XLink есть атрибут `label` с похожим назначением — пометить ссылку. В отличие от атрибута `title` значение атрибута `label` может быть только простым именем без специальных знаков, пробелов, двоеточий, точнее, значением типа `NCName`. Это имя предназначено для того, чтобы к элементу можно было обратиться из других элементов документа, а именно из элементов-ссылок типа `arc`. Поэтому атрибут типа `label` применяется только в ссылках типа `locator` или `resource`, чтобы пометить их для дальнейшего использования ссылками типа `arc`. В листинге 4.2 приведен пример такой конструкции.

Листинг 4.2. Метки и их использование

```
<multLink xmlns:xlink="http://www.w3.org/1999/xlink"
          xlink:type="extended"
          xlink:title="Связь документов">

  <src xlink:type="resource" xlink:label="s0012" />

  <tgt xlink:type="locator"
        xlink:href="http://domain.com/docs/udr01.xml"
        xlink:label="t0012" />

  <ref xlink:type="arc" from="s0012" to="t0012" />

</multLink>
```

Листинг 4.2 показывает элемент `multLink` — сложную ссылку типа `extended`. Сложные ссылки содержат в себе другие ссылки. В частности, в элементе `multLink` содержатся три ссылки.

Элемент-ссылка `src` типа `resource` помечен меткой `s0012`. Он ссылается на текущий документ и используется в элементе `ref` как начальная точка определяемой ссылки, поскольку его метка `s0012` записана в атрибуте `from`.

Элемент-ссылка `tgt` с меткой `t0012` используется как конечная точка ссылки, т. к. эта метка присутствует в атрибуте `to` элемента `ref`. Начальной

точкой ссылки становится текущий документ, а конечной — документ `udr01.xml`.

Если мы запишем элемент `ref` по-другому:

```
<ref xlink:type="arc" from="t0012" to="s0012" />
```

то получим обратную ссылку документа `udr01.xml` на текущий документ.

Атрибут *href*

Атрибут `href`, значением которого служит адрес ресурса в форме URI, встречается только в ссылках типа `simple` и `locator`, как показано в листингах 4.1 и 4.2, причем для ссылки типа `locator` он обязателен. Его можно не записывать в элементе-ссылке типа `locator` явно, но тогда у него должно быть значение по умолчанию, определенное при описании схемы документа. Как видно из предыдущих примеров, атрибут `href` используется точно так же, как соответствующий атрибут в языке HTML.

Атрибут *type*

Основной атрибут, определенный в языке XLink, — это атрибут `type`. Он обязателен для всех ссылок. Больше того, именно он определяет элемент как ссылку. Следующий пример показывает, как можно оформить ссылку в духе гиперссылки языка HTML:

```
<myLink xmlns:xlink="http://www.w3.org/1999/xlink"
        xlink:title="Дополнительные сведения"
        xlink:type="simple"
        xlink:href="http://host.domain/newdoc.html">
```

Новая версия документа

```
</myLink>
```

Первый атрибут `xmlns:xlink` элемента `myLink` определяет префикс `xlink` имен языка XLink. Второй атрибут `xlink:type` указывает, что элемент `myLink` — ссылка типа `simple`. Третий атрибут `xlink:href` задает адрес URI документа, на который делается ссылка.

Примечание

В дальнейшем я не всегда буду записывать в элементах атрибут `xmlns`, считая, что префикс имен атрибутов языка XLink уже определен и равен `xlink`.

Хотя атрибут `type` обязателен, можно сделать его присутствие неявным, задав ему значение по умолчанию при описании схемы документа. Например,

при описании структуры документа XML элемент `myLink` из предыдущего примера можно объявить на языке XSD следующим образом:

```
<xsd:element name="myLink" type="xsd:string">

  <xsd:complexType>

    <xsd:attribute name="xlink:type" type="xsd:name"
      use="optional" default="simple" />

    <xsd:attribute name="xlink:href" type="xsd:anyURI"
      use="required" />

  </xsd:complexType>

</xsd:element>
```

После такого объявления атрибут `xlink:type` можно не записывать в элементе `myLink`, при этом значением атрибута и, следовательно, типом элемента `myLink` будет тип `simple`.

Типы ссылок

Как видно из предыдущих примеров, ссылки бывают разных типов. Во-первых, ссылка может быть *простой* (`simple`) или *расширенной* (`extended`). Во-вторых, ссылка может быть информационным ресурсом (`resource`), указателем на ресурс (`locator`), дугой графа (`arc`) или просто заголовком (`title`). Кроме того, значением атрибута `type` может служить слово `none`, означающее, что остальные атрибуты и содержимое элемента не имеют никакого отношения к ссылкам.

Рассмотрим подробнее каждый из этих семи типов. Для полноты описания включим в него и тип `none`.

Тип *none*

Указание в качестве значения атрибута `xlink:type` слова `none` показывает, что элемент вовсе не является ссылкой, несмотря на то, что другие атрибуты и его содержимое могут выглядеть так, как будто относятся к языку XLink. Это полезно, если в каких-то ситуациях элемент надо использовать как ссылку, а в других ситуациях — нет. Вот, например, определение простого типа, сделанное на языке XSD. Определяемый тип состоит из двух значений: `"simple"` и `"none"`.

```
<xsd:simpleType name="locType">

  <xsd:restriction base="xsd:name">
```

```

        <xsd:enumeration value="simple" />
        <xsd:enumeration value="none" />

    </xsd:restriction>

</xsd:simpleType>

```

Далее идет объявление элемента `anElem` с атрибутом `xlink:type` только что определенного типа `locType`. Такой атрибут принимает значения "simple" или "none".

```

<xsd:element name="anElem" type="xsd:string">

    <xsd:complexType>
        <xsd:attribute name="xlink:type" type="locType"
            use="required" />
        <xsd:attribute name="xlink:href" type="xsd:anyURI"
            use="optional" />
    </xsd:complexType>

</xsd:element>

```

После этого объявления элемент `anElem` можно использовать в документах XML как ссылку:

```

<anElem xlink:type="simple" xlink:href="f1.xml">

    Ссылка на описание

</anElem>

```

или как простой элемент:

```

<anElem xlink:type="none">

    Простое описание

</anElem>

```

Тип *locator*

Ссылка типа `locator` описывает удаленный информационный ресурс. Описание выполняется обязательным атрибутом `href`, записанным явно или имеющим значение по умолчанию. Значением атрибута `href` служит адрес URI описываемого информационного ресурса точно так же, как это делалось в гиперссылке языка HTML. Например:

```
<myBiogr xlink:type="locator"
          xlink:href="http://hsh.com/names/ivanov.xml"
          xlink:label="L1234" />
```

Описание удаленного ресурса используется затем при создании ссылки-дуги типа `arc`.

Вы уже видели в предыдущих примерах, что ссылку на удаленный ресурс можно задать просто атрибутом `href` элемента-ссылки типа `simple`. Зачем же в язык XLink введен еще один способ указания удаленного ресурса?

Дело в том, что ссылка типа `locator` не применяется самостоятельно, а только как один из элементов сложной ссылки типа `extended`. Пример такого использования приведен в листинге 4.2. Если элемент типа `locator` не записан внутри сложной ссылки типа `extended`, то он не будет обрабатываться как ссылка, а будет считаться просто неким элементом XML.

Кроме обязательного атрибута `href`, элементы типа `locator` могут содержать атрибуты `role`, `title` и `label`, а также любые атрибуты, определенные вами. У элементов типа `locator` может быть любое тело, в том числе и вложенные элементы. Вложенные элементы не будут обрабатываться как ссылки.

Тип *simple*

Ссылки типа `simple` более всего похожи на тег `<a>` языка HTML. Они связывают один документ, тот, в котором определена ссылка, с другим документом или другой частью того же самого документа. Таким образом, связь устанавливается только между двумя документами, направление связи идет только от текущего документа, в котором записана ссылка типа `simple`, к удаленному ресурсу, адрес URI которого указан в атрибуте `href`.

В отличие от ссылки типа `locator`, в ссылке типа `simple` атрибут `href` необязателен. В случае его полного отсутствия, если у него нет даже значения по умолчанию, ссылка типа `simple` просто описывает информационный ресурс подобно ссылке типа `resource`. Пример ссылки типа `simple` приведен в листинге 4.1.

Кроме необязательного атрибута `href`, в элементах-ссылках типа `simple` можно использовать атрибуты `role`, `arcrole`, `title`, `show`, `actuate` и атрибуты, не относящиеся к языку XLink. У элемента типа `simple` может быть любое содержимое, но вложенные элементы не будут рассматриваться как ссылки.

Упражнение

Посмотрите на текст в формате HTML просматриваемых вами Web-страниц и перепишите теги `<a>` и `` в виде ссылок языка XLink типа `simple`.

Тип *extended*

Ссылка типа `extended`, пример которой уже приведен в листинге 4.2, может связывать любое число документов и других информационных ресурсов. Некоторые из этих связей могут быть входящими, другие — выходящими, третьи могут связывать сторонние документы. Описание таких связей производится элементами, вложенными в элемент-ссылку типа `extended`. Эти вложенные элементы — тоже ссылки, их типы — `title`, `resource`, `locator` или `arc`. Ссылка типа `locator` нам уже известна, рассмотрим другие типы, используемые в сложных ссылках типа `extended`.

Упражнение

Перепишите ссылки, созданные в предыдущем упражнении, в виде ссылок типа `extended` с вложенной ссылкой типа `locator`.

Тип *title*

Самый простой тип ссылки — тип `title`. Атрибут `xlink:type` — это единственный обязательный атрибут языка XLink в такой ссылке. Более того, кроме него в ссылке типа `title` нет ни одного атрибута языка XLink, хотя могут быть атрибуты из другого пространства имен. Ссылка типа `title` ни на что не ссылается. Ее обычное назначение — дать описание расширенной ссылке, предназначенное для человека, а не для программы-обработчика. Например:

```
<rem xlink:type="title">
  Другие материалы по теме
</rem>
```

Содержимое элемента типа `title` может быть выведено на экран, записано в журнальный файл, отправлено на печать. Казалось бы, для этого можно было не вводить особый тип ссылки, а воспользоваться специально придуманным для таких случаев атрибутом с тем же именем `title`. Разумеется, часто так и поступают, но дело в том, что с помощью ссылки типа `title` можно дать несколько описаний, вложив в ссылку расширенного типа `extended` несколько элементов-ссылок типа `title`. Это удобно, если надо сделать описания на разных языках. Например:

```
<anim xlink:type="locator"
      xlink:href="http://hsh.com/zoo/animals.xml">

  <descr xlink:type="title" xml:lang="en">
    The Animals
  </descr>
```



```
<descr xlink:type="title" xml:lang="ru">
    Животные
</descr>

<descr xlink:type="title" xml:lang="de">
    Die Tieres
</descr>

</anim>
```

Кроме того, значение атрибута `title` — простая строка символов, а содержимым элемента-ссылки типа `title` может быть что угодно, в том числе какой-то размеченный текст, содержащий вложенные элементы. Например:

```
<im xlink:type="title">
    <html><body>
        <h2>Чертеж 2</h2>
        
    </body></html>
</im>
```

Элементы-ссылки типа `title` можно вложить не только в ссылку типа `extended`, но и в ссылки типа `locator` или `arc`.

Тип *resource*

Под словом *ресурс* (*resource*) в языке XLink понимается всякая единица информации или услуга, имеющая свой адрес URI. Ссылка типа `resource` записывается в теле сложной ссылки типа `extended` и отмечает локальный ресурс, чаще всего текущий документ XML, в котором записана ссылка. Правильнее было бы сказать, что это не ссылка, а пометка, сделанная в каком-то месте локального ресурса атрибутом `label`, как это представлено, например, в листинге 4.2.

Кроме атрибута `label` в ссылках типа `resource` можно использовать атрибуты `title` и `role` языка XLink, а также любые атрибуты, определенные вами. У элемента типа `resource` может быть любое содержимое, в том числе и вложенные элементы, хотя чаще всего такой элемент бывает пустым.

Тип *arc*

Слово "дуга" (*arc*) (имеется в виду дуга ориентированного графа) в языке XLink обозначает описание связи между двумя ресурсами, выполненное с помощью ссылки. Это описание включает направление ссылки и способ обработки дуги программой-анализатором документа XML. Дуга, описывающая связь текущего ресурса с другим ресурсом, называется *выходящей*

(outbound) дугой. Пример такой дуги показан в листинге 4.2. Дуга, описывающая связь удаленного ресурса с текущим ресурсом, называется *входящей* (inbound). Такая дуга записана в примере сразу после листинга 4.2. Дуга, описывающая связь двух удаленных ресурсов, называется *сторонней* (third-party) дугой. Пример такой дуги приведен в листинге 4.3.

Ссылка типа `arc` создает дугу. Как правило, каждая такая ссылка создает дугу только одного вида: входящую, исходящую или стороннюю. В создании дуги участвуют атрибуты `from` и `to`, о которых сейчас пойдет речь. Они задают начало и конец дуги.

Кроме этих атрибутов в ссылках-дугах можно использовать атрибуты `arcrole`, `title`, `show` и `actuate`. Содержимое элемента типа `arc` может быть любым, но вложенные в него элементы не будут рассматриваться как ссылки.

Атрибуты *from* и *to*

Атрибуты `from` и `to` записываются только в дугах — ссылках типа `arc` — для указания начальной и конечной точки дуги. У элементов-ссылок, описывающих эти точки (это ссылки типа `resource` — точки в локальном ресурсе, или `locator` — точки, расположенные в удаленном ресурсе), должен быть атрибут `label`, помечающий ссылки. Значениями атрибутов `from` и `to` служат эти метки. В листинге 4.3 элемент-дуга `load` типа `arc` использует метки `"loc"` и `"base"`, которыми помечены элементы `sresource`, описывающие начальную и конечную точки дуги.

Листинг 4.3. Связь удаленных ресурсов

```
<tplink xlink:type="extended">

  <sresource xlink:type="locator" xlink:label="loc"
    xlink:href="http://some.com/pub/res024.xml" />

  <sresource xlink:type="locator" xlink:label="base"
    xlink:href="http://some.com/pub/res043.xml" />

  <load xlink:type="arc" xlink:from="loc" xlink:to="base" />

</tplink>
```

Описания начальной и конечной точек (в листинге 4.3 они сделаны элементами `sresource`) обязательно должны располагаться в той же ссылке типа `extended`, что и описание дуги (в листинге 4.3 сделанное элементом `load`), связывающей их.

Упражнение

Используйте ссылки-дуги для создания ссылок на различные части одного и того же документа.

Атрибут *show*

Необязательный атрибут `show` можно употреблять в ссылках типа `simple` и `arc` в тех случаях, когда предполагается показать полученный по ссылке ресурс на экране. Он указывает место визуального представления полученного по ссылке ресурса подобно атрибуту `target` тега `<a>` языка HTML. У атрибута `show` всего пять значений, аналогичных значениям атрибута `target`:

- ☐ `"new"` — полученный ресурс следует показать в новом окне, фрейме, на новой панели, короче говоря, в новом месте, а не там, где была ссылка на него;
- ☐ `"replace"` — полученный ресурс следует показать в том же окне, фрейме, или на той же панели, где была ссылка на ресурс;
- ☐ `"embed"` — полученный ресурс следует показать в том же окне, где была ссылка на него, при этом первоначальный ресурс не должен подвергаться преобразованиям;
- ☐ `"other"` — какой-то способ представления, описанный в других элементах документа;
- ☐ `"none"` — способ представления не описан ни в одном элементе документа.

Например:

```
<sign xlink:type="simple"
      xlink:href="http://some.domain/pub/math/sigma.gif">
  xlink:show="embed"
  xlink:title="Знак суммы" />
```

Упражнение

Сопоставьте значения атрибута `show` языка XLink со значениями атрибута `target` тега `<a>` языка HTML.

Атрибут *actuate*

Гиперссылки языка HTML активизируются пользователем при щелчке мыши на выделенном фрагменте текста. После этого отыскивается и загружается ресурс, с которым связана ссылка. Документ XML, как правило, не

просматривается пользователем, а обрабатывается какой-либо программой, которой надо указать, когда следует активизировать ссылку.

Атрибут `actuate` показывает момент времени для активизации ссылки. Он, как и атрибут `show`, тоже используется в ссылках типа `simple` и `arc`. Атрибут `actuate` может принимать четыре значения:

- `"onLoad"` — активизировать ссылку сразу же при загрузке документа;
- `"onRequest"` — активизировать ссылку по какому-то событию — действию пользователя, таймеру и т. п.;
- `"other"` — какой-то способ активизации, описанный в других элементах документа;
- `"none"` — способ активизации не описан ни в одном элементе документа.

Например:

```
<sign xlink:type="simple"
      xlink:href="http://some.domain/pub/intro.xml">
  xlink:actuate="onLoad" />
```

Атрибут *role*

Атрибут `role` указывает на ресурс, описывающий ссылку. Значения атрибута — абсолютные адреса URI. Его удобно использовать в тех случаях, когда несколько ссылок описываются одним ресурсом, определяющим роль, которую играют эти ссылки. Атрибут `role` можно записать в элементе-ссылке любого типа, за двумя исключениями: его нельзя записать в ссылке типа `title`, в которой вообще нет других атрибутов языка XLink, кроме атрибута `type`, и в ссылке типа `arc`, для которой определен атрибут `arcrole`. Например:

```
<problem xlink:type="simple"
          xlink:href="http://some.domain/pub/articles/integral.xml">
  xlink:role="http://some.domain/pub/roles/math.xml">
    Решение интегрального уравнения
</problem>
```

Строка URI, записанная в атрибуте `role` или в атрибуте `arcrole`, не обязана указывать на действительно существующий в Интернете ресурс. Она может служить просто идентификатором, назначающим ту или иную роль ссылке и помогающим классифицировать ресурсы, на которые создается ссылка. Например:

```
<person xlink:type="simple"
         xlink:href="http://some.domain/pub/staff/ivanov.xml">
  xlink:role="http://role/student">
```

```
Иванов П. С.  
  
</person>  
  
<person xlink:type="simple"  
  xlink:href="http://some.domain/pub/staff/petrov.xml">  
  xlink:role="http://role/professor">  
  
  Петров И. С.  
  
</person>
```

Атрибут *arcrole*

Атрибут *arcrole* служит той же цели, что и атрибут *role*, но применяется только в ссылках типа *simple* и *arc*. Его значением тоже может быть любая строка URI, указывающая на реальный ресурс или служащая только идентификатором. Пример такого использования атрибута *arcrole* приведен в следующем разделе.

Создание банка ссылок

При создании какого-либо документа на языке HTML, назовем его для определенности *doc.html*, в него вставляются гиперссылки на предыдущие, ранее созданные, документы и изображения. Пусть эти ресурсы лежат в файлах *old1.html*, *old2.html*, *img1.gif*. Через некоторое время появляются новые документы, назовем их *new1.html*, *new2.html*, на которые необходимо сослаться из документа *doc.html*. Для этого придется отыскать файл *doc.html* и внести в него новые ссылки. Это очень неудобно. Не говоря уже о том, что файл *doc.html* может быть недоступен, его уже могли скопировать на множество сайтов. Придется вносить изменения во все копии, что совершенно невозможно.

Язык XLink, в котором можно сделать ссылки и в прямом, и в обратном направлении, позволяет создать обратные ссылки из новых документов на старый документ. Но это не лучший выход из положения, потому что старый документ при каждом открытии должен отыскать и просмотреть новые документы в поисках этих ссылок. Это требует времени и знания тех адресов, где лежат эти новые документы.

К счастью, язык XLink предлагает другой, более удобный выход из этой ситуации. Мы выносим все ссылки в отдельный файл — "банк ссылок" — и в случае необходимости изменяем ссылки только в этом файле. Все документы, которым нужны ссылки, обращаются за ними в банк ссылок. Такие обращения оформляются обыкновенными ссылками, но для того чтобы ука-

зать программе-обработчику, что идет обращение к банку ссылок, в элемент-дугу записывается атрибут `arcrole` со следующим значением:

```
xlink:arcrole="http://www.w3.org/1999/xlink/properties/linkbase"
```

Такая запись обеспечивает обязательность реализации ссылки программой-обработчиком, но накладывает одно ограничение — банк ссылок должен быть документом XML. В листинге 4.4 приведен пример связи с банком ссылок, активизируемой при загрузке документа `content.xml`.

Листинг 4.4. Связь с банком ссылок

```
<initLinks xlink:type="extended">

  <startcont xlink:type="locator" xlink:label="cont"
    xlink:href="content.xml" />

  <base xlink:type="locator" xlink:label="base"
    xlink:href="linkbase.xml" />

  <load xlink:type="arc"
    xlink:arcrole="http://www.w3.org/1999/xlink/properties/linkbase"
    xlink:from="cont" xlink:to="base" xlink:actuate="onLoad" />

</initLinks>
```

Программы-обработчики атрибутов XLink

Хотя язык XLink создан недавно, уже появилось много программных продуктов, обрабатывающих ссылки, облегчающих их разработку и даже создающих ссылки, не вставляя их в документы XML. Далее перечислены наиболее широко известные продукты.

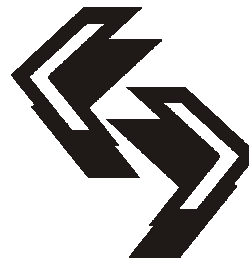
- ❑ Программное средство X2X фирмы UK, Ltd, описанное на сайте фирмы http://www.empolis.co.uk/products/prod_X2X.asp, создает ссылки, не вставляя их в документы XML.
- ❑ Обработчик XLink Processor фирмы Fujitsu реализует языки XLink и XPointer. С ним можно ознакомиться по адресу <http://www.labs.fujitsu.com/free/xlip/en/>.
- ❑ Программный продукт [xlinkit.com](http://www.xlinkit.com) создает ссылки по указанным ему правилам. Он расположен по адресу <http://www.xlinkit.com/>.
- ❑ Популярные свободно распространяемые браузеры Mozilla, <http://www.mozilla.org/>, и Amaya, <http://www.w3.org/Amaya/>, содержат средства обработки атрибутов языка XLink.

- Свободно распространяемый обработчик атрибутов языка XLink, называемый XtoolX, <http://www.xlinkit.com:8080/xtoolx/>, вставляет ссылки, взятые из банка ссылок, в документы XML.

Вопросы для самопроверки

1. На какие информационные ресурсы можно создать ссылки средствами языка XLink?
2. Почему язык XLink определяет только атрибуты элементов XML, но не сами элементы?
3. Как сослаться на тот документ, в котором записана ссылка, т. е. "на самого себя"?
4. Какая разница между атрибутом `title` и элементом-ссылкой типа `title`?
5. Может ли ссылка-дуга связать две точки одного и того же документа XML?
6. Что отличает начальную точку дуги от конечной точки?
7. Обязательно ли значения атрибутов `role` и `arcrole` должны быть реальными адресами Интернета или они могут служить лишь идентификаторами ролей?

ГЛАВА 5



Уточненные ссылки XPointer

Язык XLink, описанный в *главе 4*, позволяет организовать только внешние ссылки на информационный ресурс. Они не могут сослаться на определенное место удаленного документа или на какое-то произвольное место того документа, в котором они записаны. Такие ссылки могут быть полезны, поскольку очень часто в документах нужно организовать ссылку на определенное место того же самого документа, скажем, при создании оглавления, предметного указателя, глоссария.

В языке HTML для этой цели используется все тот же тег `<a>`. В нем атрибутом `href` указывается метка того места документа, на которое мы хотим перейти. Перед меткой ставится символ "решетка" `#`, по-английски называемый "hash". Например:

```
<a href="#ref0012">Глава 5</a>
```

В том месте документа, на которое мы хотим перейти, записывается тег `<a>` с атрибутом `name` и той же меткой:

```
<a name="ref0012"></a>
```

Подобную метку можно записать в удаленном документе, допустим, в файле `remfile.html`, и сослаться на нее следующим образом:

```
<a href="http://some.com/pub/remfile.html#ref0012">Глава 5</a>
```

Браузер загрузит документ `remfile.html` и покажет ту его часть, в которой записан элемент

```
<a name="ref0012"></a>
```

Аналогичная конструкция, разумеется, есть и в XML. По правилам XML метки создаются с помощью атрибутов типа `ID` (см. *главы 2 и 3*), которые можно объявить в любом элементе. Напомню, что тип `ID` образуют простые

имена, состоящие из букв, цифр, точек, дефисов, знаков подчеркивания и начинающиеся с буквы.

Ссылки на помеченные элементы указываются атрибутами типа IDREF или IDREFS, которые тоже можно объявить в любом элементе. Проверяющий анализатор, просматривая документ XML, следит за соответствием меток и ссылок на них, отмечая как ошибку ссылку на несуществующую метку. Знак решетки # в ссылках записывать не нужно, сам тип IDREF показывает, что значение атрибута — ссылка.

Предыдущий пример, переписанный по правилам XML, будет выглядеть следующим образом. Ссылку на помеченный элемент можно записать в виде:

```
<item ref="ref0012">Глава 5</item>
```

а пометить элемент можно так:

```
<ch id="ref0012" />
```

Объявление элементов item и ch на языке XSD (см. главу 3) будет иметь вид:

```
<xsd:element name="item">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="ref" type="xsd:IDREF" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name="ch">
  <xsd:attribute name="id" type="xsd:ID" use="required" />
</xsd:element>
```

Эта простая конструкция языка XML очень скоро перестала удовлетворять разработчиков документов XML. Им потребовалось ссылаться:

- на определенную точку — букву, символ, позицию — внутри элемента;
- на множество точек, выбранных по образцу, скажем, на каждое появление в тексте слова "монитор";

- ❑ не на определенный элемент или точку внутри элемента, а на какую-то часть документа, границы которой не совпадают с началом и концом элемента;
- ❑ на что-нибудь вроде "третьего абзаца пятого параграфа предыдущей версии документа".

Кроме того, не всегда возможно расставлять в элементах атрибуты типа ID. В конце концов, документ не всегда доступен, или у разработчика просто может отсутствовать право записи в документ, на который надо сослаться.

Следуя духу XML, консорциум W3C создал для записи таких уточненных ссылок и меток язык XPointer. Общую структуру языка описывает рекомендация "XPointer Framework", текущую версию которой можно посмотреть по адресу <http://www.w3.org/TR/2003/REC-xptr-framework-20030325/>.

В отличие от рассмотренных ранее в этой книге языков, XPointer не является реализацией XML. Он не определяет никакие типы данных и не объявляет элементы и атрибуты. Он задает только правила записи меток и обращения к ним с помощью ссылок языка XLink.

На языке XPointer метки называются *указателями* (pointers). XPointer определяет два вида указателей: простые указатели и указатели, основанные на схеме. Рассмотрим подробнее каждый из этих видов.

Простые указатели

Простые указатели (shorthand pointers, bare names) языка XPointer и ссылки на них — это смешанный аналог приведенных выше конструкций HTML и XML. Простой указатель представляет собой имя типа NCName языка XSD (см. главу 3), состоящее из букв, цифр, точек, дефисов и знаков подчеркивания. Имя должно начинаться с буквы. Как обычно в XML, указатель записывается в любом атрибуте-идентификаторе типа ID (см. главы 2 и 3), который может содержаться в любом элементе документа XML. Например:

```
<someElem myid="label02">  
    Содержимое элемента  
</someElem>
```

Атрибут-указатель в примере myid обязательно должен быть объявлен при описании схемы документа. Объявить его следует с типом ID. Например, на языке XSD объявление элемента someElem с атрибутом myid может выглядеть так:

```
<xsd:element name="someElem">  
    <xsd:complexType>
```

```
<xsd:simpleContent>

  <xsd:extension base="xsd:string">
    <xsd:attribute name="myid" type="xsd:ID" />
  </xsd:extension>

</xsd:simpleContent>

</xsd:complexType>

</xsd:element>
```

Использование простых указателей в ссылках

Ссылки на информационный ресурс, содержащий указатели, записываются по правилам языка XLink (см. главу 4), в который добавлена конструкция, взятая из языка HTML, а именно в ссылке на ресурс перед указателем ставится знак решетки #:

```
<myLink xlink:type="simple"
  xlink:href="mydoc.xml#label02" />
```

Ссылка, записанная в том же самом документе mydoc.xml, начинается с решетки и выглядит так:

```
<myLink xlink:type="simple"
  xlink:href="#label02" />
```

Если в документе записано несколько одинаковых указателей label02, то ссылка будет связана с первым из них.

Как видите, простые указатели языка XPointer только дублируют конструкции, давно применяемые в HTML и XML. Все новые возможности языка реализованы через *указатели, основанные на схеме* (scheme-based pointers).

Упражнение

Перепишите по правилам языка XPointer ссылки, встречаемые вами в документах HTML.

Указатели, основанные на схеме

Указатели, основанные на схеме, состоят, как и следует из их названия, из одной или нескольких *схем*, записанных через пробелы. Вот пример такого указателя:

```
xpointer(/book/chapter/section) element(color/3)
```

В этом примере указатель состоит из двух схем. Первая схема задает ссылку на элемент `section`, вложенный в элемент `chapter`, который, в свою очередь, вложен в корневой элемент `book`. Эти элементы в документе XML, на который ссылается указатель, могут выглядеть так:

```
<book>

  <chapter name="ch5" title="Глава 5">

    <section name="sect1">
      Язык XLink, описанный в предыдущей главе, ...
    </section>

  </chapter>

</book>
```

Вторая схема ссылается на третий по счету элемент из всех непосредственно вложенных в помеченный простым указателем `color` элемент. В следующем примере:

```
<attr id="color">

  <rgb red="127" green="200" blue="46">
    <depth>24</depth>
  </rgb>

  <hsb hue="35" sat="105" br="45" />

  <cmypk c="25" mag="87" yel="110" bl="80" />

  <icc profile="cl.icm" />

</attr>
```

таким третьим по счету элементом является элемент `cmypk`.

Использование указателей в ссылках

Указатели, основанные на схеме, используются точно так же, как и простые указатели. Они записываются в атрибутах элементов-ссылок после пути к файлу и отделяются от него "решеткой":

```
<myLink xlink:type="simple"
  xlink:href="mydoc.xml#element(color/3)" />
```

Понятие схемы в языке XPointer

Слово "схема" (scheme) в языке XPointer получило новое значение. Это запись вида

```
element(color/3)
```

похожая на запись функции и состоящая из имени схемы — в примере это имя `element` — и данных, записанных в скобках, в примере данные — это строка `color/3`.

Схемы, записанные в указателе, просматриваются последовательно до тех пор, пока не будет найдена точка в документе, отвечающая какой-либо схеме. После этого просмотр указателя прекращается, оставшиеся схемы не рассматриваются. В приведенном выше примере, если будет найден элемент `section`, то схема `element(color/3)` рассматриваться уже не будет.

Имя схемы — это уточненное имя типа `QName` (см. главу 1), состоящее из необязательного префикса, связанного с идентификатором пространства имен, и локальной части, отделенной от префикса двоеточием.

На момент написания этой книги в языке XPointer определены всего три схемы с именами `xmlns`, `element` и `xpointer`. Имена без префиксов зарезервированы за схемами, которые определяются в рекомендациях консорциума W3C. Каждая схема, предложенная консорциумом W3C, описывается отдельной рекомендацией. Разработчики могут вводить свои схемы, снабжая их имена префиксами.

Смысл и правила записи данных зависят от вида схемы. Есть только одно общее правило, вытекающее из того, что данные записываются в скобках — все скобки, относящиеся к данным, должны быть парными или предваряться символом "каре" ("крышечкой"): `xxx^(yyy или xxx^)^yyy`. Если же в данных встречается символ каре, то его следует удваивать: `xxx^^yyy`.

Разумеется, схема — это не функция, она ничего не вычисляет и не выдает никакого результата. Это просто форма записи, несколько неожиданная и непривычная для языков, основанных на XML. Выбрана она, видимо, потому, что в языке XPointer есть настоящие функции. Кроме того, XPointer использует функции языка XPath, которому посвящена следующая глава. Мы изучим такие функции далее, а сейчас рассмотрим стандартные схемы языка XPointer.

Схема `element()`

Схема `element()` реализует потребность ссылаться на элемент документа XML примерно в таком стиле: "сослаться на второй абзац третьего параграфа договора № 5". Реализация очень проста и выглядит следующим образом:

```
element(/1/3/2)
```

В этом примере начальная наклонная черта и единица показывают, что отсчет начинается с корневого элемента. В нем выбирается третий по счету непосредственно вложенный элемент. В этом элементе выбирается второй по счету вложенный в него элемент. Документ, на который сделана эта ссылка, может выглядеть так:

```
<contract numb="5">

  <section id="sect1a">
    <paragraph>Первый абзац первого параграфа</paragraph>
    <paragraph>Второй абзац первого параграфа</paragraph>
  </section>

  <section id="sect2a">
    <paragraph>Первый абзац второго параграфа</paragraph>
  </section>

  <section id="sect3a">
    <paragraph>Первый абзац третьего параграфа</paragraph>
    <paragraph>Второй абзац третьего параграфа</paragraph>
  </section>

</contract>
```

Запись вида /1/3/2 называется *последовательностью вложений* (child sequence). Она напоминает запись пути к файлу в UNIX. Наклонная черта отмечает вложенный элемент подобно вложенному каталогу файловой системы, но вместо имени вложенного каталога или файла стоит натуральное число. Число, записанное за наклонной чертой, показывает порядковый номер непосредственно вложенного элемента. Отсчет элементов начинается с 1.

Поскольку в хорошо оформленном документе XML может быть только один корневой элемент, последовательность вложений обычно начинается с наклонной черты и единицы: /1. За следующей наклонной чертой перечисляются элементы, непосредственно вложенные в корневой элемент и т. д.

Перед последовательностью вложений может стоять простой указатель. В этом случае вложения отсчитываются не от корневого элемента, а от элемента, помеченного этим указателем. Применяя простой указатель, предыдущий пример можно записать так:

```
element(sect3a/2)
```

Наконец, данные в схеме element() могут состоять только из простого указателя:

```
element(sect3a)
```

Это эквивалентно написанию простого указателя без всякой схемы.

Схема `element()` описана в рекомендации "XPointer `element()` Scheme", находящейся по адресу <http://www.w3.org/TR/xptr-element/>.

Упражнения

1. Составьте указатель на второй абзац первого параграфа приведенного ранее примера.
2. Сошлитесь на третий параграф целиком.

Схема `xpointer()`

Схема `xpointer()` использует для создания указателей и ссылок на них мощные средства адресации элементов и других частей документа, предоставляемые языком XPath. В схеме `xpointer()` данными, записываемыми в скобках, могут служить любые выражения, допускаемые в языке XPath, а также их расширения, введенные в язык XPointer. Выражения языка XPath мы подробно рассмотрим в следующей главе, а пока приведем примеры таких выражений и запишем указатели, созданные с помощью языка XPath.

Запишем фрагмент некоторого документа XML, на который будем ссылаться в наших примерах и упражнениях. Он приведен в листинге 5.1.

Листинг 5.1. Фрагмент документа XML

```
<contract numb="5">

  <section id="sect1a" name="intro">
    <paragraph>Первый абзац первого параграфа</paragraph>
    <paragraph>Второй абзац первого параграфа</paragraph>
  </section>

  <remark>Примечание 1</remark>

  <section id="sect2a">
    <paragraph>Первый абзац второго параграфа</paragraph>
    <remark>Отметим, что такого быть не может.</remark>
  </section>

  <remark>Примечание 2</remark>
  <remark>Примечание 3</remark>

  <section id="sect3a" name="comment">
    <paragraph>Первый абзац третьего параграфа</paragraph>
    <paragraph>Второй абзац третьего параграфа</paragraph>
  </section>

</contract>
```

Примеры

1. Для того чтобы сослаться на корневой элемент `contract` листинга 5.1, достаточно написать

```
xpointer (/contract)
```

2. Запись

```
xpointer (/contract/section)
```

обращается ко всем элементам `section`, имеющимся в документе.

3. Указатель

```
xpointer (/contract/section/paragraph)
```

выделяет все элементы `paragraph`.

4. Того же результата в листинге 5.1 можно добиться, записав:

```
xpointer (/contract/section/*)
```

Символ "звездочка" означает "все элементы". В данном примере это значит: "Все элементы, вложенные во все элементы `section`". Совпадение результатов примеров 3 и 4 произошло потому, что в элементы `section` вложены только элементы `paragraph`, других элементов, "подпадающих под звездочку", нет.

5. Еще один способ получить указатель на все элементы `paragraph`:

```
xpointer (/contract/*/paragraph)
```

Здесь звездочка выбирает все элементы, вложенные в элемент `contract`, в том числе и элементы `remark`, но в элементах `remark` нет элементов `paragraph`. То же самое получится, если записать

```
xpointer (/*/*/paragraph)
```

поскольку в хорошо оформленном документе есть только один корневой элемент.

6. Следующий способ получить указатель на все элементы `paragraph`:

```
xpointer (/contract//paragraph)
```

Две наклонные черты подряд — это сокращение записи

```
/descendant-or-self::node()
```

Длинное и непонятное выражение между наклонными чертами означает: "Выбрать сам узел и все узлы, вложенные в него". Поскольку оно встречается очень часто, договорились опускать его, оставляя только наклонные черточки.

7. Запись

```
xpointer(//section/paragraph)
```

опять ссылается на все элементы `paragraph`, т. к. две наклонные черты в начале выражения выбирают все элементы `section` во всем документе, где бы они ни были — непосредственно в корневом элементе или во вложенных элементах.

8. `xpointer(//*)` — эта запись указывает на весь документ.

Предыдущие примеры показывают очевидное сходство записи элементов в выражениях XPath с записью пути к файлу в файловой системе UNIX. Это сходство не случайно — язык XPath представляет документ XML в виде дерева. Чтобы подчеркнуть это сходство, дерево нашего документа `contract` изображено на рис. 5.1 в том виде, в каком обычно показывают файловую систему.

Сходство с файловой системой усугубляется еще и тем, что язык XPath тоже различает абсолютный путь, начинающийся с наклонной черты, и относительный путь, а также использует точку для обозначения текущего элемента и две точки подряд для обозначения "родительского" элемента. Следующие примеры показывают применение этих обозначений.

9. Указатель на все элементы `paragraph`, вложенные в текущий элемент:

```
xpointer(../paragraph)
```

10. Указатель на все элементы, вложенные в "родительский" элемент:

```
xpointer(../*)
```

До сих пор мы выбирали элементы по имени и получали совокупность всех элементов с данным именем. Чтобы выбрать отдельный элемент, указывается его порядковый номер в квадратных скобках, т. е. применяется запись, аналогичная выбору элемента массива. Следующие примеры показывают употребление такой записи.

11. Указатель на первый элемент `section`:

```
xpointer(/contract/section[1])
```

12. Для указания на последний элемент есть функция `last()`, подсчитывающая число элементов:

```
xpointer(/contract/section[last()])
```

13. Более сложный пример — выбираются элементы `section`, содержащие ровно два вложенных элемента:

```
xpointer(/contract/section[count(*)=2])
```

14. Аналогично можно выбрать элементы `section`, содержащие более одного вложенного элемента:

```
xpointer(/contract/section[count(*)>1])
```

Аргументом функции `count()` может быть конкретное имя вложенного элемента. Функция будет подсчитывать только заданные элементы.

15. Указатель на все элементы `section`, в которые вложено более одного элемента `paragraph`:

```
xpointer(/contract/section[count(paragraph)>1])
```

16. Указатель на все элементы документа, в которые вложено ровно два любых элемента:

```
xpointer(//*[count(*)=2])
```

17. Указатель на первый абзац второго раздела:

```
xpointer(/contract/section[2]/paragraph[1])
```

В квадратных скобках может быть очень сложное выражение, позволяющее сделать практически любой выбор. Мы познакомимся с выражениями языка XPath в следующей главе, а пока приведем еще несколько примеров.

18. Выбор нескольких элементов:

```
xpointer(//section | //remark)
```

Вертикальная черта означает "выбор и первого и второго выражения". Выбираются все элементы `section`, где бы они ни были, и все элементы `remark`. Таким же образом можно объединить выбор не только двух, но и нескольких выражений.

До сих пор мы выбирали элементы только по их именам. Язык XPath позволяет выбирать элементы и по их атрибутам. При этом имена атрибутов отмечаются "собачкой" `@`, чтобы отличить их от имен элементов.

19. Выбор всех атрибутов `id`:

```
xpointer(//@id)
```

20. Выбор всех элементов `section`, имеющих атрибут `name`:

```
xpointer(//section[@name])
```

21. Выбор всех элементов `section`, имеющих хотя бы один любой атрибут:

```
xpointer(//section[@*])
```

22. Противоположный выбор — всех элементов `section`, не имеющих ни одного атрибута:

```
xpointer(//section[not(@*)])
```

23. Выбор всех элементов `section`, имеющих атрибут `name` со значением `"remark"`:

```
xpointer(//section[@name="remark"])
```

24. Указатель на атрибут `id` "родительского" узла:

```
xpointer(..//@id)
```

Дерево документа

Язык XPath рассматривает документ XML как дерево, подобное дереву, изображенному на рис. 1.1. Корнем дерева будет корневой элемент документа, а узлами (nodes) — вложенные элементы, содержимое элемента (текстовый узел) или его атрибуты. Кроме того, в узлах дерева могут находиться комментарии, инструкции по обработке, пространства имен.

Такое расширенное понятие узла приводит к неожиданному следствию. Поясним его на примере листинга 5.1.

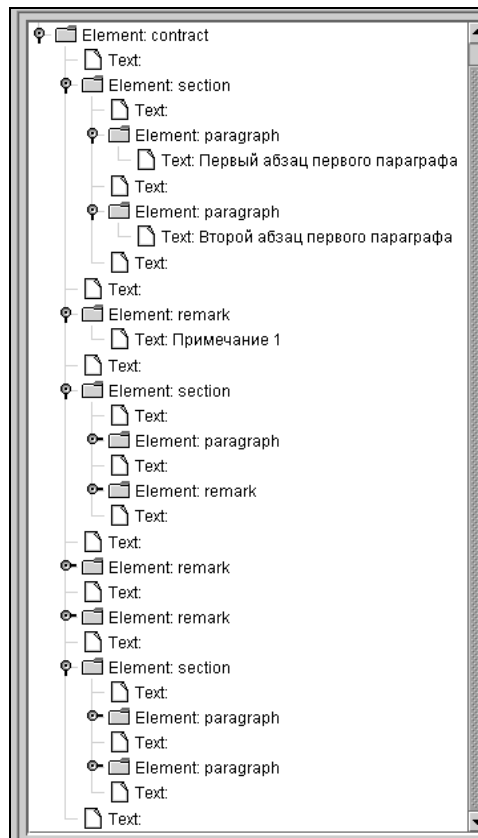


Рис. 5.1. Дерево документа, описанного в листинге 5.1

На рис. 5.1 показано дерево, построенное в соответствии с документом листинга 5.1. Для простоты на рисунке не показаны узлы атрибутов. Зато в де-

реве кроме узлов `contract`, `section`, `paragraph` и `remark` вида "Element", построенных по элементам нашего документа, и текстовых узлов вида "Text" с содержимым элементов, неожиданно появились текстовые узлы вида "Text" без всякого текста. Дело в том, что после тега `<section id="sect1a">` сделан перевод строки, а на следующей строке оставлено шесть пробелов. То же самое сделано и в следующих строках листинга. Символы перевода строки и пробелы попали в документ XML и отражены в дереве как текстовые узлы. Пробелы и переводы строки, записанные в эти узлы, не видны на рисунке.

Схема `element()`, рассмотренная в предыдущем разделе, не учитывает текстовые узлы и узлы-атрибуты. Она отмечает только вложенные элементы. Поэтому в примерах предыдущего раздела мы не принимали во внимание переводы строк, пробелы и другое содержимое элемента. Но схема `xpointer()` должна строго следовать правилам языка XPath и учитывать текстовые узлы, даже если они содержат только пробельные символы. Впрочем, многие программы-анализаторы языка XPointer не следуют этому правилу. Например, функция `last()` разными программами-анализаторами будет вычислена по-разному в зависимости от того, как они построят дерево документа.

Все предыдущие примеры, не содержащие функцию `last()`, не опирались на пересчет узлов дерева и не зависят от этой особенности построения дерева документа. Более сложные примеры и все тонкости составления выражений языка XPath мы рассмотрим в следующей главе, а сейчас посмотрим, какие дополнения внес язык XPointer в данные схемы `xpointer()` и что в них может встречаться, кроме выражений XPath.

Упражнения

1. В листинге 5.1 сошлитесь на примечание 2.
2. Постройте указатель на все элементы `remark`.
3. Запишите указатель на все элементы `remark`, находящиеся внутри элементов `section`.
4. Сошлитесь на последний элемент `remark`.

Дополнения языка XPointer

Исторически сложилось так, что язык XPath был создан на два года раньше языка XPointer. Поэтому создатели языка XPointer внесли в него дополнения, расширившие конструкции языка XPath.

Во-первых, кроме узлов дерева, язык XPointer рассматривает точки (`points`) и области (`ranges`).

Точкой язык XPointer называет позицию между символами документа XML. Точное определение, которое мы не будем здесь приводить, отталкивается

от понятия узла, определяя положение точки индексом, отсчитываемым от начала узла.

Область занимает пространство между двумя точками: начальной точкой (start point) и конечной точкой (end point). Начальная и конечная точки могут располагаться в любом месте документа XML, следовательно, область может пересекать элементы документа XML, не совпадая с узлами дерева. Разумеется, начальная точка должна встретиться в документе раньше конечной точки.

Точки, области и узлы вместе образуют *местоположение* (location). В результате всякого поиска отыскивается некоторое местоположение или *набор местоположений* (location set).

Во-вторых, в схему xpointer() введены новые функции range(), string-range(), range-to(), range-inside(), here(), origin(), start-point(), end-point(), работающие с точками и областями.

Эти дополнения позволяют гораздо точнее адресовать различные части документа, вплоть до отдельного символа.

Все действия с точками и областями выполняются с помощью перечисленных функций, поэтому рассмотрим их подробнее.

Функции языка XPointer

Аргументом всех функций, введенных в язык XPointer, за исключением функций here() и origin(), служит набор местоположений. В простейшем случае задается одно местоположение: точка, область или узел. Функции анализируют этот набор и возвращают в качестве результата новый набор местоположений, определяемый по правилам, заложенным в эту функцию. В простейшем случае результатом будет одно местоположение.

Функция string-range()

Самая мощная и полезная функция, введенная в XPointer, — это функция string-range(). В простейшем случае у нее два аргумента: набор местоположений и строка символов. Функция вычисляется в три этапа.

На первом этапе каждое местоположение преобразуется в строку символов по приведенным далее правилам.

- Точка преобразуется в пустую строку.
- Область преобразуется в строку, составленную из содержимого всех текстовых узлов, содержащихся в ней.
- Корневой элемент преобразуется в строку, содержащую все символы всех текстовых узлов документа.
- Элемент преобразуется в строку, содержащую все символы всех вложенных в него текстовых узлов.

- ❑ Текстовый узел сам является строкой символов.
- ❑ Атрибуты представляются своими значениями в нормализованной форме, т. е. все пробельные символы заменяются пробелами, удаляются начальные и конечные пробелы, затем из нескольких подряд идущих пробелов оставляется только один пробел.
- ❑ Комментарий представляется своим значением.
- ❑ Пространство имен представляется своей строкой URI.
- ❑ Инструкции по обработке представляются только комментариями, включенными в них.

На втором этапе отбираются те строки символов, которые содержат заданную вторым аргументом функции строку.

На третьем, заключительном этапе, отобранные строки представляются областями, и функция возвращает набор областей.

Следующие примеры объясняют использование функции.

25. Указатель на все слова "абзац" в документе:

```
xpointer(string-range(//*, "абзац"))
```

26. Указатель на все строки "граф" в элементах paragraph:

```
xpointer(string-range(/contract/section/paragraph, "граф"))
```

27. Указатель на вторую по счету строку "граф":

```
xpointer(string-range(//*, "граф")[2])
```

28. Указатель на область, начинающуюся со слов "второго параграфа" и заканчивающуюся словом "Отметим". Теги `</paragraph>` и `<remark>` не входят в область:

```
xpointer(string-range(//*, "второго параграфаОтметим"))
```

В более сложном случае у функции `string-range()` появляется третий аргумент. Этот аргумент — число (отсчитываемое от единицы), показывающее позицию перед символом, которая будет начальной точкой возвращаемой области. Таким образом, область, выбранная по первым двум аргументам, сужается перемещением вправо начальной точки на число позиций, показываемое третьим аргументом функции.

29. Указатель на строки "графа":

```
xpointer(string-range(//*, "параграфа", 4))
```

Наконец, в полной форме функции `string-range()` есть четвертый аргумент — число, показывающее количество отбираемых символов. Если это число равно нулю, то функция возвратит область, состоящую из одной точки.

30. Указатель на букву "ф" в слове "параграфа":

```
xpointer(string-range(//paragraph, "параграфа", 8, 1))
```

31. Указатель на позицию после слова "параграфа":

```
xpointer(string-range(//paragraph, "параграфа", 10, 0))
```

Функция *start-point()*

Функция `start-point()` определяет начальную точку каждого местоположения, заданного в ее аргументе. В результате работы функции получается набор подобных точек. Приведем примеры использования этой функции.

32. Указатель на точку, лежащую сразу после закрывающей угловой скобки тега `<contract numb="5">`, выглядит так:

```
xpointer(start-point(//contract))
```

33. Указатель на точку, лежащую сразу после первого открывающего тега `<paragraph>`:

```
xpointer(start-point(/contract/section[1]/paragraph[1]))
```

34. Указатель на точки перед словами "абзац":

```
xpointer(start-point(string-range(//*, "абзац")))
```

Функция *end-point()*

Функция `end-point()` определяет конечную точку каждого местоположения, заданного в ее аргументе. В результате работы функции получается набор конечных точек. Если в предыдущих примерах поменять наименование функции, то получим такие результаты.

35. Указатель на точку, лежащую сразу перед закрывающим тегом `</contract>`:

```
xpointer(end-point(//contract))
```

36. Указатель на точку перед первым закрывающим тегом `</paragraph>`:

```
xpointer(end-point(/contract/section[1]/paragraph[1]))
```

37. Указатель на точку после закрывающего тега `</contract>`:

```
xpointer(end-point(/))
```

Функция *range-to()*

Функция `range-to()` преобразует каждое местоположение-аргумент в область, используя для определения ее начальной и конечной точки функции `start-point()` и `end-point()`.

Функция *range()*

Функция `range()` преобразует каждое местоположение, заданное в ее аргументе, в область, целиком покрывающую местоположение (*covering range*), и возвращает набор полученных областей. При этом точка понимается как область, у которой начальная и конечная точки совпадают.

38. Указатель на весь документ:

```
xpointer(range(/))
```

Функция *range-inside()*

Функция `range-inside()` действует так же, как и функция `range()`, но для аргументов-элементов не возвращает их атрибуты, а только содержимое элементов и вложенные элементы.

Функция *here()*

У функции `here()` нет аргументов. Ее имеет смысл использовать только в текстовом узле или атрибуте. Она возвращает в виде местоположения тот элемент, в котором находится данный текстовый узел или атрибут.

Функция *origin()*

Функция `origin()`, не имеющая аргументов, возвращает в качестве результата местоположение элемента, из которого получена входящая или сторонняя ссылка языка XLink.

Вот все, что включено в схему `xpointer()`. Впрочем, рекомендация по применению схемы `xpointer()` пока находится в черновом состоянии. С ее текущей версией можно ознакомиться по адресу <http://www.w3.org/TR/xptr-xpointer/>. Возможно, в окончательной версии схемы `xpointer()` появятся новые конструкции.

Упражнения

1. Сошлитесь на элемент `remark`, содержащий примечание 2.
2. Создайте указатель на элемент `section`, атрибут `name` которого имеет значение `"comment"`.
3. Напишите указатель на точку, расположенную сразу после слова "такого".

Схема *xmlns()*

В документах XML, как правило, используются уточненные имена элементов и атрибутов типа `QName`, относящие их к тому или иному пространству имен. Отсутствие префикса означает принадлежность имени пространству

имен по умолчанию. В ссылках на какой-либо элемент необходимо указывать его уточненное имя или пространство имен. Как мы видели в предыдущих главах, область действия префикса ограничена элементом, в котором он определен, включая вложенные элементы.

Если указатель записан в области действия префикса, то использование в нем уточненных имен не вызывает никаких затруднений. Если же указатель записан вне области действия префикса, то программа-обработчик не сможет его распознать. Нужно какое-то средство для того, чтобы обработчик "понял", с каким пространством имен связан префикс. Таким средством служит схема `xmlns()`. Она описана в рекомендации "XPointer `xmlns()` Scheme", опубликованной по адресу <http://www.w3.org/TR/xptr-xmlns/>.

Схема `xmlns()` очень проста. В скобках записывается префикс, связанный знаком равенства с идентификатором пространства имен. Все идущие следом схемы могут использовать имена с этим префиксом. Например:

```
xmlns(abc=http://example.com/ns/abc)
```

Как видите, идентификатор пространства имен не надо записывать в кавычках или в апострофах.

Напомню, что префикс нельзя начинать со строки символов `xml` с любым регистром входящих в нее букв. В качестве идентификатора пространства имен нельзя применять строку `http://www.w3.org/XML/1998/namespace` или строку `http://www.w3.org/2000/xmlns/`.

Вот два примера использования схемы `xmlns()`, взятые из рекомендации. В первом примере в указателе встречается схема `img:rect()`, созданная разработчиком. По соглашению, ее имя снабжено префиксом, который должен быть описан в схеме `xmlns()`:

```
xmlns(img=http://example.org/image) img:rect(10,10,50,50)
```

Во втором примере строится указатель на элемент:

```
<customer xmlns="http://example.org/customer">
  <name xmlns="http://example.org/personal-info">John Doe</name>
</customer>
```

Этот элемент использует пространства имен, которые должны быть описаны в схеме `xmlns()`:

```
xmlns(c=http://example.org/customer) xmlns(p=http://example.org/
personal-info) xpointer(/c:customer/p:name)
```

Программы-обработчики XPointer

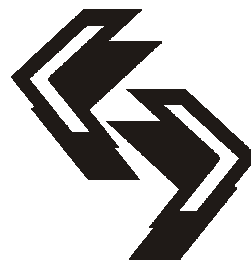
Обработка конструкций языка XPointer включена во все современные обработчики документов XML. Кроме того, различные фирмы выпускают отдельные коммерческие и свободно распространяемые программы-анализаторы и интерпретаторы языка XPointer. Наиболее известны разработки, перечисленные ниже.

- ❑ Уже упомянутый в *главе 4* XLink Processor, разработанный фирмой Fujitsu Laboratories Ltd., кроме языка XLink "понимает" почти все конструкции языка XPointer.
- ❑ Программа libxml — библиотека XML, — с которой можно ознакомиться по адресу <http://xmlsoft.org/>, полностью реализует синтаксис язык XPointer.
- ❑ Обработчик 4XPointer фирмы Fourthought, Inc., <http://www.fourthought.com/>, написан полностью на языке Python.
- ❑ В университете города Болонья, <http://www.cs.unibo.it/~fabio/xpointer/>, разрабатываются сразу две реализации языка XPointer.
- ❑ В популярные браузеры Mozilla и Netscape Communicator встраивается обработчик XPointerLib, разработанный по проекту Connexions, <http://www.mozdev.org/>.

Вопросы для самопроверки

1. Для чего нужны указатели на различные части документа?
2. Из каких частей состоит указатель?
3. Что понимается под "схемой" в указателях?
4. Как строится последовательность вложений в схеме `element()`?
5. Как применяется язык XPath в схеме `xpointer()`?
6. Для чего понадобились расширения языка XPath, сделанные в схеме `xpointer()`?
7. Можно ли считать точку вырожденной областью?
8. Какие виды узлов встречаются в дереве, построенном по документу XML?
9. Как узел преобразуется в строку?
10. Каково содержание строки, построенной по корневому элементу?
11. Как элемент XML преобразуется в область?
12. Как корневой элемент преобразуется в область?

ГЛАВА 6



Адресация на языке XPath

Как вам уже, наверное, стало понятно из предыдущих глав книги, для удобной работы с документом XML необходимы средства, позволяющие точно адресовать ту или иную его часть: отдельную точку, множество точек, какой-либо участок или множество участков документа. Такие средства предоставляет язык XPath, разработанный консорциумом W3C. В то время, когда писалась эта книга, на уровне рекомендации находилась первая версия языка, выпущенная в ноябре 1999 года. Ее спецификацию "XML Path Language (XPath)" можно посмотреть по адресу <http://www.w3.org/TR/xpath.html>. В то же время в стадии разработки находилась вторая версия языка. Текущий черновой вариант спецификации второй версии языка "XML Path Language (XPath) 2.0" можно посмотреть по адресу <http://www.w3.org/TR/xpath20/>. Сейчас, когда вы держите в руках эту книгу, вторая версия должна быть уже в окончательной стадии разработки. В этой главе мы познакомимся с версией XPath 2.0 и будем отмечать ее отличия от первой версии.

Язык XPath, как и XPointer, — не реализация XML. Его основу составляют выражения различных типов, в числе которых логический, числовой и строковый тип. В выражениях записываются константы, переменные и функции, входящие в состав XPath. Они связываются операциями, характерными для данного типа. В результате вычисления выражения получается указание на какой-то один или сразу несколько участков документа.

Очень часто выражение строится подобно пути к файлу в файловой системе, откуда и происходит название языка "XML Path", сокращенно XPath. При этом документ XML рассматривается как дерево (см. рис. 1.1, 5.1, 6.1). Познакомимся подробнее со структурой документа, как ее представляет язык XPath.

Дерево документа

Язык XPath представляет документ в виде дерева, корнем которого служит корневой элемент документа. От корня отходят ветви, заканчивающиеся

узлами. *Узлами* (nodes) служат, например, вложенные элементы, их атрибуты и тексты, составляющие содержимое корневого элемента. От каждого вложенного элемента отходят свои ветви, рекурсивно повторяющие ветви корневого элемента. Таким образом, у каждого узла может быть только один узел-"предок", причем предком может быть только узел корневого или другого элемента дерева, но не атрибут и не текстовый узел. Неполный пример дерева документа, без атрибутов, был показан на рис. 5.1.

Узлы дерева

Язык XPath различает семь видов узлов.

- ❑ *Узлы документа* (document nodes). В первой версии XPath они назывались корневыми узлами (root nodes). Не отождествляйте узел документа с узлом корневого элемента документа. Узел корневого элемента вложен в узел документа наряду с узлами-комментариями и узлами инструкций по обработке. Кроме того, в языке XPath предусмотрена возможность работы с другими типами документов, возможно, содержащими несколько корневых элементов. Имя узла документа совпадает с именем корневого элемента документа.
- ❑ *Узлы-элементы* (element nodes). Имя узла-элемента состоит из идентификатора пространства имен, получаемого по префиксу уточненного имени элемента, и локального имени элемента. Это *расширенное имя* (expanded-name) узла. Если у элемента есть атрибут типа ID, то он служит идентификатором узла.
- ❑ *Узлы-атрибуты* (attributes nodes). Их предок — узел-элемент, к которому относятся атрибуты, хотя они не считаются потомками этого узла. Узел-атрибут тоже определяется расширенным именем, полученным из уточненного имени атрибута.
- ❑ *Узлы пространств имен* (namespace nodes). С каждым узлом-элементом связаны узлы тех пространств имен, в область действия которых входит данный элемент. Так же как и узлы-атрибуты, они не считаются потомками узла-элемента, хотя считают его своим предком. Поэтому программа-анализатор, обходя дерево, не будет автоматически просматривать узлы пространств имен. Имя узла пространства имен — это префикс, связанный с ним.
- ❑ *Узлы инструкций по обработке* (processing instruction nodes). Это отдельные узлы, имена которых — это имена целевых приложений, выполняющих инструкцию. Первая строка пролога документа XML `<?xml version="1.0"?>` не считается инструкцией по обработке и не входит в дерево документа.
- ❑ *Узлы-комментарии* (comment nodes). Каждый комментарий заносится в дерево как узел без имени.

- ❑ **Текстовые узлы** (text nodes). Это строка символов, записанная в теле элемента между вложенными элементами. У текстовых узлов нет расширенного имени.

Узел любого вида можно представить строкой символов. Для каждого вида узла правила представления свои, они перечислены далее.

- ❑ Строка, представляющая узел документа или узел-элемент, состоит из всех строк, представляющих его узлы-потомки текстового вида.
- ❑ Строка, представляющая узел-атрибут, содержит его значение.
- ❑ Строка, представляющая узел пространства имен, содержит его строку URI.
- ❑ Строка узла инструкции по обработке составлена из ее содержимого.
- ❑ Узел-комментарий и текстовый узел сами являются строками, причем начальные символы `<!--` и конечные символы `-->` комментария в строку не входят.

Атомарные значения

Атомарные значения (atomic values) появились в версии XPath 2.0 для записи листьев дерева. Это значения одного из встроенных простых типов (atomic types) языка XSD (см. главу 3): `string`, `boolean`, `decimal`, `float`, `double`, `duration`, `dateTime`, `time`, `date`, `gYearMonth`, `gYear`, `gMonthDay`, `gDay`, `gMonth`, `hexBinary`, `base64Binary`, `anyURI`, `QName`, `NOTATION`. От названных типов путем сужения с помощью фасеток (см. главу 3) можно породить новые атомарные значения. Данные типы определены в пространстве имен `http://www.w3.org/2001/XMLSchema`. Мы дадим этому пространству имен его обычный префикс `xs`.

Кроме типов языка XSD, язык XPath использует свои типы `anyAtomicType`, `untypedAtomic`, `dayTimeDuration` и `yearMonthDuration`. Они определены в пространстве имен `http://www.w3.org/2003/05/xpath-datatypes`, которому обычно дают префикс `xdt`.

Как следует из названия, тип `anyAtomicType` — это абстрактный тип, обобщающий все атомарные значения. Он является подтипом типа `xs:anySimpleType`, а все атомарные значения — его подтипы.

Тип `untypedAtomic` состоит из всех значений, не имеющих типа, не проверяемых анализатором, таких как простой текст.

Тип `dayTimeDuration` — это подтип типа `xs:duration`, содержащий только день, час, минуты и секунды.

Тип `yearMonthDuration` — это подтип типа `xs:duration`, содержащий только год и месяц.

Последовательности

В результате вычисления каждого выражения получается адрес какого-то участка документа XML в виде упорядоченной последовательности (sequence) узлов и/или атомарных значений. В крайнем случае, последовательность может быть пустой. Первая версия языка XPath, в которой не было атомарных значений, вместо последовательности работала с *множеством узлов* (node set). В отличие от последовательности множество узлов неупорядочено, следовательно, не может содержать повторяющихся значений.

Выражения, определяющие путь

Чаще всего выражение языка XPath показывает путь к определяемому участку документа XML, начинающийся в корневом узле документа или каком-то начальном узле. Такие выражения (expression paths) состоят из нескольких *шагов поиска* (location steps), выполняемых последовательно слева направо. Последовательность, полученная на предыдущем шаге, передается следующему шагу, который использует ее как исходный материал для поиска. Шаги в выражении разделены наклонными чертами. В *главе 5* мы видели много примеров таких выражений, применявшихся в схеме `xpointer()` для получения указателя. В тех простейших выражениях каждый шаг поиска содержал, чаще всего, только имя элемента. Например, выражение

```
/contract/section/paragraph
```

состоит из трех шагов поиска, содержащих имена элементов `contract`, `section` и `paragraph`. В результате вычисления первого шага этого выражения получится последовательность узлов, вложенных в узел документа `contract`. На втором шаге из этой последовательности выбираются узлы `section`, на третьем — узлы `paragraph`. В результате вычисления всего выражения получается последовательность узлов, состоящая из всех элементов `paragraph`, вложенных в элементы `section`, которые, в свою очередь, вложены в элементы `contract`.

В общем случае шаг поиска устроен гораздо сложнее. Язык XPath 2.0 различает два вида шагов поиска: *шаг, направляемый осью поиска* (axis step), и *шаг, направляемый фильтром* (filter step). После выполнения шага, направляемого осью, получается последовательность узлов, а после выполнения фильтра в последовательности кроме узлов могут встретиться атомарные значения.

Шаг, направляемый осью поиска

Шаг, направляемый осью поиска, состоит из трех частей: *оси поиска* (axis), *теста узла* (node test) и необязательного *предиката* (predicate). Ось отделя-

ется от теста узла двумя двоеточиями, а предикат записывается после теста узла в квадратных скобках:

```
ось::тест узла[предикат].
```

Например, шаг поиска может выглядеть так:

```
child::section[1]
```

В этом примере ось поиска `child` показывает, что поиск охватывает все узлы, непосредственно вложенные в просматриваемый узел, за исключением узлов-атрибутов и узлов пространств имен, тест узла `section` выбирает из этих узлов узлы-элементы `section`, а предикат `1` выбирает первый из встреченных узлов-элементов `section`.

Каждая из трех частей шага поиска сужает первоначальную область поиска.

Ось поиска задает направление поиска, отсчитываемое от текущего узла, и его объем. Например, поиск может идти в сторону вложенных элементов, или, наоборот, просматривать родительские узлы. Можно просматривать только атрибуты элементов или только соседние элементы.

Тест узла выбирает в области поиска, заданной осью, определенные узлы по их имени или типу.

Предикат содержит условия, по которым проверяет узлы, уже отобранные осью поиска и тестом узла, и делает окончательный выбор.

Итак, ось поиска говорит о том, КУДА мы движемся, тест узла показывает, ЧТО мы ищем, а предикат уточняет, КАКИМИ свойствами должен обладать искомый объект.

Познакомимся подробнее с каждой из трех частей шага поиска, определяемого осью.

Оси поиска

В дереве документа естественно выделяются два противоположных направления поиска: "вниз" — от текущего узла документа по ветвям к вложенным узлам-элементам и листьям, и "вверх" — от листьев или текущих узлов-элементов к узлу документа. Оси поиска позволяют организовать поиск и в других направлениях: среди всех потомков данного узла или только среди непосредственных потомков, среди атрибутов, соседних узлов, всех предков данного узла. Всего насчитывается тринадцать осей поиска.

Восемь осей, перечисленных далее, задают прямой поиск (forward axis) — поиск "вниз":

- `self` — сам текущий узел;
- `child` — все непосредственные узлы-потомки, кроме узлов-атрибутов и узлов пространств имен; эта ось принимается осью по умолчанию;

- ❑ `descendant` — все узлы-потомки с их потомками и т. д., не включая узлы-атрибуты и узлы пространств имен — рекурсивное применение оси `child`;
- ❑ `descendant-or-self` — сам узел и все его потомки кроме узлов-атрибутов и узлов пространств имен — объединение осей `self` и `descendant`;
- ❑ `following` — все узлы, лежащие "ниже" текущего узла без узлов-потомков, узлов-атрибутов и узлов пространств имен;
- ❑ `following-sibling` — узлы с тем же непосредственным предком, что и у текущего узла, следующие за текущим узлом в том порядке, в котором они записаны в документе; если текущий узел является узлом-атрибутом или узлом пространств имен, то ось пуста;
- ❑ `attribute` — узлы-атрибуты текущего узла-элемента; для других видов узлов эта ось пуста;
- ❑ `namespace` — узлы пространств имен текущего узла-элемента; для других видов узлов эта ось пуста.

Пять осей, приведенных далее, направляют поиск "вверх", к корневому узлу документа (*reverse axis*):

- ❑ `parent` — непосредственный предок текущего узла; если его нет, то ось пуста;
- ❑ `ancestor` — все узлы-предки текущего узла;
- ❑ `ancestor-or-self` — текущий узел вместе со всеми своими предками — объединение осей `parent` и `ancestor`;
- ❑ `preceding` — узлы, предшествующие текущему узлу в документе XML; не содержит предков текущего узла, узлов-атрибутов и узлов пространств имен;
- ❑ `preceding-sibling` — узлы с тем же непосредственным предком, что и у текущего узла, следующие перед текущим узлом в том порядке, в котором они записаны в документе; если текущий узел является узлом-атрибутом или узлом пространств имен, то ось пуста.

Таким образом, оси `self` и `parent` определяют только по одному узлу. Оси `descendant`, `ancestor`, `following` и `preceding` определяют непересекающиеся области узлов, которые вместе с осью `self` содержат все узлы дерева документа, за исключением узлов-атрибутов и узлов пространств имен.

На рис. 6.1 показаны области, определяемые различными осями относительно текущего элемента, за исключением осей атрибутов и осей пространств имен.

В шаге поиска, определяемом осью, сама ось может отсутствовать, при этом (за одним исключением) по умолчанию подразумевается ось `child`.

Поэтому предыдущий пример можно записать проще:

```
section[1]
```

С другой стороны, полная запись первого примера этого раздела выглядит так:

```
/child::contract/child::section/child::paragraph
```

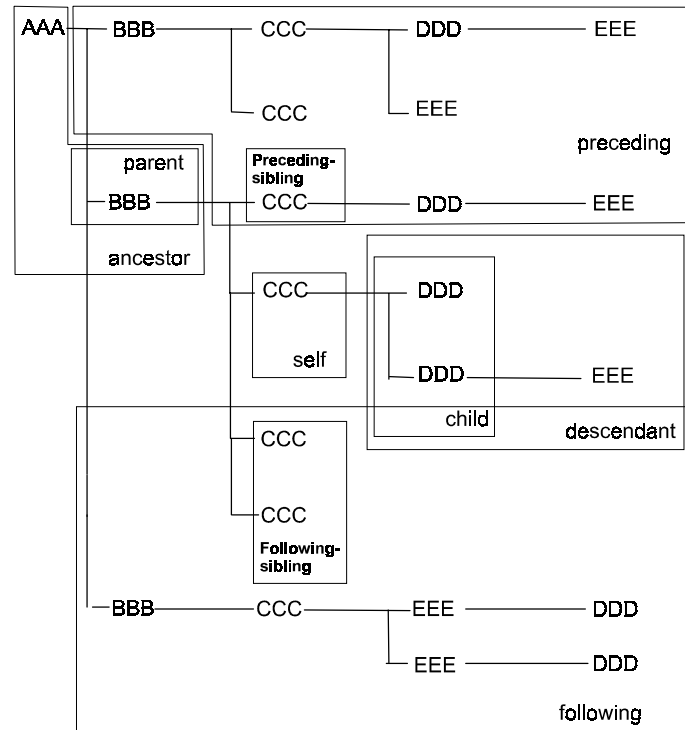


Рис. 6.1. Области, определяемые осями

Оговоренное исключение касается поиска атрибутов. Если шаг поиска отыскивает атрибуты, то по умолчанию подразумевается ось `attribute`, например, запись

```
//attribute(@name)
```

эквивалентна записи

```
//attribute::attribute(@name)
```

Запись оси `attribute::` часто сокращается до одного символа "собачки" `@`. Такое сокращение нам уже встречалось в предыдущей главе и не раз встре-

тится далее. Пользуясь этим сокращением можно сделать еще одну эквивалентную запись предыдущего выражения:

```
//@attribute(@name)
```

Тесты узла

После того как с помощью оси определена область поиска, в ней отыскиваются узлы определенных видов или узлы с определенными именами. Для этого применяется *тест узла* (node test). В соответствии со своими задачами тесты узла делятся на две группы. Рассмотрим каждую из них подробнее.

Тест по имени узла

Первая разновидность теста — *тест по имени узла* (name test) — содержит просто уточненное имя типа QName, состоящее из префикса и локального имени, отделенного от префикса двоеточием. Он отбирает узлы по имени, записанному в нем. Большинство примеров, приведенных в предыдущей главе для схемы `xpointer()`, применяло тест по имени. В тесте можно использовать два шаблона уточненных имен.

Если вместо префикса имени стоит звездочка, например `*:section`, то отбираются узлы с локальным именем `section` в любом пространстве имен.

Если вместо локального имени стоит звездочка, например `xsd:*`, то отбираются узлы с любыми локальными именами, объявленными в указанном пространстве имен.

Приведем примеры. Пусть текущий узел дерева носит имя `person`.

1. Сам узел `person`:

```
self::person
```

2. Все непосредственные потомки узла `person`:

```
child::person или просто person
```

3. Все потомки узла `person`:

```
descendant::person
```

4. Все потомки узла `person` и сам узел `person`:

```
descendant-or-self::person
```

5. Родительский узел узла `person`:

```
parent::person
```

6. Все предки узла `person`:

```
ancestor::person
```

7. Все атрибуты узла person:

```
attribute::person
```

Тест по виду узла

Тест по виду узла (kind test) записывается в функциональной форме и может быть одним из следующих:

- ☐ `node()` — отбирает узел любого вида;
- ☐ `text()` — отбирает текстовые узлы;
- ☐ `comment()` — отбирает узлы-комментарии;
- ☐ `element()` — отбирает все узлы-элементы, эквивалентно `element(*)` и `element(*,*)`;
- ☐ `element(name)` — отбирает все узлы-элементы с именем `name`;
- ☐ `element(name, type)` — отбирает узлы-элементы с именем `name` и типом `type`, определенным в схеме XSD документа. На месте первого или второго аргумента может стоять звездочка `*`, означающая "все элементы" или "все типы". Второй аргумент может заканчиваться словом `nillable`, показывающим, что тип определен в схеме XSD со значением `xsi:nil="true"`;
- ☐ `attribute()` — отбирает все узлы-атрибуты; эквивалентно `attribute(@*)` и `attribute(@*,*)`;
- ☐ `attribute(@name)` — отбирает все узлы-атрибуты с именем `name`;
- ☐ `attribute(@name, type)` — отбирает все узлы-атрибуты с именем `name` и типом `type`;
- ☐ `processing-instruction()` — отбирает все инструкции по обработке;
- ☐ `processing-instruction(name)` — отбирает инструкции по обработке с именем обрабатывающей программы `name`;
- ☐ `document-node()` — отбирает все корневые узлы документа;
- ☐ `document-node(element(args))` — отбирает все корневые узлы документа, содержимое которых состоит в точности из одного вложенного элемента, удовлетворяющего тесту `element(args)`. Тест `element()` может принимать любую из трех перечисленных выше форм.

Тесты узла первой версии языка XPath включали в себя только `node()`, `text()`, `comment()` и `processing-instruction()`.

Таким образом, шаг поиска `child::node()` отбирает все непосредственные узлы-потомки текущего узла, за исключением узлов-атрибутов и узлов пространств имен, а шаг `child::element()` отбирает только непосредственные

узлы-потомки вида узлы-элементы. И первую, и вторую запись можно сделать короче, опустив ось `child`, которая принимается по умолчанию.

Тест `self::comment()` отберет текущий узел, если это узел-комментарий.

Тест `descendant-or-self::text()` отберет все текстовые узлы, вложенные в текущий узел, и сам текущий узел, если это текстовый узел.

Тест `attribute::attribute(@id)` отберет все узлы-атрибуты с именем `id`. Эту запись можно сократить до записи `attribute(@id)`.

Для некоторых часто используемых сочетаний оси и теста придуманы стандартные сокращения, которые приведены далее.

- ❑ Запись `/descendant-or-self::node()` сокращается до записи `//`, т. е. отбор текущего узла вместе со всеми его потомками за исключением узлов-атрибутов и узлов пространств имен понимается по умолчанию и вообще не записывается, остаются только ограничивающие наклонные черточки. Это сокращение мы уже использовали в предыдущей главе.
- ❑ Запись `attribute::node()`, отбирающая все атрибуты текущего узла, сокращается применением звездочки до записи `attribute::*`.
- ❑ Запись `namespace::node()` — все узлы пространств имен текущего узла — сокращается до `namespace::*`.
- ❑ Для других осей запись вида `axis::element()`, отбирающая все узлы-элементы в области, определяемой осью `axis`, сокращается до `axis::*`, например, `ancestor::*` или `following::*`.
- ❑ Следуя предыдущему правилу и правилу умолчания для осей, запись `child::element()` сокращается просто до звездочки `*` и отбирает все узлы-элементы — непосредственные потомки текущего узла.
- ❑ Запись оси `attribute::` сокращается до "собачки" `@`, например, `@*`, `@name`. Примеры такого сокращения нам уже встречались в предыдущей главе.
- ❑ Запись `self::node()`, означающая "текущий узел", сокращается до точки, подобно тому, как обозначается текущий каталог в файловой системе UNIX.
- ❑ Запись `parent::node()`, означающая "родительский узел", сокращается до двух точек, как "родительский каталог" в файловой системе UNIX.

Перейдем к примерам. В листинге 6.1 записан документ XML, по которому построено дерево на рис. 6.1.

Листинг 6.1. Документ XML

```
<AAA>
  <BBB>
```

```
<CCC>
  <DDD id="1" name="d1">
    <EEE />
  </DDD>
  <EEE />
</CCC>

<CCC />

</BBB>
<BBB>

  <CCC>
    <DDD id="2">
      <EEE />
    </DDD>
  </CCC>

  <CCC>
    <DDD id="3" name="d3" />
    <DDD id="4">
      <EEE />
    </DDD>
  </CCC>

  <CCC />

  <CCC />

</BBB>
<BBB>
  <CCC>

    <EEE>
      <DDD id="5" />
    </EEE>

    <EEE>
      <DDD />
    </EEE>

  </CCC>
</BBB>
</AAA>
```

Запишем несколько примеров поиска в документе, представленном листингом 6.1.

1. Узел документа:
`/`
2. Корневой узел-элемент:
`/AAA`
3. Все элементы CCC, вложенные в BBB:
`//BBB/CCC`
4. Все элементы CCC в документе:
`//CCC`
5. Все элементы, лежащие во всех элементах CCC:
`//CCC/*`
6. Все элементы, лежащие во всех элементах CCC, вложенных в BBB:
`/AAA/BBB/CCC/*`
7. Все элементы CCC третьего уровня вложенности:
`/*/*/CCC`
8. Все элементы документа:
`//descendant::*`
9. Все элементы, вложенные во все элементы BBB:
`//AAA/BBB/descendant::*`
10. Все элементы, вложенные в элемент CCC, где бы он ни был:
`//CCC/descendant::*`
11. Все непосредственные предки всех элементов DDD:
`//DDD/parent::*`
12. Все предки всех элементов DDD:
`//DDD/ancestor::*`
13. Все элементы с атрибутом id:
`//attribute(@id)`

Упражнения

1. Укажите путь ко всем элементам EEE пятого уровня вложенности.
2. Укажите путь ко всем элементам EEE четвертого уровня вложенности.
3. Отметьте все элементы EEE в документе листинга 6.1.

4. Считая текущим узел CCC на рис. 6.1, опишите каждую из областей, отмеченных на рис. 6.1.
5. Перечислите способы выбора всех узлов дерева документа XML.
6. Считая текущим элемент DDD с атрибутом `id="3"`, опишите все предыдущие и следующие за ним элементы.

Предикаты

После отбора, проведенного осью поиска и тестом узла, из полученной последовательности узлов можно выделить те, которые удовлетворяют весьма сложным условиям, задаваемым в третьей, необязательной части шага поиска — предикате. *Предикат* записывается в квадратных скобках и представляет собой логическое выражение, значением которого может быть истина `true` или ложь `false`. Например, `[a=5]`, `[x<2.6]`, `[a>1 and a<8]`.

Последовательность узлов, к которой применяется предикат, сортируется в направлении, указанном осью: от начала документа к его концу, или наоборот. Отсортированные узлы нумеруются, начиная с 1, и просматриваются по очереди. Для каждого узла вычисляется предикат. Узлы, для которых значение предиката равно `false`, отбрасываются.

Номер текущего узла называется его *позицией* (`position`) в отсортированной последовательности узлов. Точное значение позиции текущего узла возвращается функцией `position()`. Поскольку очень часто узел выбирается по своему порядковому номеру, предикат вида

```
[position()=2]
```

записывают сокращенно как `[2]`. В предыдущей главе мы видели примеры такой записи.

Количество узлов в последовательности можно получить функцией `last()`. Используя ее и применяя указанное сокращение, последний узел можно выделить предикатом `[last()]`, а предпоследний — предикатом `[last()-1]`.

Приведем примеры по листингу 6.1.

1. Первый из элементов CCC, вложенных в элементы BBB:

```
/AAA/BBB/CCC[1]
```

2. Последний из элементов CCC, вложенных в элементы BBB:

```
/AAA/BBB/CCC[last()]
```

3. Все элементы документа с двумя вложенными элементами CCC:

```
//*[count(CCC)=2]
```

4. Все элементы документа с двумя любыми вложенными элементами:

```
//*[count(*)=2]
```

5. Все элементы DDD, имеющие атрибут name:

```
//DDD[@name]
```

6. Элемент DDD с атрибутом id, равным 3:

```
//DDD[@id="3"]
```

7. Все элементы DDD, не имеющие атрибут name:

```
//DDD[not (@name)]
```

Упражнения

1. Подсчитайте число всех элементов документа.
2. Подсчитайте число всех элементов без атрибутов.
3. Выберите те элементы документа, у которых есть вложенные элементы.

Шаг, направляемый фильтром

Шаг, направляемый фильтром, использует вместо оси и теста узла *первичное выражение* (primary expression):

Первичное выражение [Предикат]

Первичное выражение выдает в качестве результата последовательность узлов и/или атомарных значений, которая затем фильтруется предикатом.

Первичное выражение — это:

- ☐ число типа `xs:integer`, `xs:decimal`, `xs:double`;
- ☐ строка символов типа `xs:string`;
- ☐ значение переменной, начинающееся со знака доллара \$;
- ☐ вызов функции;
- ☐ наконец, произвольное выражение, заключенное в скобки.

Поэтому на шаге, направляемом фильтром, можно применять любое выражение, надо только заключить его в скобки.

Настало время познакомиться подробнее с выражениями языка XPath.

Выражения

Мы уже говорили о том, что все конструкции, имеющиеся в языке XPath, понимаются как выражения, в результате вычисления которых получается последовательность узлов и/или атомарных значений. Выражения состояются из констант различных типов, переменных, вызовов функций, связанных знаками операций и скобками. Но это не все. Мы видели выражения,

определяющие путь. Кроме них, в языке XPath есть условные выражения, выражения-циклы, выражения-последовательности, кванторы. Вместе они образуют целый язык программирования, на котором можно записать весьма сложный алгоритм.

Более того, набор выражений в скобках, разделенных запятыми, тоже понимается как выражение. Это так называемая "операция запятая". Поэтому всюду, где можно записать выражение, можно записать и набор выражений в скобках, если не оговорено противное.

Рассмотрим последовательно различные виды операций и выражений, их содержащих. Но сначала посмотрим, как в языке XPath используются переменные.

Переменные

Имя переменной в языке XPath — это обычное для XML имя типа QName. Для того чтобы отличить переменные от других имен, перед именем переменной ставится знак доллара \$, например, \$var, \$n, \$i. Переменные обычно определяются во внешних языках, использующих XPath, скажем, в языке XSLT или в языке XQuery. Сам язык XPath не определяет переменные явно, в нем нет описаний типов и оператора присваивания. Переменные, как мы увидим ниже, появляются только в циклах и кванторах.

Арифметические операции

В арифметические выражения входят числа типа `xs:integer`, `xs:decimal`, `xs:double` и переменные этих типов. С этими данными выполняются операции сложения +, вычитания -, умножения * и деления `div`, а также операция взятия остатка от деления `mod` и операция целочисленного деления `idiv` двух целых чисел.

Арифметические операции, кроме операций `mod` и `idiv`, можно выполнять не только с числами, но и с датами типа `xs:datetime`, `xs:date`, `xs:time`, `xdt:dayTimeDuration`, `xdt:yearMonthDuration`. Разумеется, арифметические операции с датами выполняются только там, где это имеет смысл.

Как видите, для обозначения деления применяется слово `div`, а не наклонная черта, потому что наклонная черта в языке XPath разделяет шаги поиска. Обратите внимание на еще одно важное обстоятельство.

Поскольку дефис в XML применяется в именах, в арифметических выражениях его надо отделять пробелами. Например, запись `a-b` будет понята как какое-то имя, а запись `a - b` — как вычитание.

Вычисление арифметических операций лучше всего пояснить на примерах.

Выражение `7 div 2` даст в результате число 3.5.

Выражение `7 idiv 2` даст в результате целое число 3.

Выражение `-7 idiv 2` даст в результате отрицательное целое число `-3`.

Выражение `7 mod 2` даст в результате целое число `1`.

Выражение `7.6 mod 2` даст в результате число `1.6`.

Выражение `-7.6 mod 2` даст в результате число `-1.6`.

Выражение `-7.6 mod -2` даст в результате число `-1.6`.

В арифметических операциях могут участвовать строки, если они содержат запись чисел. Например, в результате сложения `"2"+"2"` получится число `4`.

Сравнения

В языке XPath применяются обычные операции сравнения: "равно" `=`, "не равно" `!=`, "больше" `>`, "меньше" `<`, "больше или равно" `>=` и "меньше или равно" `<=`. В результате сравнения получается логическое значение "истина" `true` или "ложь" `false`. В этих операциях сравнения могут участвовать числа, строки символов, даты.

В двух других операциях сравнения `is` и `isnot` участвуют узлы. Операция `is` дает значение `true`, если сравниваемые узлы совпадают, и `false` — в противном случае, операция `isnot`, наоборот, дает `true`, если узлы не совпадают. Например:

```
//DDD[@id="3"] is //DDD[@name="d3"]
```

Еще две операции сравнения `>>` и `<<` тоже применяются к узлам. Они отслеживают порядок записи узлов в документе. Операция `<<` дает истину `true`, если ее левый операнд встречается в документе раньше, чем правый операнд, операция `>>`, наоборот, дает в этом случае значение `false`. Например:

```
//DDD[@name="d1"] << //DDD[@id="1"]
```

Логические операции

Логические операции `and` и `or` применяются к логическим данным, имеющим значения `true` или `false`. В результате получается логическое значение `true` или `false` по следующим правилам:

- ❑ операция `A and B` даст в результате значение `true` в том и только том случае, когда и `A` и `B` имели значение `true`;
- ❑ операция `A or B` даст в результате значение `false` в том и только том случае, когда и `A` и `B` имели значение `false`.

Например, выражение `1 < 2 and 3 = 4` даст `false`, а выражение `1 < 2 or 3 = 4` даст `true`.

Приоритет операции `and` выше приоритета операции `or`, поэтому выражение `false or true and false`

будет вычисляться справа налево. Порядок вычисления всегда можно изменить скобками.

Заметьте, что в языке XPath нет операции отрицания. Вместо нее используется функция `not()`, например, `not(1 < 2 and 3 = 4)` даст в результате значение `true`.

Еще одна логическая операция проверяет наличие элемента, заданного в ее первом операнде, в последовательности, заданной во втором операнде. Операция записывается словами `instance of`. Например, выражение:

```
1000 instance of xs:integer
```

даст истину `true`, а результатом выражения

```
/AAA/BBB[1] instance of attribute()
```

будет ложь `false`.

Условные выражения

Условное выражение имеет вид:

```
if (выражение1) then выражение2 else выражение3
```

Оно вычисляется следующим образом. Сначала вычисляется `выражение1`. Оно должно дать логический результат: `true` или `false`. Если получилось значение `true`, то вычисляется `выражение2`, его значение и будет результатом всего условного выражения. `Выражение3` при этом даже не вычисляется. Если получилось значение `false`, то результатом условного выражения будет результат `выражения3`, `выражение2` при этом не вычисляется. Например:

```
if ($p/sex = "M") then "father" else "mother"
```

В отличие от многих других языков программирования, ветвь `else` в условном выражении нельзя опускать, она обязательна. Если при невыполнении условия ничего не надо делать, то просто записывается пара скобок, означающая пустое выражение:

```
if (/count/price > 0) then /count/price else ()
```

Циклы

Циклы в языке XPath образуются с помощью конструкции вида:

```
for $имя in выражение1 return выражение2
```

Переменная `$имя` последовательно принимает значения узлов и/или атомарных выражений, полученных в результате вычисления `выражения1`. При этом

каждый раз вычисляется `выражение2`, в котором, как правило, применяется переменная `$имя`. Например, следующий цикл, использующий дерево лис-тинга 6.1, сделает три повтора, поскольку выражение `//BBB`, определяющее путь, даст последовательность узлов, состоящую из трех узлов `/AAA/BBB`.

```
for $n in //BBB return $n/CCC
```

Этот цикл выберет три узла `CCC`, вложенные в узлы `BBB`. В следующем, более сложном примере, с помощью функции `sum()` подсчитывается суммарная стоимость некоторой продукции, полученная умножением цены на количество продукции каждого вида и последующим суммированием:

```
sum(for $cost in /order return $cost/price * $cost/quantity)
```

Внутри цикла `выражение2` может также содержаться цикл `for`. Так в языке XPath получаются вложенные циклы. Например:

```
for $x in //BBB return
  (for $y in $x/CCC return $y/DDD)
```

Такую запись можно сократить следующим образом:

```
for $x in //BBB, $y in $x/CCC return $y/DDD
```

Эта конструкция может повторяться. В общем виде запись цикла выглядит так:

```
for $имя1 in выражение1, $имя2 in выражение2, ... return выражение
```

В качестве выражений в цикле можно использовать простые последовательности-перечисления, например, в результате работы следующего цикла

```
for $i in (1, 2, 3), $j in (10, 20) return ($i + $j)
```

получится последовательность чисел 11, 21, 12, 22, 13, 23.

Последовательность-перечисление может быть более сложной. В ней могут встречаться диапазоны, образованные с помощью слова `to`, например, последовательность:

```
(10, 1 to 5, 20, 15 to 10)
```

состоит из чисел

```
(10, 1, 2, 3, 4, 5, 20, 15, 14, 13, 12, 11, 10)
```

Как видите, диапазон может идти и в сторону увеличения, и в сторону уменьшения чисел.

Кванторы

Кванторы существования и всеобщности впервые появились в математической логике для записи в математической форме существования объектов

с определенным свойством, например "существуют белые лошади", или для записи всеобщего обладания свойством — "у всех лошадей четыре ноги". Затем кванторы распространились на теорию множеств для выделения хотя бы одного элемента множества с заданным свойством — "существует простое число" — или всех элементов, обладающих свойством — "все простые числа, большие 2, нечетны".

В языке XPath кванторы (quantifiers) используются для выделения хотя бы одного узла из последовательности узлов или выделения всех узлов последовательности с определенным свойством. *Квантор существования* (existential quantifier) записывается так:

```
some $имя in выражение1 satisfies выражение2
```

Здесь сначала вычисляется `выражение1`. Переменная `$имя` одно за другим принимает значения узлов из последовательности, полученной в результате вычисления `выражения1`. Эти значения используются в `выражении2`. Как только `выражение2` даст значение `true`, вычисления прекращаются и квантор выдает значение `true`. Если этого не случится, значением квантора будет `false`.

Например, выясним, есть ли у нас хотя бы один сотрудник по имени "Федор":

```
some $n in /person/name satisfies $n = "Федор"
```

Квантор всеобщности (universal quantifier) записывается так:

```
every $имя in выражение1 satisfies выражение2
```

Он действует аналогично квантору существования, но значение `true` получается только в том случае, когда `выражение2` всегда равно `true`. Если хотя бы один раз `выражение2` примет значение `false`, то вычисления прекращаются и значением квантора будет `false`.

Например, проверим, все ли сотрудники имеют высшее образование:

```
every $n in /person/education satisfies $n = "высшее"
```

Операции с множествами

Язык XPath имеет дело с последовательностями, упорядоченными множествами. При работе с множествами очень полезны операции объединения, пересечения и дополнения множеств. Эти операции введены и в язык XPath. Они применяются к последовательностям узлов.

Операция объединения последовательностей записывается словом `union` или вертикальной чертой `|`, оставшейся от первой версии языка XPath. В результате ее применения получается последовательность, содержащая все узлы и первой, и второй последовательности. Следующий пример выбирает все элементы BBB и CCC, вложенные во все элементы AAA:

```
//AAA/(BBB union CCC)
```

Узлы, повторяющиеся в обеих последовательностях, заносятся в объединение только один раз.

Операция пересечения последовательностей записывается словом `intersect`. В результате пересечения получается последовательность, содержащая только общие узлы двух последовательностей. Например:

```
doc("bids.xml")/*bid[bid-amount > 1000.00]
intersect
doc("bids.xml")/*bid[bid-date > date("2002-01-01")]
```

Операция дополнения `except` выбирает из первой последовательности узлы, не содержащиеся во второй последовательности:

```
doc("items.xml")//itemno
except
doc("bids.xml")//itemno
```

Функции

В язык XPath встроено множество функций, облегчающих поиск узлов. Мы уже применяли функции `position()`, `last()`, `count()`, `sum()`, `not()`. Более того, все операции, встроенные в язык: сложение, вычитание и др., определяются в XPath через соответствующие функции. Например, сложением чисел $A + B$ занимается функция `numeric-add(A, B)`, проверкой равенства чисел $A = B$ — функция `numeric-equal(A, B)` и т. д.

Полный список встроенных функций языка XPath 2.0 и их строгие определения приведены в документе "XQuery 1.0 and XPath 2.0 Functions and Operators", черновая версия которого опубликована на странице <http://www.w3.org/TR/2003/WD-xpath-functions-20030502/>. Имена этих функций расположены в пространстве имен <http://www.w3.org/2003/05/xpath-functions>, которому обычно дают префикс `fn`. Имена функций — операций сложения, вычитания и др., — объявлены в другом пространстве имен <http://www.w3.org/2003/05/xpath-operators>, префикс которого обычно равен `op`.

Мы не будем подробно рассматривать все встроенные функции языка XPath 2.0, тем более, что их состав может измениться в окончательной версии. Познакомимся только с наиболее полезными и часто употребляемыми функциями.

Числовые функции

К числовым функциям относятся следующие:

- `floor(x)` — округление аргумента x до меньшего целого числа. Например, `floor(3.5)` равно 3, `floor(-3.5)` равно -4;

- ❑ `ceiling(x)` — округление аргумента `x` до большего целого числа. Например, `ceiling(3.5)` равно 4, `ceiling(-3.5)` равно -3;
- ❑ `round(x)` — округление аргумента `x` до ближайшего целого числа, эквивалентно функции `floor(x + 0.5)`;
- ❑ `round-half-to-even(x, p)` — округление аргумента `x` в сторону ближайшего числа с точностью, определяемой вторым аргументом `p`, который показывает положение младшей значащей цифры справа от десятичной точки, если `p > 0`, и слева от десятичной точки, если `p < 0`. Если последняя отбрасываемая цифра равна 5, то последняя оставляемая цифра заменяется ближайшей четной. Например, `round-half-to-even(124.675, 2)` равно 124.68, а `round-half-to-even(124.675, -2)` равно 100;
- ❑ `round-half-to-even(x)` — эквивалентна функции `round-half-to-even(x, 0)`.

Строковые функции

Используются следующие строковые функции:

- ❑ `compare(s1, s2)` — сравнивает лексикографически строку `s1` со строкой `s2` и возвращает:
 - -1, если `s1` меньше `s2`;
 - 0, если строки совпадают;
 - +1, если `s1` больше `s2`;
- ❑ `concat(s1, s2, s3, ...)` — сцепляет строки `s1, s2, s3, ...` в одну строку;
- ❑ `string-join(s1, s2, ...), sep` — сцепляет строки `s1, s2, ...` в одну строку, вставляя между ними строку `sep`;
- ❑ `starts-with(s, sub)` — возвращает значение `true`, если строка `s` начинается с символов строки `sub`;
- ❑ `ends-with(s, sub)` — возвращает значение `true`, если строка `s` заканчивается символами строки `sub`;
- ❑ `contains(s, sub)` — возвращает `true`, если строка `s` содержит строку `sub`;
- ❑ `substring(s, n, len)` — выделяет из строки `s` подстроку длиной `len` символов, начинающуюся с позиции `n`; номера символов начинаются с 1;
- ❑ `substring(s, n)` — выделяет из строки `s` подстроку, начинающуюся с позиции `n` до конца строки. Номера символов начинаются с 1;
- ❑ `string-length(s)` — вычисляет число символов в строке `s`;
- ❑ `string-length()` — вычисляет число символов в строке, представляющей текущий узел;

- ❑ `substring-before(s, sub)` — выделяет из строки `s` подстроку, предшествующую первому появлению строки `sub` в строке `s`;
- ❑ `substring-after(s, sub)` — выделяет из строки `s` подстроку, следующую за последним символом первого появления строки `sub` в строке `s`;
- ❑ `normalize-space(s)` — удаляет из строки `s` "лишние" пробелы, т. е. удаляет начальные и конечные пробелы, а из нескольких подряд идущих пробелов оставляет только один;
- ❑ `upper-case(s)` — переводит все буквы строки `s` в верхний регистр;
- ❑ `lower-case(s)` — переводит все буквы строки `s` в нижний регистр;
- ❑ `translate(s, from, to)` — заменяет каждый символ строки `s`, встретившийся на `N`-м месте в строке `from`, на `N`-й символ строки `to`;
- ❑ `matches(s, pat)` — возвращает `true`, если строка `s` удовлетворяет шаблону `pat`, который может быть любым регулярным выражением;
- ❑ `replace(s, pat, to)` — заменяет подстроки строки `s`, удовлетворяющие шаблону `pat`, на подстроку `to`;
- ❑ `tokenize(s, pat)` — разбивает строку `s` на последовательность строк, используя в качестве разделителя шаблон `pat`.

Функции даты и времени

Следующие функции возвращают различные составляющие даты и времени, заданные своим аргументом `t`: год, месяц, день, час, минуты, секунды и т. д. Тип аргумента совпадает с последним словом в имени функции:

- ❑ `get-years-from-yearMonthDuration(t)` — возвращает год;
- ❑ `get-months-from-yearMonthDuration(t)` — возвращает месяц;
- ❑ `get-days-from-dayTimeDuration(t)` — возвращает день месяца;
- ❑ `get-hours-from-dayTimeDuration(t)` — возвращает час;
- ❑ `get-minutes-from-dayTimeDuration(t)` — возвращает минуты;
- ❑ `get-seconds-from-dayTimeDuration(t)` — возвращает секунды;
- ❑ `get-year-from-dateTime(t)` — возвращает год;
- ❑ `get-month-from-dateTime(t)` — возвращает месяц;
- ❑ `get-day-from-dateTime(t)` — возвращает день месяца;
- ❑ `get-hours-from-dateTime(t)` — возвращает час;
- ❑ `get-minutes-from-dateTime(t)` — возвращает минуты;
- ❑ `get-seconds-from-dateTime(t)` — возвращает секунды;
- ❑ `get-timezone-from-dateTime(t)` — возвращает временную зону;

- ❑ `get-year-from-date(t)` — возвращает год;
- ❑ `get-month-from-date(t)` — возвращает месяц;
- ❑ `get-day-from-date(t)` — возвращает день месяца;
- ❑ `get-timezone-from-date(t)` — возвращает временную зону;
- ❑ `get-hours-from-time(t)` — возвращает час;
- ❑ `get-minutes-from-time(t)` — возвращает минуты;
- ❑ `get-seconds-from-time(t)` — возвращает секунды;
- ❑ `get-timezone-from-time(t)` — возвращает временную зону;
- ❑ `current-dateTime()` — возвращает текущую дату и время;
- ❑ `current-date()` — возвращает текущую дату;
- ❑ `current-time()` — возвращает текущее время.

Функции узлов

Мы уже встречались со следующими функциями узлов:

- ❑ `position()` — возвращает порядковый номер, начинающийся с 1, текущего узла в последовательности узлов;
- ❑ `last()` — возвращает количество узлов в текущей последовательности узлов;
- ❑ `count(expr)` — возвращает число узлов в последовательности, задаваемой выражением `expr`.

В стандартную библиотеку входят еще такие функции:

- ❑ `name()` — уточненное имя текущего узла;
- ❑ `name(node)` — уточненное имя узла `node`;
- ❑ `local-name()` — локальное имя текущего узла;
- ❑ `local-name(node)` — локальное имя узла `node`;
- ❑ `namespace-uri()` — пространство имен имени текущего узла;
- ❑ `namespace-uri(node)` — пространство имен имени узла `node`;
- ❑ `root()` — узел документа, в котором расположен текущий узел;
- ❑ `root(node)` — узел документа, в котором расположен узел `node`.

Функции последовательности

Аргументом `seq` следующих функций может быть любое выражение, в результате вычисления которого получается последовательность узлов и атомарных значений:

- ❑ `concatenate(seq1, seq2)` — сцепляет две последовательности `seq1` и `seq2`;
- ❑ `item-at(seq, n)` — возвращает `n`-й элемент последовательности `seq`;
- ❑ `index-of(seq, node)` — возвращает номер элемента `node` последовательности `seq`;
- ❑ `empty(seq)` — дает значение `true`, если последовательность `seq` пуста;
- ❑ `exists(seq)` — дает значение `true`, если последовательность `seq` непустая;
- ❑ `distinct-nodes(seq)` — удаляет из последовательности повторяющиеся узлы;
- ❑ `distinct-values(seq)` — удаляет из последовательности повторяющиеся значения;
- ❑ `insert-before(seq, n, node)` — вставляет элемент `node` в последовательность `seq` перед элементом, стоящим в позиции `n`;
- ❑ `remove(seq, n)` — удаляет из последовательности `seq` элемент с номером `n`;
- ❑ `subsequence(seq, n, len)` — выделяет из последовательности `seq` подпоследовательность, начинающуюся с позиции `n`, состоящую из `len` элементов;
- ❑ `subsequence(seq, n)` — выделяет из последовательности `seq` подпоследовательность от позиции `n` до конца.

Функции, создающие последовательности

К ним относятся следующие функции:

- ❑ `to(n1, n2)` — создает последовательность целых чисел от `n1` до `n2`;
- ❑ `id(ref1, ref2, ...)` — создает последовательность из узлов-элементов с идентификаторами, ссылки на которые `ref1`, `ref2`, ... типа `IDREF` перечислены в аргументе функции;
- ❑ `idref(id1, id2, ...)` — создает последовательность узлов, содержащих ссылки типа `IDREF` на узлы, идентификаторы которых `id1`, `id2`, ... типа `ID` перечислены в аргументе функции;
- ❑ `doc(uri)` — возвращает корневой узел документа XML, найденный по его адресу URI, заданному аргументом функции в виде строки `uri`;
- ❑ `collection(uri)` — создает последовательность корневых узлов документов, полученных по адресу `uri`;
- ❑ `input()` — возвращает входную последовательность.

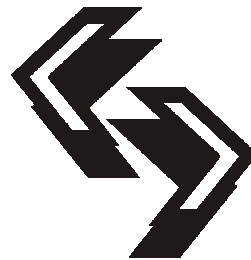
Упражнения

1. Вычислите выражение `//BBB[position() mod 2 = 0]`.
2. Вычислите выражение `substring("What is this?", 5, 3)`.
3. Вычислите выражение `//BBB[position() = floor(last() div 2 + 0.5) or position() = ceiling(last() div 2 + 0.5)]`

Вопросы для самопроверки

1. Для чего применяется язык XPath?
2. Как строится дерево документа для XPath?
3. Какие составляющие документа образуют узлы дерева?
4. Какие виды узлов могут быть предками узлов дерева?
5. Что такое ось поиска?
6. Что проверяется в тесте узла?
7. Что получается в результате вычисления предиката?
8. Что получается в результате вычисления каждого выражения XPath?
9. Что получится в результате вычисления выражения `*/@*`?
10. Как узнать позицию элемента в последовательности?
11. Как подсчитать число элементов в последовательности?
12. Что вычисляет квантор?

ГЛАВА 7



Язык запросов XQuery

После того как язык XML научился делать ссылки на языке XLink, создавать указатели XPointer на каждую точку документа и тонко адресовать любые части документа с помощью XPath, остался последний шаг: научиться извлекать найденную по адресу информацию из любых участков документа и оформлять ее в виде элементов документа XML. Язык XQuery как раз и призван сделать этот последний шаг.

Поскольку перед извлечением информации из документа надо ее отыскать, язык XQuery разрабатывался в тесной связи с языком XPath 2.0. Более того, большинство конструкций языка XPath 2.0, которые мы рассмотрели в *главе 6*, созданы в расчете на написание запросов. Многие выражения XPath, такие как выражение `//AAA/BVV`, делают запрос на поиск информации в документе XML. Поэтому язык XQuery можно считать расширением языка XPath 2.0. В нем можно применять почти все, что есть в языке XPath 2.0.

Единственное ограничение — пока, в первой версии языка XQuery, выражения, определяющие путь, можно направлять только по 6 осям из 13, имеющихся в языке XPath (*см. главу 6*): `child` (по умолчанию), `descendant`, `attribute`, `self`, `descendant-or-self` и `parent`.

Основной единицей языка XQuery, как и языка XPath, служит выражение, причем набор выражений, перечисленных через запятую, тоже считается выражением. Это так называемая "операция запятая", объединяющая несколько выражений в одно, вычисляемое последовательно слева направо. Результат вычисления выражения, как и в языке XPath, — последовательность узлов и/или атомарных значений. Виды узлов и типы атомарных значений таковы же, как и в XPath 2.0.

Запрос (`query`) в языке XQuery тоже оформляется как выражение. Разница заключается в том, что в результате вычисления выражения получается последовательность, а в результате выполнения запроса должен получиться один или несколько элементов XML, а то и целый документ XML. Значит,

в первую очередь язык XQuery должен научиться конструировать элементы XML. Посмотрим, как работают конструкторы элементов и других узлов.

Конструкторы

Конструктор узлов — это выражение, которое создает и добавляет к дереву документа новые узлы.

Конструкторы могут создавать узлы каждого вида из перечисленных в *главе 6*, за исключением узлов пространств имен. При этом узел-элемент можно создать средствами прямого или вычисляемого конструктора, узел-атрибут, корневой узел документа и текстовый узел — только вычисляемого конструктора, а узлы-комментарии и узлы инструкций по обработке просто записываются в конструкторе прямо в том виде, в каком они затем появятся в документе.

Прямой конструктор элемента

Самый простой способ сконструировать элемент XML — это записать его в явном виде с открывающим тегом, атрибутами, содержимым и закрывающим тегом. Эта форма называется *прямым конструктором элемента* (direct element constructor). Например:

```
<person id="92-3456" sex="male">
  <name>Иван Петров</name>
  <age>30</age>
</person>
```

Этот конструктор формирует и вносит в дерево новый узел-элемент `person` и вложенные в него узлы-элементы `name` и `age`, узлы-атрибуты `id` и `sex`, а также текстовые узлы с содержимым элементов и значениями атрибутов.

Выражения в содержимом конструктора

В прямом конструкторе, в содержимом конструируемого элемента, можно записать выражение в фигурных скобках. Оно будет вычислено, а результат вычисления подставлен в конструируемый элемент. Например, результат предыдущего примера будет тем же самым, если записать элемент `age` в таком виде:

```
<age>{ 10 + 20 }</age>
```

Заметьте, что если не написать фигурные скобки, то выражение не будет вычисляться, а попадет в конструируемый элемент как простой текст, и мы получим элемент `age` в следующем виде:

```
<age>10 + 20</age>
```

Если фигурные скобки надо понимать как простые символы, а не как команду вычисления выражения, то их следует удваивать:

```
"int main(){{return 0;}}"
```

В общем случае прямой конструктор элемента состоит из элемента XML, в нашем примере это элемент `person`, у которого могут быть атрибуты. В элемент могут быть вложены другие элементы, выражения в фигурных скобках и наборы символов — содержимое элемента. Конструктор преобразует наборы символов в текстовые узлы, вычисляет выражения, рекурсивно обрабатывает вложенные элементы и формирует новый узел-элемент с вложенными узлами-элементами, узлами-атрибутами и текстовыми узлами.

Каждое выражение после вычисления даст последовательность узлов и/или атомарных значений. Для узлов из этой последовательности конструктор создаст точную копию со всеми их узлами-потомками, узлами-атрибутами, узлами пространств имен, если они есть. Для каждой подпоследовательности идущих подряд атомарных значений конструктор создаст один текстовый узел, содержащий строковые представления атомарных значений с пробелом между ними.

В полученной после этого новой последовательности нет атомарных значений, а есть только узлы.

Затем все содержимое конструируемого элемента представляется одной последовательностью узлов, составленной из текстовых узлов конструктора, последовательностей, полученных после вычисления и преобразования выражений, и вложенных узлов-элементов, полученных после рекурсивной обработки вложенных в конструктор элементов. В этой последовательности не должно быть корневых узлов документа, а все узлы-атрибуты должны находиться в начале последовательности. Если эти условия не выполнены, то конструктор выдаст сообщение об ошибке и прекратит работу. Если же они выполнены, то конструктор сделает еще один шаг: он сольет все идущие подряд текстовые узлы, не оставляя между ними пробелов, в один текстовый узел.

После этого прямой конструктор создает узел-элемент, в нашем примере `person`, с узлами-потомками из последовательности, полученной на предыдущем шаге.

Выражения в атрибутах конструктора

В атрибутах конструктора тоже можно записать выражения в фигурных скобках, например:

```
<person id="92-3456" sex="{ /notion/sex[1] }">
```

Выражение вычисляется, каждый элемент полученной последовательности представляется строкой, а строки разделяются пробелами. Кроме выраже-

ний, значение атрибута может содержать наборы символов, которые сцепляются со строками, полученными из выражений, без всяких пробелов. Например:

```
<person id="9{1 + 1}-3456" sex="male">
```

Вычисляемый конструктор

Прямой конструктор использует заданные заранее имена элементов и атрибутов. Иногда это неудобно или невозможно сделать. В таких случаях применяют вычисляемые конструкторы.

Вычисляемые конструкторы (computed constructors) могут создавать узлы четырех видов: узлы-элементы, узлы-атрибуты, корневые узлы документа или текстовые узлы. Поэтому в начале конструктора надо указать вид создаваемого узла одним из слов `element`, `attribute`, `document` или `text`. После этого слова в фигурных скобках записывается выражение, конструирующее узел. Для узла-элемента и узла-атрибута нужно еще записать его имя, которое тоже можно задать выражением.

Вычисляемые конструкторы элемента и атрибута

Вот как можно задать вычисляемый конструктор для элемента `person`, приведенного в начале главы:

```
element person {
  attribute id { "92-3456" }
  attribute sex { /notion/sex[1] }
  element name { "Иван Петров" }
  element age { 10 + 20 }
}
```

Если надо сконструировать имя элемента и атрибута, то предыдущая запись дополнится еще несколькими выражениями и будет выглядеть примерно так:

```
element
{ /notion/name[1] }
{ attribute id { "92-3456" }
  attribute
    { /notion/attr[1] }
    { /notion/sex[1] }
  element name { "Иван Петров" }
  element age { 10 + 20 }
}
```

Конструктор корневого узла документа

Конструктор корневого узла документа нельзя вкладывать внутрь другого конструктора корневого узла документа, он должен быть внешним и применяется, если надо создать документ XML целиком. Непосредственно в нем не должно быть конструкторов атрибутов, хотя они могут быть во вложенных конструкторах. Например:

```
document {  
  <?xml version="1.0" ?>  
  element person {  
    attribute id { "92-3456" }  
    attribute sex { /notion/sex[1] }  
    element name { "Иван Петров" }  
    element age { 10 + 20 }  
  }  
}
```

Конструктор текстового узла

Конструктор текстового узла выполняется так: его содержимое вычисляется, каждый элемент полученной последовательности преобразуется в строку, строки сцепляются, и между ними оставляется пробел. Полученная строка становится содержимым текстового узла.

Выражение запроса FLWOR

Запрос к документу XML, как и все в языке XQuery, записывается в виде выражения, которое в простейшем случае выглядит так:

```
let $n := ( "Петр", "Иванов")  
return <name>{ $n }</name>
```

Здесь после знака присваивания `:=` может стоять произвольное выражение. Оно вычисляется, и переменная `$n` получает его значение, в данном случае — "Петр Иванов", которое подставляется в возвращаемый запросом элемент

```
<name>Петр Иванов</name>
```

В операторе `return` можно записать любое выражение, хотя чаще всего здесь располагается конструктор.

Вторая форма запроса начинается со слова `for`. Она выглядит и выполняется как цикл языка XPath 2.0. Вот простейший пример:

```
for $n in ( "Петр", "Иванов")  
return <name>{ $n }</name>
```


Здесь переменная `$n` последовательно принимает значения узлов и/или атомарных значений, входящих в результат вычисления выражения, стоящего после знака операции `in`. Для каждого значения переменной `$n` выполняется оператор `return`, и запрос возвращает несколько элементов:

```
<name>Петр</name>
<name>Иванов</name>
```

Переменная заголовка получает тип своего значения, но этот тип можно преобразовать, указав новый тип после слова `as`. Новый тип должен быть совместим со старым. Например:

```
for $n as xs:string in ("Петр", "Иванов")
```

или

```
let $n as xs:string := ( "Петр", "Иванов")
```

У заголовка `for` есть одна дополнительная возможность, которой нет у заголовка `let`. Параллельно переменной заголовка можно определить еще одну целочисленную переменную типа `xs:integer`, которая будет принимать значения 1, 2, 3, ..., отсчитывая повторения цикла. Для создания порядковой переменной надо записать ее после слова `at`:

```
for $n at $i in ("Петр", "Иванов")
```

Этот заголовок задает по два значения переменных `$n` и `$i`:

```
($n = "Петр", $i = 1)
($n = "Иванов", $i = 2)
```

Итак, заголовок запроса принимает две формы: `let` и `for`. В заголовке их можно совместить. Например:

```
for $n in ("Петр", "Иванов")
let $p := ( "****", $n, "****")
return <name>{ $p }</name>
```

В результате получаем элементы:

```
<name>*** Петр ***</name>
<name>*** Иванов ***</name>
```

Такое совмещение заголовков `let` и `for` допускается в любом порядке и в любом количестве. Можно записать несколько заголовков `for` подряд или перемежать их заголовками `let`.

Как вы знаете по предыдущей главе, циклы могут быть вложены друг в друга, при этом часто используется сокращенная запись. Такую запись можно сделать и в заголовках запроса.

Например, запрос:

```
for $i in (10, 20), $j in (1 to 3)
return <x>{ ($i, ",", $j) }</x>
```

даст в результате элементы:

```
<x>10,1</x>
<x>10,2</x>
<x>10,3</x>
<x>20,1</x>
<x>20,2</x>
<x>20,3</x>
```

Такая же запись возможна и в запросе с заголовком `let`. Попытка применить ее в предыдущем примере:

```
let $i := (10, 20), $j := (1 to 3)
return <x>{ ($i, ",", $j) }</x>
```

приведет к следующему результату:

```
<x>10 20,1 2 3</x>
```

В общем случае в запросе может быть несколько заголовков `let` и `for`, а в каждом заголовке можно записать несколько переменных. Каждый заголовок `let` и каждое выполнение заголовка `for` вместе формируют *кортеж* (tuple) — упорядоченный набор из одной или нескольких переменных. Все выполнения всех заголовков `for` приводят к образованию *потока кортежей* (tuple stream). Для каждого кортежа вычисляется выражение, расположенное в операторе `return`.

Кортеж, сформированный в заголовках запроса, может быть пропущен через фильтр, образованный выражением, записанным в операторе `where` запроса. Кортеж проходит через фильтр к оператору `return`, если выражение оператора `where` для него истинно. Например:

```
for $x in /count/incr
where $x > 0
return $x
```

Поток кортежей, прошедших через фильтр, можно отсортировать, задав порядок сортировки оператором `order by`, в котором записывается одно или несколько выражений через запятую. Каждое выражение может завершаться словом `ascending` (принимается по умолчанию) или словом `descending`, задающим порядок сортировки: по возрастанию или по убыванию значений переменных соответственно. В результате вычисления выражений определяются переменные, по которым производится сортировка. Вот простейший пример:

```
for $n in ("Петр", "Иванов")
order by $n descending
return <name>{ $p }</name>
```

Итак, запрос в полном виде состоит из пяти частей "for-let-where-order-by-return", откуда и произошло его сокращенное название FLWOR (читается так же, как слово "flower"). Полная форма выражения-запроса выглядит так:

```
for $a1 as Тип1 at $i1 in Выражение1, $a2 as Тип2 at $i2 in Выражение2,...
let $b1 as Тип3 := Выражение3, $b2 as Тип4 := Выражение4,...
where Выражение5
order by Выражение6
return Выражение7
```

Хотя бы один из заголовков `for` и `let` должен присутствовать в запросе обязательно. Заголовков может быть несколько, и они могут встречаться в запросе в любом порядке. Части `where` и `order by` необязательны, они могут встретиться только по одному разу в указанном порядке. Заключительная часть `return` обязательна, она только одна. В части `return` может встретиться еще один запрос FLWOR, а в нем другой, таким образом можно сделать вложенные запросы.

Примеры запросов

Перейдем к более полным примерам. Консорциум W3C в 2002 году издал документ "XML Query Use Cases", рабочая версия которого расположена по адресу <http://www.w3.org/TR/xquery-use-cases/>. В нем приведено множество примеров использования запросов XQuery, которые можно считать эталонным применением языка XQuery. При изучении языка очень полезно внимательно просмотреть эти примеры. Здесь приводятся только некоторые из них. Они изложены в версии документа от 2 мая 2003 года.

В феврале 2003 года консорциум W3C издал дополнение "XQuery and XPath Full-Text Use Cases", которое можно посмотреть по адресу <http://www.w3.org/TR/xmlquery-full-text-use-cases>. Во время написания данной книги в дополнении были перечислены только запросы, но не было их реализации на языке XQuery. Сейчас, когда вы читаете эту главу, дополнение, должно быть, уже обновлено и содержит тексты запросов на языке XQuery.

Примерный документ XML, записанный в файле `bib.xml`, содержит список книг. Он приведен в листинге 7.1.

Листинг 7.1. Примерный документ XML

```
<?xml version="1.0" encoding="windows-1251" ?>

<bib>
```

```
<book year="2003">

  <title>Протоколы TCP/IP. Практическое руководство</title>

  <author>
    <last>Стивенс</last>
    <first>Y.</first>
  </author>

  <publisher>Невский Диалект, БХВ-Петербург</publisher>

  <price>220.00 </price>

</book>

<book year="1992">

  <title>Advanced Programming in the Unix environment</title>

  <author>
    <last>Stevens</last>
    <first>W.</first>
  </author>

  <publisher>Addison-Wesley</publisher>

  <price>65.95</price>

</book>

<book year="2000">

  <title>Data on the Web</title>

  <author>
    <last>Abiteboul</last>
    <first>Serge</first>
  </author>

  <author>
    <last>Buneman</last>
    <first>Peter</first>
  </author>

  <author>
    <last>Suciu</last>
    <first>Dan</first>
  </author>
```

```

    <publisher>Morgan Kaufmann Publishers</publisher>

    <price>39.95</price>

</book>

<book year="1999">

    <title>The Economics of Technology</title>

    <editor>
        <last>Gerbarg</last>
        <first>Darcy</first>
        <affiliation>CITI</affiliation>
    </editor>

    <publisher>Kluwer Academic Publishers</publisher>

    <price>129.95</price>

</book>

</bib>

```

Сделаем несколько запросов к документу bib.xml. Для их обработки может быть использовано программное средство IPSI-XQ (IPSI XQuery Interpreter), разработанное в институте Fraunhofer IPSI, Германия.

1. Выделить из списка книг название и год издания книг, опубликованных Addison-Wesley после 1991 года:

```

<bib> {
  for $b in doc("bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
      { $b/title }
    </book>
}
</bib>

```

Применение этого запроса к документу листинга 7.1 приводит к такому результату:

```

<bib>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
  </book>
</bib>

```

2. Получить список названий книг и их авторов:

```
<results> {  
  for $b in doc("bib.xml")/bib/book,  
    $t in $b/title,  
    $a in $b/author  
  return  
    <result>  
      { $t }  
      { $a }  
    </result>  
}  
</results>
```

Этот запрос дает следующий результат:

```
<results>  
  <result>  
    <title>Протоколы TCP/IP. Практическое руководство</title>  
    <author>  
      <last>Стивенс</last>  
      <first>У.</first>  
    </author>  
  </result>  
  <result>  
    <title>Advanced Programming in the Unix environment</title>  
    <author>  
      <last>Stevens</last>  
      <first>W.</first>  
    </author>  
  </result>  
  <result>  
    <title>Data on the Web</title>  
    <author>  
      <last>Abiteboul</last>  
      <first>Serge</first>  
    </author>  
  </result>  
  <result>  
    <title>Data on the Web</title>  
    <author>  
      <last>Buneman</last>  
      <first>Peter</first>  
    </author>
```

```

</result>
<result>
  <title>Data on the Web</title>
  <author>
    <last>Suciu</last>
    <first>Dan</first>
  </author>
</result>
</results>

```

3. Создать список имен авторов и книг, написанных каждым автором:

```

<results> {
  let $a := doc("bib.xml")//author
  for $last in distinct-values($a/last),
    $first in distinct-values($a[last=$last]/first)
  return
    <result>
      { $last, $first }
      {
        for $b in doc("bib.xml")/bib/book
        where some $ba in $b/author
          satisfies ($ba/last = $last and $ba/first=$first)
        return $b/title
      }
    </result>
}
</results>

```

Результат запроса не очень красив:

```

<results>
  <result>СтивенсУ.
    <title>Протоколы TCP/IP. Практическое руководство</title>
  </result>
  <result>StevensW.
    <title>Advanced Programming in the Unix environment</title>
  </result>
  <result>AbiteboulSerge
    <title>Data on the Web</title>
  </result>
  <result>BunemanPeter
    <title>Data on the Web</title>
  </result>

```

```

    <result>SuciuDan
      <title>Data on the Web</title>
    </result>
  </results>

```

4. Поправим предыдущий запрос, отделив имя автора от его фамилии и записав их в отдельных элементах:

```

<results> {
  let $a := doc("bib.xml")//author
  for $last in distinct-values($a/last),
    $first in distinct-values($a[last=$last]/first)
  return
    <result>
      <last>{ $last }</last>
      <first>{ $first }</first>
      {
        for $b in doc("bib.xml")/bib/book
        where some $ba in $b/author
          satisfies ($ba/last = $last and $ba/first=$first)
        return $b/title
      }
    </result>
}
</results>

```

Результат этого запроса выглядит лучше:

```

<results>
  <result>
    <last>Стивенс</last>
    <first>У.</first>
    <title>Протоколы TCP/IP. Практическое руководство</title>
  </result>
  <result>
    <last>Stevens</last>
    <first>W.</first>
    <title>Advanced Programming in the Unix environment</title>
  </result>
  <result>
    <last>Abiteboul</last>
    <first>Serge</first>
    <title>Data on the Web</title>
  </result>

```



```

    <result>
      <last>Buneman</last>
      <first>Peter</first>
      <title>Data on the Web</title>
    </result>
    <result>
      <last>Suciu</last>
      <first>Dan</first>
      <title>Data on the Web</title>
    </result>
  </results>

```

5. Следующий пример выбирает книги, названия которых заканчиваются на "or" и узлы, содержащие слово "Suciu". Функции, использованные в этом примере, приведены в *главе 6*.

```

for $b in doc("bib.xml")//book
let $e := $b/*[contains(string(.), "Suciu") and
                  ends-with(local-name(.), "or")]
where exists($e)
return
  <book>
    { $b/title }
    { $e }
  </book>

```

В результате получаем следующие элементы:

```

<book>
  <title>Data on the Web</title>
  <author>
    <last>Suciu</last>
    <first>Dan</first>
  </author>
</book>

```

Как видно даже из этих примеров, на языке XQuery можно сделать очень сложные запросы, позволяющие выбрать из документа XML самую разнообразную информацию. Но это еще не все, что входит в язык, в него добавлен оператор варианта. Кроме того, язык позволяет определять переменные, функции, модули. Ознакомимся с этими конструкциями языка XQuery.

Упражнения

1. Выберите из листинга 7.1 названия книг и их цену.
2. Составьте список книг одного автора.

3. Выберите все книги, изданные в одном издательстве.
4. Подсчитайте количество книг в списке.
5. Подсчитайте количество книг определенного автора.

Оператор варианта

Оператор варианта, обычный во многих языках программирования, не был включен в язык XPath 2.0. Эта "оплошность" исправлена в языке XQuery. В него введен оператор варианта `typeswitch`. Правда, выбор варианта в языке XQuery своеобразен. Он основан на типе выражения, а не на его значении.

Оператор варианта вычисляет выражение, записанное в скобках, и выбирает первый вариант, тип которого совпадает с типом результата выражения. На этом работа оператора заканчивается, оставшиеся варианты не рассматриваются.

Каждый вариант записывается после слова `case`. В варианте записывается тип и, после слова `return`, выражение — результат выполнения оператора. Вариант, подходящий под любой выбор, необязателен, он записывается после слова `default`. Этот вариант имеет смысл записывать только в последнюю очередь, иначе выбор не пройдет дальше него. Например:

```
typeswitch (//address)
  case element(*, USAddress) return //address/state
  case element(*, RussiaAddress) return //address/region
  default return "Unknown address type"
```

В каждом варианте может быть определена переменная, получающая значение типа этого варианта. Ее можно использовать только в этом варианте, вне варианта она неизвестна. Вот пример из спецификации языка XQuery:

```
typeswitch ($customer/billing-address)
  case $a as element(*, USAddress) return $a/state
  case $a as element(*, CanadaAddress) return $a/province
  case $a as element(*, JapanAddress) return $a/prefecture
  default return "unknown"
```

Здесь одна и та же переменная `$a` принимает три разных значения в трех вариантах выбора. Каждое значение известно только в своем варианте.

Упражнение

Выберите из листинга 7.1 фамилии либо авторов книг, либо редакторов.

Функции пользователя

Язык XQuery позволяет использовать все функции языка XPath 2.0. Недавно документ консорциума W3C, описывающий их, называется "XQuery 1.0 and XPath 2.0 Functions and Operators". В предыдущих примерах мы видели, как работают встроенные функции.

Кроме встроенных функций, каждый разработчик может применять собственноручно написанные. Синтаксис определения функции таков:

```
define function Имя ($x1 as Тип1, $x2 as Тип2,...) as Тип3 {
    Выражение
}
```

После слов `define function` записывается имя функции, которым может служить любое уточненное имя типа `QName`. В скобках перечисляется список аргументов функции, который может быть пустым. Для каждого аргумента после слова `as` можно записать его тип. Если тип аргумента не указан, то по умолчанию понимается произвольная последовательность. После скобки, закрывающей список аргументов, и слова `as` записывается тип значения, возвращаемого функцией. Если он отсутствует, то по умолчанию понимается произвольная последовательность. Наконец, в фигурных скобках записывается тело функции, в котором, чаще всего, располагается конструктор.

При записи типа можно использовать обозначения, обычные в описании DTD документа XML. Мы уже применяли их в первых главах книги, они заключаются в следующем:

- если значения какого-то типа, записанного в заголовке функции, могут встречаться несколько раз или отсутствовать, то после имени типа записывается звездочка `*`;
- если хотя бы одно значение обязательно должно присутствовать, то записывается плюс `+`;
- если значение может встретиться нуль или один раз, то записывается вопросительный знак `?`.

Для вызова функции достаточно написать ее имя и в скобках перечислить аргументы функции.

Например, функция `section-summary` создает список разделов книги — элементов `section`, — содержащий названия разделов и количество иллюстраций в них:

```
define function section-summary($bs as element()?)
    as element()*
{
    for $section in $bs/section
```

```

return
  <section>
    { $section/@* }
    { $section/title }
    <figcount>
    { count($section/figure) }
    </figcount>
    { section-summary($section) }
  </section>
}

```

Затем функция вызывается в конструкторе элемента:

```

<toc>
  {
    for $s in doc("book.xml")/book/section
    return section-summary($s)
  }
</toc>

```

Язык XQuery допускает рекурсивные функции, вызывающие сами себя, и взаимно рекурсивные функции, вызывающие друг друга. Классический пример — функция `depth()`, подсчитывающая глубину вложенности элементов в данный элемент — аргумент функции:

```

define function depth($e as element())
  as xs:integer
{
  if (empty($e/*)) then 1
  else 1 + max(for $c in $e/* return depth($c))
}

```

Функция используется примерно так:

```
depth(doc("bib.xml")/bib)
```

Где же записывается определение функции? Для записи всех определений перед выражением-запросом записывается пролог.

Упражнение

Оформите в виде функций все предыдущие упражнения данной главы.

Пролог

Перед выражением-запросом можно написать необязательный *пролог* (prolog), содержащий глобальные определения. Пролог вместе с выражением-запросом образуют так называемый *главный модуль* (main module). В на-

чале пролога можно записать номер версии языка XQuery в следующем виде:

```
xquery version "1.0"
```

Пока у языка XQuery есть только одна версия 1.0, эта запись не имеет никакого значения. В дальнейшем, когда понадобится различать версии языка, она будет полезна.

Далее в прологе в любом порядке и любом количестве можно записать необязательные определения пространств имен, переменных, импортировать схему документа и другие модули.

Определения функций, если они есть, записываются в конце пролога.

Определение пространств имен

Для определения пространства имен и его префикса достаточно записать их в прологе строкой вида:

```
declare namespace xyz = "http://some.domain/myns"
```

Таких строк в прологе может быть сколько угодно.

В языке XQuery определены префиксы пяти пространств имен, которые можно использовать по умолчанию:

- `xml` = "http://www.w3.org/XML/1998/namespace" — пространство имен XML (см. главу 1);
- `xs` = "http://www.w3.org/2001/XMLSchema" — пространство имен языка XSD (см. главу 3);
- `xsi` = "http://www.w3.org/2001/XMLSchema-instance" — пространство экземпляров схем XML (см. главу 1);
- `fn` = "http://www.w3.org/2003/05/xpath-functions" — пространство имен встроенных функций языка XPath 2.0 (см. главу 6);
- `xdt` = "http://www.w3.org/2003/05/xpath-datatypes" — пространство имен типов данных языка XPath 2.0 (см. главу 6).

Кроме того, в прологе можно определить два пространства имен по умолчанию для записи имен без префикса: одно для имен элементов, другое для имен функций. Их определения выглядят примерно так:

```
declare element namespace = "http://some.domain/mynames"  
declare function namespace = "http://some.domain/myfuncs"
```

Напомню, что в прямом конструкторе элемента, в атрибуте `xmlns`, тоже можно определить пространство имен по умолчанию. Это определение будет действовать только в данном конструкторе, вне его действует определение, сделанное в прологе.

Импорт схемы

Язык XQuery позволяет импортировать объявления элементов, атрибутов и пространств имен, сделанные в некоторой схеме. При этом сначала записывается ее целевое пространство имен, а затем через предлог `at` — адрес URI или файл, в котором записана схема. Например:

```
import schema "http://some.domain/mynames"
  at "http://some.domain/names.xsd"
```

Оператор `import` позволяет сразу же назначить префикс целевому пространству имен импортируемой схемы:

```
import schema namespace myns = "http://some.domain/mynames"
  at "http://some.domain/names.xsd"
```

или сделать это пространство имен пространством имен по умолчанию:

```
import schema default element
  namespace = "http://some.domain/mynames"
  at "http://some.domain/names.xsd"
```

Местоположение схемы, идущее за предлогом `at`, может отсутствовать, если есть какой-то другой способ указать адрес схемы.

Определение переменных

В прологе можно дать определение глобальных переменных, используемых потом в выражениях-запросах. При определении указывается имя переменной и ее значение, задаваемое произвольным выражением. Кроме того, после предлога `as` можно указать тип переменной, если он отличается от типа выражения. При этом типы переменной и выражения должны быть совместимы. Например:

```
define variable $x (-3.5)
define variable $n as xs:integer (-3)
```

Импорт модуля

Кроме главного модуля, в языке XQuery существуют *библиотечные модули* (library module), состоящие только из пролога. Они отмечаются словом `module`, за которым записывается целевое пространство имен модуля и его пролог. Целевое пространство имен модуля будет пространством имен по умолчанию для имен переменных и функций, определенных в модуле. Например:

```

module "http://some.domain/module1"
import schema namespace ipo = "http://some.domain/IPO"
import schema namespace zips = "http://some.domain/zips"

define function zip-ok($a as element(*, ipo:USAddress))
  as xs:boolean {
    some $i in doc("zips.xml")/zips:zips/element(zips:row)
      satisfies $i/zips:city = $a/city
        and $i/zips:state = $a/state
        and $i/zips:zip = $a/zip
  }

```

Определения, сделанные в библиотечном модуле, можно импортировать в другой модуль, записав в этом, другом, модуле оператор импорта, имеющий вид:

```

import module "http://some.domain/module1"
  at "http://some.domain/module1.xqr"

```

В операторе импорта указывается целевое пространство имен библиотечного модуля. После предлога `at` строкой URI указывается местоположение модуля. Эту часть оператора можно опустить, если есть какой-то другой способ указать расположение модуля.

Целевое пространство имен модуля можно сразу снабдить префиксом:

```

import module namespace xyz = "http://some.domain/module1"
  at "http://some.domain/module1.xqr"

```

Пример главного модуля

Итак, в общем виде запись на языке XQuery, образующая главный модуль, состоит из пролога и выражения, которым чаще всего служит выражение-запрос. Вот пример главного модуля, взятый из документации XQuery. В нем определена функция `names-match()`, проверяющая совпадение имен, заданных ее аргументами. Эта функция применяется в запросе `for` для отбора узлов с несовпадающими именами:

```

import schema namespace ipo="http://www.example.com/IPO"

define function names-match(
  $s as element(ipo:purchaseOrder/shipTo),
  $b as element(ipo:purchaseOrder/billTo)
)
  as xs:boolean

```

```
{
    $s/name = $b/name
}

for $p in doc("ipo.xml")//element(ipo:purchaseOrder)
where not(names-match($p/shipTo, $p/billTo))
return $p
```

Реализации XQuery

Хотя первая версия языка XQuery еще находится в черновом состоянии, потребность в нем такова, что многие фирмы уже выпустили коммерческие или свободно распространяемые программные продукты, реализующие и XQuery и XPath 2.0. Вот далеко не полный список таких продуктов.

- ❑ Фирма BEA предоставляет реализацию языка XQuery под названием Liquid Data в составе своего сервера приложений WebLogic: <http://edocs.bea.com/liquiddata/docs10/prodover/concepts.html>.
- ❑ Корпорация Bluestream Database Software Corp. выпускает сервер баз данных XStreamDB 3.0, содержащий обработчик языка XQuery: <http://www.bluestream.com/dr/?page=Home/Products/XStreamDB/>.
- ❑ Корпорация Cerisent Corp. выпускает обработчик запросов XQE (XML Query Engine): <http://www.cerisent.com/cerisent-xqe.html>.
- ❑ Компания Cognetic Systems разработала реализацию языка XQuery под именем XQuantum: <http://www.cogneticsystems.com/xquery/xquery.html>.
- ❑ Компания Enosys Software распространяет основанный на XQuery сервер ЕИ (Enterprise Information Integration): <http://www.enosyssoftware.com/>.
- ❑ Подразделение Sonic Software корпорации Progress Software Corp. выпускает сервер XIS 3.1 (eXtensible Information Server): <http://www.sonicsoftware.com/>.
- ❑ Канадская фирма Fatdog Software свободно распространяет под лицензией GNU обработчик XQEngine: <http://www.fatdog.com/>.
- ❑ Французская фирма GAEL Consultant выпускает аналитическую систему Derby, которая делает запросы на языке XQuery: <http://www.gael.fr/derby/>.
- ❑ В состав популярного инструментального средства Кава входит частичная реализация языка XQuery под названием Qexo. Она свободно распространяется под лицензией GNU: <http://www.gnu.org/software/qexo/>.
- ❑ Компания Ipedo выпускает сервер баз данных XML Database v3.0 с доступом к данным на языке XQuery: <http://www.ipedo.com/>.
- ❑ В институте Fraunhofer IPSI, Германия, выпускается постоянно обновляемая реализация XQuery под названием IPSI-XQ (IPSI XQuery Inter-

preter): http://ipsi.fhg.de/oasys/projects/ipsi-xq/index_e.html. Она свободно распространяется под собственной лицензией. Все примеры настоящей главы проверены этим средством.

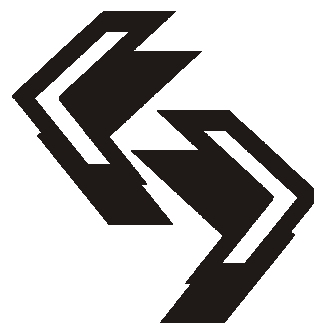
- ❑ Подразделение лабораторий Белла Lucent-Bell Labs начало разработку Galax — реализации языка XQuery: <http://db.bell-labs.com/galax/>.
- ❑ Компания Nimble Technology поставляет обработчик XQuery в составе своего основного продукта Nimble Integration Suite: <http://www.nimble.com/>.
- ❑ Компания OpenLink Software вставила запросы XQuery в свой сервер Virtuoso Universal Server: <http://demo.openlinksw.com:8890/xqdemo>.
- ❑ В сервер Oracle XML DB вставлена возможность запросов на языке XQuery: http://otn.oracle.com/tech/xml/xmldb/htdocs/querying_xml.html.
- ❑ Фирма QuiLogic поставляет компонент SQL/XML-IMDB (In Memory Database) с доступом к данным на языках SQL и XQuery: <http://www.quilogic.cc/xml.htm>.
- ❑ Известная фирма Software AG выпускает и свободно распространяет инструментальное средство QuiP для создания и выполнения запросов на языке XQuery: <http://www.softwareag.com/developer/downloads/default.htm>.
- ❑ Та же фирма Software AG включает в состав своего сервера Tamino XML Server анализатор и процессор языка XQuery: http://www.softwareag.com/tamino/News/tamino_41.htm.
- ❑ На сайте SourceForge.net реализуется на Java общественный проект XQuench: <http://xquench.sourceforge.net/>.
- ❑ Фирма X-Hive реализует запросы XQuery к своему серверу баз данных X-Hive/DB: <http://www.x-hive.com/xquery>.
- ❑ Компания XML Global выпускает набор инструментальных средств XML Integration Workbench, в который входит процессор XQuery: <http://www.xmlglobal.com/prod/xmlworkbench/>.

Вопросы для самопроверки

1. Что такое конструктор узла?
2. Какие виды узлов можно создать конструктором?
3. Для каких видов узлов применяются прямые конструкторы?
4. Какие виды узлов создают вычисляемые конструкторы?
5. Какие выражения можно записывать в конструкторах?
6. Можно ли задать имя конструируемого узла выражением?
7. Могут ли быть атрибуты у элементов, вложенных в конструктор?
8. Каков минимальный состав выражения-запроса?

9. Сколько заголовков `for` и `let` может быть в запросе?
10. В каком порядке в запросе должны располагаться заголовки `for` и `let`?
11. Сколько раз в запросе можно записать оператор `where`?
12. В каком порядке можно задать сортировку оператором `order by`?
13. Какое выражение можно записать в операторе `return`?
14. Можно ли на языке XQuery организовать вложенные запросы?
15. В каком месте модуля записываются определения функций пользователя?
16. Что можно записать в прологе запроса?
17. По каким правилам записывается библиотечный модуль?
18. Как можно использовать библиотечный модуль?

ЧАСТЬ II



Обработка документов XML

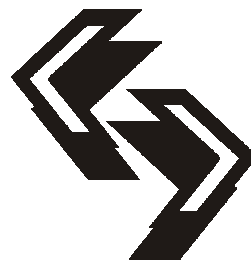
Глава 8. Преобразование документов средствами XSLT

Глава 9. Форматирование объектов XSL-FO

Глава 10. Обработка документов XML при помощи событий

Глава 11. Обработка документов при помощи DOM

ГЛАВА 8



Преобразование документов средствами XSLT

В языке HTML часто применяются таблицы стилей CSS (Cascading Style Sheet), задающие общие правила оформления документов HTML: цвет текста и фона, размеры шрифта для разных частей документа, размер, цвет и расположение заголовков. Выполнение этих правил придает документам единый стиль оформления.

Правила оформления CSS устанавливают стиль для каждого тега в отдельности. Стиль записывается весьма непривычно. Он начинается с *селектора* (selector), чаще всего это просто имя тега или несколько имен через запятую. После селектора через пробел в фигурных скобках записываются *объявления* (declarations). Каждое объявление состоит из *свойства* (property) и его *значения* (value), записанного через двоеточие. Одно объявление отделяется от другого точкой с запятой.

Например, большинство браузеров разделяет абзацы пустой строкой, не оставляя красной строки. Если вы хотите оформить абзацы в классическом стиле с отступом в первой строке (свойство text-indent), не увеличивая расстояние (margin) перед (margin-top) и после (margin-bottom) абзаца, то можете записать стиль тега <p> следующим образом:

```
p { text-indent: 2em; margin-top: 0; margin-bottom: 0; }
```

Если вы хотите выделить какой-то текст красным полужирным шрифтом, то вам достаточно заключить его в тег , установив для этого тега следующий стиль:

```
strong { color: red; font-weight: bold; }
```

Если вы хотите один и тот же тег оформить разными стилями, чтобы применять их в разных ситуациях, то после имени тега через точку укажите класс стиля. Класс стиля — это просто какое-то имя. Например, вы задали общий стиль заголовка <h2> так, чтобы заголовок был синим:

```
h2 { margin-top: 8em; margin-bottom: 3em; color: blue }
```

но хотите, чтобы заголовок подраздела был черным. Вы пишете еще одну строку, определяя класс `subsection`:

```
h2.subsection { margin-top: 8em; margin-bottom: 3em; color: black }
```

Теперь, чтобы заголовок был черным, надо писать тег `<h2>` с указанием класса:

```
<h2 class="subsection">Заголовок подраздела</h2>
```

Список всех стилей (stylesheet) можно записать в отдельном файле, например, `contract.css`, как показано в листинге 8.1.

Листинг 8.1. Файл стилей `contract.css`

```
body {
    margin-left: 10%;
    margin-right: 10%;
    font-family: sans-serif
}
p { text-indent: 2em; margin-top: 0; margin-bottom: 0; }
p.remark {
    padding-left: 0.2em;
    border-left: solid;
    border-right: none;
    border-top: none;
    border-bottom: none;
    border-left-width: thin;
    border-color: red;
}
strong { color: red; font-weight: bold; }
h2 { margin-top: 8em; margin-bottom: 3em; color: blue }
h2.subsection { margin-top: 8em; margin-bottom: 3em; color: black }
```

Затем этот файл следует подключать к каждому документу HTML, который вы хотите оформить в данном стиле, с помощью тега `<link>`, как показано в листинге 8.2.

Листинг 8.2. Документ HTML со стиливым оформлением

```
<html><head>

<meta content="text/html; charset=windows-1251"
      http-equiv="Content-Type">
<title>Документ HTML со стилями</title>

<link type="text/css" rel="stylesheet" href="contract.css">
```

```

</head><body>

<h2>Обычный заголовок</h2>

<p>Это абзац обычного для данного документа стиля с
<strong>выделенным текстом</strong>.</p>

<h2 class="subsection">Заголовок подраздела</h2>

<p class="remark">Это замечание, оформленное как особый абзац.</p>

</body></html>

```

Второй способ задать стиль тегов документа — записать стили прямо в документе HTML, в теге `<style>`, как показано в листинге 8.3.

Листинг 8.3. Документ HTML с перечислением стилей тегов

```

<html><head>

<meta content="text/html; charset=windows-1251"
      http-equiv="Content-Type">

<style type="text/css">

  body {
    margin-left: 10%;
    margin-right: 10%;
    font-family: sans-serif
  }
  p { text-indent: 2em; margin-top: 0; margin-bottom: 0; }
  p.remark {
    padding-left: 0.2em;
    border-left: solid;
    border-right: none;
    border-top: none;
    border-bottom: none;
    border-left-width: thin;
    border-color: red;
  }
  strong { color: red; font-weight: bold; }
  h2 { margin-top: 8em; margin-bottom: 3em; color: blue }
  h2.subsection { margin-top: 8em; margin-bottom: 3em; color: black }

</style>

</head><body>

```

```
<h2>Обычный заголовок</h2>

<p>Это абзац обычного для данного документа стиля с
<strong>выделенным текстом</strong>.</p>

<h2 class="subsection">Заголовок подраздела</h2>

<p class="remark">Это замечание, оформленное как особый абзац.</p>

</body></html>
```

Третий способ задать стиль — указать его свойства прямо в теге:

```
<strong style = "color:red; font-weight:bold">
```

Все эти способы можно комбинировать, при этом стиль, указанный непосредственно в теге, оказывается предпочтительнее стиля, указанного в теге `<style>`, а тот предпочтительнее стиля, записанного в файле стилей.

Таблицы стилей CSS в языке XML

Идея оформления документов в едином стиле очень привлекательна и не могла не быть реализована в XML. Вместе с документами XML тоже можно применять таблицы стилей CSS. Для этого в язык XML введена инструкция по обработке `stylesheet`, которая используется так, как показано в листинге 8.4.

Листинг 8.4. Документ XML, использующий таблицу стилей CSS

```
<?xml version="1.0" encoding="windows-1251"?>

<?xml:stylesheet type="text/css" href="xmlcontract.css"?>

<contract>

  <type>Трудовой договор</type>

  <name>Иванов Петр Сидорович</name>

  <date>01.07.03</date>

  <period>12</period>

  <!-- И так далее... -->

</contract>
```

Всякий браузер, "понимающий" XML, например Mozilla или Internet Explorer 6.x, покажет приведенный в листинге 8.4 документ в соответствии со стилями, записанными в файле `xmlcontract.css`.

Таблица стилей, включенная в файл `xmlcontract.css`, оформляется по правилам CSS и может выглядеть так, как показано в листинге 8.5.

Листинг 8.5. Таблица стилей для документа XML

```
type { margin-top: 8em; margin-bottom: 3em;
      font-size: 2em; color: blue }

name { text-indent: 2em; margin-top: 3em;
      font-size: 1.5em; margin-bottom: 2em; }

date, period { margin: 0.5em; }
```

Хотя таблицы стилей CSS и можно использовать в XML, но реализация стилей для документов XML должна быть другой. Как видно из приведенных примеров, синтаксис CSS весьма оригинален и никак не похож на синтаксис XML. Кроме того, стили CSS определяют способы показа документа HTML в окне браузера, его визуализацию, а язык XML выявляет структуру документа, ничего не говоря о его представлении в виде, удобном для чтения. Поэтому в технологии XML для записи стилей был разработан специальный язык XSL — одна из реализаций XML.

Язык описания стилей XSL

Таблицы стилей для документов XML записываются при помощи специально сделанной реализации языка XML, названной XSL (XML Stylesheet Language). В то время, когда писалась эта книга, действовала первая версия XSL, изложенная в рекомендации "Extensible Stylesheet Language (XSL). Version 1.0". Она опубликована на Web-странице <http://www.w3.org/TR/xsl/>.

Язык XSL, как и язык XPath, представляет документ в виде дерева. Процессор языка XSL преобразует это дерево, руководствуясь таблицей стилей, и форматирует его для вывода в окно браузера, на принтер, экран проектора или на какое-то другое устройство. Таким образом, обработка проходит два этапа: преобразование дерева документа (XML transform) и форматирование (formatting) дерева, полученного после преобразования.

Первый этап — этап преобразования — может быть достаточно сложным и кардинально поменять структуру дерева: изменить уровни вложенности, удалить или добавить новые узлы, создать оглавление, предметный указатель, индекс. Результат преобразования может стать новым самостоятельным документом или даже несколькими документами. Таблица стилей, по которой идет преобразование, содержит правила, состоящие из двух частей: образцов (patterns) для отбора узлов, предназначенных для преобразования, и шаблонов (templates) или конструкторов (sequence constructors) для построения преобразованных узлов.

Второй этап — этап форматирования — абстрагируется от конечного устройства, хотя может выполняться непосредственно в нем, например, в браузере, пейджере, принтере, проекторе. Форматирование формулируется в терминах классов и объектов. Под объектами форматирования FO (formatting objects) понимаются узлы дерева, а под их классами — некие конечные структуры: страницы, абзацы, таблицы, списки. Таблица стилей определяет правила форматирования (formatting properties). По этим правилам строится дерево, но уже не дерево узлов, а дерево геометрических областей (area tree), на которые разбивается документ, и определяются их характеристики: размеры, цвет, шрифт.

Вскоре после выхода рекомендации языка XSL стало ясно, что преобразование документа XML — это самостоятельная и независимая задача, которую можно выполнять не только для приведения их к одному стилю, но и для многих других целей. Например, можно преобразовать документ XML в документ HTML, XHTML или даже в документ PDF. Можно обновить документ, изменив некоторые узлы, или разделить его на несколько документов.

Поэтому преобразование документов XML было выделено в отдельную область исследования и описано отдельным языком XSLT (XSL Transformation), первая версия которого изложена в рекомендации "XSL Transformation (XSLT). Version 1.0". Эта версия рекомендации, действующая на время написания данной книги, расположена по адресу <http://www.w3.org/TR/xslt>. Ее русский перевод, сделанный Радиком Усмановым, можно посмотреть в нескольких местах Рунета, например, по адресу <http://www.online.ru/it/helpdesk/xslt01.htm>.

После выделения преобразований в отдельный язык XSLT первоначальная рекомендация языка XSL была переработана, сейчас основной акцент в ней сделан на форматировании, поэтому ее часто называют рекомендацией "XSL-FO". Мы рассмотрим форматирование в следующей главе, а в этой займемся преобразованием документов XML с помощью языка XSLT.

Язык записи преобразований XSLT

Язык преобразований XSLT — это одна из реализаций XML. По традиции, документ, записанный на языке XSLT, называется *таблицей стилей* (stylesheet), хотя его правильнее назвать документом, содержащим правила преобразований.

Все элементы XML, объявленные в языке XSLT, относятся к пространству имен <http://www.w3.org/1999/XSL/Transform>. Обычно они записываются с префиксом `xsl`. Если принят этот префикс, то корневой элемент документа XSLT — таблицы стилей — будет называться `xsl:stylesheet`. Как синоним

этого имени можно записывать корневой элемент таблицы стилей с именем `xsl:transform`.

У корневого элемента `xsl:stylesheet` есть один обязательный атрибут `version`, указывающий версию языка. Во время написания книги его значением был номер "1.0". Для процессоров XSLT 2.0 значение атрибута `version="1.0"` будет означать, что надо обеспечить обратную совместимость с версией XSLT 1.0.

Простейшая таблица стилей XSLT выглядит так, как записано в листинге 8.6.

Листинг 8.6. Простейшая таблица стилей XSLT

```
<?xml version="1.0" encoding="windows-1251" ?>

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

  <xsl:output method="text" encoding="CP866" />

</xsl:stylesheet>
```

Здесь только определяется префикс `xsl` пространства имен `http://www.w3.org/1999/XSL/Transform` и задается один стиль — шаблон `xsl:output` для вывода на стандартное устройство, а именно выводится "плоский" текст, на что показывает значение "text" атрибута `method` (другие значения — "html", "xhtml" и "xml"). Вывод текста будет сделан в кодировке CP866, указываемой атрибутом `encoding`. Такая кодировка выбрана для вывода кириллицы на консоль MS Windows, для других устройств можно выбрать кодировку UTF-8, CP1251 или KOI8-R.

Эту таблицу стилей записываем в файл, например, `simple.xml`. Ссылку на таблицу стилей можно поместить в документ XML как одну из инструкций по обработке, а именно инструкцию `xml-stylesheet`. Пример такой ссылки приведен в листинге 8.7.

Листинг 8.7. Документ XML со ссылкой на таблицу стилей

```
<?xml version="1.0" encoding="windows-1251" ?>

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

<notebook>

  <person>

    <name first="Иван" second="Петрович" surname="Сидоров" />
```

```

<address>
  <street>Садовая, 12 – 34</street>
  <city>Новокозловск</city>
  <zip>123321</zip>
</address>

<phone-list>
  <work-phone>123456</work-phone>
  <home-phone>654321</home-phone>
</phone-list>

</person>

  <person>

<name first="Мария" second="Ивановна" surname="Федорова" />

<address>
  <street>Нижняя, 12</street>
  <city>Зареченск</city>
  <zip>321123</zip>
</address>

<phone-list>
  <home-phone>224321</home-phone>
</phone-list>

</person>

</notebook>

```

"Увидев" инструкцию по обработке `xml-stylesheet`, программа-обработчик XML, если она, кроме того, является процессором XSLT, выполнит преобразование, заданное в файле `simple.xml`. Преобразование заключается в выводе содержимого элементов листинга 8.7 на стандартное устройство вывода, например, на консоль, без всякого форматирования. Будет сделано только преобразование текста в кодировку CP866 и выведен слитный текст даже без пробелов между содержимым элементов.

В листинге 8.8 показан результат преобразования листинга 8.7, сделанного XSLT-процессором Xselerator фирмы MarrowSoft Ltd, при помощи таблицы стилей листинга 8.6. Переход на вторую строку сделан только в листинге, процессор все выводит на одну строку.

Листинг 8.8. Результат преобразования с таблицей стилей `simple.xml`

```

Садовая, 12 – 34Новокозловск123321123456654321Нижняя, 12Зареченск
321123224321

```

Некоторые процессоры XSLT вставляют пробел между содержимым элементов или делают перевод на следующую строку после вывода содержимого каждого элемента. Более "умные" процессоры XSLT могут пойти дальше и сделать отступы, показывающие уровень вложенности элементов. Для того чтобы заставить и прочие процессоры делать отступы, нужно в элементе `xsl:output` записать атрибут `indent`:

```
<xsl:output method="text" encoding="CP866" indent="yes" />
```

Несложное форматирование вывода

Приведем пример чуть более развитой таблицы стилей. "Облагородим" вывод на консоль документа XML листинга 8.5, записав пробелы, перевод строки и поясняющие надписи. Все эти дополнительные символы, даже пробелы, записываются просто в содержимом элементов XSLT или в содержимом специального элемента `xsl:text`, причем символ перевода строки записывается в шестнадцатеричной форме строкой "
" или просто "отбивается" клавишей <Enter>.

Кроме того, мы выведем не только содержимое элементов документа XML, но и значения их атрибутов. Таблица стилей XSLT для такого преобразования приведена в листинге 8.9.

Листинг 8.9. Таблица стилей XSLT для показа адресной книжки

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

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

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="person">

    <xsl:apply-templates />
    <xsl:text>&#xA;&#xA;</xsl:text>

  </xsl:template>

  <xsl:template match="name">

    ФИО:

    <xsl:value-of select="@first" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
    <xsl:value-of select="@surname" /> <xsl:text>&#xA;</xsl:text>

  </xsl:template>
```

```

<xsl:template match="address">

  <xsl:text>Адрес: </xsl:text>

  <xsl:value-of select="street" /> <xsl:text>, </xsl:text>
  <xsl:value-of select="city" />  <xsl:text>, </xsl:text>
  <xsl:value-of select="zip" /> <xsl:text>&#xA;</xsl:text>

</xsl:template>

<xsl:template match="phone-list">

  <xsl:text>Список телефонов:&#xA;</xsl:text>

  <xsl:text>Рабочие: </xsl:text>

  <xsl:value-of select="work-phone" /> <xsl:text>, </xsl:text>

  <xsl:text>&#xA;Домашние: </xsl:text>

  <xsl:value-of select="home-phone" /> <xsl:text>, </xsl:text>

</xsl:template>

</xsl:stylesheet>

```

Как видите, основной элемент в таблице стилей — это элемент `xsl:template`, задающий шаблон отбора элементов для преобразования. В него вложены правила преобразования — элементы `xsl:text`, в которых записан выводимый поясняющий текст, и элементы `xsl:value-of`, указывающие, что надо преобразовать содержимое элемента или значение атрибута, имя которого записано в атрибуте `select`.

Обработка исходного дерева документа `notebook` начинается с выполнения правила:

```

<xsl:template match="person">

  <xsl:apply-templates />
  <xsl:text>&#xA;&#xA;</xsl:text>

</xsl:template>,

```

которое предписывает просмотреть все узлы-элементы `person` и применить к ним правила, записанные в его теле:

```

<xsl:apply-templates />
<xsl:text>&#xA;&#xA;</xsl:text>

```

Правило `<xsl:apply-templates />` заключается в рекурсивном просмотре всех узлов-потомков узла `person` и применении к ним всех подходящих пра-

вил, записанных в таблице стилей. Эти правила записаны в листинге 8.7 в элементах `xsl:template` отдельно для узлов `name`, `address`, `phone-list`.

Правило `<xsl:text>

</xsl:text>` заставляет вставить пустую строку после вывода преобразованного элемента `person`.

Вывод в свое окно результата преобразования листинга 8.7, сделанного XSLT-процессором Xselerator с помощью таблицы стилей листинга 8.9, показан в листинге 8.10.

Листинг 8.10. Результат преобразования листинга 8.7

ФИО: Иван Петрович Сидоров
Адрес: Садовая, 12 — 34, Новокословск, 123321
Список телефонов:
Рабочие: 123456,
Домашние: 654321,

ФИО: Мария Ивановна Федорова
Адрес: Нижняя, 12, Зареченск, 321123
Список телефонов:
Рабочие:,
Домашние: 224321,

Как видите, мы еще не добились хорошего вывода: в списке телефонов появляются лишние запятые, слово "Рабочие" выводится, даже если в списке номеров телефонов нет ни одного рабочего телефона. Для окончательного оформления нам понадобится проверка наличия или отсутствия тех или иных узлов. Средства такой проверки в XSLT есть, мы их рассмотрим позднее.

Некоторые процессоры XSLT не требуют включения имени файла с таблицей стилей в документ XML. Для них файл с таблицей стилей указывается как входной параметр вместе с документом XML. Например, популярным XSLT-обработчиком Xalan C++, продуктом сообщества Apache Software Foundation, <http://xml.apache.org/xalan-c/>, можно воспользоваться из командной строки так:

```
$ Xalan -o notebook.out notebook.xml simple.xml
```

На вход процессора Xalan подаются файлы `notebook.xml` и `simple.xml`. Результат преобразования записывается в файл `notebook.out`, что помечено флагом `-o`.

Это удобно, если надо один и тот же документ XML представить разными стилями. В таком случае мы просто запускаем процессор XSLT несколько раз с различными таблицами стилей и получаем разные преобразованные документы в выходных файлах.

Включение таблицы стилей в документ XML

С другой стороны, таблицу стилей можно записывать не в отдельный файл, а непосредственно в преобразуемый документ XML. Для этого в элементе `xsl:stylesheet` предусмотрен атрибут-идентификатор `id`, на который можно ссылаться обычным образом из инструкции по обработке `xml-stylesheet`, как сделано в листинге 8.11.

Листинг 8.11. Таблица стилей внутри документа XML

```
<?xml version="1.0" encoding="windows-1251" ?>

<?xml-stylesheet type="text/xsl" href="#simple" ?>

<notebook>

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

    <xsl:output method="text" encoding="CP866" />

  </xsl:stylesheet>

  <person>

    <name first="Иван" second="Петрович" surname="Сидоров" />

    <address>
      <street>Садовая, 12 - 34</street>
      <city>Новокозловск</city>
      <zip>123321</zip>
    </address>

    <phone-list>
      <work-phone>123456</work-phone>
      <home-phone>654321</home-phone>
    </phone-list>

  </person>

  <person>

    <name first="Мария" second="Ивановна" surname="Федорова" />

    <address>
      <street>Нижняя, 12</street>
      <city>Зареченск</city>
      <zip>321123</zip>
    </address>
```



```
<phone-list>
  <home-phone>224321</home-phone>
</phone-list>

</person>

</notebook>
```

Преобразование документа XML в документ HTML

Таблицы стилей, записанные в листингах 8.6 и 8.9, предназначены для преобразования документа XML в простой "плоский" текст для вывода на консоль или в текстовый файл. Приведем пример преобразования документа XML листинга 8.7 в документ HTML. Соответствующая таблица стилей записана в листинге 8.12. Обратите внимание на то, что в элементе `xsl:output` указан метод вывода "html".

Листинг 8.12. Таблица стилей для преобразования XML в HTML

```
<?xml version="1.0" encoding="windows-1251"?>

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

  <xsl:output method="html" encoding="windows-1251"/>

  <xsl:template match="/">

    <html><head><title>Адресная книжка</title></head>

    <body><h2>Фамилии, адреса и телефоны</h2>

    <xsl:apply-templates />

  </body></html>

</xsl:template>

<xsl:template match="name">

  <p />
  <xsl:value-of select="@first" /> <xsl:text> </xsl:text>
  <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
  <xsl:value-of select="@surname" /> <br />

</xsl:template>
```

```

<xsl:template match="address">

    <br />
    <xsl:value-of select="street" /> <xsl:text> </xsl:text>
    <xsl:value-of select="city" /> <xsl:text> </xsl:text>
    <xsl:value-of select="zip" /> <br />

</xsl:template>

<xsl:template match="phone-list">

    Рабочий: <xsl:value-of select="work-phone" /> <br />
    Домашний: <xsl:value-of select="home-phone" /> <br />

</xsl:template>

</xsl:stylesheet>

```

Из листинга 8.12 хорошо видно, что теги языка HTML записываются в таблице стилей XSL как простой текст. Процессор XSLT не обрабатывает их, поскольку они объявлены в другом пространстве имен. Теги HTML перейдут в преобразованный документ без изменений, но будут записаны по правилам языка HTML, т. к. у элемента `xsl:output` указан атрибут `method="html"`. Например, тег `
` будет записан как `
`.

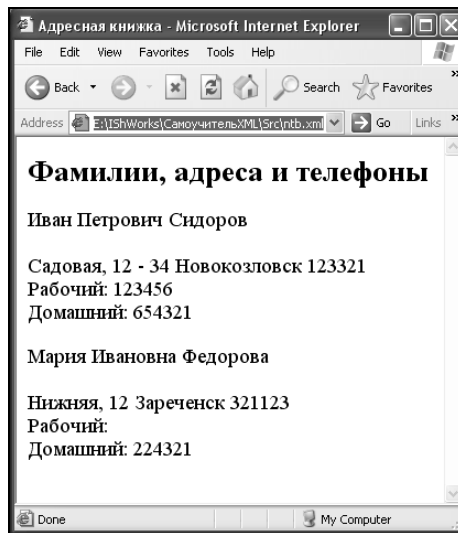


Рис. 8.1. Вывод преобразования в окно браузера

Эту таблицу стилей можно записать в файл, например `ntb.xsl`, и сослаться на него в листинге 8.7, описывающем адресную книжку, начав листинг следующими строками:

```
<?xml version="1.0" encoding="windows-1251"?>

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

<notebook>

    <!-- Содержимое адресной книжки -->

</notebook>
```

После этого любой браузер, "понимающий" XML и XSLT, например Mozilla или Internet Explorer 6.x, покажет файл notebook.xml, содержащий листинг 8.7, как предписано таблицей стилей листинга 8.12. На рис. 8.1 показано окно браузера с результатом преобразования.

Ресурсы языка XSLT

Мы не будем в этой книге подробно разбирать все конструкции языка XSLT — одно описание языка будет толще всей этой книги. К счастью, языку XSLT посвящено уже много хороших книг.

На русский язык переведена "библия" XSLT [4]. Ее автор, Майкл Кэй (Michael H. Kay), создал и свободно распространяет популярный XSLT-процессор Saxon. Домашняя Web-страница Майкла Кэя находится по адресу <http://homepage.ntlworld.com/michael.h.kay/>.

Очень подробное и понятное изложение первой версии языка XSLT сделал Алексей Валиков [2], его книгой вы можете воспользоваться для дальнейшего изучения XSLT 1.0. В книге А. Н. Валикова вы найдете описание XSLT-процессоров и их сравнение.

Еще одна переводная книга С. Холзнера [8] содержит полное и точное описание языка XSLT 1.0.

Все опубликованные до сих пор в России книги описывают язык XSLT 1.0. Мы опишем следующую, вторую версию XSLT 2.0. Она появилась в 2002 году и на момент написания данной книги была в черновом варианте. Последний вариант описания версии XSLT 2.0, который я видел, находился по адресу <http://www.w3.org/TR/2003/WD-xslt20-20030502/>. Кстати, этот документ редактируется Майклом Кэем, уже выпустившим версию Saxon, обрабатывающую таблицы стилей XSLT 2.0. На момент написания книги это была версия Saxon 7.6. Я использовал ее для проверки всех примеров этой главы. Новейшую версию XSLT-процессора Saxon всегда можно загрузить со странички <http://saxon.sourceforge.net/>.

Главная особенность версии XSLT 2.0 заключается в том, что она опирается на язык XPath версии 2.0. Переход в языке XPath от множества узлов к последовательностям узлов и атомарных значений привел ко многим мелким изменениям в языке XSLT 2.0. Мы будем отмечать эти изменения по ходу

изложения. Из других новшеств следует отметить то, что в результате обработки могут получиться сразу несколько документов, допускаются функции, определенные пользователем, и значительно облегчена группировка значений.

Перейдем к описанию языка XSLT 2.0.

Образцы (patterns)

Прежде чем сделать преобразование дерева документа XML, из него следует выбрать те узлы, которые подвергнутся тому или иному преобразованию. Их можно выбирать по имени, содержимому, атрибутам и другим признакам. Условия отбора узлов задаются *образцом* (pattern), записанным в виде одного или нескольких выражений языка XPath 2.0, рассмотренного в *главе 6*. Выражения, содержащиеся в образце, объединяются вертикальной чертой |, означающей, что выбираемый узел должен удовлетворять хотя бы одному выражению образца. Вместо вертикальной черты можно записывать слово `union`.

В образце можно записать не всякое выражение XPath, а только путь, каждый шаг которого определяется осью (axis path), причем допускаются только три оси: `child` (по умолчанию), `//` и `attribute`. Это означает, что, находясь в каком-то узле, мы можем "увидеть", кроме него самого, только его атрибуты и узлы-потомки. Обратите внимание на то, что явная запись оси `descendant-or-self` недопустима, применяется только сокращенная запись `//`. Напомню также, что запись оси `attribute::` можно сократить до одной "собачки" @, а текущий узел очень часто обозначается точкой.

Хотя образец и строится как выражение языка XPath, но его цель — не отобразить последовательность узлов, а проверить соответствие узла данному образцу. Можно сказать, что некоторый узел *Node* соответствует некоторому образцу *Pattern* тогда и только тогда, когда узел *Node* принадлежит последовательности узлов — результату вычисления выражения `//Pattern`. Например, образцу `person//street` будут соответствовать все узлы из последовательности `//person//street`, а именно все узлы `street`, вложенные в узлы-элементы `person` даже через несколько промежуточных узлов.

Надо сразу же отметить, что перечисленные ограничения касаются только образцов. В других конструкциях языка XSLT можно применять выражения XPath в полном объеме.

Функции *id()* и *key()*

Образец может начинаться с вызова функции `id()` или функции `key()`. Единственным аргументом функции `id()` служит уникальный идентификатор узла, т. е. значение его атрибута типа `ID`. Например, образцу `id(@xref)`

соответствует узел-элемент, на идентификатор которого ссылается значение атрибута `xref`. Образцу `id("h12")` соответствует узел-элемент с идентификатором `h12`, а образцу `id("h12")//street` — его потомки `street` любого уровня вложенности. Внутренние перекрестные ссылки элементов XML друг на друга, организованные атрибутами типов `ID`, `IDREF` и `IDREFS`, мы обсуждали в начале главы 5.

Функция `id()`, работающая с атрибутами типа `ID`, подвержена всем ограничениям этого типа, а именно:

- ❑ у элемента может быть только один атрибут типа `ID`, значит, можно ссылаться только на этот атрибут, не говоря уже о том, что такой атрибут может вообще отсутствовать;
- ❑ значение атрибута типа `ID` уникально, значит, нельзя обратиться сразу к нескольким элементам;
- ❑ значение атрибута типа `ID` может быть только именем типа `Name`, значит, не может быть числом, содержать пробелы и многие другие символы;
- ❑ схема, в которой определен тип `ID` атрибута, может оказаться недоступной для процессора XSLT и он не "узнает" тип атрибута;
- ❑ функция `id()` работает только с атрибутами, но не с элементами, их содержимым и прочими узлами дерева документа;
- ❑ для внесения новых перекрестных ссылок в документ XML надо его менять, вставляя атрибуты типа `ID`, что может быть нежелательно или невозможно.

Функция `key()` менее требовательна. Она снимает ограничения функции `id()`, обращаясь не к атрибуту элемента, находящегося в документе XML, а к атрибуту `use` специального элемента `xsl:key`, расположенного в таблице стилей. У нее два аргумента:

- ❑ имя ключа, задаваемое строкой типа `QName`; это значение атрибута `name` элемента `xsl:key`, позволяющее выбрать нужный элемент `xsl:key`.
- ❑ значение ключа, представляющее собой выражение, в результате вычисления которого получается последовательность из одного или нескольких атомарных значений типа `xdt:anyAtomicType`.

Образец с функцией `key()` выглядит примерно так:

```
key("addr", 230007)//street.
```

Элемент `xsl:key`, который мы опишем ниже, определяет ключ и множество его значений, а функция `key()` выбирает из этого множества те значения, что совпадают со вторым аргументом функции `key()`. Таким образом, при необходимости создания новых ссылок в исходном документе XML, его не надо изменять, достаточно добавить новые элементы `xsl:key` в таблицу стилей.

Элементы, объявленные в XSLT

Язык XSLT объявляет около полусотни элементов. Из них 17 элементов верхнего уровня (top-level), которые могут быть непосредственно вложены в корневой элемент `xsl:stylesheet` таблицы стилей. Они называются *декларациями* (declarations). Это элементы `xsl:import`, `xsl:include`, `xsl:attribute-set`, `xsl:character-map`, `xsl:date-format`, `xsl:decimal-format`, `xsl:function`, `xsl:import-schema`, `xsl:key`, `xsl:namespace-alias`, `xsl:output`, `xsl:param`, `xsl:preserve-space`, `xsl:sort-key`, `xsl:strip-space`, `xsl:template`, `xsl:variable`. Более того, все эти элементы, кроме `xsl:param` и `xsl:variable`, можно записывать только на верхнем уровне вложенности, непосредственно в корневом элементе `xsl:stylesheet`.

Кроме элементов верхнего уровня вложенности, в языке XSLT объявлено более тридцати элементов, которые можно записывать в теле элементов верхнего уровня. Наиболее часто применяются элементы `xsl:apply-templates`, `xsl:value-of`, `xsl:copy-of`, `xsl:sort`, `xsl:text`.

Из всех элементов XSLT не верхнего уровня вложенности выделяются *инструкции* (instructions). Не путайте инструкции XSLT и инструкции по обработке XML. Формально инструкции XSLT определяются как элементы, которые можно вставлять в конструктор последовательности (sequence constructor). К инструкциям, в частности, относятся элементы, создающие узлы всех семи видов и последовательности узлов. Это элементы `xsl:element`, `xsl:attribute`, `xsl:text`, `xsl:comment`, `xsl:processing-instruction`, `xsl:namespace`, `xsl:result-document`, `xsl:sequence`.

Кроме них, инструкциями считаются элементы `xsl:apply-templates`, `xsl:value-of`, `xsl:variable` и др., всего более двадцати элементов. К инструкциям относятся как элементы, управляющие выбором правил преобразования: `xsl:if`, `xsl:for-each`, `xsl:choose`, так и элементы, копирующие узлы: `xsl:copy`, `xsl:copy-of`.

Мы не будем подробно рассматривать все элементы языка XSLT. Сначала кратко опишем наиболее часто употребляемые элементы, затем научимся их применять.

Все элементы необязательны и могут располагаться в любом порядке, за одним исключением: декларации `xsl:import`, если они есть, должны быть записаны первыми. С них и начнем.

Декларация *xsl:import*

Элемент `xsl:import` записывается очень просто:

```
<xsl:import href="адрес URI таблицы стилей" />
```

Его можно записать только непосредственно в корневом элементе `xsl:stylesheet` и только в самом начале таблицы стилей. Элементов `xsl:import` может быть несколько.

Процессор XSLT отыскивает таблицу стилей по указанному атрибуту `href` адресу и подставляет ее на место элемента `xsl:import` перед преобразованием. Например:

```
<xsl:import href="namts.xml" />
<xsl:import href="http://some.domain/nautus.xml" />
```

Некоторые правила преобразования из таблиц, импортируемых элементами `xsl:import`, могут конфликтовать с правилами, импортированными из других таблиц или определенными в самой таблице стилей. В таком случае чаще всего применяются те правила, которые записаны последними. Поэтому порядок записи элементов `xsl:import` в таблицу стилей имеет большое значение. Мы будем отмечать влияние порядка записи стилей при описании элементов XSLT.

Декларация *xsl:include*

Второй элемент, включающий внешние таблицы стилей в данную таблицу, — это элемент `xsl:include`. Он записывается точно так же, как элемент `xsl:import`, и оказывает такое же действие:

```
<xsl:include href="адрес URI таблицы стилей" />
```

Его можно записать не только непосредственно в корневом элементе `xsl:stylesheet`, но, в отличие от элемента `xsl:import`, в любом месте таблицы стилей. Второе отличие от элемента `xsl:import` заключается в том, что порядок записи элементов `xsl:include` в таблице стилей не имеет значения.

Декларация *xsl:import-schema*

Элемент `xsl:import-schema` чуть более сложен:

```
<xsl:import-schema namespace="идентификатор"
  schema-location="адрес URI схемы" />
```

Его тоже можно записать только непосредственно в корневом элементе `xsl:stylesheet`. Элементов `xsl:import-schema` может быть несколько.

Процессор XSLT отыскивает схему с идентификатором, записанным в атрибуте `namespace`, по указанному атрибуту `schema-location` адресу и подставляет ее на место элемента `xsl:import` перед преобразованием. Например:

```
<xsl:import-schema namespace="http://some.domain/names"
  schema-location="http://some.domain/pub/defs/names.xsd" />
```

Если местоположение схемы известно процессору XSLT, то атрибут `schema-location` можно не записывать. Если не записан атрибут `namespace`, то импортированные определения типов и объявления элементов и атрибутов попадают в пространство имен по умолчанию.

Декларация *xsl:variable*

Элемент `xsl:variable` определяет имя объекта. Оно записывается обязательным атрибутом `name`. Имя объекта, как обычно, должно быть уточненным именем XML типа `QName`. Кроме того, атрибутом `select` переменной можно задать сам объект, а атрибутом `as` определить тип объекта. Например:

```
<xsl:variable name="var1" select="1 to 10" />
```

После такого определения под именем `var1` будет пониматься последовательность атомарных значений 1, 2, ..., 10. Еще пример:

```
<xsl:variable name="var2" select="count(//person)" as="xs:integer" />
```

Имя `var2` будет хранить число элементов `person`.

Объект может быть получен из содержимого элемента `xsl:variable`:

```
<xsl:variable name="var3">10</xsl:variable>
```

или создан конструктором последовательности:

```
<xsl:variable name="var4">
  <xsl:value-of select="count(//person)" />
</xsl:variable>
```

Если объект не получен из атрибута `select` или содержимого элемента `xsl:variable`, то по умолчанию имя связывается с пустой строкой.

Для того чтобы получить объект, связанный с именем, определенным элементом `xsl:variable`, перед именем надо поставить знак доллара: `$var1`, `$var2`. При этом следует учитывать область действия имени.

Область действия имени простирается на весь элемент, в котором оно определено, начиная с места определения и включая вложенные элементы, если только в них не определено то же самое имя. Например:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:variable name="x" select="1" />

  <xsl:template match="/a">

    <xsl:variable name="y" select="0" />
    Здесь $x = 1, $y = 0.

  </xsl:template>
```



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

    <xsl:variable name="x" select="2" />
    Здесь $x = 2, $y неизвестно

</xsl:template>

<xsl:template match="/c">

    Здесь $x = 1, $y неизвестно

</xsl:template>

</xsl:stylesheet>
```

Имена, определенные непосредственно в корневом элементе `xsl:stylesheet`, называются *глобальными* именами, остальные — *локальными* именами.

Хотя слово "variable" и переводится с английского языка как "переменная", имя, созданное элементом `xsl:variable` — это не имя переменной, его значение нельзя изменить. Это только название некоторого объекта, которое удобно использовать в тех случаях, когда объект надо использовать во многих местах таблицы стилей, а его вычисление сложно или громоздко.

Декларация `xsl:param`

Элемент `xsl:param` записывается или непосредственно в элементе `xsl:stylesheet`, чтобы задать параметр преобразования, или в элементе `xsl:template`, чтобы задать параметр правила, или в элементе `xsl:function` как аргумент функции. У него один обязательный атрибут `name`, определяющий имя параметра. Кроме него, часто присутствует необязательный атрибут `select`, в котором записывается выражение для получения значения параметра:

```
<xsl:param name="p1" select="10 + 20" />
```

Если атрибут `select` отсутствует, то значение параметра берется из содержимого элемента, которым может быть конструктор последовательности узлов и атомарных значений:

```
<xsl:param name="p2">10</xsl:param>
```

Если отсутствует и атрибут `select`, и содержимое элемента, то параметр получает значение пустой строки.

Для получения значения параметра надо записывать его имя со знаком доллара: `$p1`, `$p2`. Например:

```
<xsl:when test="$p1=10">
```

Правила, определяющие область видимости параметров, такие же, как и у имен объектов, определенных декларацией `xsl:variable`.

Еще один необязательный атрибут `as` содержит желательный тип, к которому будет приведено значение параметра.

Наконец, последний атрибут `required`, принимающий значения `yes` или `no` (по умолчанию), указывает обязательность параметра. Если параметр обязателен, `required="yes"`, то элемент `xsl:param` должен быть пустым и не содержать атрибут `select`. В таком случае он получит определенное значение при вызове функции или элементом `xsl:with-param` при вызове шаблона.

Элемент *xsl:with-param*

Элемент `xsl:with-param` ссылается на некоторый параметр, имя которого записано в обязательном атрибуте `name`. Необязательным атрибутом `select` можно задать выражение, результат вычисления которого будет новым значением параметра:

```
<xsl:with-param name="p1" select="100 * 20" />
```

Новое значение можно задать и в содержимом элемента:

```
<xsl:with-param name="p1">100</xsl:with-param>
```

Элемент `xsl:with-param` используется только в инструкциях `xsl:apply-templates`, `xsl:apply-imports`, `xsl:call-template`.

Инструкция *xsl:value-of*

Элемент `xsl:value-of` вычисляет выражение, записанное в его обязательном атрибуте `select`, и преобразует его в строку. Например, выше мы определили имя объекта `var2`. Если попробовать записать значение этого имени просто в элементе XSLT, например:

```
<xsl:text>$var2</xsl:text>
```

то мы так и получим текст `$var2`. Чтобы получить сам объект, надо написать элемент `xsl:value-of`:

```
<xsl:value-of select="$var2" />
```

Если в результате вычисления выражения получается последовательность, то процессор XSLT версии 1.0 выберет из нее только первый элемент, преобразованный в строку. Процессор XSLT 2.0 преобразует каждый элемент последовательности в строку и объединит полученные строки в единую строку, оставив между ними по одному пробелу. Впрочем, Saxon 7.6 не подчиняется этому правилу и действует как процессор версии 1.0.

Если такое поведение процессоров вас не устраивает, то можно записать атрибут `separator`, а в нем явно указать строку, которая будет разделять элементы последовательности. Тогда и процессор XSLT версии 1.0 будет выводить всю последовательность. Например, следующий элемент `xsl:value-of`:

```
<xsl:variable name="x" select="(1,2,3)" />
<xsl:value-of select="$x" separator=" | " />
```

выведет строку `1 | 2 | 3`. В ней разделителем служит строка, состоящая из пробела, вертикальной черты и еще одного пробела. Если записать другой разделитель, состоящий из запятой и пробела:

```
<xsl:value-of select="$x" separator=", " />
```

то получим строку `1, 2, 3`.

Инструкции управления

xsl:if, xsl:for-each, xsl:choose

Язык XSLT наделен элементами, играющими роль операторов управления, таких же как в алгоритмических языках: условного оператора, оператора цикла, оператора варианта.

Элемент `xsl:if` запускает конструктор последовательности, содержащийся в его теле, только если истинно выражение, записанное в обязательном и единственном атрибуте `test`:

```
<xsl:if test="$x > $y">
  <xsl:value-of select="$x" /> больше <xsl:value-of select="$y" />
</xsl:if>
```

У элемента `xsl:for-each` в обязательном и единственном атрибуте `select` записывается выражение, дающее в результате последовательность. Для каждого члена этой последовательности выполняется конструктор, содержащийся в теле элемента `xsl:for-each`. В результате получается цикл, выполняющийся столько раз, сколько элементов у последовательности, полученной в результате вычисления выражения атрибута `select`. Следующий пример, приведенный в листинге 8.13, строит таблицу HTML, содержащую список имен.

Листинг 8.13. Построение таблицы с помощью цикла

```
<table>

<xsl:for-each select="name">
```

```
|  |  |  |
| --- | --- | --- |
| <xsl:value-of select="@first" /></td>  <xsl:value-of select="@second" /></td>  <xsl:value-of select="@surname" /></td> | | |

</xsl:for-each>
</table>

```

У элемента `xsl:choose` нет ни одного атрибута, но в его теле записывается один или несколько элементов `xsl:when` и один необязательный элемент `xsl:otherwise`:

```

<xsl:choose>
  <xsl:when test="$day=1">Понедельник</xsl:when>
  <xsl:when test="$day=2">Вторник</xsl:when>
  <xsl:when test="$day=3">Среда</xsl:when>
  <xsl:when test="$day=4">Четверг</xsl:when>
  <xsl:when test="$day=5">Пятница</xsl:when>
  <xsl:when test="$day=6">Суббота</xsl:when>
  <xsl:when test="$day=7">Воскресенье</xsl:when>
  <xsl:otherwise>Ошибка определения дня недели</xsl:otherwise>
</xsl:choose>

```

У элемента `xsl:when` есть только один обязательный параметр `test`, содержащий логическое выражение, и тело, содержащее конструктор последовательности.

Элемент `xsl:otherwise` не имеет атрибутов, у него есть только тело, содержащее конструктор последовательности.

В инструкции `xsl:choose` всегда выполняется не больше одного варианта. Варианты `xsl:when` просматриваются в порядке их написания в инструкции `xsl:choose`. Как только будет найден вариант с истинным значением выражения `test`, он будет выполнен, и результат этого варианта будет результатом всей инструкции. Варианты, следующие за ним по порядку, не рассматриваются. Если ни один вариант не подойдет, то результатом инструкции будет результат конструктора, записанного в элементе `xsl:otherwise`. Если элемента `xsl:otherwise` в инструкции нет, то результатом будет пустая строка.

Упражнения

1. Используя инструкцию `xsl:if`, уберите лишнюю запятую из списка телефонов листинга 8.10.

2. Уберите лишнее слово "Рабочие:" из листинга 8.10.
3. С помощью функции `position()` перенумеруйте строки таблицы, создаваемой в листинге 8.13.

Декларация *xsl:function*

Элемент `xsl:function` позволяет описывать самые настоящие пользовательские функции. Имя функции записывается в обязательном атрибуте `name`, аргументы функции задаются элементами `xsl:param`, а тело функции — это конструктор последовательности, записанный в содержимом элемента `xsl:function`. Результатом функции будет последовательность, созданная конструктором. Тип функции можно указать необязательным атрибутом `as`.

Имя функции — это уточненное имя типа `QName`, причем оно должно обязательно записываться с префиксом.

Все аргументы функции позиционные, следовательно, все элементы `xsl:param` должны быть записаны в начале тела функции и порядок их записи имеет значение при вызове функции. У аргументов функции не может быть значений по умолчанию, следовательно, элементы `xsl:param` должны быть пустыми и у них не должно быть атрибутов `select`. Например:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:usr="http://myexamples.com"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:function name="usr:sum" as="xs:integer">

    <xsl:param name="x" as="xs:integer" />
    <xsl:param name="y" as="xs:integer" />

    <xsl:value-of select="$x + $y" />

  </xsl:function>
```

Вызвать функцию можно в любом выражении подходящего типа, записав ее имя и аргументы в скобках. Например:

```
<xsl:value-of select="10 + 2 * usr:sum(2, 3)" />
```

Функция может вызываться рекурсивно. В листинге 8.14 приведен классический пример вычисления факториала с помощью функции `usr:fact()`, вызывающей саму себя.

Листинг 8.14. Рекурсивная функция, вычисляющая факториал

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

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```

xmlns:usr="http://myexamples.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xsl:output method="text" />

<xsl:function name="usr:fact">

    <xsl:param name="n" as="xs:integer" />

    <xsl:number value="if ($n = 1) then 1
                      else $n * usr:fact($n - 1)" />

</xsl:function>

<xsl:template match="/">
    <xsl:value-of select="usr:fact(10)" />
</xsl:template>

</xsl:stylesheet>

```

Упражнение

Запишите функцию, вычисляющую факториал не рекурсивно, а в цикле.

Декларация *xsl:template*

Элемент `xsl:template` определяет *шаблонное правило* (template rule) преобразования. Своим атрибутом `match` он задает образец (pattern) для отбора узлов, подлежащих преобразованию, а в теле содержит конструктор последовательности (sequence constructor) узлов и атомарных значений, которая и будет результатом преобразования отобранных по образцу узлов.

```
<xsl:template match="Образец" name="Имя">Конструктор</xsl:template>
```

Каждый из атрибутов `match` и `name` не обязателен, но хотя бы один из них должен присутствовать.

Атрибут `match` содержит образец для отбора преобразуемых узлов. Например:

```

<xsl:template match="zip">
    Почтовый код: <xsl:value-of select="." />
</xsl:template>

```

Это правило преобразует элементы `zip`, снабжая их пояснительным текстом.

Атрибут `name` определяет имя шаблона, превращая правило в *именованный шаблон* (named template). Имя шаблона — это обычное уточненное имя XML типа `QName`. Шаблон можно вызвать по имени элементом `xsl:call-template`, а если он не содержит атрибута `match`, то такой вызов обязателен,

поскольку неизвестны узлы, к которым его надо применить, и он не будет применяться автоматически. Очень часто именованный шаблон содержит параметры, заданные элементами `xsl:param`, и вызывается с различными параметрами совсем как обычная функция. Например:

```
<xsl:template name="compare">

  <xsl:param name="x" select="2" />
  <xsl:param name="y" select="2" />

  <xsl:if test="$x < $y">Меньше</xsl:if>
  <xsl:if test="$x >= $y">Не меньше</xsl:if>

</xsl:template>
```

У элемента `xsl:template` могут быть дополнительные атрибуты `mode`, `as`, `priority`.

Атрибут `mode` определяет режим обработки. Его значение — одно или несколько имен типа `QName`, под которыми будут известны режимы данного правила. Затем эти имена будут использованы для выбора правила с нужным режимом. Например:

```
<xsl:template match="address" mode="red">

  <div style="color:red">
    <xsl:value-of select="." />
  </div>

</xsl:template>

<xsl:template match="address" mode="blue">

  <div style="color:blue">
    <xsl:value-of select="." />
  </div>

</xsl:template>
```

Значение `#default` или отсутствие атрибута `mode` означает режим по умолчанию. Значение `#all` указывает на то, что правило применяется во всех режимах.

Атрибут `as` указывает желаемый тип результата. Полученная последовательность будет приведена к этому типу.

Атрибут `priority` назначает правилу приоритет, который будет учитываться при отборе правил, применимых к некоторому узлу. Приоритет — это просто действительное число, положительное или отрицательное. Большее число означает больший приоритет.

При вызове именованного шаблона элементом `xsl:call-template` атрибуты `match`, `mode` и `priority` игнорируются.

Если у именованного шаблона нет атрибута `match`, то у него не должно быть и атрибутов `mode` и `priority`, в них просто нет никакого смысла.

Упражнения

1. Запишите шаблон выбора всех населенных пунктов из записной книжки листинга 8.7.
2. Запишите шаблон выбора всех фамилий из записной книжки листинга 8.7.

Инструкция `xsl:apply-templates`

Элемент `xsl:apply-templates`, записываемый чаще всего внутри элемента `xsl:template`, в простейшем виде пуст:

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

Он предписывает обработать рекурсивно все узлы-потомки узлов, отобранных родительским элементом `xsl:template`.

У элемента `xsl:apply-templates` есть два необязательных атрибута.

Атрибут `select`, значением которого должно быть выражение, дающее последовательность узлов, может ограничить обработку только указанными в нем узлами. Например:

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

Атрибут `mode` выбирает режим обработки из режимов, уже определенных в элементах `xsl:template`. Режим — это любое имя типа `QName`, но два режима предопределены. Это текущий режим, отмечаемый словом `#current`, и режим по умолчанию, принимаемый при отсутствии атрибута `mode`, или отмечаемый явно словом `#default`. Например, правило, определенное выше, можно применить так:

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

Содержимым элемента `xsl:apply-templates` могут служить элементы `xsl:sort` и `xsl:with-param`.

Инструкция `xsl:call-template`

Элемент `xsl:call-template` вызывает именованный шаблон, имя которого записано в единственном и обязательном атрибуте `name`. Элемент может содержать значения параметров, заданные элементами `xsl:with-param`. Например:


```
<xsl:call-template name="compare">
  <xsl:with-param name="x" select="0" />
  <xsl:with-param name="y" select="10" />
</xsl:call-template>
```

В результате применения инструкции `xsl:call-template` получается последовательность, созданная конструктором вызванного шаблона.

Инструкции *xsl:attribute*, *xsl:element*, декларация *xsl:attribute-set*

Язык XSLT позволяет не только преобразовывать уже существующие узлы, но и создавать новые. Мы уже встречались с инструкциями `xsl:text` и `xsl:value-of`, создающими текстовые узлы. Подобные инструкции есть для каждого из семи видов узлов. Мы рассмотрим только инструкции, создающие узлы-атрибуты и узлы-элементы, остальные инструкции очень просты.

У инструкций `xsl:attribute` и `xsl:element` по одному обязательному атрибуту `name`, задающему имя атрибута или элемента. Значение атрибута или содержимое элемента записывается в теле инструкций конструктором последовательности. Например:

```
<xsl:attribute name="attr1">Value1</xsl:attribute>
<xsl:element name="elem1">Content1</xsl:element>
```

Кроме обязательного атрибута `name` у обеих инструкций есть одинаковые необязательные атрибуты. Необязательный атрибут `namespace` определяет пространство имен. Необязательные атрибуты `validation` и `type` позволяют процессору сделать проверку типа атрибута или элемента, определенного в схеме документа.

Еще один полезный элемент — декларация `xsl:attribute-set` — позволяет объединить определения нескольких атрибутов под одним именем, которое затем можно будет использовать в элементах `xsl:element`, `xsl:copy` или в другом элементе `xsl:attribute-set`.

У декларации `xsl:attribute-set` есть один обязательный атрибут `name`, имеющий множество атрибутов. Сами атрибуты определяются в теле декларации. В следующем примере создается два множества атрибутов `AS1` и `AS2`, каждое множество содержит определения двух атрибутов:

```
<xsl:attribute-set name="AS1">
  <xsl:attribute name="attr11">attr11-value</xsl:attribute>
  <xsl:attribute name="attr12">attr12-value</xsl:attribute>
</xsl:attribute-set>
```

```
<xsl:attribute-set name="AS2">

  <xsl:attribute name="attr21">attr21-value</xsl:attribute>
  <xsl:attribute name="attr22">attr22-value</xsl:attribute>

</xsl:attribute-set>
```

Один необязательный атрибут `use-attribute-sets` декларации `xsl:attribute-set`, в котором перечисляются имена множеств атрибутов, позволяет включить в определяемое множество атрибутов одно или несколько уже определенных множеств.

У инструкции `xsl:element` тоже есть необязательный атрибут `use-attribute-sets`, с помощью которого создаются атрибуты элемента из предварительно определенного множества атрибутов. В следующем примере создается элемент `someelem` с четырьмя атрибутами из множеств `AS1` и `AS2`:

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

  <xsl:element name="someelem" use-attribute-sets="AS1 AS2">
    Какое-то содержимое
  </xsl:element>

</xsl:template>
```

Инструкции *xsl:copy*, *xsl:copy-of*

Элемент `xsl:copy` просто копирует текущий узел, не копируя его узлы-атрибуты и узлы-потомки. Копирование узлов пространств имен зависит от значения `"yes"` или `"no"` атрибута `copy-namespaces`. По умолчанию принимается значение `"yes"`. Например:

```
<xsl:copy copy-namespaces="no" />
```

Если копируется узел, у которого могут быть потомки, т. е. узел-элемент или корневой узел документа, то инструкция `xsl:copy` может не только копировать, но и добавить к нему узлы-потомки, узлы-атрибуты и узлы пространств имен. Для этого в содержимом элемента `xsl:copy` надо записать конструктор добавляемых узлов, например:

```
<xsl:copy>

  <xsl:attribute name="xref">
    <xsl:text>http://www.bhv.ru/</xsl:text>
  </xsl:attribute>

  <xsl:element name="nested">
```

```
<xsl:attribute name="nested-attr">
  <xsl:text>nested-attr-value</xsl:text>
</xsl:attribute>

<xsl:text>nested-content</xsl:text>

</xsl:element>

</xsl:copy>
```

Узлы-атрибуты копируемого узла-элемента или корневого узла документа можно создать другим способом — записать в атрибуте `use-attribute-sets` имена предварительно созданных множеств атрибутов. Следующий пример добавляет к элементу `elem` четыре атрибута, предварительно определенные и записанные во множества атрибутов `AS1` и `AS2`, созданные в предыдущем разделе:

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

  <xsl:copy use-attribute-sets="AS1 AS2">
    <xsl:value-of select="." />
  </xsl:copy>

</xsl:template>
```

В тех случаях, когда надо рекурсивно скопировать все дерево или некоторое его поддерево, удобнее использовать инструкцию `xsl:copy-of`. Она копирует не текущий узел, а узлы, полученные из выражения, записанного в обязательном атрибуте `select`. При этом рекурсивно копируются все узлы-потомки, узлы-атрибуты и узлы пространств имен. Копирование последних можно отменить, записав атрибут `copy-namespaces="no"`.

Элемент `xsl:copy-of` пуст, у него нет содержимого, следовательно, он не может ничего добавить к копируемому элементу. Например, копирование всего документа XML можно сделать так:

```
<xsl:template match="/">
  <xsl:copy-of select="." />
</xsl:template>
```

Для копируемых узлов-элементов и узлов-атрибутов дополнительные атрибуты `validation` и `type` элементов `xsl:copy` и `xsl:copy-of` позволяют сделать проверку их типов, определенных в схеме документа.

Элемент *xsl:sort*, декларация *xsl:sort-key*

Элементы `xsl:sort` сортируют узлы, уже отобранные для преобразования, поэтому их можно записать только в элементах `xsl:apply-templates`,

xsl:for-each, xsl:for-each-group, и еще в декларации xsl:sort-key, причем элементы xsl:sort следует записывать первыми по порядку. Ключ сортировки записывается в необязательном атрибуте select. Пусть, например, надо отсортировать список сотрудников, составленный из элементов emp, по их табельным номерам, записанным в атрибуте id:

```
<emp id="234002">Сведения о сотруднике</emp>
```

Правило сортировки может выглядеть так:

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

  <xsl:sort select="@id" />
  <!-- Еще какие-то преобразования -->

</xsl:for-each>
```

Если атрибут select не указан, то ключом сортировки считается строковое значение всего узла, т. е. значение ".".

Порядок сортировки определяется необязательным атрибутом order, принимающим одно из двух значений: сортировка по возрастанию ключа "ascending" (по умолчанию) или сортировка по убыванию ключа "descending".

Еще один необязательный атрибут case-order, тоже принимающий одно из двух значений "upper-first" или "lower-first", определяет, обрабатывать буквы верхнего регистра до букв нижнего регистра, или наоборот. Правило умолчания для этого атрибута устанавливается процессором XSLT.

Учитывать национальные особенности сортировки позволяет необязательный атрибут lang. Его значение по умолчанию равно значению атрибута xml:lang.

Наконец, вместо двух последних атрибутов можно записать атрибут collation, в котором надо прямо указать все правила сравнения узлов для их сортировки.

От первой версии языка XSLT остался еще один атрибут data-type, разделяющий сортировку строк (значение "text") и сортировку чисел (значение "number"). Дело в том, что если сортировать числа как строки, состоящие из цифр, то мы получим лексикографический порядок 1, 10, 11, ..., 19, 2, 20, ... Подобная несуразица получится и при сортировке по дате. Вторая версия языка XSLT автоматически определяет тип сортировки по типу сортируемых данных и не нуждается в этом атрибуте. При необходимости изменить тип сортировки во второй версии предпочитают изменять тип сортируемых данных функциями number(), date(), string() и другими функциями изменения типа, взятыми из языка XSD. Например:

```
<xsl:sort select="xs:date(@dob)" />
```

В тех случаях, когда надо сделать сортировку по нескольким ключам, записывается несколько элементов `xsl:sort`. Поскольку они записываются в начале содержимого своего родительского элемента, они будут записаны подряд. При этом элемент, записанный первым, является первичным ключом сортировки, следующий элемент — вторичным ключом и т. д. Пусть, например, мы хотим отсортировать элементы

```
<name first="Иван" second="Петрович" surname="Сидоров" />
```

сначала по фамилиям, затем по именам и отчествам. Правила сортировки будут выглядеть так:

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

  <xsl:sort select="@surname" />
  <xsl:sort select="@first" />
  <xsl:sort select="@second" />

</xsl:for-each>
```

Если такой составной ключ сортировки надо применять несколько раз, то удобно записать его заранее в декларации `xsl:sort-key`, назвав как-нибудь ключ в обязательном атрибуте `name`, например:

```
<xsl:sort-key name="famsort">

  <xsl:sort select="@surname" />
  <xsl:sort select="@first" />
  <xsl:sort select="@second" />

</xsl:sort-key>,
```

а затем только указывать имя ключа сортировки в функции `sort()`:

```
<xsl:copy-of select="sort(//name, 'famsort')"/>
```

Декларацию `xsl:sort-key` можно применять и просто для того, чтобы дать имя ключу сортировки.

Декларация *xsl:key*

Элемент `xsl:key` выделяет некоторую совокупность узлов и/или атомарных значений, образующих множество значений ключа, и дает ей имя — имя ключа. Он играет ту же роль, что индекс в книге или индекс таблицы в базе данных. У элемента `xsl:key` три обязательных атрибута:

```
<xsl:key name="Имя_ключа" match="Образец" use="Признак" />
```

Атрибут `name` типа `QName` определяет имя ключа. Атрибут `match`, значением которого может быть любой образец, задает шаблон для выделения множе-

ства значений ключа. Атрибут `use` — произвольное выражение XPath — показывает признак, по которому будут отбираться значения ключа. Например, следующий ключ `addr`:

```
<xsl:key name="addr" match="//address" use="zip" />
```

содержит все элементы `address`, а нужный элемент будет отбираться по почтовому индексу — вложенному элементу `zip`. Созданный ключ затем используется функцией `key()`, например, таким образом:

```
<xsl:value-of select="key('addr', 230005)/street" />
```

Декларация *xsl:output*

Язык XSLT не предназначен для вывода результата преобразования на конечное устройство вывода. Для этого можно применить вторую часть языка XSL — форматирование объектов FO. Тем не менее, элемент `xsl:output` позволяет сформировать результат преобразования в виде документа для вывода на стандартное устройство, которое может быть связано с консолью, принтером, сетевой картой. При этом на устройство вывода идет поток байтов, поэтому процесс вывода называется *сериализацией* (serialization), т. е. "вытягиванием" дерева, полученного в результате преобразования, в последовательность байтов.

Элемент `xsl:output` вкладывается непосредственно в корневой элемент `xsl:stylesheet` таблицы стилей. Он пуст и не содержит ни одного обязательного атрибута, но зато содержит много необязательных атрибутов. Перечислим их.

```
<xsl:output
  name = имя, используемое затем в элементе xsl:result-document
  method = "xml" или "html" или "xhtml" или "text" или имя типа QName
  cdata-section-elements = имена элементов, выводимых в секции CDATA
  doctype-public = публичный идентификатор для создаваемого DTD
  doctype-system = системный идентификатор для создаваемого DTD
  encoding = кодировка выходного документа, по умолчанию "UTF-8"
  escape-uri-attributes = "yes" (по умолчанию) или "no"
  include-content-type = "yes" (по умолчанию) или "no"
  indent = "yes" (по умолчанию для html и xhtml) или
           "no" (по умолчанию для xml)
  media-type = "text/plain" или "text/xml" или "text/html" или
              "text/xhtml" или другой MIME-тип
  normalize-unicode = "yes" или "no" (по умолчанию)
  omit-xml-declaration = "yes" или "no" (по умолчанию)
  standalone = "yes" или "no"
  undeclare-namespaces = "yes" или "no" (по умолчанию)
```

```
use-character-maps = список имен элементов xsl:character-map  
version = версия языка выходного документа />
```

Назначение почти всех атрибутов понятно даже из такого краткого пояснения. Большинство атрибутов формируют пролог документа XML, следовательно, нужны в том случае, если указан атрибут `method="xml"`. Последний атрибут `version` равен "1.0" для вывода в формате XML, "4.0" для HTML и т. д.

Инструкция `xsl:result-document`

Результирующее дерево всегда строится неявно, если преобразование корневого узла исходного документа привело к появлению непустой последовательности. Если же надо указать корневой узел результирующего дерева явно, или надо построить несколько деревьев, то применяется инструкция `xsl:result-document`. В самом простом виде это элемент без атрибутов, содержащий в теле конструктор последовательности, по которой будет построено результирующее дерево. Неявное построение дерева в точности эквивалентно действию элемента `xsl:result-document` без атрибутов.

В более сложном случае инструкция `xsl:result-document` позволяет сформировать не только дерево, но и документ в окончательном виде. Для этого надо указать в атрибуте `format` имя декларации `xsl:output`, формирующей документ, а в атрибуте `href` — файл, в который будет записан результирующий документ. Имя файла записывается в форме URI. Если атрибут `format` отсутствует, то будет применена декларация `xsl:output` без имени, т. е. без атрибута `name`. Пример формирования нескольких документов приведен в листинге 8.22.

Последовательность преобразований

Теперь, после краткого описания основных элементов языка XSLT, можно понять, каким образом они выполняют преобразование документа XML.

Как видно из листингов 8.6, 8.9 и 8.12, документ XSLT — таблица стилей — состоит из шаблонных правил преобразования (template rules), записанных элементами `xsl:template`. В каждом правиле атрибутом `match` перечисляется последовательность исходных узлов, для преобразования которых предназначено данное правило, а в содержимом элемента `xsl:template` задается конструктор последовательности (sequence constructor) преобразованных узлов. Из каждого узла исходного дерева получается один или несколько узлов преобразованного дерева. Порядок записи правил не имеет значения, поскольку для каждого узла исходного дерева просматривается вся таблица стилей в поисках подходящего правила.

Применение правил преобразования

Обычно просмотр исходного документа XML начинается с его корневого узла. Для него подбирается правило преобразования, конструктор которого явно или неявно создает корневой узел преобразованного документа. Для этого он может явно применить элемент `xsl:result-document`. Затем конструктор создает последовательность узлов, которые будут потомками корневого узла нового документа.

Если для какого-то узла дерева документа в таблице стилей не оказалось соответствующего ему правила, то к нему применяется встроенное в процессор XSLT правило по умолчанию. Правила по умолчанию опираются на вид узла.

- ❑ Для корневого узла и узла-элемента правило по умолчанию означает просмотр его потомков, т. е. неявно выполняется правило

```
<xsl:apply-templates mode="#current" />
```

- ❑ Для узла-атрибута и текстового узла встроенное правило по умолчанию создает текстовый узел, содержащий их значения. Если значение атрибута или текстового узла пусто, то правило не делает ничего, точнее говоря, создает пустую последовательность узлов.
- ❑ Для узла-комментария, инструкций по обработке и узла пространства имен по умолчанию создается пустая последовательность узлов.

Например, таблица стилей листинга 8.6 состояла только из правил умолчания и выводила только текстовые узлы, как предписано декларацией `xsl:output`.

С другой стороны, для какого-то исходного узла в таблице стилей может оказаться несколько подходящих правил преобразования. Одним из них всегда будет правило по умолчанию. Как бы то ни было, к узлу всегда применяется только одно правило. Оно отбирается так:

- ❑ во-первых, из всех импортированных правил отбирается то, которое записано в таблицу стилей в последнюю очередь;
- ❑ во-вторых, импортированное правило отбрасывается, если есть подходящее правило в самой таблице стилей;
- ❑ в-третьих, отбирается правило с наибольшим значением атрибута `priority`. Например, из следующих двух правил, применимых к элементу `address`, будет взято первое:

```
<xsl:template match="address" priority="3">  
<xsl:template match="address" priority="2">
```

Если атрибут `priority` отсутствует, то правилу назначается приоритет по умолчанию, вычисляемый по виду образца, записанного в атрибуте `match`.

При этом:

- ❑ низший приоритет, равный $(-0,5)$, получают правила с образцами, содержащими тест узла `element()`, `element(*, *)`, `attribute()`, `attribute(@*, *)`, а также с образцом корневого узла `(/)`;
- ❑ приоритет $(-0,25)$ получают правила, чьи образцы содержат имена узлов вида `xxx:*` и `*:yyy`;
- ❑ нулевой приоритет у правил с образцами, содержащими конкретное имя узла или тест узла вида `element(E, *)`, `element(*, T)`, `attribute(@A, *)`, `attribute(*, T)`;
- ❑ приоритет $0,25$ получают образцы с тестом узла вида `element(E)`, `element(E, T)`, `attribute(@A)`, `attribute(@A, T)`;
- ❑ прочие тесты узла дают низший приоритет $(-0,5)$;
- ❑ остальные образцы дают своему правилу наивысший приоритет $0,5$.

Из этого описания видно, что больший приоритет получают правила, действующие на узлы, более конкретно описанные образцами. Скажем, правило с образцом `/AAA/BVV` будет приоритетнее правила с образцом `//BBV`.

Хотя назначение приоритета столь детально расписано, оно не всегда очевидно. Например, образец `address`, состоящий из одного явного имени узла (приоритет 0), будет менее приоритетен, чем образец `node()[self::*]`, указывающий на то же имя косвенно (приоритет 0,5). Поэтому все-таки лучше в сомнительных случаях явно указывать приоритет правила его атрибутом `priority`.

Наконец, если после всего этого отбора останется несколько подходящих правил с одинаковым приоритетом, то процессор XSLT выдаст сообщение об ошибке.

Создание преобразованных узлов

После того как к текущему узлу подобрано правило его преобразования, выполняется конструктор последовательности узлов и атомарных значений, записанный в этом правиле. Определение конструктора носит чрезвычайно общий характер.

Конструктор последовательности (sequence constructor) — это последовательность узлов таблицы стилей, имеющих общий родительский узел, применение которой дает последовательность узлов и атомарных значений. Общий родительский узел, о котором идет речь в определении, — это чаще всего узел-элемент `xsl:template`, хотя им могут служить элементы `xsl:variable`, `xsl:param` и многие другие элементы XSLT.

В конструкторе последовательности могут встречаться четыре вида узлов:

- ❑ текстовые узлы, которые будут просто скопированы в конструируемую последовательность. Таков, например, текст "ФИО:" из листинга 8.9;

- ❑ узлы-инструкции XSLT, генерирующие последовательности узлов и атомарных значений;
- ❑ внешние, не XSLT-инструкции с именами из пространства имен, отличного от пространства имен XSLT. Они тоже генерируют последовательности узлов и атомарных значений;
- ❑ элементы из пространства имен, отличного от пространства имен XSLT, не являющиеся внешними инструкциями (*literal result elements*). Они становятся узлами-элементами конструируемой последовательности. Это, например, теги HTML из листинга 8.12.

Узлы-инструкции, входящие в конструктор, выполняются и вместе с остальными узлами конструктора образуют одну последовательность узлов и/или атомарных значений. Все подряд идущие атомарные значения из созданной конструктором последовательности преобразуются в строки и собираются в один текстовый узел, в котором они записываются через пробел. Затем в полученной последовательности узлов все подряд идущие текстовые узлы собираются в один текстовый узел без всяких разделителей.

Итак, после применения каждого правила получается последовательность узлов, а после использования всех правил таблицы стилей ко всем выбранным узлам исходного дерева получается набор таких последовательностей. Осталось собрать из этого набора последовательностей одно или несколько деревьев, которые и будут результатом преобразования. Они создаются или явно элементами `xsl:result-document`, или неявно начальным правилом преобразования.

Очень часто преобразованное дерево выводится на какое-то устройство в виде документа HTML, XHTML, XML или просто в виде плоского ASCII-текста. В самом простом случае выполняется сериализация, для которой в языке XSLT предусмотрен уже упоминавшийся элемент `xsl:output`.

Тожественное преобразование

Типичная задача преобразования документа XML — это изменить один или несколько элементов, оставив остальные в неприкосновенности. Пусть нам понадобилось изменить элемент `name` в записной книжке из листинга 8.7 так, чтобы имя, отчество и фамилия содержались в теле элемента, а не в его атрибутах, т. е. вместо записи

```
<name first="Иван" second="Петрович" surname="Сидоров" />
```

написать

```
<name>Иван Петрович Сидоров</name>
```

Все остальное содержимое документа изменять не надо.

Недолго думая, пишем таблицу стилей, преобразующую все элементы `name` так, как задано. Она приведена в листинге 8.15.

Листинг 8.15. Попытка тождественного преобразования

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

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

  <xsl:output method="text" encoding="UTF-8" />

  <xsl:template match="name">

    <name>
      <xsl:value-of select="@first" /> <xsl:text> </xsl:text>
      <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
      <xsl:value-of select="@surname" />
    </name>

  </xsl:template>

</xsl:stylesheet>
```

Проверим это преобразование с помощью какого-либо процессора XSLT. Листинг 8.16 показывает результат преобразования листинга 8.7, сделанного процессором Xselerator по правилам листинга 8.15.

Листинг 8.16. Результат преобразования листинга 8.7 по правилам листинга 8.16

```
<?xml version="1.0" encoding="UTF-16"?>
<name>Иван Петрович Сидоров</name>
Садовая, 12 — 34Новокозловск123321123456654321
<name>Мария Ивановна Федорова</name>
Нижняя, 12Зареченск321123224321
```

Получилось совсем не то, что мы хотели. Куда-то пропали все теги, кроме тега `<name>`. Элемент `name` получился таким, каким мы его хотели видеть, но остальные элементы подверглись преобразованию по умолчанию, которое оставило только текстовые узлы, убрав теги всех элементов.

Необходимо написать еще одно правило, применяемое ко всем элементам и оставляющее их без изменения. Его приоритет должен быть ниже, чем приоритет преобразований, меняющих элементы. Это правило, вставленное в листинг 8.15, показано в листинге 8.17. Оно взято мною из спецификации языка XSLT. Поскольку ось `child` не показывает узлы-атрибуты и узлы пространств имен, кроме теста узла `node()` в образец вставлено выражение `@*`,

отбирающее все узлы-атрибуты. Преобразование заключается в простом копировании отобранных узлов инструкцией `xsl:copy`. Эта инструкция по умолчанию копирует узлы пространств имен, поэтому мы их не отбираем специально. Однако она не копирует автоматически узлы-атрибуты и узлы-потомки, поэтому в элемент `xsl:copy` вложен элемент `xsl:apply-templates`, перебирающий их.

Листинг 8.17. Тожественное преобразование всех элементов, кроме одного

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

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

  <xsl:template match="name">

    <name>
      <xsl:value-of select="@first" />   <xsl:text> </xsl:text>
      <xsl:value-of select="@second" />  <xsl:text> </xsl:text>
      <xsl:value-of select="@surname" />
    </name>

  </xsl:template>

  <xsl:template match="@* | node()">

    <xsl:copy>
      <xsl:apply-templates select="@* | node()" />
    </xsl:copy>

  </xsl:template>

</xsl:stylesheet>
```

Отбор отдельных узлов

Задача, противоположная предыдущей, — выбрать из документа XML только некоторые элементы. Пусть мы решили выбрать из записной книжки только элементы `name`, получив в результате документ, показанный в листинге 8.18.

Листинг 8.18. Отдельные элементы, выбранные из документа XML

```
<?xml version="1.0" encoding="windows-1251" ?>

<names>
```

```
<name first="Иван" second="Петрович" surname="Сидоров" />
<name first="Мария" second="Ивановна" surname="Федорова" />

</names>
```

Для этого надо опять подавить правила преобразования по умолчанию, заменив их правилами, создающими пустую последовательность узлов. Следует оставить только правило отбора элементов `name` и копирования их без изменения в результирующий документ. Соответствующая таблица стилей записана в листинге 8.19.

Листинг 8.19. Выбор отдельных элементов из документа XML

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

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="name">
    <xsl:copy-of select="." />
  </xsl:template>

  <xsl:template match="/">
    <names>
      <xsl:apply-templates />
    </names>
  </xsl:template>

  <xsl:template match="@* | node()">
    <xsl:apply-templates />
  </xsl:template>

</xsl:stylesheet>
```

Первое правило листинга 8.19 копирует элементы `name`. Второе правило создает корневой элемент `names` нового документа и вкладывает в него результаты преобразования. Третье правило подавляет правила преобразования по умолчанию.

Группировка элементов

Еще одна часто возникающая задача — сгруппировать элементы документа XML по какому-нибудь признаку. Например, пусть мы решили список имен, подобный списку из листинга 8.18, сгруппировать по фамилиям, получив документ XML, подобный тому, что записан в листинге 8.20.

Листинг 8.20. Имена, сгруппированные по фамилиям

```
<?xml version="1.0" encoding="windows-1251" ?>

<names>

  <family surname="Сидоров">

    <name first="Иван" second="Петрович" />
    <name first="Семен" second="Богданович" />

  </family>

  <family surname="Федорова">

    <name first="Мария" second="Ивановна" />

  </family>

</names>
```

В первой версии языка XSLT эта задача решалась остроумными трюками, подробно описанными в литературе. Вторая версия XSLT упростила ее, введя в язык инструкцию `xsl:for-each-group`. Дадим ее формальное описание.

Инструкция *xsl:for-each-group*

Элемент `xsl:for-each-group` отбирает последовательность узлов и атомарных значений своим обязательным атрибутом `select` и группирует ее элементы по признаку, заданному выражением, записанным в атрибуте `group-by`. Этот признак называется *ключом группы* (grouping key). Ключ группы вычисляется заново для каждого элемента последовательности. В результате исходная последовательность, отобранная атрибутом `select`, разбивается на несколько последовательностей. В следующем примере в одну группу собираются все узлы-элементы `name` с одинаковым значением атрибута `surname`:

```
<xsl:for-each-group select="name" group-by="@surname">
```

Выражение, являющееся значением атрибута `group-by`, может давать несколько ключей группы, и один узел может попасть сразу в несколько групп. Например:

```
<xsl:for-each-group select="name" group-by="(@second, @surname)">
```

Второй способ разбить последовательность на группы дает атрибут `group-adjacent`. Он собирает в группу все *подряд идущие* элементы последовательности с одинаковым значением ключа. Такой отбор возможен, если в атрибуте `group-adjacent` содержится только один ключ группы. Процессор XSLT

следит за тем, чтобы значением атрибута `group-adjacent` был один и только один ключ группы.

Третий и четвертый способы применимы к последовательностям, состоящим только из узлов, без атомарных значений. Эти способы применяют атрибут `group-starting-with` или атрибут `group-ending-with`. Значением этих атрибутов может быть не любое выражение, а только образец. В одну группу собираются все подряд идущие узлы, первый (последний) из которых удовлетворяет образцу. Остальные узлы из этой группы не будут удовлетворять образцу. Если несколько подряд идущих элементов последовательности удовлетворяют образцу, то они попадут в разные группы.

Итак, группы узлов создаются одним из четырех атрибутов элемента `xsl:for-each-group`: `group-by`, `group-adjacent`, `group-starting-with`, `group-ending-with`. В каждом элементе `xsl:for-each-group` должен быть один и только один из этих атрибутов. Полученные группы существуют и могут использоваться только в содержимом элемента `xsl:for-each-group`. В теле элемента `xsl:for-each-group` записывается конструктор последовательности, который выполняется по одному разу для каждой группы.

Для удобства работы с группами в язык XSLT введены функции `current-group()` и `current-group-key()`. Их можно использовать только в теле элемента `xsl:for-each-group`. У обеих функций нет аргументов. Результатом функции `current-group()` будет последовательность — текущая группа, а результатом функции `current-group-key()` — значение ключа текущей группы.

Решение задачи

Теперь, имея в распоряжении инструкцию `xsl:for-each-group`, нетрудно получить преобразованный документ листинга 8.20. Таблица стилей XSLT, решающая эту задачу, приведена в листинге 8.21.

Листинг 8.21. Группировка элементов листинга 8.19

```
<?xml version="1.0" encoding="windows-1251"?>

<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="names">

    <names>

      <xsl:for-each-group select="*" group-by="@surname">

        <family surname="{@surname}">
```

```

        <xsl:for-each select="current-group()">
            <name first="{@first}" second="{@second}" />
        </xsl:for-each>

    </family>

</xsl:for-each-group>

</names>

</xsl:template>

</xsl:stylesheet>

```

Вывод нескольких документов

В листинге 8.22 приведен пример таблицы стилей, просматривающей записную книжку листинга 8.7 и создающей несколько файлов с документами HTML. Файл `toc.html` содержит список абонентов из записной книжки. Он создается с помощью элемента `xsl:result-document`, ссылающегося на безымянный элемент `xsl:output`.

Каждый абонент ссылается с помощью тега `<a>` на отдельный HTML-файл `personN.html`, содержащий подробные сведения об абоненте. Число `N` принимает значения 1, 2, ..., вычисляемые функцией `position()`. Файлы `personN.html` создаются с помощью элемента `xsl:result-document`, ссылающегося на элемент `xsl:output` с именем `"person-format"`.

Листинг 8.22. Создание нескольких документов

```

<?xml version="1.0" encoding="windows-1251"?>

<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" indent="yes"
        encoding="windows-1251" />

    <xsl:output name="person-format" method="html" indent="yes"
        encoding="windows-1251" />

    <xsl:template match="/">

        <xsl:result-document href="file:///C:/xslt/proba/toc.html">

            <html><head><title>Abonent List</title></head>
            <body><h2>Список абонентов</h2>

```



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

  <p><a href="person{position()}.html">

    <xsl:value-of select="@surname"/><xsl:text> </xsl:text>
    <xsl:value-of select="@first"/><xsl:text> </xsl:text>
    <xsl:value-of select="@second"/>

  </a></p>

</xsl:for-each>

</body></html>

</xsl:result-document>

<xsl:for-each select="//person">

  <xsl:result-document format="person-format"
    validation="strip"
    href="file:///C:/xslt/proba/person{position()}.html">

    <html><head><title>
      <xsl:value-of select="name/@surname"/>
    </title></head><body>

    <xsl:apply-templates />

    </body></html>

  </xsl:result-document>

</xsl:for-each>

</xsl:template>

<xsl:template match="name">

  <p />
  <xsl:value-of select="@first" /> <xsl:text> </xsl:text>
  <xsl:value-of select="@second" /> <xsl:text> </xsl:text>
  <xsl:value-of select="@surname" /> <br />

</xsl:template>

<xsl:template match="address">

  <br />
  <xsl:value-of select="street" /> <xsl:text> </xsl:text>
```

```
<xsl:value-of select="city" /> <xsl:text> </xsl:text>
<xsl:value-of select="zip" /> <br />

</xsl:template>

<xsl:template match="phone-list">

    Рабочий: <xsl:value-of select="work-phone" /> <br />
    Домашний: <xsl:value-of select="home-phone" /> <br />

</xsl:template>

</xsl:stylesheet>
```

Процессоры XSLT

За недолгую историю языка XSLT выпущено множество коммерческих и свободно распространяемых процессоров XSLT. В этом разделе перечислены только некоторые из них.

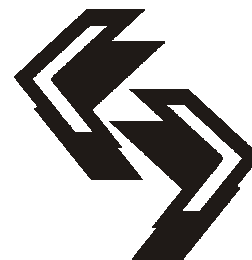
- ❑ Уже упоминавшийся нами процессор SAXON, написанный на Java Майклом Кэем (Michael H. Kay), реализует вторую версию языка XSLT. Он свободно распространяется и доступен на Web-странице <http://saxon.sourceforge.net/>.
- ❑ Процессор Xalan, разрабатываемый сообществом Apache Software Foundation, выпускается в двух вариантах: на языке Java (его называют Xalan-J), <http://xml.apache.org/xalan-j/>, и на языке C++ (Xalan-C++), <http://xml.apache.org/xalan-c/>. Как и все продукты Apache, процессор Xalan распространяется свободно.
- ❑ Свободно распространяемый инструментальный набор XML Core Services 4.0 корпорации Microsoft, <http://msdn.microsoft.com/xml/>, содержит процессор XSLT.
- ❑ Еще одна реализация XSLT-процессора на Java — это свободно распространяемый процессор XT, <http://www.blnz.com/xt/>.
- ❑ XSLT-процессор Napa реализован на языке C++ для Windows, FreeBSD и Linux. Он свободно распространяется на сайте <http://www.tfi-technology.com/xml/napa.html>.
- ❑ Еще один свободно распространяемый XSLT-процессор Unicorn написан на C++ для Windows, <http://www.unicorn-enterprises.com/>.

Кроме того, процессоры XSLT входят во все обработчики XML и во все серверы приложений, работающие с XML.

Вопросы для самопроверки

1. Почему в технологии XML предпочитают использовать таблицы стилей XSL, а не CSS?
2. В чем разница между языками XSL и XSLT?
3. Что понимается под преобразованием документа XML?
4. Что такое форматирование объектов FO?
5. Из чего состоит правило преобразования?
6. Как сделать преобразование только одного заданного узла?
7. Как выделить для последующего преобразования все узлы с одинаковым содержанием?
8. Какими способами можно перебрать все узлы дерева документа?
9. Какими способами можно перебрать только узлы-элементы?
10. Преобразуются ли узлы, для которых не написаны правила преобразования?
11. Можно ли применить несколько правил преобразования к одному узлу?
12. Каким образом подбирается наиболее подходящее правило преобразования для данного узла?
13. В каком порядке обрабатываются узлы, выделенные для преобразования?
14. Как можно изменить порядок обработки предназначенных для преобразования узлов?
15. Как задается ключ сортировки узлов перед их преобразованием?

ГЛАВА 9



Форматирование объектов XSL-FO

Мы уже обсуждали процесс форматирования в начале предыдущей главы. Напомню, что форматирование (formatting) применяется к дереву, полученному в результате преобразования исходного документа XML средствами языка XSLT. Цель форматирования — подготовить преобразованный документ к выводу на конечное устройство: экран дисплея, принтер, пейджер, голосовое устройство, или записать в файл в определенном формате: PDF, PCL, SVG. Для этого надо разбить документ на страницы, задать поля, заголовки, подобрать стили абзацев, выбрать шрифт и цвет, разместить рисунки и чертежи. Все это и входит в задачу форматирования документа. Форматирование выполняется программой, которая так и называется *форматером* (formatter). Форматер может быть отдельным приложением или располагаться в устройстве вывода, например, в браузере или принтере.

Теоретически, в результате форматирования опять получается дерево, но уже не дерево узлов-элементов и узлов-атрибутов, а *дерево геометрических областей* (area tree). Каждый формируемый элемент занимает некоторую геометрическую область (region), чаще всего это прямоугольник. Скажем, один символ будет занимать область, достаточную для размещения его начертания. Области могут вкладываться друг в друга, например, область символа вкладывается в область строки, та в область абзаца и т. д.

Каждая геометрическая область помещается на *страницу* (page) в определенную *позицию* (position). Окно браузера обычно содержит одну большую страницу. На печать, как правило, выводится много небольших страниц.

Дерево геометрических областей строится в несколько этапов. На первом этапе каждый узел-элемент дерева элементов и их атрибутов, полученного после преобразования средствами XSLT, представляется *объектом форматирования* (formatting object), а каждый узел-атрибут — набором *свойств* (property specifications). После этого создается *дерево объектов форматирования* (formatting object tree), состоящее из объектов и наборов их свойств.

На втором этапе дерево объектов форматирования *очищается* (refine) путем расшифровки всех сокращений, вычисления всех выражений, реализации наследования. При этом свойства объектов переходят в так называемые *качества* (traits).

На третьем этапе строится дерево геометрических областей по указаниям, содержащимся в каждом объекте очищенного дерева.

Узлы дерева геометрических областей, полученные из объектов форматирования, называются *зонами* (areas). Каждый объект форматирования создает одну или несколько зон, а каждая зона формирует прямоугольную область. Область состоит из трех вложенных друг в друга прямоугольников. Внешний прямоугольник образует границу (border). В него вложен промежуточный прямоугольник (padding), а уж в него — прямоугольник с содержимым области (content). Граница и промежуточная область могут отсутствовать, зона может формировать только область с содержимым.

Зоны бывают двух типов: *блочные зоны* (block-areas) и *встроенные зоны* (inline-areas). Типичным примером блочной зоны служит абзац, примером встроенной зоны — отдельный символ. Встроенные зоны распределяются в линию в порядке, предписываемом правилами "человеческого" языка документа. По умолчанию это направление слева направо. Для арабского языка или иврита это направление справа налево. Блочные зоны также располагаются одна за другой в порядке, устанавливаемом языком, по умолчанию, сверху вниз.

Все эти теоретические построения деревьев нужны для тех, кто разрабатывает формтеры. Тем, кому надо сформатировать документ XML при помощи готового формтера, достаточно знать принципы форматирования и терминологию, а также уметь использовать элементы, объявленные языком XSL. Как вы увидите далее, создание документа XSL очень похоже на создание документа HTML с использованием стилей CSS2 внутри тегов, как мы это делали в начале *главы 8*.

Язык XSL

Язык форматирования XSL — это одна из реализаций XML. Как уже говорилось в начале *главы 8*, сначала язык XSL объединял и язык преобразований XSLT, и язык адресации XPath, и язык форматирования объектов FO. Затем первые два языка стали самостоятельными. Сейчас в язык XSL входят только средства форматирования, поэтому его часто называют "XSL-FO". Мы в этой главе будем использовать официальное название языка "XSL". Рекомендация языка XSL расположена по адресу <http://www.w3.org/TR/xsl/>.

Язык XSL объявляет более пятидесяти элементов XML. Имена этих элементов находятся в пространстве имен <http://www.w3.org/1999/XSL/Format>, ко-

торому обычно дают префикс `fo`. Эти элементы как раз и образуют классы объектов форматирования.

Корневым элементом документа XSL служит элемент `fo:root`. В него вкладывается один или несколько элементов `fo:page-sequence`, описывающих весь набор страниц в целом, и ровно один элемент `fo:layout-master-set`, содержащий подробные описания страниц во вложенных в него элементах. Кроме этих элементов непосредственно в корневой элемент можно вложить только элемент `fo:declaration`, содержащий информацию для форматера.

Элемент `fo:page-sequence` содержит необязательный заголовок — вложенный элемент `fo:title`, один обязательный вложенный элемент `fo:flow`, задающий динамически формируемую выводимую информацию, и сколько угодно необязательных элементов `fo:static-content`, содержащих определенный неизменный текст, например, колонтитулы.

Элемент `fo:layout-master-set` содержит элементы `fo:simple-page-master`, описывающие строение страницы, и/или элементы `fo:page-sequence-master`, описывающие порядок применения к страницам правил форматирования. Эти элементы необязательны, но хотя бы один из них должен присутствовать.

Итак, простейший FO-документ, формирующий вывод бессмертной фразы "Hello, World!", выглядит примерно так, как показано в листинге 9.1.

Листинг 9.1. Простейшие правила форматирования

```
<?xml version="1.0" encoding="windows-1251"?>

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

  <fo:layout-master-set>

    <fo:simple-page-master master-name="mypage"
      page-height="29.7cm"
      page-width="21cm"
      margin-top="2.5cm"
      margin-bottom="2.5cm"
      margin-left="3cm"
      margin-right="1.5cm">

      <fo:region-body margin="1cm" />

    </fo:simple-page-master>

  </fo:layout-master-set>
```

```
<fo:page-sequence master-reference="mypage">

  <fo:flow flow-name="xsl-region-body">

    <fo:block
      font-family="serif"
      font-size="3em"
      font-style="italic"
      font-weight="bold"
      color="red"
      background-color="blue">

      Hello, World!

    </fo:block>

  </fo:flow>

</fo:page-sequence>

</fo:root>
```

Сам текст "Hello, World!", выводимый в листинге 9.1, заключен внутри элемента `fo:block`, формирующего блок текста. Атрибуты этого элемента, записанные в листинге, определяют свойства шрифта, которым будет выведен текст, а именно выбран шрифт с засечками (`font-family="serif"`), величиной в три раза большей, чем величина обычного шрифта (`font-size="3em"`). Текст должен выводиться курсивом (`font-style="italic"`) красного цвета (`color="red"`) на синем фоне (`background-color="blue"`). Кроме этих атрибутов в элементе `fo:block` можно записать массу других, например, задать свойства пограничной и промежуточной областей, и даже правила произношения текста для голосового устройства.

Элемент `fo:block` находится внутри элемента `fo:flow`, формирующего выходной поток. Атрибут `flow-name` элемента `fo:flow` направляет поток в область `xsl-region-body`. Это центральная область страницы. Другие значения этого атрибута направляют поток в другие области страницы: `xsl-region-before`, `xsl-region-after`, `xsl-region-start`, `xsl-region-end` или в область, созданную оформителем.

Весь выходной поток находится в теле элемента `fo:page-sequence`, описывающего набор страниц, оформленных в одном стиле. Это может быть глава книги или вся книга, оглавление или индекс. Кроме элемента `fo:flow`, содержащего динамическое наполнение страниц, в элемент `fo:page-sequence` можно вложить элементы `fo:static-content`, имеющие постоянное наполнение, повторяющееся на каждой странице. Элемент `fo:page-sequence` атрибутом `master-reference` ссылается на элемент `fo:simple-page-master`.

Элемент `fo:simple-page-master` своими атрибутами формирует параметры страницы, на которую будет помещен текст, а именно длину (`page-height`) и ширину (`page-width`) страницы, верхнее (`margin-top`), нижнее (`margin-bottom`), левое (`margin-left`) и правое (`margin-right`) поля. В него вложен элемент `fo:region-body`, формирующий область вывода текста. Это центральная часть страницы, в которой сделаны дополнительные поля с одинаковой шириной 1 см по всему периметру, на что указывает атрибут `margin`.

Многие элементы языка XSL похожи на стили языка CSS второй версии, о котором мы вкратце говорили в начале *главы 8*. Это не случайно — при разработке языка XSL было принято сознательное решение следовать разработкам, сделанным в CSS2. В частности, многие атрибуты элементов XSL просто являются копиями свойств стилей CSS2, в чем можно убедиться на примере листинга 9.1. Поэтому атрибуты элементов XSL и называются *свойствами* (`properties`). Но таблицы стилей CSS2 задают только стиль отображения тегов HTML или элементов XML в окне браузера, а элементы XSL полностью описывают все необходимое для представления документа XML в устройстве вывода.

Все это привело к появлению огромного количества свойств — атрибутов элементов XSL. Их набралось около двухсот пятидесяти. Даже при простейшем форматировании в листинге 9.1 одной страницы, на которой написаны два слова, мы использовали двенадцать свойств. Такое разнообразие свойств позволяет очень гибко форматировать самые сложные страницы, но значительно усложняет FO-документы и затрудняет изучение языка XSL.

Мы не будем подробно изучать все 56 элементов языка XSL и 250 их свойств. Вместо этого посмотрим, как на языке XSL решаются основные задачи форматирования. Начнем с построения "кирпичиков", из которых складывается отформатированный документ.

Единицы измерения

В листинге 9.1 мы использовали единицу измерения `"em"`. Она взята из языка CSS. `"1em"` представляет собой средний размер текущего шрифта. Эта единица удобна для увеличения или уменьшения размера некоторых участков текста.

Еще одна такая же относительная единица — `"ex"` — это размер малой латинской буквы `"x"`.

Кроме этих относительных единиц измерения, можно использовать абсолютные: типографский пункт `"pt"`, например `font-size="14pt"`, и "пику" `"pc"`, равную 12 типографским пунктам. Типографский пункт равен 1/72 английского дюйма, т. е. 0,353 мм.

Для форматирования графических элементов удобны пиксели, записываемые сокращением "px", например, `page-width="1024px"`.

Конечно, можно использовать и метрические единицы измерения: миллиметры "mm" и сантиметры "cm", как мы это делали в листинге 9.1. Кроме них можно применять дюймы, обозначаемые сокращением "in". Один дюйм равен 25,4 мм.

Цвет

Как и в языке HTML 4.0, в XSL для атрибутов, определяющих цвет, есть шестнадцать обозначений цвета: "aqua", "black", "blue", "fuchsia", "gray", "green", "lime", "maroon", "navy", "olive", "purple", "red", "silver", "teal", "white" и "yellow".

Кроме того, можно задать цвет в системе RGB (Red-Green-Blue, "красный-зеленый-синий"), в которой каждая составляющая принимает десятичные значения от 0 до 255. Эти значения можно записать в шестнадцатеричной форме от 00 до FF, а можно — в процентах от 0% до 100%. При этом шестнадцатеричные значения указываются как в HTML, после "решетки", например, #FF00BB, а десятичные значения и проценты записываются как аргументы функции `rgb()`, например, `rgb(255, 0, 204)`, `rgb(100%, 0%, 80%)`. Здесь сначала следует красная составляющая цвета, затем зеленая, а последний аргумент показывает синюю составляющую цвета.

Наконец, цвет можно определить в системе ICC (International Color Consortium) с помощью функции

```
rgb-icc(red, green, blue, name, number, number)
```

Первые три аргумента функции `rgb-icc()` задают модель цвета RGB на тот случай, если профиль ICC не будет обнаружен. Четвертый аргумент `name` указывает имя профиля цвета, пятый и шестой аргументы задают параметры профиля цвета ICC. Профиль цвета должен быть предварительно определен элементом `fo:color-profile`.

Пустой элемент `fo:color-profile` ссылается на профиль цвета ICC, задавая его адрес URI в атрибуте `src`, и дает ему имя в атрибуте `color-profile-name`. Это имя указывается затем в четвертом аргументе функции `rgb-icc()`. Третий атрибут `rendering-intent`, принимающий значения "auto", "perceptual", "relative-colorimetric", "saturation", "absolute-colorimetric", указывает, каким образом использовать профиль в системе CMYK (cyan-magenta-yellow-black, голубой-сиреневый-желтый-черный) при печати.

Форматирование блока

Под блоком (block) в языке XSL понимается какой-либо выделенный участок документа: абзац, таблица, список, заголовок, подпись. В языке HTML в таких случаях применяется тег `<div>`. Блок описывается элементом `fo:block`. Текст записывается в теле элемента, а атрибуты элемента определяют правила форматирования блока. В теле можно записать другие элементы `fo:block`, получив, таким образом, вложенные блоки. Кроме того, в теле элемента `fo:block` можно записать элементы `fo:table`, `fo:table-and-caption`, `fo:block-container`, `fo:list-block`, `fo:character`, `fo:external-graphics`, `fo:leader` и др.

Таким образом, структура блока может быть довольно сложной, блок может включать в себя таблицы, списки, изображения, другие блоки со своими вложенными блоками. Более того, у блока может быть граница, промежуточная область, его можно окружить полями. Мы опишем эти свойства ниже, вместе с подобными свойствами страницы, а пока введем еще две разновидности атрибутов.

Стенографические свойства

В листинге 9.1 мы уже задавали шрифт и цвета для блока, используя с этой целью отдельные атрибуты. Иногда несколько параметров форматирования можно записать в одном атрибуте. Например, блок листинга 9.1 можно сформатировать и так:

```
<fo:block font="italic bold 3em serif"
          color="red" background-color="blue">
```

Такие совмещенные атрибуты, как атрибут `font`, называются *стенографическими свойствами* (shorthand properties). В языке XSL их более двадцати. Они очень часто используются для сокращения записи.

Составные атрибуты

Вторую разновидность сложных атрибутов образуют *составные* (compound) атрибуты, задающие сразу несколько свойств. Каждое свойство записывается отдельно, а к атрибуту через точку приписывается суффикс, показывающий, какое именно свойство определяется. К составным относятся, например, атрибуты, задающие расстояние между блоками.

Свободное пространство перед блоком определяется составным атрибутом `space-before`, после блока — `space-after`. Оба атрибута определяют по пять свойств, например:

```
<fo:block
  space-before.minimum="10pt"
  space-before.maximum="20pt"
  space-before.optimum="15pt"
  space-before.conditionality="retain"
  space-before.precedence="force"

  space-after.minimum="5pt"
  space-after.maximum="5pt"
  space-after.optimum="5pt"
  space-after.conditionality="discard"
  space-after.precedence="0">
```

Минимальное расстояние перед блоком равно 10 пунктам,
предпочтительное — 15 пунктам, максимальное — 20 пунктам.

```
</fo:block>
```

Как видно из этого примера, у каждого расстояния есть минимальное, предпочтительное и максимальное значение. Разумеется, минимальное значение должно быть меньше максимального, а предпочтительное значение должно лежать между ними. По умолчанию все расстояния нулевые.

Атрибут с суффиксом `conditionality` может принимать всего два значения: `"retain"` или `"discard"`. Значение по умолчанию — `"discard"`. Оно означает, что расстояние `space-before` не будет учитываться для первого блока на странице, а расстояние `space-after` — для последнего блока на странице.

Атрибуты `space-before` и `space-after` допустимо записывать и без суффикса, указав только общее значение всех трех расстояний. В предыдущем примере можно было записать просто `space-after="5pt"`.

Расстояние между блоками

Расстояние `space-after` одного блока и расстояние `space-before` следующего блока — это одно и то же расстояние. Форматер вычисляет расстояние между блоками, основываясь на значениях атрибутов с суффиксом `precedence` соседних блоков. Значениями могут быть неотрицательные целые числа или константа `"force"`. Значение по умолчанию `"0"`.

Правила вычисления расстояния между блоками таковы:

1. Форматер выбирает из двух расстояний то, у которого значение `precedence` равно `"force"`.
2. Если оба значения равны `"force"`, то форматер суммирует минимальные, оптимальные и максимальные расстояния, заданные атрибутами `space-after` и `space-before`.

3. В противном случае выбирается расстояние с наибольшим значением `preference`.
4. Если значения `preference` совпадают, то выбирается расстояние с наибольшим значением `optimum`.
5. Если и значения `optimum` совпадают, тогда выбирается большее из двух значений `minimum` и меньшее из двух значений `maximum`.

Например, пусть у двух следующих друг за другом блоков заданы такие атрибуты:

```
<fo:block space-after.minimum="3pt">
    space-after.maximum="24pt"
    space-after.optimum="12pt">
    Какое-то содержимое...
</fo:block>

<fo:block space-before.minimum="6pt"
    space-before.maximum="18pt"
    space-before.optimum="12pt">
....Какое-то содержимое...
</fo:block>
```

Форматер выберет минимальное расстояние между блоками, равное 6pt, предпочтительное расстояние 12pt и наибольшее расстояние 18pt.

Форматирование абзаца

Чаще всего блок используется для форматирования абзаца. При этом нужно задать расстояние между строками, красную строку, отступ слева и/или справа, выравнивание текста по левому или правому краю, обработку висячих строк. Все это делают атрибуты элемента `fo:block`.

Расстояние между строками (интерлиньяж, `leading`) входит в характеристики шрифта и определяется атрибутом `font-height`. Расстояние можно увеличить с помощью атрибута `fo:line-height`, который задает минимальную высоту строки, равную расстоянию между базовыми линиями соседних строк. Значение этого атрибута можно записать различными единицами, например, `fo:line-height="18pt"`. Если единица измерения не указана, например, `fo:line-height="1.2"`, то значение атрибута понимается как множитель высоты шрифта, т. е. значения атрибута `font-size`. Тот же смысл приобретает значение, записанное процентами, например, `fo:line-height="120%"`. Наконец, в качестве значения атрибута можно записать слово `"normal"`, возложив обязанность определения высоты строки на форматер.

Красная строка отмечается атрибутом `text-indent`. Его значение можно задать абсолютной величиной, например `text-indent="2.5cm"`, или в процентах к ширине блока, например `text-indent="8%"`.

Иногда надо сделать один абзац уже или шире других абзацев. Этого можно достигнуть разными способами, но проще всего воспользоваться атрибутами `start-indent` и `end-indent`. Они задают отступ слева и справа соответственно в любых единицах измерения, в том числе и в процентах к ширине блока.

Способ выравнивания текста определяется атрибутом `text-align`. Его значения для языков, в которых запись идет слева направо, таковы:

- `"left"` или `"start"` — текст прижимается к левому краю блока;
- `"center"` — текст собирается к середине блока;
- `"right"` или `"end"` — текст прижимается к правому краю блока;
- `"justify"` — текст распространяется на всю ширину блока с помощью дополнительных пробелов;
- `"inside"` — совпадает со значением `"start"`, если страница будет расположена справа на развороте книги (нечетная страница), или со значением `"end"`, если страница будет расположена слева на развороте (четная страница);
- `"outside"` — совпадает со значением `"end"` для нечетной страницы и со значением `"start"` для четной страницы.

При печати абзац часто разрывается между двумя страницами. Атрибут `widows` задает минимальное количество строк, которые можно перенести на следующую страницу (висячие строки). Атрибут `orphans` задает минимальное количество строк, которые можно оставить внизу страницы.

Приведем пример оформления абзаца с красной строкой, выравниванием по всей ширине абзаца и обработкой висячих строк:

```
<fo:block
  line-height="1.5"
  text-align="justify"
  text-indent="25mm"
  widows="2"
  orphans="2"
  start-indent="2cm"
  end-indent="2cm">
```

Абзац уже других абзацев на 2 см, у него выделена красная строка. Текст выравнивается по всей ширине абзаца, висячие строки не допускаются.

```
</fo:block>
```

Упражнения

1. Сформатируйте абзац без красной строки, с расстоянием между абзацами в два раза большим, чем расстояние между строками.
2. Сформатируйте абзац, неразрывный между страницами.

Форматирование текста

Часто возникает необходимость выделить некоторый участок текста или даже одну букву курсивом, подчеркиванием, другим цветом, изменить размер шрифта. Для форматирования текста в языке XSL есть очень мощный элемент `fo:inline` с огромным количеством атрибутов, позволяющих, кроме указанных действий, даже задать рамку и поля для выделяемого текста. В частности, в элементе `fo:inline` можно записывать все атрибуты элемента `fo:block`.

Следующий пример показывает применение некоторых атрибутов элемента `fo:inline`:

```
<fo:block line-height="1.5" text-align="justify">

  <fo:inline font-style="italic">
    Этот текст будет написан курсивом.
  </fo:inline>

  <fo:inline color="red" background-color="blue">
    Здесь будет красный текст на синем фоне.
  </fo:inline>

  <fo:inline-container text-decoration="underline">
    Этот текст будет подчеркнут.
  </fo:inline-container>

  <fo:inline letter-spacing="2mm">
    Расстояние между буквами этого текста будет 2 мм.
  </fo:inline>

  <fo:inline word-spacing="1cm">
    Расстояние между словами этого текста будет 1 см.
  </fo:inline>

</fo:block>
```

Таким же образом можно отформатировать отдельные символы, но для этого удобнее применять специальный элемент `fo:character`. У него есть все атрибуты элемента `fo:inline`. В отличие от элемента `fo:inline`, элемент

`fo:character` пустой, символ, который он форматирует, записывается в атрибуте `character`.

Упражнение

Сформируйте отдельно первую заглавную букву каждого предложения.

Вставка изображения

В любой блок или в строку можно вставить изображение, взятое из файла или другого ресурса Интернета. Это делается элементом `fo:external-graphic`, в атрибуте `src` которого указывается адрес URI файла. Конструкция похожа на тег `` языка HTML. Например:

```
<fo:block>

  <fo:external-graphic src="images/myface.jpg"/>

  Следующее изображение вставляется

  <fo:inline alignment-baseline="before-edge">
    <fo:external-graphic src="images/l.gif"/>
  </fo:inline>

  прямо в текст.

</fo:block>
```

Размеры изображения можно отрегулировать атрибутами `content-height` и `content-width`.

Упражнение

Вставьте изображение так, чтобы оно занимало левую половину блока, а текст располагался справа.

Горизонтальные линии

Горизонтальные линии разной длины и формы создаются элементом `fo:leader`. Он допускает создание границы, полей и промежуточной области, в него можно вложить элементы `fo:inline`, `fo:character`, `fo:external-graphic`. Пользуясь этим, можно нарисовать очень сложную линию. Но чаще всего этот элемент пуст, все характеристики линии задаются его атрибутами.

Длина линии в метрических единицах или в процентах к ширине блока определяется атрибутом `leader-length`, цвет линии — атрибутом `color`, форма

линии — атрибутом `leader-pattern`. Он принимает одно из следующих значений:

- `"space"` — линия состоит из пробелов;
- `"rule"` — стиль линии задается атрибутами `rule-thickness` и `rule-style`;
- `"dots"` — линия состоит из точек;
- `"use-content"` — стиль линии определяется вложенными элементами.

Атрибут `rule-thickness` задает толщину линии в метрических единицах, атрибут `rule-style` принимает следующие значения:

- `"none"` — линия отсутствует, т. е. ее толщина равна 0;
- `"dotted"` — пунктир;
- `"dashed"` — штрихи;
- `"solid"` — сплошная линия;
- `"double"` — сплошная двойная линия;
- `"groove"` — "вдавленная" линия;
- `"ridge"` — "выпуклая" линия.

Размещение линии в блоке регулируется атрибутом `alignment-baseline`, принимающим следующие значения:

- `"auto"` — положение линии в блоке определяется форматером;
- `"baseline"` — линия проводится по базовой линии шрифта;
- `"before-edge"` — линия проходит по верху букв, не выделяющихся из строки;
- `"text-before-edge"` — линия проходит выше строки, по верху таких букв, как "б";
- `"middle"` — линия проходит посередине между `before-edge` и `after-edge`;
- `"central"` — линия проходит посередине высоты строки, определяемой единицей измерения `"em"`;
- `"after-edge"` — линия проходит по низу букв, не выделяющихся из строки;
- `"text-after-edge"` — линия проходит ниже строки, по низу таких букв, как "у";
- `"ideographic"` — линия проходит на расстоянии $7/10$ от `before-edge` до `after-edge`, это базовая линия для иероглифов;
- `"alphabetic"` — линия проходит на расстоянии $3/5$ от `before-edge` до `after-edge`;
- `"hanging"` — линия проходит на расстоянии $1/5$ от `before-edge` до `after-edge`;

- "mathematical" — базовая линия для математических символов, обычно проходит посередине между before-edge и after-edge.

Вот, например, строка из оглавления с горизонтальной чертой, проведенной точками:

```
<fo:block text-align-last="justify">

  Глава 5

  <fo:leader leader-pattern="dots" leader-length="60%"
    alignment-baseline="middle"
    rule-thickness="0.5pt" color="black" />

  125

</fo:block>
```

После форматирования эта строка должна выглядеть так:

Глава 5125

Упражнение

Проведите горизонтальную черту, состоящую из трех тонких линий.

Форматирование страницы

Самое простое форматирование страницы выполняет элемент `fo:simple-page-master`. Размеры страницы задаются его атрибутами `page-height` и `page-width`. Кроме числовых значений, у них есть специальные значения "auto" и "indefinite". Если один или оба атрибута получили значение "auto", то соответствующий размер страницы будет определять устройство вывода. Значение "indefinite" нельзя давать сразу обоим атрибутам, его может получить только один из двух атрибутов. В таком случае длина или ширина страницы определяется количеством ее содержимого. При этом выделяется *видимая зона страницы* (page viewport area), а атрибут `overflow` элемента `fo:simple-page-master` должен получить значение "scroll", чтобы видимая зона могла перемещаться по экрану дисплея или другому устройству вывода.

Атрибут `reference-orientation` задает ориентацию страницы. По умолчанию его значение "0". Кроме нуля есть только шесть значений этого атрибута. Строки "90", "180", "270" означают поворот страницы по часовой стрелке на указанное число градусов. Отрицательные значения "-90", "-180", "-270" задают поворот против часовой стрелки и эквивалентны значениям "270", "180", "90" соответственно.

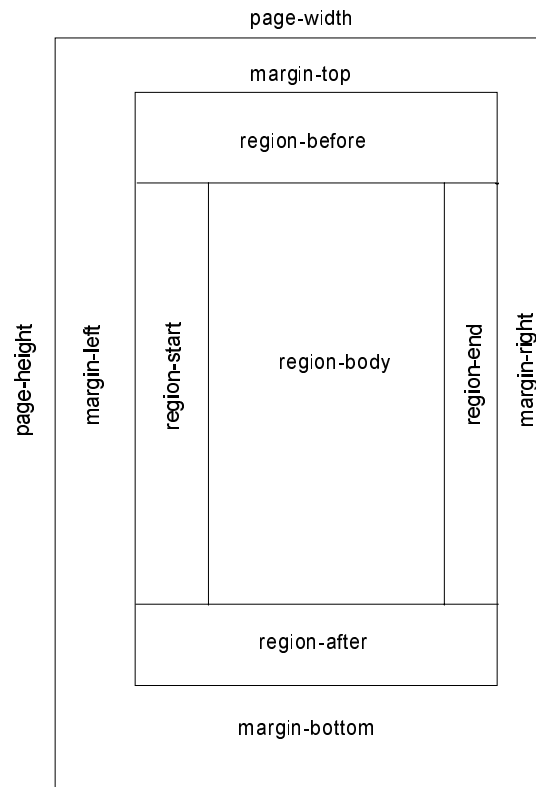


Рис. 9.1. Страница XSL

У страницы могут быть поля, задаваемые атрибутами `margin-top`, `margin-bottom`, `margin-left`, `margin-right`, как записано в листинге 9.1 и показано на рис. 9.1. Кроме абсолютных величин полей, можно указать значение "auto". Тогда размеры полей будут вычисляться по правилам, установленным в языке стилей CSS2. Кроме того, можно задать размеры в процентах к общей длине или ширине страницы, например, `margin-top="10%"`.

Вместо того чтобы записывать четыре атрибута для полей, можно воспользоваться стенографическим свойством `margin`. В атрибуте `margin` можно записать одно число, например, `margin="1cm"`. Оно устанавливает одинаковый размер всех четырех полей.

В записи атрибута с двумя числами, например `margin="1.5cm 2.5cm"`, первое число показывает размер верхнего и нижнего поля, а второе — размер правого и левого поля.

Запись с тремя числами, `margin="1.5cm 2cm 2.5cm"`, дает размеры верхнего, боковых и нижнего полей. Боковые поля будут одинаковыми, в примере они равны 2cm.

Наконец, запись с четырьмя числами, `margin="2.5cm 1.5cm 2cm 2.5cm"`, определяет по часовой стрелке размеры верхнего, правого, нижнего и левого полей.

Точно так же можно задать поля для блока, записав такие же атрибуты элемента `fo:block`.

Элемент `fo:simple-page-master` делит оставшуюся после выделения полей и промежуточной области страницу на пять областей: центральную область `fo:region-body`, верхнюю — `fo:region-before`, нижнюю — `fo:region-after`, левую — `fo:region-start` и правую — `fo:region-end`. На рис. 9.1 указано их расположение для языков с записью слева направо, но оно зависит от порядка записи текста в языке. Для языков с записью текста справа налево, например арабского, область `fo:region-start` окажется справа, а область `fo:region-end` — слева. На каждой странице обязана присутствовать центральная область, остальные области необязательны.

Эти области аналогичны фреймам языка HTML. Каждую область можно отформатировать независимо от других областей, в каждую из них можно вывести свой текст.

Размер каждой области зависит от размера страницы и размера полей. Он может оказаться бесконечным в каком-то направлении, чаще всего по длине. В таком случае выделяется *видимая зона области* (region viewport area). Возможность ее прокрутки следует задать значением `"scroll"` атрибута `overflow`, присутствующего в каждом элементе `fo:region-*`.

Каждая из пяти областей описывается одноименным элементом языка XSL, вложенным в элемент `fo:simple-page-master`. Все пять элементов пустые, они описывают области только своими атрибутами.

Центральная область, описываемая элементом `fo:region-body`, отличается от прочих областей тем, что в ней тоже могут быть поля, задаваемые теми же атрибутами `margin-top`, `margin-bottom`, `margin-left`, `margin-right`, или стенографическим атрибутом `margin`, как показано в листинге 9.1. У каждого из этих атрибутов есть дополнительное значение `"inherit"`, означающее, что значение атрибута такое же, как и у страницы, в которой лежит область.

Второе отличие центральной области от прочих заключается в том, что текст в ней можно записать в несколько колонок. Для этого надо указать число колонок атрибутом `column-count`. Расстояние между колонками будет равно 12pt, но его можно изменить атрибутом `column-gap`.

Наиболее важны атрибуты, определяющие границу и промежуточную область. Эти атрибуты одинаковы для страницы и для каждой из ее областей, а также и для блоков. У атрибутов, описывающих область и блок, всегда есть дополнительное значение `"inherit"`, означающее наследование данного свойства от соответствующего свойства страницы, в которой лежит область, а также от области, в которой расположен блок.

Граница

Атрибуты, описывающие границу (border), одинаковы для страницы в целом и для каждой из ее областей. При описании границы можно установить ее толщину, цвет и стиль линии. Чаще всего граница служит рамкой для текста или изображения, и ее толщина бывает небольшой.

Четыре атрибута: border-before-width, border-start-width, border-end-width и border-after-width — содержат толщину верхней, левой, правой и нижней стороны границы соответственно. Значением этих атрибутов, кроме числа, может быть одна из трех констант:

- "thin" — тонкая линия;
- "medium" — линия средней толщины;
- "thick" — толстая линия.

Например:

```
<fo:region-body
  border-before-width="5px"
  border-start-width="thin"
  border-end-width="medium"
  border-after-width="thick" />
```

Цвет каждой стороны границы по отдельности можно задать атрибутами border-before-color, border-start-color, border-end-color и border-after-color. Например:

```
<fo:region-body
  border-before-color="yellow"
  border-start-color="inherit"
  border-end-color="#00008F"
  border-after-color="rgb(0, 0, 128)" />
```

Стиль линии задается атрибутами border-before-style, border-start-style, border-end-style и border-after-style. Эти атрибуты принимают следующие значения:

- "none" — линия отсутствует;
- "hidden" — граница скрыта под другой границей;
- "dotted" — пунктир;
- "dashed" — штрихи;
- "solid" — сплошная линия;
- "double" — сплошная двойная линия;
- "groove" — вдавленная линия;

- "ridge" — выпуклая линия;
- "inset" — линия такова, что вся область выглядит вдавленной;
- "outset" — линия такова, что вся область выглядит выпуклой.

Перечисленные атрибуты объявлены в языке XSL. Они учитывают правила записи текста (writing mode), принятые в разных "человеческих" языках: запись слева направо или справа налево, или сверху вниз. Эти правила задаются атрибутом writing-mode, принимающим следующие значения:

- "lr-tb" или просто "lr" — текст пишется слева направо, блоки текста идут сверху вниз;
- "rl-tb" или просто "rl" — текст пишется справа налево, блоки текста идут сверху вниз;
- "tb-rl" или просто "tb" — текст пишется сверху вниз, блоки идут справа налево.

Эквивалентные атрибуты языка CSS2, у которых вместо слов before, start, end, after стоят слова top, left, right, bottom, не учитывают особенности языка. Атрибуты CSS2 border-top-width, border-left-width, border-right-width, border-bottom-width, border-top-color и т. д. тоже можно использовать в языке XSL.

Все три свойства каждой стороны границы можно задать одним стенографическим свойством: border-top, border-left, border-right или border-bottom. Например:

```
<fo:region-body
  border-top="dashed yellow 1pt"
  border-left="blue dotted 1pt"
  border-right="1pt #00008F solid"
  border-bottom="3pt rgb(0, 0, 128) double" />
```

Как видите, свойства можно перечислять в любом порядке.

С другой стороны, можно задать определенное свойство для каждой стороны границы стенографическими свойствами border-width, border-color или border-style. Например:

```
<fo:block
  border-width="5px 3px 2px 5px"
  border-color="blue yellow"
  border-style="inherit">

  Текст, окруженный рамкой.

</fo:block>
```

Стенографические свойства могут содержать одно, два, три или четыре значения через пробелы. Одно значение распространяется на все стороны границы. Если заданы два значения, то первое из них применяется к верхней и нижней стороне, второе — к боковым сторонам. Три значения — это "верх-бока-низ". Четыре значения применяются по часовой стрелке — "верх-право-низ-лево".

При выводе на печать область может быть расположена на нескольких страницах. При этом возникает вопрос: замыкать границу на каждой странице (поведение XSL по умолчанию) или проводить ее только на первой и последней странице. В последнем случае используется добавление `conditionality` и значение атрибута `"discard"`, например:

```
<fo:region-body border="thin blue groove"
  border-before-width.conditionality="discard"
  border-after-width.conditionality="discard" />
```

Упражнение

Обведите страницу рамкой, образованной двойной линией.

Промежуточная область

Промежуточная область (`padding`) характеризуется только толщиной каждой своей стороны. Толщину задают атрибуты `padding-before`, `padding-start`, `padding-end`, `padding-after`. Они дублируются атрибутами языка CSS2 `padding-top`, `padding-left`, `padding-right`, `padding-bottom`, не учитывающими направление записи текста: слева направо, справа налево или сверху вниз. Например:

```
<fo:region-body
  padding-top="10pt"
  padding-left="5pt"
  padding-right="3pt"
  padding-bottom="10pt" />
```

Задать толщину всех сторон промежуточной области сразу можно стенографическим свойством `padding`, например:

```
<fo:region-body padding="10pt 3pt 5pt 10pt" />
```

Как всегда, в этом атрибуте можно записать одно, два, три или четыре значения. Одно значение применяется сразу ко всем сторонам промежуточной области, два значения — отдельно к "верху-низу" и к боковым сторонам. Три значения — это размеры верхней стороны, обеих боковых сторон и нижней стороны. Четыре значения применяются по часовой стрелке к верхней, правой, нижней и левой стороне.

Сноски

Сноски, располагаемые внизу страницы, создаются элементом `fo:footnote`. Он помещается в то место страницы, где делается ссылка на сноску, отмечаемую обычно надстрочным номером или звездочкой. Положение самой сноски определяется форматером, обычно он помещает сноску в область `region-after`.

В элемент `fo:footnote` вкладывается один элемент `fo:inline`, показывающий вид ссылки (звездочка, номер и т. д.), и один элемент `fo:footnote-body`, в котором записывается сама сноска. Например:

```
<fo:footnote>

  <fo:inline font-size="smaller" vertical-align="super">
    *
  </fo:inline>

  <fo:footnote-body font-size="smaller">

    <fo:inline font-size="smaller" vertical-align="super">
      *
    </fo:inline>

    Здесь записывается сноска.

  </fo:footnote-body>
</fo:footnote>
```

Колонтитулы, номера страниц и другое оформление

Полное оформление страницы включает в себя колонтитулы, номер текущей страницы, другие элементы оформления и украшения, повторяющиеся на каждой странице. Колонтитулы и сноски часто отделяются от текста горизонтальной чертой. Все эти постоянные составляющие страницы описываются элементом `fo:static-content`, вкладываемым в элемент `page-sequence`.

У элемента `fo:static-content` только один атрибут `flow-name`, направляющий постоянные составляющие страницы в определенную область: `xsl-region-body`, `xsl-region-before`, `xsl-region-start`, `xsl-region-end`, `xsl-region-after`, `xsl-footnote-separator` или в другую область страницы. Сами составляющие записываются в элементах `fo:block`, вложенных в `fo:static-content`.

Здесь часто используется элемент `fo:page-number`, выводящий номер текущей страницы. У него множество атрибутов, превращающих номер страницы в полноценный блок с границей, промежуточной областью, полями, возможностью выбора шрифта и цвета. Но чаще всего используются значения атрибутов по умолчанию. Например:

```
<fo:page-sequence
  master-reference="p1"
  initial-page-number="1"
  language="ru"
  country="ru">

  <fo:static-content flow-name="xsl-region-before">

    <fo:block text-align="end">
      Страница <fo:page-number />
    </fo:block>

  </fo:static-content>

  <fo:static-content flow-name="xsl-footnote-separator">

    <fo:block>
      <fo:leader leader-pattern="rule"
        leader-length="100%"
        rule-style="solid"
        rule-thickness="0.5pt"/>
    </fo:block>

  </fo:static-content>

</fo:page-sequence>
```

Номер первой страницы можно задать явно атрибутом `initial-page-number` элемента `fo:page-sequence`. Кроме числа, этому атрибуту можно дать одно из трех следующих значений:

- ☐ "auto" — продолжение нумерации предыдущего элемента `fo:page-sequence` или 1;
- ☐ "auto-odd" — нумерация начинается с нечетного числа; четное число увеличивается на 1;
- ☐ "auto-even" — нумерация начинается с четного числа; нечетное число увеличивается на 1.

У книг номера нечетных страниц обычно пишутся справа, а номера четных страниц — слева. Сначала печатаются нечетные страницы книги, затем листы бумаги переворачиваются и на обороте печатаются четные страницы.

Учесть эти особенности помогает атрибут `force-page-count` элемента `fo:page-sequence`. Значения атрибута таковы:

- ☐ `"auto"` — последняя страница делается четной, если следующий набор страниц начинается с нечетной страницы, и наоборот, последняя страница делается нечетной, если следующий набор страниц начинается с четной страницы;
- ☐ `"even"` — на страницу выводится четный номер, затем добавляется пустая страница;
- ☐ `"odd"` — на страницу выводится нечетный номер, затем добавляется пустая страница;
- ☐ `"end-on-even"` — если последняя страница нечетная, то добавляется пустая страница;
- ☐ `"end-on-odd"` — если последняя страница четная, то добавляется пустая страница;
- ☐ `"no-force"` — не учитывает четность номера.

Упражнение

Запишите на языке XSL оформление этой страницы.

Фон

Для страницы целиком, для каждой из ее областей, а также для каждого блока можно определить свойства фона. Наиболее часто применяется свойство `background-color`, задающее цвет фона. Не менее часто на фон помещается изображение. Это можно сделать атрибутом `background-image`, значением которого служит строка URI. Цвет фона будет просвечивать сквозь прозрачные места изображения, поэтому атрибут `background-image`, как правило, используется вместе с атрибутом `background-color`.

Если на фоне расположено изображение, то можно задать его повторяемость. Это делается атрибутом `background-repeat`, принимающим следующие значения:

- ☐ `"repeat"` — изображение повторяется по вертикали и по горизонтали;
- ☐ `"repeat-x"` — изображение повторяется по горизонтали;
- ☐ `"repeat-y"` — изображение повторяется по вертикали;
- ☐ `"no-repeat"` — изображение выводится только один раз.

Расположение изображения на фоне можно уточнить атрибутами `background-position-horizontal` и `background-position-vertical`. Их значения показывают положение левого и верхнего угла изображения относительно левого

верхнего угла промежуточной области. Значение может быть числовым, можно задать его в процентах от размеров промежуточной области или записать одно из трех слов:

- "top" — изображение прижимается к левому или верхнему краю области;
- "center" — изображение располагается в центре области;
- "bottom" — изображение прижимается к правому или нижнему краю области.

Вместо двух атрибутов: `background-position-horizontal` и `background-position-vertical` — можно записывать стенографическое свойство `background-position`. В нем первое значение относится к горизонтали, а второе — к вертикали.

Наконец, атрибутом `background-attachment`, принимающим два значения: "scroll" или "fixed", — можно указать, будет изображение перемещаться вместе с текстом или останется неподвижным.

Приведем пример оформления блока с изображением на фоне.

```
<fo:block border="0.5pt solid silver"
  background-image="url('spots.jpg')"
  background-repeat="no-repeat"
  background-attachment="fixed"
  background-position-horizontal="center"
  background-position-vertical="center"
  padding="18pt"
  text-align="center">

  Одно изображение, взятое из файла spots.jpg,
  располагается по центру.
```

```
</fo:block>
```

Фон можно оформить одним стенографическим свойством `background`. С его использованием предыдущий пример будет выглядеть так:

```
<fo:block border="0.5pt solid silver"
  background="url('spots.jpg') no-repeat fixed center center"
  padding="18pt"
  text-align="center">

  Одно изображение, взятое из файла spots.jpg,
  располагается по центру.
```

```
</fo:block>
```

Упражнение

Поместите на фон в правый верхний угол страницы логотип своей фирмы.

Списки

Списки в языке XSL рассматриваются как разновидность блока. Они организуются элементом `fo:list-block`, в который вкладывается нужное число элементов `fo:list-item` — пунктов списка. Как у всякого блока, у списка может быть граница, промежуточная область, поля.

Каждый пункт списка содержит одну метку — элемент `fo:list-item-label` и одно тело — элемент `fo:list-item-body`. Метка — это то, что отмечает пункты списка: порядковый номер или буква, "горох", квадратик, галочка, но по синтаксису языка XSL в элемент `fo:list-item-label` можно вложить один или несколько любых блоков. Это значит, что в качестве метки пункта списка допускается все, что только можно сделать в блоке.

В языке XSL нет никаких специальных обозначений для наиболее распространенных графических символов, отмечающих пункты списка: квадратика, кружочка, диска, галочки. Они задаются в элементе `fo:block` своей кодировкой Unicode.

Расстояние от метки до пункта списка постоянно для всех пунктов. Оно задается атрибутом `provisional-label-separation` элемента `fo:list-block`. Еще один атрибут `provisional-distance-between-starts` определяет расстояние между началом метки и началом тела.

В тело пункта списка — элемент `fo:list-item-body` — тоже можно вложить один или несколько блоков, значит, в теле каждого пункта списка тоже может быть все, что можно записать в блоке, в том числе таблица или другой, вложенный список.

С учетом всего сказанного простейший список выглядит так, как показано в листинге 9.2. Символ с кодировкой Unicode `•`, использованный для пометки пунктов списка, — это черный кружок, обычно называемый "горохом".

Листинг 9.2. Простейший список

```
<fo:list-block
  provisional-distance-between-starts="18pt"
  provisional-label-separation="3pt">

  <fo:list-item>

    <fo:list-item-label>
      <fo:block>&#x2022;</fo:block>
    </fo:list-item-label>
```

```

    <fo:list-item-body>
      <fo:block>Первый пункт списка</fo:block>
    </fo:list-item-body>

  </fo:list-item>

  <fo:list-item>

    <fo:list-item-label>
      <fo:block>&#x2022;</fo:block>
    </fo:list-item-label>

    <fo:list-item-body>
      <fo:block>второй пункт списка</fo:block>
    </fo:list-item-body>

  </fo:list-item>

</fo:list-block>

```

Вложенные списки организуются с помощью элемента `fo:block`, в который всегда можно вложить элемент `fo:list-block`. Листинг 9.3 показывает пример вложенного списка. Символ с кодировкой `➘` — это незаполненный квадратик. Он включен, например, в шрифт `ZapfDingbats`.

Листинг 9.3. Вложенный список

```

<fo:list-block start-indent="5mm"
  provisional-distance-between-starts="10mm">

  <fo:list-item>

    <fo:list-item-label>
      <fo:block>&#x2022;</fo:block>
    </fo:list-item-label>

    <fo:list-item-body>

      <fo:block>Сюда вкладывается список.</fo:block>

      <fo:list-block>
        <fo:list-item>

          <fo:list-item-label>

            <fo:block font-family="ZapfDingbats">
              &#x2798;
            </fo:block>

          </fo:list-item-label>

```

```

        <fo:list-item-body>
          <fo:block>Пункт вложенного списка.</fo:block>
        </fo:list-item-body>

      </fo:list-item>
    </fo:list-block>

  </fo:list-item-body>
</fo:list-item>
</fo:list-block>

```

Для создания нумерованного списка надо просто перечислять номера пунктов в элементах `fo:list-item-label`. Если в списке много пунктов, то это трудно сделать, но такие списки обычно генерируются автоматически из некоторой исходной информации. Здесь на помощь приходит язык XSLT с его возможностью организации циклов. Листинг 9.5 показывает вариант нумерованного списка, сгенерированного с помощью языка XSLT. Этот пример взят мною из спецификации языка XSL. Исходный список выглядит так:

```

<ol>
  <item>Первый пункт списка.</item>
  <item>Второй пункт списка.</item>
  <item>Третий пункт списка.</item>
</ol>

```

Пункты этого списка будут пронумерованы малыми латинскими буквами а, b, с, ... Листинг 9.4 показывает преобразование с помощью XSLT.

Листинг 9.4. Преобразование списка с помощью XSLT

```

<?xml version="1.0" encoding="windows-1251"?>

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

  <xsl:template match="ol">

    <fo:list-block
      provisional-distance-between-starts="15mm"
      provisional-label-separation="5mm">

      <xsl:apply-templates />

    </fo:list-block>

  </xsl:template>

```

```

<xsl:template match="ol/item">
  <fo:list-item>
    <fo:list-item-label start-indent="5mm">
      <fo:block>
        <xsl:number format="a."/>
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>
        <xsl:apply-templates/>
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>
</xsl:stylesheet>

```

Результат преобразования — нумерованный список — показан в листинге 9.5.

Листинг 9.5. Нумерованный список, полученный с помощью XSLT

```

<fo:list-block
  provisional-distance-between-starts="15mm"
  provisional-label-separation="5mm">
  <fo:list-item>
    <fo:list-item-label start-indent="5mm">
      <fo:block>a.</fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>Первый пункт списка.</fo:block>
    </fo:list-item-body>
  </fo:list-item>
  <fo:list-item>
    <fo:list-item-label start-indent="5mm">
      <fo:block>b.</fo:block>
    </fo:list-item-label>

```

```

    <fo:list-item-body>
      <fo:block>Второй пункт списка.</fo:block>
    </fo:list-item-body>

  </fo:list-item>

  <fo:list-item>

    <fo:list-item-label start-indent="5mm">
      <fo:block>с.</fo:block>
    </fo:list-item-label>

    <fo:list-item-body start-indent="body-start()">
      <fo:block>Третий пункт списка.</fo:block>
    </fo:list-item-body>

  </fo:list-item>

</fo:list-block>

```

Упражнение

Запишите на языке XSL оформление маркированного списка, принятое в этой книге.

Таблицы

Таблица в языке XSL подобна таблице языка HTML. Она описывается элементом `fo:table`, в который вкладываются элементы `fo:table-row`, описывающие строки таблицы. Каждая строка состоит только из ячеек, описываемых элементами `fo:table-cell`. В элемент `fo:table-cell` вкладывается один или несколько элементов `fo:block` с содержимым ячейки таблицы. Все вместе выглядит примерно так, как показано в листинге 9.6.

Листинг 9.6. Простейшая таблица

```

<fo:table border="0.5pt solid black" text-align="center">

  <fo:table-body>

    <fo:table-row>

      <fo:table-cell padding="6pt" border="0.5pt solid black">
        <fo:block>Первая ячейка первой строки.</fo:block>
      </fo:table-cell>
    </fo:table-row>
  </fo:table-body>
</fo:table>

```

```
<fo:table-cell padding="6pt" border="0.5pt solid black">
  <fo:block>Вторая ячейка первой строки.</fo:block>
</fo:table-cell>

</fo:table-row>

<fo:table-row>

  <fo:table-cell padding="6pt" border="0.5pt solid black">
    <fo:block>Первая ячейка второй строки.</fo:block>
  </fo:table-cell>

  <fo:table-cell padding="6pt" border="0.5pt solid black">
    <fo:block>Вторая ячейка второй строки.</fo:block>
  </fo:table-cell>

</fo:table-row>

</fo:table-body>

</fo:table>
```

Заметьте, что в элемент `fo:table` вложен элемент `fo:table-body`. Это обязательный элемент, таких элементов можно вложить в `fo:table` сколько угодно. Дело в том, что строение таблицы в языке XSL сложнее, чем в языке HTML. В элемент `fo:table` можно вложить еще, кроме элементов `fo:table-body`, один элемент `fo:table-header` и один элемент `fo:table-footer`. Эти элементы строят шапку и итоговые строки таблицы — в них вкладываются элементы `fo:table-row` или непосредственно элементы `fo:table-cell`, причем хотя бы один из этих элементов обязателен.

Кроме того, в таблицах XSL можно задать общие свойства для каждого столбца. Для этого в элемент `fo:table` вкладываются элементы `fo:table-column`, каждый из них описывает общие свойства одного столбца. С его помощью можно задать фон, общий для всего столбца, ширину столбца, его номер, как показано в листинге 9.6.

Заголовок таблицы или подпись под ней описывается отдельным элементом `fo:table-caption`, не связанным с элементом `fo:table`. В элемент `fo:table-caption` вкладывается один или несколько блоков, содержащих заголовок или подпись.

Для того чтобы таблица была целостным объектом, элементы `fo:table-caption` и `fo:table` вкладываются в элемент `fo:table-and-caption`. Этот элемент подобен тегу `<table>` языка HTML. Его атрибут `caption-side` определяет положение заголовка следующими значениями:

- "before" или "top" — заголовок перед таблицей;
- "after" или "bottom" — подпись под таблицей;

- "left" или "start" — слева от таблицы;
- "right" или "end" — справа от таблицы.

Полная структура таблицы, с учетом этого, выглядит так, как показано в листинге 9.7.

Листинг 9.7. Полная структура таблицы

```
<fo:table-and-caption
  text-align="center"
  start-indent="100pt">

  <fo:table-caption
    start-indent="0pt"
    text-align="start">

    <fo:block>Заголовок таблицы.</fo:block>

  </fo:table-caption>

  <fo:table width="12cm" table-layout="fixed">

    <fo:table-column column-width="100pt" column-number="1">
    </fo:table-column>

    <fo:table-column column-width="150pt" column-number="2">
    </fo:table-column>

    <fo:table-column column-width="75pt" column-number="3">
    </fo:table-column>

    <fo:table-body start-indent="0pt" text-align="start">

      <fo:table-row>

        <fo:table-cell>
          <fo:block>Первая ячейка первой строки.</fo:block>
        </fo:table-cell>

        <fo:table-cell>
          <fo:block>Вторая ячейка первой строки.</fo:block>
        </fo:table-cell>

        <fo:table-cell>
          <fo:block>Третья ячейка первой строки.</fo:block>
        </fo:table-cell>

      </fo:table-row>
```

```
</fo:table-body>

</fo:table>

</fo:table-and-caption>
```

Форматеры XSL

В то время, когда писалась эта книга, форматеры XSL еще не были встроены в браузеры. Как правило, каждый форматер преобразует документ XSL в определенный формат, чаще всего PDF. Перечислим некоторые наиболее известные форматеры.

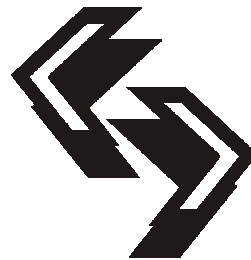
- ❑ Свободно распространяемый Java-форматер FOP (FO Processor), создаваемый сообществом Apache Software Foundation, <http://xml.apache.org/fop>, до сих пор не вышел из подготовительной стадии. Он преобразует документ XSL в форматы PDF, PCL, PS, SVG, XML, Print, AWT, MIF.
- ❑ Коммерческий XSL-Formatter фирмы Antenna House написан под Windows и UNIX, <http://www.antennahouse.com/>.
- ❑ Java-форматер XSLFast фирмы jCatalog Software AG предлагает графический редактор XSL, <http://www.xslfast.com/>.
- ❑ Форматер XEP написан на Java фирмой RenderX, <http://renderx.com/>.
- ❑ Форматер Арос XSL-FO написан фирмой Chive Software для платформы .NET, <http://www.chive.com/products/apoc/>.
- ❑ Свободно распространяемый XML-браузер X-Smiles содержит форматер XSL, <http://www.xsmiles.org/>.
- ❑ Свободно распространяемый Java-форматер jfor преобразует документы XSL в формат RTF, <http://www.jfor.org/>.
- ❑ Продукт RTF2FO фирмы Novosoft Inc. делает обратное преобразование из RTF в XSL-FO, <http://rtf2fo.novosoft-us.com/>.
- ❑ Еще один конвертер html2fo преобразует документ HTML в XSL-FO, <http://sourceforge.net/projects/html2fo/>.

Вопросы для самопроверки

1. Что такое "форматирование документа"?
2. Какова цель форматирования?
3. Какие устройства способны "понимать" команды форматирования?
4. Какова связь между языками XSL и CSS2?
5. Каково строение страницы XSL?

6. Как направить информацию в определенную область страницы?
7. Какие средства предлагает язык XSL для форматирования абзаца?
8. Как определить размер красной строки?
9. Как задать расстояние между абзацами?
10. Как вставить изображение в текст страницы?
11. Каким образом поместить номер текущей страницы?
12. Что может служить меткой маркированного списка?
13. Как создать нумерованный список?
14. Что может служить номером пункта списка?
15. Как задать разный цвет фона для разных строк таблицы?
16. Как задать разный цвет фона для разных столбцов таблицы?
17. Как выделить цветом отдельные ячейки таблицы?
18. Как выделить рамкой строки, столбцы и отдельные ячейки таблицы?

ГЛАВА 10



Обработка документов XML при помощи событий

В предыдущих главах мы все время ссылались на программы-обработчики XML, называемые также процессорами XML. Эти программы читают документы XML, анализируют их элементы, извлекают нужную информацию, преобразуют ее по правилам, заложенным в них или указываемым в процессе их работы. Все правила оформления документов XML для того и придуманы, чтобы облегчить работу программ-обработчиков, помочь им быстро и точно отобрать нужную информацию. Наша книга будет неполной, если мы не ознакомимся со строением процессоров XML. Понимание принципов их работы значительно облегчит вам создание документов XML. Может быть, вы хотите сами написать обработчик ваших документов? В этом нет ничего невозможного, надо только знать какой-нибудь язык программирования и познакомиться с библиотекой функций или классов, предназначенных для работы с XML.

В этой книге я выбрал для описания процессоров XML язык программирования Java. На то есть три причины. Во-первых, этот язык хорошо подходит для описания алгоритмов. Программы, написанные на Java, легко читаются и почти не нуждаются в комментариях. Во-вторых, большинство промышленных процессоров XML, коммерческих и свободно распространяемых, написано на Java. В-третьих, в Интернете много свободно распространяемых библиотек классов Java, предназначенных для работы с XML. Конечно, для понимания этой и следующей главы вам нужно иметь некоторые навыки программирования и знать Java хотя бы в начальном объеме.

Стандартные средства Java для обработки XML

В стандартную поставку Java 2 SDK (Software Development Kit) Standard Edition, имеющуюся во всех программных инструментальных средствах ра-

боты с Java, входит обширная библиотека классов, предназначенных для работы с XML. Она расположена в пакетах `javax.xml.parsers`, `javax.xml.transform`, `javax.xml.transform.dom`, `javax.xml.transform.sax`, `javax.xml.transform.stream`, `org.w3c.dom`, `org.w3c.dom.css`, `org.w3c.dom.stylesheets`, `org.w3c.dom.html`, `org.w3c.dom.views`, `org.w3c.dom.traversal`, `org.w3c.dom.events`, `org.xml.sax`, `org.xml.sax.ext`, `org.xml.sax.helpers`.

Общее название интерфейсов и классов, входящих в эти пакеты, — JAXP (Java API for XML Processing). В то время, когда писалась эта книга, последней версией JAXP была версия 1.2.3. Кроме того, что библиотека JAXP входит в стандарт Java 2 SDK, она свободно доступна вместе с исходными текстами по адресу <http://java.sun.com/xml/jaxp/>. Вы всегда можете скопировать последнюю версию библиотеки JAXP на компьютер, уже содержащий Java 2 SDK или другое инструментальное средство разработки Java, и пользоваться всеми интерфейсами и классами JAXP.

Анализ документа XML

На первом этапе разбора документа XML проводится его *лексический анализ* (lexical parsing). Документ разбивается на отдельные неделимые элементы (tokens), которыми являются теги, служебные слова, разделители, текстовые константы. Проводится проверка полученных элементов и их связей между собой. Лексический анализ выполняют специальные программы — *сканеры* (scanners).

Примерами простейших сканеров обычного "плоского" текста могут служить классы `java.util.StringTokenizer` и `java.io.StreamTokenizer` из стандартной поставки Java 2 SDK Standard Edition. Они разбивают текст на слова, используя в качестве разделителей слов пробельные символы или символы, указанные в конструкторе класса. В листинге 10.1 написана простая программа на Java, разбивающая строку обычного текста на слова и сохраняющая их в векторе.

Листинг 10.1. Сканирование простого текста

```
import java.util.*;

public class SimpleTextScanner{

    public static void main(String[] args){

        String text="Это текст, который будет разбит на слова.";

        StringTokenizer st =
            new StringTokenizer(text, " \t\r\n,.;");
```

```
Vector v = new Vector();

while (st.hasMoreTokens())
    v.add(st.nextToken());
}
```

Первая строка листинга 10.1 `import java.util.*` указывает компилятору Java, что классы `StringTokenizer` и `Vector`, использованные в программе, следует искать в пакете `java.util`. Затем создается класс `SimpleTextScanner`, состоящий лишь из одного метода обработки информации `main()`. В методе `main()` в объект `text` типа `String` записывается исходный текст и конструируется объект-сканер `st`. В его конструкторе указывается исходный объект сканирования `text` и строка `" \t\r\n,.;"`, перечисляющая разделители слов. В нее включен пробел, символ табуляции, символ возврата каретки, символ перевода строки, запятая, точка, двоеточие и точка с запятой. При желании можно добавить и другие разделители.

Все сканирование проводится двумя методами класса `StringTokenizer`. Первый метод, `hasMoreTokens()`, следит за тем, есть ли еще слова в тексте, и возвращает в зависимости от результата проверки истину `true` или ложь `false`. В первом случае будет выполнена следующая итерация цикла `while`, сработает метод `nextToken()`, извлекающий из текста следующее слово, а потом будет выполнен метод `add()`, заносающий это слово в очередной элемент вектора `v` типа `Vector`.

Как видите, вся сложность лексического анализа "плоского" текста спрятана внутри класса `StringTokenizer` и сведена к двум простым методам. Мы только пользуемся этими готовыми методами. Это обычное правило объектно-ориентированного программирования — инкапсулировать в классы и их методы всю сложность обработки информации.

После сканирования выполняется *грамматический анализ* (grammar parsing) документа. При этом анализируется логическая структура документа, элементы, полученные в результате сканирования, собираются в выражения, выражения объединяются в блоки, блоки — в модули, которыми могут являться абзацы, параграфы, пункты, главы. Грамматический анализ проводят программы-анализаторы, так называемые *парсеры* (parsers).

Создание сканеров и парсеров — любимое развлечение программистов. За недолгую историю XML написаны десятки, если не сотни XML-парсеров, многие из них — на языке Java.

Все XML-анализаторы можно разделить на две группы.

В первую группу входят сканеры и парсеры, проводящие анализ, основываясь на структуре дерева, отражающего вложенность элементов документа (tree-based parsing). Они строят дерево документа в оперативной памяти. Та-

кие анализаторы проще реализовать, их удобно использовать для интерактивной работы с документом, но создание дерева требует большого объема оперативной памяти, ведь размер документов XML не ограничен. Кроме того, необходимость частого просмотра дерева замедляет работу анализатора.

Во вторую группу входят сканеры и парсеры, проводящие анализ, основываясь на событиях (event-based parsing). Событием считается появление какого-либо элемента XML: открывающего или закрывающего тега, текста, содержащегося в теле элемента. При возникновении события вызывается соответствующий метод его обработки: `startElement()`, `endElement()`, `characters()`, ... Такие анализаторы сложнее в реализации, зато они не строят дерево в оперативной памяти и могут анализировать не весь документ, а его отдельные элементы вместе с вложенными в них элементами. Существенный недостаток этих анализаторов заключается в том, что они просматривают документ только один раз от начала к концу, и не могут возвратиться назад.

Фактическим стандартом анализаторов, основанных на событиях, стал свободно распространяемый набор классов и интерфейсов SAX (Simple API for XML Parsing), созданный Дэвидом Меггинсоном (David Megginson). Основной сайт этого проекта <http://www.saxproject.org/>. Сейчас применяется второе поколение этого набора, называемое SAX2. Набор SAX2 входит во многие XML-анализаторы, например, Xerces2.

Библиотека интерфейсов и классов JAXP позволяет быстро создать XML-анализаторы обоих типов. С помощью одной из частей этой библиотеки, называемой DOM API (Document Object Model API), можно создавать анализаторы первого типа, создающие дерево объектов. Мы рассмотрим DOM API в следующей главе. С помощью второй части библиотеки JAXP, называемой SAX API, можно создавать SAX-анализаторы. Интерфейсы и классы SAX2 собраны в пакеты `org.xml.sax`, `org.xml.sax.ext`, `org.xml.sax.helpers`, `javax.xml.parsers`. Рассмотрим их подробнее.

Принципы анализа с помощью SAX2 API

Набор интерфейсов и классов SAX2, реализованный в JAXP, представляет собой реализацию SAX-анализатора на языке Java и шаблоны для записи правил обработки документов XML. Правила обработки, как и все в Java, следует оформить в виде класса. Создание класса-обработчика (handler) событий выпадает на долю разработчика. Для этого надо реализовать некоторые интерфейсы, наполнив их командами разбора конкретных документов XML и выделения нужной информации.

Основу построения SAX2-обработчика составляет интерфейс `org.xml.sax.ContentHandler`, описывающий методы обработки определенных событий: начала документа, появления открывающего тега, появление тела

элемента, появление закрывающего тега, окончание документа. При возникновении каждого такого события SAX2-анализатор обращается к методу-обработчику события, передавая ему аргументы, содержащие информацию о событии. Дело разработчика анализатора — реализовать эти методы, обеспечив правильный анализ документа.

Как только документ XML поступает на обработку, SAX2-анализатор обращается к методу `public void startDocument();` класса-обработчика. У этого метода нет аргументов. В нем можно задать какие-то начальные действия по обработке документа на усмотрение разработчика. Он выполняется только один раз перед просмотром документа. Этот метод можно не расписывать, если никакие начальные действия не нужны.

Далее документ начинает просматриваться от начала к концу. При каждом появлении символа "<", с которого начинается открывающий тег элемента XML, вызывается еще один метод

```
public void startElement(String uri, String name,
    String qname, Attributes attrs);
```

класса-обработчика. В метод `startElement()` передаются три имени, два из которых связаны с пространством имен: идентификатор пространства имен `uri`, имя тега без префикса `name` и уточненное имя с префиксом `qname`. Если пространство имен не определено, то значение первого аргумента равно `null`, а второй и третий аргументы содержат одно и то же локальное имя элемента.

В четвертом аргументе в метод передаются атрибуты элемента `attrs`, если они есть. Атрибуты оформляются в виде объекта типа `Attributes`. В этом объекте атрибуты нумеруются в порядке их записи в элементе XML, начиная с нуля. Количество атрибутов можно получить методом

```
int getLength();
```

интерфейса `Attributes`, а значения — методами того же интерфейса:

```
String getValue(int index);
String getValue(String qname);
String getValue(String uri, String localName);
```

Первый метод дает значение атрибута по его номеру `index`, второй — по уточненному имени `qname`, а третий — по идентификатору пространства имен `uri` и локальному имени `localName`. Если атрибут не найден, то эти методы возвращают пустое значение `null`.

Следующие методы интерфейса `Attributes`:

```
String getLocalName(int index);
String getQName(int index);
String getURI(int index);
```


возвращают различные части имени атрибута: локальное имя, уточненное имя и идентификатор пространства имен. Методы:

```
String getType(int index);  
String getType(String qname);  
String getType(String uri, String localName);
```

возвращают тип атрибута. Наконец, методы:

```
int getIndex(String qname);  
int getIndex(String uri, String localName);
```

дают порядковый номер атрибута по его имени.

Если у элемента нет атрибутов, то в метод `startElement()` в четвертом аргументе передается ссылка на пустой объект `attrs`.

Таким образом, разработчик класса-обработчика получает от SAX2-анализатора всю информацию, заключенную в открывающем теге элемента. Ему остается только расписать метод `startElement()`, обработав имя элемента и его атрибуты. При этом разработчик может использовать аргументы `uri`, `name`, `qname`, `attrs` для того, чтобы узнать имя элемента, количество атрибутов, их имена и значения.

При появлении символов "</", начинающих закрывающий тег, вызывается метод

```
public void endElement(String uri, String name, String qname);
```

Его аргументы передают в метод имя элемента, которое можно использовать для завершения обработки элемента. Это особенно полезно при обработке вложенных элементов.

При появлении строки символов, находящейся внутри элемента или в другом месте документа, SAX-анализатор вызывает метод

```
public void characters(char[] ch, int start, int length);
```

В него передается массив символов `ch`, индекс начала строки символов `start` в этом массиве, обычно это 0, и количество символов `length`.

При появлении в тексте документа XML инструкции по обработке вызывается метод

```
public void processingInstruction(String target, String data);
```

В него передается имя программы-обработчика инструкции `target` и дополнительные сведения `data`, т. е. содержимое инструкции по обработке.

При появлении "лишних" пробельных символов, которые должны быть пропущены парсером, вызывается метод

```
public void ignorableWhitespace(char[] ch, int start, int length);
```

В него передается массив `ch` идущих подряд пробельных символов, индекс начала символов в массиве `start` и количество символов `length`. Не следует слишком полагаться на этот метод для удаления повторяющихся пробельных символов, поскольку многие SAX-анализаторы при их появлении вызывают метод `characters()`, а не метод `ignorableWhitespace()`.

Наконец, после обработки всего документа, SAX2 обращается к методу

```
void endDocument();
```

в котором можно записать заключительные действия.

Интерфейс `org.xml.sax.ContentHandler` уже реализован в JAXP классом `org.xml.sax.helpers.DefaultHandler`. Впрочем, эта реализация пустая, в теле методов класса `DefaultHandler` ничего нет. Ее польза в том, что разработчику остается реализовать только те методы, которые ему нужны, оставив другие методы пустыми.

Интерфейс `Attributes` тоже реализован в JAXP классом `org.xml.sax.helpers.AttributesImpl`. Кроме реализации всех методов вида `getXxx()` интерфейса `Attributes`, класс `AttributesImpl` содержит следующие конструкторы:

```
AttributesImpl();  
AttributesImpl(Attributes atts);
```

а также методы заполнения объекта типа `AttributesImpl` полезной информацией:

```
void setAttribute(int index, String uri,  
    String localName, String qName, String type, String value);  
void setAttributes(Attributes atts);  
void setLocalName(int index, String localName);  
void setQName(int index, String qName);  
void setType(int index, String type);  
void setURI(int index, String uri);  
void setValue(int index, String value);
```

Пользуясь конструкторами и методами вида `setXxx()`, разработчик парсера может создать новые атрибуты любого элемента.

После того как класс-обработчик документов XML написан, остается запустить SAX-анализатор и передать ему документ XML и экземпляр класса-обработчика. Стандартным анализатором в JAXP служит класс `javax.xml.parsers.SAXParser`, но разработчик может создать свою версию анализатора.

Анализатор читает документ XML и обращается к классу-обработчику методами, описанными в интерфейсе `org.xml.sax.XMLReader`. Всякий SAX2-

анализатор должен реализовать этот интерфейс, а большинство промышленных анализаторов содержат в себе даже несколько реализаций интерфейса XMLReader.

Извлечение содержимого документа XML

В качестве первого примера напишем класс-обработчик, выделяющий полезное содержимое документа XML и выводящий это содержимое на консоль, разделив значения атрибутов пробелами и внося символы перевода строки после списка значений атрибутов и содержимого каждого элемента. Программа-обработчик, выполняющая эти действия, записана в листинге 10.2.

Листинг 10.2. Извлечение содержимого документа XML

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;

public class ContentParser extends DefaultHandler{

    static private String fileName;

    public static void main(String[] args){

        if (args.length != 1){
            System.err.println("Usage: java ContentParser <filename>");
            System.exit(1);
        }

        fileName=args[0];

        DefaultHandler handler = new ContentParser();

        SAXParserFactory fact = SAXParserFactory.newInstance();

        try{

            SAXParser saxParser = fact.newSAXParser();
            saxParser.parse(new File(fileName), handler);

        }catch (Throwable t){
            t.printStackTrace();
        }
    }
}
```

```
public void startDocument() throws SAXException{

    System.out.println("Содержимое файла " + fileName + ":");
}

public void startElement(String uri, String sName,
    String qName, Attributes attrs) throws SAXException{

    if (attrs != null){

        for (int i = 0; i < attrs.getLength(); i++){
            System.out.print(attrs.getValue(i) + " ");

            System.out.println();
        }
    }

    public void characters(char buf[], int offset, int len)
        throws SAXException{

        System.out.print(new String(buf, offset, len));
    }
}
```

Если вы используете для выполнения программ Java 2 SDK, то программу листинга 10.2 надо записать в файл с именем `ContentParser.java` и откомпилировать:

```
$ javac ContentParser.java
```

Затем надо запустить интерпретатор Java, передав на его вход откомпилированный класс `ContentParser` и имя файла с документом XML:

```
$ java ContentParser notebook.xml
```

После этого вы увидите на консоли результат обработки файла `notebook.xml`.

Программа листинга 10.2 реализует три метода интерфейса `ContentHandler`:

- ❑ `startDocument()` — выводит на консоль заголовок "Содержимое файла", добавляя к нему имя файла с документом XML;
- ❑ `startElement()` — выбирает атрибуты, если они есть, и выводит их значения на консоль через пробел;
- ❑ `characters()` — просто выводит на консоль переданные ему символы.

Метод `main()` занимается инициализацией SAX2-анализатора и передачей ему документа XML и правил его обработки, содержащихся в классе-обработчике. Рассмотрим этот процесс подробнее.

Инициализация SAX2-анализатора

Работа SAX2-анализатора тесно связана с программным окружением машины, на которой он выполняется. Поэтому анализатор создается не конструктором, как обычные объекты Java, а фабричным методом `newSAXParser()` класса `javax.xml.parsers.SAXParserFactory`. Этот метод создает стандартный SAX2-анализатор — объект класса `SAXParser` — с параметрами, определяемыми "фабрикой" `SAXParserFactory`. Сама "фабрика" создается своим собственным статическим методом `newInstance()`. После ее создания можно установить возможность проверки документа методом `setValidating(true)` и возможность использования уточненных имен методом `setNamespaceAware(true)`. По умолчанию эти возможности отключены.

Все вместе выглядит так:

```
SAXParserFactory fact = SAXParserFactory.newInstance();

fact.setValidating(true);
fact.setNamespaceAware(true);

SAXParser saxParser = fact.newSAXParser();
```

Кроме того, после создания "фабрики" можно методами

```
void setFeature(String name, boolean value);
```

задать дополнительные свойства создаваемых "фабрикой" SAX2-парсеров. Эти свойства меняются с каждой версией SAX, поэтому список всех доступных свойств прилагается к документации, он находится в описании пакета `org.xml.sax`.

Например, после

```
fact.setFeature("http://xml.org/sax/features/namespace-prefixes", true);
```

парсеры, создаваемые фабрикой `fact`, будут учитывать префиксы имен элементов и атрибутов.

После того как SAX2-анализатор создан, остается только применить один из его методов `parse()`, передав этому методу имя или местоположение файла с анализируемым документом XML и экземпляр класса-обработчика событий.

В классе `javax.xml.parsers.SAXParser` есть добрый десяток методов `parse()`. Они различаются местоположением исходного файла с документом XML. Кроме метода `parse(File, DefaultHandler)`, использованного в листинге 10.2, есть еще методы, позволяющие извлечь документ из входного потока класса `InputStream`, объекта класса `InputSource`, адреса URI или из специально созданного источника класса `InputSource`.

Сразу после создания экземпляра SAX-анализатора, его можно настроить, установив некоторые свойства. Для этого сначала надо извлечь из анализатора ссылку на объект типа `XMLReader`. Это можно сделать так:

```
XMLReader xmlReader = saxParser.getXMLReader();
```

Затем методами `setFeature()` и `setProperty()`, описанными в интерфейсе `XMLReader`, можно задать различные свойства обработчика. В листинге 10.5 таким образом подключается дополнительная обработка событий, а в листинге 10.9 метод `setProperty()` использован для того, чтобы парсер проверял правильность документа с помощью схемы XSD. Эта возможность включена в JAXP, начиная с версии JAXP 1.2.

Если парсер выполняет проверки, т. е. применен метод `setValidating(true)`, то имеет смысл сделать развернутые сообщения об ошибках. Это предусмотрено интерфейсом `ErrorHandler`. Он различает предупреждения, ошибки и фатальные ошибки и описывает три метода, которые автоматически выполняются при появлении ошибки соответствующего вида:

```
public void warning(SAXParserException ex);  
public void error(SAXParserException ex);  
public void fatalError(SAXParserException ex);
```

Класс `DefaultHandler` предоставляет пустую реализацию этого интерфейса. При расширении этого класса можно сделать реализацию одного или всех методов интерфейса `ErrorHandler`. Пример такой реализации приведен в листинге 10.9. Класс `SAXParserException` хранит номер строки и столбца проверяемого документа, в котором замечена ошибка. Их можно получить методами `getLineNumber()` и `getColumnNumber()`, как это сделано в листинге 10.9.

Упражнение

Вспомните, что символы перевода строки и пробелы образуют текстовые узлы дерева документа, и подумайте, как избавиться от вывода их на консоль.

Поиск элемента в документе XML

Второй пример SAX2-парсера, приведенный в листинге 10.3, показывает, как осуществить поиск элементов с указанным именем в документе XML и извлечение атрибутов этих элементов и их значений на консоль. Имя файла с документом и имя элемента передаются парсеру как аргументы командной строки, т. е. после компиляции в командной строке набираем что-нибудь вроде

```
$ java ElemParser notebook.xml name
```

Листинг 10.3. Поиск элемента XML

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;

public class ElemParser extends DefaultHandler{

    static private String elem;
    private boolean isElem, isAttrs;

    public static void main(String[] args){

        if (args.length != 2){

            System.err.println(
                "Usage: java ElemParser <fileName> <elemName>");

            System.exit(1);
        }

        elem=args[1];

        DefaultHandler handler = new ElemParser();

        SAXParserFactory fact = SAXParserFactory.newInstance();

        fact.setValidating(true);
        fact.setNamespaceAware(true);

        try{
            SAXParser saxParser = fact.newSAXParser();
            saxParser.parse( new File(args[0]), handler);
        }catch(Throwable t){
            t.printStackTrace();
        }
    }

    public void startDocument() throws SAXException{

        System.out.println("Элемент " + elem);
    }

    public void endDocument() throws SAXException{

        if (!isElem) System.out.println("не найден.");
        else if (!isAttrs)
            System.out.println("не имеет атрибутов.");
    }
}
```

```

public void startElement(String uri, String sName,
    String qName, Attributes attrs) throws SAXException{

    if (qName.equals(elem)){

        isElem = true;

        if (attrs.getLength() != 0){

            isAttrs = true;

            for (int i = 0; i < attrs.getLength(); i++)
                System.out.print(
                    attrs.getQName(i) + ": " +
                    attrs.getValue(i) + " ");

            System.out.println();

        }

    }

}

```

Упражнение

Задайте поиск элементов по указанному атрибуту.

Извлечение различных сведений

Решим более конкретную и немного более сложную задачу — выведем на консоль все телефоны заданного лица из нашей записной книжки. Имя, отчество и фамилию будем записывать в командной строке примерно так:

```
$ java PhoneParser Иван Петрович Сидоров
```

Листинг 10.4. Поиск телефонов заданного лица

```

import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;

public class PhoneParser extends DefaultHandler{

    static private String firstname, secondname, surname;
    private boolean isElem, isName, inPhoneList;

```



```
public static void main(String[] args){

    if (args.length != 3){

        System.err.println("Usage: java PhoneParser" +
            " <firstname> <secondname> <surname>");

        System.exit(1);
    }

    firstname = args[0];
    secondname = args[1];
    surname = args[2];

    DefaultHandler handler = new PhoneParser();

    SAXParserFactory fact = SAXParserFactory.newInstance();

    fact.setValidating(true);
    fact.setNamespaceAware(true);

    try{

        SAXParser saxParser = fact.newSAXParser();
        saxParser.parse("notebook.xml", handler);

    }catch(Throwable t){
        t.printStackTrace();
    }
}

public void startDocument() throws SAXException{

    System.out.println(firstname + " " + secondname +
        " " + surname);
}

public void endDocument() throws SAXException{

    if (!isName) System.out.println("не найден.");
}

public void startElement(String uri, String sName,
    String qName, Attributes attrs) throws SAXException{

    if (qName.equals("phone-list")) inPhoneList = true;

    if (qName.equals("name"))
```

```
        if (attrs.getLength() != 0)

            if (attrs.getValue(0).equals(firstname) &&
                attrs.getValue(1).equals(secondname) &&
                attrs.getValue(2).equals(surname))

                isElem = isName = true;
    }

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

        if (qName.equals("phone-list"))

            inPhoneList = isElem = false;
    }

    public void characters(char buf[], int offset, int len)
    throws SAXException{

        if (isElem && inPhoneList)

            System.out.print(new String(buf, offset, len));
    }
}
```

В листинге 10.4 очевиден недостаток парсеров, основанных на событиях, вытекающий из того, что документ просматривается парсером один раз от начала к концу, — метод `characters()` не связан с элементом, чье содержимое он извлекает. Поэтому пришлось вводить логические переменные `isElem`, `inPhoneList`, чтобы вывести на консоль телефонные номера указанного лица, а не какую-то другую информацию.

Упражнения

1. Добавьте к номерам телефонов пояснения: "Рабочий", "Домашний".
2. Выведите адрес указанного лица.
3. По заданному номеру телефона найдите его владельца.

Дополнительные события SAX

Интерфейс `ContentHandler` описывает не все события, которые могут возникнуть при просмотре документа XML. Он отмечает начало и окончание просмотра обращением к методам `startDocument()` и `endDocument()`, появление открывающего и закрывающего тегов элемента вызовом методов `startElement()` и `endElement()`, появление текста методом `characters()` и

появление инструкции по обработке методом `processingInstruction()`. "За бортом" остаются секции CDATA, комментарии, объявления DTD. В большинстве случаев при обработке документов XML эти сведения просто не нужны, но метод `characters()` выводит содержимое секций CDATA.

Для того чтобы все-таки отслеживать эти события, в SAX2 введен пакет `org.xml.sax.ext` с двумя интерфейсами `DeclHandler` и `LexicalHandler`. Интерфейс `LexicalHandler` содержит описание следующих методов:

```
void comment(char[] ch, int start, int length);
void startCDATA();
void endCDATA();
void startDTD(String name, String publicId, String systemId);
void endDTD();
void startEntity(String name);
void endEntity(String name);
```

Из названий этих методов понятно, что они отмечают появление комментария, начала и окончания секции CDATA, начала и окончания ссылки на сущности.

В интерфейсе `DeclHandler` описаны методы, обрабатывающие объявления DTD:

```
void attributeDecl(String eName, String aName, String type,
    String valueDefault, String value);
void elementDecl(String name, String model);
void externalEntityDecl(String name,
    String publicId, String systemId);
void internalEntityDecl(String name, String value);
```

Эти методы вызываются при объявлении атрибутов, элементов, внешних и внутренних сущностей, т. е. при появлении элементов `<!ATTLIST>`, `<!ELEMENT>` и `<!ENTITY>`.

Интерфейсы `LexicalHandler` и `DeclHandler` считаются дополнениями к пакету SAX2. Программы-анализаторы не обязаны их реализовывать. Поэтому перед использованием методов этих интерфейсов следует установить свойства SAX2-анализатора, обеспечивающие обращение его к этим методам. Для установки этих свойств следует получить от объекта `saxParser` ссылку на объект типа `XMLReader` и выполнить его методы `setProperty()`, передав в них ссылку на класс-обработчик:

```
XMLReader xmlReader = saxParser.getXMLReader();

xmlReader.setProperty(
    "http://xml.org/sax/properties/lexical-handler", handler);
```

```
xmlReader.setProperty(  
    "http://xml.org/sax/properties/declaration-handler", handler);
```

Если парсер не обрабатывает дополнительные события, то эти методы выбросят исключения типа `SAXNotRecognizedException` или `SAXNotSupportedException`.

Кроме того, надо явно реализовать интерфейс `LexicalHandler` и/или `DeclHandler`, поскольку класс `DefaultHandler` этого не делает.

Дополним, в качестве примера, листинг 10.2 выводом комментариев. Результат показан в листинге 10.5.

Листинг 10.5. Дополнительные события SAX2

```
import java.io.*;  
import org.xml.sax.*;  
import org.xml.sax.ext.*;  
import org.xml.sax.helpers.*;  
import javax.xml.parsers.*;  
  
public class LexParser extends DefaultHandler  
    implements LexicalHandler{  
  
    static private String fileName;  
  
    public static void main(String[] args){  
  
        if (args.length != 1){  
  
            System.err.println("Usage: java LexParser <filename>");  
            System.exit(1);  
        }  
  
        fileName=args[0];  
  
        DefaultHandler handler = new LexParser();  
  
        SAXParserFactory fact = SAXParserFactory.newInstance();  
  
        fact.setValidating(true);  
        fact.setNamespaceAware(true);  
  
        try{  
            SAXParser saxParser = fact.newSAXParser();  
  
            XMLReader xmlReader = saxParser.getXMLReader();
```

```
        xmlReader.setProperty(
            "http://xml.org/sax/properties/lexical-handler",
            handler);

        saxParser.parse( new File(fileName), handler);
    }catch (Throwable t){
        t.printStackTrace();
    }
}

public void startDocument() throws SAXException{

    System.out.println("Содержимое файла " + fileName + ":");
}

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

    System.out.println("Комментарий: "+
        new String(ch, start, length));
}

// Пустые реализации остальных методов интерфейса LexicalHandler.
public void startCDATA(){}
public void endCDATA(){}
public void startDTD(String name, String publicId, String systemId){}
public void endDTD(){}
public void startEntity(String name){}
public void endEntity(String name){}

public void ignorableWhitespace(char[] ch, int start, int length){};

public void startElement(String uri, String sName,
    String qName, Attributes attrs) throws SAXException{

    if (attrs != null){

        for (int i = 0; i < attrs.getLength(); i++)
            System.out.print(attrs.getValue(i) + " ");

        System.out.println();
    }
}

public void characters(char buf[], int offset, int len)
    throws SAXException{

    System.out.print(new String(buf, offset, len));
}
}
```

Упражнение

Реализуйте оставшиеся пустыми в листинге 10.5 методы интерфейса `LexicalHandler`.

Цепочка анализаторов

До сих пор, просматривая документы XML SAX-анализатором типа `XMLReader`, мы пользовались одним обработчиком, расширяя класс `DefaultHandler`. Пакет `SAX2` позволяет создать цепочку обработчиков, по очереди просматривающих документ и вносящих в него изменения. Такие обработчики сами должны быть SAX-анализаторами. Каждый обработчик в цепочке должен быть фильтром, принимающим на вход документ XML от предыдущего SAX-анализатора, изменяющим документ и передающим его следующему анализатору.

Методы создания такого обработчика-фильтра описаны в интерфейсе `org.xml.sax.XMLFilter`. Хотя в нем всего два метода:

```
XMLReader getParent();  
void setParent(XMLReader parent);
```

связывающих обработчик с предыдущим SAX-анализатором типа `XMLReader`, но интерфейс `XMLFilter` расширяет интерфейс `XMLReader` и тем самым наследует все методы, описанные в `XMLReader`.

Таким образом, интерфейс `XMLFilter` описывает `XMLReader` с дополнительным свойством — у него есть родительский `XMLReader`. Это означает, что фильтр — это SAX-анализатор, который может брать документы XML не только из файла, но и от предыдущего SAX-анализатора. Интерфейс `XMLFilter` реализован классом `org.xml.sax.helpers.XMLFilterImpl`, который заодно реализует интерфейсы `ContentHandler` и `ErrorHandler`, являясь, таким образом, полноценным SAX-анализатором и в то же время классом-обработчиком событий SAX.

В конструктор класса `XMLFilterImpl`

```
XMLFilterImpl(XMLReader parent);
```

передается ссылка на объект типа `XMLReader` — родительский SAX-анализатор. Тем самым данный экземпляр класса `XMLFilterImpl` включается в цепочку как следующий фильтр.

Для создания первого фильтра в цепочке применяется конструктор по умолчанию

```
XMLFilterImpl();
```

В листинге 10.6 приведен пример фильтра, меняющего содержимое `oldContent` указанного элемента `name` на новое содержимое `newContent`.

Листинг 10.6. Фильтр, меняющий содержимое элемента

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;

public class ElemFilter extends XMLFilterImpl{

    String name, oldContent, newContent;
    boolean inElem;

    public ElemFilter(){}

    public ElemFilter(XMLReader parent, String name,
        String oldContent, String newContent){

        super(parent);

        this.name = name;
        this.oldContent = oldContent;
        this.newContent = newContent;
    }

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

        if (qName.equals(name)) inElem = true;

        super.startElement(uri, localName, qName, atts);
    }

    public void characters(char buf[], int offset, int len)
        throws SAXException{

        if (inElem){

            String s = new String(buf, offset, len);

            if (s.equals(oldContent)){

                len = newContent.length();
                buf = new char[len]; offset = 0;
                newContent.getChars(0, len, buf, 0);
            }
        }
    }
}
```

```

        super.characters(buf, offset, len);
    }

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

        if (qName.equals(name)) inElem = false;

        super.endElement(uri, localName, qName);
    }
}

```

Применим этот фильтр к нашей адресной книжке листинга 1.2, заменив, например, во всех элементах `city` название "Новокозловск" на "Париж", а затем выведем новое содержимое на консоль, как в листинге 10.2. Для этого метод `main()` листинга 10.2 надо переписать следующим образом:

```

public static void main(String[] args){

    if (args.length != 4){

        System.err.println("Usage: java ContentParser <filename>" +
            " <elem> <old> <new>");

        System.exit(1);
    }

    fileName=args[0];
    String elem = args[1];
    String oldContent = args[2];
    String newContent = args[3];

    DefaultHandler handler = new ContentParser();

    SAXParserFactory fact = SAXParserFactory.newInstance();

    try{
        SAXParser saxParser = fact.newSAXParser();

        XMLReader r = saxParser.getXMLReader();

        ElemFilter ef =
            new ElemFilter(r, elem, oldContent, newContent);

        ef.setContentHandler(handler);
        ef.parse(fileName);
    }
}

```



```
    } catch (Throwable t) {  
        t.printStackTrace();  
    }  
}
```

После компиляции обоих файлов выполняем команду

```
$ java ContentParser notebook.xml city Новокозловск Париж
```

и видим на консоли измененное имя города.

Созданный нами фильтр `ElemFilter` можно применять несколько раз, образовав цепочку анализаторов. Например:

```
XMLReader r = saxParser.getXMLReader();  
  
ElemFilter ef1 =  
    new ElemFilter(r, "city", "Новокозловск", "Париж");  
  
ElemFilter ef2 =  
    new ElemFilter(ef1, "zip", "123456", "654321");  
  
ElemFilter ef3 =  
    new ElemFilter(ef2, "street", "Нижняя, 12", "Broadway, 10");
```

Преобразование элементов XML в объекты Java

Запись документа на языке XML удобна для выявления структуры документа, но неудобна для работы с документом в объектно-ориентированной среде. Поэтому для решения более сложных задач, чем те, что были показаны выше, содержимое документа XML предварительно представляется в виде одного или нескольких объектов Java, называемых *объектами данных JDO* (Java Data Objects). Эта операция называется *связыванием данных* (data binding) с объектами JDO. После связывания данных объектно-ориентированная программа обычным для себя образом манипулирует полученными объектами.

Свяжем содержимое нашей адресной книжки с объектами Java. Для этого сначала опишем классы Java, которые представят содержимое адресной книжки, как это показано в листингах 10.7 и 10.8.

Листинг 10.7. Класс, описывающий адрес

```
public class Address{  
  
    private String street, city, zip, type = "город";
```

```
public Address(){}

public String getStreet(){ return street; }
public void setStreet(String street){ this.street = street; }

public String getCity(){ return city; }
public void setCity(String city){ this.city = city; }

public String getZip(){ return zip; }
public void setZip(String zip){ this.zip = zip; }

public String getType(){ return type; }
public void setType(String type){ this.type = type; }

public String toString(){
    return "Address: " + street + " " + city + " " + zip;
}
}
```

Листинг 10.8. Класс, описывающий запись адресной книжки

```
public class Person{

    private String firstName, secondName, surname, birthday;
    private Vector address;
    private Vector workPhone;
    private Vector homePhone;

    public Person(){}

    public Person(String firstName, String secondName,
                  String surname){

        this.firstName = firstName;
        this.secondName = secondName;
        this.surname = surname;
    }

    public String getFirstName(){ return firstName; }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }

    public String getSecondName(){ return secondName; }
    public void setSecondName(String secondName){
        this.secondName = secondName;
    }
}
```

```
public String getSurname(){ return surname; }
public void setSurname(String surname){
    this.surname = surname;
}

public String getBirthday(){ return birthday; }
public void setBirthday(String birthday){
    this.birthday = birthday;
}

public void addAddress(Address addr){

    if (address == null) address = new Vector();

    address.add(addr);
}

public Vector getAddress(){ return address; }

public void removeAddress(Address addr){
    if (address != null) address.remove(addr);
}

public void addWorkPhone(String phone){

    if (workPhone == null) workPhone = new Vector();

    workPhone.add(new Integer(phone));
}

public Vector getWorkPhone(){ return workPhone; }

public void removeWorkPhone(String phone){

    if (workPhone != null)
        workPhone.remove(new Integer(phone));
}

public void addHomePhone(String phone){

    if (homePhone == null) homePhone = new Vector();

    homePhone.add(new Integer(phone));
}

public Vector getHomePhone(){ return homePhone; }

public void removeHomePhone(String phone){
```

```

        if (homePhone != null)
            homePhone.remove(new Integer(phone));
    }

    public String toString(){
        return "Person: " + surname;
    }
}

```

После определения классов Java, в экземпляры которых будет занесено содержимое адресной книжки, напишем программу, читающую адресную книжку и связывающую ее с объектами Java.

В листинге 10.9 приведен пример класса-обработчика `NotebookHandler` для адресной книжки, описанной в листинге 1.2. Методы класса `NotebookHandler` анализируют содержимое адресной книжки и помещают его в вектор, составленный из объектов класса `Person`, описанного в листинге 10.8.

Листинг 10.9. Класс-обработчик документа XML

```

import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;
import java.util.*;
import java.io.*;

public class NotebookHandler extends DefaultHandler{

    static final String JAXP_SCHEMA_LANGUAGE =
        "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

    static final String W3C_XML_SCHEMA =
        "http://www.w3.org/2001/XMLSchema";

    Person person;
    Address address;
    static Vector pers = new Vector();
    boolean inBirthday, inStreet, inCity, inZip,
        inWorkPhone, inHomePhone;

    public void startElement(String uri, String name,
        String qname, Attributes attrs)
        throws SAXException{

        if (qname.equals("name"))

            person = new Person(attrs.getValue("first"),
                attrs.getValue("second"), attrs.getValue("surname"));

```

```
else if (qname.equals("birthday"))
    inBirthday = true;
else if (qname.equals("address"))
    address = new Address();
else if (qname.equals("street"))
    inStreet = true;
else if (qname.equals("city")){
    inCity = true;
    if (attrs != null) address.setType(attrs.getValue("type"));
}else if (qname.equals("zip"))
    inZip = true;
else if (qname.equals("work"))
    inWorkPhone = true;
else if (qname.equals("home"))
    inHomePhone = true;
}

public void characters(char[] buf, int offset, int len)
    throws SAXException{

    String s = new String(buf, offset, len);

    if (inBirthday){
        person.setBirthday(s);
        inBirthday = false;
    }else if (inStreet){
        address.setStreet(s);
        inStreet = false;
    }else if (inCity){
        address.setCity(s);
        inCity = false;
    }else if (inZip){
```

```
        address.setZip(s);
        inZip = false;

    }else if (inWorkPhone){

        person.addWorkPhone(s);
        inWorkPhone = false;

    }else if (inHomePhone){

        person.addHomePhone(s);
        inHomePhone = false;

    }
}

public void endElement(String uri, String name,
    String qname) throws SAXException{

    if (qname.equals("address")){

        person.addAddress(address);
        address = null;

    }else if (qname.equals("person")){

        pers.add(person);
        person = null;

    }
}

public static void main(String[] args){

    if (args.length < 1){

        System.err.println("Usage: java NotebookHandler ntb.xml");
        System.exit(1);

    }

    try{
        NotebookHandler handler = new NotebookHandler();

        SAXParserFactory fact = SAXParserFactory.newInstance();

        fact.setNamespaceAware(true);
        fact.setValidating(true);

        SAXParser saxParser = fact.newSAXParser();

        saxParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

```
File f = new File(args[0]);

saxParser.parse(f, handler);

for (int k = 0; k < pers.size(); k++)
    System.out.println(((Person)pers.get(k)).getSurname());

} catch (SAXNotRecognizedException x) {

    System.err.println("Неизвестное свойство: " +
        JAXP_SCHEMA_LANGUAGE);
    System.exit(1);

} catch (Exception ex) {

    System.err.println(ex);
}

}

public void warning(SAXParseException ex) {

    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
        " col = " + ex.getColumnNumber());
}

public void error(SAXParseException ex) {

    System.err.println("Error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
        " col = " + ex.getColumnNumber());
}

public void fatalError(SAXParseException ex) {

    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() +
        " col = " + ex.getColumnNumber());
}

}
```

Связывание данных XML с объектами Java

В приведенном выше примере мы сами создали классы `Address` и `Person`, представляющие документ XML. Этот процесс можно автоматизировать. Поскольку структура документа XML четко определена, можно разработать

стандартные правила связывания данных XML с объектами Java и создать программные средства для их реализации.

Фирма Sun Microsystems разработала пакет интерфейсов и классов JAXB (Java API for XML Binding), облегчающий связывание данных. Домашняя страница пакета JAXB расположена по адресу <http://java.sun.com/xml/jaxb/>. На время написания данной книги пакет JAXB поставлялся только в составе средства разработчика Web-служб Java WSDP (Java Web Services Developer Pack). Набор WSDP можно скопировать с Web-страницы <http://java.sun.com/webservices/downloads/webservicespack.html>.

Для работы с пакетом JAXB анализируемый документ XML обязательно должен быть снабжен схемой XSD. Именно по схеме документа JAXB определяет, какие классы надо создавать. Затем JAXB читает документ, создает экземпляры классов и заполняет их информацией, взятой из документа XML.

В состав пакета JAXB входит компилятор `xjc` (XML-Java Compiler). Он просматривает схему XSD документа и строит по ней объекты Java в оперативной памяти, а также создает исходные файлы объектов Java. Например, после выполнения команды

```
$ xjc-d sources ntb.xsd
```

в которой `ntb.xsd` — файл с содержимым листинга 3.2 — в каталоге `sources` (по умолчанию в подкаталоге с именем `generated`) будут созданы файлы `Notebook.java`, `NotebookType.java`, `ObjectFactory.java` с описаниями соответствующих интерфейсов Java и методов создания из них объектов Java. В листинге 10.10 показано значительно сокращенное содержимое файла `NotebookType.java`. Полный файл включает содержимое файла `ntb.xsd`, аккуратно переписанное компилятором. Но и без того, как видите, компилятор весьма "словоохотлив" и снабжает текст подробными комментариями на английском языке.

Кроме этих файлов, компилятор формирует подкаталог `impl` с классами-реализациями созданных им интерфейсов.

Листинг 10.10. Интерфейсы, сгенерированные компилятором `xjc`

```
//
// This file was generated by the JavaTM Architecture for XML
// Binding(JAXB) Reference Implementation, v1.0.1-05/30/2003 05:06
// AM(java_re)-fcs
// See <a href="http://java.sun.com/xml/jaxb">
// http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost upon recompilation
// of the source schema.
// Generated on: 2003.07.27 at 11:28:38 MSD
//
```



```
package generated;

public interface NotebookType {

    /**
     * Gets the value of the Person property.
     *
     * This accessor method returns a reference to the live list,
     * not a snapshot. Therefore any modification you make to the
     * returned list will be present inside the JAXB object.
     * This is why there's any setter method for the Person property.
     *
     * For example, to add a new item, do as follows:
     * <pre>
     *     getPerson().add(newItem);
     * </pre>
     *
     * Objects of the following type(s) are allowed in the list
     * {@link generated.NotebookType.PersonType}
     */
    java.util.List getPerson();

    public interface PersonType {

        /**
         *
         * @return possible object is
         * {@link generated.NotebookType.PersonType.PhoneListType}
         */
        generated.NotebookType.PersonType.PhoneListType getPhoneList();

        /**
         *
         * @param value allowed object is
         * {@link generated.NotebookType.PersonType.PhoneListType}
         */
        void setPhoneList(
            generated.NotebookType.PersonType.PhoneListType value);

        /**
         *
         * @return possible object is
         * {@link java.lang.String}
         */
        java.lang.String getBirthday();
    }
}
```

```

/**
 *
 * @param value allowed object is
 * {@link java.lang.String}
 */
void setBirthday(java.lang.String value);

/**
 * Gets the value of the Address property.
 *
 * This accessor method returns a reference to the live list,
 * not a snapshot. Therefore any modification you make to the
 * returned list will be present inside the JAXB object.
 * This is why there's any setter method for the Address property.
 *
 * For example, to add a new item, do as follows:
 * <pre>
 *     getAddress().add(newItem);
 * </pre>
 *
 * Objects of the following type(s) are allowed in the list
 * {@link generated.NotebookType.PersonType.AddressType}
 */
java.util.List getAddress();

/**
 *
 * @return possible object is
 * {@link generated.NotebookType.PersonType.NameType}
 */
generated.NotebookType.PersonType.NameType getName();

/**
 *
 * @param value allowed object is
 * {@link generated.NotebookType.PersonType.NameType}
 */
void setName(generated.NotebookType.PersonType.NameType value);

public interface AddressType {

    /**
     *
     * @return possible object is

```

```

        * {@link java.math.BigInteger}
        */
        java.math.BigInteger getZip();

        /**
         *
         * @param value allowed object is
         * {@link java.math.BigInteger}
         */
        void setZip(java.math.BigInteger value);

        /**
         *
         * @return possible object is
         * {@link
* generated.NotebookType.PersonType.AddressType.CityType}
        */
        generated.NotebookType.PersonType.AddressType.CityType getCity();

        /**
         *
         * @param value allowed object is
         * {@link
* generated.NotebookType.PersonType.AddressType.CityType}
        */
        void setCity(
        generated.NotebookType.PersonType.AddressType.CityType value);

        /**
         *
         * @return possible object is
         * {@link java.lang.String}
         */
        java.lang.String getStreet();

        /**
         *
         * @param value allowed object is
         * {@link java.lang.String}
         */
        void setStreet(java.lang.String value);

        public interface CityType {

            /**
             *
             * @return possible object is

```

```

        * {@link java.lang.String}
        */
        java.lang.String getType();

        /**
         *
         * @param value allowed object is
         * {@link java.lang.String}
         */
        void setType(java.lang.String value);

        /**
         *
         * @return possible object is
         * {@link java.lang.String}
         */
        java.lang.String getValue();

        /**
         *
         * @param value allowed object is
         * {@link java.lang.String}
         */
        void setValue(java.lang.String value);
    }
}

public interface NameType {

    /**
     *
     * @return possible object is
     * {@link java.lang.String}
     */
    java.lang.String getFirst();

    /**
     *
     * @param value allowed object is
     * {@link java.lang.String}
     */
    void setFirst(java.lang.String value);

    /**
     *
     * @return possible object is

```

```
    * {@link java.lang.String}
    */
    java.lang.String getSurname();

    /**
     *
     * @param value allowed object is
     * {@link java.lang.String}
     */
    void setSurname(java.lang.String value);

    /**
     *
     * @return possible object is
     * {@link java.lang.String}
     */
    java.lang.String getSecond();

    /**
     *
     * @param value allowed object is
     * {@link java.lang.String}
     */
    void setSecond(java.lang.String value);
}

public interface PhoneListType {

    /**
     * Gets the value of the HomePhone property.
     *
     * This accessor method returns a reference to the live list,
     * not a snapshot. Therefore any modification you make to the
     * returned list will be present inside the JAXB object.
     * This is why there's any setter method for the HomePhone
     * property.
     *
     * For example, to add a new item, do as follows:
     * <pre>
     *     getHomePhone().add(newItem);
     * </pre>
     *
     * Objects of the following type(s) are allowed in the list
     * {@link java.lang.String}
     */
    java.util.List getHomePhone();
```

```

/**
 * Gets the value of the WorkPhone property.
 *
 * This accessor method returns a reference to the live list,
 * not a snapshot. Therefore any modification you make to the
 * returned list will be present inside the JAXB object.
 * This is why there's any setter method for the WorkPhone
 * property.
 *
 * For example, to add a new item, do as follows:
 * <pre>
 *     getWorkPhone().add(newItem);
 * </pre>
 *
 * Objects of the following type(s) are allowed in the list
 * {@link java.lang.String}
 */
java.util.List getWorkPhone();
}
}
}

```

Кроме схемы XSD, компилятору `xjc` можно дать дополнительное описание схемы документа на специальном языке (binding language) — реализации XML. Это язык, похожий на язык XSD, но с меньшими возможностями. По-видимому, в дальнейшем он будет расширен или заменен языком XSD. Если дополнительное описание записано в файл `ntb.xjs` (XML-Java binding schema), то вызов компилятора будет выглядеть следующим образом:

```
$ xjc-d sources ntb.dtd ntb.xjs
```

Созданные компилятором `xjc` исходные файлы обычным образом, с помощью компилятора `javac`, компилируются в классы Java.

Процесс переноса информации из документа XML в полученные после компиляции объекты данных называется *разборкой* (unmarshalling) документа. Обратный процесс создания документа XML из объектов данных называется *сборкой* (marshalling) документа. Методы сборки и разборки описаны в одноименных интерфейсах `javax.xml.bind.Marshaller` и `javax.xml.bind.Unmarshaller` пакета JAXB.

В процессе разборки JAXB строит дерево объектов данных в оперативной памяти. После того как дерево построено, все связи узлов дерева проверяются методами `validate()` интерфейса `javax.xml.bind.Validator`.

Получив объекты данных, можно перенести в них содержимое документа XML методом `unmarshal()`, который создает дерево объектов, или, наоборот, записать объекты Java в документы XML методом `marshal()`. Эти методы описаны в интерфейсах `Unmarshaller` и `Marshaller`. Объекты этих типов создаются методами

```
Unmarshaller createUnmarshaller();
Marshaller createMarshaller();
```

класса `JAXBContext`. Экземпляры класса `JAXBContext` создаются одним из статических методов

```
static JAXBContext newInstance(String context);
static JAXBContext newInstance(String context, ClassLoader class);
```

Аргумент `context` этих методов представляет собой список пакетов Java, созданных компилятором `xjc`, перечисленных через двоеточие.

Все названные действия по связыванию данных с объектами Java при помощи пакета JAXB приведены в листинге 10.11. В нем использованы типы данных, порожденные JAXB. Они взяты из листинга 10.10.

Листинг 10.11. Связывание данных с помощью JAXB

```
import java.io.*;
import java.util.*;
import javax.xml.bind.*;
import generated.*;

public class UnmarshallNtb{

    public static void main(String[] args){

        try{
            JAXBContext jc = JAXBContext.newInstance("generated");

            Unmarshaller u = jc.createUnmarshaller();

            Notebook ntb = (Notebook)u.unmarshal(
                new FileInputStream("notebook.xml"));

            List persons = ntb.getPerson();

            NotebookType.PersonType person =
                (NotebookType.PersonType)persons.get(0);

            prPerson(person);
```

```
        } catch (JAXBException je) {
            je.printStackTrace();
        } catch (Exception ioe) {
            ioe.printStackTrace();
        }
    }

    public static void prPerson(NotebookType.PersonType person) {

        NotebookType.PersonType.NameType name = person.getName();

        System.out.println("Имя:      " + name.getFirst());
        System.out.println("Отчество: " + name.getSecond());
        System.out.println("Фамилия:  " + name.getSurname());
    }
}
```

Объекты данных JDO

Задачу связывания данных естественно обобщить — связывать объекты Java не только с документами XML, но и с текстовыми файлами, реляционными или объектными базами данных, другими хранилищами данных.

Фирма Sun Microsystems опубликовала спецификацию JDO и разработала интерфейсы для работы с JDO. Их можно посмотреть по адресу <http://java.sun.com/products/jdo/>.

Спецификация JDO рассматривает более широкую задачу связывания данных, полученных не только из документа XML, но и из любого источника данных, называемого *информационной системой предприятия* EIS (Enterprise Information System). Спецификация описывает два набора классов и интерфейсов:

- JDO SPI (JDO Service Provider Interface) — вспомогательные классы и интерфейсы, которые следует реализовать в сервере приложений для обращения к источнику данных, создания объектов, обеспечения их сохранности, выполнения транзакций, проверки прав доступа к объектам; эти классы и интерфейсы составляют пакет `javax.jdo.spi`;
- JDO API (JDO Application Programming Interface) — интерфейсы, предоставляемые пользователю для доступа к объектам, управления транзакциями, создания и удаления объектов; эти интерфейсы собраны в пакет `javax.jdo`.

Фирма Sun выпустила реализацию пакета JDO, которая доступна на сайте <http://java.sun.com/products/jdo/>. Свои реализации сделали и другие фирмы.

Фирма Prism Technologies, <http://www.prismtechnologies.com/>, выпускает OpenFusion JDO как составную часть своего основного продукта OpenFusion.

Фирма SolarMetric, <http://www.solarmetric.com/>, выпускает свою реализацию спецификации JDO под названием Kodo JDO. Ее можно встроить в серверы приложений WebLogic, WebSphere, JBoss.

Фирма ObjectFrontier, <http://www.objectfrontier.com>, выпустила уже третью версию своего продукта Frontier Suite for JDO. Его можно встроить в популярную интегрированную среду разработки Eclipse.

Фирма Riflexo анонсировала реализацию JDO, названную JCredo, <http://www.jcredo.com/>. Ее можно встроить во многие Java-средства разработки, включая Borland JBuilder и Eclipse, а также в большинство серверов приложений.

Есть и другие разработки, их обзор можно посмотреть на сайте разработчиков JDO <http://jdocentral.com/>.

Некоторые фирмы, не дожидаясь выхода спецификации JDO, разработали свои реализации JDO, более или менее соответствующие спецификации. Наиболее известна свободно распространяемая разработка фирмы Exolab, названная Castor. Ее можно посмотреть по адресу <http://castor.exolab.org/>.

С помощью Castor можно предельно упростить связывание данных. Например, разборка, т. е. создание объекта Java из простого документа XML, если отвлечься от проверок и обработки исключительных ситуаций, выполняется одним действием:

```
Person person = (Person)Unmarshaller.unmarshal(  
    Person.class, new FileReader("person.xml"));
```

Обратная процедура сохранения объекта Java в виде документа XML (сборка) в самом простом случае выглядит так:

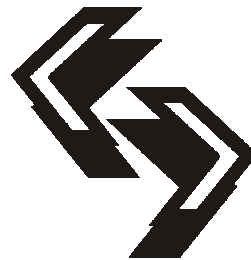
```
Marshaller.marshall(person, new FileWriter("person.xml"));
```

В более сложных случаях надо написать файл XML, аналогичный схеме XSD, с указаниями по связыванию данных (mapping file).

Вопросы для самопроверки

1. В чем заключается сканирование документа XML?
2. Какова цель грамматического разбора документа XML?
3. На какие две группы делятся программы-обработчики XML?
4. Что называется событием в системе SAX?
5. Что делает SAX-анализатор при возникновении события?
6. Какую роль играет объект типа XMLReader в SAX-анализаторе?

7. Что делает класс-обработчик событий?
8. Какие методы надо реализовать при создании класса-обработчика?
9. Каким образом запускается SAX-анализатор?
10. Как передать SAX-анализатору исходный документ XML и экземпляр класса-обработчика событий?
11. Как создать цепочку SAX-анализаторов?
12. Что такое объекты данных JDO и для чего они нужны?
13. Что такое связывание данных с объектами JDO?
14. Что такое сборка и разборка документа XML?
15. Какие средства применяются для автоматического создания объектов данных?



Обработка документов при помощи DOM

В начале предыдущей главы мы обсуждали способы анализа и обработки документов XML. Там же мы подробно рассмотрели способ анализа, основанный на событиях. Этот способ не требует больших ресурсов от компьютера, обработка документа идет быстро, но документ просматривается только один раз, от начала к концу. Это неудобно в тех случаях, когда надо изменить документ: вставить новый элемент, изменить состав атрибутов, содержимое одних элементов в зависимости от содержимого других элементов. Определенный результат дает связывание элементов XML с объектами данных, но оно значительно замедляет работу SAX-анализатора, лишая его главного достоинства — быстроты.

У второго способа анализа документов XML, основанного на построении дерева документа в оперативной памяти, тоже есть свои достоинства и недостатки, противоположные названным выше. Дерево документа строится довольно долго, занимает много места в оперативной памяти, но зато потом его можно просматривать много раз во всех направлениях и изменять каким угодно образом.

Консорциум W3C разработал спецификации и набор интерфейсов DOM API (Document Object Model Application Programming Interface), которые описывают анализатор, основанный на построении дерева. Их можно посмотреть на сайте этого проекта <http://www.w3.org/DOM/>. Методами интерфейсов из набора DOM API документ XML можно загрузить в оперативную память в виде дерева объектов, описанных в соответствующих интерфейсах `Element`, `Attr`, `Comment`, `Entity`, `Notation`, `CDATASection`, `Node`, `Text` и др. Это позволяет не только анализировать документ анализаторами, основанными на структуре дерева, но и менять дерево, добавляя или удаляя объекты. Кроме того, можно обращаться непосредственно к каждому объекту в дереве и не только читать, но и изменять информацию, хранящуюся в нем. Но все это требует значительного объема оперативной памяти для загрузки большого дерева и времени для поиска в нем.

Интерфейсы DOM API описаны в спецификациях, которые можно найти на Web-странице консорциума W3C <http://www.w3.org/DOM/DOMTR>. У спецификаций DOM есть несколько уровней, детализирующих правила создания деревьев DOM и их анализа. Каждая спецификация описывает один модуль DOM API. В каждом DOM-анализаторе обязательно должен быть реализован модуль DOM API Core, остальные модули могут быть представлены в той или иной полноте.

Первый уровень, Level 1, описывает построение дерева DOM и правила его обхода, не учитывающие особенности конкретных языков. Этот уровень ориентирован только на XML и HTML. В нем есть только модуль Core и модуль HTML.

Второй уровень, Level 2, вводит пространства имен, итераторы, фильтры, обработку событий, представления (views), определяет новые модули для работы с тонкостями языков XML, HTML, CSS.

Третий уровень, Level 3, находится в черновой стадии разработки. Он описывает тонкости проверяющих XML-анализаторов, способы записи дерева в файл (сериализацию) и чтения его из файла (десериализацию), правила связи с языком XPath. Окончательный состав третьего уровня DOM API может значительно измениться в любую сторону.

Каждый следующий уровень вносит в интерфейсы DOM API новые типы, константы и методы, поэтому его можно считать новой версией DOM API.

Интерфейсы DOM API написаны на языке IDL (Interface Definition Language), разработанном в свое время группой OMG (Object Management Group) для системы CORBA (Common Object Request Broker Architecture). Этот язык выбран потому, что он достаточно точно описывает интерфейсы и не зависит от какого-либо конкретного языка программирования. Тем не менее, интерфейсы DOM API можно реализовать на любом объектно-ориентированном языке программирования. Консорциум W3C в приложениях к спецификациям DOM API дал описание всех интерфейсов на языке Java. Кроме того, уже есть множество реализаций DOM API на языках C++, Python, Perl, JavaScript, Scheme, Lingo.

DOM-анализатор фирмы Sun

Фирма Sun Microsystems поместила основные интерфейсы DOM Level 2 Core, написанные на языке Java, в пакет `org.w3c.dom`, входящий в стандартный состав набора JAXP, о котором уже шла речь в начале предыдущей главы.

Интерфейсы для работы с таблицами стилей вообще и CSS в частности — модуль DOM Level 2 Style — помещены в пакеты `org.w3c.dom.css` и `org.w3c.dom.stylesheets`.

Интерфейсы, связанные с обработкой документов HTML, — модуль DOM Level 2 HTML — собраны в пакет `org.w3c.dom.html`, а описание представлений — модуль DOM Level 2 Views — в пакет `org.w3c.dom.views`.

Интерфейсы, описывающие правила обхода дерева, — модуль DOM Level 2 Traversal — составляют пакет `org.w3c.dom.traversal`.

События — модуль DOM Level 2 Events — включены в пакет `org.w3c.dom.events`.

В том же наборе JAXP, в классе `javax.xml.parsers.DocumentBuilder`, реализован DOM-анализатор документов XML.

Экземпляр DOM-анализатора, т. е. класса `DocumentBuilder`, конструируется фабричным методом `newDocumentBuilder()` класса `DocumentBuilderFactory`. Фабрика же создается своим собственным методом `newInstance()`. Все вместе выглядит так:

```
DocumentBuilderFactory fact =
    DocumentBuilderFactory.newInstance();

DocumentBuilder builder = fact.newDocumentBuilder();
```

После конструирования DOM-анализатора `builder` остается только применить один из его методов `parse()` к документу XML:

```
Document doc = builder.parse("ntb.xml");
```

Все, дерево построено!

Метод `parse()` строит дерево объектов и возвращает ссылку на него в виде объекта типа `Document`. В классе `DocumentBuilder` есть пять методов `parse()`, позволяющих загрузить файл с адреса URI, из входного потока, как объект класса `File`, или из источника класса `InputStream`.

Сразу же после создания фабрики DOM-анализаторов `fact`, ее можно настроить перечисленными далее методами.

- ❑ `void setValidating(boolean validating);` — предписывает фабрике создавать проверяющие (`true`) или не проверяющие (`false`) DOM-анализаторы. По умолчанию фабрика создает не проверяющие анализаторы.
- ❑ `void setNamespaceAware(boolean awareness);` — включает (`true`) или выключает (`false`) в создаваемых анализаторах использование уточненных имен. По умолчанию уточненные имена не используются.

❑ Методы

```
void setIgnoringComments(boolean ignoreComments);
void setIgnoringElementContentWhitespace(boolean whitespace);
```

показывают, будет ли анализатор включать (`true`) или нет (`false`) комментарии и пробельные символы в дерево объектов DOM.

- ❑ `void setCoalescing(boolean coalescing);` — с аргументом, равным `true`, заставляет анализаторы, создаваемые фабрикой, преобразовывать секции CDATA в текстовые узлы. Если соседний узел тоже текстовый, то полученный текстовый узел сливается с ним.
- ❑ `void setExpandEntityReferences(boolean expandEntityRef);` — с аргументом, равным `true`, заставляет создаваемый анализатор заменять ссылки на сущности их значениями.
- ❑ `void setAttribute(String name, Object value);` — устанавливает специфические свойства анализатора. Некоторые такие свойства использованы в листинге 11.3.

Основные интерфейсы DOM API

Дерево DOM состоит из узлов, типы и общие свойства которых описаны интерфейсом `Node`. Все дерево в целом описывается интерфейсом `Document`, расширяющим интерфейс `Node`. У интерфейса `Node` есть еще один наследник — интерфейс `Element`, описывающий лист дерева, соответствующий элементу документа XML. Как видно из структуры наследования этих интерфейсов, и само дерево, и каждый его лист считаются узлами дерева. Это позволяет вкладывать деревья друг в друга и рекурсивно просматривать под-деревья.

Каждый атрибут элемента XML описывается интерфейсом `Attr`. Еще несколько интерфейсов — `CDATASection`, `Comment`, `Text`, `Entity`, `EntityReference`, `ProcessingInstruction`, `Notation` — описывают разные конструкции XML.

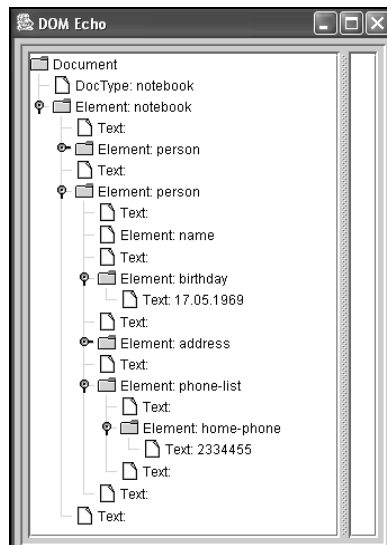


Рис. 11.1. Дерево объектов документа XML

На рис. 11.1 показано начало дерева объектов, построенного по документу, приведенному в листинге 1.2. Обратите внимание на то, что текст, находящийся в теле элемента, хранится в отдельном текстовом узле дерева — потомке узла элемента. Для каждого атрибута элемента тоже создается отдельный узел.

Интерфейс *Node*

Интерфейс *Node* описывает общие свойства всех узлов дерева DOM. Он отмечает тип узла одной из следующих констант:

- `ATTRIBUTE_NODE` — узел типа `Attr`, содержит атрибут элемента;
- `CDATA_SECTION_NODE` — узел типа `CDATASection`, содержит данные типа `CDATA`;
- `COMMENT_NODE` — узел типа `Comment`, содержит комментарий;
- `DOCUMENT_FRAGMENT_NODE` — в узле типа `DocumentFragment` находится фрагмент документа;
- `DOCUMENT_NODE` — корневой узел типа `Document`;
- `DOCUMENT_TYPE_NODE` — узел типа `Document`;
- `ELEMENT_NODE` — узел является листом дерева типа `Element`;
- `ENTITY_NODE` — в узле типа `Entity` хранится сущность `ENTITY`;
- `ENTITY_REFERENCE_NODE` — в узле типа `EntityReference` хранится ссылка на сущность;
- `NOTATION_NODE` — в узле хранится нотация типа `Notation`;
- `PROCESSING_INSTRUCTION_NODE` — узел типа `ProcessingInstruction`, содержит инструкцию по обработке;
- `TEXT_NODE` — в узле типа `Text` хранится текст.

Методы интерфейса *Node* описывают действия с узлом дерева. Узнать тип узла можно методом

```
short getNodeType();
```

Имя узла возвращает метод

```
String getNodeName();
```

Значение, хранящееся в узле, можно получить методом

```
String getNodeValue();
```

Проверить, есть ли атрибуты у элемента XML, хранящегося в узле в виде объекта типа `NamedNodeMap`, если это узел типа `Element`, позволяет метод

```
boolean hasAttributes();
```

Атрибуты возвращает метод

```
NamedNodeMap getAttributes();
```

Метод возвращает null, если у элемента нет атрибутов.

Метод

```
boolean hasChildNodes();
```

проверяет, есть ли у данного узла узлы-потомки. Если они есть, то можно получить их список в виде объекта типа `NodeList` методом

```
NodeList getChildNodes();
```

Первый и последний узлы в этом списке можно получить методами

```
Node getFirstChild();
```

```
Node getLastChild();
```

Родительский узел можно получить методом

```
Node getParentNode();
```

а соседние узлы с тем же предком, что и данный узел — методами

```
Node getPreviousSibling();
```

```
Node getNextSibling();
```

Можно получить и ссылку на весь документ методом

```
Document getOwnerDocument();
```

Следующие методы позволяют изменить дерево объектов.

Добавить новый узел-потомок `newChild` можно методом

```
Node appendChild(Node newChild);
```

Вставить новый узел-потомок `newChild` перед существующим потомком `refChild` можно методом

```
Node insertBefore(Node newChild, Node refChild);
```

Заменить один узел-потомок `oldChild` новым узлом `newChild` можно методом

```
Node replaceChild(Node newChild, Node oldChild);
```

Наконец, удалить узел-потомок можно методом

```
Node removeChild(Node child);
```

Интерфейс *Document*

Интерфейс `Document` добавляет к методам своего предка `Node` методы работы с документом в целом.

Метод

```
DocumentType getDocType();
```

возвращает общие сведения о документе в виде объекта типа `DocumentType`.

Методы `getName()`, `getEntities()`, `getNotations()` и другие методы интерфейса `DocumentType` возвращают конкретные сведения о документе.

Метод

```
Element getDocumentElement();
```

возвращает корневой элемент дерева объектов, а методы

```
NodeList getElementsByTagName(String name);  
NodeList getElementsByTagNameNS(String uri, String qname);  
Element getElementById(String id);
```

возвращают все элементы с указанным именем `tag` без префикса или с префиксом, а также элемент, определяемый значением атрибута с именем `ID`.

Несколько методов позволяют изменить структуру и содержимое дерева объектов.

Создать новый пустой элемент по его локальному имени или по уточненному имени с префиксом можно методами

```
Element createElement(String name);  
Element createElementNS(String uri, String name);
```

Создать узел типа `CDATA_SECTION_NODE` можно методом

```
CDATASection createCDATASection(String name);
```

Создать узел типа `ENTITY_REFERENCE_NODE` можно методом

```
EntityReference createEntityReference(String name);
```

Создать узел типа `PROCESSING_INSTRUCTION_NODE` можно методом

```
ProcessingInstruction createProcessingInstruction(String name);
```

Создать узел типа `TEXT_NODE` можно методом

```
TextNode createTextNode(String name);
```

Создать узел-атрибут с именем `name` можно методом

```
Attr createAttribute(String name);  
Attr createAttributeNS(String uri, String name);
```

Узел-комментарий создается методом

```
Comment createComment(String comment);
```

Наконец, можно создать пустой документ — фрагмент данного документа с целью его дальнейшего заполнения

```
DocumentFragment createDocumentFragment();
```

Вставить созданный узел, а значит, и все его поддерево, в дерево документа, можно методом

```
Node importNode(Node importedNode, boolean deep);
```

Этим методом можно соединить два дерева объектов. Если второй аргумент равен `true`, то рекурсивно вставляется все поддерево.

Интерфейс *Element*

Интерфейс `Element` добавляет к методам своего предка `Node` методы работы с атрибутами элемента XML и методы, позволяющие обратиться к вложенным элементам. Только один метод

```
String getTagName();
```

дает сведения о самом элементе, а именно, он возвращает имя элемента.

Прежде чем получить значение атрибута с именем `name`, надо проверить его наличие методами

```
boolean hasAttribute(String name);  
boolean hasAttributeNS(String uri, String name);
```

Второй из этих методов учитывает пространство имен с идентификатором `uri`, записанным в виде строки URI. Имя `name` во втором методе должно быть уточненным, с префиксом.

Получить атрибут в виде объекта типа `Attr` или его значение в виде строки по имени `name` с учетом префикса или без него можно методами

```
Attr getAttributeNode(String name);  
Attr getAttributeNodeNS(String uri, String name);  
String getAttribute(String name);  
String getAttributeNS(String uri, String name);
```

Удалить атрибут можно методами

```
Attr removeAttributeNode(Attr name);  
void removeAttribute(String name);  
void removeAttributeNS(String uri, String name);
```

Установить значение атрибута можно методами

```
void setAttribute(String name, String value);  
void setAttributeNS(String uri, String name, String value);
```

Добавить атрибут в качестве потомка можно методами

```
Attr setAttributeNode(String name);  
Attr setAttributeNodeNS(Attr name);
```

Два метода позволяют получить список узлов-потомков:

```
NodeList getElrmentsByTagName(String name);  
NodeList getElrmentsByTagNameNS(String uri, String name);
```

Итак, методы перечисленных интерфейсов позволяют перемещаться по дереву, менять его структуру, просматривать информацию, хранящуюся в узлах и листьях дерева и изменять ее.

Обход дерева DOM

Обход дерева и просмотр его узлов — это наиболее часто встречающееся действие при работе с деревьями. Методы обхода описаны в модуле DOM Level 2 Traversal в интерфейсах `DocumentTraversal`, `NodeFilter`, `NodeIterator`, `TreeWalker`. В наборе JAXP эти интерфейсы составляют пакет `org.w3c.dom.traversal`.

Мы рассмотрим интерфейсы обхода дерева чуть позднее, а пока, в качестве первого примера использования DOM API, организуем обход "вручную". Напишем метод `iter(Node node)`, рекурсивно просматривающий поддерево, начинающееся с узла `node`, определяющий тип каждого узла и выводящий на консоль его содержимое. Последнее действие выполняется специально написанным методом `prNode(Node node)`.

Программа, выполняющая эти действия, приведена в листинге 11.1. При ее запуске надо набрать в командной строке имя файла с документом XML и название элемента, с которого начинается обход, например:

```
$ java TreeContentDOM notebook.xml notebook
```

Листинг 11.1. Вывод на консоль содержимого дерева DOM

```
import org.w3c.dom.*;  
import javax.xml.parsers.*;  
import org.xml.sax.*;  
  
class ErrorHandler implements ErrorHandler{  
    public void warning(SAXParseException ex){  
        System.err.println("Warning: " + ex);  
        System.err.println("line = " + ex.getLineNumber() +
```

```
        " col = " + ex.getColumnNumber());
    }

    public void error(SAXParseException ex) {

        System.err.println("Error: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            " col = " + ex.getColumnNumber());
    }

    public void fatalError(SAXParseException ex) {

        System.err.println("Fatal error: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            " col = " + ex.getColumnNumber());
    }
}

public class TreeContentDOM{

    private void prNode(Node node) {

        System.out.print(" nodeName= " + node.getNodeName());

        String val = node.getNamespaceURI();

        if (val != null)
            System.out.print(" uri= " + val);

        val = node.getPrefix();

        if (val != null)
            System.out.print(" pre= " + val);

        val = node.getLocalName();

        if (val != null)
            System.out.print(" local= " + val);

        val = node.getNodeValue();

        if (val != null) {

            System.out.print(" nodeValue= ");

            if (val.trim().equals(""))
                System.out.print("[WS]");
            else System.out.print(node.getNodeValue());
        }
    }
}
```

```
        System.out.println();
    }

    private void iter(Node node) {

        int type = node.getNodeType();

        switch (type) {

            case Node.ATTRIBUTE_NODE:
                System.out.print("ATTR:");
                prNode(node);
                break;

            case Node.CDATA_SECTION_NODE:
                System.out.print("CDATA:");
                prNode(node);
                break;

            case Node.COMMENT_NODE:
                System.out.print("COMM:");
                prNode(node);
                break;

            case Node.DOCUMENT_FRAGMENT_NODE:
                System.out.print("DOC_FRAG:");
                prNode(node);
                break;

            case Node.DOCUMENT_NODE:
                System.out.print("DOC:");
                prNode(node);
                break;

            case Node.DOCUMENT_TYPE_NODE:
                System.out.print("DOC_TYPE:");
                prNode(node);

                NamedNodeMap nodeMap = ((DocumentType) node).getEntities();

                for (int i = 0; i < nodeMap.getLength(); i++) {
                    Entity entity = (Entity) nodeMap.item(i);
                    iter(entity);
                }
                break;

            case Node.ELEMENT_NODE:
                System.out.print("ELEM:");
                prNode(node);
```

```
NamedNodeMap atts = node.getAttributes();

for (int i = 0; i < atts.getLength(); i++){
    Node att = atts.item(i);
    iter(att);
}
break;

case Node.ENTITY_NODE:
    System.out.print("ENT:");
    prNode(node);
    break;

case Node.ENTITY_REFERENCE_NODE:
    System.out.print("ENT_REF:");
    prNode(node);
    break;

case Node.NOTATION_NODE:
    System.out.print("NOTATION:");
    prNode(node);
    break;

case Node.PROCESSING_INSTRUCTION_NODE:
    System.out.print("PROC_INST:");
    prNode(node);
    break;

case Node.TEXT_NODE:
    System.out.print("TEXT:");
    prNode(node);
    break;

default:
    System.out.print("UNSUPPORTED NODE: " + type);
    prNode(node);
    break;
}

for (Node child = node.getFirstChild(); child != null;
     child = child.getNextSibling())
    iter(child);
}

public static void main(String[] args) throws Exception{

    if (args.length != 2){
```

```
        System.err.println(
            "Usage: java TreeContentDOM <fileName> <elemName>");

        System.exit(-1);
    }

    DocumentBuilderFactory fact =
        DocumentBuilderFactory.newInstance();

    fact.setNamespaceAware(true);

    fact.setValidating(true);

    DocumentBuilder builder = fact.newDocumentBuilder();

    builder.setErrorHandler(new ErrorHandler());

    Document doc = builder.parse(args[0]);

    NodeList list = doc.getElementsByTagName(args[1]);

    int n = list.getLength();

    if (n == 0) {

        System.err.println("Элемент пуст.");
        System.exit(-1);
    }

    TreeContentDOM tcd = new TreeContentDOM();

    for (int k = 0; k < n; k++) tcd.iter(list.item(k));
}
}
```

Дерево объектов можно вывести на экран дисплея, например, как графическое дерево `JTree` — компонент графической библиотеки Java, называемой `Swing`. Именно так сделано на рис. 11.1. Для вывода применена программа `DomEcho` из электронного учебника "Web Services Tutorial". Исходный текст программы слишком велик, чтобы приводить его здесь, но его можно посмотреть по адресу <http://java.sun.com/webservices/tutorial.html>. В состав XML-анализатора `Xerces` в качестве примера анализа документа в раздел `samples/ui/` входит программа `TreeView`, которая тоже показывает дерево объектов в виде дерева `JTree` библиотеки `Swing`.

Упражнение

Упростите метод `iter()`, оставив вывод на консоль только текстовых узлов.

Обход дерева методами DOM API

Посмотрим, насколько облегчается задача обхода дерева средствами интерфейсов из пакета `org.w3c.dom.traversal`. Методы обхода описаны в двух интерфейсах — `NodeIterator` и `TreeWalker`.

Интерфейс *NodeIterator*

Интерфейс `NodeIterator` описывает методы последовательного обхода дерева и представляет его в виде линейного списка, по которому можно перемещаться в прямом направлении и возвращаться назад.

Метод

```
Node getRoot();
```

возвращает ссылку на узел, с которого начинается обход поддерева. Методы

```
Node nextNode();
```

```
Node previousNode();
```

перемещают итератор к следующему или предыдущему узлу и возвращают ссылку на него. Методы

```
int getWhatToShow();
```

```
NodeFilter getFilter();
```

возвращают значение параметра `whatToShow` и ссылку на объект-фильтр.

Метод

```
boolean getExpandEntityReferences();
```

показывает значение флага `entityRef`.

Последний метод

```
void detach();
```

переводит итератор в состояние `INVALID` и освобождает ресурсы, занятые им.

Интерфейс *TreeWalker*

Интерфейс `TreeWalker` описывает методы обхода дерева с учетом его структуры.

Метод

```
Node getRoot();
```

возвращает ссылку на узел, с которого начинается обход поддерева.

Метод

```
Node getCurrentNode();
```

возвращает ссылку на текущий узел.

Методы

```
Node nextNode();  
Node previousNode();  
Node parentNode();  
Node firstChild();  
Node lastChild();  
Node previousSibling();  
Node nextSibling();  
void setCurrentNode(Node currentNode);
```

перемещают итератор к следующему, предыдущему, родительскому узлу, к первому или последнему узлу-потомку или к предыдущему или следующему узлу в ряду соседних узлов. Все эти методы возвращают ссылку на полученный узел. Методы

```
int getWhatToShow();  
NodeFilter getFilter();
```

возвращают значение параметра `whatToShow`, полученного из констант интерфейса `NodeFilter`, и ссылку на объект-фильтр. Метод

```
boolean getExpandEntityReferences();
```

показывает значение флага `entityRef`.

Объекты типа `NodeIterator` и `TreeWalker` создаются методами интерфейса `DocumentTraversal`.

Интерфейс *DocumentTraversal*

Интерфейс `DocumentTraversal` описывает два метода создания объектов типа `NodeIterator` и `TreeWalker`. Это методы

```
NodeIterator createNodeIterator(Node root, int whatToShow,  
    NodeFilter filter, boolean entityRef);  
  
TreeWalker createTreeWalker(Node root, int whatToShow,  
    NodeFilter filter, boolean entityRef);
```

Созданные объекты будут рекурсивно просматривать поддерево, начиная с элемента `root`. Узлы для просмотра отбираются объектом `filter`, а их типы закодированы числом `whatToShow`. Последний аргумент `entityRef` указывает, раскрывать ссылки на сущности или нет.

Перед тем как создавать итераторы, надо создать объект типа `DocumentTraversal`. Интерфейс `DocumentTraversal` реализуется тем же объектом, что и интерфейс `Document`. Поэтому нужный объект создается приведением типа:

```
DocumentBuilderFactory fact =
    DocumentBuilderFactory.newInstance();

DocumentBuilder builder = fact.newDocumentBuilder();

Document doc = builder.parse(args[0]);

DocumentTraversal dt = (DocumentTraversal)doc;

NodeIterator iter = dt.createNodeIterator(
    someElement, NodeFilter.SHOW_ALL, null, true);
```

Здесь надо сделать одно замечание. Не все DOM-анализаторы реализуют интерфейсы пакета `org.w3c.dom.traversal`. Поэтому имеет смысл проверить эту возможность. Это можно сделать так:

```
DOMImplementation imp = builder.getDOMImplementation();

System.out.println(imp.hasFeature("Traversal", "2.0"));
```

Если вы получите на консоли значение `true`, то анализатор реализует пакет `org.w3c.dom.traversal`, если нет, то надо подыскать более свежую версию пакета JAXP.

Параметр `whatToShow` получается как комбинация констант, определенных в интерфейсе `NodeFilter`.

Интерфейс *NodeFilter*

Интерфейс `NodeFilter` вводит константы, определяющие значение параметра `whatToShow`, используемого при создании объектов-итераторов:

```
int SHOW_ALL = 0xFFFFFFFF;
int SHOW_ELEMENT = 0x00000001;
int SHOW_ATTRIBUTE = 0x00000002;
int SHOW_TEXT = 0x00000004;
int SHOW_CDATA_SECTION = 0x00000008;
int SHOW_ENTITY_REFERENCE = 0x00000010;
int SHOW_ENTITY = 0x00000020;
int SHOW_PROCESSING_INSTRUCTION = 0x00000040;
int SHOW_COMMENT = 0x00000080;
int SHOW_DOCUMENT = 0x00000100;
```

```
int SHOW_DOCUMENT_TYPE = 0x00000200;
int SHOW_DOCUMENT_FRAGMENT = 0x00000400;
int SHOW_NOTATION = 0x00000800;
```

Например, если мы хотим просматривать только элементы и их атрибуты, то создаем итератор следующим образом:

```
NodeIterator ni = NodeIterator.createNodeIterator(root,
    NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_ATTRIBUTE, null, false);
```

В интерфейсе `NodeFilter` описан всего один метод

```
short acceptNode(Node node);
```

определяющий, пропускать через фильтр узел, ссылка `node` на который передается методу, или нет. Метод возвращает одну из трех следующих констант:

```
short FILTER_ACCEPT = 1;
short FILTER_REJECT = 2;
short FILTER_SKIP = 3;
```

Первая константа `FILTER_ACCEPT` предписывает пропускать узел `node` через фильтр. Вторая — `FILTER_REJECT` — отвергает узел `node` и все его вложенные узлы, а третья — `FILTER_SKIP` — отвергает только данный узел `node`. Две последние возможности различаются итератором типа `TreeWalker`, но не различаются итератором `NodeIterator`, не вникающим в структуру дерева. Он отвергает только данный узел `node`.

Разработчик должен сам задать этот метод в собственной реализации интерфейса `NodeFilter`. Например:

```
public class MyFilter implements NodeFilter{

    public short acceptNode(Node node){

        return (node.getNodeType() == Node.COMMENT_NODE) ?
            NodeFilter.FILTER_ACCEPT :
            NodeFilter.FILTER_SKIP;
    }
}
```

После этого можно определить итератор с этим фильтром:

```
MyFilter mf = new MyFilter();

NodeIterator ni = NodeIterator.createNodeIterator(root,
    NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_ATTRIBUTE, mf, false);
```

После этого DOM-анализатор сам будет обращаться к объекту-фильтру `mf` для отбора указанных узлов.

Теперь, вооружившись стандартными методами обхода дерева DOM API, мы можем переписать листинг 11.1, убрав из него метод `iter()` и дописав необходимые операторы в метод `main()`. Результат записан в листинге 11.2.

Листинг 11.2. Обход дерева методами DOM API

```
import org.w3c.dom.*;
import org.w3c.dom.traversal.*;
import javax.xml.parsers.*;
import org.xml.sax.*;

class ErrorHandler implements ErrorHandler{

    public void warning(SAXParseException ex){

        System.err.println("Warning: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            "   col = " + ex.getColumnNumber());
    }

    public void error(SAXParseException ex){

        System.err.println("Error: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            "   col = " + ex.getColumnNumber());
    }

    public void fatalError(SAXParseException ex){

        System.err.println("Fatal error: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            "   col = " + ex.getColumnNumber());
    }
}

public class TreeContentDOMAPI{

    private void prNode(Node node){

        System.out.print(" nodeName= " + node.getNodeName());

        String val = node.getNamespaceURI();

        if (val != null)
            System.out.print(" uri= " + val);
    }
}
```

```
        val = node.getPrefix();

        if (val != null)
            System.out.print(" pre= " + val);

        val = node.getLocalName();

        if (val != null)
            System.out.print(" local= " + val);

        val = node.getNodeValue();

        if (val != null){
            System.out.print(" nodeValue= ");
            if (val.trim().equals(""))
                System.out.print("[WS]");
            else
                System.out.print(node.getNodeValue());
        }

        System.out.println();
    }

    public static void main(String[] args) throws Exception{

        if (args.length != 2){

            System.err.println(
                "Usage: java TreeContentDOMAPI <fileName> <elemName>");

            System.exit(-1);
        }

        DocumentBuilderFactory fact =
            DocumentBuilderFactory.newInstance();

        fact.setNamespaceAware(true);

        fact.setValidating(true);

        DocumentBuilder builder = fact.newDocumentBuilder();

        builder.setErrorHandler(new ErrorHandler());

        Document doc = builder.parse(args[0]);

        NodeList list = doc.getElementsByTagName(args[1]);

        int n = list.getLength();
```

```
if (n == 0){

    System.err.println("Элемент пуст.");
    System.exit(-1);
}

TreeContentDOMAPI tcd = new TreeContentDOMAPI();

for (int k = 0; k < n; k++){

    NodeIterator iter = ((DocumentTraversal)doc).
        createNodeIterator(list.item(k), NodeFilter.SHOW_ALL,
            null, true);

    Node node = null;
    while((node = iter.nextNode()) !=null) tcd.prNode(node);
}
}
```

Прочие узлы дерева DOM API

Для конструкций XML каждого вида в составе DOM API есть интерфейс, описывающий их свойства. Некоторые такие интерфейсы мы уже использовали в листинге 11.1. Перечислим их.

Интерфейс *Attr*

Интерфейс *Attr* описывает свойства атрибута. Метод

```
Element getOwnerElement();
```

возвращает элемент, которому принадлежит атрибут.

Методы

```
String getName();
String getValue();
```

возвращают имя и значение атрибута, а метод

```
String setValue(String value);
```

дает атрибуту новое значение.

Интерфейс *ProcessingInstruction*

Интерфейс *ProcessingInstruction*, расширяющий интерфейс *Node*, описывает методы получения целевого приложения инструкции и данных для нее:

```
String getTarget();  
String getData();
```

и один метод занесения данных в объект:

```
void setData(String data);
```

Интерфейс *CharacterData*

Интерфейс `CharacterData` расширяет интерфейс `Node`, описывая узел, содержащий текст. Соответственно, он добавляет к методам интерфейса `Node` методы работы с текстом.

Методы

```
void appendData(String arg);  
void insertData(int offset, String arg);  
void setData(String data);  
void deleteData(int offset, int count);
```

добавляют к текстовому узлу, вставляют в него, заменяют или удаляют строку текста. В последнем методе отсчитывается количество 16-битных символов.

Методы

```
String getData();  
int getLength();
```

выдают содержимое текстового узла и его длину в 16-битных единицах.

Метод

```
void replaceData(int offset, int count, String arg);
```

позволяет заменить участок текста длиной `count`, начинающийся от точки `offset`, новым текстом `arg`. Единица длины текста — 16 битов.

Наконец, метод

```
String substringData(int offset, int count);
```

возвращает участок текста длиной `count` 16-битных символов, начинающийся от точки `offset`.

У интерфейса `CharacterData` есть три интерфейса-наследника: `Text`, `Comment` и `CDATASection`, — описывающих особенности каждой из этих текстовых конструкций XML.

Интерфейс *Text*

Самый простой наследник интерфейса `CharacterData` — это интерфейс `Text`. Он ничего не добавляет к интерфейсу `CharacterData` и служит только пометкой для анализатора, указывающей вид узла.

Интерфейс *Comment*

Интерфейс `Comment` тоже расширяет интерфейс `CharacterData`, не добавляя к нему ни одного метода, а только указывая анализатору, что начало `<!--` и конец `-->` комментария не надо включать в создаваемый текстовый узел.

Интерфейс *CDATASection*

Интерфейс `CDATASection`, расширяющий интерфейс `CharacterData`, точно так же указывает анализатору, что символы начала и конца секции не надо включать в узел. Он тоже не вносит ничего нового в интерфейс `CharacterData`.

Интерфейсы *Entity*, *EntityReference*, *Notation*

Еще три интерфейса, не добавляющие ничего к своему предку, — это интерфейсы `Entity`, `EntityReference` и `Notation`, расширяющие интерфейс `Node`. Они служат указанием на типы "сущность", "ссылка на сущность" и "нотация".

Интерфейс *NodeList*

При анализе дерева очень часто приходится иметь дело с упорядоченной последовательностью узлов, например, если надо просмотреть всех потомков какого-то узла. Такая последовательность описывается интерфейсом `NodeList`. Он пронумеровывает узлы, начиная от нуля, и описывает два метода:

метод

```
int getLength();
```

получения количества узлов, и метод

```
Node item(int index);
```

получения ссылки на узел с индексом `index`.

Конструирование нового дерева

Методами описанных выше интерфейсов легко построить новое дерево с нуля. Построение начинается с определения документа.

```
DocumentBuilderFactory fact =  
    DocumentBuilderFactory.newInstance();  
  
DocumentBuilder builder = fact.newDocumentBuilder();  
  
Document doc = builder.newDocument();
```


Затем создается корневой узел дерева:

```
Element root = doc.createElement("notebook");
```

При необходимости он снабжается атрибутами:

```
root.setAttribute("attr1", "value1");  
root.setAttribute("attr2", "value2");
```

Атрибуты можно задать по-другому — сначала создать объект типа `Attr`:

```
Attr attribute = doc.createAttribute("attr3");
```

Затем определить значение атрибута:

```
attribute.setValue("value3");
```

и присоединить объект к элементу:

```
root.setAttributeNode(attribute);
```

При записи содержимого элемента надо помнить, что это текстовый узел. Сначала создадим его вместе с содержимым:

```
Text txt = doc.createTextNode("Содержимое элемента.");
```

а потом присоединим к элементу:

```
root.appendChild(txt);
```

Вложенный элемент создается таким же образом:

```
Element newElem = doc.createElement("person");  
root.appendChild(newElem);
```

Точно так же, методами `createXxx()` интерфейса `Document`, создаются другие конструкции XML — комментарии, секции CDATA, инструкции по обработке, ссылки на сущности. Они заполняются содержимым сразу же при создании или своими методами и затем присоединяются к дереву.

Упражнение

Постройте дерево адресной книжки листинга 1.2.

Добавление элемента в дерево DOM

Приведем пример вставки нового узла в дерево объектов, построенного по документу XML. Добавим в адресную книжку листинга 1.2 новый рабочий или домашний телефон уже имеющегося там человека. Это действие приведено в листинге 11.3. Программу, записанную в нем, можно использовать так:

```
$ java TreeInsertDOM notebook.xml home 243716 Иванов
```

Листинг 11.3. Вставка в адресную книжку с помощью DOM API

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.xml.sax.*;

class ErrHand implements ErrorHandler{

    public void warning(SAXParseException ex){

        System.err.println("Warning: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            "   col = " + ex.getColumnNumber());
    }

    public void error(SAXParseException ex){

        System.err.println("Error: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            "   col = " + ex.getColumnNumber());
    }

    public void fatalError(SAXParseException ex){

        System.err.println("Fatal error: " + ex);
        System.err.println("line = " + ex.getLineNumber() +
            "   col = " + ex.getColumnNumber());
    }
}

public class TreeInsertDOM{

    static final String JAXP_SCHEMA_LANGUAGE =
        "http://java.sun.com/xml/jaxp/properties/schemaLanguage";

    static final String W3C_XML_SCHEMA =
        "http://www.w3.org/2001/XMLSchema";

    public static void main(String[] args) throws Exception{

        if (args.length != 4){

            System.err.println("Usage: java TreeInsertDOM " +
                "<file-name> {work|home} <phone> <surname>");
            System.exit(-1);
        }

        DocumentBuilderFactory fact =
            DocumentBuilderFactory.newInstance();
```

```
fact.setNamespaceAware(true);

fact.setValidating(true);

try{
    fact.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
}catch(IllegalArgumentException ex){

    System.err.println("Неизвестное свойство: " +
        JAXP_SCHEMA_LANGUAGE);
    System.exit(-1);
}

DocumentBuilder builder = fact.newDocumentBuilder();

builder.setErrorHandler(new ErrHand());

Document doc = builder.parse(args[0]);

NodeList list = doc.getElementsByTagName("notebook");

int n = list.getLength();

if (n == 0){

    System.err.println("Документ пуст.");
    System.exit(-1);
}

Node thisNode = null;

for (int k = 0; k < n; k++){

    thisNode = list.item(k);
    String elemName = null;

    if (thisNode.getFirstChild() instanceof Element){
        elemName = (thisNode.getFirstChild()).getNodeName();

        if (elemName.equals("name")){

            if (!thisNode.hasAttributes()){

                System.err.println("Атрибуты отсутствуют " + elemName);
                System.exit(1);
            }

            NamedNodeMap attrs = thisNode.getAttributes();

            Node attr = attrs.getNamedItem("surname");
```

```

        if (attr instanceof Attr)
            if (((Attr) attr).getValue().equals(args[3])) break;
    }
}

NodeList topics = ((Element) thisNode)
    .getElementsByTagName("phone-list");

Node newNode;

if (args[1].equals("work"))

    newNode = doc.createElement("work-phone");

else newNode = doc.createElement("home-phone");

Text textNode = doc.createTextNode(args[2]);

newNode.appendChild(textNode);

thisNode.appendChild(newNode);
}
}

```

Прочие модули DOM API

В предыдущих примерах мы пользовались модулями Core и Traversal, входящими в DOM API Level 2. Во второй уровень Level 2 входят еще модули HTML, Style, Events, Range и Views. Они описаны в соответствующих спецификациях, которые можно посмотреть по адресу <http://www.w3.org/DOM/DOMTR>. В наборе JAXP они занимают пять пакетов:

```
org.w3c.dom.html, org.w3c.dom.stylesheets, org.w3c.dom.css,
org.w3c.dom.events, org.w3c.dom.views.
```

Модуль Range, составляющий пакет `org.w3c.dom.ranges`, в JAXP пока не реализован.

Все модули, кроме Core, необязательны. Проверить, реализует ли DOM-анализатор тот или иной модуль, можно следующим образом:

```

DocumentBuilderFactory fact =
    DocumentBuilderFactory.newInstance();

DocumentBuilder builder = fact.newDocumentBuilder();

DOMImplementation impl = builder.getDOMImplementation();

System.out.println(impl.hasFeature(module, "2.0"));

```

Аргумент `module` метода `hasFeature()` может принимать значения "Core", "HTML", "XHTML", "XML", "Traversal", "StyleSheets", "CSS", "Events", "MouseEvents", "UIEvents", "HTMLEvents", "MutationEvents", "Range" или "Views". Если в анализаторе реализован соответствующий модуль, то на консоли появится слово `true`, если нет, то `false`.

Модуль HTML

Модуль DOM Level 2 HTML содержит множество интерфейсов, описывающих документ HTML или XHTML в целом (`HTMLDocument`), общие свойства элементов HTML/XHTML (`HTMLElement`), список тегов (`HTMLCollection`) и отдельные теги HTML/XHTML (`HTMLAnchorElement`, `HTMLHeadElement`, `HTMLBodyElement`, `HTMLFontElement`, `HTMLFormElement`, `HTMLTableElement` и т. д., всего около пятидесяти интерфейсов). Такие же интерфейсы были и на уровне Level 1, но на уровне Level 2 они переработаны с учетом требований языка XHTML. В наборе JAXP модуль HTML расположен в пакете `org.w3c.dom.html`.

Интерфейс `HTMLDocument` расширяет интерфейс `Document`, интерфейс `HTMLElement` — интерфейс `Element`. Интерфейсы `HTMLXxxElement`, описывающие отдельные теги HTML, расширяют интерфейс `HTMLElement`. Это означает, что интерфейсы модуля HTML только добавляют к узлам модуля Core узлы, специфичные для элементов HTML и XHTML, но ничего не добавляют к методам построения и обработки дерева DOM.

Например, у всякого элемента HTML может быть атрибут ID, поэтому в интерфейсе `HTMLElement` описаны методы чтения и записи этого атрибута `getId()` и `setId()`. У тега `<a>` обязателен атрибут `href`, поэтому в интерфейсе `HTMLAnchorElement` есть методы чтения и записи его значений `getHref()` и `setHref()`. У тега `<form>` обязательны атрибуты `method` и `action`, поэтому в интерфейсе `HTMLFormElement` предусмотрены специальные методы их чтения и записи `getMethod()`, `getAction()`, `setMethod()`, `setAction()`.

Итак, можно считать, что модуль HTML добавляет к модулю Core интерфейсы, описывающие дополнительные узлы — элементы языков HTML/XHTML. Если надо построить дерево, моделирующее документ HTML, то можно создавать узлы более конкретного типа, чем просто тип `Element`, и затем извлекать из них более точные сведения.

Модуль Style

Модуль DOM Level 2 Style состоит из двух частей: первая часть, `StyleSheets`, описывает таблицы стилей вообще, вторая часть, `CSS`, — таблицы стилей типа "text/css" и набор свойств стилей, составляющих таблицу CSS. В наборе JAXP модуль Style разложен по двум пакетам: `org.w3c.dom.stylesheets` и `org.w3c.dom.css`.

Модуль `Style` работает с таблицами стилей `CSS` точно так же, как модуль `Core` — с документами `XML`. Интерфейсу `Document` соответствует интерфейс `DocumentStyle`, в котором, правда, описан только один метод:

```
StyleSheetList getStyleSheets();
```

Этот метод дает список таблиц стилей, связанных с документом. Интерфейс `StyleSheetList` содержит два метода:

```
int getLength();
StyleSheet item(int index);
```

позволяющие получить таблицы стилей `StyleSheet` из списка. Если мы работаем с таблицами стилей `CSS`, то можем воспользоваться расширением интерфейса `StyleSheet`, которое называется `CSSStyleSheet`. Эти два интерфейса обычно реализуются одним объектом.

В интерфейсе `CSSStyleSheet` описан метод получения списка правил `CSS` из таблицы стилей

```
CSSRuleList getCssRules();
```

Каждое правило из списка описывается интерфейсом `CSSRule`. Его можно получить методами:

```
int getLength();
CSSRule item(int index);
```

интерфейса `CSSRuleList`.

Наконец, интерфейс `CSSRule` содержит методы работы с правилом `CSS`:

```
String getCssText();
void setCssText(String cssText);
short getType();
```

Метод `getType()` позволяет различить более конкретные типы правил `CSS`, чем тип `CSSRule`:

- ☐ `CHARSET_RULE` — правило `@charset`, описанное в модуле `Style` интерфейсом `CSSCharsetRule`;
- ☐ `FONT_FACE_RULE` — правило `@font-face`, описанное интерфейсом `CSSFontFaceRule`;
- ☐ `IMPORT_RULE` — правило `@import`, описанное интерфейсом `CSSImportRule`;
- ☐ `MEDIA_RULE` — правило `@media`, описанное интерфейсом `CSSMediaRule`;
- ☐ `PAGE_RULE` — правило `@page`, описанное интерфейсом `CSSPageRule`;
- ☐ `STYLE_RULE` — правило неопределенного типа, описанное интерфейсом `CSSStyleRule`;

- UNKNOWN_RULE — правило неизвестного типа, оно описано интерфейсом `CSSUnknownRule`.

Если в DOM-анализатор встроен модуль `Style`, то интерфейс `DocumentStyle` реализуется тем же объектом, что и интерфейс `Document`. Поэтому схема работы с модулем `Style` такова:

```
DocumentStyle ds = (DocumentStyle)doc;

StyleSheetList ssl = ds.getStyleSheets();

for (int k = 0; k < ssl.getLength(); k++){

    CSSStyleSheet css = (CSSStyleSheet)ssl.item(k);
    CSSRuleList crl = css.getCssRules();

    for (int m = 0; m < crl.getLength(); m++){

        CSSRule cr = crl.item(m);
        String s = cr.getCssText();
    }
}
```

Модуль Views

Представление (view) — это способ показа документа XML. Например, документ, связанный с таблицей стилей CSS, даст одно представление, а тот же документ, связанный с таблицей стилей XSL-FO, — другое представление. Поскольку способов представления документа очень много, не меньше, чем таблиц стилей, модуль DOM Level 2 Views дает только очень расплывчатое понятие о представлениях. В этом модуле, расположенном в JAXP-пакете `org.w3c.dom.views`, всего два интерфейса: `AbstractView` и `DocumentView`.

Интерфейс `AbstractView` дает только один, стандартный способ представления, который можно получить единственным методом интерфейса

```
DocumentView getDocument();
```

Реализации этого интерфейса могут давать и другие представления.

Интерфейс `DocumentView`, с которым должно быть связано какое-то стандартное представление, тоже содержит только один метод

```
AbstractView getDefaultView();
```

позволяющий получить это представление.

Модуль Events

Модуль DOM Level 2 Events описывает средства, позволяющие отслеживать и реагировать на какие-то действия с деревом DOM: изменение узла, вставку или удаление узла. Кроме того, можно отслеживать действия мыши и другие действия пользователя. В наборе JAXP модуль Events помещен в пакет `org.w3c.dom.events`.

Событие может произойти при обработке любого узла дерева DOM. Возникшему событию указывается *целевой узел* (event target), в котором событие должно быть обработано. Далее с событием может произойти одно из трех действий:

- ❑ событие, достигшее целевого узла, может быть обработано в нем и передано узлу-предку (bubbling);
- ❑ по пути к целевому узлу событие может быть "захвачено" узлом-предком целевого узла и после обработки передано потомку (capturing);
- ❑ событие может быть обработано самим анализатором по заложенным в нем правилам или уничтожено без обработки целевым узлом (cancelable).

Для обработки события в узле регистрируется объект-обработчик, "прослушивающий" узел в ожидании наступления события. Как только возникло событие, объект-обработчик выполняет свои методы обработки события.

Как и все в Java, *событие* — это объект, в данном случае — объект типа `Event`. Точнее говоря, при наступлении события создается такой объект. В модуле Events рассматриваются три типа событий и им соответствуют три интерфейса, расширяющих интерфейс `Event`:

- ❑ интерфейс `MutationEvent` описывает события, происходящие при изменении дерева или его узлов;
- ❑ интерфейс `UIEvent` описывает события, происходящие при действиях пользователя;
- ❑ интерфейс `MouseEvent`, расширяющий интерфейс `UIEvent`, описывает события, происходящие при действии с мышью и нажатии специальных клавиш.

Интерфейс `Event`

В интерфейсе `Event` описаны общие свойства всех событий. Интерфейс содержит три константы, отмечающие состояния события:

- ❑ `BUBBLING_PHASE` — событие передается от узла к узлу;
- ❑ `CAPTURING_PHASE` — событие "захвачено" узлом;
- ❑ `AT_PHASE` — событие обрабатывается.

Получить состояние события в виде одной из этих констант можно методом

```
short getEventPhase();
```

Проверить состояние можно логическими методами:

```
boolean getBubbles();  
boolean getCancelable();
```

Тип события можно узнать методом

```
String getType();
```

Метод возвращает одно из значений: "MouseEvents", "UIEvents", "HTMLEvents", "MutationEvents".

Целевой узел и узел, обрабатывающий событие, можно определить методами:

```
EventTarget getTarget();  
EventTarget getCurrentTarget();
```

Получить время возникновения события в миллисекундах, прошедших с 1 января 1970 года, можно методом

```
long getTimestamp();
```

Прекратить передачу события узлам-предкам можно методом

```
void stopPropagation();
```

Интерфейс *MutationEvent*

Интерфейс `MutationEvent` описывает события, происходящие при удалении, вставке или изменении узла, его содержимого или атрибутов. Он добавляет к интерфейсу `Event` три константы, отмечающие изменение атрибута:

- `ADDITION` — добавлен узел типа `Attr`;
- `MODIFICATION` — изменен узел типа `Attr`;
- `REMOVAL` — удален узел типа `Attr`.

Метод

```
short getAttrChange();
```

возвращает одну из этих констант.

Методы:

```
String getAttrName();  
String getNewValue();  
String getPrevValue();
```

возвращают имя измененного атрибута, его новое и старое значение.

Интерфейс *UIEvent*

Интерфейс `UIEvent` описывает события, происходящие при действиях пользователя, не детализируя их. Он добавляет к интерфейсу `Event` методы, сообщающие сведения о событии:

```
int getDetail();
AbstractView getView();
```

Интерфейс *MouseEvent*

Интерфейс `MouseEvent`, расширяющий интерфейс `UIEvent`, детализирует события, происходящие в результате действий пользователя, выделяя события мыши и клавиш-модификаторов `<Shift>`, `<Ctrl>`, `<Alt>`, `<Meta>`. Изменение состояния этих клавиш отслеживается логическими методами:

```
boolean getShiftKey();
boolean getCtrlKey();
boolean getAltKey();
boolean getMetaKey();
```

Нажатие или отпускание той или иной кнопки мыши отслеживается методом

```
short getButton();
```

возвращающим 0, если действует первая (левая) кнопка мыши; 1, если действует средняя кнопка; и 2, если действует вторая (правая) кнопка.

Положение курсора мыши во время возникновения события в координатах DOM и в координатах дисплея отмечают методы:

```
int getClientX();
int getClientY();
int getScreenX();
int getScreenY();
```

Интерфейс *EventTarget*

Итак, произошло событие и создан объект-событие типа `Event`, `MutationEvent`, `UIEvent` или `MouseEvent`. Он перемещается от узла-источника события к целевому узлу, в котором событие должно быть обработано. В целевом узле следует зарегистрировать предварительно созданный объект-обработчик события. Для этого целевой узел должен реализовать интерфейс `EventTarget`. В этом интерфейсе всего три метода. Метод

```
void addEventListener(String type, EventListener listener,
    boolean useCapture);
```

регистрирует объект-обработчик `listener`, "прослушивающий" события типа `type`. Третий аргумент `useCapture` указывает, проводить ли "захват" события.

Метод

```
void removeEventListener(String type, EventListener listener,
    boolean useCapture);
```

отсоединяет обработчик.

Третий метод

```
boolean dispatchEvent(Event evt);
```

возбуждает событие `evt`.

Интерфейс *EventListener*

Объект-обработчик события — это экземпляр класса, реализующего интерфейс `EventListener`. В этом интерфейсе описан всего один метод:

```
void handleEvent(Event event);
```

Этот-то метод и обрабатывает событие. При этом он использует объект-событие `event`, переданное методу DOM-анализатором.

Обработка события

Приступая к обработке события, прежде всего, нужно написать класс-обработчик события. Как только что было сказано, этот класс должен реализовать интерфейс `EventListener`, а значит, в нем надо написать метод `handleEvent()`. Вот пример класса-обработчика событий:

```
public class MyEventHandler implements EventListener{
    public void handleEvent(Event event){
        if (event.getType().equals("MutationEvents"))
            System.out.println("Дерево изменено!");
    }
}
```

Затем надо зарегистрировать обработчик в целевом узле события. Всякий DOM-анализатор, реализующий модуль `Events`, должен создавать объекты типа `Node`, в которых уже реализован интерфейс `EventTarget`. Значит, у них есть метод `addEventListener()`. Воспользуемся им:

```
node.addEventListener("MutationEvents", new MyEventHandler(), true);
```

Теперь узел `node` при возникновении события типа `"MutationEvents"` обратится к методу `addEventListener()`, а тот вызовет метод `handleEvent()` нашего класса `MyEventHandler`.

Модуль Range

Модуль DOM Level 2 Range содержит средства, позволяющие выделить некоторый диапазон объектов дерева DOM и работать с ним. Он размещается в пакете `org.w3c.dom.ranges`. Его основу составляет интерфейс `Range`, описывающий диапазон и методы работы с ним. Объект типа `Range` создается методом

```
Range createRange();
```

интерфейса `DocumentRange`. Интерфейс `DocumentRange` реализуется тем же объектом, что и интерфейс `Document`. Поэтому объект-диапазон можно получить из объекта `doc` типа `Document` следующим образом:

```
Range range = ((DocumentRange)doc).createRange();
```

После создания объекта `range` следует определить границы диапазона. Это делается следующими методами интерфейса `Range`:

```
void setStartBefore(Node refNode);  
void setStartAfter(Node refNode);  
void setEndBefore(Node refNode);  
void setEndAfter(Node refNode);
```

Эти методы получают ссылку `refRange` на начальный или конечный узел диапазона и могут включить этот узел в диапазон или не включать его.

Начальный узел диапазона называется его начальным контейнером, конечный — конечным контейнером. Ссылки на них всегда можно получить методами:

```
Node getStartContainer();  
Node getEndContainer();
```

Диапазон может начинаться и заканчиваться где-то посередине своего контейнера. Например, начальным контейнером может быть текстовый узел, а диапазон начинаться с середины содержащегося в нем текста. В таком случае образуется смещение от начала контейнера. Начало и конец диапазона с учетом смещений можно задать методами:

```
void setStart(Node refNode, int offset);  
void setEnd(Node refNode, int offset);
```

Смещения можно получить методами:

```
int getStartOffset();  
int getEndOffset();
```

Весь диапазон целиком лежит в каком-то узле-предке, в крайнем случае, это корневой узел дерева. Получить ссылку на этот узел-предок можно методом

```
Node getCommonAncestorContainer();
```

После определения диапазона можно извлечь его содержимое. Метод

```
void deleteContents();
```

удаляет диапазон из исходного дерева. Метод

```
DocumentFragment extractContents();
```

тоже удаляет диапазон из дерева, но заносит его в объект типа `DocumentFragment`.

Метод

```
DocumentFragment cloneContents();
```

копирует диапазон в объект типа `DocumentFragment`, не удаляя его из дерева.

Интерфейс `DocumentFragment`, участвующий в этих методах, расширяет интерфейс `Node`, не добавляя к нему ничего. Это, так сказать, минимальное расширение, ведущее к интерфейсу `Document`. Получив объект типа `DocumentFragment`, мы можем работать с ним как с корневым узлом поддерева.

В начало диапазона можно вставить новый узел методом

```
void insertNode(Node newNode);
```

Прекращая работу с диапазоном, следует освободить ресурсы, занятые им, методом

```
void detach();
```

Другие DOM-анализаторы

Сообщество Apache Software Foundation свободно распространяет XML-анализатор Xerces2, в котором уже реализованы некоторые конструкции DOM Level 3. Его можно скопировать с Web-страницы <http://xml.apache.org/xerces2-j/>.

Участники общественного проекта JDOM не стали реализовывать модель DOM, а разработали свою модель дерева объектов, получившую название JDOM. Они выпускают одноименный свободно распространяемый программный продукт, с которым можно ознакомиться на сайте проекта <http://www.jdom.org/>. Этот продукт все еще находится в стадии разработки, но уже широко используется для обработки документов XML средствами Java.

Участники другого общественного проекта dom4j приняли модель W3C DOM, но упростили и упорядочили DOM API. С их одноименным продуктом dom4j можно ознакомиться на сайте <http://www.dom4j.org/>.

Фирма TME (The Mind Electric), <http://www.themindelectric.com/>, выпускающая известные программные продукты GLUE и GAIA, поставяет в составе GLUE или отдельно коммерческий набор инструментальных средств EXML (Electric XML) и его расширение EXML+ (Electric XML+). Этот продукт полностью реализует W3C DOM, весьма компактен, экономичен и быстр.

Вопросы для самопроверки

1. Что такое дерево DOM?
2. Что называется узлом Node дерева?
3. Почему весь документ Document считается узлом дерева?
4. Какие уровни различаются в спецификациях DOM?
5. На каких языках можно реализовать спецификации DOM?
6. Какая связь между DOM и Java?
7. Как можно обойти дерево DOM?
8. Как узнать тип узла?
9. Как получить содержимое узла?
10. Как вставить новый узел в дерево?
11. Как удалить узел из дерева?
12. Какие дополнительные модули включены в DOM Level 2?
13. Как узнать, реализует ли ваш DOM-анализатор тот или иной модуль?
14. Какие расширенные возможности дают DOM-анализатору дополнительные модули?
15. Что понимается под событием в DOM?
16. Каков путь происхождения и обработки события?

Список использованной литературы

1. Броден Б., Минник К. Электронный магазин на Java и XML. — СПб.: Питер, 2002. — 400 с., ил.
2. Валиков А. Н. Технология XSLT. — СПб.: БХВ-Петербург, 2002. — 544 с., ил.
3. Даконта М., Саганич А. XML и Java 2. — СПб.: Питер, 2001. — 384 с., ил.
4. Кэй М. XSLT. Справочник программиста. — СПб.: Символ-Плюс, 2002. — 1016 с., ил.
5. Мак-Лахлин Б. Java и XML. — СПб.: Символ-Плюс, 2002. — 544 с., ил.
6. Фридл Дж. Регулярные выражения. — СПб.: Питер, 2001. — 352 с., ил.
7. Хабибуллин И. Ш. Создание распределенных приложений на Java 2. — СПб.: БХВ-Петербург, 2002. — 704 с., ил.
8. Холзнер С. XSLT. Библиотека программиста. — СПб.: Питер, 2003. — 544 с.

Предметный указатель

A

Arc 97
ASCII 13
Atomic values 125
Attribute 31
Attribute information item 40
Attribute node 124
Axis 126
Axis step 126

B

Bare name 106
Binding language 286
Border 221, 236

C

Castor 289
Character data 30
Character information item 41
Child sequence 110
Comment information item 41
Comment node 124
Components 62
Compound attribute 226
Computed constructor 151
Content 28, 221
Covering range 120
CP1251 14
CP866 14
CSS 16, 173

D

Data binding 273
DCD 87
DDML 87
Declaration 190
Default namespace 35
Direct element constructor 149
Document information item 39
Document node 124
Document Type Declaration 24, 27
Document type declaration information item 41
Document Type Definition 27
DOM API 255, 291
dom4j 325
DTD 16, 24, 27, 28

E

EIS 288
Electric XML 60
Element information item 40
Element node 124
Empty element 29
End-tag 28
Entity reference 30
Existential quantifier 141
EXML 326
EXML+ 326
Expanded-name 124
Expression path 126

F

Facet 66
Filter step 126
FO 178
Formatter 220
Formatting objects 178
Formatting properties 178
Forward axis 127
Fundamental facet 67

G

GLUE 326
Grammar parsing 254

H

HTML 15
Hyper reference 88

I

IDL 292
Information item 39
Infoset 39
Instruction 190
ISO8859-1 13

J

Java XML Pack 60
JAXB 280
JAXP 60, 253, 255, 262, 292
JBoss 289
JDO 273, 288
JDOM 325

K

Kind test 131
Kodo JDO 289
KOI8-R 14

L

Lexical parsing 253
Library module 166

List 65
Location 117
Location set 117
Location step 126

M

MacCyrillic 14
Main module 164
Mapping file 289
Markup declaration 49
Marshalling 286
Model group 73
MSXML 60

N

Name test 130
Named template 198
Namespace 34
Namespace information item 41
Namespace node 124
Node 124
Node set 126
Node test 126, 130
Notation information item 41

O

OpenFusion 289
Oracle XML Parser 60

P

Padding 221, 238
Page 220
Page viewport area 233
Parser 254
Point 116
Pointer 106
Position 135
Predicate 126
Processing instruction 38
Processing instruction information item 40
Processing instruction node 124
Prolog 164
Property 173

Q

QName 34
Qualified Name 34, 65
Query 148

R

Range 116
Region viewport area 235
Relax 86
RELAX NG 86
Resource 97
Restriction 65
Reverse axis 128
Root element 26

S

SAX 255
SAX API 255
SAX2 255
Scanner 253
Schematron 86
Scheme 109
Scheme-based pointer 107
Selector 173
Sequence 126
Sequence constructor 209
SGML 16
Shorthand pointer 106
Shorthand properties 226
SOX 87
SSI 16
Start-tag 28
Stylesheet 174, 178

T

Tag 15
Target namespace 82
Template rule 198
Text node 125
Token 253
TREG 87

Tuple 154
Tuple stream 154

U

Unexpanded entity reference information item 42
Unicode 14
Union 65
Universal quantifier 141
Unmarshalling 286
Unparsed entity information item 42
UTF-8 14

V

Valid 24
Validating parser 60

W

W3C 17
WebLogic 289
WebSphere 60, 289
Well-formed 23
Writing mode 237
WSDP 280

X

XDR 87
Xerces 60, 85, 303
XHTML 35
XML 11, 17
XML Declaration 24
XML Schema 61, 62
XML Schema Definition Language 28
XML schema instance 84
xml4j 60
XP 60
XSchema 87
XSD 28, 76
XSL 177
XSL-FO 178
XSV 85

А

Ассоциированное имя 83
Атомарные значения 125
Атрибут 31
◊ составной 226

Б

Базовый тип 75
Библиотечный модуль 166

В

Видимая зона:
◊ области 235
◊ страницы 233
Выражение 136

Г

Гиперссылка 88
Главный модуль 164
Глобальное имя 83
Грамматический анализ 254

Д

Декларация 190
Документ:
◊ верный 24, 27
◊ хорошо оформленный 23
Дополнение 142
Дуга 97

Е

Единица информации 39
◊ DTD 41
◊ атрибута 40
◊ документа 39
◊ инструкции по обработке 40
◊ комментария 41
◊ необрабатываемой секции 42
◊ объявления 41
◊ пространства имен 41
◊ символа 41
◊ ссылки на сущность 42
◊ элемента 40

З

Запрос 148
Зона:
◊ блочная 221
◊ встроенная 221

И

Именованный шаблон 198
Имя XML 64
Инструкция 190
◊ по обработке 38
Информационная система предприятия 288
Информационное множество 39
Инфосет 39

К

Квантор:
◊ всеобщности 141
◊ существования 141
Компоненты 62
Конструктор:
◊ вычисляемый 151
◊ последовательности 209
◊ узлов 149
Корневой элемент 26
Кортеж 154

Л

Лексический анализ 253

М

Местоположение 117
Множество узлов 126
Модель группы 73

О

Область 117
Объединение 65, 141
Объект данных 273
Объявление:
◊ XML 24
◊ разметки 49
◊ типа документа 24, 27

Определение типа документа 27
Ось поиска 126

П

Парсер 254
Пересечение 142
Позиция узла 135
Последовательность 126
◊ вложений 110
Поток кортежей 154
Предикат 126, 135
Проверяющий анализатор 60
Пролог 164
Простой тип 62
Простой указатель 106
Простой элемент 62
Пространство имен 34
◊ по умолчанию 35
Прямой конструктор элемента 149
Пустой элемент 29

Р

Разборка 286
Расширенное имя 34, 124
Реализация 23
Ресурс 97

С

Сборка 286
Свойство 173
Связывание данных 273
Секция CDATA 30
Селектор 173
Сканер 253
Сложный тип 62
Сложный элемент 62
Список 65
Ссылка на сущность 30
Стенографические свойства 226
Страница 220
Сужение 65
Сущность 30
Схема 28, 49, 109
◊ XML 61, 62

Т

Таблица стилей 178
Тег 15
◊ закрывающий 28
◊ открывающий 28
Тело элемента 28
Тест:
◊ по виду узла 131
◊ по имени узла 130
◊ узла 126, 130
Точка 116

У

Узел 124
◊ документа 124
◊ инструкции по обработке 124
◊ пространства имен 124
◊ текстовый 125
Узел-атрибут 124
Узел-комментарий 124
Узел-элемент 124
Указатель 106
Уточненное имя 34

Ф

Фасетка 66
◊ базисная 67
Форматер 220

Ц

Целевое пространство имен 82

Ш

Шаблонное правило 198
Шаг:
◊ поиска 126
◊ направляемый осью поиска 126
◊ направляемый фильтром 126, 136

Э

Экземпляр схемы 84
Элемент XML 28