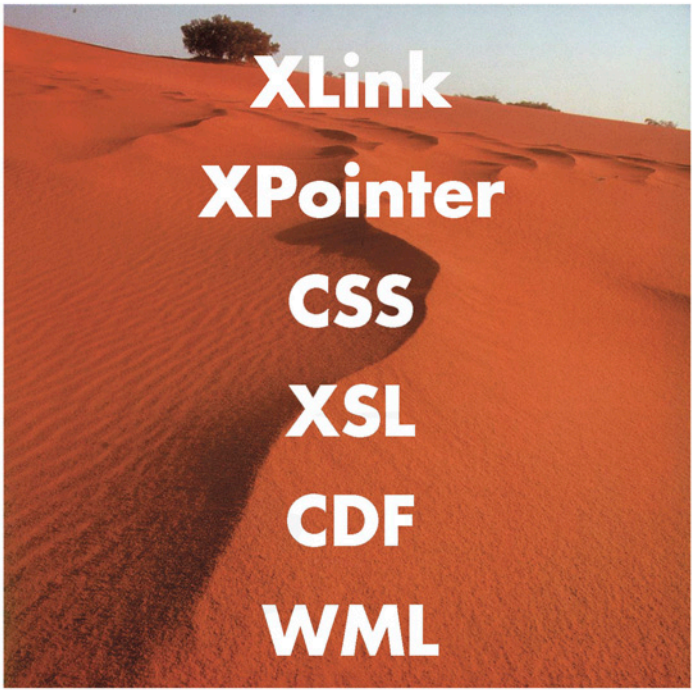
И. Шапошников

bhv
www.bhv.ru
www.bhv.kiev.ua

СПРАВОЧНИК WEB-МАСТЕРА

XML

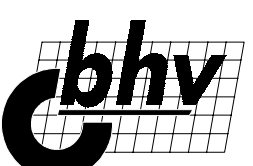
МАСТЕР



Игорь Шапошников

Справочник Web-мастера

XML



Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Справочник по современным технологиям создания и обработки документов, предназначенных для опубликования в сети Интернет, — стандарту XML и его расширениям. Приведены определения структурных элементов языка разметки XML и его синтаксис, вопросы стилевого оформления XML-документов (CSS и XSL), сведения о создании гиперссылок (XLink) и идентификации ресурсов (XPointer), о каналах CDF в Интернете и WAP-ресурсах. Описание сопровождается большим количеством примеров. Дополнительно включены официальные спецификации XML, XML Schema и WML.

Для широкого круга программистов и Web-дизайнеров

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Редактор	*Евгений Васильев*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Ангелины Лужиной*
Зав. производством	*Николай Тверских*

Шапошников И. В.
Справочник Web-мастера. XML. — СПб.: БХВ-Петербург, 2001. — 304 с.: ил.

ISBN 5-94157-049-X

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 22.01.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 24,51.
Тираж 5000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99 от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

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

Содержание

Благодарности

Даниленко Ольге

Who wants to live forever without you?

Прежде всего, спасибо вам, что вы держите сейчас книгу в руках и читаете ее. Но поверьте мне, один я не в состоянии был ее сделать. Всегда есть люди, чей вклад в книгу является не менее весомым, чем авторский. Это, прежде всего, редакторский коллектив: главный редактор Екатерина Кондукова, с ее потрясающим знанием специфики компьютерной индустрии и чувством перспективы, заведующая редакцией Наталья Таркова, дирижировавшая всем процессом редактирования, и принимающий редактор Васильев Евгений, который приложил поистине титанические усилия для превращения текста в книгу.

Особо следует отметить корректора, на чью долю выпал поиск ошибок, пропущенных мной и моим спеллчекером. А они, поверьте мне, были.

Честно говоря, упоминать надо весь технический состав издательства "БХВ-Петербург", который принимал участие в работе над этой книгой. Спасибо вам.

Отдельное и просто огромное спасибо моей Ольге, которая постоянно поддерживала меня в работе.

Спасибо всем моим друзьям в Сети. EILA, Bellefi, Платон, Нюта, Анабелька, Ventral, Финист, Риоха и Готик — мой ящик всегда открыт для вас.

Шапошников Игорь

shiva1@yandex.ru

Введение

Абсолютное большинство всех документов в WWW написано на языке HTML (HyperText Markup Language). Но, к сожалению, на данный момент возможностей этого языка не хватает Web-мастерам для адекватного воплощения их идей. HTML-документы предназначены для отображения в браузерах, и поэтому не могут служить полноценным интерфейсом между пользователями и базами данных. Содержимое баз данных мы можем публиковать в HTML-документах, но вот обратный перенос является уже нетривиальной задачей.

Косвенным свидетельством того, что HTML исчерпал себя, может служить количество вспомогательных технологий, которые позволяют оживить Web-страницы, придать им интерактивность. Языки сценариев VBScript и JavaScript, CGI-приложения, Java-апплеты, встраиваемые модули. Какие только средства не были созданы для того, чтобы обойти ограничения HTML. Но, несмотря на ряд недостатков, HTML все равно остается сердцевиной всех технологий WWW. Мы можем пытаться обойти его ограничения, но эффективности нашей работе это не прибавит.

Исходя из этих соображений, Консорциум WWW (W3C) разработал более мощную и гибкую технологию XML (eXtensible Markup Language), призванную заменить устаревший HTML.

Данная технология на момент написания книги не является стандартом. Версия, действующая в настоящее время, является только кандидатом на стандарт (Candidate Recommendation). Однако, несмотря на то, что формально статус XML еще не определен, этой технологией уже широко пользуются Web-мастера во всем мире (например, при создании онлайновых магазинов или бюро путешествий). Активно создаются XML-приложения, многие документы преобразуются к стандарту XML. Люди сознательно идут на риск. Ведь если правила не утверждены, то многое может еще измениться.

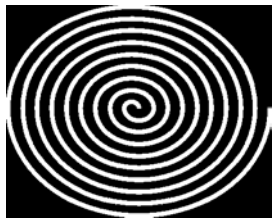
Это, кстати, объясняет ситуацию и с пособиями по XML. Очень часто в разных книгах можно встретить даже различные наборы ключевых слов и предопределенных констант. В этой книге мы не отступим ни на символ от официальной рекомендации W3C. Вы держите в руках справочник, который максимально точно и полно описывает последнюю доступную версию рассматриваемого стандарта.

В первой главе мы рассмотрим непосредственно стандарт XML, точнее — его актуальную версию, которая объявлена кандидатом на стандарт. Разберем примеры, научимся составлять собственные XML-документы. Вторая глава

посвящена обзору обособленной части стандарта — спецификации ссылок XLink. В третьей главе представлены принципы идентификации документов и их фрагментов в XML — язык XPointer. Четвертая глава содержит обзор наиболее популярного приложения XML для push-технологии — специализированного языка разметки CDF (Channel Definition Format, формат определения каналов). В следующей главе мы перейдем к рассмотрению технологии создания Web-страниц, ориентированных на доступ к ним с беспроводных тонких терминалов, то есть с сотовых телефонов. Для этих целей обычно применяется язык WML (Wireless Markup Language), который также является приложением XML. Шестая глава посвящена вопросам правильного отображения XML-документов при помощи стилевых таблиц CSS (Cascading Style Sheets). И, наконец, последняя глава ознакомит вас с преемником CSS, созданным специально для XML. В ней мы рассмотрим язык правил представления (листов стилей) XML-документов для различных устройств и сред — XSL (eXtensible Stylesheet Language).

В каждой главе излагается по возможности предельно подробная информация относительно каждой из перечисленных выше технологий. Вам предлагается описание последней, наиболее свежей версии стандарта. Весь текст в этой книге выверен, а примеры не содержат ошибок. В мир XML мы войдем вместе.

Глава 1



Расширяемый язык разметки XML

Зачем нам это надо?

Как мы уже говорили в предисловии, основой WWW является HTML (HyperText Markup Language). Этот язык представляет собой набор *тэгов* (управляющих дескрипторов), которые позволяют создавать *разметку* документа. То есть помимо указания содержимого документа, мы можем при помощи тэгов HTML управлять отображением этого документа. Технология проста. Браузер получает HTML-документ и анализирует его. Как только в коде документа встречается какой-либо тэг, он распознается, и фрагмент документа, к которому относится тэг, оформляется соответствующим образом.

На данный момент доступна уже четвертая версия языка HTML. Первоначального набора тэгов не хватало для адекватного представления более сложных документов. Поэтому компании Microsoft и Netscape — владельцы двух наиболее популярных браузеров, периодически дополняли наборы тэгов, распознаваемых их браузерами. Вследствие конкуренции этих двух фирм дополнения, естественно, не совпадали. В силу этого разработчики либо не использовали дополнительные средства, либо отказывались от возможности адекватно отображать свой документ в любом браузере. Создавался документ, предназначенный для просмотра в конкретном обозревателе, а часть посетителей сайта, использующих иной браузер, не могла увидеть страницу, содержащую данный HTML-документ (либо видела с искажениями). Проблемой HTML стала его врожденная ограниченность. Даже самый большой список тэгов не в состоянии полностью удовлетворить запросы создателей документов просто в силу того, что этот список ограничен.

Еще одной проблемой стала излишняя популярность WWW. На основе этой технологии стали строить даже локальные сети, которые, соответственно, получили название *интрасетей*. Все корпоративные документы переводились в HTML-формат. Общение внутри сети происходило при помощи электронной почты, создавались сайты, не имеющие выхода в Интернет и служащие хранилищем для корпоративных документов и средством совместной деятельности рабочих групп. Но HTML не мог служить интерфейсом между пользователями и данными. HTML-документы ориентированы прежде всего на отображение, а не на автоматическую обработку. Из базы дан-

ных можно передать данные в HTML-документ. Обратная операция намного сложнее.

Исходя из этого, стало ясно, какими свойствами должен обладать преемник HTML. Требовалось, чтобы новый язык был расширяемым, то есть не зависел от конкретного набора тэгов. Требовалась возможность расширять язык автоматически, исходя из нужд пользователя. Также было необходимо создавать файлы, которые могли бы не только отображаться, но и обрабатываться сторонними приложениями. То есть структура документов должна была напрямую привязываться к ним самим. Каждый документ должен был содержать как информацию, так и ее структуру.

Учитывая эти и многие другие соображения, Консорциум World Wide Web (W3C) в 1996 году приступил к разработке спецификаций нового языка, который должен был прийти на смену HTML. Его назвали XML (eXtensible Markup Language). На данный момент вторая редакция спецификации языка версии 1.0 является кандидатом на официальный стандарт. В этой книге мы рассмотрим данную спецификацию полностью, воспользовавшись документацией самого W3C.

Корни XML

Прежде всего, давайте разберемся, откуда появился XML. Очень давно (по меркам компьютерной индустрии, естественно), в 1986 году организацией ISO (International Organization for Standardization) язык SGML (Standard Generalized Markup Language) был принят в качестве официального стандарта. А использоваться он начал еще до этого момента. Язык SGML применяется в качестве стандарта по настоящее время. Он позволяет описывать структурированные данные, организовывать и представлять информацию, содержащуюся в документах. Стандарт SGML позволяет разработчику создавать свои конструкции разметки. Его потрясающие гибкость и универсальность, охватывающие практически все случаи, возникающие в работе над Web-проектами, казалось бы, выдвигали этот язык идеальным кандидатом для принятия его в качестве основного языка WWW, но существовали и другие обстоятельства, которые помешали ему занять лидирующее место.

Большинство документов в WWW предназначены для просмотра специализированными программами — браузерами. Браузеры анализируют код полученного документа, и на основе инструкций разметки, называемых также тэгами, отображают его в окне просмотра соответствующим образом. Но описание спецификации SGML занимает более 500 страниц текста. Отсюда видно, сколько труда потребовалось бы от разработчиков, чтобы заставить браузеры правильно отображать документы, написанные на SGML. Требовалось что-то гораздо более компактное. Естественным образом и был соз-

дан язык HTML, являющийся очень ограниченным и нерасширяемым подмножеством SGML. Так как набор тэгов HTML был невелик, разрабатывать HTML-документы и программы их просмотра было достаточно легко.

Но впоследствии это достоинство HTML превратилось в его недостаток. Посетителям и владельцам сайтов хотелось получать от HTML все больше и больше. Компании — участники "браузерных войн" добавляли все новые и новые тэги в наборы распознаваемых своими браузерами инструкций. Основная проблема состояла в том, что добавленные тэги у различных компаний тоже были разными. Возникли проблемы с совместимостью, которые не решило и доведение стандарта HTML до версии 4.0. Практически всем стало очевидно, что поскольку каждая версия HTML представляет собой ограниченный и нерасширяемый набор тэгов, то рано или поздно она окажется недостаточной.

На смену HTML пришел стандарт (а точнее, рекомендация к стандарту) XML (eXtensible Markup Language). Это — расширяемый язык разметки. Набор тэгов XML много меньше по объему, чем в HTML, но в данном случае это неважно. Изменилась сама парадигма создания документов. Появилась возможность создавать собственные тэги и конструкции из них, наподобие строительных блоков конструктора Lego. Мы теперь можем при помощи тэгов XML создавать свой язык для каждого типа документов, или даже для каждого документа отдельно.

Подобная гибкость языка позволила практически прозрачно стыковать документы с различными источниками данных для них. При помощи XML стало максимально легко интегрировать данные из различных приложений. А ведь именно интеграции различной информации из разнородных приложений подчас не хватает настоящим рабочим, а не презентационным, корпоративным сайтам. XML подоспел вовремя.

XML, как и его предшественник — HTML, является подмножеством SGML. Но XML представляет собой намного более компактный язык. Поэтому реализация браузеров для XML-документов не является излишне сложной задачей.

Любой документ из WWW в конце концов необходимо отобразить на экране компьютера удаленного пользователя. Для этой цели применяются сейчас программы-обозреватели. Браузер Internet Explorer 5-ой версии технически способен распознавать и анализировать XML-файлы, но до полного счастья еще очень далеко. Адекватное отображение содержимого XML-файлов достигается далеко не всегда.

Намного лучше дело обстоит с Netscape Communicator 5. Помимо распознавания и анализа этот браузер может еще и правильно отображать XML-документы. В него встроен полноценный XML-анализатор. Для отладки документов рекомендуется воспользоваться именно браузером Netscape.

Структура XML-документов

В XML-документах можно выделить две основные части. Первая часть XML-документа предназначена для описания его структуры, а во второй находится непосредственно содержание документа. В описании структуры документа мы можем использовать инструкции для XML-процессора, объявление элементов структуры документа, атрибуты для каждого элемента и так называемые *сущности*. Каждую из этих структурных единиц мы рассмотрим отдельно и подробно.

Описание структуры документа называют DTD-блоком (Document Type Definition). В нем мы указываем отношения на иерархии древовидной структуры элементов документа. Наиболее близкой аналогией для этих элементов могут служить объекты. Как и объекты, элементы разметки структуры могут иметь свойства, которые в данном случае называются *атрибутами*.

Как и в любой объектной иерархии, в XML-документе существует некий корневой элемент, от которого наследуются все остальные элементы.

Содержимое XML-документа форматируется при помощи тэгов, которые определяются в описании типа документа. Наименования тэгов полностью совпадают с наименованиями элементов. А параметры тэгов позволяют устанавливать значения атрибутов элементов.

Инструкции XML-процессора

В качестве первой строки каждого XML-документа должна использоваться исполняемая инструкция, предназначенная для XML-процессора (исполняемые инструкции используются для управления процессом разбора документа). В своем минимально применимом виде она выглядит следующим образом:

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

Как видно из примера, исполняемые инструкции обрамляются специальными ограничителями, состоящими из угловой скобки и знака вопроса. Ключевым словом для каждой исполняемой инструкции является сокращение `xml`. Следом за ним указываются параметры инструкции и соответствующие им значения. Иногда блок исполняемых инструкций называют *прологом* XML-документа.

Приведенный нами пример предназначен для указания браузеру, что данный документ написан на XML. Значение параметра `version` указывает на тот факт, что будет использоваться первая версия стандарта XML. (А другой версии у нас пока и нет.)

В этих инструкциях мы можем не только указывать номер версии стандарта. Для XML-документов заготовлено несколько видов кодировок, состоящих из символов набора Unicode. Большинство кодировок предложено международной организацией по стандартизации (ISO).

Для указания конкретной кодировки, которая будет использоваться для отображения содержимого документа, применяется параметр `encoding`, как в следующем далее примере:

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

В качестве значения параметра здесь задана текстовая строка `"UTF-8"`. Кодировка UTF-8 является одной из наиболее часто используемых кодировок.

Следующие параметры исполняемых инструкций XML-процессора напрямую связаны с обработкой блоков описания структуры документа (DTD-блоков). Забегая немного вперед, необходимо сказать, что подобные блоки могут внедряться в сам XML-документ, или находиться во внешнем файле. Для того чтобы XML-процессор смог правильно их обработать, применяется параметр `standalone`. При помощи этого параметра можно указать, где именно находится описание структуры для данного XML-документа. В следующем примере мы используем объявление XML-документа, в котором указывается, что DTD-блок для него оформлен в виде отдельного файла.

```
<?xml standalone='no'?>
```

У параметра `standalone` есть два предопределенных значения: `yes` и `no`. Значение `no`, как мы уже видели, извещает XML-процессор, что для данного документа DTD-блок выделен в отдельный файл. Значение `yes` указывает на то, что DTD-блок размещен в теле XML-файла.

Мы рассмотрели возможные варианты исполняемых инструкций, помещаемых в начале документа. Практически эти инструкции могут находиться не только в начале документа, но и в любом другом его месте. Но использование исполняемой инструкции в качестве первой строки XML-документа является обязательным условием.

Любое рассмотрение языка на примерах, без четких определений, будет всего лишь обзором. Чтобы избежать этой участи, для каждой из рассматриваемых нами структурных единиц XML-документа мы будем приводить их определение из спецификации XML. Для исполняемых инструкций, помещаемых в пролог документа, оно выглядит следующим образом:

```
[23]  XMLDecl ::=  '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24]  VersionInfo ::=  S 'version' Eq (' VersionNum ' | " VersionNum ")
[25]  Eq ::=  S? '=' S?
[26]  VersionNum ::=  ([a-zA-Z0-9_.:] | '-')+
```

На первый взгляд смотрится достаточно устрашающе. Тем не менее, все спецификации XML выглядят подобным образом. В примере приведена запись конструкций XML в форме Бэкуса-Наура (EBNF, Extended Backus-Naur Form). Разберемся с правилами чтения деклараций в нотации EBNF. В левой части каждой декларации указывается имя конструкции, затем следует оператор эквивалентности (::=), а в правой части следует расшифровка имени, которая содержит его формат и правила оформления. Перед каждой конструкцией в квадратных скобках указывается номер строки спецификации.

Итак, из приведенного фрагмента мы видим, что определение исполняемой инструкции находится в 23-й строке спецификации языка XML. При изучении спецификации необходимо помнить несколько правил записи определений в форме EBNF.

- Если в определение включено несколько терминов, разделенных вертикальной чертой (|), то следует выбрать только один из них. Подобным образом определяется перечислимое множество компонентов.
- Для указания диапазона символов, которые могут использоваться в каком-либо месте определения термина, этот диапазон символов помещается в квадратные скобки.
- Круглые скобки ограничивают так называемые *регулярные выражения*. Проще всего их воспринимать как обычное средство группировки элементов. В случае, когда согласно синтаксису вместо одиночного литерала нужно указать несколько, используются круглые скобки, группирующие их.
- Знак звездочки (*) примыкает справа к символьному выражению, если требуется, чтобы это выражение в результирующей строке могло быть повторено несколько раз или вовсе там не использоваться.
- Знак плюса (+) применяется в качестве модификатора символьного выражения подобно знаку звездочки. Но есть некоторое отличие. Символьное выражение, после которого присутствует этот модификатор, должно встречаться в результирующей строке хотя бы один раз.
- Вопросительный знак (?) используется для указания, что тот или иной элемент могут не найтись в результирующей строке. То есть при помощи этого модификатора указываются опциональные элементы.
- Если в выражении не должны встречаться некоторые символы, то к нему добавляется последовательность [^, после которой указываются исключаемые символы. Например, правило [^&] исключает символ амперсанда.
- Если внутри какого-либо литерала необходимо вставить пробел, то этот пробел обозначается при помощи символа S.
- Любая последовательность символов, заключенная в двойные или одиночные кавычки, является литералом, и в результирующей строке должна появляться именно в том виде, в каком она указана в декларации.

Зная эти правила, рассмотрим синтаксис деклараций из вышеприведенного примера. Начнем, естественно, с директивы [23]. Начинает определение элемент `XMLDecl`, который и является объявлением XML-документа. После оператора эквивалентности указан литерал `'<?xml'`, означающий, что строка объявления XML документа должна начаться с открывающей угловой скобки, вопросительного знака и последовательности символов `xml`. Впрочем, этот факт мы выяснили еще до разбора синтаксиса декларации. Следующим элементом декларации является обязательный элемент `VersionInfo`, указывающий номер версии применяемого стандарта XML. Помимо этого в объявлении могут присутствовать конструкции `EncodingDecl` и `SDDecl`. Завершается декларация последовательностью символов `'?>'`, перед которой может быть размещен необязательный пробел.

В приведенном выше фрагменте спецификации XML отсутствуетопределение конструкции `SDDecl`. Приведем его сейчас:

```
[32]  SDDecl ::=  S 'standalone' Eq (("'" ('yes' | 'no') "'") |
('"' ('yes' | 'no') '"'))
```

Как видно из этой декларации, значения параметра `standalone` (использование внешних DTD-описаний) могут заключаться как в двойные, так и в одиночные кавычки.

Объявление типа документа

Сразу после исполняемой инструкции, указывающей на тот факт, что данный документ создан с применением языка XML, в этом документе помещается объявление типа документа, называемое также DTD (Document Type Definition). DTD-блок является основой для структуризации содержимого XML-файла. То есть этот блок включает в себя правила, по которым происходит разметка содержимого документа. Здесь определяются элементы документа, атрибуты для этих элементов, сущности и комментарии. Фактически DTD-блок не менее важен для XML-документа, чем его значимое содержимое. Рассмотрим пример объявления типа документа:

```
<!DOCTYPE body [
<!ELEMENT body (#PCDATA)>
<!ENTITY name "Igor">
]>
```

В первой строке примера начинается объявление типа документа. Нетрудно заметить, что DTD-блок начинается ключевым словом `DOCTYPE`, помещаемым после открывающей угловой скобки и восклицательного знака. (Необходимо отметить, что все конструкции XML помещаются в угловые скобки. Но перед большинством ключевых слов ставится восклицательный

знак. А исполняемые инструкции XML-процессора, как мы помним, предваряются вопросительным знаком.)

После ключевого слова `DOCTYPE` указывается наименование типа. Это, по сути, имя корневого элемента объектной иерархии данного документа. Здесь необходимо сделать некоторое отступление, и рассказать немного об *элементах* XML-документа, которые более полно мы будем рассматривать уже в следующем разделе. Но в XML все настолько переплетено и взаимосвязано, что есть только один путь строго последовательного и непротиворечивого изложения правил оформления XML-документов. Это — следование официальной спецификации, фрагменты которой мы приводим в тексте этой книги. Но поскольку изучать XML по спецификации трудно, нам придется немного менять местами изложение различных структурных единиц, принося последовательность изложения в жертву его доступности.

Итак, элементы в XML являются основными структурными единицами. Их ближайшей аналогией являются тэги HTML, посредством которых мы могли создавать абзацы, элементы списков, ссылки и прочие элементы разметки. Безусловно их можно назвать прародителями элементов XML. Но в HTML элементы разметки не были взаимосвязаны. В XML же был сделан качественный шаг вперед. Теперь элементы могут быть структурированы. Все элементы зависят друг от друга. В XML-документе обязателен основной элемент, составляющими которого являются другие элементы. То есть несколько упрощенно моделируется объектная иерархия, в которой в качестве объектов выступают элементы XML. Как мы увидим немного позже, это — достаточно хорошая аналогия, и мы еще не раз воспользуемся ею.

Таким образом, в качестве наименования типа XML-документа мы используем имя самого старшего элемента, который включает в себя все остальные элементы. В нашем примере это элемент с именем `body`.

После указания этого имени в квадратных скобках описывается структура всего документа. А после закрывающей квадратной скобки следует закрывающая угловая скобка. Получается, что все определение типа документа находится в рамках одного тэга `DOCTYPE`. Все эти признаки нетрудно обнаружить в коде рассмотренного нами примера.

DTD-блоки могут находиться как внутри XML-документа, так и вне его. Для нескольких XML-документов можно создать один файл с описанием DTD, и использовать его для каждого из этих XML-документов. Для подключения внешнего DTD-блока можно использовать конструкцию, подобную следующей:

```
<!DOCTYPE main PUBLIC "-//New Boundaries//Sweet Immersing//EN"
↳"http://www.newboundaries.com/sweet/dtd/main.dtd">
```

Разберем этот пример. Мы объявляем тип документа `main`, спецификация которого находится в отдельном DTD-файле. В данном случае объявлено,

что это DTD-описание не было принято международной организацией по стандартизации (ISO) в качестве стандарта, оно принадлежит компании New Boundaries, создано для проекта Sweet Immersing, ориентировано на англоязычные документы, и файл со спецификацией находится по адресу `http://www.newboundaries.com/sweet/dtd/main.dtd`.

Ключевое слово `PUBLIC` применяется для так называемых "публичных" DTD. Для них помимо URL указывают еще и наименование. Делается это для того, чтобы XML-браузер мог отыскать эти DTD на других серверах, которые, может быть, будут ближе. Конечно, в критериях Интернета понятие "ближе" относится к географии весьма опосредованно, но выбирается всегда тот DTD-файл, который будет быстрее всего загружен на локальную систему удаленного пользователя. Публичные DTD отличаются от обычных обособленных DTD-файлов тем, что могут находиться на нескольких различных серверах.

Еще одна особенность публичных DTD состоит в том, что их рассматривает международная организация по стандартизации (ISO). В том случае, если этот DTD-блок принят в качестве стандарта, перед именем файла (а не перед его URL) ставится префикс `ISO`. Если в качестве стандарта этот DTD не принят, но, тем не менее, рассматривается группой стандартизации, в качестве префикса используется знак плюса (`+`). Если же он не принят группой стандартизации, применяется префикс в виде минуса (`-`), как мы это видели в нашем примере.

Если же DTD-блок не отправлялся в ISO, то он объявляется "приватным". В таком случае вместо ключевого слова `PUBLIC` указывается ключевое слово `SYSTEM`. При этом в описании используется конструкция следующего вида:

```
<!DOCTYPE main SYSTEM "http://www.newboundaries.com/sweet/main.dtd">
```

Теперь, когда мы знаем, как использовать внешние файлы, содержащие DTD-блоки, следует научиться создавать эти файлы. Любой DTD-файл подчиняется спецификациям XML. То есть, по сути, он является обычным XML-документом, но без значимого содержимого. Поэтому первой строкой здесь также будет исполняемая инструкция XML-процессора. Но в данном случае совсем необязательно указывать номер версии стандарта XML. Обычно во внешних DTD-файлах указывают используемую кодировку, после чего определяются структурные элементы. Так как ссылка на DTD-файл встраивается в соответствующий XML-документ, нет нужды в DTD-файле использовать объявление типа документа с ключевым словом `DOCTYPE`.

Мы уже рассматривали параметр `standalone` для исполняемой инструкции, которая объявляет использование XML. В том случае, если используется внешний DTD-файл, в качестве значения параметра необходимо использовать значение "`no`".

В зависимости от того, как используется DTD-блок, какие правила оформления DTD применяются в XML-документах, эти документы могут называться "хорошо оформленными" (well-formed) и "правильными" (valid). Кстати, некоторые правила спецификации XML являются обязательными для оформления документа, если мы хотим, чтобы этот документ являлся хорошо оформленным или правильным. Для таких правил в спецификации указываются индексы `WFC` (Well-Formedness Constraint) и `VC` (Validity Constraint) соответственно. Для того чтобы избавить вас от штудирования неудобочитаемой спецификации, изложим здесь эти правила более понятным языком.

Итак, для того чтобы документ считался *хорошо оформленным*, необходимо выполнение нескольких правил.

- У документа должен быть только один элемент верхнего уровня, и содержимое документа должно полностью располагаться внутри тэга, соответствующего этому элементу-прародителю.
- В одном тэге не могут употребляться несколько раз одни и те же атрибуты элемента, соответствующего этому тэгу.
- Все сущности должны объявляться до их использования.
- Все тэги должны быть правильно вложены друг в друга, согласно иерархии элементов, каждый открывающий тэг должен иметь своего закрывающего "напарника" (нельзя опускать закрывающие тэги).

Вне сомнения, все XML-документы, которые будут использоваться в публичных целях, просматриваться в различных браузерах и обрабатываться различными приложениями, должны быть, как минимум, хорошо оформленными. Обязательное использование вышеперечисленных правил обусловлено тем, что при несоблюдении какого-либо из них обычный XML-процессор выдаст сообщение о фатальной ошибке, после чего приложение, в котором действует XML-процессор, будет остановлено. Более того, кое-где встречается информация о том, что отдельные приложения и браузеры "зависают" при попытке просмотра документов с ошибками.

Следует обратить внимание на то, что в перечне правил для хорошо оформленных документов нет ни слова о DTD-блоках. Связано это с тем, что согласно спецификации DTD-блоки не являются обязательными для использования в XML-документах. Следующий XML-документ считается правильным, так как не нарушено ни одно из правил:

```
<?xml version="1.0" standalone="yes"?>
<body>well-formed document </body>
```

Следует также учитывать, что XML-процессоры чувствительны к регистру символов.

Правильным считается XML-документ, который удовлетворяет требованиям, предъявляемым к хорошо оформленным документам, и при этом имеет соответствующий DTD-блок и подчиняется всем правилам, описанным в нем.

Для того чтобы вышеприведенный пример XML-документа можно было бы считать правильным, необходимо внести в код примера некоторые изменения, после чего он будет выглядеть следующим образом:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE body [<!ELEMENT body (#PCDATA)>
]>
<body>valid document </body>
```

Теперь же, после рассмотрения примеров DTD-блоков и всех правил, связанных с ними, приведем фрагмент спецификации, описывающий определение типа документа:

```
[28]  doctypedecl ::=  '<!DOCTYPE' S Name (S ExternalID)? S?
      ↳('['(markupdecl | PEReference | S)* ']' S?)? '>'
      [VC: Root Element Type ]
[29]  markupdecl ::=  elementdecl | AttlistDecl | EntityDecl
      ↳| NotationDecl | PI | Comment
      [VC: Proper Declaration/PE Nesting ]
      [WFC: PEs in Internal Subset ]
```

Здесь видно, каким образом в спецификации описывается дополнительное условие наличия корневого элемента, необходимое для создания правильного документа.

Элементы XML-документа

В этой главе мы уже говорили об элементах. Напомним, что элементы являются основными структурными единицами XML-документов. Мы объявляем все элементы документа в DTD-блоке, а потом при разметке значимого содержимого документа мы используем тэги, наименования которых совпадают с наименованиями элементов.

Объявление элемента в самом упрощенном виде мы уже наблюдали в одном из примеров, но сути пока не раскрывали. Приведем текст примера еще раз:

```
<!DOCTYPE body [
<!ELEMENT body (#PCDATA)>
]>
```

Здесь мы объявляем тип документа `body`, для которого, в свою очередь, объявляется одноименный элемент. Как видно, объявление элемента производится при помощи ключевого слова `ELEMENT`, после которого указывается наименование элемента и его тип. Все значимые и обязательные части инструкции разделяются пробелами. В данном случае мы объявили элемент с символьным содержимым, применив для этого стандартную конструкцию `(#PCDATA)` (что означает `parseable character data` — любая информация, с которой может работать XML-процессор).

На самом деле элементы в качестве своего содержимого могут использовать не только символьные данные. Элементы могут содержать другие элементы, которые, в свою очередь, могут тоже содержать элементы, и так далее. То есть выстраивается некая иерархия элементов, вложенных по типу матрешки в основной элемент, объявленный как тип документа при помощи ключевого слова `DOCTYPE`.

Рассмотрим пример такой усложненной организации. Предположим, что мы создаем XML-документы для некоей компании, которая поддерживает базу данных предприятий города. Нам потребуется создать основной элемент `firm`, к которому будут привязаны вложенные элементы, отражающие название фирмы, ее род деятельности, адрес, телефоны, сетевые реквизиты и т. д. Такую организацию можно проиллюстрировать следующим примером:

```
<!ELEMENT firm (name+, address, phone*, fax*, email, info)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT info (#PCDATA)>
```

В этом примере мы создаем элемент, в состав которого входят шесть других элементов с содержимым символьного типа. При этом для группировки субэлементов используются в соответствии с синтаксисом круглые скобки. Все, казалось бы, достаточно прозрачно, но если приглядеться, видно, что после наименований некоторых вложенных элементов установлены дополнительные модифицирующие знаки. Назовем их *модификаторами*. Всего в XML применяется три модификатора.

- Знак плюса (`+`) применяется для таких элементов, наличие которых в родительском элементе является обязательным. Данные элементы могут встречаться в рамках родительского элемента несколько раз. Но, по крайней мере один раз они должны быть задействованы обязательно.
- Знак звездочки (`*`) указывает на то, что данный элемент может встречаться в описании родительского элемента любое количество раз. Впрочем, может там вообще не присутствовать.
- Вопросительный знак (`?`) используется для тех элементов, которые могут появляться в описании родительского элемента только один раз, или вообще не появляться.

Все остальные символы, которые могут встречаться в описании типа элемента, и в то же время не входят в наименования субэлементов, служат либо для группировки, либо в качестве разделителей.

Группировка элементов, как мы знаем, осуществляется при помощи круглых скобок. При помощи же запятой мы фактически устанавливаем порядок следования элементов. То есть в значимом содержимом XML-документа все субэлементы должны следовать именно в том порядке, в каком они перечислены в DTD-блоке.

В том случае, когда необходимо использовать только один из некоторого множества субэлементов, это множество дополнительно заключается в круглые скобки, и все наименования субэлементов разделяются вертикальной чертой (|). Например, если бы мы хотели, чтобы в нашем примере для элемента `firm` можно было бы использовать в качестве реквизита либо номер факса, либо номер телефона, мы могли бы написать следующую декларацию:

```
<!ELEMENT firm (name+, address, (phone | fax), email, info)>
```

Элементы не обязательно должны состоять из других элементов. В любой иерархии должны найтись такие элементы, которые не будут содержать дочерних элементов. Подобные элементы называются *атомарными*.

Это, впрочем, не означает, что в атомарных элементах будут находиться единичные и неделимые фрагменты информации. Каждый элемент может обладать *атрибутами*, которые в свою очередь и будут содержать подобные информационные атомы. Хорошей естественной аналогией для атрибутов являются свойства объектов. (В этой части мы рассматриваем только элементы. Рассмотрение же атрибутов отложим до следующего раздела.)

Итак, нас интересует, как задавать тип для атомарных элементов. На выбор имеются три варианта.

- Если мы собираемся сохранять в элементе какие-либо данные так называемого *смешанного типа* (mixed content), мы должны указать в круглых скобках тип `#PCDATA`. По сути, под эвфемизмом "смешанный тип" подразумевается использование любых символьных данных.
- Если заранее не определено, какой тип данных будет содержать этот элемент, то для указания его типа используется ключевое слово `ANY`.
- Для обозначения типа *пустых* элементов (заглушки) используется ключевое слово `EMPTY`.

Все типы могут комбинироваться в объявлении элемента различным образом для достижения именно того результата, который нам необходим. В качестве примера можно рассмотреть следующие декларации:

```
<!ELEMENT b (#PCDATA)>
<!ELEMENT br EMPTY>
<!ELEMENT container ANY>
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
```

Это определение имеет следующий смысл: элемент `b` содержит символьные данные (используется, так называемый *смешанный* тип содержимого), эле-

мент br изначально задан пустым, элемент container предназначен для хранения данных неопределенного типа, а элемент p может содержать либо символьные данные, либо один из других элементов, наименования которых указаны в скобках и разделенны вертикальной чертой.

И в заключение обзора элементов в соответствии с нашими правилами мы должны привести фрагмент спецификации, описывающий объявление элементов XML-документа.

```
[45]  elementdecl ::=  '<!ELEMENT' S Name S contentspec S? '>'
      [ VC: Unique Element Type Declaration ]
[46]  contentspec ::=  'EMPTY' | 'ANY' | Mixed | children
[47]  children ::=  (choice | seq) ('?' | '*' | '+')?
[48]  cp ::=  (Name | choice | seq) ('?' | '*' | '+')?
[49]  choice ::=  '(' S? cp ( S? '|' S? cp )* S? ')'
      [ VC: Proper Group/PE Nesting ]
[50]  seq ::=  '(' S? cp ( S? ',' S? cp )* S? ')'
      [ VC: Proper Group/PE Nesting ]
```

Атрибуты элементов

В предыдущем разделе мы уже говорили о том, что неделимые порции информации записываются и хранятся в атрибутах элементов. При сравнении элементной модели с объектной иерархией атрибуты можно считать свойствами объектов. При помощи атрибутов мы можем максимально полно детализировать информацию, предназначенную для отображения или использования в данном элементе.

Список атрибутов задается при помощи ключевого слова ATTLIST. Рассмотрим маленький пример:

```
<!ELEMENT text (#PCDATA)>
<!ATTLIST text
       create_date   CDATA #REQUIRED
       last_modified CDATA #IMPLIED
       author        CDATA #REQUIRED
       editor        CDATA "Kondukova E.V.">
```

В этом примере мы объявили элемент text, которому присвоили четыре атрибута. Как видно, после ключевого слова ATTLIST указывается наименование элемента, к которому присоединяются атрибуты, определяемые в данном тэге ниже. После наименования элемента записывается список атрибутов. Для каждого атрибута указывается наименование, тип содержимого атрибута и модификатор атрибута.

Что касается наименования, то с ним все ясно. Наименования атрибутов полностью подчиняются правилам формирования имен в XML. Перечислим их. Имя обязано начинаться с буквенного символа, символа подчеркивания или двоеточия, и может содержать в себе любые буквы, цифры и знаки подчеркивания, двоеточия, дефиса и точки.

После имени атрибута указывается его тип. Всего может быть три типа. В нашем примере мы объявили атрибуты только символьного типа. Для обозначения этого типа применяется ключевое слово `CDATA`. В значениях этих атрибутов могут храниться данные только в виде строк.

Атрибуты могут также иметь *перечислимый* тип. Для подобных атрибутов при их объявлении заранее указывается список возможных значений, заключаемых в круглые скобки и разделяемых вертикальной чертой. В качестве примера можно привести следующее объявление атрибута:

```
<!ATTLIST my_element
        type (a|b|c) "c">
```

В этом примере объявляется атрибут `type`, который может принимать только значения `a`, `b` и `c`, причем по умолчанию используется значение `c`.

В XML применяется также и третий тип атрибутов. Атрибуты этого типа называются *маркерами*. Это — специализированные атрибуты, значения которых несут заранее предопределенный тип информации об элементе. При использовании маркеров в качестве типа атрибута мы должны включить в описание одно из семи ключевых слов, идентифицирующих вид маркера. Так, например, в последней версии стандарта HTML в каждом тэге присутствует дополнительный параметр `ID`, при помощи которого задают идентификатор блока, обрабатываемого данным тэгом. В XML этот параметр заложен изначально и является атрибутом-маркером. Приведем пример объявления подобного атрибута:

```
<!ATTLIST my_element
        id ID #REQUIRED>
```

Данный пример демонстрирует присвоение элементу `my_element` атрибута `id`, имеющего предопределенный тип `ID`.

Ниже рассмотрим все семь упомянутых ранее предопределенных типов атрибутов-маркеров.

- Атрибут типа `ID` предназначен для указания уникального идентификатора данного элемента. Значение этого атрибута должно полностью удовлетворять соглашению об именах в XML. Естественно, идентификаторы всех экземпляров элементов в содержимом XML-документа должны быть разными, иначе нарушается условие уникальности. Если же это условие будет нарушено, XML-процессор укажет на ошибку в документе.

- Атрибут IDREF содержит идентификатор другого элемента, с которым данный элемент связан по смыслу. Атрибут данного типа предназначен для реализации двунаправленных ссылок — одной из наиболее широко разрекламированных возможностей, вошедших в стандарт XML версии 1.0. Конечно, элемент, идентификатор которого указан в данном атрибуте, должен присутствовать в XML-документе, иначе XML-процессор объявит о недействительности данного документа.
- Если данный элемент семантически связан с несколькими другими документами, то мы можем указать все их идентификаторы сразу в одном атрибуте. Для этого используется атрибут типа IDREFS. В его значении все идентификаторы связанных элементов перечисляются, разделенные пробелами.
- Атрибут ENTITY указывает на внешнюю сущность, связанную с данным элементом. В качестве значения этого атрибута указывается имя сущности. Подробно механизм сущностей мы рассмотрим в следующей части.
- Атрибут ENTITIES весьма похож на предыдущий. Разница лишь в том, что в его значении мы можем записать сразу несколько имен внешних сущностей.
- Атрибут NMTOKEN содержит любую последовательность символов из имени данного элемента.
- Идею предыдущего атрибута расширяет атрибут с именем NMTOKENS, который в своем значении может содержать сразу несколько подстрок имени данного элемента.

Мы рассмотрели возможные идентификаторы типов атрибута, размещаемые в объявлениях сразу после наименования. Осталось объяснить смысл ключевых слов, завершающих определение атрибута. Мы можем задать атрибут, который будет обязателен для применения в каждом экземпляре элемента, задать значение по умолчанию для каждого атрибута, и т. д. Всего имеется три стандартных модификатора для атрибутов.

- Модификатор #IMPLIED указывает на то, что для данного атрибута значение может быть не определено, то есть этот атрибут не является обязательным для применения.
- Модификатор #REQUIRED применяется, наоборот, для указания обязательных атрибутов. То есть, если у данного атрибута некоего элемента указан этот модификатор, то во всех экземплярах элемента в значимом содержимом XML-документа должно быть обязательно присвоено значение этому атрибуту.
- Модификатор #FIXED применяется в тех случаях, когда заданное для атрибута значение по умолчанию является фиксированным, то есть не поддается изменению. Проще говоря, атрибут должен иметь только указан-

ное значение. Понятно, что этот модификатор не применяется для атрибутов перечислимого типа.

Для каждого атрибута можно задать значение, применяемое по умолчанию. Причем оно может как замещать стандартный модификатор, так и использоваться совместно с ним. Рассмотрим еще раз наш пример.

```
<!ELEMENT text (#PCDATA)>
<!ATTLIST text
       create_date   CDATA #REQUIRED
       last_modified CDATA #IMPLIED
       author        CDATA #REQUIRED
       editor        CDATA "Kondukova E.V.">
```

Теперь нам видно, что атрибуты `create_date` и `author` являются обязательными, а атрибут `last_modified` может и не использоваться в элементах `text`. Для атрибута же `editor` предусмотрено значение по умолчанию `Kondukova E.V.`, ограниченное двойными кавычками, как строковая переменная. В этом примере значение по умолчанию не комбинируется с модификатором. Пример сочетания модификатора и значения по умолчанию может выглядеть следующим образом:

```
<!ATTLIST hyperlink
↳xlink:type CDATA #FIXED "simple">
```

В этом примере мы объявляем элемент `hyperlink` с атрибутом `xlink:type`, для которого установлено фиксированное значение `simple`. Следовательно, каждый раз, когда в тексте XML-документа будет использоваться элемент `hyperlink`, то его атрибут `xlink:type` будет всегда иметь значение `simple`, и изменить это значение в тексте XML-документа для какого-либо экземпляра элемента будет нельзя.

Атрибуты перечислимого типа мы можем комбинировать со списком условных обозначений (`NOTATION`). Механизм подобных списков мы будем рассматривать в одной из следующих частей. А сейчас нам необходимо просто привести пример подобного сочетания. После наименования атрибута мы указываем в качестве его типа ключевое слово `NOTATION`. Затем (как обычно в скобках) мы указываем сами условные обозначения, составляющие список. А после перечисления мы вписываем значение, используемое по умолчанию. Такой подход демонстрируется в следующем примере:

```
<!ATTLIST new_element
↳attr NOTATION (first|second|third) ”third”>
```

Естественно, в DTD-блоке должно присутствовать определение данного списка и всех определенных в нем элементов.

Итак, мы рассмотрели все правила создания атрибутов. Напоследок приведем фрагмент спецификации, описывающей создание атрибута:

```
[52]  AttlistDecl ::=  '<!ATTLIST' S Name AttDef* S? '>'
[53]  AttDef ::=  S Name S AttType S DefaultDecl
[54]  AttType ::=  StringType | TokenizedType | EnumeratedType
[55]  StringType ::=  'CDATA'
[56]  TokenizedType ::=  'ID' [ VC: ID ]
      [ VC: One ID per Element Type ]
      [ VC: ID Attribute Default ]
      | 'IDREF' [ VC: IDREF ]
      | 'IDREFS' [ VC: IDREF ]
      | 'ENTITY' [ VC: Entity Name ]
      | 'ENTITIES' [ VC: Entity Name ]
      | 'NMTOKEN' [ VC: Name Token ]
      | 'NMTOKENS' [ VC: Name Token ]
[57]  EnumeratedType ::=  NotationType | Enumeration
[58]  NotationType ::=  'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'
      [ VC: Notation Attributes ]
[59]  Enumeration ::=  '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'
      [ VC: Enumeration ]
[60]  DefaultDecl ::=  '#REQUIRED' | '#IMPLIED'
      ↳| (('#FIXED' S)? AttValue) [ VC: Required Attribute ]
      [ VC: Attribute Default Legal ]
      [ WFC: No < in Attribute Values ]
      [ VC: Fixed Attribute Default ]
```

Сущности

Сущности являются одним из ключевых понятий XML. Под ними обычно подразумевают единые и неделимые блоки информации, не анализируемые XML-процессором. Сущности могут представлять собой файлы с бинарной информацией, такой как рисунки, аудио- и видеофайлы, документы, обрабатываемые специализированными приложениями, и многое другое. Также в качестве сущностей используются блоки часто обновляемой информации. Достаточно такой блок объявить в качестве сущности в DTD-описании, и потом во всех XML-документах, подключенных к этому DTD-блоку, можно применять обозначение этой сущности. Таким образом, при необходимости единовременного изменения какого-либо часто применяемого элемента во всей совокупности документов достаточно изменить его в одном месте — в объявлении сущности, и изменения во все связанные с этим DTD-блоком

XML-документы будут внесены автоматически. Также сущности могут применяться и в самом DTD-блоке для оформления часто встречающихся последовательностей атрибутов или элементов, то есть выполнять обычную функцию текстовой подстановки. При помощи все тех же сущностей мы можем вставлять в текст XML-документа неотображаемые символы. Обобщая, можно сказать, что любая сущность является контейнером для каких-либо данных, помещаемых в XML-документ. Все эти возможности мы тщательно рассмотрим и приведем соответствующие примеры.

Итак, официально сущности являются особого рода элементами разметки, которые содержат объявление текста или данных, вставляемых в значимое содержимое XML-документа при помощи связанных с ними псевдонимов.

Сущности по своему типу делятся на текстовые, бинарные и параметрические. А по месту определения (декларирования) мы можем сущности делить на внешние и внутренние.

Любая сущность объявляется в DTD-блоке при помощи ключевого слова `ENTITY`. После него, отделенное пробелом, следует наименование сущности — ее *псевдоним*. А затем, уже в самом конце декларации мы вписываем значение сущности, заключенное в двойные кавычки. В качестве примера наиболее простой текстовой сущности приведем следующую конструкцию:

```
<!ENTITY cp "copyright disclaimer">
```

Теперь, если в тексте XML-документа мы вставим наименование этой сущности, обрамленное знаками амперсанда и точки с запятой, то оно при отображении документа браузером будет заменено на соответствующее словосочетание. То есть в тексте для вызова сущности `cp` мы должны использовать конструкцию `&cp;`.

Также существуют и предопределенные текстовые сущности. В значимом содержимом XML-документа мы можем применять только символы из набора ASCII. Все остальные символы могут быть вставлены в текст только при помощи сущностей. По существу, все кодировки представляют собой набор сущностей.

В ASCII-наборе есть несколько символов, которые не могут быть в явном виде вставлены в текст XML-документа. Это — служебные символы, применяемые в разметке кода XML-документа, такие как знаки "больше" и "меньше", знак амперсанда (`&`), а также одинарные и двойные кавычки. Для вставки этих символов в текст значимого содержимого XML-документа используются сущности `>`, `<`, `&`, `&apos;` и `&quot;` соответственно.

Бинарные сущности позволяют встраивать в текст XML-документа любые внешние файлы, от графических изображений до файлов Microsoft Office (не исключаются, естественно, и мультимедиа-файлы). В общем случае к XML-документу можно присоединить любые файлы, необходимо лишь связать

эти файлы с приложениями, способными их обрабатывать. Приведем пример сущности, которая подключает к документу графический GIF-файл:

```
<!ENTITY picture SYSTEM "images/pic1.gif" NDATA gif>
```

По определению, все бинарные сущности являются *внешними*, так как они встраивают внешние по отношению к XML-документу файлы. Поэтому после объявления имени, так называемого *псевдонима* сущности, следует ключевое слово `SYSTEM`, за которым указывается URL подключаемого файла. Эти правила достаточно прозрачны, так как не противоречат ничему, что мы узнали о разметке XML ранее. Но есть одно достаточно тонкое место. XML-процессор не знает о том, какой именно тип файла мы используем. Он не обязан автоматически распознавать тип файла по расширению. Поэтому для каждого типа файлов, применяемого в оформлении документов, необходимо уточнить программу, применяемую для обработки данного типа файлов. Для этих целей применяется механизм условных обозначений (`NOTATION`), который мы детально рассмотрим в следующей части. Здесь же мы приведем только маленький пример. Если мы в объявлении сущности `picture` использовали условное обозначение `gif`, указав его после ключевого слова `NDATA`, то в DTD-блоке должно присутствовать объявление для данного типа, подобное следующему:

```
<!NOTATION gif SYSTEM "http://www.mysite.ru/progs/gifview.exe">
```

В этом объявлении мы указываем, что для обработки файлов типа `gif` необходимо вызвать приложение, URL которого указан после ключевого слова `SYSTEM` в двойных кавычках.

В объявлении бинарной сущности необходимо внимательно следить за типом файла, указываемого после ключевого слова `NDATA`. Каким бы ни было расширение подключаемого файла, XML-процессор не будет принимать его во внимание. Тип устанавливается по соответствующему обозначению.

Однако есть одна возможность создавать и использовать бинарные внешние сущности без указания типа присоединяемого файла. Это возможно, если вставляются сторонние XML-файлы. При помощи этого средства мы можем собирать объемлющий XML-документ из других документов. Например, если у нас уже есть три XML-файла, содержимое которых необходимо объединить в одно целое, мы можем объявить их как внешние бинарные сущности, и вызвать в основном документе через их псевдонимы. Приведем соответствующий пример:

```
<!xml version="1.0">
<!DOCTYPE body [
<!ENTITY doc1 SYSTEM "http://www.parts.ru/part1.xml">
<!ENTITY doc2 SYSTEM "http://www.parts.ru/part2.xml">
<!ENTITY doc3 SYSTEM "http://www.parts.ru/part3.xml">
]>
```

При этом в теле документа следует использовать следующую конструкцию:

```
<body>
&doc1;
&doc2;
&doc3;
</body>
```

Тем не менее мы можем использовать в объявлении внешней бинарной сущности не только ключевое слово SYSTEM, но и PUBLIC. Но в этом случае мы должны подставлять только те файлы, которые рассматривались международной организацией по стандартизации. Хотя с другой стороны, так ли уж необходимо вам стандартизовывать свои данные?

Нам осталось рассмотреть третий вид сущностей — *параметрических*. Такие сущности применяются для обозначения часто обновляемых групп атрибутов или содержания элемента. Другими словами, здесь используется понятие тривиальной текстовой подстановки.

Такие сущности объявляются и вызываются внутри DTD-блока. Естественно, параметрическую сущность сначала необходимо объявить, и только потом — использовать.

Есть и еще одно отличие от обычных сущностей. Для вызова параметрических сущностей перед их псевдонимом мы должны поставить не знак амперсанда, а знак процента. Этот же знак процента ставится и при объявлении параметрической сущности.

Приведем пример. Нам необходимо объявить два разных элемента, причем у каждого из них частично будут совпадать наборы атрибутов. Эту-то совпадающую часть мы и объявим как параметрическую сущность, а затем используем ее в декларировании самих элементов.

```
<!ENTITY % common_atts "
            last_modified CDATA #IMPLIED
            description   CDATA #IMPLIED
            id            ID    #REQUIRED">
<!ELEMENT el_first (#PCDATA)>
<!ELEMENT el_second (#PCDATA)>
<!ATTLIST el_first
            special_attribute CDATA #IMPLIED
            %common_atts;>
<!ATTLIST el_second
            sex (male|female)
            %common_atts;>
```

Из примера видно, как объявляются и используются параметрические сущности. Следует отметить, что при декларировании параметрической сущности необходимо знак процента отделять пробелом от псевдонима сущности. Если пробела не будет, то XML-процессор выдаст сообщение об ошибке.

Теперь детально рассмотрим механизмы объявления внутренних и внешних сущностей. Объявление внутренних сущностей не представляет особых проблем. Вся информация, входящая во внутреннюю сущность, находится внутри одного DTD. Для декларирования внутренней сущности необходимо после псевдонима сущности указать само содержимое сущности. В случае использования внешних сущностей после псевдонима нужно дополнительно указать тип ссылки на содержимое сущности. Как и все внешние ресурсы, содержимое сущности может быть как обычным, так и стандартизованным. Если используется обычный внешний ресурс, то в объявлении после псевдонима мы указываем ключевое слово `SYSTEM`, а затем URL ресурса, являющегося содержимым ссылки. Если же содержимое сущности по какому-либо недоразумению было отослано в ISO, и прошло там рассмотрение, то после псевдонима мы вписываем ключевое слово `PUBLIC`, затем официальное наименование ресурса, и уж только после него — URL содержимого.

Разберем маленький пример, в текст которого включены три декларации. Первая из них объявляет обычную внутреннюю сущность, вторая — нестандартизованную внешнюю бинарную сущность, третья же обозначает внешнюю сущность, содержимое которой было одобрено международной организацией по стандартизации.

```
<!ENTITY cp "All rights are protected">
<!ENTITY pic SYSTEM "http://www.graphics.com/images/c3.gif" NDATA gif>
<!ENTITY firm PUBLIC "+//New Boundaries//Sweet Immersing//EN"
"http://www.bound.com/dtd/f.xml">
```

Теперь, когда мы усвоили особенности объявления и применения сущностей, узнали, что это такое, и для чего они нужны, разобрали несколько примеров, для порядка необходимо привести фрагмент официальной спецификации XML, содержащий правила описания сущностей. Что ж, вот он:

```
[67]  Reference ::=  EntityRef | CharRef
[68]  EntityRef ::=  '&' Name ';' [ WFC: Entity Declared ]
      [ VC: Entity Declared ]
      [ WFC: Parsed Entity ]
      [ WFC: No Recursion ]
[69]  PEReference ::=  '%' Name ';' [ VC: Entity Declared ]
      [ WFC: No Recursion ]
      [ WFC: In DTD ]
```

```
[70]  EntityDecl ::=  GEDecl | PEDecl
[71]  GEDecl ::=  '<!ENTITY' S Name S EntityDef S? '>'
[72]  PEDecl ::=  '<!ENTITY' S '%' S Name S PEDef S? '>'
[73]  EntityDef ::=  EntityValue | (ExternalID NDataDecl?)
[74]  PEDef ::=  EntityValue | ExternalID
[75]  ExternalID ::=  'SYSTEM' S SystemLiteral
      ↳| 'PUBLIC' S PubidLiteral S SystemLiteral
[76]  NDataDecl ::=  S 'NDATA' S Name [ VC: Notation Declared ]
```

Комментарии и условные обозначения

В этой части мы рассмотрим все остальные элементы XML. Оказывается, таких осталось немного — всего два. Это — комментарии и условные обозначения, о которых мы упоминали выше по тексту. Начнем с условных обозначений — они нам нужны больше.

Итак, условные обозначения являются специальными элементами DTD, позволяющими ассоциировать тип файла, встраиваемого в значимое содержимое XML-файла с приложением, которое будет обрабатывать эти файлы. XML-процессоры самостоятельно не занимаются отображением внешних файлов, таких как графика, аудио- или видеофайлы, в отличие от HTML-браузеров. Эта задача в XML возложена на "родные" приложения для этих файлов.

Условное обозначение объявляется при помощи ключевого слова `NOTATION`, после которого задается наименование типа файла (но не расширение, поскольку тип файла совсем не обязательно должен совпадать с его расширением) без ограничивающих кавычек, а затем указывается URL обрабатывающего приложения. Так как все приложения являются внешними ресурсами по отношению к данному XML-файлу, то, соответственно, необходимо использовать ключевые слова `SYSTEM` или `PUBLIC`.

Приведем пример связки внешней бинарной сущности и соответствующего ей условного обозначения. Внешняя сущность будет ссылаться на графический GIF-файл, а условное обозначение задаст URL приложени, которое будет отображать этот файл.

```
<!ENTITY pict SYSTEM "http://www.graphix.net/images/c2.gif" NDATA gif>
<!NOTATION gif SYSTEM "file://c:/Program Files/acdsystem/acdsee.exe">
```

Конечно, не нужно забывать, что если в объявлении внешней бинарной сущности после ключевого слова `NDATA` будет указан неверный тип файла, например `avi`, то будет вызван обработчик не для GIF-файлов, а именно для AVI-ресурсов, несмотря на расширение самого файла.

В официальной спецификации объявление условных обозначений регламентируется следующим образом:

```
[82]  NotationDecl ::=  '<!NOTATION' S Name S (ExternalID
      ↳| PublicID) S? '>'
```

Теперь переходим к комментариям. Сейчас уже ни у кого не возникает сомнений в их необходимости. Любой достаточно объемный код следует тщательно документировать. Иначе уже через месяц понять, что значат такие понятные и простые ранее обозначения, становится весьма сложно.

В XML комментарии обозначаются очень просто. Они ограничиваются последовательностями символов <!-- и -->. В качестве примера можно привести следующую конструкцию:

```
<!-- Это комментарий -->
```

Официальная спецификация XML предписывает оформлять комментарии следующим образом:

```
[15]  Comment ::=  '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
```

На этом мы заканчиваем рассмотрение элементов DTD и переходим к оформлению самих XML-документов.

Тело XML-документа

Теперь, после того, как мы полностью обозначили структуру документа, можно приступать к созданию самого документа. В конечном счете, пользователей интересует не разметка (markup), а его информационное наполнение (content). Как и в HTML, разметка значимого содержимого производится при помощи тэгов. Наименования тэгов совпадают с наименованиями элементов, объявленных в DTD-блоке. При этом должна полностью выдерживаться последовательность вложенности тэгов. То есть все содержимое документа должно быть заключено между тэгами, соответствующими основному элементу DTD — типу документа. Для элементов, вложенных в другие элементы, соответствующие им тэги должны точно так же вкладываться друг в друга с соблюдением иерархии. Данные правила можно проиллюстрировать следующимпримером цельного XML-документа:

```
<?xml version="1.0"?>
<!DOCTYPE body [
<!ELEMENT book (chapter)*>
<!ELEMENT chapter (#PCDATA)>
<!ENTITY name "Igor">
]>
```

```
<body>
<book>
<chapter>Hello, world. My name is &name;</chapter>
<chapter>Hello, world. It's me again</chapter>
</book>
</body>
```

В этом примере мы объявляем DTD-блок, в котором объявлен тип документа `body`, затем объявляем элемент `book`, составной частью которого является атомарный элемент `chapter`. Этот атомарный элемент может несколько раз встречаться в одном экземпляре элемента `book`.

После DTD-блока мы начинаем работать с самим документом. Так как основной элемент объявлен типом документа, мы, соответственно, сначала открываем тэг `<body>`, а в самом конце документа закрываем его. В этот блок мы вкладываем пару тэгов `<book>`, а в ней уже размещаем два набора тэгов `<chapter>`.

Разбирая пример, еще раз обратим внимание на использование обычной внутренней сущности. В DTD-блоке мы объявляем ее с псевдонимом `name`, а затем используем в текстовой строке, относящейся к одному из экземпляров элемента `chapter`.

Естественно, та информация, которую мы ограничиваем теми или иными тэгами, должна полностью соответствовать по типу тому элементу, к которому она относится. Есть только один тип элементов, для которого приведенный пример не будет работать. В том случае, если мы объявили пустой элемент, то есть при указании его типа в декларации этого элемента мы использовали ключевое слово `EMPTY`, то и тэг, соответствующий этому элементу, должен быть пустым. Но экземпляры пустых элементов — это не просто открывающий и закрывающий тэги, без какой-либо информации между ними. Для обозначения пустых элементов мы должны использовать соответствующие им пустые тэги. Их применение показано в следующем примере:

```
<!ELEMENT nothing EMPTY>
. . . . .
<nothing/>
```

Из примера видно, что пустые элементы отражаются в XML-документе при помощи одного тэга, в конце которого стоит наклонная черта.

Возникает вполне резонный вопрос, а для чего вообще нужны пустые элементы? Отвечаю — на самом деле экземпляр пустого элемента может хранить информацию. Она не будет отображена браузером, но она может быть обработана при помощи специализированных XML-приложений. Для того чтобы занести некую информацию в пустые элементы, используются их атрибуты.

Как вообще мы можем использовать атрибуты элементов в XML-документе? Рассмотрим пример:

```
<!ELEMENT computer EMPTY>
<!ATTLIST computer
       stone   CDATA        #IMPLIED       memory  CDATA        #IMPLIED
       sound   (yes|no)     #IMPLIED>
. . . . .
<computer stone="Coppermine" memory="256" sound="yes"/>
```

На этом примере видно, как мы сначала объявляем пустой элемент `computer` с набором атрибутов, которые мы позже уточняем в содержимом XML-документа. Мы используем тэг пустого элемента, в котором между наименованием тэга и завершающей наклонной чертой находятся все требуемые нам атрибуты.

Атрибуты в тэгах содержимого XML-документа используются для задания дополнительной информации, относящейся к данному экземпляру элемента. Как следует из примера, данные для атрибутов помещаются в тело тэга в виде `имя=значение`, причем значение заключается в двойные кавычки.

Теперь, когда мы знаем, как создавать значимое содержимое XML-документа, мы можем рассмотреть законченный документ в качестве примера.

Листинг 1.1. Файл 1-1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE body [
<!ELEMENT firm (name+, address+, management, phone*, e-mail,
               description)>
<!ELEMENT name        (#PCDATA)>
<!ELEMENT address     (#PCDATA)>
<!ELEMENT management  (#PCDATA)>
<!ELEMENT phone       (#PCDATA)>
<!ELEMENT e-mail      (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT paragraph   (#PCDATA)>
<!ATTLIST address
       property (real|official) #REQUIRED>
<!ATTLIST management
       owner  CDATA #IMPLIED
       CEO    CDATA #IMPLIED
       CFO    CDATA #IMPLIED
       CIO    CDATA #IMPLIED>
```

```
<!ENTITY copyright "Businee SuperBase Inc, 2000, All rights reserved">
]>
<body>
<firm>
<name>IBM</name>
<address property="real">unknown</address>
<management>respectable men</management>
<e-mail>ibm@ibm.com</e-mail>
<description>small hardware company</description>
</firm>
<firm>
<name>Microsoft</name>
<address property="real">Redmond, USA</address>
<management owner="Billy, great and awful">cool</management>
<e-mail>info@microsoft.com</e-mail>
<description>big software company</description>
</firm>
<paragraph>&copyright;</paragraph>
</body>
```

Прежде всего автор должен заявить, что при составлении документа он не придерживался исторической правды и реального положения дел в компьютерной индустрии.

Итак, в этом примере мы объявляем элемент `body` в качестве типа данного документа, то есть в качестве корневого элемента. Элемент `body` состоит из двух обычных — `paragraph` и `firm`. Элемент `paragraph` является атомарным, а в элемент `firm` включает дочерние субэлементы. Поскольку все субэлементы имеют "говорящие" имена, в особой расшифровке они не нуждаются.

Легко заметить, что для двух субэлементов объявлены наборы атрибутов, которые позволяют детализировать информацию, предназначенную для отображения или использования в этих элементах.

Также в DTD-блоке объявлена текстовая сущность с информацией об авторских правах. Так как мы не можем напрямую использовать какие-либо текстовые данные в XML-документе, мы специально создали элемент `paragraph`, который и будет служить контейнером для всех текстовых данных. В предпоследней строке кода XML-документа мы можем видеть, как используется предварительно объявленная текстовая сущность.

Внешний вид XML-документа, приведенного в листинге 1.1, полученный в браузере Internet Explorer, показан на рис. 1.1.

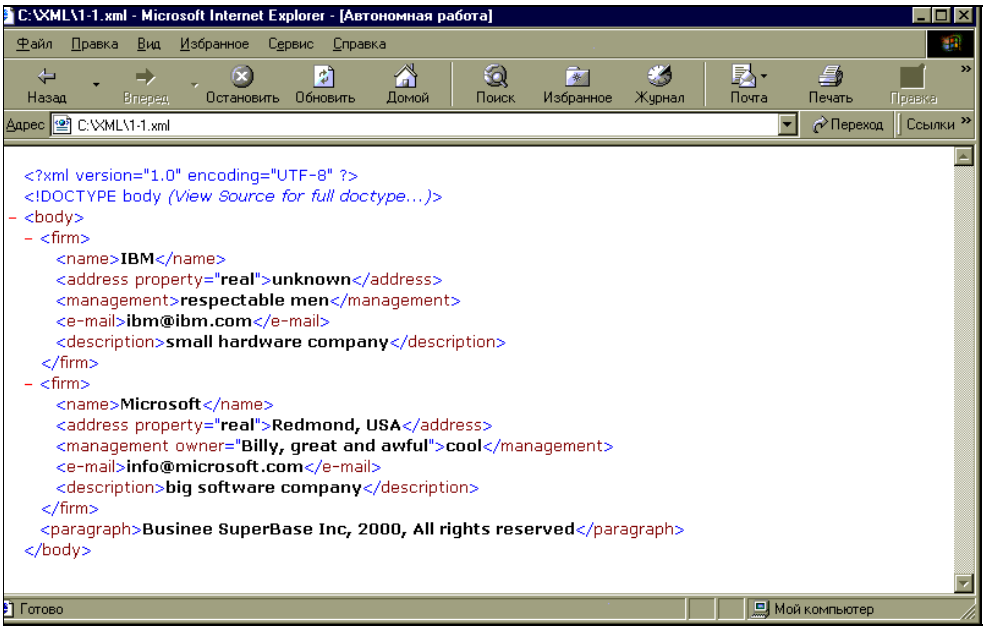


Рис 1.1. Внешний вид XML-документа при его просмотре в браузере Internet Explorer

Необходимо заметить, что DTD-блок не является обязательной частью для XML-документов. Так, если мы удалим из нашего примера весь DTD-блок, документ, тем не менее, будет отображаться практически так же. Не забудьте, что вместе с DTD-блоком мы обязаны будем удалить из значимого содержимого XML-документа все ссылки на сущности, объявленные в DTD-блоке. И конечно, нужно учитывать, что без DTD-блока XML-документ не будет считаться правильным (valid). А отображаться документ будет точно так же.

Вообще, о браузерах для XML-документов стоит поговорить особо. Для отображения документа 1-1.xml использовался Internet Explorer с внутренним номером версии 5.00.2614.3500. Этот браузер понимает XML, но еще не отображает его адекватно. На рис. 1.1 мы видим, что вместе с содержимым XML-документа выводится информация о его разметке, то есть видны все тэги. Таким образом мы можем использовать пятую версию Internet Explorer как анализатор XML-документов, так как он может показывать ошибки в загружаемых документах. Но как браузер он не подходит.

Наиболее распространенные на момент написания данной книги версии Netscape Navigator (4.x) также не могут отображать XML-документы. Но уже объявлено, что пятая версия Netscape Navigator, созданная на абсолютно новом движке Gecco, будет понимать и адекватно отображать XML-файлы. А поскольку пятая версия еще официально не вышла (хотя предварительные

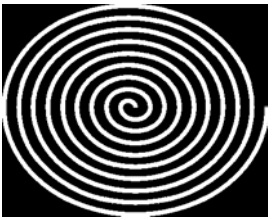
релизы уже появились), мы не можем с абсолютной уверенностью говорить, что скоро у нас будет полноценный XML-браузер от Netscape.

Возможно, что проблему в состоянии решить специализированные XML-браузеры, которые были разработаны мелкими компаниями. Их можно найти в Интернете без особого труда. Стоит, однако, отметить, что все эти браузеры были созданы не слишком успешными компаниями, что наложило свой отпечаток на спектр их возможностей. Налицо некоторое ограничение функциональности.

В плане воспроизведения XML-документов нельзя обойти еще один аспект. В HTML правила отображения тех или иных тэгов четко регламентированы. В XML же тэги мы создаем сами, поэтому и правила их отображения мы должны задавать самостоятельно. Для этого существуют особые технологии, такие как CSS (Cascading Style Sheets), унаследованная из HTML, и XSL (eXtensible Stylesheet Language), разработанная специально для XML. Их мы рассмотрим в четвертой и пятой главе соответственно.

А пока мы закончим рассмотрение стандарта XML и перейдем к технологии XLink, которая была разработана для создания гиперссылок нового поколения в XML-документах.

Глава 2



Расширенные гиперссылки — XLink

Умные гиперссылки

Навыки, приобретенные нами в первой главе, позволяют создавать удивительно структурированные документы. Но создаем мы эти документы, как правило, для опубликования в Интернете. А в Интернете, пожалуй, главную роль играют гиперссылки. Именно легкость перехода от одного документа к другому является важнейшей причиной, по которой WWW завоевала всемирную популярность.

Для создания гиперссылок в XML-документах существует свой язык — XLink.

Основной идеей XML было радикальное расширение возможностей HTML. Благодаря XLink не менее революционный прорыв был осуществлен и в области гиперссылок. Теперь связь между документами не обязана быть однонаправленной. При помощи технологии XLink мы можем создавать двунаправленные ссылки (туда — обратно), мультинаправленные ссылки, формировать кольцо ссылок на однородные ресурсы. Мы можем управлять поведением XML-браузера при осуществлении перехода по ссылкам. При помощи гиперссылок, созданных в XLink, мы в состоянии теперь заранее устанавливать режимы просмотра ресурсов, на которые эти гиперссылки указывают. Теперь мы умеем делать если не все, то очень многое.

Все гиперссылки в XML-документе являются обычными элементами XML. Спецификация Xlink предусматривает для них ряд предопределенных параметров. В подробностях эти параметры мы рассмотрим в следующих частях этой главы.

Одна из важнейших частей концепции новых гиперссылок (которые в XML называются "умными ссылками" (smart links)) — это тип указания ресурса, к которому обращена ссылка. Посредством гиперссылки мы можем как напрямую адресовать конкретный документ, так и указывать относительное положение ресурса. Для этого был создан специализированный язык для целеуказания, получивший название XPointer, который мы будем рассматривать в следующей главе.

На сегодня одной из основных проблем Интернета являются так называемые "висячие" ссылки, возникающие в том случае, когда ресурс, на который они указывают, был удален, изменен или перемещен. В такой ситуации ссылка чаще всего ведет в никуда, и браузер регистрирует стандартную "ошибку 404", возникающую при попытке перехода по оборванной ссылке. Знакомая картина, не правда ли?

Так вот, XLink если и не решает эту проблему полностью, то, по крайней мере, убирает некоторые весьма насущные сложности. Уже за одно это стоит использовать XML, если всех ранее перечисленных достоинств вам не достаточно. В конце концов, а зачем вы купили эту книгу?

Создание гиперссылок в XML

Как мы уже говорили, гиперссылки являются обычными элементами. Но слово "обычные" не означает "простые". Все ссылки подлежат должному оформлению. Элементы, которым отведена роль гиперссылок, имеют свои специфические атрибуты.

В XML мы можем создавать как обычные гиперссылки (simple links), знакомые нам еще по HTML, так и расширенные гиперссылки (extended links). А из готовых расширенных ссылок мы можем формировать группы расширенных ссылок (extended link groups) и иные виды гиперссылок.

Рассмотрим атрибуты элементов, необходимые для создания различных видов ссылок в XML.

У каждого элемента, которому отводится роль ссылки, вне зависимости от типа создаваемой ссылки, должен присутствовать атрибут `xlink:href`, значение которого определяет точку назначения гиперссылки, и атрибут `xlink:type`, в котором указывается тип гиперссылки. Простейший вариант объявления обычной гиперссылки выглядит следующим образом:

```
<!ELEMENT my_link ANY>
<!ATTLIST my_link
      xlink:type  CDATA  #FIXED    "simple"      xlink:href  CDATA
#REQUIRED>
```

Как вытекает из примера и усвоенного ранее материала, атрибуту `xml:type` всегда следует присваивать модификатор `#FIXED`, поскольку тип ссылки динамически не изменяется. Атрибут `xml:href`, в свою очередь, всегда должен присутствовать во всех экземплярах элемента `my_link`, следовательно, он всегда должен объявляться с модификатором `#REQUIRED`.

Этот маленький пример дал нам некоторое представление о принципах создания несложных ссылок. Теперь перейдем к рассмотрению более изощренных форм.

Ссылки бывают разные

Выше уже было сказано, что ссылки в XLink делятся на обычные и расширенные. Разумеется, подобное разбиение на категории не претендует на уникальность, но чаще всего применяется именно такая категоризация.

Итак, *обычные* ссылки (simple links) являются прямыми наследниками ссылок из HTML, так хорошо нам знакомых. В силу их простоты на такие ссылки не накладывается особых ограничений по внутреннему синтаксису. Нам достаточно указать их тип и ресурс, на который они указывают.

С *расширенными* ссылками (extended links) немного сложнее. В связи с тем, что их функциональность значительно выше, чем у обычных ссылок, на синтаксис расширенных ссылок накладываются некоторые правила. Прежде всего необходимо указать тип ссылки при помощи атрибута `xlink:type`, использовав для него предопределенное значение `extended`.

Каждая расширенная ссылка является элементом, который включает в себя другие элементы в качестве содержимого. А уже с помощью входящих в расширенную ссылку субэлементов мы управляем действием нашей ссылки. Рассмотрим пример:

```
<!ELEMENT exlink (source)+>
<!ELEMENT source ANY>
<!ATTLIST exlink
       xlink:type CDATA #FIXED "extended">
<!ATTLIST source
       xlink:type CDATA #FIXED "locator"
       xlink:href CDATA #REQUIRED>
. . . . . .
<exlink>
<source xlink:href="http://www.site1.com">First site</source>
<source xlink:href="http://www.site2.com">Second site</source>
<source xlink:href="http://www.site3.com">Third site</source>
</exlink>
```

В этом примере мы объявляем расширенную ссылку `exlink`, в которую входят несколько субэлементов `source` типа `locator`. В результате эта расширенная гиперссылка позволяет адресовать сразу три внешних ресурса. Именно для таких целей и применяются элементы типа `locator`.

Также в расширенную гиперссылку могут входить элементы типа `arc`, определяющие правила прохождения гиперссылок, элементы типа `title`, позволяющие устанавливать отображаемые метки для расширенных гиперссылок, и элементы типа `resource`, адресующие внутренние ресурсы. Субэлементы последнего типа могут входить в расширенную гиперссылку как поодиноч-

ке, так игруппами. Рассмотрим пример, включающий в себя все указанные типы элементов. Пример разбит на два блока, первый из которых представляет собой объявление расширенной гиперссылки.

```
<!ELEMENT courseload ((tooltip|person|course|gpa|go)*)>
<!ATTLIST courseload
  xlink:type      (extended)      #FIXED "extended"
  xlink:role      CDATA           #IMPLIED
  xlink:title     CDATA           #IMPLIED>
<!ELEMENT tooltip ANY>
<!ATTLIST tooltip
  xlink:type      (title)         #FIXED "title"
  xml:lang        CDATA           #IMPLIED>
<!ELEMENT person EMPTY>
<!ATTLIST person
  xlink:type      (locator)       #FIXED "locator"
  xlink:href      CDATA           #REQUIRED
  xlink:role      CDATA           #IMPLIED
  xlink:title     CDATA           #IMPLIED
  xlink:label     NMTOKEN         #IMPLIED>
<!ELEMENT course EMPTY>
<!ATTLIST course
  xlink:type      (locator)       #FIXED "locator"
  xlink:href      CDATA           #REQUIRED
  xlink:role      CDATA           #FIXED
       "http://www.example.com/linkprops/course"
  xlink:title     CDATA           #IMPLIED
  xlink:label     NMTOKEN         #IMPLIED>
<!ELEMENT gpa ANY>
<!ATTLIST gpa
  xlink:type      (resource)      #FIXED "resource"
  xlink:role      CDATA           #FIXED
       "http://www.example.com/linkprops/gpa"
  xlink:title     CDATA           #IMPLIED
  xlink:label     NMTOKEN         #IMPLIED>
<!ELEMENT go EMPTY>
<!ATTLIST go
  xlink:type      (arc)           #FIXED "arc"
  xlink:arcrole   CDATA           #IMPLIED
  xlink:title     CDATA           #IMPLIED
```

```
  xlink:show      (new
                  |replace
                  |embed
                  |other
                  |none)          #IMPLIED
  xlink:actuate   (onLoad
                  |onRequest
                  |other
                  |none)          #IMPLIED
  xlink:from      NMTOKEN         #IMPLIED
  xlink:to        NMTOKEN         #IMPLIED>
```

А использовать это объявление можно при помощи следующего кода:

```
<courseload xlink:title="Course Load for Pat Jones">
  <person
    xlink:href="students/patjones62.xml"
    xlink:label="student62"
    xlink:role="http://www.example.com/linkprops/student"
    xlink:title="Pat Jones" />
  <person
    xlink:href="profs/jaysmith7.xml"
    xlink:label="prof7"
    xlink:role="http://www.example.com/linkprops/professor"
    xlink:title="Dr. Jay Smith" />
  <!-- more remote resources for professors, TAs, etc. -->
  <course
    xlink:href="courses/cs101.xml"
    xlink:label="CS-101"
    xlink:title="Computer Science 101" />
  <!-- more remote resources for courses, seminars, etc. -->
  <gpa xlink:label="PatJonesGPA">3.5</gpa>
  <go
    xlink:from="student62"
    xlink:to="PatJonesGPA"
    xlink:show="new"
    xlink:actuate="onRequest"
    xlink:title="Pat Jones's GPA" />
  <go
    xlink:from="CS-101"
```

```
    xlink:arcrole="http://www.example.com/linkprops/auditor"
    xlink:to="student62"
    xlink:show="replace"
    xlink:actuate="onRequest"
    xlink:title="Pat Jones, auditing the course" />
  <go
    xlink:from="student62"
    xlink:arcrole="http://www.example.com/linkprops/advisor"
    xlink:to="prof7"
    xlink:show="replace"
    xlink:actuate="onRequest"
    xlink:title="Dr. Jay Smith, advisor" />
</courseload>
```

Это достаточно сложный пример, и его необходимо разобрать обстоятельно. В начальном блоке мы объявляем головной элемент `courseload`, который является расширенной гиперссылкой. В него могут входить один или несколько субэлементов, каждый из которых мы объявляем несколько позже.

Здесь мы используем три различных атрибута для расширенной гиперссылки, а именно: `xlink:type`, `xlink:role` и `xlink:title`. С атрибутом `xlink:type` мы уже знакомы, а остальные мы внимательно рассмотрим в последующих частях этой главы. Пока же только скажем, что атрибут `xlink:role` позволяет задавать роль ссылки, а атрибут `xlink:title` — ее заголовок.

Теперь рассмотрим субэлементы, используемые в гиперссылке. Элемент `tooltip` имеет тип `title`, и ему присвоен также пока незнакомый нам атрибут `xml:lang`, при помощи которого указывается язык элемента.

Следующий субэлемент — `person`. Он является пустым, как нетрудно заметить из его объявления. Общий тип (`xlink:type`) этого элемента объявлен как `locator`. *Локаторами* обычно называют фрагменты расширенных гиперссылок, идентифицирующих ресурс. На этот ресурс указывает или может указывать расширенная гиперссылка. По сути, локаторы являются адресами ресурсов.

Необязательными атрибутами для данного элемента являются `xlink:role`, `xlink:title` и `xlink:label`.

Элемент `person` в нашем примере используется для указания на ресурс, который содержит персональную информацию о личности, как это видно из кода значимого содержимого XML-документа. Если быть точным до конца, то этим элементом адресуется какой-либо студент или преподаватель, поскольку наш пример ориентирован на применение в рамках предметной области, связанной с высшим образованием.

Следующий элемент, входящий в состав расширенной ссылки, предназначен для указания на ресурс, на котором размещено описание какого-либо обучающего курса, включенного в учебную программу. В субэлементе `course`, также имеющем тип `locator`, мы помимо обязательного атрибута `xlink:href` объявляем набор дополнительных атрибутов. Как и для предыдущего субэлемента, в этот список входят атрибуты `xlink:role`, `xlink:title` и `xlink:label`. Но есть и отличие. Атрибут `xlink:role` теперь имеет модификатор `#FIXED` и значение по умолчанию, которым является URL конкретного ресурса. Данный атрибут отвечает за так называемую "роль" субэлемента, а по указанному URL расположен ресурс с ее описанием.

Следующий субэлемент — `gpa`, входит в расширенную ссылку в качестве ресурса, о чем свидетельствует значение `resource`, приписанное фиксированному (`#FIXED`) атрибуту `xlink:type`. В нашем случае этот ресурс описывает некие свойства того объекта, на который указывает расширенная ссылка. Вообще говоря, *ресурсами* называют некие сервисы или объекты, на которые указывает гиперссылка. Но здесь мы сталкиваемся с примером так называемого "локального ресурса", который является частью ссылки.

В субэлемент `gpa` мы, как и во все ранее рассмотренные, включили набор из трех дополнительных атрибутов.

Нам осталось рассмотреть объявление последнего субэлемента, входящего в состав расширенной ссылки. Он носит наименование `go` и применяется для описания процесса прохождения данной гиперссылки. Так как вся информация о процессе прохождения гиперссылки задается обычно при помощи атрибутов, то данный элемент объявлен пустым (`EMPTY`). Для этого субэлемента атрибут `xlink:type` установлен в значение `arc` (контейнер). В элементах типа *контейнер* хранится служебная информация о расширенной гиперссылке.

В объявлении этого элемента мы используем специализированный атрибут `xlink:arcrole`, который, по существу, является полным аналогом атрибута `xlink:role`, используется исключительно для элементов типа `arc`. Необязательный атрибут `xlink:title` нам уже знаком по объявлениям предыдущих субэлементов. А вот остальные атрибуты требуют пристального рассмотрения.

Атрибут перечислимого типа `xlink:show` характеризует принцип отображения ресурса, на который указывает гиперссылка, в XML-браузере. Предустановленные значения, которые могут быть приписаны этому атрибуту, мы рассмотрим в нижеследующих частях.

Атрибут `xlink:actuate` также имеет перечислимый тип. Смысл его применения заключается в возможности указывать момент перехода по ссылке. В HTML переход осуществлялся только тогда, когда пользователь производил щелчок по гиперссылке. В XML мы можем обойти это ограничение как раз при помощи атрибутов данного типа.

Атрибуты `xlink:from` и `xlink:to` позволяют указывать информацию о том, откуда и куда осуществляется переход. То есть, это некие аналоги кнопок **Back** и **Forward** обычных HTML-браузеров. При помощи этих атрибутов обычно формируются кольца гиперссылок.

Теперь, после того, как мы рассмотрели объявление расширенной ссылки `courseload`, поглядим, как она используется в XML-документе. С ее помощью можно выстроить целую систему взаимоотношений в предметной области университетского обучения.

В начале рабочего кода мы ставим открывающий тэг `courseload` с атрибутом `xlink:title`. В него мы вкладываем пустой тэг `person`, который описывает студента по имени Pat Jones с идентификационным номером 62. Атрибут этого тэга `xlink:href` указывает на XML-документ с URL `students/patjones62.xml`. При этом роль данного экземпляра элемента `person` описывается документом, который находится по адресу `http://www.example.com/linkprops/student`.

Затем мы создаем еще один экземпляр элемента `person`. На сей раз нас интересует профессор Jay Smith. По аналогии с предыдущим тэгом `person` можно легко разобраться в атрибутах данного экземпляра элемента.

После этих двух тэгов мы инициируем ссылку на курс, входящий в программу обучения студента Pat Jones, при помощи тэга `course`.

Посредством тэга `gpa` мы дополнительно указываем среднюю оценку нашего студента по этому курсу. Следует обратить внимание на то, что элемент `gpa` не является пустым, поэтому в коде должны присутствовать как открывающий, так и закрывающий тэги.

В заключение мы вписываем три тэга `go`, налаживающие взаимосвязи между всеми сущностями выбранной предметной области. Первый тэг устанавливает прохождение ссылки от студента к его средней оценке, второй — от курса к студенту, указывая при этом, что данный курс изучает наш студент Pat Jones, третий — от студента к его научному руководителю, которым является все тот же Jay Smith.

Для двух последних экземпляров элементов `go` принудительно указывается роль переходов при помощи атрибута `xlink:arcrole`.

Теперь, после того, как мы рассмотрели этот достаточно объемный пример, мы уже готовы к созданию собственных расширенных ссылок, которые, как мы убедились, позволяют связывать воедино множество ресурсов, указывая при этом их отношение друг с другом, правила адресации и переходов между ними.

А сейчас перейдем к более тщательному описанию элементов, входящих в расширенные гиперссылки.

Локальные ресурсы

Локальным (внутренним) ресурсом называют такой ресурс, который является частью расширенной гиперссылки. Локальные ресурсы определяются при помощи субэлементов, включаемых в состав расширенной гиперссылки, посредством выставления атрибута `xlink:type` в значение `resource`. В выше приведенном примере такой элемент носил наименование `gpa`.

В подобные элементы мы можем помещать некоторое содержимое (в примере этот элемент тоже не был пустым), которое в общем случае может и не иметь отношения к самой ссылке. То есть это содержимое не обязательно должно являться служебной информацией, необходимой для обработки ссылки.

Тем не менее, элементы подобного типа могут иметь семантические атрибуты, такие как `xlink:role` и `xlink:title`, или атрибут `xlink:label`, используемый при описании прохождения ссылок.

Внешние ресурсы

Внешний ресурс, как и локальный, является частью расширенной ссылки, но при этом позволяет гиперссылке адресовать другие XML-документы и иные ресурсы.

Объявляются внешние ресурсы при помощи элементов, у которых атрибут `xlink:type` установлен в значение `locator`.

Подобные элементы могут иметь любое содержание. Вернемся еще раз к примеру создания группы расширенных ссылок, каждая из которых указывает на внешний ресурс.

```
<!ELEMENT exlink (source)+>
<!ELEMENT source ANY>
<!ATTLIST exlink
       xlink:type CDATA #FIXED "EXTENDED">
<!ATTLIST source
       xlink:type CDATA #FIXED "LOCATOR"
       xlink:href CDATA #REQUIRED>
. . . . . .
<exlink>
<source xlink:href="http://www.site1.com">First site</source>
<source xlink:href="http://www.site2.com">Second site</source>
<source xlink:href="http://www.site3.com">Third site</source>
</exlink>
```

Как мы можем видеть, внутри каждой пары открывающих и закрывающих тэгаов `source` находится некий содержательно значимый тэг, который и идентифицирует для пользователя какой-либо внешний ресурс.

Элементы-локаторы должны иметь при объявлении атрибут `xlink:href`, который подлежит обязательному заполнению в каждом экземпляре этого элемента. Атрибуты `xlink:role`, `xlink:title` и `xlink:label` являются для локаторов необязательными.

Локаторы указывают на удаленные ресурсы при помощи URI (Universal Resource Identifier), который расширяет понятие URL. URI какого-либо ресурса формируется из его URL и выражения языка XPointer, созданного в рамках XML специально для этой цели. Этот язык мы рассмотрим в третьей главе.

Правила прохождения ссылок

Используя HTML, мы не могли уклониться от фиксированного правила прохождения ссылки. При щелчке на гиперссылке, размещенной в теле документа, в браузер загружался ресурс, на который указывала ссылка. Единственным "глотком свободы" была возможность указывать окно просмотра или фрейм, в который предстояло загружать связанный документ.

XML принес нам новые возможности и в этой области. Теперь мы можем полностью управлять процессом перехода по ссылке или даже по нескольким ссылкам сразу, если будем использовать расширенную ссылку на группу ресурсов.

В XML *правила прохождения* (traversal rules) оформляются в виде элементов, входящих в состав расширенной ссылки. Их атрибуту `xlink:type` обязательно приписывается значение `arc`.

Такие элементы могут иметь любое содержимое, но обычно эта возможность не используется. Правила прохождения являются сугубо служебной информацией, поэтому далеко не всегда есть смысл отображать их в XML-браузере.

Элементы типа `arc` могут иметь атрибуты *прохождения*, такие как `xlink:from` и `xlink:to`, атрибуты *поведения*, такие как `xlink:show` и `xlink:actuate`, и семантические атрибуты `xlink:arcrole` и `xlink:title`.

Атрибуты прохождения характеризуют пары ресурсов, между которыми осуществляется переход. При этом в качестве значений атрибутов `xlink:from` и `xlink:to` указываются значения атрибутов `xlink:label` соответствующих ресурсов.

Атрибуты поведения описывают действия XML-приложения, которое обрабатывает данный XML-документ в процессе перехода по ссылке. Стандартные варианты поведения XML-приложений и XML-браузеров мы рассмотрим ниже, когда придет пора обсудить сами атрибуты поведения.

В нашем большом примере, который мы рассматривали в *разд. "Ссылки бывают разные"*, в качестве элементов, управляющих правилами прохождения расширенной ссылки, мы использовали элементы `go`. Но в том примере роль и механизм действия таких элементов прослеживаются не совсем отчетливо. Приведем дополнительный пример:

```
<extendedlink xlink:type="extended">
  <loc xlink:type="locator" xlink:href="..." xlink:label="parent"
  ↳xlink:title="p1" />
  <loc xlink:type="locator" xlink:href="..." xlink:label="parent"
  ↳xlink:title="p2" />
  <loc xlink:type="locator" xlink:href="..." xlink:label="child"
  ↳xlink:title="c1" />
  <loc xlink:type="locator" xlink:href="..." xlink:label="child"
  ↳xlink:title="c2" />
  <loc xlink:type="locator" xlink:href="..." xlink:label="child"
  ↳xlink:title="c3" />
  <go xlink:type="arc" xlink:from="parent" xlink:to="child" />
</extendedlink>
```

В данном случае мы при помощи элемента `go`, задающего правила прохождения расширенной гиперссылки, позволяем осуществлять переходы от любого элемента-локатора с меткой `parent` к любому элементу-локатору с меткой `child`. При этом все остальные направления перехода заблокированы. Таким образом и осуществляется управление переходами между ресурсами, входящими в сферу действия расширенной гиперссылки.

Если бы мы удалили из тэга `go` атрибут `xlink:from`, оставив только `xlink:to`, тем самым мы бы разрешили переход от всех элементов-локаторов ко всем элементам, у которых атрибут `xlink:label` имеет значение `child`.

Есть некоторое ограничение, накладываемое на элементы типа `arc`. У каждого такого элемента должна быть уникальная пара значений атрибутов `xlink:from` и `xlink:to`. Все остальное — в ваших руках.

Идентифицирующие элементы ссылок

Расширенная гиперссылка всегда может быть идентифицирована. Для этой цели применяется специализированный элемент, входящий в ее состав. Подобные элементы имеют значение `title` присваиваемое атрибуту `xlink:type`. Необходимо отличать элементы типа `title` от одноименных атрибутов. Атрибуты применяются для идентификации какого-либо элемента XML-процессором, а элемент `title` содержит информацию, которая

предназначена для пользователя. То есть с помощью элементов со значением `title` организуется отображение информации о расширенной гиперссылке, ресурсе или локаторе в XML-браузере. Изначально такие элементы задумывались для нужд локализации XML-документов. Я бы хотел обратить внимание на то, что теперь мы локализуем не только программные приложения, но и документы. Связано это с тем, что XML-документы имеют как минимум равную ценность контента и механизмов работы, заложенных в структуру документа.

В качестве примера можно привести следующую конструкцию:

```
<!ELEMENT advisorname (name)>
<!ATTLIST advisorname
  xlink:type      (title)          #FIXED "title"
  xml:lang        CDATA            #IMPLIED>
```

Атрибут типа элемента

Мы переходим к рассмотрению всех предопределенных атрибутов, которые используются в элементах-ссылках.

Самым актуальным, первым из них является, само собой, атрибут с наименованием `xlink:type`. Чаще всего совместно с этим атрибутом используется модификатор `#FIXED`, что, впрочем, не является необходимым условием. У данного атрибута есть список предустановленных значений. Иные значения не будут распознаваться XML-процессором, и, следовательно, данный элемент не будет считаться ссылкой.

Атрибут `xlink:type` позволяет задавать и обозначать тип ссылки, которая будет создаваться данным элементом. Для этого атрибута могут быть использованы значения `simple`, `extended`, `locator`, `arc`, `resource`, `title` или `none`.

Механизм действия всех значений, кроме последнего, мы уже рассмотрели в предыдущих частях. Значение `none` применяется в тех случаях, когда в XML-документе данный элемент будет использоваться не только как ссылка, но и в качестве обычного элемента. Объявление подобного элемента будет выглядеть следующим образом:

```
<!ATTLIST commandname
  xlink:type      (simple|none)    #REQUIRED
  xlink:href      CDATA            #IMPLIED>
```

В этом примере мы объявляем, что в тексте XML-документа элемент `commandname` может использоваться либо в качестве простой ссылки, либо как обычный элемент разметки содержимого.

Атрибут `xlink:type` является обязательным для всех элементов, которые будут действовать в качестве ссылок.

Атрибут целеуказания

Еще одним обязательным атрибутом для всех элементов-ссылок является атрибут целеуказания с именем `xlink:href`. В качестве значения для этого атрибута мы указываем URI того ресурса, на который указывает данная ссылка. Кстати, полное описание спецификации URI содержится в документе RFC 2396, который, как и все RFC, более чем доступен через Интернет.

Мы уже не раз рассматривали примеры декларирования и использования данного атрибута, но эта часть не может считаться полной, если мы все-таки не поместим в нее хоть какой-нибудь пример.

```
<!ATTLIST simplelink
  xlink:href      CDATA      #REQUIRED
```

Так как этот атрибут является обязательным для любого элемента, который используется в качестве ссылки, то в его объявлении используется модификатор `#REQUIRED`.

Семантические атрибуты

Существует группа необязательных атрибутов, которые позволяют неким образом идентифицировать ту или иную ссылку, тот или иной экземпляр соответствующего элемента. Их принято называть *семантическими* атрибутами. В эту группу входят атрибуты `xlink:title`, `xlink:role` и `xlink:arcrole`.

Значение атрибута `xlink:title` представляет собой символьную информацию, используемую для дополнительной, чаще всего визуальной, идентификации экземпляра элемента-ссылки. Если говорить более понятным языком, это значение является описанием ссылки, которое показывается пользователю. То есть, попросту подсказка, обычный хинт. Чаще всего XML-браузеры отображают эту подсказку, если пользователь наводит курсор мыши на отображение экземпляра элемента-ссылки в окне просмотра.

Атрибуты `xlink:role` и `xlink:arcrole` идентичны по своему значению и механизму действия. Единственное их отличие в том, что атрибут `xlink:role` применяется для элементов типа `simple`, `extended`, `locator` и `resource`, а атрибут `xlink:arcrole` — для элементов типа `arc`.

Значения этих атрибутов обязаны быть указателями на некие ресурсы, описывающие роль данных ссылок, то есть иметь формат URI.

Все три вышеописанных атрибута являются опциональными, необязательными. Поэтому при их декларировании мы применяем модификатор `#IMPLIED`.

Поведенческие атрибуты

В эту группу входят два атрибута, которые управляют поведением XML-приложений, в том числе и XML-браузера, при отображении ресурса ссылки. К *поведенческим* атрибутам относятся атрибуты `xlink:show` и `xlink:actuate`. Эти атрибуты применяются к элементам-ссылкам, имеющим тип `simple` или `arc`.

Ресурс, на который указывает ссылка, может быть обработан несколькими способами. Выбор метода отображения ресурса зависит от значения атрибута `xlink:show`. У этого атрибута существует пять фиксированных значений, которые регулируют обработку и отображение целевого ресурса ссылки. Значение `embed` указывает на то, что содержимое целевого ресурса необходимо встроить в текущий документ прямо с того места, где размещена ссылка. Если необходимо целевой ресурс отобразить в новом окне браузера, применяется значение `new`. А если мы используем значение `replace`, то содержимое текущего документа будет заменено содержимым целевого ресурса ссылки при ее активации. Значение `other` мы используем, если наше XML-приложение будет отображать ресурс, на который указывает ссылка, каким-либо иным способом. Описание поведения XML-браузера при использовании этого значения атрибута `xlink:show` в текущую версию стандарта не вошло, и вряд ли войдет, так как это значение явно добавлено в спецификацию как некий слот для нестандартных реакций. Если же ресурс, на который указывает ссылка, мы вообще не собираемся отображать, используется значение `none` (но с другой стороны, кому и зачем нужна такая ссылка?).

В XML мы можем также управлять процессом активации ссылок. В HTML ссылка активировалась только пользователем. Переход осуществлялся только в том случае, если удаленный пользователь отдавал команду. В XML мы, в дополнение к этому механизму, можем автоматически активировать ссылку. Для этого используется параметр `xlink:actuate`, у которого есть четыре предустановленных значения. Значение `onRequest` указывает на то, что загрузка содержимого целевого ресурса ссылки не будет производиться до тех пор, пока пользователь сам явно не даст команду. А вот значение `onLoad` следует использовать в тех случаях, когда мы хотим, чтобы переход по ссылке и загрузка документа произошла автоматически, как только XML-анализатор обнаружит эту ссылку в процессе загрузки и отображения текущего ресурса. При помощи ссылок такого типа мы можем без особых усилий формировать объемные XML-документы из отдельных небольших частей.

Мы также можем задавать для этого атрибута значения `other` и `none`. При этом поведение XML-браузера ничем не отличается от описанного при использовании их двойников в атрибуте `xlink:show`.

Атрибуты прохождения ссылки

Мы уже упоминали пресловутые "правила прохождения" ссылки (traversal rules). Они задают поведение XML-приложения при перемещении по ссылке или целой цепочке ссылок. Для управления этим процессом применяются атрибуты *прохождения* (traversal attributes).

Атрибут `label` предназначен для использования в элементах типа `resource` и `locator`. Атрибуты `to` и `from` могут использоваться только в элементах типа `arc`. Все атрибуты, входящие в эту группу, являются опциональными, и их наличие в объявлении элемента-ссылки не требуется. Но иногда бывает полезно. Немного ранее мы видели, как эти атрибуты позволяют организовывать группу взаимосвязанных элементов-ссылок.

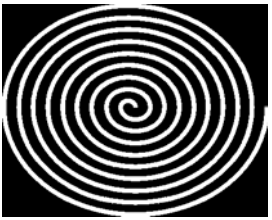
Атрибут `label`, как мы могли заметить, имеет тип `NMTOKEN`, благодаря чему возможна идентификация того или иного экземпляра элемента, например, присвоением "токенизированному" атрибуту значения `ID`.

Атрибуты `from` и `to` содержат значения атрибутов того элемента `label`, с которого осуществляется переход по ссылке, и элемента, куда он производится соответственно.

Теперь мы знаем, как объявляются элементы ссылки, и какие у них бывают атрибуты. На этом можно было бы и закончить главу, но один момент остался недосказанным. Мы перечислили лишь обязательные и предустановленные атрибуты для элементов-ссылок. Но не следует забывать, что все ссылки являются обычными (ну, почти обычными) элементами, и, следовательно, мы всегда можем добавить туда те атрибуты, которые нам нужны. Это же XML! Теперь нас почти ничто не ограничивает. Мы можем создавать настолько сложные иерархические структуры, насколько захотим. Главное — не забывать их документировать.

Итак, мы рассмотрели правила и методы создания гиперссылок в XML. Но это еще далеко не все. Каждая гиперссылка должна куда-то вести (хороший девиз для Интернета, не правда ли?). В этом плане XML намного расширяет наши возможности. Теперь помимо обычных URL, которые указывают на конкретные удаленные ресурсы, мы можем использовать URI (Universal Resource Identifier), описываемые при помощи средств языка целеуказания для XML — XPointer. Этому языку целиком и полностью посвящена следующая глава книги.

Глава 3



Технология идентификации ресурсов — XPointer

Предназначение

XPointer (XML Pointer Language), как мы уже говорили, является особым языком, описывающим местонахождение тех или иных ресурсов. С одной стороны, у нас уже есть механизм, который позволяет описывать местонахождение ресурсов. Мы называем его URL. Зачем, в таком случае, понадобилась его модификация?

Выражения языка XPointer позволяют создавать так называемые URI (Universal Resource Identifier), которые являются надстройками над URL. В связи с тем, что HTML не производит контроля целостности ссылок, мы вынуждены раз за разом натыкаться на оборванные гиперссылки, ведущие в никуда. Со ссылками, организованными по правилам технологии XPointer, подобная ситуация будет встречаться намного реже. Язык XPointer позволяет не только адресовать внешние документы, но и максимально точно описывать местонахождение отдельных частей внутри какого-либо документа путем создания некоей иерархической структуры выражений. При этом можно создавать ссылки не только на какие-либо поименованные закладки, как в HTML, но и на произвольно выбранное содержание. Например, мы можем указать ссылку на конкретное слово, экземпляр элемента или значение атрибута в XML-документе.

XPointer является надстройкой над языком XPath. В отличие от него XPointer позволяет дополнительно адресовать отдельные экземпляры элементов и целые разделы, умеет обрабатывать текстовые строки и устанавливать точку перехода в зависимости от результата анализа текстовой строки, использовать идентификаторы URI без ограничений в XML-документе.

Основные правила

Итак, сначала — несколько фактов, которые указаны в спецификации XPointer. Язык XPointer позволяет адресовать документы и их фрагменты, являющиеся XML-документами. То есть, говоря официальным языком, эти

документы должны иметь тип `text/xml` или `application/xml`. Все выражения XPointer должны записываться с использованием символов Unicode. Но если последних не хватает (хотя трудно представить ситуацию, когда для кодирования текстовой информации не хватает набора символов Unicode) или нет возможности использовать эти символы напрямую, мы можем применять так называемые Escape-последовательности. На самом деле, они не являются истинными Escape-последовательностями, как, например, команды для принтеров, но, тем не менее, они имеют такое название в спецификации. Будем придерживаться официальной терминологии. (В XML мы Escape-последовательности относили к символьным сущностям.) Так, например, мы не можем явно выписывать в выражениях XPointer символы больше-меньше, которые, кстати, в англоязычной терминологии называют брекетами, и символ амперсанда. Для них могут использоваться стандартные символьные сущности, которые мы рассмотрели в предыдущих главах. Но мы можем представить любой символ, входящий в выражения языка XPointer, в виде Escape-последовательности. В начале последовательности ставится символ процента, а затем заносится порядковый номер кода символа Unicode в шестнадцатеричном виде. То есть правила создания Escape-последовательностей полностью совпадают с правилами построения символьных сущностей.

Теперь рассмотрим основные термины, которые нашли применение в XPointer.

- *Субресурс* (sub-resource) — основная единица адресации языка XPointer. Обычно это часть целого XML-ресурса, такая как отдельная глава или абзац. Именно для адресации подобных мелких частей и предназначен XPointer.
- *Набор указателей* (location-set) — упорядоченный список точек адресации XML-документа, в который входят все адресуемые субресурсы различных типов.
- *Точка* (point) — место в XML-документе, где конкретно расположен адресуемый субресурс.
- *Интервал* (range) — часть содержимого XML-документа, размещенная между двумя точками, адресованными при помощи выражений XPointer.

Именно в этих терминах ведется объявление конструкций XPointer.

Теперь разберем типы ошибок, которые распознает XML-процессор при анализе выражений языка XPointer.

- *Синтаксическая ошибка* (syntax error) возникает в том случае, если выражение XPointer содержит конструкцию, не поддерживающую какое-либо правило оформления выражений XPointer.
- *Ошибка адресации* (resource error) возникает, если синтаксически правильное выражение XPointer указывает на несуществующий XML-документ. То

есть оборванные ссылки считаются ошибками, распознаваемыми и обрабатываемыми на уровне XML-процессора, на уровне самого engine.

- *Ошибка адресации субресурса* (subresource error) возникает, если синтаксически правильное выражение XPointer указывает на существующий XML-документ, но конкретный субресурс, на который и должен осуществляться переход по ссылке, отсутствует.

Абсолютные указатели

Существует несколько ключевых слов, которые позволяют в XPointer адресовать субресурсы в абсолютном, а не в относительном порядке. Рассмотрим их.

- Ключевое слово `root` указывает на точку местонахождения открывающего тэга самого главного элемента, который объявлен как тип документа (`DOCTYPE`).
- Ключевое слово `here` является указателем (и, строго говоря, функцией) на текущее местоположение самого указателя. То есть отсчет будет вестись от самой ссылки. Естественно, адресовать подобным образом можно только те субресурсы, которые находятся в том же XML-документе, что и сама ссылка.
- Конструкция `ID(name)` позволяет адресовать элемент в XML-документе, у которого значение атрибута `id` установлено в `name`.
- Если мы используем два идентификатора, построенные при помощи средств XPointer, например, для адресации некоего интервала, то мы можем для второго идентификатора использовать ключевое слово `ditto`, указывающее на равенство начальных точек адресации для обоих идентификаторов.

Эти четыре ключевых слова задают первую точку адресации в XML-документе. Отталкиваясь от них, мы можем выстраивать цепочку уточняющих инструкций при помощи относительных указателей. Именно эти относительные указатели мы и рассмотрим в следующем разделе.

Относительные указатели

При помощи ключевых слов, которые мы рассмотрим в этом разделе, мы сможем производить адресацию по иерархии элементов, из которых и состоит содержание XML-документа.

Так как при движении по иерархической структуре нам хотелось бы иметь полную свободу адресации, то и использование относительных указателей немного отличается от применения абсолютных указателей. Относительные ука-

затели после ключевого слова, указывающего направление перемещения, в круглых скобках содержат список значений параметров, которые позволяют более точно найти необходимый нам субресурс. Примеры подобных указателей мы сможем увидеть в *разд. "Относительная адресация"*. Там будет легко понять их структуру и механизм применения. В этом разделе мы лишь рассматриваем ключевые слова, позволяющие создавать подобные указатели.

- Ключевое слово `child` позволяет адресовать все дочерние элементы источника, на который мы указали при помощи абсолютной адресации. При этом адресуются не все потомки исходного элемента, а только те, которые находятся на один уровень ниже, то есть — дети.
- Для адресации всех потомков начального элемента мы можем использовать ключевое слово `descendant`. При перемещении относительно ключевого элемента мы уже не учитываем иерархию потомков, а просто обходим содержание и используем только порядковый номер.
- Если же нам необходимо пройти по иерархии объектов вверх от ключевого элемента, к его предкам, мы должны использовать ключевое слово `ancestor`.
- А если нас не интересует иерархическая зависимость элементов друг от друга, и мы хотим просто пройти по содержимому XML-документа и отсчитать то или иное количество экземпляров элементов вниз от исходного, то надо использовать ключевое слово `following`.
- Для смещения вверх по содержимому XML-документа от начального элемента поиска мы можем применить ключевое слово `preceding`.
- Ключевое слово `psibling` задействуется для указания на "предшествующих братьев", то есть на элементы, у которых родитель совпадает с родителем исходного элемента, и которые в содержимом XML-документа находятся перед исходным элементом.
- Ключевое слово `fsibling` позволяет адресовать "последующих братьев". То есть таких, которые находятся в содержимом XML-документа после исходного адресуемого элемента.
- Кроме того, мы всегда можем использовать ключевое слово `ALL`, позволяющее адресовать все без исключения экземпляры элементов, входящих в содержимое XML-документа. Само собой, при использовании этого ключевого слова мы добавляем ограничения в скобках, как операнд функции, но, так или иначе, использование этого ключевого слова в указателе в качестве результата возвращает интервал.

Теперь разберем способы задания дополнительных условий поиска для относительных способов адресации. Для этого мы после каждого ключевого слова, определяющего тип выбора, в круглых скобках задаем уточняющие параметры.

Первый из них — порядковый номер субресурса. Если указанное число является положительным, то отсчет ведется по направлению к концу документа. Если задается отрицательное число, направление отсчета — противоположное.

Второй дополнительный уточняющий параметр позволяет задавать тип элемента, который мы ищем. Это может быть наименование элемента. Или же значение `CDATA`, указывающее на то, что адресоваться будут элементы, содержащие обычный текст. Символ звездочки "`*`" определяет, что мы ищем любой субресурс. Символ точки "." ограничивает поиск только элементами.

Затем в тех же скобках мы можем указать дополнительные условия поиска по атрибуту элемента. В третьей позиции мы можем записать наименование атрибута, по которому производится поиск, а в четвертой — значение этого атрибута.

Без сомнения, все, о чем мы узнали на протяжении этой главы, пока что выглядит очень голословно, но без примеров мы не обойдемся. Все примеры будут рассмотрены в следующих частях. Оставайтесь с нами.

Абсолютная адресация

В этом разделе мы рассмотрим примеры абсолютной адресации субресурсов средствами XPointer.

Для начала обратимся к примеру некоего обобщенного до требуемого нам уровня XML-документа:

```
<!DOCTYPE body[
<!ELEMENT p (#PCDATA)>
<!ELEMENT group (item*)>
]>
. . . . . .
<body>
<p>First paragraph</p>
<p>Second paragraph</p>
</body>
```

Будем считать, что URL этого XML-файла имеет вид `http://www.site.com/xml/f1.xml`. Если мы в каком-либо ином файле собираемся установить ссылку на этот файл, то мы можем использовать следующую конструкцию:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#root()">link</hlink>
```

В этой строке мы в открывающем тэге ссылки указываем в качестве значения атрибута `xlink:href` не URL XML-документа, а URI некоего субресур-

са. Нетрудно заметить, что после URL подключаемого XML-документа мы должны ставить в качестве разделителя знак решетки. И только после него уже записывать выражение XPointer. В данном случае мы использовали компонент `root()`, следовательно, будет загружен документ из файла `f1.xml`, а фокус будет передан на открывающий тэг `<body>`, так как именно элемент `<body>` является корневым для нашего документа.

Как мы знаем, в спецификации XML описан предопределенный "токенизированный" атрибут `id`. Этот атрибут позволяет задавать уникальные идентификаторы для каждого экземпляра любого элемента. Естественно, при помощи конструкций XPointer мы можем использовать эти идентификаторы в целях адресации. Рассмотрим соответствующий пример. Предположим, у нас есть XML-документ приблизительно следующего вида:

```
<!DOCTYPE body[
<!ELEMENT p (#PCDATA)>
<!ELEMENT group (item*)>
<!ATTLIST p
       id   ID   #REQUIRED>
]>
. . . . . .
<body>
<p id='p1'>First paragraph</p>
<p id='p2'>Second paragraph</p>
</body>
```

Теперь определим ссылку на этот документ. Если нам необходимо установить гиперссылку на первый параграф с содержимым `"First paragraph"`, мы должны будем использовать приблизительно следующую конструкцию:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#ID(p1)">
       ↳link on first paragraph</hlink>
```

Из списка ключевых слов абсолютной адресации мы уже знакомы с ключевым словом `here`. В частном случае оно позволяет элементу-ссылке адресовать самого себя. То есть, если в содержимом XML-документа, который находится, скажем, в файле все с тем же именем f1.xml, мы определим экземпляр элемента-ссылки вида

```
<hlink xlink:href= "http://www.site.com/xml/f1.xml#here()">
       ↳link</hlink>
```

то гиперссылка будет направлена на этот же экземпляр элемента.

Естественно, сам по себе компонент `here()` практически бесполезен. Он всегда используется в качестве основы для относительной адресации. Это — стандартное правило. Практически всегда элементы абсолютной адресации

являются базой, от которой ведут отсчет элементы относительной адресации. Они потому и называются относительными, что указывают на элементы относительно некоей заранее установленной точки.

Относительная адресация

После того, как мы установили точку отсчета при помощи элементов абсолютной адресации, необходимо произвести более точное целеуказание при помощи средств относительной адресации. Их мы рассматривали в *разд. "Относительные указатели"*. В этом разделе мы увидим примеры, разъясняющие действие каждого ключевого слова, которые могут использоваться в качестве относительных указателей.

Как мы знаем, относительные указатели позволяют адресовать достаточно отдаленные элементы, находящиеся в самых различных иерархических отношениях друг с другом. Поэтому для примера нам потребуется создать достаточно сложный документ. Вот как он будет выглядеть:

```
<!DOCTYPE body [
<!ELEMENT list (item*)>
<!ELEMENT item (#PCDATA|part)>
<!ELEMENT part (#PCDATA)>
<!ATTLIST item
       id ID #REQUIRED>
<!ATTLIST part
       id ID #REQUIRED>
]>
. . . . . .
<body>
<list id='l1'>
<item id='i1'>
<part id='p1'>Part 1</part>
<part id='p2'>Part 2</part>
</item>
<item id='i2'>Second item</item>
</list>
</body>
```

Теперь, когда у нас есть XML-документ с некоей иерархической структурой, мы можем приступить к рассмотрению выражений XPointer, использующих относительные указатели.

Для начала попробуем адресовать второй экземпляр дочернего элемента корневого элемента `body`. Для этого сначала мы установим указатель на

корневой элемент при помощи абсолютного указателя root, а затем используем относительный указатель child. При этом, согласно правилам, указатели отделяются друг от друга символом точки. В итоге мы получим следующее выражение:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#root().child(2)">
        ↳link to second item</hlink>
```

Как и требовалось, эта гиперссылка указывает на элемент item с идентификатором i2.

Теперь пришло время показать, как на один и тот же элемент можно ссылаться при помощи разных конструкций XPointer. В качестве цели ссылки мы выберем элемент part с идентификатором p2. Мы можем адресовать его как при помощи последовательных приближений относительных указателей child, так и при помощи одного-единственного относительного указателя descedant. Вариант с последовательными приближениями выглядит так:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#root().
        ↳child(1).child(1).child(2)">link to part 2</hlink>
```

Во втором случае мы применим более простой способ адресации посредством относительного указателя descedant, который позволяет адресовать любого наследника. Ссылка приобретет следующий вид:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#root().
        ↳descedant(4)">link to part 2</hlink>
```

Но есть и еще один вариант. Мы сначала установим абсолютный указатель на элемент item с идентификатором i1, а уже от него двинемся к искомому элементу part. Для этого мы используем конструкцию вида:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#ID(i1).
        ↳child(2)">link to part 2</hlink>
```

Мы рассмотрели относительные указатели, которые смещают точку адресации вниз по иерархии элементов с соблюдением правил наследования. Далее мы объясним, как можно смещаться по этой иерархии вверх, против правил наследования. Для этого применяется ключевое слово ancestor.

В следующем примере мы будем двигаться от самого глубокого элемента part к элементу body. Конечно, для того чтобы установить точку адресации на элемент body, мы могли бы напрямую использовать абсолютный указатель root(), но сейчас у нас несколько иная цель. Нам всего лишь надо продемонстрировать правила применения выражений XPointer. Итак, в данном случае для установки гиперссылки на элемент body относительно элемента part мы должны использовать приблизительно следующую конструкцию:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#ID(p2).
        ↳ancestor(3)">link to root</hlink>
```

В нашем примере XML-документа есть элемент `part`, который находится в самом низу иерархии элементов. Экземпляры этого элемента являются "братьями" друг для друга, так как у них есть общий родитель — экземпляр элемента `item` с идентификатором `i1`. Ниже рассмотрим примеры адресации подобных "братских" элементов.

Поскольку нам предоставлены два ключевых слова, позволяющие указывать на предшествующего и на последующего "брата", то и примеров будет два. В качестве базовой точки отсчета мы будем использовать поочередно оба экземпляра элемента `part`.

Для перехода от экземпляра документа с идентификатором `p1` к экземпляру с идентификатором `p2` мы можем воспользоваться выражением вида:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#ID(p1).
       ↳fsibling(1)">link to part 2</hlink>
```

Адресация от первой части ко второй может быть выполнена при помощи следующей конструкции:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#ID(p2).
       ↳psibling(1)">link to part 1</hlink>
```

Если же у нас возникает необходимость произвести относительную адресацию вне зависимости от иерархических отношений элементов, мы можем использовать ключевые слова `following` и `preceding`.

Например, для установки гиперссылки на элемент с идентификатором `i2` мы можем воспользоваться приблизительно следующей конструкцией:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#root().
       ↳following(4)">link to part 2</hlink>
```

Мы можем также установить абсолютный указатель на элемент, находящийся достаточно близко к концу документа, и от него уже двинуться вверх по содержимому по направлению к корневому элементу. А точка назначения гиперссылки останется такой же, как и в предыдущем примере. Это будет выглядеть следующим образом:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#ID(i1).
       ↳preceding(1)">link to part 2</hlink>
```

Итак, мы рассмотрели примеры относительной адресации. Мы видели, что в выражениях XPointer сначала записываются абсолютные указатели, а уже на их основе действуют относительные указатели. Но ведь нас никто не ограничивает в количестве последовательных приближений. После установки одного относительного указателя мы можем добавить еще один. Серия последовательных приближений позволяет создавать выражения, максимально приближенные к семантической структуре сколь угодно разветвленных документов.

Адресация интервалов

Рассмотренные нами примеры позволяли определять гиперссылку на тот или иной семантический субресурс XML-документа. Этот субресурс, по сути, являлся лишь единичной точкой входа в общий ресурс. Но мы можем адресовать не только единичные элементы, но и целые фрагменты XML-документов, поскольку фрагменты тоже являются субресурсами.

Ссылка на некий фрагмент производится при помощи двух выражений XPointer. При этом в качестве результата действия гиперссылки возвращается блок XML-документа, находящийся между двумя точками, на которые и указывают выражения XPointer. Разумеется, первое выражение должно адресовать элемент, находящийся ближе к началу документа, нежели элемент, адресуемый вторым выражением. В противном случае XML-процессор выдаст сообщение об ошибке.

В спецификации XPointer субресурс интервального типа называется `range`. Создание ссылки на подобный ресурс осуществляется при помощи ключевого слова `range-to`.

Самым простым примером установки гиперссылки на некий интервал можно считать следующее выражение:

```
<hlink xlink:href="http://www.site.com/xml/f1.xml#id("chap1")/range to
      ↳(id("chap2"))">link to chapter 1</hlink>
```

Эта гиперссылка ориентирована на фрагмент XML-документа, который находится между элементами с идентификаторами `chap1` и `chap2`. Как видно из приведенного примера, `range-to` является обычной функцией, которой в качестве параметра передается конечная точка интервала. При этом мы можем в качестве параметра функции передавать не только абсолютные указатели, как в нашем примере, но и любые выражения XPointer, в том числе и "многоэтажные" конструкции с использованием любых относительных указателей и строчных функций поиска, которые мы рассмотрим в следующей части.

Существует еще несколько "продвинутых" функций, которые позволяют оперировать в выражениях XPointer с интервалами. Рассмотрим их, не опираясь на примеры.

Функция `range`, как и ее предшественник `range-to`, в качестве результата выдает множество точек входа, или, как мы договорились их именовать, точек адресации. В официальной спецификации XPointer подобные наборы называют `location-set`. Но функция `range`, в отличие от `range-to`, в качестве параметра принимает не одну точку входа, а их набор. То есть декларация этой функции будет выглядеть следующим образом:

```
location-set range(location-set )
```

Данная функция применяется для создания указателя на фрагмент, который будет объединять все точки адресации, указанные в наборе, переданном в функцию в качестве параметра. То есть функция `range` является более общим вариантом функции `range-to`.

Во всех функциях для задания входных параметров мы можем использовать дополнительные функции, которые помогают объявлять стартовую и конечную точку интервала.

Функция `start-point` принудительно задает начальную точку интервала. Параметр и результат имеют тип `point`. Для принудительной установки конечной точки интервала используется функция `end-point`.

Необходимо осознавать, что обе эти функции применяются для управления неким множеством точек адресации, то есть, в конечном счете, действия должны выполняться над неким операндом с типом `location-set`. При этом если тип аргумента не совпадает с `location-set`, результат применения функции не всегда бывает таким, каким он задумывался разработчиком.

Если операнд фактически является единичной точкой входа (`point`), то его значение замещается на значение параметра, переданного функции `start-point` или `end-point`, в зависимости от того, какая функция была применена к операнду.

Если аргумент является атрибутом или неким идентификатором (например, псевдонимом сущности), то XML-процессор сообщит о возникновении ошибки.

Адресация строчных субресурсов

При помощи функций, которые мы рассмотрим в этой части, мы сможем осуществлять адресацию, ориентируясь на строковые элементы. То есть, по сути, это механизм текстового поиска в документе, встроенный непосредственно в язык XPointer.

Строчные субресурсы являются частным случаем адресуемых интервалов, которые мы рассматривали в предыдущей части. Их адресация происходит при помощи функции `string-range`. Рассмотрим несколько примеров применения этой функции.

Для начала мы разберем одну из самых простых конструкций:

```
string-range(//title,"Thomas Pynchon")[17]
```

Данное выражение означает, что XPointer будет разыскивать в тексте XML-документа семнадцатое вхождение элемента типа `title`, значение которого равно `Thomas Pynchon`.

На примере этого выражения XPointer мы можем понять, какие параметры используются в функции `string-range`.

Первый параметр имеет тип `location-set`. То есть в качестве первого параметра функции `string-range` мы можем передать как наименование элемента, по экземплярам которого будет проводиться поиск, так и результат действия других конструкций XPointer, которые в качестве результата возвращают интервал или множество точек адресации. В нашем случае мы указали условие, что поиск должен вестись по экземплярам элемента `title`.

Второй параметр имеет тип `string`. Это, собственно, и есть основной параметр поиска и, соответственно, включения элемента в результирующий набор точек адресации или в результирующий интервал.

После определенного нами списка парамеров операции поиска в квадратных скобках мы указали, какой по счету элемент войдет в результирующую выборку и станет точкой адресации гиперссылки. Мы можем не использовать последнюю возможность, и тогда будут возвращены все элементы, удовлетворяющие условию поиска.

Рассмотренные нами параметры не являются единственно возможными для функции `string-range`. Мы можем добавить в список еще два параметра. Приведем соответствующий пример:

```
string-range(//title,"Thomas Pynchon",8,4)
```

Теперь функция при успехе поиска возвратит точку адресации подстроки `"Pync"`, которая начинается с восьмого символа строки поиска, и содержит четыре символа. Причем, в данном случае мы можем получить в качестве результата не одну подстроку, а несколько, если в содержимом документа найдется не один элемент типа `title` со значением `Thomas Pynchon`.

Как мы помним, первый параметр нашей функции имеет тип `location-set`, поэтому мы можем создавать вложенные конструкции, как в следующем примере:

```
string-range(string-range(//P,"Thomas Pynchon")[17],"P",1,0)
```

Здесь основной функции `string-range` в качестве базы поиска передаются результаты первичного поиска, то есть семнадцатый экземпляр элемента `P`, который содержит строку `Thomas Pynchon`. Уточняющий поиск выполняется на основе первичного. Нам требуется найти символ `P`, от которого мы возьмем подстроку, начинающуюся с этого символа и не содержащую ни одного символа. В данной ситуации мы получаем на выходе вырожденный интервал, не содержащий никакого текстового наполнения. В спецификации подобный случай называется *collapsed range*.

Адресация элементов

Идеология интеллектуальных гиперссылок в XML ориентирует составителей XML-документов на создание семантических, а не навигационных ссылок. При помощи относительных указателей мы можем создавать ссылки не на конкретный URL, который может измениться, а напрямую, на искомый элемент, используя иерархические отношения элементов содержимого XML-документа. В этой части мы рассмотрим все возможные способы адресации отдельных экземпляров элементов содержимого XML-документа.

Для начала мы рассмотрим способ адресации элемента по значению атрибута. Мы можем задать выражение XPointer, которое адресовало бы только тот экземпляр элемента, у которого значение некоего атрибута совпало бы с предустановленным значением. Для этого мы можем использовать следующую конструкцию:

```
root().following(2,paragraph,shadow,"none")
```

Таким выражением мы адресуем второй элемент типа `paragraph`, у которого значение параметра `shadow` установлено в `none`.

Как видно из примера, для адресации элементов по значению атрибута используются уже знакомые нам относительные указатели. До сих пор мы использовали всего один параметр для относительных указателей, тот, который указывал их порядковый номер. Необходимо напомнить, что этот номер не обязан быть положительным. Если мы укажем отрицательный индекс, то отсчет пойдет к началу содержимого XML-документа.

В качестве второго параметра мы передаем наименование типа элемента, на который мы хотим установить точку адресации. Этот параметр является необязательным, равно как и третий с четвертым параметры.

Третий параметр содержит наименование атрибута, по которому осуществляется уточняющий поиск. Само собой, одного имени атрибута для уточняющего поиска недостаточно, поэтому в четвертом параметре передается значение данного атрибута, по которому и будет производиться поиск.

В *разд. "Абсолютные указатели"* мы рассматривали функцию `here()`, которая применяется для абсолютного позиционирования. Следующий фрагмент еще раз демонстрирует ее действие на примере адресации элементов:

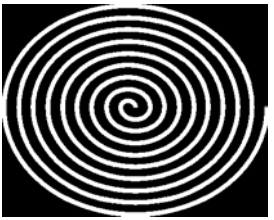
```
<hlink href="http://www.site.com/xml/x1.xml#here().
       ↳following(3)">link to second item</hlink>
<list>
<item id="i1">First item</item>
<item id="i2">Second item</item>
</list>
```

В данном примере мы определяем гиперссылку на элемент с идентификатором `i2`. После применения функции `here()` гиперссылка указывает на саму себя, а затем при помощи условного указателя смещает точку адресации на три элемента вниз, попадая прямо на второй элемент списка.

Мы не упомянули еще одну полезную функцию XPointer. Функция `origin()` указывает на точку, из которой произошла ссылка на данный субресурс. Разумеется, эта функция предназначена для XML-приложений, так как HTML-браузеры пользуются параметром ссылки `from`.

На этом мы заканчиваем рассмотрение стандарта XML и переходим к обзору наиболее известных его приложений.

Глава 4



Схемы XML-документов

Причина появления

Видимо, у всех компьютерных технологий есть одно общее правило: причины поражения кроются в причинах успеха. То, что давало импульс к развитию, потом будет сдерживать технологию. То, ради чего технология создавалась, потом будет служить ее ограничением.

То же самое случилось и с XML. Его максимальная свобода и расширяемость сослужили плохую службу в тех областях, где требовалась определенная унификация документов. Каждый старался сделать свой набор элементов, не обращая внимания на коллег и конкурентов. Естественно, это создавало определенные проблемы.

Обратив внимание общественности на эту проблему, Microsoft (по крайней мере, так заявляют представители этой компании) предложила создать так называемую схему XML — XML Schema. Предложенная компанией технология позволяет вообще обходиться без DTD-блоков.

Основой ее является использование так называемых пространств имен и очень точная типизация содержимого элементов документов. Пространства имен, которые мы будем тщательно рассматривать в этой главе несколько позже, позволяют задавать имена элементов без использования DTD-блоков, которые являлись до этого времени основой технологии XML. В технологии XML Schema мы можем применять имена элементов без их предварительного объявления в DTD-блоке. Вместо этого мы будем использовать специализированный файл с определением применяемой схемы. На первый взгляд такая замена не выглядит особым преимуществом, так как все равно нам приходится четко описывать элементы, входящие в состав документа. Однако необходимо учесть тот факт, что в документе данные могут быть представлены под одними и теми же именами, но в разных контекстах. Концепция согласования пространств имен позволяет четко различать имена элементов в зависимости от контекста. Это достижение уже можно считать большим прогрессом.

Вторым ценным качеством технологии XML Schema является расширенная система типизации данных. Как мы помним, одним из наиболее интересных свойств XML-документов является возможность не только их отображения, но и обработки специализированными программами. Для чего, конечно, необходимо четко типизировать данные, содержащиеся в XML-документе.

Мы уже говорили немного ранее, что типизация данных в XML не в пример выше по сравнению с HTML, тем более, что в HTML эта типизация отсутствует в принципе. Однако элементы в XML могут состоять лишь из текстовых строк и вложенных субэлементов. То есть, по сути, вся информация может представляться лишь в текстовом виде.

Для обработки XML-документов специализированными XML-приложениями можно определить для элемента дополнительный атрибут, который будет указывать тип значения этого элемента, а затем приложение-обработчик будет анализировать этот атрибут и соответствующим образом конвертировать значение элемента из строкового типа в требуемый формат. Но данное решение является, как минимум, неизящным.

Поэтому в спецификации XML Schema были введены возможности строгой типизации содержимого элементов. Более того, мы получили возможность создавать сложные типы данных и наследовать их. В XML Schema мы можем устанавливать даже шаблоны для строковых данных и ограничения, накладываемые на структуру данных. В рамках этой технологии нам предоставляются достаточно широкие возможности типизации данных, что должно облегчить их дальнейшую обработку специализированными приложениями.

Первый пример

Для того чтобы иметь возможность детально обсуждать решения, применяемые в XML Schema, необходимо сначала рассмотреть один простой пример. Тогда, руководствуясь им, мы можем перейти к детальному рассмотрению применяемых концепций. Итак, чуть ниже, а именно в листинге 4.1, представлен пример простого документа, созданного по технологии XML Schema. Пример взят с официального сайта Microsoft. Конечно, было бы более правильным использовать образец, приведенный World Wide Web Consortium, но в качестве вводного примера там выбран достаточно сложный и объемный документ.

Листинг 4.1. XML-документ

```
<?xml version="1.0" ?>
<PGROUP>
  <PERSONA>МАКБЕТ</PERSONA>
  <PERSONA>БАНКО</PERSONA>
  <GRPDESCR>Высший командный состав королевской армии</GRPDESCR>
</PGROUP>
```

Как видно, в приведенном XML-документе присутствуют всего три элемента. Описать структуру этого документа можно при помощи DTD-блока, код которого показан в листинге 4.2.

Листинг 4.2. DTD-блок

```
<!DOCTYPE PGROUP [
<!ELEMENT PGROUP          (PERSONA+, GRPDESCR) >
<!ELEMENT PERSONA         (#PCDATA)  >
<!ELEMENT GRPDESCR        (#PCDATA)  >
]>
```

Однако нас интересует описание структуры данного документа в терминах XML Schema. Это описание формируется в виде обособленного файла, который, в свою очередь, является XML-документом. В этом заключается основное отличие файлов XML Schema от DTD-блоков. Как мы знаем, DTD-блоки имеют свой собственный синтаксис, отличный от синтаксиса XML.

Итак, в листинге 4.3 приводится код документа-схемы, определяющего структуру представленного выше XML-документа.

Листинг 4.3. Схема XML-документа

```
<?xml version="1.0"?>
<Schema name="schema_sample_1"
    xmlns="urn:schemas-microsoft-com:xml-data"
    xmlns:dt="urn:schemas-microsoft-com:datatypes">
  <ElementType name="PERSONA" content="textOnly" model="closed"/>
  <ElementType name="GRPDESCR" content="textOnly" model="closed"/>
  <ElementType name="PGROUP" content="textOnly" model="closed">
       <element type="PERSONA" minOccurs="1" maxOccurs="*"/>
       <element type="GRPDESCR" minOccurs="1" maxOccurs="1"/>
  </ElementType>
</Schema>
```

Невооруженным глазом видно, что файл схемы документа объемнее, чем DTD-блок для того же документа. Однако этот дополнительный объем позволяет получать и дополнительные возможности.

Рассмотрим структуру приведенного файла. Как мы уже говорили, файлы-схемы являются полноправными XML-документами, поэтому в первой строке мы наблюдаем исполняемую инструкцию для XML-процессора, указывающую номер версии применяемого стандарта XML.

Затем следует открывающий тэг основного элемента `Schema`. При этом мы явно указываем значение атрибута `name`, объявляющего наименование схемы.

После чего мы определяем префиксы используемых элементов, используя для этой цели так называемые "пространства имен". То есть мы устанавли-

ваем правила, по которым обрабатывающее приложение будет распознавать элементы и типы данных.

Для правильной обработки элементов документа сначала необходимо указать тип интересующих нас элементов. Эта операция производится при помощи тэгов `<ElementType>` с соответствующими параметрами. При помощи параметров мы указываем наименование создаваемого элемента, тип его содержимого и модель представления.

После того, как мы определили все применяемые элементы, необходимо уточнить возможное количество их вхождений в тело родительских элементов. Из листинга легко понять, что элементы `PERSONA` и `GRPDESCR` входят в состав элемента `PGROUP`. При этом элемент `PERSONA` может использоваться несколько раз, но не менее одного раза. Элемент `GRPDESCR` мы можем задействовать только один раз.

По окончании определения всех элементов мы ставим закрывающий тэг `</Schema>`.

Уже на этом примере мы видим, в общих чертах, механизм действия технологии XML Schema. Однако мы сознательно рассмотрели очень простой пример. На самом деле все немного сложнее. Да и возможностей, соответственно, больше. Эта глава будет полностью отведена под их обзор.

Структура схемы

Когда мы объявляем схему документа при помощи тэгов `<Schema>`, необходимо придерживаться правильного порядка декларирования всех элементов. В спецификации этой технологии содержимое компонента `Schema` определяется следующим образом:

```
{type definitions}
{attribute declarations}
{element declarations}
{attribute group definitions}
{model group definitions}
{notation declarations}
{annotations}
```

То есть сначала мы определяем применяемые типы данных, как простые, так и комплексные, составные. Затем следует объявление глобальных атрибутов, которые относятся ко всем элементам сразу. После этого следует самая важная, пожалуй, секция — объявление элементов, входящих в состав XML-документа. Следом за ними мы объявляем используемые группы атрибутов. Затем мы описываем дополнительные элементы схемы, такие как шаблоны, накладываемые на значения элементов, или же модели представления. И в самом конце размещаются так называемые аннотации.

Строго говоря, в качестве отдельных компонентов схемы применяются двенадцать элементов, разбиваемых обычно на три группы. В первую группу входят определения простых типов, определения составных типов, объявления атрибутов и объявления элементов. Ко второй группе относятся определения групп атрибутов, определения уникальности элементов, определения моделей представления и так называемых *нотаций*, то есть элементов, расшифровывающих предназначение того или иного элемента. В третью и последнюю группу входят аннотации, группы моделей представления информации, дополнительные данные о моделях и объявления шаблонов, накладываемых на содержимое элементов.

Вот из этих частей и складывается полное описание *схемы документа*. Однако, скорее всего, нам не придется использовать все перечисленные компоненты. Обычно объявляются лишь применяемые типы и элементы с атрибутами. Этого усеченного набора компонентов вполне достаточно для создания полноценных схем документов. Более того, совсем необязательно объявлять все необходимые компоненты самостоятельно. О механизме заимствования уже существующих компонентов мы узнаем в следующем разделе этой главы.

Пространства имен

В XML мы могли использовать определения элементов документов, которые сделали другие люди. Для этих целей мы импортировали уже существующие DTD-блоки. Нечто похожее есть и в технологии XML Schema. Здесь совокупности элементов и типов данных могут быть собраны в единые блоки, получившие наименование *пространств имен*. Эти блоки распространяются разработчиками так же свободно, как и DTD-описания.

Для того чтобы отличать импортируемые элементы и конструкции, перед ними ставится соответствующий префикс. Немного позже мы будем рассматривать вопросы объявления элементов и их атрибутов, и в одном из примеров будет предложен следующий фрагмент кода:

```
<xs:element name="picture">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="xs:binary">
    <xs:attribute name="pictype" type="xs:NOTATION"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>
```

Несложно заметить, что перед каждой конструкцией находится префикс, включающий группу символов `xs` и двоеточие. Это и есть ссылка на внешнее пространство имен.

Разумеется, перед тем, как применять некое пространство имен, необходимо связать его с нашим документом.

Для этого в стартовый тэг схемы документа мы должны включить конструкцию следующего вида:

```
xmlns:fo="http://www.w3.org/1999/XSL/Format"
```

Здесь мы объявляем, что все элементы и конструкции с префиксом `fo` находятся в пространстве имен, которое расположено по адресу `http://www.w3.org/1999/XSL/Format`.

Элементы и атрибуты

Описание структуры XML-документа сводится по большей части к описанию элементов и их атрибутов. При этом элементы, как мы уже знаем, практически всегда образуют некую иерархию вхождений друг в друга. При описании схем XML-документов начинать всегда следует с определения простейших, атомарных элементов, которые не содержат дочерних субэлементов. То есть перед описанием некоего родительского элемента мы сначала должны определить все его дочерние субэлементы. Данная концепция естественным образом заимствована из высокоструктурированных языков программирования, в которых нельзя использовать переменные, не объявив их предварительно.

В схеме сначала необходимо определить тип элемента при помощи тэга `<ElementType>` с обязательным параметром `name`. В качестве строкового значения этого параметра используется наименование определяемого элемента. Продемонстрировать применение этого тэга можно на следующем примере:

```
<ElementType name="book" model="closed">
    <element type="title" />
    <element type="author" />
    <element type="pages" />
    <AttributeType name="copyright" />
    <attribute type="copyright" />
</ElementType>
```

В данном случае мы объявляем тип элемента с именем `book`, в который входят субэлементы типов `title`, `author` и `pages`, а также атрибут с наименованием `copyright`. Безусловно, типы субэлементов и атрибутов, используемых в данном определении элемента, должны быть объявлены заранее.

После определения типа мы можем объявить сам элемент при помощи тэга `<element>` с обязательным параметром `type`, определяющим тип элемента. Желательно также использовать параметр `name`, в котором указывается наименование элемента. В качестве примера использования данного тэга можно привести следующее объявление элемента:

```
<element name="myelement" type="mySimpleType"/>
```

В данном случае вся информация находится внутри одного тэга, поэтому он и объявлен закрытым. Однако для элементов не простого типа, а комплексного мы можем использовать объявление типа прямо внутри объявления элемента. Выглядеть это будет приблизительно следующим образом:

```
<element name="el1">
 <complexType>
  <attribute ...>Набор атрибутов комплексного типа</attribute>
 <complexType>
</element>
```

На самом деле все рассмотренные нами конструкции являются облегченными. Объявление элемента в полном объеме является достаточно сложной конструкцией. В подавляющем большинстве случаев все предусмотренные возможности не используются. Но знать, как специфицируется объявление элемента, мы должны. Согласно спецификациям XML образцовое объявление выглядит следующим образом:

```
<element
  abstract = boolean : false
  block = (#all | List of (substitution | extension | restriction))
  default = string
  final = (#all | List of (extension | restriction))
  fixed = string
  form = (qualified | unqualified)
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  nullable = boolean : false
  ref = QName
  substitutionGroup = QName
  type = QName >
  Content: (annotation? , ((simpleType | complexType)? , (key |
  ↳keyref | unique)*))
</element>
```

Каждый элемент может иметь несколько атрибутов. В рамках XML Schema определение атрибутов несколько отличается от стандартного, и выглядит, как показано ниже:

```
<attribute
  form = (qualified | unqualified)
  id = ID
  name = NCName
  ref = QName
  type = QName
  use = (prohibited | optional | required | default |
↳fixed) : optional
  value = string >
  Content: (annotation? , (simpleType?))
</attribute>
```

Как видно, объявление атрибута весьма похоже на объявление элемента. Это и неудивительно. Функционально они весьма похожи на субэлементы, у которых нет вложений.

В качестве примера объявления атрибута можно привести следующую конструкцию:

```
<attribute name="MyAttribute" use="default" value="42">
 <simpleType>
  <restriction base="integer">
   <minInclusive value="0"/>
  </restriction>
 </simpleType>
</attribute>
```

В этом фрагменте кода мы объявляем атрибут, основанный на целочисленном типе. При этом мы устанавливаем для него значение по умолчанию, равное 42. А также задаем минимальное значение для этого атрибута. Нетрудно заметить, что объявление значения по умолчанию производится внутри открывающего тэга `<attribute>` при помощи его параметров, а объявление типа атрибута является содержимым этого тэга. Полный обзор применяемых типов и их свойств мы увидим несколько позже.

На самом деле в XML Schema атрибуты не обязаны состоять из одного значения атомарного типа. Мы можем создавать так называемые *комплексные типы*, которые могут объединять несколько атомарных типов. При использовании комплексных типов значение атрибута все равно останется единым целым, хотя и будет являться объединением разнородных данных.

Определение комплексных типов выглядит следующим образом:

```
<complexType
  abstract = boolean : false
  block = (#all | List of (extension | restriction))
  final = (#all | List of (extension | restriction))
  id = ID
  mixed = boolean : false
  name = NCName >
  Content: (annotation? , (simpleContent | complexContent | ((group
  ↳| all | choice | sequence)? , ((attribute | attributeGroup)* ,
  ↳anyAttribute?))))
</complexType>
```

В стартовом тэге `<complexType>` обычно определяется лишь идентифицирующее имя комплексного типа. А практически все определение комплексного типа производится в содержимом данного XML-элемента. Определение содержимого указывается в спецификации элемента после обозначения `Content`. Как видно, там перечислено достаточно много конструкций. Придется их все рассмотреть.

Конструкция с наименованием `simpleContent` позволяет объявлять комплексный тип на основе ограничений или расширений существующих типов. Объявляется она следующим образом:

```
<simpleContent
  id = ID >
  Content: (annotation? , (restriction | extension))
</simpleContent>
```

Ограничения задаются элементом `restriction`, а расширения — `extension`.

Элемент `restriction` позволяет устанавливать ограничения, накладываемые на значения создаваемого комплексного типа при помощи свойств и атрибутов. Его объявление имеет вид:

```
<restriction
  base = QName
  id = ID >
  Content: (annotation? , (simpleType? , (duration | encoding |
  ↳enumeration | length | maxExclusive | maxInclusive | maxLength |
  ↳minExclusive | minInclusive | minLength | pattern | period |
  ↳precision | scale | whiteSpace)*)? , ((attribute | attributeGroup)* ,
  ↳anyAttribute?))
</restriction>
```

Из определения видно, что в качестве содержимого данного тэга мы можем использовать любую комбинацию свойств и атрибутов, накладываемых на

базовый тип. Наименование применяемого базового типа записывается в качестве значения обязательного атрибута `base`.

Если свойства базовых типов органически приспособлены для наложения ограничений, то для расширения типов желательно подбирать наиболее подходящие атрибуты. Это расширение производится при помощи элемента `extension`. Определяется он следующим образом:

```
<extension
  base = QName
  id = ID >
  Content: (annotation? , ((attribute | attributeGroup)* ,
  ↳anyAttribute?))
</extension>
```

Определение этого элемента достаточно прозрачно для понимания. Мы обязательно указываем наименование базового типа в атрибуте `base`, а затем расширяем его при помощи одного или нескольких атрибутов.

Итак, мы рассмотрели определение конструкции `simpleContent`. На очереди обзор конструкции `complexContent`. Она определяется следующим блоком кода:

```
<complexContent
  id = ID
  mixed = boolean >
  Content: (annotation? , (restriction | extension))
</complexContent>
```

На первый взгляд разница невелика. На самом деле основное отличие кроется в способах задания ограничений и расширений. Для элемента `complexContent` вложенные субэлементы `restriction` и `extension` задаются иным образом, нежели для элемента `simpleContent`.

Здесь элемент `restriction` объявляется так:

```
<restriction
  base = QName
  id = ID >
  Content: (annotation? , (group | all | choice | sequence)? ,
  ↳((attribute | attributeGroup)* , anyAttribute?))
</restriction>
```

То есть мы можем задавать группу атрибутов, позволять делать выбор из группы или же задавать последовательность использования атрибутов. Мы получаем новые возможности управления содержимым ограничения.

Точно так же дополнено и определение расширения. Для составного типа оно выглядит так:

```
<extension
  base = QName
  id = ID >
  Content: (annotation? , ((group | all | choice | sequence)? ,
  ↳((attribute | attributeGroup)* , anyAttribute?)))
</extension>
```

В качестве примера создания подобного усложненного комплексного типа можно привести следующее объявление:

```
<complexType name="Itemlength">
 <complexContent>
  <restriction base="anyType">
   <sequence>
    <element name="size" type="nonNegativeInteger"/>
    <element name="unit" type="NMTOKEN"/>
   </sequence>
  </restriction>
 </complexContent>
</complexType>
<element name="depth" type="Itemlength"/>
```

В этом фрагменте кода мы объявляем комплексный тип `Itemlength`. Значение этого типа состоит из последовательности двух элементов, один из которых является неотрицательным целым числом, а второй — предустановленным ключевым словом. Но при этом содержимое объявленного комплексного типа не перестает оставаться единым целым. А затем мы объявляем элемент `depth`, которому присваиваем только что объявленный тип.

Применить созданный элемент в документе можно приблизительно следующим образом:

```
<depth>
   <size>3</size><unit>cm</unit>
  </depth>
```

Итак, теперь мы знаем как правильно объявлять комплексные типы. Существует еще одна возможность объявлять свои типы, но ее мы рассмотрим в следующем разделе этой главы.

Когда мы объявляем серию однотипных элементов, незначительно отличающихся друг от друга, всегда может возникнуть ситуация, когда эти схожие элементы имеют некоторое множество идентичных атрибутов. Записывать их каждый раз в каждом объявлении элемента представляется непродуктивным. Необходимо изыскать способ учитывать набор этих совпадающих атрибутов как единое целое. В XML для подобных целей использова-

лись параметрические сущности. А в XML Schema было введено понятие *группы атрибутов*. Описывается такая группа следующим образом:

```
<attributeGroup
  id = ID
  name = NCName
  ref = QName >
  Content: (annotation? , ((attribute | attributeGroup)* ,
  ↳anyAttribute?))
</attributeGroup>
```

То есть между открывающим и закрывающим тэгом `<attributeGroup>` мы помещаем атрибуты, входящие в состав группы, или вложенные группы атрибутов. Естественно, для того, чтобы идентифицировать создаваемую группу, нам необходимо задать ее уникальное имя при помощи параметра `name` открывающего тэга `<attributeGroup>`.

Определение группы атрибутов достаточно прозрачно и не требует приведения дополнительного примера.

В этой части нам осталось рассмотреть лишь вопросы создания условных обозначений. Мы уже встречались с подобными элементами XML-документов при обсуждении стандарта XML. Как мы помним, условные обозначения являются специальными элементами, которые позволяют ассоциировать тип файла, присоединяемого к содержимому документа, и приложение, которое будет обрабатывать этот файл. В спецификации XML Schema объявление условного обозначения, разумеется, отличается от способа его объявления в XML, и выглядит следующим образом:

```
<notation
  id = ID
  name = NCName
  public = A public identifier, per ISO 8879
  system = uriReference>
  Content: (annotation?)
</notation>
```

У тэга `<notation>` есть два обязательных атрибута. Атрибут `name` содержит наименование условного обозначения для типа файла, которое применяется в данном документе, а атрибут `public` уточняет наименование этого типа согласно стандарту ISO 8879. В качестве значения атрибута `system` применяется URI, указывающий на расположение программы, предназначенной для отображения файлов объявленного типа. Данный атрибут является необязательным, так как предполагается, что процессор для обработки схем сам умеет обрабатывать и отображать наиболее распространенные типы присоединяемых файлов. Но слишком полагаться на его возможности все таки не стоит.

В качестве примера применения условного обозначения можно привести следующий фрагмент кода:

```
<xs:notation name="jpeg"
             public="image/jpeg" system="viewer.exe" />

<xs:element name="picture">
 <xs:complexType>
  <xs:complexContent>
   <xs:extension base="xs:binary">
    <xs:attribute name="pictype" type="xs:NOTATION"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
</xs:element>

<picture pictype="jpeg">...</picture>
```

Здесь мы просто задаем механизм отображения графических файлов формата JPEG.

На этом обзор элементов и их атрибутов заканчивается. Пришло время разобраться, какие типы данных предусмотрены в XML Schema, и как мы можем ими пользоваться.

Типы данных

Естественным образом, все сложные типы данных состоят из простых. В этой части мы перечислим все применяемые в XML Schema предопределенные типы данных. Необходимо отметить, что у каждого значения того или иного типа существует набор определенных свойств. Перечень свойств зависит, в свою очередь, от типа этого значения.

Первое из них, а именно свойство `length`, содержит меру длины какой-либо величины. Значение этого свойства может быть только целым числом больше нуля.

Свойство `minLength` может использоваться в виде ограничения, налагаемого на какую-либо величину. Значение этого свойства является минимальной длиной данной величины.

Свойство `maxLength` задает максимально возможную длину значения, к которому это свойство применимо.

В значении свойства `pattern` мы можем указывать шаблон, накладываемый на какую-либо величину. Если мы хотим, чтобы в том или ином строковом

значении некоего элемента могли находиться только цифры, то мы явно должны задать для этого элемента свойство pattern.

Свойство enumeration применяется для создания элементов перечислимого типа. В значении этого свойства указываются возможные варианты значения элемента, на который оно накладывается.

Значение свойства whiteSpace позволяет регулировать механизм обработки пробелов в содержимом элемента, на который это свойство накладывается. В качестве значения этого свойства могут использоваться только три предопределенных ключевых слова. Нам известно, что в XML (в отличие от HTML) вся информация, расположенная между открывающими и закрывающими тэгами, рассматривается как данные, то есть учитываются все символы форматирования. Что равносильно применению значения preserve, указывающего, что все символы форматирования будут оставаться на своих местах без каких-либо изменений. Ключевое слово replace определяет, что все символы форматирования, включая пробельные, символ табуляции (шестнадцатиричный код — 9), символы перевода строки (код — A) и возврата каретки (код — D) принудительно будут заменены символом обычного пробела (шестнадцатеричный код — 20). Значение collapse сворачивает последовательности пробелов в единичный пробел.

Свойства minExclusive и maxExclusive позволяют задавать границы промежутка, внутри которого находятся значения, которые не может принимать это свойство. То есть мы можем задавать исключаемые значения для содержимого какого-либо элемента. Само собой, значения, находящиеся вне заданного диапазона, могут быть использованы. Свойство minExclusive применяется при определении нижней границы исключаемого промежутка значений, а свойство maxExclusive — для верхней границы.

Помимо интервала исключаемых значений мы можем явно устанавливать промежуток, в котором могут находиться допустимые значения содержимого какого-либо элемента. Для этих целей мы используем свойства minInclusive и maxInclusive, соответственно для нижней и верхней границ интервала допустимых значений.

Свойство precision конкретизирует точность представления числовых величин. Значение этого свойства указывает максимальное количество десятичных знаков, используемых для отображения целой части рациональных чисел.

Для задания максимального количества десятичных знаков в представлении дробной части рационального числа применяется свойство scale.

Значение свойства encoding управляет порядком представления бинарных данных. В качестве значения этого свойства могут использоваться два ключевых слова. Значение hex отображает бинарные данные в шестнадцатеричном коде, то есть только цифрами и символами латинского алфавита от A

до F. Ключевое слово base64 указывает, что двоичные данные будут передаваться при помощи стандарта Base64 Content-Transfer-Encoding, описанного в документе RFC 2045.

Свойство duration связывает содержимое элемента с временным промежутком определенной длительности.

Свойство period характеризует периодичность какого-либо события.

Теперь, когда мы ознакомились с различными встроенными свойствами, мы можем перейти к рассмотрению самих типов значений, применяемых в спецификации XML Schema.

Тип string позволяет специфицировать символьные данные в виде *строк*. Строка определяется как конечная последовательность символов в одной из выбранных кодировок. К значениям этого типа могут применяться свойства length, minLength, maxLength, pattern, enumeration, whiteSpace. В обычных XML-документах мы обозначали данные этого типа при помощи ключевого слова CDATA.

Тип boolean предназначен для реализации логических значений. По сути, является перечислимым типом. В качестве допустимых значений могут использоваться только ключевые слова true и false, первое из которых обозначает истинность утверждения, второе — ложность. Так как технически этот тип является сильно усеченным строковым типом, то к данным этого типа могут применяться только свойства pattern и whiteSpace, хотя и они, по существу, функционально не нужны.

Тип float используется для представления числовых данных. Для хранения каждой числовой величины этого типа отводится тридцать два бита, то есть четыре байта. При этом значение представляется в виде произведения целого числа, чье абсолютное значение меньше или равно 16 777 216, и экспоненты, показатель степени которой может находиться в пределах от 104 до149. К данным этого типа могут применяться свойства pattern, enumeration, whiteSpace, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип double также предназначен для числовых данных. Но для каждого значения этого типа отводится уже шестьдесят четыре бита, то есть восемь байтов. Соответственно изменяется и точность представления чисел, и возможный диапазон представимых значений. Для данных этого типа мы можем использовать свойства pattern, enumeration, whiteSpace, minInclusive, maxInclusive, minExclusive и maxExclusive.

Если предыдущие два типа рассматривают число в экспоненциальной форме, то тип decimal для представления числовых данных использует десятичные степени. Естественно, в этом случае число представляется в виде обычной десятичной дроби. По умолчанию XML-процессоры распознают и обрабатывают числа с восемнадцатью десятичными знаками после запятой. Значения этого типа поддерживают свойства precision, scale, pattern,

enumeration, whiteSpace, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип timeDuration предназначен для хранения данных о времени. Значение этого типа состоит из шести составляющих. В нем указываются: год, месяц, день, час, минута и количество секунд. Формат описания значений этого типа содержится в стандарте ISO 8601. Порядок используется именно тот, в котором отдельные части были здесь перечислены. Для указания года используется четыре символа, на остальные элементы значения времени отводится по два символа.

Тип recurringDuration задает период времени, в течение которого будет с определенной частотой происходить или производиться некое действие. По существу, этот тип является набором значений типа timeDuration. Для значений этого типа обязательно должны быть явно заданы свойства duration и period. Помимо них могут использоваться еще свойства pattern, enumeration, whiteSpace, minInclusive, maxInclusime, minExclusive и maxExclusive.

Тип binary предоставляет возможность хранить и обрабатывать данные в двоичном виде, то есть, по сути, это реализация любых объектных вставок — графики, звука, видеоклипов и иных бинарных объектов. Для данных этого типа возможно применение свойств encoding, length, minLength, maxLength, pattern, enumeration и whiteSpace.

Тип uriReference позволяет задавать URI (Universal Resource Identifier) в качестве содержимого какого-либо элемента. Саму структуру URI мы уже рассматривали в предыдущих главах. Более детально она описана в документах RFC 2396 и RFC 2732. К данным этого типа применимы свойства length, minLength, maxLength, pattern, enumeration и whiteSpace.

Тип ID является функциональным близнецом токенизированного атрибута ID из спецификации обычного XML. Напомню, что значения этого типа содержат уникальные идентификаторы какого-либо экземпляра объекта в XML-документе. Этот тип позволяет использовать свойства length, minLength, maxLength, pattern, enumeration, whiteSpace, minInclusive, maxInclusime, minExclusive и maxExclusive.

Тип IDREF также является близнецом одноименного токенизированного атрибута обычных элементов XML-документа. В качестве значения этого типа могут применяться только реально существующие в XML-документе значения типа ID. Список возможных свойств такой же, как для типа ID.

Тип ENTITY означает, что величины этого типа предназначены для хранения неких сущностей. В качестве значения этого типа обязательно должно быть использовано имя предварительно объявленной сущности. Для нужд совместимости настоятельно рекомендуется значения с типом ENTITY применять только в качестве атрибутов. Значения этого типа имеют встроенные

свойства length, minLength, maxLength, pattern, enumeration, whiteSpace, minInclusive, maxInclusime, minExclusive и maxExclusive.

Все рассмотренные нами типы данных относятся к *основным* типам (primitive datatypes). На их основе были созданы *производные* типы данных (derived datatypes), которые мы тоже можем использовать для определения элементов и атрибутов. Пришло время рассмотреть и их.

Тип CDATA указывает, что в данной величине содержится символьная строка без включения символов форматирования. То есть в такой строке не могут находиться символы табуляции (шестнадцатеричный код — 9), символ перевода строки (код — A) и символ возврата каретки (код — D). Данный тип построен на основе базового типа string. Для данных этого типа можно использовать свойства length, minLength, maxlength, pattern, enumeration и whiteSpace.

На основе этого типа определяется тип token. Значения этого типа представляют собой набор символьных строк, в которых помимо уже наложенных ограничений на символы форматирования, запрещается использование обычных пробелов (шестнадцатеричный код — 20) в начале строки, а также не допускается наличие нескольких пробелов подряд в теле строки. В дальнейшем будем называть такие строки "токенизированными" (tokenized). Для данных этого типа допустимо применение тех же свойств, что и для типа CDATA. На основе данного типа строятся другие типы данных: NMTOKEN, NMTOKENS и Name.

Тип language предназначен для указания используемого языка. В качестве значений этого типа могут использоваться коды языков, определенные в документе RFC 1766. К данным подобного типа могут применяться свойства length, minLength, maxlength, pattern, enumeration и whiteSpace.

Тип IDREFS является аналогом одноименного токенизированного атрибута из спецификации XML. Как мы помним, данные подобного типа содержат набор строк, в которых хранятся значения идентификаторов элементов, логически связанных с заданным элементом, имеющим атрибут рассматриваемого типа. Для соблюдения правил совместимости для документов XML Schema настоятельно рекомендуется использовать этот тип данных только для атрибутов, но не для содержательных элементов. Значения рассматриваемого типа могут иметь свойства length, minLength, maxlength, enumeration и whiteSpace.

Тип ENTITIES, как и предыдущий рассмотренный нами тип, является аналогом одноименного типа атрибутов XML. В значении этого типа содержится набор наименований применяемых сущностей. В целях обеспечения совместимости данные типа ENTITIES также рекомендуется использовать только в качестве атрибутов. Данные этого типа обладают тем же набором свойств.

Тип NMTOKEN предназначен для хранения токенизированных величин, являющихся подстроками наименований каких-либо элементов. Данный тип построен на производном типе token. И здесь в интересах соблюдения принципов совместимости настоятельно советуется применять этот тип только для атрибутов. Данные описанного типа имеют свойства length, minLength, maxlength, pattern, enumeration и whiteSpace.

Тип NMTOKENS является логичным продолжением предыдущего типа. Данные этого типа содержат множество подстрок, выделенных из имени элемента. И конечно, этот тип тоже предназначен для атрибутивных значений, вместе с которыми разрешено использование свойств length, minLength, maxlength, enumeration и whiteSpace.

Тип Name также основан на типе token. Значения этого типа всегда содержат полное наименование соответствующего элемента. Они также обладают свойствами length, minLength, maxlength, enumeration и whiteSpace.

Тип NOTATION предназначен для поддержки одноименных атрибутов. Как мы помним, при помощи подобных атрибутов задается порядок обработки бинарных сущностей. Необходимо отметить, что в схемах документов данные этого типа обязаны быть перечислимого типа. То есть для них обязательно надо задавать значение свойства enumeration. Помимо этого свойства могут быть также использованы length, minLength, maxlength, pattern, whiteSpace, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип integer предназначен для хранения целочисленных значений. Декларирован на основе типа decimal. Значения этого типа представляют собой конечную последовательность десятичных цифр, предваряемую необязательным знаком плюса или минуса. В том случае, когда перед положительным числом ставится знак плюса, он игнорируется. Для данных этого типа определен следующий набор свойств: precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип nonPositiveInteger основан на типе integer. Этот тип предназначен для хранения неположительных целочисленных данных. То есть для величин типа integer принудительно установлено нулевое значение свойства maxInclusive. Неположительные целочисленные данные объединяют ноль и все целые отрицательные числа. При этом могут использоваться свойства precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

На основе только что рассмотренного типа определен тип negativeInteger, который используется для представления отрицательных целых чисел. Данные этого типа представляют собой конечную последовательность десятичных цифр, начинающуюся со знака унарного минуса. Значения такого типа обладают тем же набором свойств, что и тип nonPositiveInteger.

Тип long создан на основе типа integer. Этот тип хранит числа не в видепоследовательности цифровых символов, а сами бинарные представления чисел. Естественно, это налагает ограничения на диапазон хранимых данных. Для содержания одного числа отводится восемь байтов. Так как величины данного типа могут быть как отрицательными, так и положительными, границы допустимого диапазона значений установлены от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807. Данные этого типа обладают свойствами precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

На основе типа long построен тип int. Он также предназначен для представления целых чисел, но для хранения каждого числа отведено четыре байта. Таким образом, величины данного типа могут находиться в промежутке от –2 147 483 648 до 2 147 483 647. Данные типа int обладают свойствами precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Если же под хранение целого числа отводить не четыре байта, а два — мы получим тип short, декларируемый на основе типа int. Интервал допустимых значений для этого типа лежит от –32 768 до 32 767. Для данных типа short отводится все тот же набор свойств: precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Самым экономичным числовым типом в отношении расходования памяти является тип byte, основанный на базе типа short. Как следует из его названия, под хранение одного целого числа данного типа отводится один байт. Следовательно, можно использовать только числа в диапазоне от –128 до 127. Набор применяемых свойств остается все тот же: precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип nonNegativeInteger основан на типе integer и предназначен для хранения неотрицательных целых чисел. По сути, мы просто устанавливаем значение свойства minInclusive в ноль. Все остальное остается без изменений. В том числе и набор применяемых свойств: precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive. Естественно, использование типа nonNegativeInteger не заставляет нас использовать только нулевое значение свойства minInclusive. Просто значение этого свойства также обязано быть целым неотрицательным числом.

На базе последнего типа определен тип unsignedLong. Он предназначен для хранения беззнакового целого числа. Максимальное возможное значение свойства maxInclusive — 18 446 744 073 709 551 615. Минимальное, разумеется — 0. Данные этого типа обладают свойствами precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип unsignedInt является производным от типа unsignedLong. Разница между ними заключается в количестве байтов, отводимых под хранение одного числа. Тип unsignedInt — вдвое экономнее. Значения данных этого типа лежат в промежутке от нуля до 4 294 967 295 включительно. Для них установлен следующий набор свойств: precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип unsignedShort, построенный на базе только что рассмотренного нами типа unsignedInt, под одно число отводит всего два байта. Но так как мы используем в данном случае беззнаковые числа, то максимально возможное число этого типа — 65 535. Данный тип поддерживает свойства precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Последним из типов, предназначенных для хранения целых беззнаковых чисел, является unsignedByte. Легко понять, что для хранения отдельного числа этого типа применяется всего один байт. Таким образом, мы можем использовать рассматриваемый тип для чисел в промежутке от 0 до 255 включительно. К данным этого типа мы можем применять свойства precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive. Рассматриваемый тип основан на базе типа unsignedShort.

Тип positiveInteger предназначен для хранения и представления положительных целых чисел. Он построен на основе типа nonNegativeInteger. При хранении данных этого типа знак плюса и нули, стоящие в начале числа,игнорируются. Для этих чисел мы можем использовать свойства precision, scale, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип timeInstant предназначается для отображения данных, хранящих информацию о времени и дате. Он генерируется на основе типа recurringDuration при помощи жесткого задания значения свойств duration и period. Для обоих этих свойств используется значение "P0Y" (описание формата временных значений будет приведено ближе к концу главы). Таким образом, если мы хотим применить рассматриваемый тип для хранения момента времени, соответствующего 13 часам 20 минутам 31 марта 2000 года в часовом поясе, смещенном на пять часов относительно времени по Гринвичу, отображаться это значение будет как "2000-03-31T13:20:00-05:00". К данным этого типа могут применяться свойства duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип time позволяет хранить данные только о времени, без указания даты. Он также генерируется на основе типа recurringDuration при помощи жесткого задания значений свойств duration и period. При этом для свойства duration мы используем значение "P0Y", а для свойства period — "P1D". Та-

ким образом, то же самое время 13.20 в том же часовом поясе по прежнему будет отображено как "13:20:00-05:00". Данные рассматриваемого типа обладают свойствами duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип timePeriod используется для обработки и отображения неких промежутков времени, для которых явно задано время начала и окончания. Тип базируется на рассмотренном нами ранее типе recurringDuration и практически идентичен ему, за исключением того, что значение свойства period устанавливается в "P0Y" то есть исключается повторение данного промежутка времени. На самом деле данный тип не рекомендуется напрямую использовать в XML-документах, созданных по спецификации XML Schema. Рекомендуется же использовать типы данных, построенные на его основе. К данным этого типа могут применяться значения duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип date является первым из рассматриваемых нами типов данных, созданных на основе типа timePeriod. Он позволяет указывать промежуток времени величиной в одни сутки, который начинается в полночь указанной даты и заканчивается в полночь следующих суток. То есть, значение рассматриваемого типа "2000-12-06" указывает не на точку времени, а на временной промежуток, который начинается в полночь шестого декабря двухтысячного года и длится до полуночи седьмого декабря двухтысячного года. Обратите внимание, что в спецификации XML Schema сказано, что данный тип предназначен для обработки суточного интервала времени "вне зависимости от количества часов в этом дне". Для данных этого типа жестко фиксируется значение "P1D" для свойства duration, а в качестве данных для него же могут использоваться только даты стандартного Грегорианского календаря в виде "YYYY-MM-DD". Если необходимо использовать дату до нашей эры, то перед годом ставится знак минуса. К данным этого типа применяются свойства duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип month также основан на типе timePeriod. Он предназначен для определения промежутков времени длиною в месяц. Этот промежуток начинается в полночь первого дня указанного месяца и заканчивается в полночь первого дня следующего месяца, не включая ее. Добиться этого можно самостоятельно, установив значение "P1M" свойства duration. Данные этого типа имеют формат "YYYY-MM". Таким образом, чтобы обозначить декабрь двухтысячного года, мы должны использовать строку "2000-12". К величинам этого типа мы можем применять те же свойства: duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип year используется для хранения годичных промежутков времени. Он сформирован на основе типа timeDuration путем установки значения "P1Y"

свойства duration. Данные этого типа указываются в виде строк формата "YYYY". Искомый промежуток времени начинается в полночь первого дня указанного года, а заканчивается в полночь первого дня следующего года, не включая ее. Таким образом, весь прошедший двухтысячный год мы можем обозначить строкой "2000". Если необходимо указать год до нашей эры, то перед номером этого года, в котором, кстати, может быть больше четырех цифр, мы должны поставить знак минуса. Данные рассматриваемого типа обладают свойствами duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип century предназначен для обработки столетий. Период времени длиною в век получается из типа timeDuration путем принудительной установки свойства duration в значение "P100Y". При этом значения данного типа представляются в виде двух цифр, являющихся обозначением века в нумерации годов. Таким образом, если мы хотим задать двадцатый век, то мы должны использовать значение "19". Данный промежуток времени начинается в полночь первого дня указанного столетия, а заканчивается в полночь первого дня следующего века, не включая ее. Величины этого типа могут использовать свойства duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип recurringDate предназначается для обработки неких ежегодно повторяющихся дат. Он генерируется из типа recurringDuration путем жесткого задания значений свойств duration и period. При этом для свойства duration используется значение "P1D", а для свойства period — "P1Y". Значениями данного типа являются строки формата "MM-DD". То есть, если мы задаем значение "12-06", то мы указываем ежегодно повторяющуюся дату — шестое декабря. Естественно, при помощи величин такого типа чрезвычайно удобно задавать самые различные годовщины, такие, как дни рождения. Данные этого типа обладают свойствами duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

Тип recurringDay позволяет задавать ежемесячно повторяющиеся дни. То есть мы обозначаем только число без указания месяца. Этот тип генерируется на основе типа recurringDuration. Для этого фиксируются свойства duration и period. Для свойства duration задается значение "P1D", а для свойства period — "P1M". Данные этого типа записываются в виде двусимвольных строк, которые обозначают число повторяющегося дня. Для величин этого типа предусмотрены свойства duration, period, pattern, whiteSpace, enumeration, minInclusive, maxInclusive, minExclusive и maxExclusive.

На этом мы заканчиваем обзор предустановленных типов данных спецификации XML Schema. Несмотря на все их разнообразие, их может не хватить для каких-либо целей. Просто потому, что набор конечен. Поэтому в XML Schema добавлена возможность создавать и объявлять свои собственные типы данных. Ее мы рассмотрим в следующей части.

Создание новых типов данных

Как мы помним, в схеме документа мы можем самостоятельно разрабатывать новые типы данных. Для этого существует определенная схема объявления типа. Она выглядит следующим образом:

```
[name]
Необязательное наименование данного фрагмента схемы.
[target namespace]
Присоединяемое пространство имен. Мы можем использовать или нулевое значение, или URI подключаемого пространства имен.
[variety]
Указание вида создаваемого типа данных. Мы можем использовать атомарный тип (atomic), списочный (list) или объединение (union).
        atomic
        [атомарный тип]
        В этой секции указывается наименование одного из
        предустановленных типов данных XML Schema.
        list
        [Объявление типа элементов списка]
        Для элементов списка могут применяться атомарные типы или
        объединения.
        union
        [Определения типов элементов структуры]
        Непустая последовательность определений элементов
        объединения.
[facets]
Необязательный набор накладываемых ограничений и шаблонов на данные генерируемого типа.
[fundamental facets]
Основные ограничения и шаблоны.
[base type definition]
Необязательное указание базового типа, на основе которого генерируется данный тип.
[annotation]
Необязательное описание генерируемого типа.
```

Как видно, предусмотрены три основных класса создаваемых типов. Обычный атомарный тип генерируется при помощи наложения условий на один из базовых типов, рассмотренных в предыдущей части, и предназначен для хранения неделимых данных. Также мы можем создавать списочные типы, которые позволяют хранить набор однотипных элементов. И третий вариант — создание объединений, то есть случай, когда величина состоит из не-

скольких частей, для каждой из которых мы можем использовать свой собственный тип. Подобным образом мы можем создавать достаточно сложные структуры данных.

В спецификации правило объявления простого типа определяется следующим образом:

```
<simpleType
  id = ID
  name = NCName
{объявление дополнительных атрибутов}>
  Content: (annotation? , (restriction | list | union))
</simpleType>
```

Объявление типа является полноценным компонентом схемы, и, следовательно, может обладать неким набором свойств. В данном случае мы можем применять свойства `name`, `targetNamespace` и `annotation`. В них мы можем хранить наименования применяемых атрибутов, URL родительского пространства имен и аннотацию, соответствующую одному из дочерних элементов, если таковой, конечно, имеется.

Теперь попробуем расшифровать приведенный фрагмент спецификации. Для объявления типа необходимо использовать тэг `<simpleType>`, в котором в качестве параметров следует указать используемые атрибуты для объявляемого типа. Затем в качестве содержимого элемента мы можем использовать одну из трех конструкций: `restriction`, `list` или `union`. Каждая из них является самостоятельным компонентом схемы. Компонент `restriction` обычно применяется для генерации каких-либо атомарных типов данных, компонент `list` используется для создания списочных типов, а компонент `union` позволяет создавать типы в виде объединений. Завершает блок закрывающий тэг `</simpleType>`.

Определение компонента `restriction` выглядит следующим образом:

```
<restriction
  id = ID
  base = QName
   {объявление дополнительных атрибутов}>
  Content: (annotation? , (simpleType? , (minExclusive |
  ↳minInclusive | maxExclusive | maxInclusive | precision | scale |
  ↳length | minLength | maxLength | encoding | period | duration |
  ↳enumeration | pattern)*))
</restriction>
```

Из текста определения видно, что мы сначала должны использовать тэг `<restriction>`, в теле которого обязательно нужно задействовать один из идентифицирующих атрибутов. Чаще всего для этого используется атрибут `base`, в качестве значения которого мы должны указать наименование базо-

вого типа, на основе которого мы генерируем новый тип. Затем, в качестве основного содержимого компонента мы указываем набор из нескольких свойств, которые позволяют накладывать дополнительные ограничения и шаблоны на базовый тип. Завершается объявление типа закрывающим тэгом </restriction>. В качестве примера создания простого типа данных можно привести следующий код:

```
<simpleType name='Sku'>
    <restriction base='string'>
      <pattern value='\d{3}-[A-Z]{2}'/>
    </restriction>
</simpleType>
```

В этом блоке кода мы объявляем простой тип с наименованием Sku, построенный на базе основного типа string. Данные этого типа представляют собой строки, в начале которых находятся три десятичные цифры, а затем две заглавных буквы латинского алфавита, отделенные от цифр знаком дефиса. Этот шаблон мы задаем при помощи одного из параметров тэга <pattern>.

Списочные типы данных строятся с помощью компонента list. Последний определяется следующим образом:

```
<list
  id = ID
  itemType = QName
   {дополнительные атрибуты}>
  Content: (annotation? , simpleType?)
</list>
```

Легко заметить, что объявляется списочный тип при помощи тэга <list> с встроенными идентифицирующими атрибутами. В качестве содержимого этого компонента мы должны использовать определение типа элементов списка. В том случае, если все элементы списка имеют один из предопределенных базовых типов, наименование этого типа мы можем использовать в качестве значения атрибута itemType. Не подлежит обсуждению, что все элементы списка должны иметь один и тот же тип. Другое дело, что этот тип не обязан быть атомарным. Элементы списка сами могут являться структурами: списками или объединениями.

Следующий компонент схемы является объявлением списочного типа данных:

```
<simpleType name='listOfFloat'>
  <list itemType='float'/>
</simpleType>
```

В тэге <simpleType> мы задаем наименование генерируемого типа listOfFloat. В качестве содержимого этого элемента мы используем тэг <list>, в котором сразу указываем, что элементы данного списка имеют пре-

дустановленный тип `float`. Так как иного содержимого у этого тэга нет, мы оставляем его *пустым*, о чем сигнализирует наклонная черта в конце тэга.

В этой части нам осталось рассмотреть порядок создания *объединенного типа* данных — объединения. Объединенный тип фактически является аналогом структур в языке программирования Си или записей в языке Паскаль. В отличие от списочных типов, отдельные элементы данных объединенного типа не обязаны быть однотипными. Таким образом, мы получаем возможность оперирования набором разнородной информации как единым целым.

Объявляется объединенный тип данных при помощи компонента `union`. В официальной спецификации этот компонент определяется следующим образом:

```
<union
  memberTypes = list of QName
  id = ID
   {дополнительные атрибуты}>
  Content: (annotation? , simpleType*)
</union>
```

Нетрудно заметить, что тип отдельных элементов объединенного типа задается в виде списка типов, являющегося значением атрибута `memberTypes`. Намного легче понять механизм задания объединенных типов на конкретном примере.

В приведенном ниже примере мы задаем атрибут `size` для элемента `font`. Так как описание самого элемента может находиться в отдельном файле, нам придется использовать префиксы для компонентов схемы, идентифицирующие применяемое стандартное пространство имен типов данных. В данном случае мы используем префикс "`xsd`".

```
<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:attribute>
```

В этом примере нами задается размер используемого шрифта. При этом мы определяем тип атрибута `size` таким образом, что получаем возможность использовать либо целочисленное значение, лежащее в пределах от 8 до 72, либо одно из трех ключевых слов. Для этого в описание компонента `union` мы включаем два компонента `simpleType`. В первом из них мы используем базовый тип `positiveInteger`, для которого явно устанавливаем значения свойств `minInclusive` и `maxInclusive`, тем самым задавая пространство возможных значений. Для определения второго типа объединения мы используем токенизированный базовый тип, для которого обозначаем три возможных значения при помощи свойства `enumeration`. Теперь, после объявления этого типа, мы вправе использовать оба механизма задания значения атрибута `size`. Проиллюстрировать это можно при помощи следующего блока кода:

```
<paragraph>
<font size='large'>Заголовок</font>
</paragraph>
<paragraph>
<font size='12'>Обычный текст</font>
</paragraph>
```

Теперь, после того, как мы рассмотрели механизм создания своих собственных типов данных, необходимо узнать, какие ограничения мы можем налагать на базовые типы при помощи встроенных свойств.

Точное определение свойств

Мы уже рассматривали предопределенные свойства типов данных несколько ранее. Но нам нужно знать, как они определяются полностью. Тогда мы сможем в полном объеме регулировать свойства генерируемых типов. Поэтому в этой части мы рассмотрим официальные определения свойств и возможных их значений.

Но прежде чем приступить к рассмотрению отдельных свойств, мы узнаем, когда и в каких случаях применяется какое-либо из них.

Итак, если мы создаем списочный тип, то к данным этого типа могут применяться свойства `length`, `minLength`, `maxLength`, `enumeration`, `whiteSpace`. Необходимо осознавать, что этот набор свойств применяется не к элементам списка, а ко всему списку в целом.

Для объединенного типа данных мы можем использовать свойства `pattern` и `enumeration`.

А для каждого атомарного типа в спецификации определен свой набор свойств. Полный перечень соответствий приведен в табл. 4.1.

Таблица 4.1. *Соответствие типов данных и присущих им свойств*

Типы данных	Свойства
String	Length, minLength, maxLength, pattern, enumeration, whiteSpace
Boolean	Pattern, whiteSpace
Float	Pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Double	Pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Decimal	Precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
TimeDuration	Pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
RecurringDuration	Duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Binary	Encoding, length, minLength, maxLength, pattern, enumeration, whiteSpace
UriReference	Length, minLength, maxLength, pattern, enumeration, whiteSpace
ID	Length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
IDREF	Length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
ENTITY	Length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Qname	Length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
CDATA	Length, minLength, maxLength, pattern, enumeration, whiteSpace
String	Length, minLength, maxLength, pattern, enumeration, whiteSpace
Token	Length, minLength, maxLength, pattern, enumeration, whiteSpace
Language	Length, minLength, maxLength, pattern, enumeration, whiteSpace
NMTOKEN	Length, minLength, maxLength, pattern, enumeration, whiteSpace

Таблица 4.1 *(продолжение)*

Типы данных	Свойства
Name	Length, minLength, maxLength, pattern, enumeration, whiteSpace
NCName	length, minLength, maxLength, pattern, enumeration, whiteSpace
NOTATION	length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Integer	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
NonPositiveInteger	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
NegativeInteger	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
Long	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
Int	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
Short	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
Byte	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
NonNegativeInteger	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
UnsignedLong	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
UnsignedInt	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
UnsignedShort	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
UnsignedByte	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive
PositiveInteger	precision, scale, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive

Таблица 4.1 *(окончание)*

Типы данных	Свойства
TimeInstant	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Time	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
TimePeriod	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Date	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Month	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Year	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
Century	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
RecurringDate	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive
RecurringDay	duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive

А теперь нам предстоит как можно более полно рассмотреть определения всех перечисленных свойств.

Начнем мы со свойства `length`. Определяется оно следующим образом:

```
<length
  id = ID
  value = nonNegativeInteger
  fixed = boolean : false
  Content: (annotation?)
</length>
```

Как видно из декларации, это свойство (и все остальные тоже) является обычным элементом XML. Это и понятно, иначе мы не смогли бы его применять в XML-документах.

Посредством этого свойства можно устанавливать длину значения какого-либо типа. Для этой цели и служит атрибут `value`. Необходимо отметить, что

мы можем создавать новые типы на основе объявленного. При этом значение свойства `length` не будет наследоваться, так как для атрибута `fixed`, управляющего правилами наследования, по умолчанию задано значение `false`. То есть, если мы хотим, чтобы при наследовании значение свойства `length` сохранялось, необходимо принудительно установить атрибут `fixed` в `true`. Это правило достаточно хорошо иллюстрируется следующим примером:

```
<simpleType name='productCode'>
   <restriction base='string'>
     <length value='8' fixed='true'/>
   </restriction>
</simpleType>
```

Этим блоком кода мы объявляем новый тип `productCode`, который является текстовой строкой, содержащей восемь символов, не больше и не меньше. При этом мы указываем, что при наследовании типа значение свойства `length` должно сохраняться.

Свойство `minLength` предназначено для установки минимально возможной длины какого-либо типа. Определяется это свойство так, как указано ниже:

```
<minLength
  id = ID
  value = nonNegativeInteger
  fixed = boolean : false
  Content: (annotation?)
</minLength>
```

Невооруженным глазом видно, что спецификация этого свойства практически не отличается от спецификации свойства `length`. Тем не менее, по традиции, приведем пример:

```
<simpleType name='non-empty-string'>
  <restriction base='string'>
    <minLength value='1'/>
  </restriction>
</simpleType>
```

Мы объявляем тип `non-emty-string` (непустая строка) на основе базового типа `string`. Для того чтобы гарантировать, что значения этого типа действительно не будут пустыми, мы устанавливаем значение свойства `minLength` в единицу.

Свойство `maxLength` применяется для ограничения максимальной длины величины определяемого типа. Данный компонент спецификации XML Schema определяется, как показано ниже:

```
<maxLength
  id = ID
  value = nonNegativeInteger
```

```
  fixed = boolean : false
  Content: (annotation?)
</maxLength>
```

Так, если мы хотим ограничить длину значений генерируемого типа пятьюдесятью символами, мы должны записать приблизительно следующее определение типа:

```
<simpleType name='form-input'>
  <restriction base='string'>
    <maxLength value='50'/>
  </restriction>
</simpleType>
```

Свойство `pattern` является, пожалуй, одним из наиболее интересных и часто востребуемых свойств. Оно позволяет указывать шаблоны, налагаемые на величины определяемого типа, устанавливать возможный формат этих величин. Определяется это свойство следующим образом:

```
<pattern
  id = ID
  value = string
  fixed = boolean : false
   Content: (annotation?)
</pattern>
```

Легко увидеть, что в этом случае мы имеем строковое значение атрибута `value`. Именно в этой строке и описывается формат величины генерируемого типа в виде налагаемого на нее шаблона. Рассмотрению вопросов создания подобных шаблонов посвящена следующая часть этой главы. А пока мы просто взглянем на пример подобного наложения:

```
<simpleType name='better-us-zipcode'>
  <restriction base='string'>
    <pattern value='[0-9]{5}(-[0-9]{4})?'/>
  </restriction>
</simpleType>
```

В данном случае мы указываем, что значения создаваемого типа будут состоять как минимум из пяти цифр, и, возможно, к ним будут добавлены еще четыре цифры, отделенные от первой обязательной группы дефисом.

Свойство `enumeration` позволяет создавать перечислимые типы данных. То есть такие типы, величины которых могут принимать только одно из нескольких предопределенных значений. Определяется это свойство в соответствии со следующим описанием:

```
<enumeration
  id = ID
```

```
  value = string
  fixed = boolean : false
  Content: (annotation?)
</enumeration>
```

Тэг данного компонента спецификации XML Schema мы будем применять в объявлении генерируемого типа столько раз, сколько нам необходимо иметь предустановленных значений величины генерируемого типа. Это отлично видно на следующем примере:

```
<simpleType name='holidays'>
    <annotation>
        <documentation>Праздники</documentation>
    </annotation>
    <restriction base='recurringDate'>
      <enumeration value='--01-01'>
        <annotation>
           <documentation>Новый год</documentation>
        </annotation>
      </enumeration>
      <enumeration value='--03-08'>
        <annotation>
          <documentation>Международный женский день</documentation>
        </annotation>
      </enumeration>
    </restriction>
</simpleType>
```

Из примера видно, что мы используем два элемента `enumeration`. Установленные нами значения основаны на типе `recurringDate`, и ограничение, заложенное в его свойства, накладывается не на каждый из экземпляров элемента `enumeration`, а применяется к элементу `simpleType` в целом. Атрибуты `value` мы заполняем конкретными значениями дат, а расшифровку этих данных осуществляем при помощи аннотаций.

Свойство `whiteSpace` задает механизм обработки пробелов и символов форматирования в строковых величинах. Определяется это свойство, соответственно, так:

```
<whiteSpace
  id = ID
  value = preserve | replace | collapse
  fixed = boolean : false
  Content: (annotation?)
</whiteSpace>
```

Как видно из определения этого свойства, в качестве его значения мы можем использовать всего три предопределенных свойства. Механизм их действия мы рассматривали в одной из предыдущих частей. Здесь нам осталось только привести пример использования данного свойства в контексте определения нового типа:

```
<simpleType name='token'>
    <restriction base='CDATA'>
      <whiteSpace value='collapse'/>
    </restriction>
</simpleType>
```

Свойство `maxInclusive` позволяет устанавливать верхнюю границу промежутка допустимых значений для величины определяемого типа. Это значение включается в промежуток допустимых значений. Данное свойство определяется следующим фрагментом спецификации:

```
<maxInclusive
  id = ID
  value = string
  fixed = boolean : false
  Content: (annotation?)
</maxInclusive>
```

В качестве примера применения этого свойства мы можем рассмотреть следующее определение типа:

```
<simpleType name='one-hundred-or-less'>
  <restriction base='integer'>
    <maxInclusive value='100'/>
  </restriction>
</simpleType>
```

В этом фрагменте кода мы объявляем тип с "говорящим" названием `one-hundred-or-less` на основе целочисленного предопределенного типа. Для того чтобы данные этого типа не могли иметь величину, большую ста, мы явно устанавливаем соответствующее значение атрибута `value` для компонента, реализующего свойство `maxInclusive`. Несмотря на то, что тип атрибута `value` — строковый, в том случае, когда данное свойство применяется для числовых значений, происходит автоматическое преобразование типов.

Свойство `maxExclusive` задает верхнюю границу промежутка значений, исключаемых из множества допустимых значений величины генерируемого типа. Определяется это свойство следующим образом:

```
<maxExclusive
  id = ID
```

```
  value = string
  fixed = boolean : false
  Content: (annotation?)
</maxExclusive>
```

Фрагмент кода, иллюстрирующий применение данного свойства, выглядит так:

```
<simpleType name='less-than-one-hundred-and-one'>
  <restriction base='integer'>
    <maxExclusive value='101'/>
  </restriction>
</simpleType>
```

Все просто и достаточно прозрачно.

Свойство `minInclusive` предназначено для установки нижней границы промежутка допустимых значений для величины определяемого типа. Оно составляет пару с уже рассмотренным свойством `maxInclusive`, но не обязано использоваться вместе с ним. Определяется это свойство следующим образом:

```
<minInclusive
  id = ID
  value = string
  fixed = boolean : false
  Content: (annotation?)
</minInclusive>
```

Необходимо отметить, что в данном случае нижняя граница отрезка допустимых значений принадлежит самому отрезку, то есть включается в него. Проиллюстрировать действие этого свойства можно следующим объявлением типа:

```
<simpleType name='one-hundred-or-more'>
  <restriction base='integer'>
    <minInclusive value='100'/>
  </restriction>
</simpleType>
```

В этом определении мы основываем целочисленный тип, величины которого должны быть больше или равны ста. Если бы мы хотели, чтобы величины создаваемого типа находились в промежутке от ста до двухсот, включая указанные границы, то мы должны были бы использовать следующее модифицированное объявление:

```
<simpleType name='one-hundred-two-hundreds'>
  <restriction base='integer'>
    <minInclusive value='100'/>
    <maxInclusive value='200'/>
```

```
  </restriction>
</simpleType>
```

При помощи свойства `minExclusive` мы можем задавать нижнюю границу промежутка, исключаемого из множества возможных значений величин определяемого типа. Определяется это свойство по нижеследующим правилам:

```
<minExclusive
  id = ID
  value = string
  fixed = boolean : false
  Content: (annotation?)
</minExclusive>
```

Процесс применения этого свойства достаточно понятен, но проиллюстрировать его мы все-таки должны. В качестве иллюстрации действия свойства `minExclusive` мы приведем следующий фрагмент кода:

```
<simpleType name='more-than-ninety-nine'>
  <restriction base='integer'>
    <minExclusive value='99'/>
  </restriction>
</simpleType>
```

Здесь мы объявляем тип, основанный на базе стандартного целочисленного типа, величины которого должны быть более девяноста девяти.

Свойство `precision` позволяет ограничивать длину строкового представления числовых величин. В качестве значения этого свойства мы указываем максимальное количество цифр, применяемых для отображения целой части числа. По требованиям спецификации определяется данное свойство так:

```
<precision
  id = ID
  value = nonNegativeInteger
  fixed = boolean : false
  Content: (annotation?)
</precision>
```

В качестве примера использования рассматриваемого свойства можно привести следующее определение:

```
<simpleType name='amount'>
  <restriction base='decimal'>
    <precision value='8'/>
    <scale value='2' fixed='true'/>
  </restriction>
</simpleType>
```

В данном фрагменте кода мы определяем некий тип, предназначенный для хранения денежных величин. При этом мы не можем отображать сумму более десяти миллионов денежных единиц, о чем свидетельствует значение атрибута `value` соответствующего тэга `precision`. Так как обычно любая денежная единица отображается с точностью до двух десятичных знаков после запятой, мы фиксируем эти десятичные знаки при помощи тэга `scale`. Он соответствует одноименному свойству, которое мы прямо сейчас и рассмотрим.

Свойство `scale` предназначено для указания количества знаков после десятичной точки при отображении десятичной дроби. Формально определяется оно следующим образом:

```
<scale
  id = ID
  value = nonNegativeInteger
  fixed = boolean : false
  Content: (annotation?)
</scale>
```

А применение этого свойства мы рассматривали немного выше, поэтому сейчас нет нужды еще раз приводить разобранный ранее пример.

Свойство `encoding` определяет тип кодировки включаемых бинарных сущностей. Рассматриваемое свойство позволяет браузеру более корректно обрабатывать включения двоичных объектов в текст документа, созданного по требованиям спецификации XML Schema. Определение этого свойства в ней выглядит так:

```
<encoding
  id = ID
  value = hex | base64
  fixed = boolean : false
  Content: (annotation?)
</encoding>
```

Как видно, мы можем использовать только два предопределенных значения этого свойства: `hex` и `base64`. Пример применения данного свойства выглядит следующим образом:

```
<simpleType name='myBinary'>
  <restriction base='binary'>
    <length value='4'/>
    <encoding value='base64'/>
  </restriction>
</simpleType>
```

В определении этого типа мы указываем, что бинарные данные будут обрабатываться порциями по четыре байта в кодировке `base64`.

Свойство `duration` предназначено для установки длины некоего промежутка времени. Определяется оно, как показано ниже:

```
<duration
  id = ID
  value = timeDuration
  fixed = boolean : false
  Content: (annotation?)
</duration>
```

Так, если мы хотим создать тип, предназначенный для хранения срока годности какого-либо медицинского препарата, и принудительно установить двухгодичный срок годности, нам стоит воспользоваться приблизительно следующей конструкцией:

```
<simpleType name='PERIOD'>
  <restriction base='timePeriod'>
    <duration value='P2Y'/>
  </restriction>
</simpleType>
```

Свойство `period` позволяет задавать периодичность возникновения того или иного события. Его определение имеет вид:

```
<period
  id = ID
  value = timeDuration
  fixed = boolean : false
  Content: (annotation?)
</period>
```

Если мы хотим объявить тип для таких праздников, как дни рождения, которые имеют обыкновение ежегодно повторяться, то следует применить примерно следующую конструкцию:

```
<simpleType name='Birthday'>
  <restriction base='recurringDuration'>
    <duration value='P1D'/>
    <period value='P1Y'/>
  </restriction>
</simpleType>
```

На этом перечень предопределенных свойств типов данных спецификации XML Schema заканчивается. Нам осталось рассмотреть только вопросы создания шаблонов для строковых и временных значений.

Создание шаблонов

В этой части нам предстоит обратиться к теме создания шаблонов, управляющих форматированием строковых значений и обсудить способы записи значений свойств, связанных с управлением периодами времени. Начнем мы с форматирования строк.

Очень часто данные представляют собой строки специфичного вида, такие как номера телефонов, номенклатурные коды изделий и т. д. Эти данные очень часто вводятся людьми, которые весьма склонны к ошибкам. Для того чтобы избежать ошибок, связанных с человеческим фактором, имеет смысл наложение на строковые данные специальных шаблонов. Идея эта не нова и достаточно популярна, поэтому ее и использовали разработчики спецификации XML Schema.

В схемах документов для этих целей используется свойство `pattern`, о чем мы уже бегло говорили не один раз. Его значение и является шаблоном, накладываемым на строковые значения.

В предыдущей части мы уже рассматривали один подобный шаблон. Мы записали его в виде строки `[0-9]{5}(-[0-9]{4})?`. В данном случае шаблон указывает, что строка будет обязательно состоять из пяти десятичных цифр и необязательного дополнения из четырех цифр, отделенных от первой группы дефисом.

Значения шаблонов называются *регулярными выражениями* (regular expression). В их состав могут входить управляющие символы — так называемые Escape-последовательности и *символы-классы* (говоря иначе, символьные классы).

Эти классы позволяют указывать, какие символы могут находиться в том или ином месте результирующей строки. Обозначения этих классов и их расшифровку мы свели в единую таблицу (табл. 4.2).

Таблица 4.2. *Расшифровка символьных классов, используемых в шаблонах*

Класс	Содержание
`L`	Любой буквенный символ из набора Unicode
`Lu`	Буквенный символ верхнего регистра
`Ll`	Буквенный символ нижнего регистра
`Lt`	Заглавный символ
`Lm`	Модифицированный символ
`Lo`	Иные буквенные символы
`M`	Любой символ верстки или пробельный символ
`Mn`	Любой непробельный символ верстки

Таблица 4.2 *(окончание)*

Класс	Содержание
`Mc`	Любой пробельный символ
`Me`	Символ перевода строки
`N`	Любой цифровой символ
`Nd`	Любая десятичная цифра
`Ni`	Любой буквенный символ, обозначающий цифру (например, группа букв от `A` до `F`, применяемая в отображении чисел в шестнадцатеричной системе счисления)
`No`	Иной символ, обозначающий цифру
`P`	Любой знак пунктуации
`Pc`	Соединяющий знак пунктуации, такой как тире
`Pd`	Точка
`Ps`	Открывающая скобка
`Pe`	Закрывающая скобка
`Pi`	Открывающие кавычки
`Pf`	Закрывающие кавычки
`Po`	Иной символ пунктуации
`Z`	Любой разделяющий символ
`Zs`	Пробел
`Zl`	Разделяющая линия
`Zp`	Символ начала нового абзаца
`S`	Любой символ
`Sm`	Любой математический символ
`Sc`	Символ денежной единицы
`Sk`	Символ-модификатор
`So`	Иной символ
`C`	Любой символ, не относящийся к перечисленным категориям
`Cc`	Управляющий символ
`Cf`	Символ форматирования
`Co`	Символ, предустановленный пользователем
`Cn`	Зарезервированное сочетание, в данный момент не используется

Таким образом, если мы хотим указать, что строка должна состоять только из пяти десятичных цифр, мы должны использовать свойство `pattern` со

значением NdNdNdNdNd. Это верное определение, но неоптимальное. Для установки количества символов и других схожих целей применяются управляющие символы. Их мы поместили в табл. 4.3. Для удобства расшифровки мы будем пояснять их действие на примере абстрактной строки S.

Таблица 4.3. *Управляющие символы*

S?	Строка S может быть повторена любое количество раз, или не использоваться вообще
S*	Фактически аналогично действию предыдущего управляющего символа
S+	Строка S может быть повторена любое количество раз, но не менее одного раза
S{n,m}	Последовательность любых подстрок S длиной не менее n символов, и не более m символов. Естественно, n должно быть меньше m.
S{n}	Подстрока S длиной точно в n символов
S{n,}	Подстрока S длиной не менее n символов
S{0,m}	Подстроки S длиной до m символов, включая и пустую строку
S{0,0}	Пустая подстрока
(S)	Группировка символов регулярного выражения
[n,m]	Любой символ из промежутка кодов от n до m

Возвращаясь к рассмотренному шаблону из пяти десятичных цифр, мы можем теперь упростить его до вида Nd{5} или [0,9]{5}.

Как видно из таблицы, существует несколько символов, которые мы используем в качестве управляющих. Естественным образом, мы не можем их применять как обычные символы. А иногда это может понадобиться. Например, нам нужно включить в шаблон вопросительный знак или звездочку, скобки фигурные или квадратные. В этом случае мы должны использовать так называемые Escape-последовательности.

Давным-давно, во времена матричных принтеров и операционной системы DOS Escape-последовательностями назывались специальные последовательности символов, которые являлись командами для принтера. Они вставлялись прямо в текст печатаемого документа. Для того чтобы их можно было распознать, эти цепочки начинались всегда с неотображаемого символа Escape.

Время идет, многое меняется. Теперь нет нужды вставлять команды для принтера прямо в текст документа. Но понятие Escape-последовательности осталось. Теперь под ней понимают последовательность символов, замещающую собой какой-либо символ. В табл. 4.4 собраны все Escape-последовательности, применяемые в шаблонах спецификации XML Schema.

Правда, начинаются теперь они не с символа Escape, а с наклонной черты, но сути дела это не меняет.

Таблица 4.4. *Escape-последовательности*

Escape-выражение	Замещаемый символ
`\n`	Символ перевода строки (шестнадцатеричный код — `A`)
`\r`	Символ возврата каретки (шестнадцатеричный код — `D`)
`\t`	Символ табуляции
`\\`	Наклонная черта (`\`)
`|`	Вертикальная черта
`\.`	Точка
`\-`	Минус
`\^`	"Крышка" (`^`)
`\?`	Вопросительный знак
`\*`	Звездочка
`\+`	Плюс
`\{`	Открывающая фигурная скобка
`\}`	Закрывающая фигурная скобка
`\(`	Открывающая круглая скобка
`\)`	Закрывающая круглая скобка
`\[`	Открывающая квадратная скобка
`\]`	Закрывающая квадратная скобка

Вот теперь, наконец, мы перечислили все символы, из которых составляется шаблон форматирования, накладываемый на строковое значение.

В этом разделе мы еще собирались рассмотреть механизм формирования значений свойства `duration`, которое применяется к большинству типов значений, связанных с длительностью событий. Эти значения являются строками определенного формата, их создание очень похоже на создание шаблонов, потому мы и объединили рассмотрение и тех, и других.

Как мы знаем, свойство `duration` показывает длительность некоего события. Задается эта длительность при помощи текстовой строки формата "`PnYnMnDTnHnMnS`".

В этом определении формата все заглавные символы являются предустановленными, а в каждом конкретном случае строчные буквы `n` заменяются на соответствующие числа. Символ `P` обязательно присутствует в самом начале

строки и указывает, что это значение является периодом времени. Группа `nY` указывает количество лет, `nM` — количество месяцев, а `nD` — количество дней, входящих в отсчитываемый промежуток времени. Символ `T` является разделителем между разделами даты и времени. Группа символов `nH` устанавливает количество часов, `nM` — минут, а `nS` — секунд, входящих в определяемый промежуток времени. При этом, если какое-либо число равно нулю, то можно не указывать всю группу символов, включающую это число. Если же мы указываем промежуток с точностью только до дня, можно также опустить разделительный символ `T`.

Так, если нам требуется, чтобы некий промежуток времени длился ровно два года, нам достаточно использовать значение `P2Y`. А форма `P1Y2M3DT10H30M` может потребоваться для обозначения периода времени в один год, два месяца, три дня, десять часов и тридцать минут ровно.

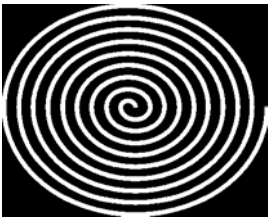
Несмотря на то, что обычно все интервальные значения, связанные со времяисчислением, имеют положительную меру длины, временные промежутки, обрабатываемые в свойстве `timeDuration`, могут быть и отрицательными. Такая возможность может пригодиться, если мы указываем момент окончания периода времени. Тогда правильное указание знака для периода времени позволит вычислить момент начала этого периода.

Для обозначения отрицательных промежутков времени достаточно перед стартовым символом `P` поставить знак минуса. Например, если нам надо обработать промежуток длительностью в минус сто двадцать дней, достаточно указать значение `-P120D`.

Все числа, используемые для определения данного свойства, являются целыми положительными. Кроме числа, показывающего количество секунд, входящих в период. Для более точного указания длительности исчисляемого периода времени количество секунд может задаваться числом с тремя десятичными знаками после запятой.

Итак, мы рассмотрели все механизмы формирования строк, являющихся значениями свойств `pattern` и `duration`. Это был последний этап, пройти который нам было необходимо для завершения рассмотрения технологии XML Schema. Теперь мы можем идти дальше.

Глава 5



Каналы CDF в Интернете

Переключая каналы

Когда я первый раз инсталлировал себе Windows'98 (как давно это было!), то больше всего меня удивили ярлычки каналов (channels) на рабочем столе. Тогда у меня даже модема не было, поэтому я не мог протестировать эту технологию, и знакомство с каналами было отложено до лучших времен. В конце концов, лучшие времена настали, и что такое каналы, я все-таки узнал.

Основным ограничением для всех субъектов Интернета является перманентно недостаточная толщина каналов доступа. К сожалению, мы не можем пока поддерживать прямое потоковое вещание видео- и аудиофрагментов в реальном времени. А использование любых активных элементов на Web-страницах связано с дополнительными затратами времени на перекачивание исполняемого кода этих активных элементов.

Давайте вспомним порядок работы обычного пользователя. При обращении к Web-странице происходит загрузка ее содержимого на локальный компьютер удаленного пользователя. Активные элементы, действующие на стороне пользователя, такие как скрипты, Java-апплеты или элементы ActiveX, также передаются пользователю. Точнее, передается их код. Естественно, код активных элементов или само содержимое страницы, особенно мультимедийные вставки, могут иметь немалый объем. И полной загрузки весьма наполненных страниц пользователь может и не дождаться. Согласно исследованиям Zone Research, среднестатистический пользователь готов ждать окончания загрузки не более семи секунд. А если на странице размещен апплет или элемент ActiveX размером около полумегабайта, то шансы на ее отображение в заданное время резко падают. Как уже было упомянуто, пропускная способность каналов связи ограничена.

Для выхода из этого технологического тупика и была создана *push-технология*. Согласно парадигме этой технологии не пользователь будет запрашивать информацию с сервера, а сам сервер переправит ему эту информацию в определенное время. Пользователь же просмотрит ее тогда, когда ему это будет удобно.

Конкретным результатом применения push-технологии и являются *каналы*. Технология использования каналов достаточно проста. После того, как пользователь подписался на канал, в определенное время согласно расписа-

нию на его локальный компьютер будет передаваться все содержимое этого канала, если, конечно, компьютер подписчика будет подключен в это время к Интернету. Если в назначенное время локальная система подписавшегося пользователя будет недоступна, то контент канала будет передан, как только подписчик войдет в Интернет. После передачи материалов клиент может в любое время просмотреть их.

Во время пересылки пользователь получает обновленное содержимое канала, включая Web-страницы, мультимедийные фрагменты и все необходимые программные коды. Естественно, при разработке канала следует использовать только те программные технологии, которые располагаются на стороне клиента. Любые интерактивные дополнения неприемлемы, поскольку к моменту просмотра канала пользователем все его содержимое уже должно находиться на локальной системе подписчика канала.

Исходя из этих соображений, все содержимое канала должно быть инкапсулировано. То есть в материалах канала не должно быть ни одной внешней ссылки, так как для перехода по ней пользователь должен послать запрос на получение этого ресурса, что противоречит идеологии push-технологии.

Сам канал описывается при помощи специализированного CDF-файла. А вот формат CDF (Channel Definition Format) является порождением технологии XML.

Здесь необходимо сделать маленькое отступление. Сам канал, то есть его содержимое состоит из Web-страниц, которые написаны на старом добром HTML. К XML, на первый взгляд, это не имеет никакого отношения. На самом деле, все немного сложнее. Все Web-страницы, входящие в состав канала, объединяются в единое целое при помощи вышеупомянутого CDF-файла, который и описывает *структуру* канала. А вот CDF-файл уже является полноценным XML-документом. В его основе лежит обычный DTD-файл.

Структура канала

У каждого канала есть несколько обязательных составляющих. В терминологии XML эти составляющие называются никак не иначе, как элементами. В этом разделе мы проведем беглый обзор самых основных структурных единиц канала — его элементов.

Итак, канал состоит из обычных Web-страниц. Если быть совсем точным, то каждый канал может содержать несколько субканалов, но общей картины это не меняет. У каждой Web-страницы, входящей в состав канала, есть свое графическое изображение, иначе называемое логотипом, идентифицирующее данную страницу. В состав Web-страниц входит обычный статичный контент и различные активные элементы, выполняемые на стороне пользователя.

Расписание обновления канала, как мы уже говорили, изначально устанавливается владельцем и издателем (publisher) канала. Удовлетворяющее пользователя расписание может быть получено путем изменения частоты обновления канала на его локальной системе. Эта возможность реализуется при помощи *элементов канала*.

В спецификации CDF все элементы разделены на старшие и младшие. Сначала, естественно, мы рассмотрим основные, старшие элементы.

- Элемент `Channel`. Является основным элементом канала, включает в себя все его содержимое. Именно посредством этого элемента канал и идентифицирует себя для браузеров.
- Элемент `Item`. Специфицирует отдельную Web-страницу, входящую в состав канала.
- Элемент `UserSchedule`. Ссылается на расписание обновлений, установленное подписчиком канала.
- Элемент `Schedule`. Предназначен для объявления общего расписания обновлений, которое предлагается по умолчанию владельцами канала.
- Элемент `Logo`. В качестве содержимого используются графические изображения предопределенного формата, являющиеся логотипами данного канала.
- Элемент `Tracking`. Описывает предпочтения пользователя, созданные на основе его действий на канале.
- Элемент `CategoryDef`. Устанавливает определение категории, к которой относится канал.

Для каждого из этих элементов существуют дочерние элементы, которые уточняют и дополняют информацию. Их рассмотрению посвящены следующие разделы.

Набор дочерних элементов всегда уникален. Но для старших элементов `Channel` и `Item` есть несколько общих дочерних элементов. Именно о них пойдет речь дальше.

Общие субэлементы

В этом разделе мы рассматриваем субэлементы, которые могут входить в качестве содержимого в элементы `Channel` и `Item`.

- Элемент `LastMod`. Содержит дату и время последнего обновления канала или Web-страницы. Тип элемента — `ISO 8601:1988 Date`, что подразумевает задание даты и времени в формате ISO.
- Элемент `Title`. Предназначен для хранения заголовка канала или Web-страницы. Заголовок хранится как обычная строка, то есть предопределенный тип элемента — `String`.

- Элемент `Abstract`. Содержит краткую аннотацию Web-страницы или канала. Рекомендуется использовать не более двух тысяч символов. Тип элемента, разумеется — `String`.
- Элемент `Author`. Позволяет хранить строку с именем автора данного канала или отдельной Web-страницы, входящей в его состав. Так как элемент строковый, то и тип у него соответствующий — `String`.
- Элемент `Publisher`. А этот элемент предназначен для указания издателя канала. Тип элемента — `String`.
- Элемент `Copyright`. Содержит официальное уведомление об авторских правах. Тип элемента — `String`.
- Элемент `PublicationDate`. Содержит строку, в которой указывается дата публикации Web-страницы или канала, то есть дата, когда владелец канала выложил его на свой сервер. Тип элемента — `String`.
- Элемент `Logo`. Содержит графическое изображение. По сути — логотип канала или отдельной Web-страницы. Элемент имеет специализированный тип — `Logo element`, который мы рассмотрим немного позже.
- Элемент `Keywords`. Имеет тип — `String`. Содержит список ключевых слов, описывающих содержимое данного канала или Web-страницы. Ключевые слова, как принято, разделяются запятой.
- Элемент `Category`. Описывает категорию, в которую входит данная Web-страница или канал. По сути, является строкой, в которой размещается URI, но так как этот URI указывает на специализированный элемент определения категории, тип обозначен как `Category element`.
- Элемент `Ratings`. Содержит значения различных рейтингов, на которых этот канал отслеживается. Тип элемента — `String`.
- Элемент `Schedule`. Является специализированным элементом, в котором указывается расписание обновления канала, установленное его издателем. Элемент имеет тип — `Shedule element`.
- Элемент `UserSchedule`. Ссылается на расписание обновления, переопределенное подписчиком канала. Тип элемента — `UserSchedule element`.

Определение перечисленных элементов в спецификации CDF выглядит следующим образом:

```
<!ELEMENT LastMod EMPTY>
<!ATTLIST LastMod VALUE CDATA #REQUIRED>
<!ELEMENT Title EMPTY>
<!ATTLIST Title VALUE CDATA #REQUIRED>
<!ELEMENT Abstract EMPTY>
<!ATTLIST Abstract VALUE CDATA #REQUIRED>
<!ELEMENT Author EMPTY>
```

```
<!ATTLIST Author VALUE CDATA #REQUIRED>
<!ELEMENT Publisher EMPTY>
<!ATTLIST Publisher VALUE CDATA #REQUIRED>
<!ELEMENT Copyright EMPTY>
<!ATTLIST Copyright VALUE CDATA #REQUIRED>
<!ELEMENT PublicationDate EMPTY>
<!ATTLIST PublicationDate VALUE CDATA #REQUIRED>
<!ELEMENT Keywords EMPTY>
<!ATTLIST Keywords VALUE CDATA #REQUIRED>
<!ELEMENT Category EMPTY>
<!ATTLIST Category VALUE CDATA #REQUIRED>
<!ELEMENT Rating EMPTY>
<!ATTLIST Rating PICS-Label CDATA #REQUIRED>
```

Это — полный список общих субэлементов, относящихся к элементам `Channel` и `Item`. Теперь переходим к детальному рассмотрению структур каждого элемента.

Элемент *Channel*

Основной элемент, который характеризует весь канал в целом. Все содержимое канала, все остальные элементы — наследуются от него. В спецификации данный элемент описывается следующим образом:

```
<!ELEMENT Channel ( LastMod | Title | Abstract | Author |
↳Publisher | Copyright | PublicationDate | Keywords | Category |
↳Rating | Channel | Item | Schedule | IntroURI | Authorization |
↳IsClonable | MinStorage | Tracking )* >
```

Как видно из описания, элемент `Item` является дочерним элементом по отношению к `Channel`. Большую часть субэлементов, относящихся к элементу `Channel`, мы обозрели в предыдущей части. А здесь нам осталось разобрать только шесть субэлементов, о которых еще не было упомянуто.

- Субэлемент `Channel`. Предназначен для создания нескольких субканалов в рамках одного общего канала. Тип элемента — `Channel` или обычная гиперссылка.
- Субэлемент `Item`. Является основной составной частью канала. Включает в себя одну или несколько Web-страниц, на которых и располагается контент канала. Тип элемента — `Item Profile`.
- Субэлемент `IntroURI`. Содержит ссылку на URI, по которому хранится вводная Web-страница для этого канала. Обычно на этой странице представлена общая информация о канале или правила его настройки. Естественно, тип этого элемента — `URI`.

- Субэлемент `Authorisation`. Содержит URI, по которому находится Web-страница с подписью создателя какого-либо активного элемента. В целях безопасности браузеры позволяют пользователю запускать исполнение активных элементов Web-страницы только в тех случаях, когда разработчик этих элементов заблаговременно провел их сертификацию. Специально для этих целей push-технология позволяет указывать страницы, на которых находится подтверждение сертификации. Тип элемента — `URI`.
- Субэлемент `MinStorage`. Позволяет указывать минимальный требуемый размер кэша для хранения канала. Тип элемента — `Number`.
- Субэлемент `Tracking`. Ссылается на профиль пользователя, показывающий один из предпочтительных вариантов прохождения по каналу. Тип элемента — `URL`.

Перечисленные выше субэлементы в спецификации описываются следующим блоком кода:

```
<!ELEMENT IntroURI EMPTY>
<!ATTLIST IntroURI VALUE CDATA #REQUIRED>
<!ELEMENT Authorization EMPTY>
<!ATTLIST Authorization VALUE CDATA #REQUIRED>
<!ELEMENT MinStorage EMPTY>
<!ATTLIST MinStorage VALUE CDATA "0">
```

Безусловно, помимо субэлементов в состав основного элемента `Channel` входят еще и его атрибуты. Для этого элемента предусмотрено всего два атрибута.

- Атрибут `HREF` указывает на местоположение последней обновленной версии данного канала. Атрибут содержит значение типа `URI`.
- Атрибут `IsClonable` определяет, можно или нет клонировать канал, то есть перемещать или копировать канал с сохранением его внутренней иерархии. Содержит значение типа `Boolean`. По умолчанию канал считается неклонируемым.

Определение этих атрибутов выглядит следующим образом:

```
<!ATTLIST Channel HREF CDATA #IMPLIED>
<!ATTLIST Channel IsClonable (YES | NO) "NO">
```

Элемент *Item*

По определению элемент `Item` содержит отдельную Web-страницу, написанную на языке HTML. Такие страницы и составляют типовое содержимое канала.

У каждой Web-страницы есть своя пиктограмма, логотип, который идентифицирует данную страницу, и позволяет переходить к ней сразу, без прохода по всей иерархии страниц канала, начиная с ее корневого элемента.

Данный элемент является одной из самых минимальных основных составляющих канала. Поэтому и субэлементов у него мало. Точнее, в его состав входит всего один субэлемент.

Субэлемент `Usage`. Показывает, каким именно образом используется данная Web-страница. В качестве значения выступает обычная текстовая строка.

Определение субэлемента в спецификации выглядит следующим образом:

```
<!ELEMENT Usage ANY>
<!ATTLIST Usage VALUE CDATA #REQUIRED>
```

Зато, в отличие от количества субэлементов, атрибутов у этого элемента вполне достаточно.

- Атрибут `HREF` указывает `URI`, по которому находится содержимое этой Web-страницы. Тип содержимого атрибута — `URI`.
- Атрибут `MIMEType`. Специфицирует тип MIME-содержимого адресуемой страницы. Тип атрибута — `String`.
- Атрибут `IsVisible` позволяет регулировать видимость данной Web-страницы. То есть при помощи этого атрибута мы можем задавать невидимые Web-страницы. Тип содержимого атрибута — `Boolean`. Значение по умолчанию — `YES`.
- Атрибут `Priority` предназначен для установки приоритета Web-страницы. В качестве содержимого атрибута может использоваться одно из трех предустановленных значений: `HI`, `NORMAL` и `LOW`. По умолчанию используется значение `NORMAL`.
- Атрибут `Precache`. При помощи этого атрибута издателем канала устанавливается необходимость кэширования страницы. Значение `YES` указывает, что данную страницу рекомендовано сохранять в кэше. Значение `NO` предлагает воздержаться от кэширования. А значение `DEFAULT`, используемое по умолчанию, перекладывает заботу об оптимизации загрузки страницы на пользователя.

Все эти атрибуты элемента `Item` определяются в спецификации следующим фрагментом кода:

```
<!ATTLIST Item HREF CDATA #REQUIRED>
<!ATTLIST Item MIMEType CDATA #IMPLIED>
<!ATTLIST Item IsVisible (YES, NO) "YES">
<!ATTLIST Item Priority (HI, NORMAL, LOW) "NORMAL">
<!ATTLIST Item Precache (YES, NO, DEFAULT) "DEFAULT">
```

На этом рассмотрение элемента `Item` мы заканчиваем и идем дальше.

Элемент *UserSchedule*

Как мы говорили, этот элемент применяется для хранения распоряжения подписчика канала о частоте обновления информации. Дочерних субэлементов этот элемент не имеет. Вся информация хранится в его единственном атрибуте.

Обязательный атрибут `VALUE`. Может содержать в качестве значения одно из трех предустановленных ключевых слов. Значение `DAILY` задает ежедневное обновление информации. Значение `WEEKLY` указывает, что подписчик канала решил обновлять его содержимое раз в неделю. А значение `HOURLY` обычно применяется нетерпеливыми пользователями, которые хотят получать новую информацию ежечасно.

Определение этого элемента имеет вид:

```
<!ELEMENT UserSchedule EMPTY>
<!ATTLIST UserSchedule VALUE (DAILY, WEEKLY, HOURLY) #REQUIRED>
```

Элемент *Schedule*

Этот элемент используется создателем канала для установки предпочтительного времени обновления информации и "закачки" ее на локальные системы подписчиков. Сам элемент определяется следующей конструкцией:

```
<!ELEMENT Schedule ( StartDate?, EndDate?, IntervalTime?,
↳EarliestTime?, LatestTime? ) >
```

Как видно, в его состав входит несколько дочерних субэлементов. Рассмотрим их поочередно.

- Субэлемент `StartDate`. Предназначен для указания даты, с которой обновление канала становится доступным для подписчиков. Тип значения элемента — `ISO 8601:1988 Date`.
- Субэлемент `EndDate`. Составляет пару с субэлементом `StartDate`. В нем указывается день, после которого обновление считается неактуальным, и прекращается его рассылка удаленным пользователям канала. Тип его значения тоже определяется как `ISO 8601:1988 Date`.
- В состав обоих этих субэлементов входит обязательный атрибут `VALUE`, в котором и хранится установленное значение элемента.
- Субэлемент `IntervalTime`. Задает интервал времени, в течение которого сервер будет пытаться передать обновление подписчику канала. Значение имеет тип `IntervalTime element`.
- Субэлемент `EarliestTime`. Предназначен для указания времени начала периода загрузки обновления канала для подписчиков. Значение субэлемента имеет тип `EarliestTime element`.

- Субэлемент `LatestTime`. Обычно работает в паре с предыдущим субэлементом. Задает время окончания периода загрузки обновления содержимого канала. Если его значение не совпадает со значением предыдущего субэлемента, то время загрузки обновления будет выбрано случайно, в пределах образовавшегося интервала. Значение субэлемента имеет тип `Latest Time element`.

Три последних субэлемента содержат в качестве значений некое время. Оно указывается при помощи атрибутов этих субэлементов. То есть перечисленные типы значений, по сути, одинаковы. Все субэлементы имеют одинаковый набор атрибутов.

Используются атрибуты `DAY`, `HOUR`, `MIN` и `SEC`. Какие значения хранятся в них — видно из их названий. Для всех атрибутов по умолчанию задаются нулевые значения.

В спецификации CDF определение всей совокупности субэлементов выглядит следующим образом:

```
<!ELEMENT StartDate EMPTY>
<!ATTLIST StartDate VALUE CDATA #REQUIRED>
<!ELEMENT EndDate EMPTY>
<!ATTLIST EndDate VALUE CDATA #REQUIRED>
<!ELEMENT IntervalTime EMPTY>
<!ATTLIST IntervalTime DAY CDATA "0">
<!ATTLIST IntervalTime HOUR CDATA "0">
<!ATTLIST IntervalTime MIN CDATA "0">
<!ATTLIST IntervalTime SEC CDATA "0">
<!ELEMENT EarliestTime EMPTY>
<!ATTLIST EarliestTime DAY CDATA "0">
<!ATTLIST EarliestTime HOUR CDATA "0">
<!ATTLIST EarliestTime MIN CDATA "0">
<!ATTLIST EarliestTime SEC CDATA "0">
<!ELEMENT LatestTime EMPTY>
<!ATTLIST LatestTime DAY CDATA "0">
<!ATTLIST LatestTime HOUR CDATA "0">
<!ATTLIST LatestTime MIN CDATA "0">
<!ATTLIST LatestTime SEC CDATA "0">
```

Элемент *LOGO*

Элемент `LOGO` предназначен для отображения логотипов канала или конкретной Web-страницы, входящей в его состав. Всего предусмотрено четыре различных типа для графических изображений с логотипами.

Один из логотипов предназначен для размещения на рабочем столе (Desktop) Windows. Другой вид логотипов применяется для отображения в панели каналов Microsoft Internet Explorer. Третий вид логотипов используется для идентификации отдельных Web-страниц. Первые два логотипа являются одновременно ссылками на основную страницу канала. Третий вид логотипов позволяет производить переход на идентифицируемую им Web-страницу. Еще один, четвертый вид логотипов предназначен для перенаправления пользователя на страницу поиска информации на Web-страницах канала.

Этот элемент не содержит каких-либо дочерних субэлементов. Но у него присутствуют два атрибута.

- Атрибут `HREF` предназначен для хранения гиперссылки на начальную страницу канала или на Web-страницу, которой принадлежит данное графическое изображение. Этот атрибут является обязательным. Тип значения атрибута — `URI`.
- Атрибут `TYPE` позволяет задавать вид данного логотипа. Атрибут является перечислимым. В качестве значения может быть применено одно из следующих четырех ключевых слов: `BIG`, `SMALL`, `WIDE` и `REGULAR`. По умолчанию используется значение `REGULAR`.

В заключение приведем определение этого элемента:

```
<!ELEMENT Logo EMPTY>
<!ATTLIST Logo HREF CDATA #REQUIRED>
<!ATTLIST Logo TYPE (BIG WIDE SMALL REGULAR) "REGULAR">
```

Элемент *Tracking*

Как мы помним, элемент `Tracking` позволяет задавать URL, по которому может находиться Web-страница, определенный ресурс, регистрирующий частоту обращения пользователя к страницам канала. Таким образом реализуется система статистики, позволяющей оценить, какое именно содержимое чаще всего просматривают подписчики канала.

Элемент `Tracking` содержит субэлемент `PostURL`, имеющий единственный атрибут `HREF`. В последнем указывается некий URL. Именно по этому адресу и отсылаются результаты статистики. Естественно, тип значения аргумента — `URL`.

Определяется этот элемент следующей конструкцией:

```
<!ELEMENT Tracking (PostURL?)>
<!ELEMENT PostURL EMPTY>
<!ATTLIST PostURL HREF CDATA #REQUIRED>
```

Элемент *CategoryDef*

При помощи данного элемента производится указание категории, к которой относится данный канал. Для более точного и полного описания категории применяются три субэлемента.

- Субэлемент `CategoryName`. Позволяет указывать наименование категории, к которой относится данный канал. Значение этого субэлемента имеет тип `String`.
- Субэлемент `Description` содержит в качестве своего значения строку, в которой расшифровывается краткое наименование категории, указанное в предыдущем субэлементе. Естественно, тип значения — тоже `String`.
- Субэлемент `CategoryDef`. Применяется для уточнения категории. Как нетрудно заметить, данный субэлемент имеет такое же наименование, как и родительский элемент. Следовательно, он позволяет организовывать обычную рекурсию, помогающую детализировать описание категории так полно, как это необходимо создателю канала. Тип значения этого субэлемента — `Category element`.

Пример создания канала

В этой части мы всего лишь приведем пример одного единственного CDF-файла, описывающего канал. Пример взят с сайта организации World Wide Web Consortium (W3C).

```
<!DOCTYPE Channel SYSTEM "http://www.w3c.org/Channel.dtd" >

<Channel HREF="http://www.foosports.com/foosports.cdf" IsClonable=YES >

<IntroUrl VALUE="http://www.foosports.com/channel-setup.html" />
<LastMod VALUE="1994.11.05T08:15-0500" />
<Title VALUE="FooSports" />
<Abstract VALUE="The latest in sports and atheletics from FooSports" />
<Author VALUE="FooSports" />

<Schedule>
<EndDate VALUE="1994.11.05T08:15-0500" />
<IntervalTime DAY=1 />
<EarliestTime HOUR=12 />
<LatestTime HOUR=18 />
```

```
</Schedule>
<Logo HREF="http://www.foosports.com/images/logo.gif"
     Type="REGULAR" />

<Item HREF="http://www.foosports.com/articles/a1.html">
<LastMod VALUE="1994.11.05T08:15-0500" />

<Title VALUE="How to get the most out of your mountain bike" />
<Abstract VALUE="20 tips on how to work your mountain-bike
                 to the bone and come out on top." />
<Author VALUE="FooSports" />
</Item>

<Channel IsClonable=NO >
<LastMod VALUE="1994.11.05T08:15-0500" />
<Title VALUE="FooSports News" />
<Abstract VALUE="Up-to-date daily sports news from FooSports" />
<Author VALUE="FooSports" />

<Logo HREF="http://www.foosports.com/images/newslogo.gif"
      Type="REGULAR" />

<Logo HREF="http://www.foosports.com/images/newslogowide.gif"
      Type="WIDE" />

<Item
HREF="http://www.foosports.com/articles/news1.html" >
<LastMod VALUE="1994.11.05T08:15-0500" />
<Title VALUE="Michael Jordan does it again!"/>
<Abstract VALUE="Led by Michael Jordan in scoring, the
                 Chicago Bulls make it to the playoffs again!" />
<Author VALUE="FooSports" />
</Item>

<Item
HREF="http://www.foosports.com/articles/news2.html" />
<LastMod VALUE="1994.11.05T08:15-0500" />
<Title VALUE="Islanders winning streak ends"/>
```

```
<Abstract VALUE="The New York islanders' 10-game winning streak
                 ended with a disappointing loss to the Rangers" />
<Author VALUE="FooSports" />
</Item>

</Channel>

<Item HREF="http://www.foosports.com/animations/scrnsvr.html" />
<Usage VALUE="ScreenSaver"></Usage>
</Item>

<Item
HREF="http://www.foosports.com/ticker.html" />
<Title VALUE="FooSports News Ticker" />
<Abstract VALUE="The latest sports headlines from FooSports" />
<Author VALUE="FooSports" />
<LastMod VALUE="1994.11.05T08:15-0500"
/>

<!-- This is an example of how Usage can be used
     for client enhancements -->
<Usage VALUE="DesktopComponent">
<Width VALUE=400 />
<Height VALUE=80 />
</Usage>

<Schedule>
<StartDate VALUE="1994.11.05T08:15-0500" />
<EndDate VALUE="1994.11.05T08:15-0500" />
<IntervalTime DAY=1 />
<EarliestTime HOUR=12 />
<LatestTime HOUR=18 />
</Schedule>
</Item>

</Channel>
```

Альтернативные стандарты

Мы рассмотрели официальную спецификацию CDF, которая была принята организацией W3C в качестве стандарта. Однако идея каналов была очень активно подхвачена компанией Microsoft. А компания Microsoft известна своей склонностью изменять стандарты.

Достаточно вспомнить, что сначала она внесла некоторые изменения в язык HTML, которые привели к тому, что Web-мастера были вынуждены создавать разные версии Web-страниц для браузеров Internet Explorer и Netscape Navigator. "Браузерная война" закончилась тем, что все дополнения были просто внесены в следующую версию стандарта HTML.

Затем Microsoft немного изменил язык JavaScript, фактически создав язык JScript, который может обрабатываться только браузером Internet Explorer. Фактически, компанией Microsoft был изменен стандарт языка Java, что послужило поводом для юридического иска компании Sun, являющейся владельцем Java.

Эта же история произошла и с CDF. Microsoft практически интегрировала ее в свою операционную систему Windows 98 (помните ярлычки каналов на рабочем столе?), введя концепцию ActiveChannel. Для того чтобы каналы могли максимально адекватно отображаться при помощи программных средств от Microsoft, в стандарт CDF были внесены некоторые изменения. Таким образом, если производители и издатели каналов хотят, чтобы их каналы максимально хорошо отображались у пользователей Windows, они вынуждены использовать измененную версию стандарта.

Однако ничего особенно предосудительного в этом нет. Все CDF-файлы являются XML-документами. XML-документы, в свою очередь, как мы знаем, базируются на DTD-блоках. Поэтому Microsoft достаточно было изготовить свой DTD-блок для технологии каналов, чтобы снять все претензии.

Microsoft разделяет каналы на три вида. Обычные, которые мы рассматривали, Microsoft называет "активными каналами". Второй вид каналов принят на вооружение для использования их в виде элементов Active Desktop. Подобные каналы могут отображать свое содержимое в виде "обоев" рабочего стола (Desktop) Windows или как обычный хранитель экрана (screensaver). И третий вид каналов, определенный Microsoft, носит название SUC (Software Update Channel) и предназначен, как легко заметить из наименования, для передачи обновлений программного обеспечения.

Так как мы планируем достаточно полно разобрать приложения XML, то нам стоит провести обзор и этих вариаций стандарта CDF. Ему мы и посвятим следующие разделы этой главы.

Стандарт Active Channel

Итак, как мы знаем, Microsoft для создания каналов рекомендует использовать собственную технологию Active Channel. Для этого сотрудниками Microsoft был разработан свой DTD-блок, который немного не совпадает с рассмотренным нами стандартом CDF.

Для создания каналов по технологии Active Channel необходимо использовать всего семнадцать тэгов. Из них один тэг является обычной исполнимой инструкцией для XML-процессора, а остальные — элементами, определенными в соответствующем DTD-блоке. Некоторые из них нам уже знакомы по рассмотренному официальному стандарту World Wide Web Consortium, а другие являются плодом творчества Microsoft. Так или иначе, мы рассмотрим их все.

В самом начале CDF-документа всегда должна стоять исполняемая инструкция XML-процессора, которая сигнализировала бы браузеру, что данный файл является XML-документом. Для CDF-файлов, созданных по технологии Active Channel, данная исполняемая инструкция будет иметь следующий вид:

```
<?XML
    ENCODING = "char-set"
    VERSION = "n.n"
?>
```

При этом необязательный параметр `ENCODING` предназначен для указания кодировки, которая применялась для создания текстового содержимого канала, а значение обязательного параметра `VERSION` указывает, какая версия стандарта XML применялась для создания CDF-файла.

В качестве примера обычно приводится следующая инструкция:

```
<?XML ENCODING="UTF-8" VERSION="1.0"?>
```

Как говорят сами специалисты Microsoft, использование именно этой инструкции является "хорошей идеей", так как для англоязычных XML-документов чаще всего используется именно кодировка UTF-8, а версия XML 1.0 на данный момент является текущей.

Эта инструкция может использоваться не только в XML-документах каналов, но и в OSD-файлах, которые применяются для каналов обновления программного обеспечения. Эти каналы созданы с использованием корпоративного стандарта OSD (Open Software Description), и рассматривать мы их будем в одном из следующих разделов этой главы.

Основным элементом в технологии Active Channel остается элемент с наименованием `CHANNEL`. Но он определен несколько иначе, нежели в спецификации World Wide Web Consortium. Для Active Channel тэг этого элемента определяется следующим образом:

```
<CHANNEL
    BASE = "url"
    HREF = "url"
    LASTMOD = "date"
    LEVEL = "n"
    PRECACHE = "sCache"
>
```

Естественно, потом, в самом конце CDF-файла, этот тэг необходимо будет закрыть, используя для этого его дополняющую конструкцию </CHANNEL>.

Как видно, в этом элементе есть некоторое количество атрибутов, которые мы не встречали в официальной спецификации. Рассмотрим все атрибуты, присутствующие в этом элементе.

Необязательный, опциональный атрибут BASE задает URL, относительно которого будут рассчитываться остальные URL. Смысл в том, что каждый субканал или Web-страница, входящая в состав канала, по сути, обладают своей собственной файловой структурой, ссылка на которую производится при помощи URL. Если мы явно зададим значение атрибута BASE, то URL всех вложенных элементов CHANNEL и ITEM будут считаться относительными, и точкой отсчета для них будет считаться URL, введенный в качестве значения атрибута BASE.

Так как атрибут BASE может входить в состав каждого экземпляра элемента CHANNEL, то для всех вложенных субканалов его действие будет одинаковым. Если в каком-либо субканале установлен атрибут BASE, то он отменяет действие атрибута BASE, обозначенного в родительском канале, и для всех элементов этого субканала URL будут отсчитываться относительно ресурса, указанного в атрибуте BASE субканала.

Необходимо отметить, что URL в атрибуте BASE обязан завершаться прямым слешем "/", иначе последнее слово в этом URL будет удалено.

Атрибут HREF содержит URL основной страницы канала. Именно на эту страницу будет осуществлен переход, если подписчик канала щелкнет кнопкой мыши на логотипе канала. Атрибут HREF не будет действовать, если основной элемент CHANNEL содержит субэлемент A. Этот субэлемент мы рассмотрим несколько позже.

Необязательный атрибут LASTMOD содержит дату и время последнего редактирования канала. Значение этого атрибута записывается в формате yyyy-mm-ddThh:mm. То есть год записывается при помощи четырех цифр, затем следует месяц (две цифры) и число (две цифры). Дату отделяет от времени символ "T". Два символа, в которых записывается час, отделяются от пары символов с минутами знаком двоеточия. Для отметки времени обновления канала используется время по гринвичскому меридиану.

Атрибут LEVEL. Является необязательным атрибутом. Используется обычно для вложенных субканалов. Значение этого атрибута указывает уровень вложенности субканала (в субканалы могут вкладываться другие дочерние каналы). Основной канал имеет нулевой уровень вложенности. Браузер Internet Explorer (а технология Active Channel адекватно распознается только этим браузером) при получении запроса на отображение канала с этим атрибутом автоматически кэширует все каналы и субканалы, у которых уровень вложенности меньше, чем у запрошенного канала.

Необязательный атрибут PRECACHE позволяет управлять принудительным кэшированием содержимого канала. В качестве его значения могут использоваться только два ключевых слова: No и Yes. Значение по умолчанию Yes заставляет браузер принудительно кэшировать содержимое канала или субканала. Значение No указывает, что кэширование содержимого канала не должно производиться. В последнем случае значение атрибута в тэге <CHANNEL> игнорируется.

Данный элемент может в качестве родительского элемента иметь только элемент типа CHANNEL.

Необходимо отметить, что в технологии Active Channel ссылка на заглавную страницу канала может производиться двумя способами. Для этой цели мы можем применять либо атрибут HREF тэга <CHANNEL>, либо субэлемент гиперссылки, реализуемый при помощи тэга <A>.

Следующим по значимости элементом технологии Active Channel мы можем назвать элемент ITEM. Мы уже встречали его, когда рассматривали официальный стандарт CDF. В данном случае его реализация и предназначение практически не отличается от исходной. Как и ранее, элемент ITEM указывает на Web-страницу, которая входит в состав канала или субканала. В технологии Active Channel данный элемент определяется следующим образом:

```
<ITEM
    HREF = "url"
    LASTMOD = "date"
    LEVEL = "n"
    PRECACHE = "sCache"
>
```

Как видно, этот элемент имеет атрибуты HREF, LASTMOD, LEVEL и PRECACHE. Мы уже встречали эти атрибуты в декларации элемента CHANNEL. Здесь они действуют точно так же. Единственное, о чем стоит упомянуть, это то, что URL, являющийся значением атрибута HREF, не должен включать в себя более 255 символов.

Элемент ITEM может являться дочерним субэлементом для элемента CHANNEL. В качестве дочерних элементов могут использоваться: все тот же CHANNEL, элементы ABSTRACT, A, LOG, LOGO и TITLE.

Как пример применения данного элемента можно предложить следующую конструкцию:

```
<ITEM HREF="http://www.joyware.com/promo.htm"
      LASTMOD="1998-08-17T10:30"
      PRECACHE="YES">
```

Видно, что в этом блоке кода мы объявляем Web-страницу, расположенную на сайте www.joyware.com, которая последний раз обновлялась 17 августа 1998 года. При этом мы указываем, что данную Web-страницу необходимо кэшировать.

Для создания статистики просмотра отдельных Web-страниц используется элемент LOG. Он позволяет вести так называемую службу логгинга. Определяется этот элемент следующим образом:

```
<LOG
    VALUE = "document:view"
/>
```

Мы видим, что в состав этого элемента входит атрибут VALUE. В качестве значения этого атрибута используется текстовая строка, указывающая, какое именно событие мы будем регистрировать. На данный момент браузер Internet Explorer распознает только одно значение: document:view. Что значит, что мы регистрируем просмотр подписчиком канала данной Web-страницы.

Для элемента LOG родительским элементом является ITEM. Дочерних субэлементов не предусмотрено.

Пусть мы хотим, чтобы обновление содержимого канала могли получать только те подписчики, которые вовремя оплатили получение новой порции информации. Для этого необходимо поддерживать некую базу данных и заставлять подписчика вводить пароль, который будет его идентифицировать. Для того чтобы канал мог проводить авторизацию пользователя, нужно включить в его состав элемент LOGIN. Определяется этот элемент следующим образом:

```
<LOGIN
/>
```

Невооруженным глазом видно, что этот элемент — пуст и не содержит никаких атрибутов. Он является всего лишь индикатором, который указывает, что данный канал должен затребовать у подписчика его идентификатор.

Рассматриваемый элемент всегда выступает как дочерний для элемента CHANNEL. Сам элемент LOGIN, естественно, не может иметь дочерних элементов.

Широко применяемые счетчики посещений всегда реализуются при помощи некоторых активных элементов. Чаще всего для этого используются приложения, связываемые с Web-страницей при помощи интерфейса CGI

(Common Gateway Interface). Для осуществления этой связи в технологии Active Channel применяется элемент LOGTARGET. Определяется он, как показано ниже:

```
<LOGTARGET
    HREF = "url"
    METHOD = "POST"
    SCOPE = "sScope"
>
```

Атрибут HREF содержит URL, указывающий на страницу с соответствующим активным элементом, который и производит подсчет посещений канала. Данный атрибут является обязательным, как и трибут METHOD, в качестве значения которого всегда используется ключевое слово POST, управляющее способом пересылки данных от Web-страницы на сервер.

Необязательный атрибут SCOPE определяет, как именно будет вестись статистика, какие события будут считаться посещениями канала его подписчиками. Web-страницы канала можно просматривать как в активном подключении, подобно обычным сайтам, так и в офлайне, если страницы были предварительно загружены в кэш.

В качестве значения данного атрибута могут использоваться всего три ключевых слова. Значение OFFLINE указывает, что подсчитывать следует только просмотры предварительно кэшированных страниц. Значение ONLINE позволяет вести статистику посещения страниц при прямом подключении. Значение по умолчанию ALL регистрирует оба вида посещений.

Необходимо отметить, что для того, чтобы статистика могла вестись корректно, пользователь в момент просмотра Web-страниц канала должен быть подключен к Интернету, и иметь активное соединение, так как иначе переход по URL, указанному в значении параметра HREF, не осуществится и счетчик посещений не сработает.

Для данного элемента родительским является элемент CHANNEL. Дочерними элементами являются PURGETIME и HTTP-EQUIV.

Как пример для данного элемента можно привести следующий фрагмент кода:

```
<LOGTARGET
      HREF="http://www.joyware.tld/logging"
      METHOD="POST"
      SCOPE="OFFLINE">
```

Так как данный элемент может иметь дочерние элементы, то он не является пустым, и, следовательно, каждый раз при его включении в документ мы должны использовать закрывающий тэг </LOGTARGET>.

Элемент PURGETIME позволяет устанавливать максимально возможный "возраст" счетчика. Мы знаем, что счетчики посещений всегда выкладываются на саму Web-страницу, а в нашем случае — на основную страницу канала. Так как Web-страницы канала могут сначала сохраняться в кэше и только потом просматриваться пользователем, который к тому времени уже может находиться в офлайне, счетчик может немного "устареть". Он не будет отображать адекватную информацию. При помощи элемента PURGETIME мы можем задавать максимальное время жизни счетчика в кэше. По истечении этого времени браузер будет пытаться обновить значение счетчика.

Тэг данного элемента определяется следующим образом:

```
<PURGETIME
    HOUR = "n"
/>
```

У этого элемента есть обязательный атрибут HOUR, в котором указывается количество часов. А именно время, в течение которого счетчик посещений будет считаться адекватным. Сам счетчик обычно хранится в личной папке пользователя. Для этой цели в ней создается папка history, а в ней дополнительно создается папка log. Именно там хранится файл со счетчиком посещений канала. Для файла счетчика устанавливается расширение .log. По истечении заданного интервала времени этот файл очищается, то есть его содержимое просто удаляется.

Элемент PURGETIME является дочерним по отношению к элементу LOGTARGET. Так как он сам является пустым элементом, у него не может быть дочерних элементов.

Вместе с этим элементом обычно применяется элемент HTTP-EQUIV. Он предназначен для задания дополнительных параметров, регулирующих передачу данных по протоколу HTTP. У протокола HTTP существует множество так называемых заголовков, которые позволяют регулировать отдельные аспекты передачи данных по каналам связи. Вот при помощи данного элемента и устанавливаются значения этих заголовков.

Вид этого определения в спецификации:

```
<HTTP-EQUIV
    NAME = "headerparam"
    VALUE = "text"
/>
```

Здесь обязательный атрибут NAME предназначен для задания имени используемого заголовка HTTP, а атрибут VALUE, также являющийся обязательным, позволяет устанавливать значение параметра заголовка.

Данный элемент является дочерним по отношению к элементу LOGTARGET.

В целом, определение статистического счетчика может быть осуществлено как в приведенном ниже фрагменте кода, в котором задействованы все три рассмотренных элемента:

```
<LOGTARGET
      HREF="http://www.joyware.com/logging"
      METHOD="POST"
      SCOPE="OFFLINE">
   <PURGETIME
         HOUR="12"/>
   <HTTP-EQUIV
         NAME="ENCODING-TYPE"
         VALUE="gzip"/>
</LOGTARGET>
```

В этом примере мы задаем подсчет статистики посещений страниц канала, которые были предварительно загружены в кэш браузера. При этом время жизни счетчика посещений устанавливается равным двенадцати часам. Для управления передачей данных мы устанавливаем параметр `ENCODING-TYPE` в значение `gzip`.

В технологии Active Channel предусмотрен двоякий способ задания ссылки на основную страницу канала. Для этого мы можем использовать либо атрибут `HREF` элемента `CHANNEL`, либо специализированный элемент `A`. Впрочем, последний элемент позволяет устанавливать ссылку не только на основную страницу канала, но и на Web-страницу, являющуюся содержимым элемента `ITEM`.

Тэг этого элемента определяется следующим образом:

```
<A
    HREF = "url"
/>
```

Обязательный атрибут `HREF` в качестве своего значения содержит URL Web-страницы, на которую необходимо установить ссылку.

Данный элемент является дочерним по отношению к элементам `CHANNEL` и `ITEM`. Собственных дочерних субэлементов он не имеет.

Следующий фрагмент кода иллюстрирует применение элемента `A` в качестве субэлемента:

```
<ITEM>
<A HREF="http://www.joyware.com/george.htm">Link to George page</A>
</ITEM>
```

Необходимо отметить, что URL, который указан в качестве значения параметра `HREF`, может отображаться браузером в качестве заголовка канала, если таковой не задан при помощи специального элемента `TITLE`.

Элемент TITLE позволяет задавать название канала или отдельной Web-страницы, входящей в его состав. Тэг этого элемента объявляется следующим образом:

```
<TITLE
    XML-SPACE = "sWhiteSpace"
/>
```

Данный элемент имеет необязательный атрибут XML-SPACE, позволяющий задавать порядок обработки пробелов, используемых в текстовом содержимом этого элемента.

Как значения для данного атрибута могут применяться два ключевых слова. Значение DEFAULT разрешает браузеру фильтровать пробелы в заданной строке заголовка. То есть браузер сам управляет размещением пробелов, убирает сдвоенные пробелы и устанавливает размеры пробельных символов. Значение PRESERVE сохраняет все пробелы в строке заголовка в том виде, в каком они были введены.

Элементы CHANNEL и ITEM являются родительскими по отношению к элементу TITLE. Сам он, в свою очередь, не имеет дочерних субэлементов.

Применение тэга данного элемента можно проиллюстрировать следующим фрагментом кода:

```
<TITLE>Самый лучший канал РуНета!</TITLE>
```

Помимо заголовка мы можем устанавливать текст оперативной подсказки, "хинта", который будет появляться на экране компьютера, если пользователь задержит курсор мыши над логотипом или заголовком канала (или отдельной Web-страницы, входящей в его состав). Для этого применяется элемент ABSTRACT. Тэг данного элемента определяется следующим образом:

```
<ABSTRACT
    XML-SPACE = "sWhiteSpace"
/>
```

Видно, что у этого элемента, подобно элементу TITLE, тоже есть атрибут XML-SPACE. Этот атрибут является необязательным, и позволяет устанавливать порядок обработки пробелов браузером. Возможные значения этого атрибута мы только что рассматривали.

Элемент ABSTRACT является дочерним по отношению к элементам CHANNEL и ITEM. Сам дочерних субэлементов не имеет.

Следующий блок кода иллюстрирует применение данного элемента:

```
<ABSTRACT>На этой странице находится анализ движения котировок акций
на Токийской фондовой бирже</ABSTRACT>
```

Элементы, управляющие обновлением содержимого канала в технологии Active Channel, тоже немного отличаются от стандартных. Основной эле-

мент носит наименование SCHEDULE. Определение тэга этого элемента имеет следующий вид:

```
<SCHEDULE
    STARTDATE = "date"
    STOPDATE = "date"
    TIMEZONE = "offset"
>
```

Как легко заметить, в состав этого элемента входят три дополнительных атрибута. Ни один из них не является обязательным, так как помимо них существуют еще дочерние субэлементы, которые позволяют задавать расписание обновления содержимого канала. Тем не менее, они помогают более точно описывать параметры обновления.

Атрибут STARTDATE позволяет устанавливать дату начала обновления канала. Значение этого атрибута имеет вид yyyy-mm-dd. То есть первые четыре цифры указывают год, следующие две — номер месяца, и последние две — число.

Атрибут STOPDATE предназначен для определения даты, по достижении которой обновление содержимого канала прекращается. Значение этого атрибута имеет такой же вид, как и для предыдущего атрибута STARTDATE.

Атрибут TIMEZONE фиксирует временной пояс. Значением этого атрибута является смещение временного пояса, в котором физически находится сервер с данным каналом, относительно стандартного восточного поясного времени Соединенных Штатов. Предусмотрено два варианта задания значения данного атрибута. Можно задавать смещение либо целыми часами (–5), либо с указанием и часов и минут (0630). К значению мы можем добавлять знаки "+" или "–", регулирующие направление смещения.

Для данного элемента родительским является элемент CHANNEL. В качестве дочерних субэлементов могут использоваться элементы EARLIESTTIME, LATESTTIME и INTERVALTIME.

Элемент INTERVALTIME позволяет указывать период времени, в течение которого будет повторяться загрузка обновления содержимого канала. Данный элемент является пустым. Его тэг определяется следующим образом:

```
<INTERVALTIME
    DAY = "n"
    HOUR = "n"
    MIN = "n"
/>
```

Атрибуты DAY, HOUR и MIN дают возможность установки длины периода времени, в течение которого будут повторяться попытки загрузить обновление на локальную машину подписчика канала. Эти атрибуты позволяют задавать продолжительность периода в днях, часах и минутах соответственно. При-

меняться может любая комбинация этих атрибутов. В качестве значений используются целые положительные числа.

Для данного элемента родительским является элемент SCHEDULE. Дочерних субэлементов он по определению иметь не может.

Элемент EARLIESTTIME предназначен для детализации данных, указанных в элементе INTERVALTIME. Он позволяет указывать начало периода повторения пересылки. Определение элемента выглядит следующим образом:

```
<EARLIESTTIME
    DAY = "n"
    HOUR = "n"
    MIN = "n"
/>
```

Атрибут DAY задает первый день внутри периода, установленного элементом INTERVALTIME. Если используется элемент EARLIESTTIME, то обновление начнется именно с указанного в нем дня. При помощи этого элемента мы определяем дополнительное смещение. Причем, мы можем использовать не только целые дни, но и часы или минуты, применяя для этого атрибуты HOUR и MIN соответственно.

Данный элемент также является дочерним по отношению к элементу SCHEDULE.

Блок элементов, управляющих расписанием обновления, завершается элементом LATESTTIME. Он позволяет указывать срок, после которого прекращается загрузка обновления. Тэг этого элемента выглядит следующим образом:

```
<LATESTTIME
    DAY = "n"
    HOUR = "n"
    MIN = "n"
/>
```

Как видно, этот элемент имеет тот же набор атрибутов, что и предыдущие два. Стоит ли говорить, что и действуют они точно так же, как и их аналоги-близнецы?

В качестве примера действия вышеописанных элементов можно привести следующий фрагмент кода:

```
<SCHEDULE STARTDATE="2000-12-01" STOPDATE="2000-12-31">
    <INTERVALTIME DAY="1" />
    <EARLIESTTIME HOUR="10" />
    <LATESTTIME HOUR="14" />
</SCHEDULE>
```

В этом примере мы указываем, что обновление содержимого канала будет передаваться на локальные машины подписчиков ежедневно в течение декабря 2000 года с 10 до 14 часов. При помощи параметров элемента SCHEDULE мы задаем даты начала и конца периода обновления, при помощи параметра INTERVALTIME мы устанавливаем ежедневные соединения с целью закачки обновления. Конкретное время, в течение которого будут производиться попытки соединения, фиксируется элементами EARLIESTTIME и LATESTTIME.

При помощи элемента LOGO мы можем задавать графические изображения, которые могут использоваться в качестве логотипов канала или отдельных Web-страниц, входящих в его состав. Тэг данного элемента определяется следующим образом:

```
<LOGO
    HREF = "url"
    STYLE = "sStyle"
/>
```

Как видно из определения — это пустой элемент, вся необходимая информация заключена в двух его обязательных атрибутах.

Атрибут HREF в качестве значения содержит URL графического файла, в котором и находится изображение, применяемое в качестве логотипа. Атрибут STYLE уточняет тип используемого изображения. В качестве значения этого атрибута может применяться только одно из трех возможных ключевых слов.

Значение ICON указывает, что данное графическое изображение является иконкой отдельной Web-страницы. Значение IMAGE специфицирует изображение как предназначенное для размещения на рабочем столе Windows в списке каналов. А ключевое слово IMAGE-WIDE устанавливает, что данное изображение нацелено на применение в панели каналов браузера Internet Explorer.

Элемент LOGO является дочерним субэлементом по отношению к элементам CHANNEL и ITEM. Сам дочерних субэлементов не имеет.

Последний элемент, который мы рассматриваем в рамках технологии Active Channel, носит наименование USAGE. Он является дочерним субэлементом по отношению к элементу ITEM, и указывает, как именно должна отображаться данная Web-страница, входящая в состав канала.

В спецификации тэг этого элемента определяется следующим образом:

```
<USAGE
    VALUE = "sValue"
/>
```

Значение обязательного атрибута VALUE и является индикатором формы использования и отображения данной части канала. В качестве значения может применяться одно из шести предустановленных ключевых слов.

Значение CHANNEL указывает, что данная Web-страница будет отображаться в панели каналов браузера. Это стандартный режим использования Web-страниц, входящих в состав канала. Если для них не использовать элемент USAGE, то результат будет таким же, как если бы мы использовали этот элемент со значением атрибута CHANNEL.

Значение DesktopComponent используется в том случае, если данный канал или Web-страница, входящая в его состав, является элементом Active Desktop и отображается на рабочем столе в виде одного из его фреймов.

Значение Email применимо в случае посылки обновления данного компонента канала по электронной почте.

Значение NONE указывает, что данная Web-страница не будет отображаться в панели каналов браузера Internet Explorer.

Значение ScreenSaver используется только для того компонента канала, который будет применяться на локальной машине подписчика в виде хранителя экрана. Согласно спецификации Active Channel такой элемент в канале может быть только один.

Значение SoftwareUpdate применяется в том случае, если данный канал создан для доставки обновлений программного обеспечения (эту возможность мы рассмотрим несколько позже).

Следующий фрагмент кода иллюстрирует применение данного элемента:

```
<ITEM HREF="http://www.joyware.com/screensaver.htm">
    <USAGE VALUE="ScreenSaver"></USAGE>
</ITEM>
```

На этом мы заканчиваем обзор технологии Active Channel. Нас ждет рассмотрение элементов Active Desktop.

Элементы Active Desktop

Как мы знаем, в свои операционные системы компания Microsoft встраивает технологию Active Desktop. Последняя позволяет отображать содержимое каналов в виде отдельных фреймов. Эти фреймы могут отображаться либо на рабочем столе, если тот находится в режиме Active Desktop, или как хранитель экрана, если при инсталляции Windows установили хранитель "Переключающиеся каналы".

Конечно, транслируемые каналы и их Web-страницы должны быть созданы по технологии Active Channel, причем значение элемента USAGE должно быть установлено в DesktopComponent.

При этом открываются дополнительные возможности по управлению внешним видом фрейма, в котором отображается содержимое какой-либо Web-страницы, входящей в состав канала.

Для компонентов активного рабочего стола предусмотрено несколько дополнительных элементов, которые являются дочерними субэлементами по отношению к элементу USAGE.

Элемент CANRESIZE дает возможность изменения размеров фрейма, в котором отображается содержимое канала или одной из его Web-страниц. Тэг этого элемента определяется следующим образом:

```
<CANRESIZE
    VALUE = "sSizes"
/>
```

Этот элемент имеет один атрибут, который не является обязательным. Атрибут может принимать только два возможных значения.

Используемое по умолчанию значение Yes указывает, что фрейм, отображающий данную страницу, может изменять свои размеры, если пользователю необходимо это сделать. Значение No "замораживает" размеры фрейма.

Действие этого элемента перекрывает действие элементов CANRESIZEX и CANRESIZEY.

Элементы CANRESIZEX и CANRESIZEY назначают статус изменения размеров фрейма по горизонтали и вертикали соответственно. Их тэги определяются при помощи следующих конструкций:

```
<CANRESIZEX
    VALUE = "sSizes"
/>
<CANRESIZEY
    VALUE = "sSizes"
/>
```

Как видно, синтаксис этих элементов схож с синтаксисом элемента CANRESIZE. И значения их атрибутов тоже повторяют значения единственного атрибута этого элемента.

Элемент WIDTH предназначен для задания ширины фрейма, в котором отображается содержимое данной Web-страницы. Тэг этого элемента имеет вид:

```
<WIDTH
    VALUE = "pixels"
/>
```

Данный элемент содержит обязательный атрибут VALUE, значением которого является устанавливаемая ширина фрейма. Ширина измеряется в пикселах.

Высота фрейма регулируется при помощи элемента HEIGHT. Форма этого элемента определяется следующей конструкцией:

```
<HEIGHT
    VALUE = "pixels"
/>
```

Как видно, конкретное значение высоты фрейма устанавливается посредством его обязательного атрибута VALUE. В качестве единиц измерения используются все те же пикселы.

При помощи элемента OPENAS мы можем указывать, в каком виде мы будем открывать Web-страницу, реализуемую родительским элементом ITEM. Есть два варианта. Открыть и отобразить этот компонент канала мы можем как обычную Web-страницу, либо как графическое изображение, то есть логотип данной страницы.

Тэг этого элемента описывается следующим фрагментом кода:

```
<OPENAS
    VALUE = "sValue"
/>
```

Обязательный атрибут VALUE в качестве своего значения может содержать одно из двух предустановленных ключевых слов.

Значение HTML, установленное по умолчанию, указывает, что данный элемент Active Desktop будет отображаться как обычная Web-страница. Значение Image отображает этот элемент в виде графического изображения.

На этом мы заканчиваем анализ элементов, входящих в состав расширения технологии Active Channel. На очереди обзор возможностей установки и обновления программного обеспечения при помощи push-технологии.

Software Update Channel

В свою технологию Active Channel компания Microsoft включила возможность доставки новых версий какого-либо программного обеспечения подписчику канала и инсталлирования на его локальной системе. В принципе, это — достаточно здравая идея, так как push-технология практически идеально для таких целей подходит. Серверу абсолютно неважно, какую именно информацию передавать подписчику канала в качестве обновления содержимого этого самого канала.

Известно, что в своем первозданном виде стандарт CDF или технология Active Channel не позволяют передавать, обновлять или инсталлировать программное обеспечение. Требовалась некоторая доработка стандарта. И Microsoft создала соответствующие расширения для своей технологии Active Channel. При создании этих расширений были приняты во внимание уже существовавшие к тому времени разработки OSD (Open Software Description) и MSICD (Microsoft Internet Component Download). Созданная технология

получила наименование SUC (Software Update Channel). Этот раздел посвящен обзору данной технологии.

Как только что было отмечено, SUC является лишь расширением технологии Active Channel, некоей надстройкой над этим неофициальным стандартом. То есть для того, чтобы создать канал, предназначенный для передачи и инсталляции новых версий программного обеспечения, необходимо в разработке использовать некоторые дополнительные элементы. При этом никак нельзя забывать, что данный канал не предназначен для отправки отображаемой информации. То есть в качестве его компонентов не будут использоваться привычные Web-страницы. Компонентами SUC-канала должны быть файлы OSD и MSICD, которые являются XML-документами. Для этих файлов существуют свои наборы элементов, применяемых для структурирования и идентифицирования передаваемой информации.

Для того чтобы иметь возможность создавать подобные каналы и файлы, их составляющие, необходимо изучить все элементы, входящие в состав спецификации SUC. Их рассмотрением мы сейчас и займемся.

Элемент `SOFTPKG` является основным элементом, регулирующим порядок передачи и установки программного обеспечения. Тэг этого элемента описывается следующим фрагментом кода:

```
<SOFTPKG
    AUTOINSTALL = "bAutoInst"
    HREF = "url"
    NAME = "string"
    PRECACHE = "bCache"
    STYLE = "sMechanism"
    VERSION = "sVersion"
>
```

Как видно, в состав этого элемента входит несколько атрибутов, позволяющих детализировать информацию о передаче программного обеспечения.

Обязательный атрибут `HREF` позволяет назначить URL, который будет указывать расположение Web-страницы, относящейся к этому элементу.

Атрибут `NAME`, также определенный как обязательный, в качестве своего значения содержит строку, являющуюся товарным именем данного программного обеспечения. Длина имени не может превышать 260 символов.

Опциональный атрибут `AUTOINSTALL` поясняет браузеру, необходимо ли автоматически инсталлировать программное обеспечение после его скачивания, или предоставить эту возможность пользователю. Значение этого атрибута является, по сути, булевым выражением, и, соответственно, здесь могут использоваться только два предустановленных ключевых слова. Значение `No`, устанавливаемое по умолчанию, указывает, что загруженное обновление

программного обеспечения не следует автоматически инсталлировать на локальную машину подписчика SUC-канала. Значение `Yes` принуждает систему запускать программу установки данного программного обеспечения сразу после окончания его загрузки.

Необязательный атрибут `PRECACHE` регулирует предварительное кэширование пересылаемой информации. Как мы помним, обновление может быть кэшировано на машине подписчика канала по требованию. Значением этого атрибута также является булева величина. Значение по умолчанию `No` указывает, что данный компонент не будет подвергаться предварительному кэшированию. Значение `Yes` заставит браузер это кэширование все-таки произвести. Естественно, если мы используем значение `Yes` и одновременно установим опцию автоматического инсталлирования, возникнет некоторая коллизия. Поэтому атрибут `AUTOINSTALL` в данном случае будет иметь больший приоритет.

Атрибут `STYLE` также является необязательным. Он применяется для выбора способа, которым будет осуществляться передача и инсталляция программного обеспечения. На выбор предлагаются технология Active Setup, построенная на исполняемых элементах ActiveX, и альтернативный вариант MSICD — Microsoft Internet Component Download. Рассмотрение различий между ними выходит за рамки нашего обзора. Для указания того, какая именно технология будет применяться в данном случае, используются значения `ActiveSetup` и `MSICD`. Что именно означают эти ключевые слова, ясно уже из их наименований.

Необязательный атрибут `VERSION` в качестве значения может содержать строку, в которой указывается версия загружаемого программного обеспечения.

Данный элемент может использоваться в рамках обеих указанных выше технологий, как MSICD, так и OSD.

Элемент `DELETEONINSTALL` относится к CDF. Он используется в документе, определяющем структуру всего канала. Применяется он только в том случае, если при успешной загрузке и установке новой версии программного обеспечения предыдущую версию необходимо удалить или деинсталлировать. Определяется тэг этого элемента очень просто:

```
<DELETEONINSTALL
/>
```

Для этого элемента родительским является только что рассмотренный нами элемент `SOFTPKG`. Очевидно, что рассматриваемый элемент не имеет дочерних субэлементов.

Элемент `JAVA` сообщает, что поставляемое программное обеспечение написано на языке Java. По сути, этот элемент является лишь контейнером. Он не содержит никаких атрибутов. У него есть лишь дочерний субэлемент

PACKAGE, который именно и предназначен для более детального указания информации о передаваемом программном обеспечении.

Элемент PACKAGE служит для описания передаваемого Java-пакета или приложения Java. Тэг данного элемента описывается следующим образом:

```
<PACKAGE
    NAME = "name"
    VERSION = "sVersion"
>
```

Обязательный атрибут NAME содержит текстовую строку, в которой указывается название поставляемого программного пакета или приложения. В необязательном атрибуте VERSION фиксируется номер его версии. Значение этого атрибута является текстовой строкой, содержащей четыре числа, разделенные точками. То есть версия формализуется в соответствии с общепринятым стандартом — основной номер, дополнительный, номер релиза, номер билда.

Элемент PACKAGE является дочерним по отношению к элементу JAVA. В качестве его дочерних субэлементов используются элементы NEEDTRUSTEDSOURCE, SYSTEM, CLASS и IPLEMENTATION.

Элементы JAVA и PACKAGE могут применяться только в составе элементов MSICD.

Элемент CLASS имеет смысл задействовать только в том случае, если передаваемое программное обеспечение является одним из классов Java. Тэг этого элемента выглядит в общем случае так:

```
<CLASS
    CLASSID = "id"
    NAME = "classname"
>
```

У данного элемента есть обязательный атрибут NAME. Значением этого атрибута является текстовая строка, в которой записывается имя файла, содержащего поставляемый класс Java.

Атрибут CLASSID является необязательным, опциональным. В его значение заносится десятичное или шестнадцатеричное число, являющееся уникальным числовым идентификатором (GUID) поставляемого класса.

Рассматриваемый элемент является дочерним по отношению к элементу PACKAGE. В свою очередь, он сам является родительским элементом для субэлементов ICON, ISBEAN и TYPELIB.

Характерным примером применения этого тэга может являться следующий:

```
<CLASS
    NAME="Browserbean"
```

```
    CLASSID="{01234567-89AB-CDEF-0123-456789ABCDEF}">
       <ISBEAN />
       <ICON FILENAME="Browser.ico" />
       <TYPELIB CLASSID="{01234567-89AB-CDEF-0123-456789ABCDEF}" />
</CLASS>
```

Как вы заметили, в этом примере мы используем упомянутые дочерние субэлементы. Что они означают — мы узнаем несколько позже.

Элемент CLASS может применяться только в составе элементов MSICD.

Элемент NATIVECODE является дочерним субэлементом по отношению к элементу SOFTPKG. По своему синтаксису и назначению он представляет собой аналог элемента PACKAGE. Но если элемент PACKAGE применяют для обработки программных пакетов Java, то элемент NATIVECODE позволяет описывать иные исполняемые компоненты, например, библиотечные модули. Но он, как и элемент JAVA, служит некоей оболочкой, контейнером для вложенных дочерних элементов. А вот уже дочерние субэлементы могут более детально описывать передаваемые исполняемые конструкции.

Данный элемент имеет только один дочерний субэлемент — CODE.

Элемент NATIVECODE может применяться только в составе элементов MSICD.

Элемент CODE предназначен для указания детализированной информации о поставляемых исполняемых блоках, если это не Java-пакеты. Тэг этого элемента описывается следующим образом:

```
<CODE
    CLASSID = "id"
    NAME = "classname"
    VERSION = "sVersion"
>
```

Обязательный атрибут NAME содержит текстовую строку, в которой указывается имя файла поставляемого компонента. Обычно это ocx- или dll-файл. Имя в текстовом значении этого атрибута указывается без полного пути.

Опциональный атрибут CLASSID содержит в качестве своего значения десятичный или шестнадцатеричный уникальный числовой идентификатор передаваемого и устанавливаемого компонента.

Атрибут VERSION также является необязательным. Его значением является текстовая строка, в которой записывается полный номер версии программного компонента. По умолчанию используется строковое значение "0.0.0.0".

Как мы знаем, рассматриваемый элемент является дочерним по отношению к элементу NATIVECODE. В свою очередь, он сам является родительским по отношению к элементам SYSTEM и IMPLEMENTATION.

Элемент CODE может применяться только в составе элементов MSICD.

Элемент IMPLIMENTATION позволяет указывать конфигурацию, необходимую для использования поставляемого программного обеспечения. Тэг этого элемента определяется следующим образом:

```
<IMPLEMENTATION
>
```

Данный элемент используется лишь в качестве оболочки для дочерних субэлементов, которые позволяют передавать более детальную информацию.

Для данного элемента родительскими являются элементы CODE, PACKAGE и SOFTPKG. В качестве дочерних субэлементов могут применяться CODEBASE, LANGUAGE, OS и PROCESSOR.

Элемент IMPLEMENTATION может использоваться как в MSICD-, так и в OSD-элементах.

Элемент CODEBASE предназначен для указания места, по которому расположен ресурс, откуда и передается обновление программного обеспечения. Тэг этого элемента определяется следующим образом:

```
<CODEBASE
    FILENAME = "file"
    HREF = "url"
    SIZE = "n"
    STYLE = "sMechanism"
/>
```

Обратите внимание, что атрибуты этого элемента схожи с атрибутами элемента SOFTPKG. Но если указанный элемент характеризовал все обновление полностью, то рассматриваемый нами сейчас элемент задает информацию об отдельном передаваемом программном компоненте или приложении. Все четыре атрибута являются необязательными.

Атрибут FILENAME содержит имя передаваемого файла. Обычно в данном случае используются архивные cab-файлы или файлы OSD. В качестве значения атрибута HREF указывается URL передаваемого файла. Атрибут SIZE содержит некое число, обозначающее размер в килобайтах для скачиваемого программного компонента. Значение атрибута STYLE определяет выбор используемого механизма перекачивания информации. Выбор осуществляется при помощи значений ActiveSetup и MSICD. Что означают эти ключевые слова, мы уже знаем.

Элемент CODEBASE может использоваться как в MSICD-, так и в OSD-элементах.

Элемент LANGUAGE предназначен для указания языка, на который рассчитано поставляемое программное обеспечение. Тэг данного элемента определяется следующим образом:

```
<LANGUAGE
    VALUE = "language"
/>
```

У данного элемента присутствует обязательный атрибут VALUE, значением которого является код используемого языка. При этом использоваться может не один язык, а несколько. В этом случае их коды в текстовой строке, являющейся значением атрибута, будут разделяться точкой с запятой. Конкретные коды указаны в документе RFC 1766.

Данный элемент является дочерним по отношению к элементам IMPLEMENTATION и SOFTPKG.

Следующий короткий пример демонстрирует применение этого элемента:

```
<LANGUAGE VALUE="en"/>
```

Этим блоком кода мы указываем, что передаваемое программное обеспечение ориентировано на англоязычных пользователей.

Элемент LANGUAGE может использоваться как в MSICD-, так и в OSD-элементах.

Элемент ISBEAN применяется только при передаче пакетов Java. Он является дочерним по отношению к элементу CLASS. Указывает, что передаваемый класс выполнен по технологии JavaBean. Тэг элемента более чем прост:

```
<ISBEAN
/>
```

Данный элемент может применяться только в составе элементов MSICD. Дочерних элементов не имеет.

Элемент ICON позволяет задавать имя графического файла, в котором находится пиктограмма для поставляемого Java-класса, выполненного по технологии JavaBean. Тэг этого элемента определяется следующей конструкцией:

```
<ICON
    FILENAME = "file"
/>
```

Обязательный атрибут FILENAME содержит текстовую строку, в которой указывается искомое имя графического файла.

Данный элемент является дочерним по отношению к элементу CLASS. Соответственно, он может использоваться только в элементах MSICD.

Элемент NAMESPACE предписывает, что для передаваемого Java-приложения (именно приложения) будет применяться правило определения частных

имен. Это одна из возможностей технологии Java. Тэг данного элемента имеет вид:

```
<NAMESPACE
/>
```

Данный элемент является дочерним по отношению к элементу JAVA. Может применяться только в MSICD-элементах.

Пустой элемент NEEDTRUSTEDSOURCE необходимо использовать в тех случаях, когда получатель поставляемого Java-пакета должен полагаться на репутацию поставщика программного обеспечения, так как данный Java-пакет не подписан специализированной цифровой подписью. Этот элемент является дочерним по отношению к элементу PACKAGE.

Элемент NEEDTRUSTEDSOURCE может применяться только в составе элементов MSICD.

Элемент OS специфицирует оперативную систему, которая поддерживает поставляемое программное обеспечение. Тэг этого элемента выглядит следующим образом:

```
<OS
    VALUE = "sOS"
>
```

Обязательный атрибут VALUE в качестве значения может содержать одно из трех зарезервированных ключевых слов. Значение Mac используется для обозначения операционных систем, действующих на платформе Macintosh. Ключевое слово Win95 предназначено для обозначения операционных систем Win'9x и Win2000. Значение Winnt указывает, что поставляемое программное обеспечение действует в операционных системах семейства Windows NT.

Данный элемент является дочерним по отношению к элементу IMPLEMENTATION. В свою очередь, он имеет дочерний субэлемент OSVERSION.

Элемент OS может использоваться как в MSICD-, так и в OSD-элементах.

Элемент OSVERSION конкретизирует номер версии необходимой операционной системы. Тэг этого элемента определяется следующим образом:

```
<OSVERSION
    VALUE = "sVersion"
/>
```

Обязательный атрибут VALUE в качестве своего значения содержит текстовую строку, в которой указывается полный номер версии. Как обычно, этот номер состоит из четырех чисел, детализирующих номер старшей и младшей версии, номер релиза и билда.

Данный элемент является дочерним по отношению к элементу OS. Элемент OSVERSION может использоваться как в MSICD-, так и в OSD-элементах.

Элемент PROCESSOR позволяет указывать тип процессора, на котором может выполняться передаваемое программное обеспечение. Определяется тэг этого элемента следующей конструкцией:

```
<PROCESSOR
    VALUE = "sProcessor"
/>
```

Значение обязательного атрибута VALUE конкретно указывает тип необходимого процессора. Всего может использоваться четыре предустановленных ключевых слова. Применяются значения Alpha, MIPS, PPC и x86, обозначающие соответствующие типы процессоров.

Данный элемент является дочерним по отношению к элементу IMPLEMENTATION.

Элемент PROCESSOR может использоваться как в MSICD-, так и в OSD-элементах.

Элемент SYSTEM применяется в тех случаях, когда поставляется системное программное обеспечение. То есть, если поставляются какие-либо Java-пакеты, то в них находятся системные классы для виртуальной машины Java, если же поставляется обычное программное обеспечение, то оно должно прописываться в системной папке.

Определяется тэг этого элемента следующим образом:

```
<SYSTEM
/>
```

Данный элемент является дочерним по отношению к элементам CODE и PACKAGE. Элемент SYSTEM может применяться только в составе элементов MSICD.

Элемент TYPELIB позволяет идентифицировать подключаемую библиотеку для поставляемых классов Java, выполненных по технологии JavaBean. Тэг этого элемента имееет вид:

```
<TYPELIB
    CLASSID = "guid"
/>
```

Файл подключаемой библиотеки имеет расширение tlb. Однако в обязательном атрибуте CLASSID указывается не его имя, а уникальный цифровой шестнадцатеричный идентификатор, соответствующий стандарту GUID.

Данный элемент является дочерним по отношению к элементу CLASS. В качестве примера применения этого элемента можно привести следующий

фрагмент кода, в котором явственно видна типовая структура значения атрибута CLASSID:

```
<TYPELIB CLASSID="{01234567-89AB-CDEF-0123-456789ABCDEF}" />
```

Элемент TYPELIB может применяться только в составе элементов MSICD.

Элемент ABSTRACT включает в себя комментарий, в котором содержится краткое резюме, характеризирующее поставляемое программное обеспечение. Тэг этого элемента определяется следующей конструкцией:

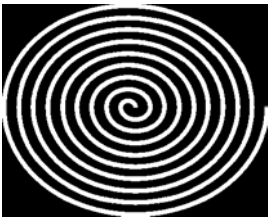
```
<ABSTRACT
/>
```

Данный элемент является дочерним по отношению к элементам IMPLEMENTATION и SOFTPKG. Он может использоваться только в OSD-компонентах.

Элемент TITLE содержит строку, которая будет отображаться в специальном поле заголовка. По сути, он является дополнением к элементу ABSTRACT. Он также является дочерним по отношению к элементам IMPLEMENTATION и SOFTPKG и тоже может использоваться только в OSD-элементах.

Для OSD-компонентов предусмотрены еще элементы DISKSIZE, IMPLTYPE, LICENSE, MEMESIZE и VM, но их спецификации еще не разработаны.

Глава 6



Интернет без проводов

Браузер в сотовом телефоне

Одним из самых модных сейчас направлений развития WWW является сектор предоставления услуг доступа к Web-сайтам с сотовых телефонов. Сотовые телефоны являются высокотехнологичным товаром, который достаточно легко поддается миниатюризации (до некоторого предела, естественно). Одновременно с миниатюризацией изготовители пытаются расширить спектр возможностей этих симпатичных приборчиков. Весь перечень дополнительных возможностей мы можем узнать от любого оператора сотовой связи. Нас будет интересовать лишь та возможность, о которой мы только что упоминали — доступ во всемирную паутину без использования компьютеров.

Ведь что нам действительно необходимо для просмотра Web-сайтов? Связь с провайдером, возможность получать и отправлять информацию по некоторому протоколу, браузер для обработки и отображения полученных Web-страничек, экран для вывода информации и несколько кнопок управления. Связь, экран на жидких кристаллах, управление и даже набор текстовой информации при помощи цифровых клавиш — всем этим сотовые телефоны уже обладали. Оставалось создать только специализированное программное обеспечение для отображения полученной информации. А также разработать специализированный стандарт, пользуясь которым, сотовые телефоны могли бы осуществлять двусторонний обмен информацией с провайдером.

Итак, необходимо было научить сотовые телефоны обращаться с Web-страницами. Причем, желательно было, чтобы все сотовые телефоны от всех производителей делали это одинаково, чтобы не допустить войны стандартов, как это не раз уже было в компьютерной индустрии. То ли игроки на рынке мобильной связи сговорчивее, то ли амбиций у них поменьше, но эти конкуренты все-таки договорились друг с другом и создали специализированный орган, который и занялся вопросами программного обеспечения для доступа в WWW с тонких терминалов, каковыми являются сотовые телефоны. Договоренность была достигнута очень быстро. Орган, получивший имя WAP Forum, был сформирован, выдал свои рекомендации, и они были приняты к исполнению. Ни одна фирма ничего не добавила в созданный стандарт и ничего не убавила из него. В сфере чисто программной индустрии такие случаи очень редки.

Изначально было ясно, что нельзя использовать стандартные подходы для таких тонких терминалов, какими являются сотовые телефоны. Слишком маленькие экраны, слишком простые процессоры, слишком маленькая скорость связи. Требовалось новое решение. Но создавать его надо было на основе уже существующих стандартов, для того, чтобы максимально облегчить миграцию разработчиков на новый стандарт.

Для начала необходимо было изменить протокол доступа к данным и доставки этих данных. Стандартный TCP/IP для этих целей просто не подходил из-за своей "тяжести". Он рассчитан на относительно высокую по сравнению с сотовыми телефонами скорость связи. На данный момент при беспроводном доступе через сотовые телефоны используется скорость 9600 бод. Конечно, когда-то и стационарные модемы работали на меньшей скорости, но это было достаточно давно. Сейчас никто не согласится добровольно вернуться к такой производительности. Нужно было создать протокол, позволяющий поддерживать уверенный обмен данными на сравнительно низкой скорости и минимально загружать трафик своими служебными данными. Такой протокол был создан и получил наименование WAP (Wireless Application Protocol). Из названия видно, что данный протокол предназначен не только для доступа к специализированным Web-страничкам. Он позволяет создавать некие приложения, которые могут выполняться на тонких клиентах с обедненной процессорной базой и поддерживающих беспроводную связь. Правильнее будет сказать, что данные приложения физически выполняются на некоем полноценном сервере, а терминалы-клиенты лишь отображают полученные данные и отсылают команды пользователя.

В качестве интерфейса вполне оправданно были выбраны Web-странички. Но так как для отображения полноценных Web-страниц, написанных на HTML, необходим достаточно мощный и объемный (проверьте, сколько "весит" Internet Explorer со всеми библиотеками, установленный на вашей машине) браузер вместе с полноценным цветным экраном, стало ясным, что нужно также изменять функциональный и отображаемый набор тэгов. Родной HTML никак не мог подойти. Поэтому понадобился еще один язык разметки, который был бы достаточно "легким" для отображения на маленьких монохромных экранчиках сотовых телефонов.

Да, конечно, можно было бы создать новый язык. Можно было бы использовать некоторую усеченную версию HTML. Но оба эти решения, как минимум, неоптимальны. При огромном количестве уже существующих языков добавление еще одного сугубо специализированного языка разметки, пусть даже такого перспективного, как язык создания Web-страниц для сотовых телефонов, все-таки неизящно. А выделение некоего подмножества HTML не может считаться стандартом. Всегда есть возможность, что кто-то захочет "немножко" расширить это подмножество. Конкуренты подхватят. И начнется "вторая браузерная война". Члены WAP Forum прекрасно помнили последствия такого рода действий.

Поэтому было принято очень изящное решение. На основе технологии XML было создано DTD-определение, принятое в качестве корпоративного стандарта. Полученный свод синтаксиса, элементов, переменных и иных компонентов получил название WML (Wireless Markup Language). Да, во многом он является подмножеством HTML. Но создан он на основе XML, в чем можно убедиться, просмотрев официальную спецификацию WML, размещенную в *приложении 3*.

Итак, технология достаточно проста. На основе языка WML создаются специализированные Web-странички, которые затем по протоколу WAP доставляются на сотовый телефон удаленного пользователя и отображаются на экране при помощи встроенного браузера.

Эти облегченные странички часто называют WAP-страницами, что немножко неверно. Написаны они на языке WML. А WAP, как мы помним, всего лишь протокол. Это все равно, что обычные Web-страницы называть TCP/IP-страницами.

Итак, рекомендации WAP Forum были приняты, и рынок доступа к ресурсам WWW с сотовых телефонов начал заполняться потребителями и деньгами. Сейчас даже в России основные Интернет-ресурсы начали "отращивать" себе WAP-сервисы. Стали предоставляться услуги хостинга для WAP-ресурсов. Уже появляются первые домашние странички, написанные на WML.

В принципе, WML очень прост. Писать его странички легко даже в обычном Notepad. А затем просматривать при помощи одного из эмуляторов WML-браузеров. Однако это не совсем удобно. Существуют специализированные WML-редакторы, которые плотно интегрированы с эмуляторами сотовых телефонов. К сожалению, при помощи этих эмуляторов можно только просматривать созданные WML-странички, а звонить пока не удается. Одним из таких редакторов является Dotwap 2.

Но для нас представляет интерес не разбор программ-редакторов. А интересует нас именно стандарт языка WML, который одновременно является предопределенным отраслевым DTD. Его мы и будем рассматривать в этой главе.

Терминология WML

Как мы знаем, Web-сайт состоит из отдельных Web-страничек, одна из которых является стартовой, "домашней". Обычно ее называют FrontPage (да, изначально это слово обозначало именно стартовую страницу, "лицо" сайта, одноименный редактор появился много позже). На стартовой странице обычно располагается оглавление, откуда можно перейти к остальным разделам сайта.

В WML все обстоит приблизительно таким же образом, но терминология там немного иная. Стартовая страница называется *декой* (deck), а присоединяемые странички — *картами* (card). Все остальное осталось без изменений.

В создаваемые странички мы можем включать текст с минимальными возможностями форматирования. Добавлять гиперссылки и устанавливать закладки для локальных гиперссылок. Вставлять графические изображения. При этом необходимо всегда помнить о маленьких размерах и монохромности экранов сотовых телефонов. Хотя, уже появились первые порнографические WAP-ресурсы, следовательно, люди все-таки каким-либо образом мирятся с несовершенством экранов. Мы можем даже создавать формы для обратной связи с удаленным пользователем.

Уже из этого перечисления видно, что из HTML было взято все необходимое для адекватного удовлетворения основных нужд Web-мастеров.

Теперь от общих слов перейдем к рассмотрению конкретных компонентов WML.

Структура WML-страниц

Поскольку любая WML-страница по определению является XML-документом, необходимо в начале файла документа указывать определенные выполняемые инструкции. Проще говоря, в первой строке документа мы обязаны использовать спецификатор XML следующего вида:

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

Это нам уже знакомо и особых вопросов вызывать не должно. Затем мы обычно объявляем корневой элемент документа. В нашем случае им всегда является элемент с наименованием `wml`, определенный в специализированном отраслевом DTD. Таким образом, код WML-страницы всегда выглядит следующим образом:

```
<?xml version="1.0"?>
   <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
          "http://www.wapforum.org/DTD/wml_1.1.xml">
   <wml>
   Текст страницы
   </wml>
```

Анализ второго оператора показывает, что применяемое отраслевое DTD не прошло стандартизацию в ISO. Также мы можем заметить, что на данный момент применяется версия 1.1 стандарта WML.

Основным элементом любой WML-страницы является элемент с наименованием `wml`. Он определяется следующим образом:

```
<!ELEMENT wml ( head?, template?, card+ )>
<!ATTLIST wml
  xml:lang        NMTOKEN        #IMPLIED
  %coreattrs;
  >
```

Мы видим, что содержимое элемента `wml` может состоять из элементов `head`, `template` и `card`. Причем, элементы `head` и `template` могут вообще не использоваться, а элементов `card` может быть несколько. Один, во всяком случае, обязательно. В качестве атрибутов элемента `wml` используется токенизированный атрибут `xml:lang`, указывающий на применяемый язык, и набор атрибутов, объявленный как параметрическая сущность с наименованием `coreattrs`. Эта сущность определяется так:

```
<!ENTITY % coreattrs  "id ID #IMPLIED  class CDATA  #IMPLIED">
```

То есть в качестве основных атрибутов используются токенизированный атрибут-идентификатор `id` и текстовый атрибут `class`.

Элемент `card`, составляющий основу WML-сайта, мы рассмотрим чуть позже. А пока что мы займемся элементом `head`.

Элемент `head` является необязательным неотображаемым заголовком страницы-деки. Определяется он следующим образом:

```
<!ELEMENT head ( access | meta )+>
<!ATTLIST head
  %coreattrs;
  >
```

Как видно, наш заголовок состоит из нескольких элементов типа `access` и `meta`, один из которых просто обязан присутствовать в содержимом элемента `head`. Дочерние субэлементы `access` и `meta`, в свою очередь, определяются так:

```
<!ELEMENT access EMPTY>
<!ATTLIST access
  domain       CDATA       #IMPLIED
  path         CDATA       #IMPLIED
  %coreattrs;
  >
<!ELEMENT meta EMPTY>
<!ATTLIST meta
  http-equiv     CDATA      #IMPLIED
  name           CDATA      #IMPLIED
  forua          %boolean;  #IMPLIED
  content        CDATA      #REQUIRED
  scheme         CDATA      #IMPLIED
  %coreattrs;
  >
```

Из определения видно, что элемент access в своих атрибутах domain и path содержит основное доменное имя и путь к данной страничке соответственно. А элемент meta является, фактически, близнецом одноименного тэга из HTML. Он позволяет устанавливать значения заголовков протокола, которые были позаимствованы из HTTP, и переменные, объявляемые создателем странички. Как мы помним, эти переменные могут регулировать частоту обновления странички и содержать информацию, необходимую для успешного индексирования странички в базах данных поисковых машин.

Элемент head может применяться только к деке. Отдельные странички-карты не могут его иметь в своем составе.

Эти странички являются экземплярами элемента card, который определяется следующим образом:

```
<!ELEMENT card (onevent*, timer?, (do | p)*)>
<!ATTLIST card
  title           %vdata;       #IMPLIED
  newcontext      %boolean;     "false"
  ordered         %boolean;     "true"
  xml:lang        NMTOKEN       #IMPLIED
  %cardev;
  %coreattrs;
  >
```

Из определения вытекает, что этот элемент может включать несколько необязательных субэлементов onevent и один субэлемент timer. Страница может содержать либо некую отображаемую информацию, либо выполняемые инструкции для браузера. Отображаемая информация формируется при помощи элементов p, а задачи на выполнение ставятся при помощи элементов do.

Помимо вложенных субэлементов, рассматриваемый элемент card обладает также рядом атрибутов. Наиболее часто используется атрибут title, значением которого является заголовок данной страницы, и набор атрибутов, объявленный при помощи параметрической сущности %cardev. Сама эта сущность определяется так:

```
<!ENTITY % cardev
 "onenterforward  %HREF;          #IMPLIED
  onenterbackward %HREF;          #IMPLIED
  ontimer         %HREF;          #IMPLIED"
  >
```

Значением атрибута onenterforward является URL, по которому будет произведен переход в том случае, если удаленный пользователь нажмет кнопку "Вперед" на своем сотовом телефоне. Атрибут onenterbackward предназна-

чен для указания адреса страницы, на которую будет осуществляться переход при нажатии кнопки "Назад". А если мы установим на страницу таймер, то по истечении заданного времени мы сможем загрузить иную страничку. Ее URL хранится в атрибуте `ontimer`.

Как мы уже говорили, отображаемое содержимое странички определяется при помощи элементов `p`. Остальные элементы предназначены для выполнения каких-либо действий, и мы рассмотрим их в следующем разделе.

Выполняемые действия

Каждая страничка может содержать описание набора действий, которые браузер должен выполнить при наступлении какого-либо заранее оговоренного события или автоматически при загрузке страницы. Управляют этими действиями несколько ниже рассматриваемых элементов.

Элемент `onevent` содержит описание действий, осуществляемых при наступлении некоторых событий. Его определение имеет вид:

```
<!ELEMENT onevent (%task;)>
<!ATTLIST onevent
  type          CDATA       #REQUIRED
  %coreattrs;
  >
```

Как видно, содержимое данного элемента определено при помощи параметрической сущности с именем `task`. Она позволяет указывать один из предопределенных видов существующих задач-действий. А при помощи обязательного атрибута `type` мы назначаем событие, при наступлении которого должна выполняться установленная задача. Типы применяемых заданий упомянуты в искомой параметрической сущности, которая, в свою очередь, определяется следующим образом:

```
<!ENTITY % task   "go | prev | noop | refresh">
```

То есть при использовании данной сущности в объявлении какого-либо элемента предоставляется возможность выбора из четырех субэлементов, каждый из которых является неким предопределенным действием. Рассмотрим их.

Элемент `go` применяется для принудительного перехода по некоему URL. При этом возможна передача значений полей или переменных. Определяется данный элемент следующими конструкциями:

```
<!ELEMENT go (postfield | setvar)*>
<!ATTLIST go
  href            %HREF;           #REQUIRED
  sendreferer     %boolean;       "false"
```

```
  method          (post|get)      "get"
  accept-charset  CDATA           #IMPLIED
  %coreattrs;
  >
```

В качестве содержимого рассматриваемого элемента используется одно или несколько полей ввода (элемент `postfield`) или системных переменных (элемент `setvar`). Эти системные переменные и способ их действия позаимствованы из стандарта HTTP. А при помощи атрибута `method` выбирается один из двух механизмов передачи данных. Впрочем, можно обойтись и без пересылки какой-либо информации, а просто осуществить переход на WML-страничку. Для этого достаточно не указывать дочерние субэлементы. Главное — задать в значении обязательного атрибута `href` конкретный URL, по которому будет осуществляться переход.

Одним из предустановленных действий WML является безусловный переход на предыдущую страницу. Осуществляется указанная операция при помощи элемента `prev`. В отраслевом DTD он объявлен следующей конструкцией:

```
<!ELEMENT prev (setvar)*>
<!ATTLIST prev
  %coreattrs;
  >
```

Как видно, в данном случае не нужно никаких атрибутов. Браузер сам хранит историю посещения WML-страниц и автоматически осуществляет откат на одну страницу назад, когда встречает в коде карты-странички данный элемент. А при помощи системных переменных мы можем управлять этим шагом назад.

В том случае, если отображаемую WML-страницу необходимо принудительно перезагрузить, мы должны использовать элемент-действие `refresh`. Он определяется так:

```
<!ELEMENT refresh (setvar)*>
<!ATTLIST refresh
  %coreattrs;
  >
```

Его определение практически идентично объявлению только что рассмотренного элемента `prev`.

Элемент `noop` является просто зарезервированной "заглушкой". Он не производит никакого действия. Тем не менее, определение его надо привести:

```
<!ELEMENT noop EMPTY>
<!ATTLIST noop
  %coreattrs;
  >
```

Для каждой страницы-карты мы можем задать такое действие, которое будет выполнено не при возникновении какого-либо события, а по истечении заданного времени. Время ожидания задается посредством объекта `timer`. Для каждой страницы может быть создан только один таймер. Объявляется этот элемент следующим образом:

```
<!ELEMENT timer EMPTY>
<!ATTLIST timer
  name       NMTOKEN          #IMPLIED
  value      %vdata;          #REQUIRED
  %coreattrs;
  >
```

В необязательном токенизированном атрибуте `name` мы указываем имя экземпляра элемента, к которому присоединяется данный таймер. А в обязательном атрибуте `value` записываем планируемое время ожидания в десятых долях секунды.

Оформление текста

Текст, помещаемый на создаваемую страницу, должен быть как-то отформатирован и оформлен. Естественным образом текст разбивается на абзацы. Каждый абзац является содержимым отдельного экземпляра элемента `p`, заимствованного из HTML. В нашем случае он определяется так:

```
<!ELEMENT p (%fields; | do)*>
<!ATTLIST p
  align       %TAlign;     "left"
  mode        %WrapMode;   #IMPLIED
  xml:lang    NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

Сначала нас будет интересовать содержимое данного элемента. Наибольший интерес вызывает субэлемент, определяемый параметрической сущностью `fields`, поскольку элемент `do` представляет собой просто задание для выполнения. Определяется эта параметрическая сущность следующим образом:

```
<!ENTITY % fields  "%flow; | input | select | fieldset">
```

Элементы `input`, `select` и `fieldset` обозначают поля ввода, элементы выбора и формы соответственно. Про текст еще нет ни слова.

Все отображаемые неуправляющие элементы скрываются за параметрической сущностью `flow`. Приведем ее определение:

```
<!ENTITY % flow    "%text; | %layout; | img | anchor | a | table">
```

И здесь становится видно, что в абзацах мы можем размещать текст, элементы разграничения, рисунки, закладки, гиперссылки и таблицы. Этого набора обычно достаточно для создания информативных и красивых страниц.

Само определение текста прячется за еще одной параметрической сущностью `text`. Ее определение выглядит так:

```
<!ENTITY % text     "#PCDATA | %emph;">
```

То есть это может быть либо простой текст, либо отформатированные строки. Мы можем определенным образом варьировать оформление обычного текста. Для этого нам потребуется расшифровать параметрическую сущность `emph`. Это несложно:

```
<!ENTITY % emph    "em | strong | b | i | u | big | small">
```

Как видно, это просто список элементов, являющихся близнецами одноименных тэгов из HTML. Определяются все эти элементы одинаково и чрезвычайно просто. Вот пример определения элемента `i`, создающего курсивный текст:

```
<!ELEMENT i       (%flow;)*>
<!ATTLIST i
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

Из него видно, что помимо стандартного набора атрибутов, этот элемент имеет еще и токенизированный атрибут `xml:lang`, при помощи которого мы можем указывать язык текстового содержимого элемента.

Но самое интересное не это. Как мы видим здесь же, в качестве дочернего субэлемента выступает все та же параметрическая сущность `flow`. Это означает, что мы можем вкладывать одни элементы в другие, комбинируя эффекты, ими оказываемые. Например, при помощи элементов `b` и `i` мы можем задавать полужирное наклонное начертание текста.

Но вернемся к исходному элементу `p`. При помощи атрибута `align` мы можем задавать выравнивание содержимого данного абзаца. Это атрибут перечислимого типа, и все установленные значения скрыты в параметрической сущности `TAlign`. Она определяется следующим образом:

```
<!ENTITY % TAlign "(left|right|center)">
```

Наименования применяемых значений достаточно узнаваемы, и, думается, нет нужды их расшифровывать. По умолчанию для каждого абзаца используется выключка к левому краю окна просмотра, задаваемая значением `left`.

Атрибут `mode` регулирует механизм разбиения строк абзаца. Как известно, окно просмотра может менять свои размеры. Соответственно, у браузера есть два выхода: либо добавить внизу полосу прокрутки, либо переформатировать абзац, изменив длину строк так, чтобы они все целиком помещались

в окно просмотра. Выбор между этими двумя вариантами регулируется значением атрибута mode. Это значение в определении атрибута представлено параметрической сущностью WrapMode. Ее определение имеет вид:

```
<!ENTITY % WrapMode "(wrap|nowrap)" >
```

Значение wrap позволяет браузеру производить переформатирование строк, а значение nowrap оставляет ширину абзацев неизменной, позволяя использовать только полосу горизонтальной прокрутки.

Таблицы

Несомненно, включая таблицы в список объектов форматирования WML, его создатели были оптимистами. Применяемые сейчас экраны слишком малы, поэтому всю мощь таблиц мы не сможем применить в WML-страницах. Но маленькие таблицы мы все-таки задействовать в состоянии. А, значит, нам надо знать, как они создаются.

Идеология таблиц проста. Каждая таблица определяется одним элементом table. Тот в качестве вложенных субэлементов содержит строки. А каждая строка, в свою очередь, содержит несколько элементов, реализующих отдельные ячейки. Вся эта иерархия достаточно хорошо описывается стандартными определениями. Начнем мы с основного элемента table. В стандарте WML он объявлен так:

```
<!ELEMENT table (tr)+>
<!ATTLIST table
  title       %vdata;    #IMPLIED
  align       CDATA      #IMPLIED
  columns     %number;   #REQUIRED
  xml:lang    NMTOKEN    #IMPLIED
  %coreattrs;
  >
```

Как видно, таблица просто обязана иметь хотя бы одну строку, содержащуюся во вложенном субэлементе tr. Также мы должны задать количество столбцов таблицы в значении обязательного атрибута columns. Заголовок таблицы, ее название задается при помощи атрибута title.

Сами строки таблицы особого интереса не представляют. Они являются всего лишь контейнерами для ячеек. Это, впрочем, хорошо видно в следующем определении элементов-строк tr:

```
<!ELEMENT tr (td)+>
<!ATTLIST tr
  %coreattrs;
  >
```

А сами ячейки определяются так:

```
<!ELEMENT td ( %text; | %layout; | img | anchor | a )*>
<!ATTLIST td
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

Отсюда следует, что мы можем использовать в ячейках таблиц текст, элементы раскладок, графические изображения, гиперссылки и закладки к гиперссылкам. А так как все эти элементы имеют свои свойства форматирования, мы можем получать достаточно сложные табличные образования. Только не стоит забывать, что отображаться все это гипотетическое великолепие будет на маленьком экранчике сотового телефона.

В качестве примера определения таблицы в коде WML-страницы можно привести следующий фрагмент:

```
<table column="2">
<tr><td>1</td><td>2</td></tr>
<tr><td>3</td><td>4</td></tr>
</table>
```

Этот блок кода создает таблицу из двух строк и двух столбцов, заполненную по порядку цифрами от единицы до четырех.

Графика и гиперссылки

В HTML гиперссылки могли указывать как на весь документ, так и на определенное его место. Правда, для того чтобы сделать ссылку на некоторое место в документе, конкретный абзац или рисунок, необходимо было поставить там закладку. При переходе по подобной гиперссылке документ все-таки загружался в браузер полностью, но фрагмент с закладкой, на который и указывала гиперссылка, позиционировался у верхней границы окна просмотра.

WML поддерживает обе эти возможности. Сначала мы разберемся с созданием гиперссылок, а затем перейдем к установке закладок.

Гиперссылки в WML-страницах задаются при помощи элемента с наименованием `a`, как и в HTML. Объявляется этот элемент следующим образом:

```
<!ELEMENT a ( #PCDATA | br | img )*>
<!ATTLIST a
  href          %HREF;        #REQUIRED
  title         %vdata;       #IMPLIED
  xml:lang      NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

Как видно, сам объект ссылки, которым может являться текст, разбитый на несколько строк, или рисунок, заключается между тэгом `<a>` и его закрывающей парой `</a>`. При этом служебная информация гиперссылки указывается при помощи атрибутов.

Обязательный атрибут `href` в качестве своего значения содержит URL, по которому будет произведен переход в момент активации гиперссылки. А атрибут `title` содержит текст подсказки-хинта, которая должна расшифровывать предназначение ссылки.

Установка закладки, или, как ее еще часто называют — *якоря*, в документе осуществляется при помощи элемента `anchor`, являющегося функциональным двойником одноименного тэга из HTML. Определяется он так:

```
<!ELEMENT anchor ( #PCDATA | br | img | go | prev | refresh )*>
<!ATTLIST anchor
  title        %vdata;      #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

Из определения видно, что закладку можно установить не только на текст или рисунок, но и на одно из перечисленных выполняемых действий. Атрибуты этого элемента совпадают с атрибутами только что рассмотренного элемента `a`, поэтому мы не будем разбирать их еще раз.

Мы говорили, что гиперссылка или закладка могут иметь графическое содержимое. Но мы до сих пор не знаем, как включать рисунки в текст WML-страницы. Это делается при помощи элемента `img`. Его определение имеет вид:

```
<!ELEMENT img EMPTY>
<!ATTLIST img
  alt          %vdata;      #REQUIRED
  src          %HREF;       #REQUIRED
  localsrc     %vdata;      #IMPLIED
  vspace       %length;     "0"
  hspace       %length;     "0"
  align        %IAlign;     "bottom"
  height       %length;     #IMPLIED
  width        %length;     #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

Сам элемент не позволяет включать в себя какое-либо содержимое, он пустой. Вся информация указывается при помощи атрибутов.

Обязательный атрибут `src` в качестве значения содержит URL включаемого графического файла. Если графика не предназначена для отображения браузером мобильного телефона удаленного клиента, то на месте рисунка должна быть показана некая надпись. Ее текст задается при помощи значения обязательного атрибута `alt`.

Ширина и высота изображения устанавливаются браузером, исходя из самого графического файла. Но если необходимо явно ограничить их, следует использовать необязательные атрибуты `width` и `height` соответственно.

Атрибуты `vspace` и `hspace` задают вертикальный и горизонтальный отступы рисунка от окружающего его текста в пикселах. По умолчанию используются нулевые значения этих атрибутов.

Атрибут `align` позволяет явно указывать вертикальное выравнивание рисунка. В качестве значения этого атрибута может использоваться одно из трех предустановленных ключевых слов. Все они в объявлении скрыты за параметрической сущностью `IAlign`. Последняя объявляется так:

```
<!ENTITY % IAlign "(top|middle|bottom)" >
```

Значение `top` устанавливает верхнее выравнивание, значение `bottom` — нижнее, а `center`, естественно, центрирует изображение по вертикали. По умолчанию используется значение `bottom`.

Как мы видим, WML очень похож на HTML. В нем присутствуют даже способы ввода данных пользователем и передачи их на WAP-сервер. Их мы и рассмотрим в следующем разделе.

Органы ввода данных

В HTML все органы ввода данных объединялись в так называемые формы. При этом в начальном тэге формы сразу указывалась ссылка, по которой должны были передаваться введенные удаленным пользователем данные, и метод, которым их надлежало передавать. В WML ссылка и метод указываются в выполняемых заданиях, а органы ввода данных объединяются не в формы, а в наборы.

Набор органов ввода данных удаленным пользователем является элементом с наименованием `fieldset`. Объявляется он при помощи следующей конструкции:

```
<!ELEMENT fieldset (%fields; | do)* >
<!ATTLIST fieldset
  title          %vdata;      #IMPLIED
```

```
  xml:lang        NMTOKEN      #IMPLIED
  %coreattrs;
  >
```

Главное, на что следует обратить внимание, это — параметрическая сущность `fields`, определяющая набор отдельных полей ввода. Есть еще несколько необязательных атрибутов, но они особого интереса не представляют. Посему возвращаемся к параметрической сущности `fields`. Она определяется так:

```
<!ENTITY % fields  "%flow; | input | select | fieldset">
```

То есть в качестве отдельных полей могут выступать объекты форматирования (текст, рисунки и прочее), такие же вложенные наборы полей и два еще пока незнакомых нам элемента `input` и `select`.

Элемент `input` предназначен для создания текстовых полей ввода. То есть предназначенных для того, чтобы пользователь самостоятельно вводил в них текст. Цифровая клавиатура сотового телефона, в принципе, позволяет делать это, но пока что процесс лишен хотя бы минимального удобства. Однако возможность таковая предусмотрена, следовательно, нам необходимо ее рассмотреть.

Элемент `select` предназначен для создания неких групп предустановленных значений, из которых пользователь делает выбор. Этот вариант, естественно, несколько удобнее.

Начнем мы с элемента `input`. Его определение имеет вид:

```
<!ELEMENT input EMPTY>
<!ATTLIST input
  name       NMTOKEN           #REQUIRED
  type       (text|password)   "text"
  value      %vdata;           #IMPLIED
  format     CDATA             #IMPLIED
  emptyok    %boolean;         "false"
  size       %number;          #IMPLIED
  maxlength  %number;          #IMPLIED
  tabindex   %number;          #IMPLIED
  title      %vdata;           #IMPLIED
  xml:lang   NMTOKEN           #IMPLIED
  %coreattrs;
  >
```

Сам элемент является пустым, он не нуждается в каком-либо дополнительном содержимом. Вся информация задается при помощи атрибутов.

Обязательный токенизированный атрибут `name` назначает уникальное имя создаваемого поля ввода, которое будет передаваться обрабатывающему приложению вместе с данными.

Атрибут `type` позволяет устанавливать тип создаваемого текстового поля ввода. Предусмотрено два типа: обычный текстовый (`text`) и секретный, предназначенный для ввода конфиденциальной информации, такой как пароль (`password`). В первом случае введенный текст отображается как есть, а во втором все символы заменяются на какой-нибудь один символ, например, звездочку. Эта возможность является оправданной для обычных настольных систем, где всегда есть опасность, что кто-то подсмотрит ваш пароль из-за спины, но в сотовых телефонах она не кажется столь необходимой. По умолчанию мы всегда создаем обычное текстовое поле, так как для атрибута `type` "умолчальным" значением является `text`.

Также мы имеем возможность устанавливать начальное значение в текстовое поле, которое будет отображаться еще до ввода пользователем своего текста. Подобным образом мы можем задавать типовые ответы, и пользователю может не понадобиться вводить свой текст, достаточно будет сразу отослать данные обрабатывающему приложению. "Умолчальный" текст является значением атрибута `value`. В этот же атрибут потом заносится введенная пользователем информация.

Если нам необходимо принудить пользователя обязательно ввести какой-либо текст в поле ввода, то мы можем использовать атрибут `emptyok`. Посредством его логического значения мы можем указывать, допустимо ли оставлять в данном поле пустое текстовое значение. По умолчанию у данного атрибута установлено значение `false`, то есть мы требуем обязательного ввода какого-то текста.

При помощи значения атрибута `maxlength` нам позволено устанавливать максимально возможную длину вводимого текста. А длина самого поля ввода в пикселах или процентном соотношении указывается в значении атрибута `size`.

Перемещение между полями ввода информации осуществляется по порядку их индексных значений. Для каждого экземпляра элемента `input` его порядковый индекс задается в атрибуте `tabindex`.

Теперь мы разберемся с элементом `select`. Он позволяет создавать списки альтернатив, среди которых пользователь выбирает одну или несколько. По сути, это аналог групп зависимых и независимых переключателей (радиоклавиши и "чекбоксы"). Объявляется этот элемент следующим образом:

```
<!ELEMENT select (optgroup|option)+>
<!ATTLIST select
  title        %vdata;          #IMPLIED
  name         NMTOKEN          #IMPLIED
  value        %vdata;          #IMPLIED
  iname        NMTOKEN          #IMPLIED
```

```
  ivalue       %vdata;          #IMPLIED
  multiple     %boolean;        "false"
  tabindex     %number;         #IMPLIED
  xml:lang     NMTOKEN          #IMPLIED
  %coreattrs;
  >
```

В качестве содержимого этого элемента мы можем использовать одну или несколько альтернатив выбора. Тип переключателей (зависимые или независимые) определяется логическим значением атрибута `multiple`. Если мы используем значение по умолчанию `false`, то создаем группу с зависимыми переключателями, из которых пользователь может выбрать только один. Если же мы используем значение `true`, то пользователь сможет установить столько переключателей, сколько захочет.

Группа переключателей, входящая в состав данного набора, визуализируется при помощи элемента `optgroup`. Он оформляется в соответствии со следующим объявлением:

```
<!ELEMENT optgroup (optgroup|option)+ >
<!ATTLIST optgroup
  title      %vdata;    #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >
```

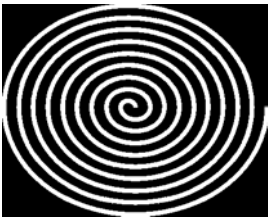
Сам по себе этот элемент неинтересен. Главное в нем то, что он служит контейнером для элементов `option`, реализующих отдельные переключатели. Эти элементы объявляются так:

```
<!ELEMENT option (#PCDATA | onevent)*>
<!ATTLIST option
  value      %vdata;    #IMPLIED
  title      %vdata;    #IMPLIED
  onpick     %HREF;     #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >
```

В качестве содержимого этого элемента используется обычный текст, который и является строкой, связанной с переключателем. А при помощи необязательных атрибутов мы задаем дополнительную информацию.

Итак, мы рассмотрели все основные возможности языка WML, который является одним из фактически действующих стандартов, построенных на основе технологии XML. И на этом мы закончим обзор приложений XML. Мы идем дальше.

Глава 7



Стилевые таблицы CSS

Стилевые таблицы

Все, что мы обсуждали в предыдущих главах, относится к созданию XML-документов, но не к их оформлению. Мы уже знаем, как создавать XML-документы. Мы можем программировать логику их поведения, задавать структурную иерархию элементов и прочее. Все упомянутые возможности предназначены, по большей части, для обработки специализированными XML-приложениями. Но практически всегда документы в Интернете необходимо кроме обработки еще и отображать. В HTML были встроены средства управления внешним видом HTML-документа, такие, как размер шрифта, цвет, выравнивание и другие параметры. В XML таких встроенных механизмов просто не может быть. Это — негативная сторона полной свободы и расширяемости.

Но и встроенных средств HTML многим не хватало. Поэтому был создан дополнительный механизм единого стилевого оформления Web-страниц, входящих в состав сайта. Его назвали CSS (Cascading Style Sheet). В рамках этой технологии создается отдельный внешний файл, в котором собираются правила отображения того или иного тэга браузером. Затем CSS-файл подключается к каждой Web-странице. Таким образом мы можем легко достигнуть единообразия стилевого оформления каждой Web-страницы, входящей в состав сайта.

Эта технология очень хорошо зарекомендовала себя в HTML и практически без изменений перешла по наследству к XML. На данный момент действует уже вторая версия стандарта CSS, и готовится к выходу третья. Поэтому в этой главе мы будем рассматривать технологию CSS level 2 и правила использования ее в комплексе с XML-документами.

Первая версия стандарта CSS содержала правила отображения текста Web-страниц, размещения текстового содержимого на странице и некоторые детали, уточняющие отображение страниц и ячеек таблиц. Вторая версия добавила несколько новых возможностей, таких как визуальные эффекты или автоматическая нумерация элементов. Все вышеперечисленное мы тщательно рассмотрим.

Но сначала нам необходимо разобраться с механизмом создания и использования CSS. *Стилевая таблица* представляет собой текстовый файл специа-

лизированного формата. В спецификациях данный формат имеет обозначение `text/css`. В CSS-файле записываются правила отображения для некоторого набора выбранных нами тэгов. Пример записи одного такого правила может выглядеть приблизительно следующим образом:

```
a {color : navy; font-family : Arial}
```

Это правило указывает на то, что текстовое содержимое всех тэгов `<a>`, которыми в HTML обозначаются гиперссылки, будет отображаться при помощи любого доступного шрифта из семейства Arial, имеющего темно-синий цвет (navy).

Весь CSS-файл состоит из таких определений. Правила синтаксиса стилевых таблиц мы рассмотрим в следующем разделе.

Синтаксис CSS

Все объявления стандарта CSS состоят из двух частей. В первой части записывается наименование тэга, к содержимому которого и будет применено данное правило оформления. Эта часть в спецификации CSS носит название `selector`. Иногда этот термин калькой переносят в русскоязычную техническую литературу. Поэтому в иных описаниях технологии CSS нередко можно увидеть, что наименования тэгов в стилевых таблицах называют селекторами.

После наименования тэга в фигурных скобках записывается правило оформления текстового содержимого. Посмотрим еще раз на наш пример:

```
a {color : navy; font-family : Arial}
```

Как видно, в фигурных скобках может записываться сразу несколько правил, разделенных символом точки с запятой. Само правило также состоит из двух частей. Сначала записывается наименование параметра отображения, например, вид применяемого шрифта, а затем, после двоеточия, мы пишем значение этого параметра. В примере все это достаточно хорошо видно.

Итак, как мы знаем, что в фигурных скобках можно записать сразу несколько правил отображения содержимого. Но иногда требуется обратное — сразу для нескольких тэгов нужно задать одинаковые правила отображения. В этом случае мы можем не выписывать одинаковые конструкции для каждого тэга отдельно, а объединить тэги в одну группу, и задать правила отображения для всех тэгов сразу. При этом объединяемые наименования тэгов разделяются запятой. Это можно увидеть на следующем примере:

```
a, h1 { color : navy }
```

Данное правило указывает, что текстовое содержимое гиперссылок и заголовков первого уровня будет отображаться темно-синим цветом.

Синтаксис CSS определяет также и правила оформления комментариев. Комментарии в CSS оформляются в стиле языка C. Начинается комментарий с символа наклонной черты и звездочки (`/*`), а завершается обратным сочетанием (`*/`). Следующий пример представляет собой комментарий CSS:

```
/*Это комментарий*/
```

Итак, мы рассмотрели принципы группировки как наименований тэгов, так и правил отображения. Мы знаем, как использовать синтаксис CSS для создания каскадных таблиц стилей. Теперь мы можем перейти к рассмотрению правил иерархии.

Порядок использования правил

Полное наименование CSS — *каскадные таблицы стилей*. Почему они называются таблицами стилей, мы уже поняли. Возникает вполне обоснованный вопрос, при чем тут каскадность, и что это вообще обозначает.

В компьютерной индустрии очень велик темп прогресса. Следствием этого является множество самых различных тенденций и течений. Часть из них вскоре после рождения отмирает, а часть переходит из разряда тенденций в ранг идеологий. Одной из таких "правильных" тенденций является повторное использование кода и наследование ранее созданных решений. К CSS это имеет самое прямое отношение.

Нам никто не мешает для каждого документа использовать только одну стилевую таблицу. Мы можем подключать их сразу несколько. Например, при добавлении новых документов в состав сайта может возникнуть потребность добавить новые правила отображения или создать новые стили. Иногда это делается коррекцией уже существующей стилевой таблицы, но никто не запрещает нам создать дополнительный файл и подключить его к документу. Как мы только что выяснили, мы можем подключать сразу несколько таблиц.

Иногда может возникнуть ситуация, когда в нескольких CSS-файлах содержатся правила отображения для одного и того же тэга, или, в случае использования XML, одного и того же элемента. И правила эти могут не совпадать. В этом случае браузером используется самое последнее объявление правила отображения.

Рассмотрим эту концепцию на примере. Предположим, у нас есть два файла стилевых таблиц. В файле style1.css мы разместили объявления правил отображения для двух элементов XML-документа с наименованиями `list` и `item`. В результате получили следующий код:

```
list {font-family: Times New Roman}
item {color: blue; font-family: Arial}
```

А в дополнительном файле style2.css мы объявляем правила отображения для элементов `item` и `part` следующим образом:

```
item {color: green; font-family: Arial}

part {color: black}
```

Теперь, если мы подключим к XML-документу эти два стилевых файла, окажется, что для элемента `item` задано два различных правила отображения. В первом файле style1.css указано, что текст элемента `item` будет отображаться синим цветом, а содержащееся во втором файле правило предписывает применение для отображения содержимого этого же элемента уже зеленого цвета. Какой цвет мы увидим на самом деле, зависит от того, в каком порядке к XML-документу будут подключаться файлы стилевых таблиц.

Используется всегда последнее правило. То есть если сначала мы присоединим файл style1.css, а потом файл style2.css, то текстовое содержимое элемента `item` будет отображаться зеленым цветом.

Но мы можем устанавливать специальный модификатор для правил отображения, который будет заставлять браузер игнорировать порядок подключения стилевых таблиц. Так, если мы приписываем к какому-либо правилу оформления модификатор `important`, то для отображения текстового содержимого элемента будет использоваться именно это правило, вне зависимости от того, были ли еще подключены правила для отображения этого же элемента. То есть использование ключевого слова `important` для какого-либо правила позволяет установить для него привилегированный уровень и убрать его из иерархии каскадных таблиц стилей. Рассмотрим на примере. Возьмем знакомый нам файл style1.css и немного модифицируем его код. Получаем следующую совокупность правил отображения:

```
list {font-family: Times New Roman}

item {color: blue ! important; font-family: Arial}
```

Теперь, при том порядке подключения стилевых файлов style1.css и style2.css, который мы рассматривали ранее, текстовое наполнение элемента `item` будет отображаться синим цветом.

Из примера виден порядок использования модификатора `important`. Если нам необходимо какое-либо правило сделать привилегированным, мы после него ставим восклицательный знак, а далее через пробел записываем ключевое слово `important`.

Впрочем, вполне возможна ситуация, когда в нескольких CSS-файлах заданы правила отображения для одного и того же элемента, и сразу несколько из них имеют модификатор `important`. В этом случае формируется дополнительная иерархия из этих привилегированных правил, и опять-таки используется для отображения самое последнее подключенное привилегированное правило.

Использование CSS в XML-документах

Для того чтобы стилевые таблицы могли действовать в XML-браузере, их необходимо подключить к XML-документу. Для этого обычно применяется конструкция следующего вида:

```
<?xml:stylesheet htef="style1.css" type="text/css"?>
```

Как видно, это выполняемая инструкция, адресованная XML-процессору. В качестве значения параметра `href` мы указываем URL подключаемого CSS-файла. Обратите внимание, здесь мы говорим об URL, а не об URI, так как мы подключаем целый файл. URI же применяются только для адресации субресурсов, которые являются частями файлов или документов.

А в параметре `type` мы указываем тип подключаемого файла. Как мы уже отмечали ранее, этот тип обозначается как `text/css`.

После того, как мы подключили стилевую таблицу, каждый раз, когда в XML-документе будет встречаться один из элементов, для которого задано правило отображения в данной стилевой таблице, XML-браузер распознает заданное правило и использует его.

Но есть и еще некоторые дополнительные возможности. Иногда может возникнуть необходимость создать несколько различных правил отображения для одного и того же элемента. В HTML, например, различные правила отображения часто употребляются для активных и пройденных посетителем ссылок. То же самое можно применить и для элементов XML. Для этого используются так называемые *классы CSS*.

В качестве основного имени класса мы используем наименование элемента, определенного в DTD-блоке XML-документа. Через точку указывается дополнительное идентифицирующее наименование класса. Увидеть это можно на следующем примере:

```
link.inner {color: red}
link.outer {color: green}
```

В данной стилевой таблице мы объявляем, что текстовое содержание элементов-гиперссылок будет отображаться красным цветом для локальных ссылок, и зеленым — для внешних.

Таким образом, мы получаем два равноценных правила отображения для одного и того же элемента. Но мы должны выборочно использовать либо одно, либо другое правило, определяя их применимость для каждого конкретного экземпляра класса отдельно. В HTML это решалось достаточно просто. Там к тэгу добавлялся предопределенный параметр `CLASS`, в качестве значения которого использовалось дополнительное идентифицирующее наименование класса. В XML подобного предустановленного атрибута нет.

Следовательно, при проектировании DTD-блока мы должны добавить атрибут CLASS в объявление тех элементов, для которых хотим предусмотреть множественные правила оформления.

То есть если следовать этому совету, в DTD-блоке XML-документа мы должны были элементу link приписать атрибут CLASS. После чего в содержимое XML-документа мы вправе включить следующий блок кода:

```
<link CLASS="inner">local link</link>
<link CLASS="outer">external link</link>
```

Тогда словосочетание "local link" будет отображаться красным цветом, а "external link" — зеленым.

Есть и еще один вариант создания множественных правил оформления для различных экземпляров одного и того же элемента. Он применяется для отображения текстового содержимого элементов, которые являются внутренними частями других элементов.

В порядке вещей, что один и тот же элемент может служить составной частью сразу для нескольких более общих элементов. Рассмотрим соответствующий фрагмент DTD-блока:

```
<!ELEMENT version (description, number)>
<!ELEMENT frontpage (url,description)>
<!ELEMENT description (#PCDATA)>
```

А теперь приведем фрагмент кода стилевой таблицы:

```
version description {font-family: Arial}
```

Это правило оформления будет использоваться для экземпляров элемента description, которые вложены в качестве содержимого в экземпляр элемента version. Оно будет срабатывать только в том случае, если в XML-документе встречается следующий фрагмент разметки:

```
<version>
<description>Final release</description>
. . . . . .
</version>
```

В этом, и только в этом случае для отображения текстового содержимого вложенного экземпляра элемента description будет использован шрифт семейства Arial.

В терминологии CSS названия элементов, к которым применяются правила оформления, называются *селекторами* (selector). Предусмотрена возможность в качестве селектора использовать идентификатор экземпляра элемента, то есть значение атрибута типа ID. При этом совершенно неважно, какой тип элемента будет у данного экземпляра. Так как все идентификаторы уникальны, то правило отображения, в котором используется в качестве селек-

тора значение атрибута ID, будет использовано всего один раз для конкретного экземпляра какого-либо элемента. Рассмотрим пример. В стилевой таблице мы можем разместить следующий фрагмент кода:

```
#z98y { letter-spacing: 0.3em }
```

Для того чтобы это правило сработало, в XML-документе должна встретиться приблизительно следующая конструкция:

```
<P ID=z98y>Wide text</P>
```

Как видно из примера, в CSS перед значением идентификатора обязательно вставляется знак решетки (#), который и указывает, что следующее за ним выражение будет расцениваться как селектор-идентификатор.

Вообще, о селекторах необходимо поговорить особо. Существует несколько типов селекторов и, соответственно, несколько вариантов их обозначения. Мы пока что рассмотрели далеко не все варианты. Первый, уже рассмотренный нами вариант позволяет создавать селекторы при помощи наименования класса, указывая тип элемента. Их так и называют — *селекторы типа* (type selector). Рассмотрим все остальные возможные варианты создания различных селекторов.

Универсальный селектор (Universal selector). Позволяет устанавливать правило отображения сразу для всех элементов. В качестве универсального селектора используется символ звездочки (*).

Вложенный селектор (Descendant selector). Применяется для установки правил отображения вложенных элементов. Подобный селектор мы уже рассматривали.

Дочерний селектор (Child selector). Определяет правила отображения для дочерних элементов. Стандартный синтаксис: E > F. То есть наименования родительского (E) и дочернего (F) элементов разделяются знаком "больше" (брекетом).

Существует селектор, позволяющий устанавливать правило отображения для первого дочернего элемента. Так как в XML нет механизма указания именно на первый дочерний элемент, для этой цели в CSS используется так называемый псевдокласс. *Псевдоклассы* являются механизмом CSS, который позволяет указывать на элементы, которые нельзя идентифицировать стандартным образом, как в данном случае.

Итак, селектор, указывающий на первый дочерний элемент, специфицируется следующей конструкцией:

```
E:first-child
```

То есть если мы хотим указать, что для отображения текстового содержимого первого дочернего элемента по отношению к элементу paragraph будет использоваться шрифт Arial, мы должны применить следующую конструкцию:

```
paragraph:first-child {font-family: Arial}
```

На основании все тех же псевдоклассов создаются и селекторы, позволяющие задавать правила отображения, основанные на действиях пользователя. К примеру, если мы имеем элемент `text`, то мы можем создать отдельные правила отображения для его текстового содержимого, применяемые при прохождении курсора мыши над текстом, получении фокуса ввода или обычной активизации. Для этого используются селекторы `text:hover`, `text:focus` и `text:active` соответственно. Подобные псевдоклассы называют динамическими.

Те же псевдоклассы, но уже не динамические, а обыкновенные, статические, применяются для обработки элементов, реализующих гиперссылки. Мы можем задавать отдельные правила отображения для пройденных и еще не пройденных пользователем гиперссылок. Для этого используются селекторы `e:link` и `e:visited`, где `e` — наименование элемента-гиперссылки.

Существуют четыре вида *атрибутивных селекторов* (attribute selector), которые позволяют создавать правила отображения для элементов в зависимости от набора атрибутов, приписанных им. Рассмотрим правила создания атрибутивных селекторов.

Для начала мы создадим правило отображения для любого экземпляра выбранного элемента, который имеет необходимый набор атрибутов. При этом само содержание данных атрибутов не имеет никакого значения. Данное правило выглядит следующим образом:

```
paragraph[indent] {font-family: Arial}
```

В этом примере мы накладываем правило применения шрифта на те экземпляры элемента `paragraph`, которые имеют атрибут `indent`.

Помимо этого мы можем задавать ограничение не только по наличию атрибута, но и по его значению. Например, следующая конструкция позволяет применять правило оформления только для тех экземпляров элемента `paragraph`, которые имеют атрибут `indent`, причем значение последнего равно `"right"`.

```
paragraph:[indent="right"] {font-family: Arial}
```

Для установки значения атрибута мы можем использовать текстовую строку, в которой значения разделяются пробелами. CSS позволяет создавать атрибутивные селекторы, которые указывают на экземпляры элементов, имеющие в списке значений атрибута некое предустановленное слово. Поясним на примере:

```
paragraph:[indent="right"] {font-family: Arial}
```

Это правило будет применено только для тех экземпляров элемента `paragraph`, у которых значение атрибута `indent` будет содержать слово `right`. Необходимо отметить, что слово `right` не является произвольной подстрокой, оно должно быть отделено пробелами от остальных возможных значений.

Последний, четвертый вид атрибутивных селекторов применяется следующим образом:

```
E[lang|="en"]
```

Этот селектор позволяет задавать правило отображения для тех экземпляров элемента с наименованием `E`, у которого атрибут `lang` содержит множество значений, разделенных вертикальной чертой (отношение "ИЛИ"), среди которых есть значение `en`. Иначе говоря, данный экземпляр содержит текстовое наполнение на английском языке.

Теперь, когда мы знаем синтаксис спецификации, правила создания и использования CSS-файлов, необходимо рассмотреть все ключевые слова CSS. Тогда мы уже сможем создавать свои стилевые таблицы.

Единицы измерения в CSS

Почти каждый элемент генерируемого в окне браузера изображения обладает рядом свойств, значения которых было бы желательно указывать в привычном нам виде, то есть совместно с единицами измерения. Подобным образом мы отсчитываем расстояния на экране для многих параметров отображения. Например, мы можем задать величину левого или правого поля отступа. Для таких случаев в CSS и предусмотрены самые различные единицы измерения. Их мы и рассмотрим в этой части.

В спецификации CSS рассматриваются три вида единиц измерения (units). Тривиальные единицы измерения длины (или размеров), единицы процентного соотношения и цветовые единицы. Да-да, цветовые обозначения тоже относятся к единицам измерения. Но, все по порядку. Начинаем с единиц длины.

Единицы длины, в свою очередь, можно разделить на абсолютные и относительные. Мы увидим, в чем их различие.

- Сантиметр. Обозначается как `cm`. Является абсолютной единицей измерения.
- Миллиметр. Стандартное обозначение — `mm`. Абсолютная единица.
- Дюйм. Обозначение `in` является сокращением от его англоязычного названия (inch). Абсолютная единица.
- Точка. Чаще называется пунктом. Применяемое обозначение — `pt` (1 pt = = 1/72 in).
- Пика. Иногда пишется как пайка. Имеет обозначение `pc`. Ранее применялась в типографской мере длин. Равна 4,23 миллиметра (12 pt). Абсолютная единица.
- Ширина строчной буквы m применяемого шрифта. Обозначается как `em`. Традиционно буква m считается самым широким символом любого

шрифта. Является относительной единицей измерения, так как зависит от применяемого шрифта.

- Высота строчной буквы "x" применяемого шрифта. Стандартное обозначение — `ex`. Относительная единица измерения.
- Пиксел. Обозначается как `px`. Несмотря на то, что является абсолютной единицей, не имеет четкой и единой протяженности, поскольку зависит от применяемого монитора и установленного графического разрешения.

Помимо уже рассмотренных нами единиц длины, которые позволяют более или менее точно указать необходимую длину, мы можем использовать процентное соотношение. Например, в случае, если необходимо указать, что величина левого поля равна одному проценту от ширины окна просмотра применяемого браузера. Процентное соотношение всегда отсчитывается от базового размера элемента, в зависимости от контекста. Горизонтальные величины отсчитываются от ширины, вертикальные — от высоты. При указании процентного соотношения вместо наименования единицы измерения просто ставится знак процента "%".

Приведем пример. Создадим два правила отображения. Зададим размер левого поля для типовых текстовых элементов. В первом правиле используем абсолютную единицу измерения, во втором — процентное соотношение. В результате получим следующее:

```
text {margin-left: 12 mm}

chapter {margin-left: 2%}
```

Теперь нам осталось рассмотреть цветовые обозначения.

Для стандартных шестнадцати цветов есть зарезервированные словесные обозначения, которые приведены в табл. 7.1.

Таблица 7.1. *Стандартные цветовые определения*

Обозначение	Цвет
aqua	Морская волна
black	Черный
blue	Синий
fuchsia	Малиновый
gray	Серый
green	Зеленый
lime	Ярко-зеленый
maroon	Темно-красный
navy	Темно-синий
olive	Оливковый

Таблица 7.1 *(окончание)*

Обозначение	Цвет
purple	Пурпурный
red	Красный
silver	Серебряный
teal	Темная морская волна
white	Белый
yellow	Желтый

То есть если мы хотим указать в стилевом файле, что для элемента `text` мы собираемся использовать белый цвет фона, мы должны использовать следующую конструкцию:

```
text {background-color: white}
```

Если же стандартных шестнадцати цветов не хватает, или просто нет желания использовать всем уже надоевшие краски, как это обычно и бывает, можно указать практически любой оттенок при помощи его RGB-кода.

Как мы знаем из курса элементарной физики, любой цвет можно составить из трех базовых цветов. В компьютерной индустрии принят стандарт RGB-представления цвета. Согласно ему любой цвет представляется в виде комбинации красного, зеленого и синего цвета различной насыщенности.

В CSS предусмотрены четыре варианта задания RGB-кода требуемого цвета. Рассмотрим их на примерах. Правило, реализующее первый вариант, выглядит следующим образом:

```
EM { color: #f00 }
```

Это — усеченный вариант задания цвета. После знака решетки записываются три шестнадцатеричные цифры. Каждая из них отвечает за насыщенность соответствующего цвета. Первая из них указывает интенсивность красного компонента (Red), вторая — зеленого (Green), третья — синего (Blue). Так как мы используем единичные цифры для указания значения, каждый компонент может иметь всего лишь шестнадцать градаций. То есть на каждый цвет отводится по четыре бита. Поскольку основных цветов всего три, нетрудно подсчитать, что подобным образом мы можем задать всего лишь 4096 цветов.

Естественно, взыскательного пользователя (а кто из нас не считает себя таковым?) подобное количество цветов удовлетворить не может. Поэтому был добавлен вариант задания цвета, предусматривающий для каждого базового цвета уже 256 градаций яркости. Этот вариант показан на следующем примере:

```
EM { color: #ff0000 }
```

Как видно, мы можем использовать двузначное шестнадцатеричное число для установки степени яркости базового цвета. Таким образом мы можем определить уже 16 777 216 различных цветов, что соответствует стандарту TrueColor. Не думаю, что для Интернет-приложений кому-то может понадобиться большее количество цветов.

Если же использование шестнадцатеричных чисел еще не вошло в привычку, можно задавать насыщенность базового цвета обычным десятичным числом. Естественно, все десятичные значения обязаны быть неотрицательными, целочисленными и не превышать число 255. Правила применения легко увидеть на следующем примере:

```
EM { color: rgb(255,0,0) }
```

Отсюда видно, что если мы не используем знак решетки, который указывает на наличие шестнадцатеричных чисел, мы должны писать оператор `rgb`, после которого в скобках указываются значения насыщенности базовых цветов, разделенных запятыми.

А четвертый вариант задания RGB-кода цвета предусматривает использование процентного соотношения насыщенности. Демонстрацией такого подхода служит пример:

```
EM { color: rgb(100%, 0%, 0%) }
```

Легко заметить, что индикатором, указывающим обработчику стилевых таблиц на то, что в данном случае применяется процентное соотношение, а не десятичные значения, служит знак процента. Что неудивительно.

Данный способ позволяет использовать значения с десятыми долями, то есть с одним знаком после запятой.

А что будет, если какое-либо значение насыщенности не будет удовлетворять наложенным ограничениям? Например, мы укажем насыщенность какого-либо цвета в 110 процентов. Оказывается, при этом не будет возбуждаться механизм генерации ошибки. Наше ошибочное значение будет всего лишь приведено к ближайшей допустимой величине. Что представляется весьма разумным.

Также в CSS level 2 добавлено несколько предустановленных цветовых значений, которые соответствуют текущим системным установкам.

- `ActiveBorder`. Обозначает цвет границы текущего активного окна.
- `ActiveCaption`. Идентифицирует цвет, которым отображается строка заголовка активного окна.
- `AppWorkspace`. Задает цвет фона, на котором отображаются документы в обычных многодокументных приложениях (например, в MS Word).
- `Background`. Указывает цвет фона рабочего стола (Desktop).

- `ButtonFace`. Совпадает с цветом передней поверхности любого трехмерного элемента управления, такого как обычная кнопка.
- `ButtonHighlight`. Идентифицирует цвет светлых границ стандартных трехмерных кнопок.
- `ButtonShadow`. Соответствует цвету, применяемому системой для отображения темной границы (обычно правой и нижней) трехмерной кнопки.
- `ButtonText`. Ссылается на цвет, применяемый для отображения надписей на обычных трехмерных кнопках.
- `CaptionText`. Устанавливает цвет, идентичный тому, какой применяется для отображения текста заголовка активного окна.
- `GrayText`. Задает цвет, которым отображается недоступный (disabled) текст. Чаще всего мы видим этот цвет в оформлении неактивных пунктов меню.
- `Highlight`. Идентифицирует стандартный цвет для текста, который пользователь выбрал в каком-либо элементе управления.
- `HighlightText`. Задает цвет, которым отображается текст выделенного элемента.
- `InactiveBorder`. Обозначает цвет, которым отображается граница неактивного окна.
- `InactiveCaption`. Ссылается на цвет, которым заполняется строка заголовка неактивного окна.
- `InactiveCaptionText`. Задает цвет, которым отображается текст на заголовке неактивного окна.
- `InfoBackground`. Задает цвет, используемый для фона всплывающих информационных окон.
- `InfoText`. Обозначает цвет, которым отображается текст информационных окон.
- `Menu`. Идентифицирует цвет фона элементов меню.
- `MenuText`. Ссылается на цвет, которым отображается текст пунктов меню.
- `Scrollbar`. Задает цвет серой области полос прокрутки.
- `ThreeDDarkShadow`. Ссылается на цвет, которым отображается темная граница трехмерных органов управления.
- `ThreeDFace`. Указывает цвет, которым отображается передний план трехмерных органов управления.
- `ThreeDHighlight`. Ссылается на цвет, используемый для отображения выбранных пользователем, подсвеченных трехмерных органов управления.

- `ThreeDLightShadow`. Ссылается на цвет, которым отображается светлая граница трехмерных органов управления.
- `ThreeDShadow`. Идентичен цвету `ThreeDDarkShadow`.
- `Window`. Ссылается на цвет фона окна.
- `WindowFrame`. Указывает на цвет дочернего окна.
- `WindowText`. Указывает цвет, которым отображается обычный текст в окне.

Все эти значения на самом деле нечувствительны к регистру, поэтому использование заглавных букв в наименованиях является необязательным. Но для удобства чтения стоит их писать в том виде, в котором они здесь представлены.

В этой части мы ознакомились с единицами измерения, применяемыми для задания значений свойств. Но мы еще не знаем наименований всех свойств, применяемых в CSS level 2. Следующие части этой главы будут посвящены рассмотрению всех свойств, регулируемых каскадными стилевыми таблицами.

Модели представления информации

CSS 2 является технологией, нацеленной в будущее. Она предполагает, что информация будет представляться в разных моделях. Не просто в разном виде, но в самых различных *моделях представления информации*. Эти модели в спецификации CSS level 2 называются media types.

В CSS level 2 различаются следующие модели представления информации:

- `all`. Применяется для обозначения модели информации, используемой на всех типах устройств воспроизведения информации.
- `aural`. Используется для задания свойств информации, которая синтезируется в виде человеческой речи. Да, действительно, CSS level 2 предполагает возможность управления синтезом речи, задавая свойства создаваемого речевого потока. Но к XML это пока что не имеет никакого отношения, поэтому и рассматривать детально эту модель мы не будем.
- `braille`. Данная модель предназначена для создания информации, отображаемой на устройствах, генерирующих последовательность символов алфавита Брайля, применяемого для чтения слепыми людьми.
- `embossed`. Применяется для отображения информации на страницах, печатаемых на специализированных брайлевых принтерах.
- `handheld`. Эта модель представления информации рассчитана на карманные портативные компьютеры, такие как, например, Palm. Их характеризует маленькая область экрана, монохромность и прочие ограничения.

- `print`. Используется для задания правил отображения информации, которая предназначена для печати на обычных принтерах и отображения в режиме предварительного просмотра.
- `projection`. Применяется для отображения информации, которая будет показана при помощи какого-либо проекционного устройства.
- `screen`. Как видно из названия, используется для отображения информации на экране стандартного монитора.
- `tty`. Данная модель предназначена для отображения информации на устройствах с ограниченными возможностями отображения, таких как телетайпы, терминалы или переносные подключаемые устройства с ограниченным экраном.
- `tv`. Применяется для задания правил отображения информации на экране телевизора.

Но нас будет интересовать отдельная модель представления информации, наиболее подходящая для отображения документов XML. Мы можем искусственно разбивать их на страницы и отображать эти страницы соответствующим образом. Для этого применяется *страничная модель представления информации* (paged media).

Для определения характера отображения информации, поделенной на страницы, применяется специализированное правило, для которого в качестве селектора записывается конструкция `@page`. Вариант задания правила отображения отдельной страницы показан в следующем примере:

```
@page { size 8.5in 11in; margin: 2cm }
```

Теперь, когда мы знаем, какую модель представления данных мы будем использовать, мы можем рассмотреть те свойства страниц, которыми возможно управлять в рамках технологии CSS level 2.

Свойство `size` позволяет задавать размер страницы. Синтаксис возможного значения данного свойства описывается следующим образом:

```
<length>{1,2} | auto | portrait | landscape
```

Как видно, мы можем задавать размеры страницы по вертикали и горизонтали при помощи пары значений. Ключевое слово `length` означает, что мы можем использовать абсолютные или относительные единицы измерения. Угловые скобки, примыкающие к нему, показывают, что вместо ключевого слова в правило записывается конкретное значение.

В качестве одного из значений данного свойства мы можем использовать ключевое слово `auto`. При этом размеры страницы будут максимально точно подогнаны под общие размеры страниц, оптимальные для данного принтера или монитора. Это значение устанавливается по умолчанию.

Значение `portrait` устанавливает так называемое портретное расположение страницы, при котором большая из сторон страницы расположена по вертикали. Значение `landscape` принудительно разворачивает страницу таким образом, что ее более длинная сторона имеет горизонтальную ориентацию.

Пример принудительной установки размеров страницы выглядит следующим образом:

```
@page { size: 8.5in 11in;}
```

В этом примере мы задаем размеры страницы равными восьми с половиной на одиннадцать дюймов.

Свойство `marks` позволяет определять внешний вид меток верстки, размещаемых в углах области отображения текста. Синтаксис возможного значения свойства описывается следующим образом:

```
[ crop | cross ] | none
```

Значение по умолчанию `none` указывает на то, что метки верстки вообще не будут отображаться. Значение `crop` отображает метки в виде прямых углов с направленными вовне концами. Значение `cross` воспроизводит метки в виде крестов.

Информация, представленная в виде отдельных страниц, может отображаться на экране или печататься на принтере. При этом страницы могут разбиваться на четные и нечетные, и, соответственно, оформляться по-разному. Для четных и нечетных страниц можно устанавливать зеркальное представление, первая страница может отображаться совершенно особым образом.

Для того чтобы различать страницы, в CSS level 2 введено три псевдокласса, которые, соответственно, позволяют задавать оформление левых и правых страниц, а также первой страницы документа.

Псевдокласс `@page:left` предназначен для создания правил отображения левых (четных) страниц. Псевдокласс `@page:right` позволяет работать с правыми (нечетными) страницами. Правила отображения первой страницы создаются при помощи псевдокласса `@page:first`.

Например, при помощи следующей конструкции мы можем задавать левое и правое поле отступа для четных страниц:

```
@page :left {
  margin-left: 4cm;
  margin-right: 3cm;}
```

Следующие три свойства относятся к размещению блоков на странице. Как мы уже упоминали, каждый элемент оформления вписывается в свой собственный блок-контейнер. Для текстовой информации такими блоками явля-

ются абзацы текстового содержимого XML-документа. При разбиении содержимого на страницы мы можем регулировать порядок размещения блоков на этих страницах.

Свойства `page-break-before` и `page-break-after` управляют разметкой страниц перед началом нового блока и после него соответственно. Приведем синтаксис их возможных значений:

```
auto | always | avoid | left | right
```

"Умолчальное" значение `auto` перекладывает проблемы постраничного разбиения контента в зависимости от наполнения блоками на плечи отображающего приложения. Значение `always` принудительно устанавливает разрыв страницы до или после каждого нового блока, в зависимости от применяемого свойства. Значение `avoid` требует игнорировать разрывы страниц перед блоками или после них. Значение `left` форматирует страницы таким образом, что следующая страница после содержащей блок будет отображена как левая (с четным номером). При этом разрыв страниц будет ставиться до или после блока-контейнера в зависимости от используемого свойства. Действие значения `right` похоже на действие предыдущего значения, с тем отличием, что следующая страница будет считаться правой.

Свойство `page-break-inside` управляет положением разрывов страниц внутри блока. Для этого свойства допускается использовать только значения `auto` и `avoid`. Что соответствует следующему описанию синтаксиса:

```
avoid | auto
```

Нетрудно сообразить, что и это свойство не оставляет нам возможности сознательного манипулирования разрывами в пределах блока.

Следующие два свойства регулируют разрывы внутри блоков, а если быть совсем точным — внутри абзацев.

Свойство `orphans` определяет минимальное количество строк абзаца, переносимое на следующую страницу. Значение свойства является целым числом. По умолчанию на новую страницу переносится не менее двух строк. Иначе говоря, использование этого свойства накладывает запрет на появление "висячих" строк текста.

Свойство `widows` указывает, какое минимальное количество строк из абзаца должно обязательно оставаться на странице. Естественно, значение свойства является целым числом. По умолчанию на странице остается как минимум две строки.

Модели ячеек

Для содержимого каждого экземпляра любого элемента XML-документа резервируется определенное место в окне просмотра браузера. Это место называют *ячейкой* (box). В CSS существует специальная концепция обработки таких зарезервированных мест, называемая *моделью ячеек* (box model). Наиболее четко выделены, распознаются ячейки в таблицах. Но вопреки расхожему мнению, ячейки могут находиться и вне таблиц. Каждый экземпляр элемента, отображаемый в окне просмотра браузера, заключается в подобную ячейку как в контейнер.

В CSS различается несколько видов ячеек-контейнеров.

- *Локальный контейнер* в спецификации CSS обозначается как inline box. Локальные контейнеры не могут содержать в себе других ячеек. Это основные, самые маленькие единицы форматирования содержимого XML-документа. Они могут входить в состав других контейнеров, но сами неделимы.
- *Ячейка-блок*, обозначаемая как block box, является базовым элементом разметки. Может в себе содержать другие ячейки.
- *Компактная ячейка* (compact box) является частным случаем локальной ячейки. В компактных ячейках может содержаться только одна строка.
- *Начальная ячейка* (run-in box) размещается в самом начале ячейки-блока. Ближайшим аналогом является первая строка в абзаце.

На основе этой классификации ячеек действуют свойства визуального отображения элементов.

Свойство `display` позволяет идентифицировать вид блока, в котором будет размещаться содержимое того или иного элемента. Описание синтаксиса возможных значений данного свойства выглядит следующим образом:

```
inline | block | list-item | run-in | compact | marker | table |
↳inline-table | table-row-group | table-header-group |
↳table-footer-group | table-row | table-column-group |
↳table-column | table-cell | table-caption | none | inherit
```

То есть для свойства `display` предусмотрено восемнадцать (!) предустановленных значений. Рассмотрим их.

Значение по умолчанию `inline` означает, что данная ячейка будет квалифицироваться как локальный контейнер. Значение `block` специфицирует ячейку как ячейку-блок. Ключевое слово `list-item` указывает уже не на тип ячейки, содержащей наполнение элемента, а на тип самого наполнения. При использовании этого значения свойства `display` считается, что содержимое данного элемента является элементом списка. Значение `run-in` определяет начальную ячейку. А значение `compact` создает компактную ячейку.

Ключевое слово marker позволяет обозначать ячейки, содержимое которых находится сразу после или до значка маркера, устанавливаемого перед элементами списков. Значение table указывает, что в данной ячейке содержится таблица. А значение inline-table создает таблицу, которая в своих клетках не может содержать вложенных таблиц. Предустановленное значение table-row-group обозначает элемент, объемлющий несколько строк таблицы. Значение table-header-group соответствует верхней строке таблицы, которая содержит заголовки столбцов. А значение table-footer-group специфицирует нижнюю строку таблицы, которая также может содержать наименование столбцов или итоговые значения. Зарезервированное слово table-row указывает, что данный элемент содержит строку таблицы. А предустановленное значение table-column-group специфицирует элемент, объединяющий несколько столбцов таблицы. Если ячейка должна содержать только один столбец таблицы, то следует использовать обозначение table-column. Значение table-cell говорит о том, что данный элемент является обычной клеткой таблицы. А для обозначения элемента, который содержит заголовок таблицы, применяется значение table-caption. Значение none используется для спецификации элементов оформления, не отображаемых при помощи ячеек-контейнеров. То есть таким элементам не присваивается какое-либо особенное расположение в окне просмотра. При этом элементы, для которых свойство display имеет значение none, принудительно передают это значение всем своим наследникам. Значение none, наследованное дочерними элементами, не может быть перекрыто. Ключевое слово inherit означает, что свойство display наследуется от родительского элемента.

На этом разбор свойства display заканчивается, и мы переходим к свойствам, регулирующим общие параметры расположения элементов в окне просмотра браузера.

Свойство position позволяет задавать модель позиционирования элемента в окне просмотра. Синтаксис его возможных значений задается следующей конструкцией:

```
static | relative | absolute | fixed | inherit
```

Значение static, используемое по умолчанию, означает, что данный элемент будет размещен в окне просмотра браузера с использованием стандартных правил размещения блоков, обрамляющих содержимое элемента. То есть специализированные свойства, позволяющие задавать координаты верхнего левого угла ячейки, в этой модели не будут иметь силы.

Значение relative устанавливает модель относительного позиционирования для данного элемента и всех остальных дочерних элементов, содержимое которых размещается в ячейке, предназначенной для родительского элемента. При этом координаты дочерних ячеек отсчитываются от верхнего левого угла родительской ячейки.

Значение absolute определяет абсолютную систему позиционирования для элемента. В этой модели можно использовать все свойства, устанавливающие координаты и размеры ячеек-контейнеров.

Значение fixed задает фиксированную систему размещения. Фиксированная система позиционирования очень похожа на абсолютную, но при этом во внимание принимаются ограничения, присущие типу информации. То есть если документ отображается в окне просмотра браузера, то учитываются размеры окна просмотра. При этом если размеры и положение ячеек выводят их за границы окна просмотра, то несоответствующие значения игнорируются, и ячейки-контейнеры отображаются в пределах окна просмотра.

Ключевое слово inherit означает, что значение этого свойства наследуется от родительского элемента.

Если мы используем системы позиционирования, позволяющие контролировать координаты и размеры ячеек-контейнеров, то есть все системы, кроме задаваемой значением static свойства position, мы можем применять четыре специализированных свойства.

Свойство top позволяет задавать вертикальное смещение элемента, то есть указывать вертикальную координату верхней границы ячейки-контейнера, содержащей наполнение данного элемента. Синтаксическое определение возможных значений данного свойства выглядит следующим образом:

```
<length> | <percentage> | auto | inherit
```

Ключевое слово length означает, что мы можем задать вертикальное смещение при помощи абсолютных или относительных единиц измерения. Либо можно использовать процентное соотношение (percentage). Значение по умолчанию auto применяется в тех случаях, когда мы в вопросах размещения ячеек-контейнеров полагаемся на сам браузер. Ключевое слово inherit означает, что значение этого свойства наследуется от родительского элемента.

Свойство bottom позволяет задавать вертикальную координату нижней границы ячейки, в которой находится содержимое элемента. Порядок задания значений этого свойства идентичен порядку для только что рассмотренного нами свойства top.

Свойства left и right предназначены для установки горизонтальных координат левой и правой границы ячейки-контейнера, соответственно. Значения для них используются те же самые, что и для свойств top и bottom.

Свойство float позволяет сдвигать блок к левой или правой границе окна просмотра. Синтаксис значений описывается следующим образом:

```
left | right | none | inherit
```

Применение значений left и right генерирует блочную ячейку, которая вне зависимости от ее позиционирования смещается влево или вправо соответственно. Значение none не смещает ячейку к краю окна просмотра браузе-

ра. Ключевое слово `inherit` означает, что значение этого свойства наследуется от родительского элемента.

Свойство `width` принудительно задает ширину ячейки-контейнера, в которой будет находиться содержимое элемента. Синтаксис возможных значений совпадает с синтаксисом значений только что рассмотренных нами четырех свойств позиционирования. Необходимо отметить, что свойство `width` не может применяться к элементам, размещаемым в локальных ячейках, так как ширина локальных ячеек определяется браузером автоматически, исходя из объема содержимого отображаемого элемента.

Помимо жесткого задания ширины ячейки-контейнера, мы можем задавать некие ограничения по ширине, которые являются более мягким условием. В CSS level 2 предусмотрены свойства, позволяющие ограничивать минимальную и максимальную ширину ячейки.

Свойство `min-width` позволяет устанавливать минимальную ширину ячейки, в которой находится содержимое данного элемента. Синтаксис возможных значений этого свойства описывается следующим образом:

```
<length> | <percentage> | inherit
```

То есть все как обычно. Мы можем задавать минимальную ширину или прямым указанием размера, или при помощи процентного соотношения. Ключевое слово `inherit` означает, что значение этого свойства наследуется от родительского элемента.

Свойство `max-width` ограничивает максимальную возможную ширину ячейки-контейнера. Синтаксис возможных значений этого свойства описывается при помощи следующей конструкции:

```
<length> | <percentage> | none | inherit
```

Как видно, особых отличий от синтаксиса значений предыдущего свойства нет. Добавлено лишь еще одно значение `none`, используемое по умолчанию. Само собой, оно указывает, что максимальная ширина ячейки не ограничена.

Естественно, кроме ширины мы можем назначать и высоту ячеек-контейнеров. Для директивного указания высоты используется свойство `height`. Синтаксис возможных значений этого свойства описывается так:

```
<length> | <percentage> | auto | inherit
```

Значение по умолчанию `auto` означает, что реальная высота ячейки вычисляется, исходя из других свойств, таких как `top` и `bottom`. Механизм действия остальных значений мы прекрасно помним.

Как мы помним, в CSS level 2 существуют свойства, задающие "мягкие условия" для ширины ячейки-контейнера. Для высоты, естественно, существуют их аналоги.

Свойство min-height позволяет ограничивать минимальную высоту ячейки-контейнера. Синтаксическое определение возможных значений этого свойства выглядит так:

```
<length> | <percentage> | inherit
```

По умолчанию свойству min-height приписывается нулевое значение.

Свойство max-height позволяет установить максимальную высоту для ячейки-контейнера, которая не может быть превышена. Синтаксис значений этого свойства описывается следующим образом:

```
<length> | <percentage> | none | inherit
```

К набору значений предыдущего свойства здесь добавлено значение none, используемое по умолчанию, которое сигнализирует браузеру, что искусственных ограничений на максимальную высоту данной ячейки нет.

Вышеперечисленные свойства принудительной установки ширины и высоты ячейки не имеют силы, если содержимое элемента отображается в локальной ячейке. По определению, ширина и высота локальных ячеек определяются их содержимым. Но свойства для указания высоты локальных ячеек в CSS level 2 все-таки существуют.

Так как чаще всего локальная ячейка является отдельной строкой, для установки ее высоты используется значение line-height. Синтаксическое определение возможных значений этого свойства выглядит следующим образом:

```
normal | <number> | <length> | <percentage> | inherit
```

Значение по умолчанию normal указывает, что высота ячейки будет вычисляться браузером, исходя из размера применяемого шрифта, полей и отступов. Конструкция <number> в синтаксическом определении показывает на то, что для определения высоты локальной ячейки можно использовать коэффициент, на который домножается высота применяемого шрифта. Помимо этого способа задания высоты, мы можем использовать стандартные единицы измерения и процентные соотношения. Ключевое слово inherit, как обычно, означает, что значение этого свойства наследуется от родительского элемента.

Вертикальное выравнивание локальной ячейки производится при помощи свойства vertical-align. Синтаксис возможных значений этого свойства описывается при помощи следующей конструкции:

```
baseline | sub | super | top | text-top | middle | bottom |
↳text-bottom | <percentage> | <length> | inherit
```

Значение по умолчанию baseline выравнивает вертикально локальную ячейку по уровню базовой линии родительского элемента. Если у родительского элемента нет базовой линии, то выбирается ближайший по объектной иерархии элемент-прародитель, у которого эта базовая линия есть. Базовой

линией называют нижний край стандартных символов применяемого шрифта, то есть тех, у которых нет "нижних хвостиков".

Значение `sub` устанавливает данную локальную ячейку чуть ниже уровня базовой линии, позиционируя ее так же, как и нижние индексы. Необходимо отметить, что данное свойство может применяться только в том случае, если размер ячейки родительского элемента больше, чем размер шрифта, которым отображается текстовое содержимое все того же родительского элемента.

Значение `super`, наоборот, позиционирует локальную ячейку на уровне верхнего индекса относительно базовой линии ячейки родительского элемента.

Значение `top` позволяет прижать локальную ячейку к самому верху родительской ячейки. Альтернативное значение `bottom` прижимает локальную ячейку к нижней границе ячейки, в которой базируется родительский элемент. А значение `middle` центрирует локальную ячейку по вертикали в пространстве родительской ячейки.

Значение `text-top` выравнивает локальную ячейку по уровню, определяемому высотой используемого шрифта. При этом, разумеется, используется самая верхняя строка. А значение `text-bottom` производит выравнивание по нижней линии используемого шрифта.

Конструкции `<percentage>` и `<length>` указывают, что мы можем производить вертикальное выравнивание с использованием процентного соотношения или стандартных единиц измерения. При этом смещение определяется относительно базовой линии, то есть того уровня, который был бы установлен при использовании значения `baseline`. Положительные значения смещают ячейку вверх от базовой линии, отрицательные — вниз.

Ключевое слово `inherit` заставляет данную локальную ячейку наследовать способ вертикального выравнивания у родительского элемента.

Помимо конкретного содержимого ячеек-контейнеров, браузер обязан учитывать дополнительные элементы отображения, входящие в состав модели ячеек. Их называют *отступами* (padding), *границами* (border) и *полями* (margin). Взаимное их расположение показано на рис. 7.1.

Для регулировки размеров боковых полей применяются свойства `margin-left`, `margin-right`, `margin-top` и `margin-bottom`. Они позволяют явно задавать размер левого, правого, верхнего и нижнего поля соответственно. Синтаксис их возможных свойств одинаков и описывается следующей конструкцией:

```
<margin-width> | inherit
```

О действии ключевого слова `inherit` мы уже знаем. Оно вынуждает элемент унаследовать значение данного свойства у родительского элемента. А синтаксис конструкции `<margin-width>` определен в версии спецификации CSS level 2. Предусмотрено три возможности. Мы можем либо явно указать раз-

мер полей при помощи стандартных единиц измерения, либо использовать процентное соотношение. Также в качестве значения этого свойства мы можем использовать ключевое слово `auto`, тогда размеры полей будут определены браузером, исходя из его общих установок. В качестве значения по умолчанию для этих свойств используется нулевое значение.

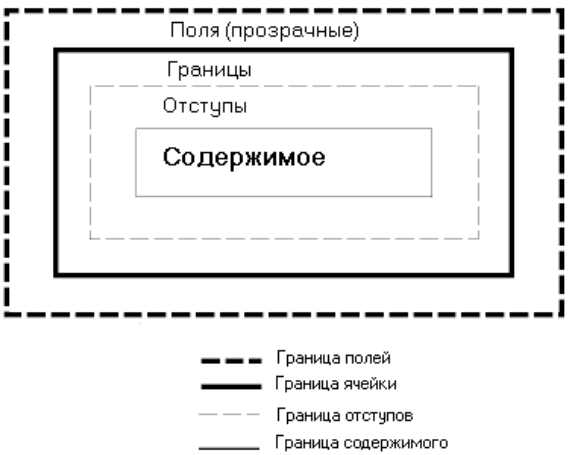


Рис. 7.1. Схема элементов обрамления ячеек-контейнеров

Существует также агрегатное свойство `margin`, позволяющее устанавливать размеры всех полей сразу, не прибегая к помощи только что рассмотренных свойств. Синтаксис его значений задается при помощи конструкции:

```
<margin-width>{1,4} | inherit
```

Как видно, мы можем контролировать от одного до четырех размеров полей. Требуется соблюдать следующий порядок указания полей: верхнее поле, правое, нижнее, левое. При этом если указаны не все величины, то горизонтальные или вертикальные поля объединяются. Более подробно это можно увидеть в рассматриваемых ниже примерах.

Так, для того чтобы установить размер всех полей равным двум миллиметрам, используется такая конструкция:

```
BODY { margin: 2mm }
```

Если же в правило включены два значения, то первое из них устанавливает размер вертикальных полей, а второе — горизонтальных. Проиллюстрировать этот случай можно следующим правилом оформления:

```
BODY { margin: 1mm 2mm }
```

В этом примере для вертикальных полей будет установлен размер величиной в один миллиметр, а для горизонтальных — будет равен двум миллиметрам.

А если мы укажем в качестве значения свойства margin три величины, то совпадут только горизонтальные поля. Проиллюстрировать данный вариант можно при помощи фрагмента кода:

```
BODY { margin: 1mm 2mm 3mm }
```

Как видно, для верхнего поля будет использовано значение, равное одному миллиметру, для правого поля — двум миллиметрам, а для нижнего — трем миллиметрам. Так как величина левого поля не была задана явно, то оно будет иметь такой же размер, как и правое, то есть — два миллиметра.

Отступы задаются при помощи соответствующих свойств. Свойства padding-top, padding-bottom, padding-right и padding-left позволяют задавать верхний, нижний, правый и левый отступы соответственно. Синтаксис их свойств описывается следующей конструкцией:

```
<padding-width> | inherit
```

Конструкция <padding-width> означает, что для указания размера отступа мы можем использовать процентное соотношение или обычные единицы измерения CSS level 2.

При этом автоматическое задание величины отступа при помощи ключевого слова auto не допускается. По умолчанию используется нулевой отступ. Ключевое слово inherit означает, что значение данного свойства необходимо унаследовать от родительского элемента.

Как и при указании размеров полей, мы можем использовать единое агрегатное свойство для задания размеров отступов. Оно носит наименование padding. Синтаксическое определение его значений выглядит следующим образом:

```
<padding-width>{1,4} | inherit
```

Как видно, механизм задания значений для этого свойства совпадает с порядком задания значений свойства margin.

При помощи свойств border-top-width, border-right-width, border-bottom-width и border-left-width мы можем устанавливать толщину границы ячейки, в которой отображается содержимое элемента. Они регулируют толщину верхней, правой, нижней и левой границы соответственно.

Возможные значения этих свойств определяются при помощи следующей конструкции:

```
<border-width> | inherit
```

Определение <border-width> в спецификации CSS level 2 позволяет установить, что для указания толщины границы мы можем использовать стандартные единицы измерения. Процентное соотношение или автоматическая установка ширины не допускаются. Помимо прямого задания толщины можно использовать три ключевых слова: thin, medium и thick, которые

задают, соответственно, узкую, среднюю и широкую границу. По умолчанию используется значение `medium`.

Естественно, имеется и агрегатное свойство `border-width`, позволяющее устанавливать толщину каждой границы ячейки. Синтаксическое определение значений данного свойства выглядит так:

```
<border-width>{1,4} | inherit
```

Как видно, стандартные правила задания значений агрегатных свойств действуют и в этом случае.

Конечно, границы не могут характеризоваться одной только толщиной. В CSS level 2 описаны свойства `border-top-color`, `border-right-color`, `border-bottom-color` и `border-left-color`, которые позволяют устанавливать цвет верхней, правой, нижней или левой границы ячейки соответственно. Синтаксис их свойств описывается простой конструкцией:

```
<color> | inherit
```

Ключевое слово `inherit`, как и ранее, предназначено для наследования значения данного свойства от родительского элемента. А способы задания цвета мы рассматривали в *разд. "Единицы измерения в CSS"*.

Как мы и ожидали, существует обобщающее свойство `border-color`, которое позволяет устанавливать цвет для всех границ сразу. Синтаксис его возможных свойств описывается следующим образом:

```
<color>{1,4} | transparent | inherit
```

Из определения видно, что помимо знакомого нам ключевого слова `inherit` и указания цвета стандартным образом, для установки значения этого свойства мы можем использовать ключевое слово `transparent`. При этом граница ячейки будет прозрачной. Естественно, для этого граница должна иметь некую толщину.

Кроме цвета, граница ячейки может (и должна) иметь определенный стиль. Стиль границы позволяет указывать, как именно будет выглядеть линия, ограничивающая ячейку, в которой отображается данный экземпляр какого-либо элемента. Применяются для этого свойства `border-top-style`, `bottom-right-style`, `border-bottom-style` и `border-left-style`. Они устанавливают внешний вид верхней, правой, нижней и левой границы ячейки соответственно.

В качестве значений этих свойств может применяться одно из нижеперечисленных ключевых слов.

Значение `none` просто убирает границу. Действие этого варианта идентично установке нулевого значения свойства `border-width`.

Ключевое слово `hidden` создает границу, но делает ее невидимой. Этот вариант отличается от предыдущего, так как, несмотря на невидимость, гра-

ница все же существует, что позволяет более корректно размещать ячейки-контейнеры без наложения их друг на друга.

Ключевое слово `dotted`, используемое в качестве значения этого свойства, создает границу в виде серии точек.

Значение `dashed` указывает, что границы ячеек-контейнеров будут проведены пунктирными линиями.

Для установки обычных границ в виде непрерывных линий используется значение `solid`.

Значение `double` заставляет браузер отображать границу ячейки-контейнера в виде двух непрерывных линий. При этом расстояние между линиями определяется шириной границы, то есть значением свойства `border-width`.

Ключевое слово `groove` позволяет оформлять границу ячейки в виде некоего углубления, создавая определенную иллюзию трехмерности.

Значение `ridge` создает эффект, обратный по отношению к значению `groove`. Оно заставляет браузер отображать границу ячейки-контейнера в виде трехмерного выступа.

Для имитации прямоугольной канавки используется значение `inset`. Отличие этого значения от рассмотренного нами ключевого слова `groove` заключается в том, что создаваемый желобок имеет прямоугольную форму.

А ключевое слово `outset`, применяемое в качестве значения рассматриваемого свойства, создает границу в виде прямоугольного выступа.

Необходимо отметить, что последние четыре рассмотренных нами значения, создающие "трехмерные" границы, реализуются посредством изменения цвета, установленного для самого содержимого элемента. То есть цвет трехмерных границ зависит от значения свойства `color`, заданного для элемента в целом.

Естественно, рассматриваемые нами свойства могут иметь не только перечисленные выше значения, но и наследоваться от родительских элементов. Для этого, как всегда, используется ключевое слово `inherit`.

Агрегатное свойство `border-style` позволяет задавать стили отображения для всех границ сразу. Возможные значения этого свойства определяются при помощи следующей конструкции:

```
<border-style>{1,4} | inherit
```

Как легко заметить, значения для всех агрегатных свойств задаются единообразно.

Помимо рассмотренных нами свойств существуют свойства более общего вида. Так, свойства `border-top`, `border-right`, `border-bottom` и `border-left` позволяют устанавливать целиком внешний вид верхней, правой, нижней и

левой границы ячейки-контейнера соответственно. Синтаксис значений этих свойств одинаков и определяется следующим образом:

```
[ <'border-width'> || <'border-style'> || <color> ] | inherit
```

То есть для каждой границы мы можем установить сразу ее ширину, стиль отображения и цвет. Например, следующее правило оформления для элемента `caption` задает параметры его нижней границы:

```
caption { border-bottom: thick solid }
```

Как нетрудно заметить, в этом примере мы устанавливаем толщину нижней границы и ее стиль. Нижняя граница будет отображена в виде жирной непрерывной линии.

Следует отметить, что если у рассматриваемых свойств какое-либо из трех значений не задано, то используется значение по умолчанию, установленное для этого параметра.

Если все четыре границы должны выглядеть одинаково, то имеет смысл для задания правила оформления использовать единое свойство `border`. Его значения определяются при помощи следующей конструкции:

```
[ <'border-width'> || <'border-style'> || <color> ] | inherit
```

Как видно, синтаксис значений этого свойства повторяет определение значений предыдущих четырех агрегатных свойств. Правила задания свойств, естественно, тоже совпадают. Следующее правило отображения позволяет устанавливать границу для всей ячейки-контейнера сразу:

```
text { border: solid red }
```

При этом вся граница ячейки, в которой находится содержимое элемента `text`, будет отображена в виде непрерывной красной линии.

Необходимо обратить внимание на тот факт, что в отличие от агрегатных свойств `margin` или `padding`, свойство `border` не позволяет устанавливать внешний вид для каждой из четырех границ отдельно. В этом случае элементы отображаются единообразно.

Некоторые ячейки могут перекрывать друг друга. Для определения, какая именно ячейка будет отображаться сверху, используется свойство `z-index`. Это свойство позволяет создавать некую иерархию слоев. Синтаксис возможных значений этого свойства задается при помощи следующей конструкции:

```
auto | <integer> | inherit
```

Значение по умолчанию `auto` позволяет браузеру самостоятельно выстраивать иерархию слоев у пересекающихся ячеек. При этом элементы, которые расположены ближе к концу XML-документа, отображаются "ближе" к пользователю. То есть отображение происходит в порядке очереди от начала

документа. При этом, если последующий элемент перекрывает предыдущий, он отображается сверху.

Мы можем также в качестве значения этого свойства использовать целое положительное число. Самое приоритетное значение — ноль. Элемент, у которого выставлено нулевое значение свойства `z-index`, обладает наибольшим приоритетом, и он будет всегда отображаться сверху. То есть чем больше значение свойства `z-index`, тем ниже приоритет элемента. Ключевое слово `inherit` означает, что значение этого свойства наследуется от родительского элемента.

Текстовое содержимое элементов может отображаться в ячейках-контейнерах в различном направлении. Для указания этого направления используется свойство `direction`. Синтаксис возможных значений этого свойства задается следующим образом:

```
ltr | rtl | inherit
```

Значение `ltr` заставляет браузер отображать текст слева направо, как мы и привыкли. Значение `rtl` вынуждает изменить направление отображения текстового наполнения данного элемента на противоположное, то есть справа налево. Значение `inherit` указывает, что направление отображения текста наследуется от родительского элемента.

Мы разобрались со стандартными вариантами размещения содержимого элементов в ячейках-контейнерах. Но в спецификации CSS level 2 предусмотрена возможность возникновения ситуации, когда объем отображаемого содержимого элемента может превысить размеры отведенной ему ячейки. Подобная ситуация называется *переполнением* (overflow). Для того чтобы указать браузеру, какие действия необходимо произвести в случае подобной коллизии, предусмотрено соответствующее свойство `overflow`. Синтаксис его возможных значений задается следующим образом:

```
visible | hidden | scroll | auto | inherit
```

Значение по умолчанию `visible` указывает браузеру, что содержимое данного элемента не прикреплено к ячейке и может отображаться за ее пределами. Значение `hidden` обозначает, что содержимое данной ячейки прикреплено к ней, и если часть содержимого выходит за пределы ячейки, то эта часть не будет доступна для просмотра, так как полосы прокрутки для данной ячейки не будут создаваться. Значение `scroll` также прикрепляет содержимое к ячейке, но при этом позволяет браузеру создавать полосы прокрутки для ячейки, если ее содержимое выходит за пределы области просмотра. Использование значения `auto` оставляет выбор поведения за самим XML-процессором. Ключевое слово `inherit` подразумевает, что значение этого свойства наследуется от родительского элемента.

Мы можем устанавливать форму и размер области, в которой находится прикрепленное содержимое. Для этого используется свойство `clip`. Его возможные значения описываются при помощи следующей конструкции:

```
<shape> | auto | inherit
```

Конструкция `<shape>` указывает, что мы можем явно задать форму и размеры области отсечения. К сожалению, в CSS level 2 можно задавать только прямоугольную форму при помощи ключевого слова `rect`. После этого ключевого слова в скобках мы должны указать координаты прямоугольника в следующем порядке: `top`, `right`, `bottom`, `left`. При этом необходимо учитывать, что мы можем задавать не только положительные, но и отрицательные значения. Например, для того чтобы задать область отсечения, смещенную на пять пикселов вправо за границу ячейки-контейнера, следует применить следующую конструкцию:

```
P { clip: rect(5px, -5px, 10px, 5px); }
```

Значение `auto`, используемое по умолчанию, распространяет область отсечения на всю ячейку, в которой отображается содержимое элемента. Ключевое слово `inherit` заставляет данный экземпляр элемента унаследовать значение этого свойства от своего родителя.

А для управления видимостью ячейки-контейнера используется свойство `visibility`. Возможные его значения описываются следующей конструкцией:

```
visible | hidden | collapse | inherit
```

Значение `visible`, естественно, заставляет браузер отображать данную ячейку. Значение `hidden` делает ее невидимой. Значение `collapse` сворачивает ячейку-контейнер. Этот вариант правильно работает только в том случае, когда ячейка является частью таблицы. Иначе, результат действия эквивалентен использованию значения `hidden`. По умолчанию используется значение `inherit`, принуждающее к наследованию свойства `visibility` от родительского элемента.

Фон и цвета

В этом разделе мы рассмотрим свойства CSS level 2, которые регулируют цветовое оформление текстового содержимого элементов XML-документа, и управляют отображением фона элементов.

Свойство `color` задает цвет, которым будет отображаться содержимое элемента. Синтаксическое определение возможного значения данного свойства обозначается следующим образом:

```
<color> | inherit
```

В качестве значения color этого свойства применяется обозначение цвета. Варианты этих обозначений мы рассматривали в *разд. "Единицы измерения в CSS"*. Значение по умолчанию отсутствует. Ключевое слово inherit означает, что для данного элемента значение этого свойства будет унаследовано от родителя.

Свойство background-color позволяет задавать цвет фона для любого элемента. Синтаксис возможных значений данного свойства определяется следующим образом:

```
<color> | transparent | inherit
```

Значение color, как и в предыдущем случае, является обозначением какого-либо цвета, и применяется для явного задания цвета фона элемента. Значение transparent используется по умолчанию и устанавливает прозрачный цвет фона. Ключевое слово inherit означает, что данное свойство наследуется.

Свойство background-image позволяет назначить в качестве фона для содержимого элемента графическое изображение. Синтаксис возможных значений данного свойства задается следующим образом:

```
<uri> | none | inherit
```

Конструкция uri указывает, что в качестве значения данного свойства мы можем использовать URI файла графического изображения. При этом адресуемое изображение будет являться фоном для содержимого нашего элемента. Значение none, используемое по умолчанию, объявляет, что графического фона у данного элемента не будет. Ключевое слово inherit означает, что для данного элемента правила использования графики в качестве фона будут унаследованы от родительского элемента.

В следующем примере показано, как для некоего элемента caption мы задаем в качестве фона графическое изображение.

```
caption {background-image :url ("http://www.mygraph.com/pict1.gif")}
```

Как видно из примера, для указания адреса графического файла мы используем функцию url, которая текстовую строку преобразует в его URL.

Свойство background-repeat позволяет более детально управлять фоном элемента. Возможные значения данного свойства определяются при помощи следующей конструкции:

```
repeat | repeat-x | repeat-y | no-repeat | inherit
```

Напомню, что это свойство имеет смысл применять только в том случае, если в качестве фона элемента явно установлено графическое изображение. Поскольку изображение не всегда полностью занимает рабочее пространство окна просмотра браузера, оно может повторяться в нем, распространяясь

по вертикали и/или горизонтали. Свойство `background-repeat` как раз и управляет подобным повторением.

Значение по умолчанию `repeat` указывает, что если графическое изображение не полностью покрывает рабочее пространство окна просмотра браузера, оно будет повторяться по вертикали и горизонтали. Значение `repeat-x` позволяет тиражироваться фоновому рисунку только по горизонтали. А значение `repeat-y` требует повторять рисунок только по вертикали. Значение `no-repeat` явно запрещает тиражирование фонового рисунка, как по вертикали, так и по горизонтали. Ключевое слово `inherit` означает, что для данного элемента значение этого свойства будет унаследовано от родительского элемента.

Свойство `background-attachment` позволяет регулировать "привязку" фонового рисунка. Дело в том, что если фоновое изображение привязано к текстовому содержимому XML-документа, то при прокрутке документа в окне просмотра браузера, фоновый рисунок будет перемещаться вместе с содержимым документа. Если же фон привязан к окну просмотра, то при прокрутке смещаться будет только содержимое документа, а фоновый рисунок будет оставаться на своем месте. Данное свойство позволяет управлять поведением фонового рисунка. Список возможных значений этого свойства выглядит следующим образом:

```
scroll | fixed | inherit
```

Значение по умолчанию `scroll` указывает, что фоновое графическое изображение будет при прокрутке перемещаться вместе с содержимым документа. Значение `fixed` "привязывает" фоновый рисунок к окну просмотра браузера. Ключевое слово `inherit` задает принудительное наследование этого свойства от родительского элемента.

Свойство `background-position` позволяет управлять позицией фонового рисунка в окне просмотра браузера. Список возможных значений этого свойства выглядит следующим образом:

```
[ [<percentage> | <length> ]{1,2} | [ [top | center | bottom] ||
↳[left | center | right] ] ] | inherit
```

Разберем внимательно это запутанное выражение. При помощи данного свойства мы фактически задаем координаты фонового рисунка в окне. Для этого мы можем использовать либо два процентных соотношения, либо два числа, в конкретных единицах измерения указывающих отступы рисунка от границ окна просмотра, либо комбинацию из двух ключевых слов. Рассмотрим детально способы задания значений этого свойства.

Пара процентных соотношений задает координаты верхнего левого угла фонового рисунка. При этом используется процентное выражение величины ячейки прямоугольника, в котором размещается содержимое элемента. Так,

следующая конструкция помещает фоновый рисунок в самый левый верхний угол прямоугольника-контейнера, содержащего головной элемент `body`:

```
body {background-position : 0% 0%}
```

Здесь первое процентное значение показывает смещение по горизонтали, а второе — по вертикали от верхнего левого угла контейнера. В том случае, если мы указываем достаточно большие процентные величины, например, `100% 100%`, то изображение прижимается к правому нижнему углу, но, тем не менее, отображается.

Помимо процентных значений мы можем напрямую указывать смещение фонового рисунка, используя абсолютные и относительные единицы измерения. Данный способ иллюстрирует следующее правило CSS:

```
body {background-position : 15 mm 20 mm}
```

В этом случае мы хотим, чтобы фоновое графическое изображение было смещено относительно левого верхнего угла рабочего окна на 15 миллиметров по горизонтали и на 20 миллиметров по вертикали.

Также мы можем использовать ключевые слова, приведенные в синтаксическом определении возможных значений. Для позиционирования рисунка по горизонтали применяются ключевые слова `left`, `center` и `right`. Они позволяют прижимать изображение к левому краю ячейки, центрировать его или прижимать к правому краю ячейки соответственно. Ключевые слова `top`, `bottom` и `center` регулируют вертикальное положение фонового рисунка и позволяют прижимать его к верхнему или нижнему краю ячейки-контейнера, или центрировать по вертикали соответственно.

Все вышеперечисленные свойства фона объединяются в одном агрегатном свойстве `background`. Синтаксическое определение его возможного значения выглядит следующим образом:

```
[<'background-color'> || <'background-image'> ||
↳<'background-repeat'> || <'background-attachment'> ||
↳<'background-position'>] | inherit
```

Как видно, мы можем в одном свойстве задать все правила оформления фона. Проиллюстрировать это можно следующим примером:

```
P { background: url("chess.png") gray 50% repeat fixed }
```

В этом правиле оформления мы для элемента `P` устанавливаем фоновое изображение. Графический файл, в котором оно находится, имеет URL "chess.png". В том случае, если графический файл не будет найден, фон будет залит серым цветом. Фоновый рисунок будет сдвинут на пятьдесят процентов по горизонтали, и может повторяться по вертикали или горизонтали, если ячейка-контейнер больше по размерам, нежели само графическое изображение. Также фоновый рисунок прикрепляется к окну просмотра, и не

будет сдвигаться вместе с текстовым содержимым при использовании полос прокрутки.

Итак, мы рассмотрели все свойства, регулирующие применение фона и установку цвета текстового содержания элемента. Можно идти дальше.

Свойства шрифтов

Как мы знаем, в документах, предназначенных для опубликования в Интернете, нет, и не может быть в принципе строго определенного шрифтового оформления. Связано это с тем, что априори неизвестно, какая операционная система, набор шрифтов, разрешение монитора и размер окна просмотра браузера установлены удаленным пользователем, который собирается просмотреть наш документ. Но мы можем предположить, что, скорее всего, у подавляющего большинства пользователей Интернета установлен в системе какой-либо шрифт из семейства Times New Roman или Arial. Следовательно, мы можем регулировать хотя бы немного параметры шрифтового оформления текста. Даже если указанных нами шрифтов и вариантов их воспроизведения не окажется у удаленного пользователя, это не беда. Браузер проигнорирует выданные нами инструкции и отобразит текст с установками, заданными по умолчанию. То есть хуже не будет. Но вполне вероятно, что мы все-таки добьемся своего. Следовательно, использовать свойства шрифтового оформления текста в каскадных стилевых таблицах стоит. Приступим к их рассмотрению.

Свойство `font-family`. Задает наименование семейства шрифта, используемого для отображения текстового содержимого элемента. Синтаксис значения имеет следующий вид:

```
[[<family-name> | <generic-family>],]* [<family-name> |
↳<generic-family>]
```

Как видно из определения, значение данного свойства может состоять из нескольких наименований семейств шрифтов. Вместо наименования семейства шрифтов мы можем задать ключевое слово `generic-family`, которое позволяет регулировать начертание шрифта без указания конкретного семейства. Параметр `generic-family` может принимать значения `serif`, `sans-serif`, `cursive`, `fantasy` и `monospace`.

Значение `serif` указывает, что для оформления применяется шрифт с засечками. Типичный представитель — Times New Roman. Значение `sans-serif` обозначает шрифт без засечек (Helvetica). Для обозначения курсивных шрифтов применяется значение `cursive` (в курсивных шрифтах прописные и строчные буквы в основном разного рисунка, в печатных шрифтах он остается одинаковым для большинства символов). Значение `fantasy` исполь-

зуется для так называемых художественных шрифтов. Например, готических. Для обозначения моноширинных шрифтов, таких как Courier, служит значение `monospace`.

Как мы уже говорили, в значении свойства `font-family` мы можем указать сразу несколько семейств шрифтов. При этом браузер будет подбирать шрифты для отображения текста, следуя порядку, в котором они перечислены. Рассмотрим пример правила отображения, устанавливающего шрифт.

```
element1 {font-family: Arial, fantasy, "Courier New"}
```

Браузер для отображения текстового содержимого элемента с наименованием `element1` сначала попытается использовать шрифт из семейства Arial. Затем, если ни один из шрифтов этого семейства не будет найден в системе пользователя, браузер попытается отобразить содержимое любым фантазийным шрифтом. Если же и их не будет, то браузер будет искать моноширинный шрифт Courier New.

Необходимо отметить, что если ни один из перечисленных шрифтов не будет найден в системе, браузер использует шрифт, установленный по умолчанию.

Также следует обратить внимание на то, что в том случае, когда наименование семейства шрифта состоит из нескольких слов, оно заключается в двойные кавычки, как это показано в примере.

Свойство `font-style`. Может принимать только одно из четырех предустановленных значений. Список значений свойства `font-style` выглядит следующим образом:

```
normal | italic | oblique | inherit
```

Первые три значения позволяют принудительно задавать ориентировку, начертание шрифта. Последнее значение `inherit` указывает, что для данного элемента значение `font-style` будет наследоваться от родительского элемента.

Значение `normal`, используемое по умолчанию, указывает, что будет использоваться прямая ориентировка шрифта. Значение `italic` принудительно переводит выбранный шрифт в курсивное состояние. Значение `oblique` создает наклонное начертание выбранного шрифта.

Данное свойство является наследуемым. Может применяться ко всем типам элементов, имеющим текстовое наполнение.

Свойство `font-variant`. Позволяет принудительно устанавливать регистр символов применяемого шрифта. Синтаксис возможного значения задается следующим образом:

```
normal | small-caps | inherit
```

Значение по умолчанию `normal` оставляет символы текста в том виде, как они были введены. Значение `small-caps` принудительно переводит все символы в

верхний регистр, но при этом их размер не превышает размера обычных строчных символов. Ключевое слово `inherit` означает, что истинное значение данного свойства необходимо унаследовать от родительского элемента.

Данное свойство может применяться к любому элементу, имеющему текстовое наполнение. Свойство может наследоваться.

Свойство `font-weight` позволяет задавать насыщенность символов применяемого шрифта для текстового содержания данного элемента. То есть при помощи этого свойства мы можем управлять толщиной символов. Синтаксис значения данного свойства определяется следующим образом:

```
normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 |
↳600 | 700 | 800 | 900 | inherit
```

Числовые значения позволяют варьировать жирность символов девятью градациями. Наименьшее значение соответствует наиболее тонкому начертанию символов. Помимо числовых значений можно использовать символьные обозначения для четырех фиксированных значений толщины символов.

Значение `normal` соответствует символам стандартной толщины, то есть, в числовом выражении, значению 400. Это значение используется по умолчанию. Значение `bold` приравнивается к числовому значению 700, `bolder` — к 900, `lighter` — к 100.

Значение `inherit` применяется, как обычно, для наследования значения этого же свойства, но заданного для родительского элемента.

Свойство `font-weight` наследуется. Применяется для любого типа элементов, имеющих текстовое наполнение.

Свойство `font-stretch` позволяет задавать величину межсимвольного интервала внутри слов. Список значений этого свойства выглядит так:

```
normal | wider | narrower | ultra-condensed | extra-condensed |
↳condensed | semi-condensed | semi-expanded | expanded |
↳extra-expanded | ultra-expanded | inherit
```

Значение по умолчанию `normal` оставляет нормальный межсимвольный интервал. Остальные значения позволяют принудительно устанавливать просвет, отличающийся от нормального. Если перечислять их в порядке задания интервала от самого узкого до наиболее широкого, получится следующая последовательность:

`ultra-condensed` (50%)

`extra-condensed` (62,5%)

`condensed` (75%)

`semi-condensed` (87,5%)

`normal` (100%)

`semi-expanded` (112,5%)

`expanded` (125%)

`extra-expanded` (150%)

`ultra-expanded` (200%)

В указанной последовательности отсутствуют значения `wider` и `narrower`. В отличие от вышеприведенных значений, эти два значения не являются абсолютными. Они относительные. Значение `wider` сдвигает унаследованное от родительского элемента значение свойства `font-stretch` на одну ступень в сторону более широкого межсимвольного экземпляра. Естественно, если унаследованное значение равно `ultra-expanded`, то дополнительного сдвига не происходит. Значение `narrower` сдвигает текущее значение на один уровень к более узкому межсимвольному промежутку.

Значение `inherit`, как обычно, используется для наследования значения свойства от родительского элемента.

Значение свойства наследуется. Свойство применимо для всех элементов, имеющих текстовое содержимое.

Свойство `font-size` регулирует размер символов применяемого шрифта. Синтаксис возможного значения данного свойства достаточно сложен, и описывается следующим образом:

```
<absolute-size> | <relative-size> | <length> | <percentage> |
↳inherit
```

Как видно, в приведенном выражении только значение `inherit` задано явно. Все остальные типы значений указаны в виде множеств.

Множество `absolute-size` является перечислимым массивом. Всего имеется семь предустановленных значений, которые задают абсолютный размер шрифта. Если перечислить их в порядке увеличения размера шрифта, получится следующая последовательность:

```
xx-small, x-small, small, medium, large, x-large, xx-large
```

Конкретные пропорциональные соотношения между размерами, которые задаются этими значениями, зависят от разрешения монитора, установленного удаленным пользователем.

Множество `relative-size` состоит из двух возможных значений данного свойства. Они используются для задания размера символов относительно высоты символов шрифта родительского элемента. Значение `larger` увеличивает размер символов относительно размера символов родительского элемента, а значение `smaller`, естественно, — уменьшает.

Ключевое слово `length` указывает, что мы можем явно задать высоту шрифта при помощи абсолютных единиц измерения, таких как пункты или миллиметры.

Также мы можем задать размер шрифта в качестве процентного соотношения относительно размера шрифта родительского элемента, о чем говорит наличие в списке ключевого слова `percentage`.

Значение `inherit` в этом случае, как и во всех других ранее, используется для наследования значения свойства от родительского элемента.

По умолчанию используется значение `medium`. Значение применимо для всех элементов, имеющих текстовое содержимое.

Свойство `font-size-adjust` позволяет задавать коэффициент увеличения шрифта. Оно применяется для более точной установки размера символов и напрямую связано с понятием коэффициента масштабирования шрифта. Данный коэффициент является отношением абсолютного размера высоты символов шрифта к высоте символа "x". То есть чем больше этот коэффициент, тем более вытянутыми становятся символы шрифта.

Перечисление возможных значений данного свойства имеет следующий вид:

```
<number> | none | inherit
```

То есть мы можем либо напрямую указать коэффициент масштабирования, либо воспользоваться значением по умолчанию `none`, которое оставляет размеры шрифта без изменения.

Коэффициент масштабирования применяется к шрифту, установленному для родительского элемента. Значение `inherit` используется для наследования значения свойства от родительского элемента. Значение данного свойства наследуется. Свойство применимо для всех элементов, имеющих текстовое содержимое.

Свойство `font` является агрегированным свойством, которое позволяет объединять все рассмотренные параметры шрифта в одном объявлении правила отображения. Синтаксис значения имеет следующий вид:

```
[ [ <'font-style'> || <'font-variant'> || <'font-weight'> ]?
↳<'font-size'> [ / <'line-height'> ]? <'font-family'> ] | caption |
↳icon | menu | message-box | small-caption | status-bar | inherit
```

Итак, как видно из объявления, мы можем использовать несколько вариантов задания значения свойства `font`. Например, указать просто триаду значений свойств `font-style`, `font-variant` и `font-weight`. Причем, если мы не обозначим явно какое-либо значение отдельного свойства, то анализатор CSS все равно его распознает и подставит значение, использующееся по умолчанию для данного свойства. В качестве примера можно рассмотреть следующую конструкцию:

```
P { font: 80% sans-serif }
```

В этом правиле мы указываем, что для текстового содержимого элемента `P` используется шрифт, размер которого составляет 80% от размера шрифта

родительского элемента, и выбран шрифт без засечек. То есть мы задаем значения свойств font-size и font-family. Для остальных свойств используются "умолчальные" значения.

Помимо уже известных нам свойств, значения которых мы можем придавать агрегатному свойству font, существует еще несколько предустановленных значений, регулирующих вид и начертание используемого шрифта.

Значение caption устанавливает шрифт, идентичный шрифту, применяемому для отображения наименований различных органов управления. Значение icon идентифицирует шрифт, которым отображаются подписи к пиктограммам. Значение menu создает шрифт, аналогичный используемому в выпадающих списках и контекстных меню. Для назначения шрифта, которым печатается информация в окнах сообщений, используется значение message-box. Ключевое слово small-caption задает шрифт, который применяется для заголовков маленьких органов управления. А значение status указывает на шрифт, которым отображается информация в строке состояния.

Для того чтобы увидеть выгоду применения агрегатного свойства font, достаточно рассмотреть два равноценных правила отображения текстового содержимого элемента P.

```
P { font: 600 9pt Charcoal }
P {
  font-style: normal;
  font-variant: normal;
  font-weight: 600;
  font-size: 9pt;
  line-height: normal;
  font-family: Charcoal
}
```

Ключевое слово inherit используется для наследования значения данного свойства от родительского элемента. Свойство применимо для всех элементов, имеющих текстовое содержимое.

На этом мы заканчиваем рассмотрение всех основных свойств, применимых при описании каскадных стилевых таблиц к шрифтовому оформлению текстового содержимого какого-либо элемента.

Свойства абзаца

Существуют и другие свойства, применимые к шрифтовому оформлению. Например, при помощи свойства text-indent можно задать отступ для первой строки каждого абзаца. Синтаксис возможных значений данного свойства описывается следующей конструкцией:

```
<length> | <percentage> | inherit
```

Из синтаксического описания видно, что отступ первой строки мы можем задавать как при помощи стандартных единиц измерения, так и с использованием процентных соотношений. Впрочем, отступ первой строки можно и унаследовать от родительского элемента, использовав для этого в качестве значения ключевое слово `inherit`. По умолчанию для данного свойства используется нулевое значение. При помощи следующей конструкции мы можем установить десятимиллиметровый отступ для первой строки абзаца:

```
P { text-indent: 10mm }
```

Выключка строк текста регулируется при помощи свойства `text-align`. Возможные значения этого свойства описываются следующей конструкцией:

```
left | right | center | justify | <string> | inherit
```

Значение `left` заставляет браузер прижимать строки к левому краю ячейки-контейнера. Значение `right` прижимает текст, что естественно, к правому краю. Для центрирования строк используется значение `center`. А значение `justify` растягивает текст на всю длину строки. Помимо этого мы можем использовать в качестве значения этого свойства ключевые слова, обозначенные в определении множеством `string`. Но они будут действовать только в том случае, если ячейка-контейнер является ячейкой таблицы. Ключевое слово `inherit` указывает, что значение этого свойства необходимо унаследовать от родительского элемента.

Свойство `text-decoration` позволяет установить дополнительные эффекты оформления текста. Синтаксис возможных свойств описывается следующим образом:

```
none | [ underline || overline || line-through || blink ] | inherit
```

Рассмотрим это выражение. Значение по умолчанию `none` оставляет отображаемый текст без изменений. Если же мы хотим каким-либо образом декорировать текст, следует использовать комбинацию специализированных значений. Значение `underline` заставляет браузер подчеркивать каждую строку абзаца текста, входящего в состав элемента, к которому мы и применяем данное правило оформления. Значение `overline` позволяет нам отобразить горизонтальную линию сверху строк текста. Перечеркнуть текст можно при помощи значения `line-through`. Впрочем, декорирование текста не ограничивается использованием горизонтальных линий. Мы можем заставить текст мигать. Для этого следует использовать значение `blink`. Ключевое слово `inherit` указывает на то, что значение данного свойства должно наследоваться от родительского элемента.

При помощи свойства `text-shadow` мы можем задавать параметры тени для нашего текста. Значения этого свойства описываются при помощи следующей конструкции:

```
none | [<color> || <length> <length> <length>? ,]* [<color> ||
↳<length> <length> <length>?] | inherit
```

Что касается ключевых слов `none` и `inherit`, то с ними все понятно. Значение по умолчанию `none` явно указывает, что символы текста отбрасывать тень не будут. Значение `inherit` указывает, что значение данного свойства должно быть унаследовано от родительского элемента. А вот об остальных способах задания значения свойства следует поговорить особо.

Как видно из синтаксического определения, для указания свойств тени мы должны задать обязательно цвет тени и, как минимум, два значения, указывающие на какое-то расстояние. Они регулируют, соответственно, горизонтальное и вертикальное смещение тени относительно текста. При этом положительные значения смещают тень вправо и вниз. Отрицательное значение для горизонтального смещения направляет тень влево от текста, а отрицательное вертикальное смещение — вверх от него. Третье числовое значение, как видно из определения синтаксиса, не является обязательным. Тем не менее, оно достаточно часто используется. Это значение позволяет задавать степень размытости тени. То есть на тень накладывается размывающий фильтр, известный под названием *gaussian blur*, аргументом-радиусом которого и служит третье значение.

Если внимательно всмотреться в синтаксическое определение возможных значений, станет видно, что мы можем задавать несколько вариантов теней. Они будут представлять собой наборы из указания цвета и смещения тени с опциональным коэффициентом размытия. Разделяться эти определения будут запятой. Таким образом может быть сформирован так называемый список определений тени. Эти определения будут порождать тени в том порядке, в котором они были перечислены в правиле оформления. При этом порожденные множественные тени могут перекрывать друг друга, но они не могут перекрывать текст и выходить за пределы ячейки-контейнера, содержащей этот текст.

В качестве примера мы можем рассмотреть следующее правило оформления для элемента с наименованием `text`:

```
text { text-shadow: 3px 3px red, yellow -3px 3px 2px, 3px -3px }
```

В этом примере мы создаем три тени. Первая располагается справа снизу от текста со смещением по обеим осям на три пиксела. При этом цвет первой порожденной тени — красный, и она не подвергается наложению размывающего фильтра. Вторая сгенерированная тень будет желтого цвета, и она будет смещена влево вниз относительно текста. К ней же будет применен эффект размытия с радиусом два пиксела. Третья тень будет смещена вправо и вверх относительно текста на три пиксела. Поскольку цвет для нее явно не указан, тень будет отображена тем же цветом, что и сам текст.

Ключевое слово `inherit` означает, что теневое оформление текста должно наследоваться от родительского элемента.

Межсимвольный просвет в тексте абзаца задается при помощи свойства `letter-spacing`. Его возможные значения описываются следующей конструкцией:

```
normal | <length> | inherit
```

Значение по умолчанию `normal` оставляет межсимвольный промежуток таким, какой используется браузером для отображения шрифта, применяемого для текстового наполнения данного элемента.

Мы можем также напрямую установить размер межсимвольного интервала, используя для этой цели стандартные единицы измерения CSS. Ключевое слово `inherit` означает, что значение для данного свойства необходимо взять такое же, как и у родительского элемента.

Помимо межсимвольного интервала, мы можем явно указывать и интервал между словами отображаемого текста. Для этого используется свойство `word-spacing`. Множество его возможных значений определяется следующим синтаксическим правилом:

```
normal | <length> | inherit
```

Как видно, синтаксическое определение значений полностью повторяет определение предыдущего свойства. Механизм их использования также одинаков, поэтому мы не станем повторяться и опустим дальнейшее рассмотрение.

При помощи свойства `text-transform` мы можем принудительно привести вид символов к одному из предустановленных типов. Синтаксис возможных значений определяется при помощи следующей конструкции:

```
capitalize | uppercase | lowercase | none | inherit
```

Значение `capitalize` заставляет браузер первую букву каждого слова принудительно переводить в верхний регистр. Значение `uppercase` преобразует все буквы к верхнему регистру. Значение `lowercase`, наоборот, все прописные буквы переводит в строчные. А используемое по умолчанию значение `none` оставляет текст без изменений. Если в качестве значения этого свойства использовать ключевое слово `inherit`, то значение данного свойства будет унаследовано от родительского элемента.

Тип используемого в тексте пробела может быть установлен при помощи свойства `white-space`. Его возможные значения описываются следующей синтаксической конструкцией:

```
normal | pre | nowrap | inherit
```

Значение по умолчанию `normal` оставляет пробелы обычными. Значение `pre` сворачивает (collapse) пробелы, и перенос текста на другую строку будет произведен только тогда, когда в исходном коде документа начнется новая строка. Значение `nowrap` преобразует обычные пробелы в неразрывные. Несомненно, пользоваться последними двумя значениями необходимо осторожно,

поскольку они блокируют естественный перенос текста на следующую строку, который необходимо выполнить в случае возможных изменений размеров окна просмотра браузера. Ключевое слово `inherit` указывает, что значение данного свойства будет унаследовано от родительского элемента.

Таблицы в CSS

Стандарт CSS level 2 поддерживает свою собственную модель отображения таблиц. Но прежде чем перейти к обзору соответствующих свойств CSS, необходимо дать определения тем объектам, к которым эти свойства применяются.

- Таблица (table) является специализированным элементом, который отображается в прямоугольной ячейке-контейнере и содержит специальным образом отформатированное содержимое.
- Строчная таблица (inline-table) является таблицей с одной строкой.
- Строка таблицы (table-row) представляет собой совокупность ячеек, находящихся в одном горизонтальном ряду.
- Группа строк (table-row-group) является элементом, в который входит одна или несколько строк таблицы.
- Группа наименований столбцов (table-header-group) обозначает элемент, похожий на обычную строку таблицы или группу строк, но этот элемент включает только те строки, которые содержат заголовки столбцов. Если быть совсем точным, то в этом элементе отображаются строки, расположенные ниже заголовка всей таблицы и до основных данных таблицы.
- Группа "подвала" таблицы (table-footer-group) также является частным случаем объекта "группа строк". Обычно этот объект содержит в себе строки с агрегатной информацией столбцов, то есть итоговыми данными. Располагается после ячеек с обычными данными и до нижнего заголовка таблицы, если такой, конечно, имеется.
- Колонка таблицы (table-column) является объектом, содержащим совокупность ячеек, расположенных в одном вертикальном ряду.
- Группа колонок таблицы (table-column-group) объединяет в себе одну или более колонок.
- Ячейка таблицы (table-cell) является основным объектом модели представления таблиц в CSS level 2. Содержит основной и неделимый элемент таблицы — клетку.
- Заголовок таблицы (table-caption) является объектом, заключающим в себе заголовок таблицы. Может отображаться как до таблицы, так и после нее, или по бокам.

Теперь перейдем к обзору свойств, регулирующих отображение табличных объектов.

Свойство caption-side позволяет явно устанавливать положение заголовка таблицы относительно ее тела. Возможные значения свойства описываются следующей синтаксической конструкцией:

```
top | bottom | left | right | inherit
```

Значение по умолчанию top указывает, что название таблицы необходимо поместить сверху таблицы. Значение bottom смещает заголовок в нижнюю часть, сразу после тела таблицы. Значения left и right заставляют браузер отображать наименование таблицы слева или справа от нее соответственно. Ключевое слово inherit указывает, что значение данного свойства необходимо унаследовать от родительского элемента.

Само собой, данное свойство применимо лишь к объектам, которые содержат заголовок таблицы (table-caption).

В качестве примера использования этого свойства можно привести следующее правило оформления элемента CAPTION.

```
CAPTION { caption-side: bottom;
          width: auto;
          text-align: left }
```

При применении этого правила заголовок таблицы будет отображаться после нее самой, при этом ширина ячейки-контейнера, содержащей заголовок, будет такой же, как ширина самой таблицы, что обеспечивается значением auto свойства width, а текст заголовка будет прижат к левому краю ячейки, на что указывает свойство text-align с приписанным ему значением left.

Необходимо отметить, что при размещении заголовка таблицы сбоку не следует использовать автоматический выбор ширины для его ячейки-контейнера, так как браузер будет стремиться записать заголовок в одну строку и сместит таблицу, освобождая пространство для заголовка. Чтобы избежать подобной ситуации, можно использовать код, подобный приведенному ниже:

```
BODY {
    margin-left: 8mm
}
TABLE {
    margin-left: auto;
    margin-right: auto
}
CAPTION {
    caption-side: left;
    margin-left: -8mm;
```

```
    width: 8mm;
    text-align: right;
    vertical-align: bottom
}
```

Если говорить кратко, то совокупность этих правил оформления создает левое поле размером восемь миллиметров, а в нем размещается заголовок таблицы.

Свойство `table-layout` позволяет явно задавать алгоритм раскладки таблицы. Для того чтобы получить представление о его действии, необходимо рассмотреть сначала порядок создания таблиц.

Вся таблица разбивается на слои. Самым первым, самым нижним слоем является слой, в котором формируется контейнер для таблицы в целом (table). Затем формируется слой, в котором находятся группы столбцов (column groups). Следом создается слой с одиночными столбцами (columns). На него накладывается слой с группами строк (row groups). А уже поверх него формируется слой, содержащий отдельные строки (rows). Самым последним, самым верхним слоем является слой, содержащий отдельные ячейки.

Вся эта схема достаточно наглядно показана на рис. 7.2.

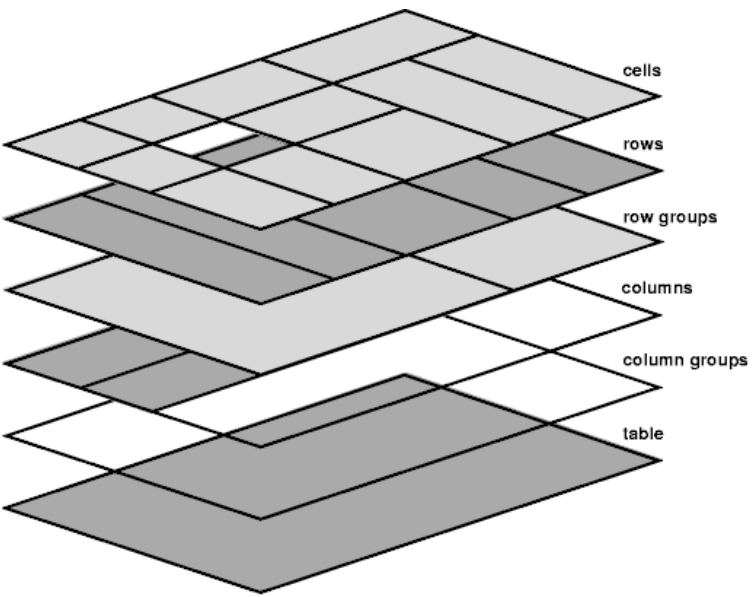


Рис. 7.2. Схема расположения слоев таблицы

Синтаксис возможных значений свойства `table-layout` задается при помощи следующей конструкции:

```
auto | fixed | inherit
```

Значение по умолчанию `auto` задает стандартный алгоритм раскладки таблицы. Он обычно требует не больше двух проходов исследования кода для браузера. Ширина всей таблицы задается как сумма ширин всех столбцов.

Сначала вычисляется минимальная ширина содержимого каждой ячейки с учетом разбиения содержимого на максимальное количество строк. Если для какой-либо ячейки при помощи свойства `width` значение ширины указано явно, то это значение и принимается за минимальную ширину содержимого. Если же для свойства `width` во всех случаях было выбрано значение `auto`, то вычисленное значение минимальной ширины содержимого закрепляется за результирующей ячейкой. Попутно вычисляется максимальная ширина ячейки, которая получается из ширины содержимого ячейки без разбиения его на несколько строк.

Затем определяются значения минимальной и максимальной ширины для каждого столбца. Для этого, естественно, выбираются минимальная и максимальная ширина из всего массива ячеек, входящих в данный столбец. Или используется свойство `width`, заданное для этого столбца, если его значение превышает полученные величины.

Далее для каждой ячейки, которые растянуты (span) на несколько столбцов, переопределяется максимальная и минимальная ширина столбцов, исходя из уже вычисленного значения ширины объединенной ячейки. Если общая ширина ячейки больше, чем сумма ширин столбцов, то переопределяется максимальная ширина затронутых столбцов.

После того, как браузер вычислил ширину для всех столбцов таблицы, необходимо вычислить ширину самой таблицы. Если таблица имеет явно указанную ширину (использовано свойство `width`), то браузер должен сравнить это явно установленное значение с суммой минимальных ширин всех столбцов. При этом для реальной ширины таблицы выбирается максимальная из этих двух величин.

Если же для свойства `width`, примененного к таблице в целом, установлено свойство `auto`, напрямую используется сумма минимальных ширин всех столбцов, входящих в таблицу.

Естественно, полная ширина таблицы зависит еще и от полей, отступов и толщины границ элементов таблицы.

Значение `fixed` свойства `table-layout` заставляет браузер использовать более быстрый алгоритм. При этом ширина таблицы не зависит от ширины содержимого ячеек. При расчете используются значения ширин таблицы и столбцов, а также величина отступов и толщина границ ячеек.

Сначала просматриваются все элементы, содержащие отдельные колонки таблицы. Если для них напрямую задано значение свойства `width`, то оно и используется в качестве ширины колонки. Для тех колонок, у которых значение этого свойства не задано явно, просматривается верхняя ячейка. Если

эта линейка имеет явно заданное значение свойства `width`, то оно присваивается и самому столбцу. Если попадается растянутая (span) на несколько столбцов ячейка, то ее ширина делится на количество столбцов, которые она объединяет, и каждому столбцу приписывается соответствующее значение. Для остальных столбцов поровну делится оставшееся доступное по горизонтали пространство. При этом учитывается место, необходимое для отображения отступов и границ ячеек.

Для получения ширины всей таблицы сравнивается значение свойства `width`, если оно для нее задано, и суммы вычисленных ширин столбцов, включающих в себя отступы и границы ячеек. Используется максимальное значение.

Ключевое слово `inherit` означает, что вариант вычисления ширины таблицы наследуется от родительского элемента.

Теперь перейдем к рассмотрению свойств, регулирующих отображение границ.

Свойство `border-collapse` устанавливает модель отображения границ. Синтаксис возможных значений этого свойства определяется следующим образом:

```
collapse | separate | inherit
```

Значение по умолчанию `collapse` задает модель объединенных границ. При этом границы соседствующих ячеек фактически сливаются в одну общую границу, и браузер рассчитывает размеры таблицы или табличного объекта, исходя из этой установки.

Значение `separate`, напротив, не позволяет границам сливаться. В этом случае каждая граница учитывается отдельно.

Ключевое слово `inherit` указывает, что значение этого свойства необходимо унаследовать от родительского элемента.

Рассматриваемое свойство может применяться только к табличным объектам.

Свойство `border-spacing` позволяет задавать отступы между границами соседствующих ячеек таблицы. Естественно, это свойство будет иметь силу только в том случае, если применяется раздельная модель границ табличных элементов (значение `separate` свойства `border-collapse`). Синтаксис его значений задается при помощи следующей конструкции:

```
<length> <length>? | inherit
```

Итак, как видно из определения, для задания размера пустого пространства между границами соседних ячеек мы можем использовать одно или два значения. Если мы укажем только один отступ, то он будет использоваться для всех границ. Если же мы укажем два значения, то первое из них будет специфицировать расстояние между границами по горизонтали, а второе — по вертикали. Данные значения, естественно, не могут быть отрицательными.

Если в качестве значения этого свойства мы используем ключевое слово `inherit`, то размеры отступов между границами соседствующих ячеек будут унаследованы от родительского табличного элемента.

Отображением границ вокруг пустых ячеек управляет свойство `empty-cells`. Это свойство, как и предыдущее, имеет смысл использовать только в тех случаях, когда для табличного элемента установлена раздельная модель границ. Ячейки считаются *пустыми*, если для них свойство `visibility` имеет значение `hidden`, или же если в них находится один из неотображаемых символов. Неотображаемыми символами считаются символы пробела и неразрывного пробела, табуляции, и два символа форматирования с кодами `10` (перевод строки) и `13` (возврат каретки). (Шестнадцатеричные коды — `0A` и `0D` соответственно.)

Синтаксис возможных значений этого свойства определен следующим образом:

```
show | hide | inherit
```

Значение по умолчанию `show` заставляет браузер отображать границы вокруг пустых ячеек. Значение `hide` скрывает эти границы, делает их невидимыми. Однако в обоих случаях сами границы всегда существуют. Ключевое слово `inherit` заставляет наследовать значение этого свойства от родительского элемента.

Необходимо отметить, что рассматриваемое нами свойство может применяться только к элементам, которые являются ячейками таблиц (`table-cell`).

На этом мы заканчиваем рассмотрение табличной модели CSS level 2 и переходим к заключительному разделу обзора технологии каскадных стилевых таблиц.

Дополнительные свойства

Нам осталось рассмотреть всего несколько свойств CSS level 2, которые относятся к категории пользовательского интерфейса.

При помощи свойства `cursor` мы можем регулировать внешний вид курсора мыши при прохождении его над ячейкой-контейнером, в которой отображается содержимое данного элемента. Синтаксис возможных значений этого свойства определяется следующей конструкцией:

```
[ [<uri> ,]* [ auto | crosshair | default | pointer | move |
↳e-resize | ne-resize | nw-resize | n-resize | se-resize |
↳swresize | s-resize | w-resize| text | wait | help ] ] | inherit
```

Как видно, мы можем указать несколько URI, ссылающихся на файлы с курсорами, или воспользоваться одним из предустановленных видов. Выбор уже установленного в системе курсора производится при помощи одного из нижеперечисленных ключевых слов.

Значение по умолчанию `auto` указывает, что будет использоваться стандартный курсор, установленный для данного элемента. Конкретный его вид определяется браузером.

Значение `crosshair` при появлении курсора мыши в ячейке-контейнере придает ему вид обычного перекрестья. В этом виде курсор больше всего похож на знак плюса.

Значение `default` заставляет браузер использовать курсор мыши, установленный в системе по умолчанию. Для Windows это обычная наклонная стрелка.

При помощи ключевого слова `pointer` мы имеем возможность при попадании курсора мыши на территорию элемента придавать ему форму указателя, как для обычных гиперссылок в HTML. Чаще всего этот курсор выглядит как рука с вытянутым указательным пальцем. Но конкретный вид курсора в каждом случае зависит от установок операционной системы.

Значение `move` создает курсор, используемый обычно при перемещении каких-либо объектов или окон по экрану. Мы привыкли видеть его в виде совокупности из четырех стрелок, направленных в разные стороны от общего центра.

Ключевые слова `e-resize`, `ne-resize`, `nw-resize`, `n-resize`, `se-resize`, `sw-resize`, `s-resize` и `w-resize` создают курсоры, применяемые в тех случаях, когда изменяются какие-либо размеры окна. Для Windows это обычно самые различные двунаправленные стрелки. Так, например, значение `se-resize` создает курсор, возникающий при перетаскивании нижнего левого угла окна.

Значение `text` заставляет браузер использовать курсор мыши, применяемый при наборе текста. Чаще всего это вертикальная черта с двумя засечками.

Для отображения курсора мыши, указывающего на процесс ожидания в тот момент, когда система занята, используется значение `wait`. Чаще всего подобный курсор выглядит как часы, обычные или песочные.

Ключевое слово `help`, используемое в виде значения данного свойства, заставляет браузер отображать курсор мыши, применяемый при запросе оперативной справки. Чаще всего подобный курсор выглядит как стрелка, совмещенная с вопросительным знаком.

Ключевое слово `inherit` указывает на то, что значение этого свойства будет таким же, как у элемента-родителя.

В качестве примера использования свойства `cursor` можно привести следующее правило оформления:

```
P { cursor : url("mything.cur"), url("second.csr"), text; }
```

Здесь для элемента с наименованием `P` мы задали три возможных типа курсора мыши. При этом доступ к ним осуществляется по очереди. Если не удается загрузить курсор из файла mything.cur, то браузер пытается отыскать файл second.csr, а если и тот не найден, использует обычный текстовый курсор.

Иногда может возникнуть необходимость помимо границ или вместе с ними отобразить еще и линии обрамления элемента (outlines). Обрамляющие линии отличаются от границ тем, что не занимают лишнего пространства и могут не иметь стандартную прямоугольную форму. Регулируется внешний вид обрамляющих линий при помощи нескольких свойств.

Свойство `outline-width` предназначено для регулировки толщины линий обрамления. Список его возможных значений задается следующим образом:

```
<border-width> | inherit
```

Как видно, по своему синтаксису это свойство идентично уже рассматривавшемуся нами свойству `border-width`.

Стиль окаймляющих линий задается при помощи значения свойства `outline-style`. Возможные значения этого свойства описываются следующей конструкцией:

```
<border-style> | inherit
```

Здесь в качестве значений мы можем использовать все ключевые слова, применяемые для задания стиля границы, за исключением ключевого слова `hidden`, так как нет никакого смысла создавать невидимые обрамляющие линии.

При помощи свойства `outline-color` мы имеем возможность устанавливать цвет обрамляющих линий. Синтаксическое определение возможных значений этого свойства выглядит следующим образом:

```
<color> | invert | inherit
```

Как видно, помимо обычных способов задания цвета, мы можем использовать в качестве значения свойства ключевое слово `invert`. Оно заставляет браузер отображать обрамляющие линии инверсным цветом по отношению к фону, на котором они прорисовываются.

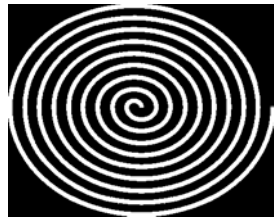
Агрегатное свойство `outline` позволяет собирать все три рассмотренных нами только что свойства в единое целое. Синтаксис значений этого свойства определяется следующей конструкцией:

```
[ <'outline-color'> || <'outline-style'> || <'outline-width'> ] |
inherit
```

Необходимо отметить, что все обрамляющие линии отображаются всегда "поверх" ячейки-контейнера, то есть их отображение считается приоритетным. Это значит, что все границы и иные элементы оформления находятся на один слой ниже.

На этом мы заканчиваем обзор технологии CSS level 2 и переходим к более "продвинутой" технологии оформления, которая предназначена специально для XML.

Глава 8



Стилевой язык XSL

История

Рассмотренная нами технология стилевых таблиц CSS достаточно мощна, но она относится все-таки к HTML. Основа его в том, что там заблаговременно предопределено, какой тэг что обозначает. Есть специализированные тэги для таблиц, есть специализированные тэги для графических изображений, и так далее. Поэтому применение CSS позволяло достаточно эффективно управлять отображением HTML-документа.

В XML мы не знаем заранее, каким элементом форматирования является тот или иной элемент. По его имени мы не можем сказать, что это: таблица, текстовый абзац, рисунок или гиперссылка. Именно эти соображения и ограничивают применение CSS для XML. Нет, CSS конечно можно использовать для оформления XML-документов, как мы видели в предыдущей главе, но при этом мы можем немного. Например, к двум элементам с одинаковыми наименованиями мы не в состоянии применить различные правила оформления без каких-либо ухищрений.

Требовался иной язык разметки, который позволил бы помимо установки неких свойств элемента еще и указывать, каким объектом именно является этот элемент. И такой язык был создан.

В начале книги мы уже говорили о связи XML с SGML. В SGML был свой стилевой язык. Назывался он DSSSL (Document Style Semantics and Specification Language) — язык семантики и спецификации стиля документа. Подобно языку SGML он был очень объемен. По этой причине для нужд XML-документов было выделено подмножество DSSSL, которое получило название DSSSL-Online, или сокращенно DSSSL-O. А затем уже на его основе был создан более универсальный и гибкий язык XSL (eXtensible Stylesheet Language), который и является сейчас официальным стилевым языком для XML.

Синтаксис и подключение XSL

Сама стилевая таблица XSL является XML-документом, поэтому на нее распространяются все правила создания XML-документов. С тем отличием, что для нее не требуется создавать DTD-блок. Вместо этого мы можем исполь-

зовать стандартные пространства имен. Общая схема XSL-файла обычно выглядит так:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fo="http://www.w3.org/1999/XSL/Format"
                version='1.0'>
…Объявление правил форматирования…
</xsl:stylesheet>
```

Первая строка служит объявлением XML-документа. Следующий тэг объявляет, что данный документ является стилевой таблицей XSL, указывает версию применяемого стандарта XSL и определяет применяемые пространства имен.

В общем случае, в XSL-файлах применяется обычно два пространства имен. Пространство имен `xsl` расположено по адресу `http://www.w3.org/1999/XSL/Transform`, и содержит объявления стандартных конструкций XSL. Пространство имен `fo` находится по адресу `http://www.w3.org/1999/XSL/Format` и включает декларации всех объектов форматирования со всеми их свойствами.

Сама же стилевая таблица состоит из перечисления правил отображения того или иного элемента, подобно CSS. Только в XSL все делается при помощи так называемых *шаблонов*. Рассмотрим пример некоего определения:

```
<xsl:template match="p">
  <fo:block>
    <fo:initial-property-set font-variant="small-caps"/>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Как видно, сначала мы объявляем шаблон при помощи тэга элемента `template`, относящегося к пространству имен `xsl`. При этом в значении атрибута `match` мы указываем наименование элемента искомого XML-документа, к которому мы будем применять правило оформления. В нашем случае мы создаем правило оформления для элемента с наименованием `p`.

Затем мы должны указать, каким именно элементом форматирования будет являться данный элемент `p`. Для этого мы применяем тэг элемента `block` из пространства имен `fo`. А в качестве содержимого этого объекта мы используем тэг, устанавливающий значение одного из свойств этого объекта форматирования, и инструкцию применения созданного шаблона `<xsl:apply-templates/>`.

Аббревиатура `fo`, являющаяся одновременно стандартным префиксом пространства имен объектов форматирования, часто применяется в XSL. Она расшифровывается как flow object, что в русскоязычной литературе часто

переводится как "объекты потока" или "растекающиеся объекты". К сожалению, эти кальки, т. е. дословные переводы, плохо ложатся на русский язык и неправильно отражают специфику данных объектов. Скорее, это все-таки будут объекты форматирования, поскольку сама идентификация объекта (таблицы, рисунка, абзаца) уже определенным образом указывает на порядок его отображения и форматирования.

Итак, что нам нужно знать для создания XSL-файлов? Нам надо знать, как их открывать, как создавать правила, какие бывают объекты форматирования, и какие у этих объектов есть свойства. Вот этим мы и займемся в следующих разделах данной главы.

Объекты форматирования

Все объекты форматирования в XSL образуют некую иерархию. Объектную иерархию. Это вполне естественно. Ведь ячейка таблицы должна входить в состав строки таблицы, а та — в саму таблицу. Поэтому мы будем стараться указывать возможные отношения между объектами форматирования.

Начнем с самого начала, то есть — с корня.

Объект `fo:root` является абстрактным объектом, находящимся на самой вершине объектной иерархии XSL. Он может включать в себя: объекты `fo:layout-master-set`, который содержит все основные объекты, управляющие раскладкой содержимого документа в области просмотра, опциональный объект `fo:declaration`, регулирующий цветовые профили для данной стилевой таблицы, и серию объектов `fo:page-sequence`, представляющих собой последовательности страниц. Наполнение абстрактного корневого объекта форматирования объявляется следующим образом:

```
(layout-master-set,declarations?,page-sequence+)
```

Так как рассматриваемый нами объект по сути своей является абстрактным, то и набор свойств, применимых к нему, не так уж велик. Мы можем использовать всего одно свойство `media-usage`.

Объект `fo:declarations` предназначен для задания некоей глобальной информации об отображаемом XML-документе. На деле он может содержать несколько объектов `fo:color-profile`, указывающих на используемые цветовые профили (по сути, цветовые палитры), применяемые к данному документу. Объект не имеет никаких свойств, и, по существу, является обычным контейнером.

Объект `fo:color-profile` служит для задания цветового профиля, применяемого при отображении данного XML-документа. Конкретная информация о применяемом цветовом профиле задается при помощи свойств `src`, `color-profile-name` и `rendering-intent`.

Объект `fo:page-sequence` задает отображение XML-документа в виде последовательности страниц. При этом содержимое документа разбивается на отдельные страницы и генерируются две области отображения информации, так как страницы разделяются по книжному образцу — на левые и правые. Данный объект может содержать дочерние объекты. Содержание рассматриваемого элемента объявляется следующим образом:

```
(title?,static-content*,flow)
```

К этому объекту могут быть применены свойства `country`, `format`, `language`, `letter-value`, `grouping-separator`, `grouping-size`, `id`, `initial-page-number`, `force-page-count` и `master-name`.

Объект `fo:layout-master-set` служит контейнером для объектов форматирования, управляющих раскладкой содержимого XML-документа в областях отображения. Содержимое этого элемента определяется так:

```
(simple-page-master|page-sequence-master)+
```

Так как объект является контейнером в чистом виде, то и свойств для него не предусмотрено.

Объект `fo:page-sequence-master` предназначен для организации страничной структуры отображения документа. Он позволяет создавать в качестве своего содержимого комбинации из различных последовательностей страниц. Его содержимое декларируется следующим образом:

```
(single-page-master-reference|repeatable-page-master-
↳reference|repeatable-page-master-alternatives)+
```

К данному объекту может применяться свойство `master-name`.

Объект `fo:single-page-master-reference` является ссылкой на одиночную мастер-страницу, такую как титульная, или страница с началом главы. То есть это страница, которая чем-либо отличается от обычных страниц. К этому объекту мы можем применить свойство `master-name`.

Объект `fo:repeatable-page-master-reference` позволяет создавать ссылку на последовательность повторяющихся мастер-страниц. Изначально количество повторений неограничено, но мы можем задать их конкретное количество. Данный объект обладает свойствами `master-name` и `maximum-repeats`.

Для подобных серий мастер-страниц мы можем задавать альтернативное представление. Выбор конкретного варианта осуществляется браузером. Объявление альтернативной последовательности производится при помощи элемента `fo:repeatable-page-master-alternatives`. В качестве содержимого данного объекта форматирования может использоваться серия элементов `conditional-page-master-reference`. Объект обладает свойством `maximum-repeats`.

Объект fo:conditional-page-master-reference указывает на отдельную страницу с наложенными на нее некоторыми условиями. Эти условия определяются при помощи свойств page-position, odd-or-even, blank-or-not-blank. Также может быть использовано свойство master-name.

Объект fo:simple-page-master предназначен для управления отображением одиночной страницы и геометрией области отображения. Имеет некое содержимое, объявленное следующим образом:

```
(region-body,region-before?,region-after?,region-start?,region-end?)
```

К данному объекту применимы свойства master-name, page-width, page-height, reference-orientation, writing-mode и стандартные свойства, управляющие полями.

Объект fo:region-body управляет отображением содержимым страницы, объявленной при помощи объекта fo:simple-page-master. Определяет область отображения содержимого страницы. К объекту применимы все свойства, управляющие полями, границами, отступами и фоном, а также свойства clip, column-count, column-gap, display-align, overflow, region-name, reference-orientation и writing-mode.

Объект fo:region-before регулирует отображение пространства перед текстом, который управляется объектом region-body. К рассматриваемому объекту форматирования применяются общие свойства управления границами, отступами и фоном, а также свойствами clip, display-align, extent, overflow, precedence, region-name, reference-orientation и writing-mode.

Объект fo:region-after позволяет настраивать отображение пространства, расположенного после текста region-body. Обладает тем же набором свойств, что и предыдущий рассмотренный объект форматирования.

Объект fo:region-start применяется для страничных блоков region-body. Он управляет оформлением начальной области блока. При стандартном порядке отображения текста (слева направо) этот регион находится у левой границы области просмотра. Объект обладает общими свойствами границ, отступов и фона, а также свойствами clip, display-align, extent, overflow, region-name, reference-orientation и writing-mode.

Объект fo:region-end составляет пару с только что рассмотренным объектом форматирования. Предназначен для управления отображением конечного региона области просмотра текста. Обладает теми же свойствами, что и предыдущий рассмотренный объект.

Объект fo:flow является еще одним абстрактным объектом, служащим контейнером для последовательности блочных элементов. К нему применимо всего одно свойство flow-name.

Объект fo:static-content также является контейнером для блочных элементов форматирования. Но данный объект имеет конкретное предназначение.

Он позволяет регулировать оформление повторяющихся элементов страниц, таких как верхние и нижние колонтитулы. Обладает свойством `flow-name`.

Объект `fo:title` предназначен для управления представлением элементов, которые являются заголовками. Содержимое таких элементов может быть либо текстовым, либо состоять из однострочных блоков. Данный объект обладает стандартными свойствами доступа, границ, отступа, полей, фона и шрифта, а также свойствами `baseline-shift`, `color`, `line-height`, `line-height-shift-adjustment`, `visibility` и `z-index`.

Объект форматирования `fo:block` обычно применяется для управления отображением абзацев, заголовков, названий рисунков и таблиц, и подобных элементов. К данному объекту применимы стандартные свойства доступа, границ, отступа, полей, фона, шрифта и относительного позиционирования, а также свойства:
`break-after`, `break-before`, `color`, `text-depth`, `text-altitude`, `hyphenation-keep`, `hyphenation-ladder-count`, `id`, `keep-together`, `keep-with-text`, `last-line-and-indent`, `linefeed-treatment`, `line-height`, `line-height-shift-adjustment`, `line-stacking-strategy`, `orphans`, `space-treatment`, `span`, `text-align`, `text-align-last`, `visibility`, `white-space-collapse`, `widows`, `wrap-option`, `z-index`.

Объект `fo:block-container` содержит в себе некоторое количество объектов `fo:block` и генерирует соответствующие области отображения. Обычно используется для совместной обработки текстовых блоков, у которых свойство `writing-mode` имеет различные значения. К объекту применимы общие свойства абсолютного позиционирования, границ, отступов, полей и фона, а также свойства:
`block-progression-dimension`, `break-after`, `break-before`, `clip`, `display-align`, `height`, `id`, `inline-progression-dimension`, `keep-together`, `keep-with-text`, `keep-with-previous`, `overflow`, `reference-orientation`, `span`, `width`, `writing-mode`.

Объект `fo:bidi-override` применяется для оформления тех текстовых объектов, для которых необходимо принудительно отменить действие стандартного алгоритма отображения двунаправленного текста Unicode. Обладает общими свойствами шрифтов и относительного позиционирования, а также свойствами:
`color`, `direction`, `id`, `letter-spacing`, `line-height`, `line-height-shift-adjustment`, `score-spaces`, `text-shadow`, `text-transform`, `unicode-bidi`, `word-spacing`.

Объект `fo:character` позволяет управлять отображением отдельного символа. К данному объекту форматирования применимы общие свойства границ, отступов, полей, фона, шрифта, относительного позиционирования и верстки, а также свойства:
`alignment-adjust`, `treat-as-world-space`, `alignment-baseline`, `baseline-shift`, `character`, `color`, `dominant-baseline`, `text-depth`, `text-altitude`, `glyph-orientation-horizontal`, `glyph-orientation-vertical`, `id`, `keep-with-next`, `keep-with-previous`, `letter-spacing`, `line-height`,

line-height-shift-adjustment, score-spaces, suppress-at-line-break, text-decoration, text-shadow, text-transform, word-spacing.

Объект fo:initial-property-set управляет отображением первой строки текста, помещенного в объект fo:block. Обладает общими свойствами доступа, отступов, границ, фона и относительного позиционирования, а также свойствами:
color, id, letter-spacing, line-height, line-height-shift-adjustment, score-spaces, text-decoration, text-shadow, text-transform, word-spacing.

Объект fo:external-graphic предназначен для управления отображением графического рисунка, который не входит в стандартную иерархию отображаемых объектов. Такой рисунок может по своим размерам выходить за пределы установленных страниц и областей отображения. К объекту применимы общие свойства доступа, границ, отступов, фона, полей, относительного позиционирования, а также свойства:
alignment-adjust, alignment-baseline, baseline-shift, block-progression-dimension, content-height, content-type, content-width, display-align, dominant-baseline, height, id, inline-progression-dimension, keep-with-next, keep-with-previous, line-height, line-height-shift-adjustment, overflow, scaling, scaling-method, src, text-align, width.

Объект fo:instream-foreign-object очень похож на предыдущий рассмотренный нами объект форматирования. Управляет отображением графики, встроенной в компактный (inline) блок. Обладает общими свойствами доступа, границ, фона, отступов, полей, относительного позиционирования, а также свойствами:
alignment-adjust, alignment-baseline, baseline-shift, block-progression-dimension, content-height, content-type, content-width, dominant-baseline, height, id, inline-progression-dimension, keep-with-next, keep-with-previous, line-height, line-height-shift-adjustment, overflow, scaling, scaling-method, text-align, width.

Объект fo:inline является одним из наиболее часто используемых объектов форматирования текста. Позволяет управлять отображением текстового фрагмента с фоном и/или обрамленного границами. Генерирует одну или несколько строковых областей отображения. Обладает общими свойствами доступа, границ, отступов, полей, фона, шрифта и относительного позиционирования, а также свойствами:
alignment-adjust, alignment-baseline, baseline-shift, color, dominant-baseline, id, keep-together, keep-with-next, keep-with-previous, line-height, line-height-shift-adjustment, text-decoration, visibility, z-index.

Объект fo:inline-container в качестве своего содержимого позволяет использовать несколько разнородных текстовых блоков, например таких, ко-

торые имеют различное значение свойства writing-mode. Данный объект форматирования обладает общими свойствами границ, отступов, полей, фона и относительного позиционирования, а также свойствами:
alignment-adjust, alignment-baseline, baseline-shift, block-progression-dimension, clip, display-align, dominant-baseline, height, id, inline-progression-dimension, keep-together, keep-with-next, keep-with-previous, line-height, line-height-shift-adjustment, overflow, reference-orientation, width, writing-mode.

Объект fo:leader предназначен для отображения начальных маркеров в таблицах содержаний, горизонтальных разделителей и прочих подобных объектов, которые предваряют основное содержимое. К объекту применимы общие свойства доступа, полей, отступов, границ, фона, шрифта и относительного позиционирования, а также свойства:
alignment-adjust, alignment-baseline, baseline-shift, color, dominant-baseline, text-depth, text-altitude, id, leader-alignment, leader-length, leader-pattern, leader-pattern-width, rule-style, rule-thickness, letter-spacing, line-height, line-height-shift-adjustment, text-shadow, visibility, word-spacing, z-index.

Объект fo:page-number применяется только в случае разбиения содержимого XML-документа на страницы и позволяет настраивать внешний вид номера страницы. Обладает общими свойствами доступа, границ, отступов, полей, фона, шрифта и относительного позиционирования, а также свойствами:
alignment-adjust, alignment-baseline, baseline-shift, dominant-baseline, id, keep-with-next, keep-with-previous, letter-spacing, line-height, line-height-shift-adjustment, score-spaces, text-decoration, text-shadow, text-transform, word-spacing.

Объект fo:page-number-citation регулирует отображение ссылок на номера страниц. Обладает общими свойствами доступа, границ, отступов, полей, фона, шрифта и относительного позиционирования, а также свойствами:
alignment-adjust, alignment-baseline, baseline-shift, dominant-baseline, id, keep-with-next, keep-with-previous, letter-spacing, line-height, line-height-shift-adjustment, ref-id, score-spaces, text-decoration, text-shadow, text-transform, word-spacing.

Объект fo:table-and-caption управляет отображением таблицы и ее заголовка вместе, как единым целым. В его содержимом обязательно присутствует один объект fo:table, и, хотя бы один объект fo:table-caption. Данный объект обладает общими свойствами доступа, полей, отступов, границ, фона и относительного позиционирования, а также свойствами caption-side, id, keep-together, keep-with-next, keep-with-previous.

Объект fo:table является основным объектом форматирования, управляющим отображением таблиц. Включает в себя остальные табличные объекты. Его содержимое определяется так:

```
(table-column*,table-header?,table-footer?,table-body+)
```

Данный объект обладает общими свойствами доступа, полей, отступов, границ, фона и относительного позиционирования, а также свойствами: `block-progression-dimension`, `border-after-precedence`, `border-before-precedence`, `border-collapse`, `border-end-precedence`, `border-separation`, `border-start-precedence`, `break-after`, `break-before`, `id`, `inline-progression-dimension`, `height`, `keep-together`, `keep-with-next`, `keep-with-previous`, `table-layout`, `table-omit-footer-at-break`, `table-omit-header-at-break`, `width`, `writing-mode`.

Объект форматирования `fo:table-column` позволяет описывать столбцы таблиц. Он обладает общими свойствами фона, а также свойствами `column-number`, `column-width`, `number-columns-repeated`, `number-columns-spanned`, `visibility`.

Объект `fo:table-caption` создает заголовок таблицы. К объекту применимы общие свойства доступа, отступов, границ, фона и относительного позиционирования, а также свойство `id`.

Объект `fo:table-footer` позволяет управлять отображением подвала таблицы. Так как любой подвал таблицы, ее итоговая часть, все равно принадлежит самой таблице, то этот объект не может обойтись без содержимого в виде либо строк, либо ячеек таблицы. Это хорошо видно в следующем синтаксическом определении содержимого данного объекта:

```
(table-row+|table-cell+)
```

К объекту применимы общие свойства доступа, отступов, границ, фона и относительного позиционирования, а также свойство `id`.

Объект `fo:table-body` управляет отображением содержимого таблицы. Таблица, по определению, разбивается на строки и ячейки, поэтому синтаксис описания данного объекта форматирования задается так:

```
(table-row+|table-cell+)
```

Как и предыдущие объекты, обладает общими свойствами доступа, отступов, границ, фона и относительного позиционирования, а также свойством `id`.

Объект `fo:table-row` реализует отдельную строку таблицы. Состоит из непустой последовательности объектов `table-cell`. Обладает общими свойствами доступа, границ, фона, отступов и относительного позиционирования, а также свойствами `block-progression-dimension`, `break-after`, `break-before`, `id`, `height`, `keep-together`, `keep-with-next`, `keep-with-previous`.

Объект `fo:table-cell` управляет отображением содержимого отдельных ячеек таблицы. Обладает общими свойствами доступа, границ, фона, отступов и относительного позиционирования, а также свойствами: `border-after-precedence`, `border-before-precedence`, `border-end-precedence`, `border-start-precedence`, `block-progression-dimension`, `column-number`, `display-align`, `relative-align`, `empty-cells`, `ends-row`, `height`, `id`, `number-columns-spanned`, `number-rows-spanned`, `starts-row`, `width`.

Объект fo:list-block позволяет управлять отображением информации, организованной в виде списка. Содержание его состоит из последовательности элементов list-item. Обладает общими свойствами доступа, полей, отступов, границ, фона и относительного позиционирования, а также свойствами: break-after, break-before, id, keep-together, keep-with-next, keep-with-previous, provisional-distance-between-starts, provisional-label-separation.

Объект fo:list-item управляет отображением элемента списка и его маркера. Содержимое данного объекта объявляется следующим образом:

```
(list-item-label,list-item-body)
```

То есть в его состав входят по отдельности тело элемента списка и метка-маркер последнего.

Данный объект обладает общими свойствами доступа, полей, границ, отступов, фона и относительного позиционирования, а также свойствами break-after, break-before, id, keep-together, keep-with-next, keep-with-previous, relative-align.

Объект форматирования fo:list-item-body предназначен для отображения содержимого отдельного элемента списка. К нему применимы общие свойства доступа, а также свойства id и keep-together.

Объект fo:list-item-label, как нетрудно догадаться, управляет отображением метки-маркера (в том числе и цифровой) элемента списка. Обладает точно таким же набором свойств, что и предыдущий рассмотренный элемент.

Объект fo:basic-link осуществляет настройки отображения обычных однонаправленных гиперссылок. К нему применимы общие свойства доступа, границ, полей, фона, отступов и относительного позиционирования, а также свойства: alignment-adjust, alignment-baseline, baseline-shift, destination-placement-offset, dominant-baseline, external-destination, id, indicate-destination, internal-destination, keep-together, keep-with-next, keep-with-previous, line-height, line-height-shift-adjustment, show-destination, target-processing-context, target-presentation-context, target-stylesheet.

Объект fo:multi-switch является контейнером для объектов типа multi-case. Предназначен для создания органов управления с множественным выбором. Обладает общими свойствами доступа, а также свойствами auto-restore и id.

Объект fo:multi-case является одним из элементов, подверженных действию множественного выбора. Обладает общими свойствами доступа, а также свойствами id, starting-state, case-name, case-title.

Объект `fo:multi-toggle` обычно используется для создания переключателей, которые меняют свой внешний вид при их выборе (например, по щелчку мыши). Все экземпляры этого объекта являются элементами множественного выбора. Объект обладает общими свойствами доступа, а также свойствами `id` и `switch-to`.

Объект `fo:multi-properties` используется для переключения между двумя или более наборами установленных свойств. Его содержимое определяется так:

```
(multi-property-set+,wrapper)
```

Обладает общими свойствами доступа и свойством `id`.

Объект `fo:multi-properti-set` предназначен для указания альтернативного набора свойств отображения какого-либо объекта. Обладает свойствами `id` и `active-state`.

Объект `fo:float` применяется для управления внешним видом элементов XML-документа, которые могут выходить за пределы стандартных границ, таких как, например, графические изображения. Обладает свойствами `float` и `clear`.

Объект `fo:footnote` регулирует внешний вид нижнего колонтитула страниц. Является контейнером для двух других элементов, которые детализируют составные части нижних колонтитулов.

Объект `fo:footnote-body` представляет содержимое нижнего колонтитула страницы.

Объект `fo:wrapper` используется для обозначения свойств группы объектов форматирования. Обладает только свойством `id`.

Объект `fo:marker` обычно используется в паре с объектом `fo:retrieve-marker`. Вместе они применяются для создания специализированных колонтитулов страниц. В качестве примера можно привести стандартный колонтитул словарей, который состоит из первого и последнего слова, находящихся на данной странице. Обладает свойством `marker-class-name`.

Объект `fo:retrieve-marker` является парным объектом для только что рассмотренного объекта форматирования. Обладает свойствами `retrieve-class-name`, `retrieve-position`, `retrieve-boundary`.

Мы рассмотрели все объекты форматирования языка XSL. Осталось разобраться только со свойствами. Им отведен следующий, последний раздел этой главы.

Свойства

Каждый объект форматирования языка XSL обладает своим собственным набором свойств, значения которых и определяют порядок отображения данного объекта. Ниже мы рассмотрим эти свойства. Они очень похожи на

свойства CSS, поэтому мы не будем еще раз останавливаться на правилах чтения определений их значений и разъяснении значений уже известных ключевых слов.

Начнем мы с общих свойств доступа.

Свойство `source-document` применяется для указания источника данных для включаемых элементов (таких как графические изображения). Возможные обозначения данного свойства определяются так:

```
<uri-specification> [<uri-specification>]* | none | inherit
```

По умолчанию используется значение `none`.

Свойство `role` позволяет указывать семантический идентификатор для объекта форматирования. Этот идентификатор описывается в виде обычной текстовой строки или URI, указывающего на нее. Возможные значения данного свойства обозначаются следующим образом:

```
<string> | <uri-specification> | none | inherit
```

По умолчанию используется значение `none`.

Теперь перейдем к общим свойствам абсолютного позиционирования.

Обобщенное свойство `absolute-position` позаимствовано из CSS level 2. Оно позволяет регулировать настройки абсолютного позиционирования для того или иного объекта форматирования. Его значения описываются следующим образом:

```
auto | absolute | fixed | inherit
```

Значение по умолчанию `auto` указывает, что абсолютное позиционирование для данного элемента форматирования не будет производиться. Он будет размещен в области просмотра по правилам относительного позиционирования. А значение `absolute` включает механизм абсолютного позиционирования, и координаты объекта будут указаны при помощи других свойств абсолютного позиционирования. Действие остальных значений нам известно по CSS level 2.

Свойство `top` предписывает, насколько далеко нужно сместить верхний край какого-либо объекта форматирования от верхней границы блока, содержащего этот объект. Возможные значения этого свойства описываются следующей конструкцией:

```
<length> | <percentage> | auto | inherit
```

По умолчанию используется значение `auto`.

Свойство `right` указывает, на какую величину должен отступать объект форматирования от правой границы содержащего блока. Значения описываются все той же конструкцией:

```
<length> | <percentage> | auto | inherit
```

По умолчанию используется значение `auto`.

Продолжает список свойств абсолютного позиционирования свойство `bottom`, назначающее смещение нижней грани объекта. Возможные значения все те же:

```
<length> | <percentage> | auto | inherit
```

По умолчанию используется значение `auto`.

И завершает этот список свойство `left`, указывающее расположение левой границы объекта форматирования. Список возможных значений этого свойства ничем не отличается от приведенных выше:

```
<length> | <percentage> | auto | inherit
```

По умолчанию используется значение `auto`.

При использовании процентных соотношений они рассчитываются на основе размеров блока, в котором находится искомый объект форматирования.

Теперь переходим к общим свойствам фона.

Начнем со свойства `background-attachment`, которое перенесено в XSL из CSS level 2. Оно регулирует привязку фонового изображения к основному отображаемому содержимому или к самой области просмотра. Значения свойства объявлены так:

```
scroll | fixed | inherit
```

По умолчанию используется значение `scroll`.

Свойство `background-color` задает цвет фона. Значения объявляются следующим образом:

```
<color> | transparent | inherit
```

По умолчанию используется прозрачный фон, задаваемый значением `transparent`.

Свойство `background-image` позволяет указывать графическое изображение, используемое в качестве фона. Используется следующий набор возможных значений:

```
<uri-specification> | none | inherit
```

По умолчанию используется свойство `none`.

Свойство `background-repeat` регулирует повторяемость фонового рисунка. Данное свойство также заимствовано из CSS level 2. Применяется следующий набор возможных значений:

```
repeat | repeat-x | repeat-y | no-repeat | inherit
```

По умолчанию используется значение `repeat`, которое тиражирует фоновое изображение по горизонтали и вертикали.

Свойство background-position-horizontal задает смещение фона по горизонтали от правой границы блока, содержащего данный объект форматирования. Его возможные значения объявляются так:

```
<percentage> | <length> | left | center | right | inherit
```

По умолчанию используется значение 0%.

Свойство background-position-vertical регулирует вертикальное смещение фонового изображения относительно верхней границы содержащего объект блока. Используется следующий набор возможных значений:

```
<percentage> | <length> | top | center | bottom | inherit
```

По умолчанию используется процентное соотношение 0%.

Свойство border-before-color позволяет указывать цвет границы так называемой "before"-грани контейнера, содержащего объект форматирования. Обычно, но необязательно, это верхняя грань. Свойство может иметь следующие значения:

```
<color> | inherit
```

По умолчанию используется значение, присвоенное свойству color.

Свойство border-before-style позволяет изменять стиль этой "before"-грани. Используются следующие значения:

```
<border-style> | inherit
```

Значение по умолчанию у этого свойства отсутствует.

Свойство border-before-width регулирует толщину границы "before"-грани. Это свойство обладает следующими значениями:

```
<border-width> | <length-conditional> | inherit
```

По умолчанию используется значение medium.

Свойство border-after-color задает цвет границы "after"-грани контейнера, содержащего объект форматирования. Обычно это нижняя грань. Данное свойство может иметь следующие значения:

```
<color> | inherit
```

По умолчанию используется значение, присвоенное свойству color.

Свойство border-after-style позволяет задавать стиль "after"-грани. Используются следующие значения:

```
<border-style> | inherit
```

Значение по умолчанию у этого свойства отсутствует.

Свойство border-after-width регулирует толщину границы "after"-грани. Используется следующий набор возможных значений:

```
<border-width> | <length-conditional> | inherit
```

Значение по умолчанию — medium.

Точно на тех же принципах определены свойства для "start"- и "end"-грани. Просто перечислим их. Это border-start-color, border-start-style, border-start-width, border-end-color, border-end-style и border-end-width.

Эти грани не всегда ориентированы по умолчанию. Возможна их перегруппировка относительно указанного порядка. Но мы можем гарантированно использовать именно те границы, которые нам нужны. Для этого применяются свойства, рассмотренные ниже.

Свойство border-top-color регулирует цвет верхней границы объекта форматирования, к которому оно применяется. Возможные значения этого свойства определяются следующим образом:

```
<color> | inherit
```

Свойство border-top-style определяет стиль верхней границы объекта форматирования. Свойство может иметь следующие значения:

```
<border-style> | inherit
```

В набор border-style входят ключевые слова none, hidden, dotted, dashed, solid, double, groove, ridge, inset и outset, известные нам по CSS level 2. По умолчанию используется значение none.

Свойство border-top-width регулирует толщину верхней границы объекта форматирования. Используется следующий набор значений:

```
<border-width> | inherit
```

В набор border-width входят ключевые слова thin, medium, thick и конструкция length.

Аналогичные свойства существуют для нижней, правой и левой границ. Это border-bottom-color, border-bottom-style, border-bottom-width, border-left-color, border-left-style, border-left-width, border-right-color, border-right-style и border-right-width соответственно.

Свойства padding-before, padding-after, padding-start и padding-end задают величину отступа содержимого объекта форматирования от соответствующих граней. Эти свойства могут иметь следующие значения:

```
<padding-width> | <length-conditional> | inherit
```

По умолчанию задается нулевой отступ при помощи значения 0pt.

Свойства padding-top, padding-bottom, padding-left и padding-right задают отступ содержимого объекта форматирования от верхней, нижней, левой и правой границы соответственно. Эти свойства обладают следующим набором значений:

```
<padding-width> | inherit
```

По умолчанию используется, разумеется, нулевой отступ, задаваемый значением `0pt`.

Теперь перейдем к свойствам шрифтов.

Свойство `font-family` заимствовано из спецификации CSS level 2, поэтому и набор возможных значений унаследовало все тот же:

```
[[ <family-name> | <generic-family> ],]* [<family-name> |
↳generic-family>] | inherit
```

Расшифрованы эти значения в предыдущей главе.

Свойство `font-selection-strategy` задает механизм выбора шрифта для отображения текстового содержимого объекта форматирования. Имеет следующие значения:

```
auto | character-by-character | inherit
```

Значение по умолчанию `auto` определяет стандартный механизм выбора шрифта, который устанавливает единый шрифт для всего текстового содержимого объекта форматирования. А значение `character-by-character` заставляет браузер использовать шрифты, персонально назначенные каждому объекту типа `fo:character`.

Свойство `font-size` позволяет задавать размер шрифта, применяемого для отображения текстового содержимого объекта форматирования. Возможные значения для этого свойства определяются так:

```
<absolute-size> | <relative-size> | <length> | <percentage> |
↳inherit
```

Набор `absolute-size` состоит из следующих значений:

```
[ xx-small | x-small | small | medium | large | x-large |
↳xx-large ]
```

Эти ключевые слова идентичны своим аналогам из CSS level 2.

Набор `relative-size` состоит всего из двух значений: `larger` и `smaller`, которые увеличивают и уменьшают размер шрифта относительно размера шрифта родительского элемента соответственно.

По умолчанию используется значение `medium`.

Свойство `font-stretch` также заимствовано из CSS level 2. Соответственно, унаследован и набор возможных значений:

```
normal | wider | narrower | ultra-condensed | extra-condensed |
↳condensed | semi-condensed | semi-expanded | expanded |
↳extra-expanded | ultra-expanded | inherit
```

Напомним, что это свойство позволяет варьировать величину межсимвольного просвета, влияющего, в свою очередь, на жирность шрифта.

Свойство `font-size-adjust` позволяет задавать коэффициент масштабирования символов выбранного шрифта. Унаследовано из CSS level 2. Обладает следующим набором значений:

```
<number> | none | inherit
```

То есть для установки нужного коэффициента достаточно просто указать числовое значение. По умолчанию используется значение `none`.

Свойство `font-style` задает ориентировку символов установленного шрифта. Используются следующий набор значений:

```
normal | italic | oblique | backslant | inherit
```

Первых три ключевых слова рассмотрены нами в предыдущей главе. Значение `backslant` специфицирует шрифты с обратным наклоном символов.

По умолчанию используется прямая ориентировка шрифта, задаваемая значением `normal`.

Свойство `font-variant` указывает способ отображения символов выбранного шрифта. Они могут отображаться как обычно, или в виде заглавных букв, но с сохранением размеров. Для этих целей используется следующий набор значений:

```
normal | small-caps | inherit
```

По умолчанию используется свойство `normal`.

Свойство `font-weight` позволяет регулировать ширину символов применяемого шрифта. Обладает следующим набором значений:

```
normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 |
↳600 | 700 | 800 | 900 | inherit
```

Мы рассматривали их в предыдущей главе, посвященной CSS level 2.

По умолчанию используется значение `normal`.

Теперь мы переходим к рассмотрению свойств, определяющих локализацию объектов форматирования.

Свойство `country` позволяет определять страну, на которую рассчитано содержимое данного объекта форматирования. Значения этого свойства определяются следующим образом:

```
none | <country> | inherit
```

По умолчанию мы используем ключевое слово `none`, которое также следует задавать в случаях, когда не имеет значения, для какой страны предназначается данный объект. Или же можно непосредственно указать код страны, который находится в стандарте RFC 1766.

Свойство `language` специфицирует язык текстового содержимого данного объекта форматирования. Свойство может иметь следующие значения:

```
none | <language> | inherit
```

Коды применяемых языков указаны все в том же стандарте RFC 1766.

Свойство `script` обычно применяется для задания дополнительных сведений о языке. Так, например, в японском языке есть несколько иероглифических систем. Выбор между ними производится при помощи этого свойства. Применяется следующий набор возможных значений:

```
none | auto | <script> | inherit
```

Обозначения, входящие в состав множества `script`, указаны в стандарте ISO 15924.

Свойство `hyphenate` включает механизм разбиения текста на отдельные строки. Используется следующий набор свойств:

```
false | true | inherit
```

Значение по умолчанию `false` указывает, что XML-процессор не должен использовать алгоритм разбиения текста на отдельные строки.

Свойства `margin-top`, `margin-bottom`, `margin-left` и `margin-right` задают размер верхнего, нижнего, левого и правого поля соответственно. Эти свойства обладают одинаковым набором возможных значений, который определяется следующим образом:

```
<margin-width> | inherit
```

По умолчанию используется нулевое значение величины любого поля.

Свойства `space-before` и `space-after` позволяют указывать предпочтительный размер пробела перед и после данного объекта форматирования соответственно. Набор возможных значений для этих свойств определяется так:

```
<space> | inherit
```

Свойства `space-end` и `space-start` задают размер пробела между областью отображения содержимого объекта форматирования и "`end`"- и "`start`"-гранями контейнера, его содержащего. Свойство обладает следующим набором возможных значений:

```
<space> | inherit
```

Теперь настала очередь разобрать свойства относительного позиционирования.

Свойство `relative-position` унаследовано из CSS level 2. Оно позволяет устанавливать механизм относительного позиционирования объекта форматирования в блоке, содержащем его. Свойство обладает следующим набором возможных значений:

```
static | relative | inherit
```

По умолчанию используется значение `static`.

Свойство `alignment-adjust` применяется для выравнивания элементов, которые не имеют явно выраженной базовой линии. Набор возможных значений объявляется следующим образом:

```
auto | baseline | before-edge | text-before-edge | middle | central |
↳after-edge | text-after-edge | ideographic | alphabetic | hanging |
↳mathematical | <percentage> | <length> | inherit
```

По умолчанию используется значение `auto`.

Свойство `alignment-baseline` задает выравнивание объекта относительно базовой линии родительского объекта. Обладает следующим набором значений:

```
auto | baseline | before-edge | text-before-edge | middle | central |
↳after-edge | text-after-edge | ideographic | alphabetic | hanging |
↳mathematical | inherit
```

Свойство `baseline-shift` предназначено для репозиционирования объектов форматирования посредством их перевода в состояние верхнего или нижнего индекса. Использование этого свойства ведет к изменению положения основной базовой линии объекта относительно базовой линии родительского объекта. Набор возможных значений данного свойства объявляется следующим образом:

```
baseline | sub | super | <percentage> | <length> | inherit
```

Свойство `display-align` регулирует выравнивание объекта относительно граней "`before`" и "`after`". Обладает следующим набором возможных значений:

```
auto | before | center | after | inherit
```

Значение по умолчанию `auto` оставляет объект на том месте, куда его отправляет свойство относительного позиционирования. Остальные значения прижимают объект к одной из граней или же производится его центрирование в пределах области просмотра.

Свойство `dominant-baseline` определяет расположение трех базовых линий объекта форматирования. Его возможные значения задаются следующим образом:

```
auto | use-script | no-change | reset-size | ideographic |
↳alphabetic | hanging | mathematical | inherit
```

Свойство `relative-align` задает относительное выравнивание для элементов списков и ячеек таблицы. Используются следующие свойства:

```
before | baseline | inherit
```

По умолчанию используется значение `before`, прижимающее объект форматирования к одноименной грани.

Свойство `block-progression-dimension` заимствовано из CSS level 2. Соответственно, унаследован и набор возможных значений:

```
auto | <length> | <percentage> | <length-range> | inherit
```

По умолчанию используется значение `auto`.

Свойство `content-height` позволяет устанавливать высоту отображаемого содержимого объекта форматирования. Это свойство обладает следующим набором значений:

```
auto | scale-to-fit | <length> | <percentage> | inherit
```

По умолчанию используется значение `auto`.

Свойство `content-width` дает возможность указывать требуемую ширину отображаемого содержимого объекта форматирования. Набор значений такой же, как и для предыдущего свойства:

```
auto | scale-to-fit | <length> | <percentage> | inherit
```

По умолчанию используется значение `auto`.

Свойство `height` задает высоту блока, в котором отображается объект форматирования. Используется следующий набор значений:

```
<length> | <percentage> | auto | inherit
```

По умолчанию используется значение `auto`.

Свойство `content-height` позволяет устанавливать высоту отображаемого содержимого объекта форматирования. Это свойство обладает тем же набором значений:

```
<length> | <percentage> | auto | inherit
```

Значение по умолчанию — `auto`.

Свойство `inline-progression-dimension` аналогично рассмотренному нами свойству `block-progression-dimension`, но применяется для компактных inline-блоков. Набор возможных значений этого свойства определяется следующим образом:

```
auto | <length> | <percentage> | <length-range> | inherit
```

По умолчанию используется значение `auto`.

Свойство `max-height` ограничивает максимально возможную высоту блока, в котором размещается содержимое объекта форматирования. Обладает следующим набором значений:

```
<length> | <percentage> | none | inherit
```

По умолчанию используется значение `0pt`.

Свойство `max-width` позволяет вводить ограничение на максимально возможную ширину блока, в котором размещается содержимое объекта форматирования. Обладает следующим набором значений:

```
<length> | <percentage> | none | inherit
```

По умолчанию используется значение `none`.

Свойство `min-height` позволяет задавать минимально возможную высоту блока, в котором размещается содержимое объекта форматирования. Обладает следующим набором значений:

```
<length> | <percentage> | inherit
```

По умолчанию используется значение `0pt`.

Свойство `min-width` определяет минимально возможную ширину блока, в котором размещается содержимое объекта форматирования. Набор возможных значений и значение по умолчанию идентичны применяемым для свойства `min-height`.

Свойство `scaling` регулирует применение масштабирования содержимого объекта форматирования. Возможные значения данного свойства определяются так:

```
uniform | non-uniform | inherit
```

Значение по умолчанию `uniform` сохраняет при масштабировании заданный коэффициент. Значение `non-uniform` указывает, что XML-процессор не будет заботиться о сохранении этого коэффициента.

Свойство `scaling-method` позволяет устанавливать метод масштабирования содержимого объекта форматирования. Применяется следующий набор значений:

```
auto | integer-pixels | resample-any-method | inherit
```

Значение по умолчанию `auto` оставляет выбор метода масштабирования за XML-процессором. Значение `integer-pixels` сигнализирует, что каждый пиксел изображения будет подогнан к ближайшему целому числу пикселов отображающего устройства. Значение `resample-any-method` указывает, что изображение должно полностью занять весь блок, отведенный под него, вне зависимости от правил трансформации.

Свойство `width` задает ширину блока, содержащего объект форматирования. Заимствовано из CSS level 2 и имеет такой же набор возможных значений:

```
<length> | <percentage> | auto | inherit
```

По умолчанию используется значение `auto`.

Свойство `hyphenation-keep` устанавливает порядок переноса слов текстового содержимого объекта форматирования. Обладает следующим набором возможных значений:

```
auto | column | page | inherit
```

Значение `auto` не накладывает никаких ограничений на расположение отдельных частей переносимого слова. Значение `column` заставляет браузер отображать обе части переносимого слова обязательно в одной колонке. А значение `page` принуждает оставлять их в пределах одной страницы.

Свойство `hyphenation-ladder-count` позволяет задавать максимальное количество последовательных строк, завершаемых переносами. Возможные значения этого свойства объявляются следующим образом:

```
no-limit | <number> | inherit
```

Значение по умолчанию `no-limit` указывает, что количество следующих друг за другом строк с переносами не ограничивается. Если все-таки необходимо это ограничение установить, то мы должны в качестве значения использовать положительное целое число.

Свойство `last-line-end-indent` задает отступ последней строки текстового содержимого объекта форматирования от "end"-грани контейнера, содержащего этот объект. Его значения описываются так:

```
<length> | <percentage> | inherit
```

По умолчанию используется значение `0pt`.

Свойство `line-height` задает высоту строки текста. Используется следующее множество возможных значений:

```
normal | <length> | <number> | <percentage> | <space> | inherit
```

По умолчанию используется значение `normal`, основанное на высоте символов применяемого шрифта.

Свойство `line-stacking-strategy` определяет механизм выбора значения высоты текстовой строки. Для этого применяется следующий набор значений:

```
line-height | font-height | max-height | inherit
```

Значение по умолчанию `line-height` указывает, что истинная высота строки будет совпадать со значением, установленным в одноименном свойстве, которое мы только что рассматривали. Остальные значения также совпадают с наименованиями характеристик, которые позволяют регулировать высоту текстовых строк.

Свойство `text-align` заимствовано из CSS level 2 и управляет выключкой текста. Обладает следующим набором значений:

```
start | center | end | justify | inside | outside | left | right |
↳<string> | inherit
```

По умолчанию используется значение `start`.

Свойство `text-align-last` позволяет задавать выравнивание последней строки текстового содержимого объекта форматирования. Его возможные значения задаются следующим образом:

```
relative | start | center | end | justify | inside | outside |
↳left | right | inherit
```

Свойство `text-indent` задает отступ первой строки текстового содержимого объекта форматирования. Обладает следующим набором значений:

```
<length> | <percentage> | inherit
```

По умолчанию используется значение `0pt`.

Свойство `white-space-collapse` позволяет указывать, можем мы сворачивать последовательные пробелы или нет. Регулируется это при помощи следующего набора значений:

```
false | true | inherit
```

По умолчанию используется значение `true`.

Свойство `wrap-option` дает возможность разбивать строки с целью их полного размещения в блоке отображения без применения полос прокрутки. Свойство обладает следующим набором значений:

```
no-wrap | wrap | inherit
```

По умолчанию используется значение `wrap`, разбивающее строки на несколько частей.

Свойство `character` позволяет устанавливать отображаемый символ для соответствующего объекта форматирования. Возможные значения объявляются так:

```
<character>
```

То есть в качестве значения данного свойства мы просто указываем символ Unicode.

Свойство `letter-spacing` назначает межсимвольный интервал для текстового содержимого объекта форматирования. Свойство обладает следующим набором значений:

```
normal | <length> | <space> | inherit
```

По умолчанию используется значение `normal`.

Свойство `text-decoration` предоставляет дополнительные возможности оформления текста объекта форматирования. Выбор конкретного эффекта, применяемого к тексту, осуществляется при помощи следующего набора значений:

```
none | [ [ underline | no-underline] || [ overline | no-overline ] ||
↳[ line-through | no-line-through ] || [ blink | no-blink ] ] |
↳inherit
```

Значения этих ключевых слов мы узнали в предыдущей главе.

Свойство `text-shadow` задает теневое оформление текста. Оно идентично своему "одноименному близнецу" из CSS level 2. Соответственно, возможные значения этого свойства таковы:

```
none | [<color> || <length> <length> <length>? ,]* [<color> ||
↳<length> <length> <length>?] | inherit
```

По умолчанию используется значение `none`.

Свойство `text-transform` позволяет изменять регистр символов применяемого шрифта. Обладает следующим набором значений:

```
capitalize | uppercase | lowercase | none | inherit
```

По умолчанию используется значение `none`.

Свойство `treat-as-word-space` применимо к отдельному символу и позволяет указывать, будет ли этот символ разделителем между словами или нет. Настройка производится на основе следующего набора возможных значений:

```
auto | true | false | inherit
```

Значение по умолчанию `auto` указывает, что для разделения слов будут использоваться стандартные символы Unicode.

Величина пробела между словами задается при помощи свойства `word-spacing`. Оно обладает следующим набором возможных значений:

```
normal | <length> | <space> | inherit
```

По умолчанию используется значение `normal`.

Свойство `color` регулирует цвет, которым будет отображаться искомый элемент форматирования. Возможные значения этого свойства определены так:

```
<color> | inherit
```

Свойство `color-profile-name` задает наименование используемого цветового профиля, фактически, цветовой палитры, применяемой для отображения внешних данных. Данное свойство обладает следующим набором возможных значений:

```
<name> | inherit
```

Свойство `rendering-intent` указывает способ, которым внешняя цветовая палитра будет приводиться к отображаемому множеству цветов применяемого средства отображения документа. Дело в том, что цвета, заданные, скажем, на системах Apple, могут частично не совпадать с цветовой системой PC-клонов. Чаще всего данное свойство применяется для CMYK-цветов. Оно обладает следующим набором значений, каждое из которых обозначает отдельный метод конвертации цветов:

```
auto | perceptual | relative-colorimetric | saturation |
↳absolute-colorimetric | inherit
```

По умолчанию используется значение `auto`.

Свойство `float` указывает, к какой границе родительского блока будет сдвинут блок данного объекта форматирования. Применяется следующий набор возможных значений:

```
before | start | end | left | right | none | inherit
```

По умолчанию используется значение `none`.

Свойство `break-after` позволяет уточнять, после какого объекта необходимо будет делать разрыв при отображении содержимого объекта форматирования. Свойство может иметь следующие значения:

```
auto | column | page | even-page | odd-page | inherit
```

Значение по умолчанию `auto` указывает, что специально производить разрыв не нужно. Остальные значения заставляют делать разрывы после колонок и страниц.

Свойство `break-before` указывает, перед каким объектом необходимо сделать принудительный разрыв. Обладает следующим набором значений:

```
auto | column | page | even-page | odd-page | inherit
```

По умолчанию используется значение `auto`.

Свойство `keep-together` определяет, сколько строк, колонок или страниц должны отображаться вместе с искомым объектом форматирования без разрыва. Используется следующий набор значений:

```
<keep> | inherit
```

Множество `keep` состоит из ключевых слов `auto` и `always` и целых положительных чисел.

Свойства `keep-with-next` и `keep-with-previous` управляют совместным отображением со следующим и с предыдущим элементом форматирования. Обладают тем же набором возможных значений, что и свойство `keep-together`:

```
<keep> | inherit
```

Свойство `orphans` указывает количество строк абзаца, которые можно оставить в нижней части страницы. Используется следующий набор значений:

```
<integer> | inherit
```

По умолчанию используется значение 2. То есть при переносе части абзаца на новую страницу браузер оставит как минимум две строки на исходной странице.

Свойство `widows` задает минимальное количество строк абзаца, которые можно перенести на следующую страницу. Обладает все тем же набором возможных значений, что и предыдущее свойство:

```
<integer> | inherit
```

По умолчанию используется значение 2. Следовательно, при разрыве абзаца на новую страницу будет перенесено не менее двух строк.

Свойство `clip` унаследовано из CSS level 2 и предназначено для указания области, к которой прикреплено отображаемое содержимое объекта форматирования. Возможные значения этого свойства объявлены следующим образом:

```
<shape> | auto | inherit
```

По умолчанию используется значение auto, указывающее, что областью отсечения будет являться весь блок-контейнер, в котором отображается объект форматирования. Если необходимо видоизменить область отсечения, следует явно задать координаты этой прямоугольной области.

Свойство overflow устанавливает реакцию браузера в тех случаях, когда содержимое объекта форматирования выходит за пределы области прикрепления. Обладает следующим набором возможных значений:

```
visible | hidden | scroll | error-if-overflow | auto | inherit
```

По умолчанию используется значение auto, оставляющее выбор действия за браузером. Значение visible отображает объект форматирования полностью, вне зависимости от его прикрепления к какой-либо области. Значение hidden заставляет браузер не отображать ту часть объекта форматирования, которая выходит за пределы области отсечения. Значение scroll указывает, что в области отсечения необходимо генерировать полосы прокрутки, которые позволят просмотреть все содержимое объекта форматирования, не изменяя при этом пределы области отсечения. Значение error-if-overflow сигнализирует, что при выходе содержимого объекта за пределы области прикрепления ситуацию необходимо расценивать как аварийную, и возбудить сообщение об ошибке.

Свойство reference-orientation позволяет выводить содержимое объекта форматирования под углом к горизонтали. Используется следующий набор возможных значений:

```
0 | 90 | 180 | 270 | -90 | -180 | -270 | inherit
```

Эти числа являются значениями углов поворота в градусах. Поворот отсчитывается по часовой стрелке. Очевидно, что такие значения, как 90 и -270 — эквивалентны. По умолчанию используется нулевое значение.

Свойство span регулирует объединение колонок, заключенных в объекте форматирования, к которому оно применяется. Используется следующий набор возможных значений:

```
none | all | inherit
```

То есть мы либо не объединяем колонки в принципе (значение по умолчанию none), либо объединяем их все (значение all).

Свойство leader-alignment регулирует выравнивание стартового блока, такого как маркер списка. Это свойство обладает следующим набором возможных значений:

```
none | reference-area | page | inherit
```

Значение по умолчанию none указывает, что данный объект форматирования не будет иметь какого-либо специализированного выравнивания. Значение reference-area прижимает объект форматирования к "start"-грани

блока, в который вложен элемент. Значение page прижимает объект форматирования к "start"-грани страницы, на которой находится весь элемент.

Свойство leader-pattern задает внешний вид маркера, используемого в качестве стартового блока. Применяется следующее множество допустимых значений:

```
space | rule | dots | use-content | inherit
```

По умолчанию используется значение space, создающее маркер в виде пробельного символа.

Свойство leader-pattern-width устанавливает ширину маркера. Допустимые значения определяются следующим образом:

```
use-font-metrics | <length> | inherit
```

По умолчанию используется значение use-font-metrics, указывающее, что необходимо использовать стандартные размеры символов применяемого шрифта.

Свойство leader-length задает длину блока, в котором отображается маркер. Свойство обладает следующим набором возможных значений:

```
<length-range> | inherit
```

Множество length-range содержит три значения: минимум (0pt), максимум (100%) и оптимум (12pt).

Свойство rule-style определяет внешний вид маркера в том случае, если для свойства leader-pattern установлено значение rule. Для него, в свою очередь, предусмотрены следующие значения:

```
none | dotted | dashed | solid | double | groove | ridge | inherit
```

По умолчанию используется значение solid.

Свойство rule-thickness устанавливает толщину символа маркера, заданного свойством rule-style. Возможные значения определяются несложно:

```
<length>
```

По сути дела, мы просто напрямую указываем размер. По умолчанию используется значение 1.0pt.

Свойство active-state задает конкретное состояние объекта с множественным набором значений, каким обычно является гиперссылка. Применяется следующий набор значений, знакомый нам по CSS level 2:

```
link | visited | active | hover | focus
```

Свойство auto-restore указывает, может ли объект самостоятельно восстанавливать свое первоначальное состояние. Обладает следующим набором значений:

```
true | false
```

По умолчанию используется значение `false`.

Свойство `destination-placement-offset` задает дистанцию от начала страницы до блока отображения содержимого объекта форматирования. Возможное значение объявлено следующим образом:

```
<length>
```

По умолчанию применяется значение `0pt`.

Свойство `external-destination` задает URI внешнего содержимого объекта форматирования. Значения данного свойства определяются так:

```
<uri-specification>
```

Свойство `indicate-destination` указывает, будет ли отображаться адрес перехода по ссылке в момент ее прохождения. Обладает следующим набором значений:

```
true | false
```

По умолчанию используется значение `false`.

Свойство `internal-destination` задает адрес субресурса по его идентификатору в текущем документе для локальной гиперссылки. Применяется следующий набор возможных значений:

```
empty string | <idref>
```

По умолчанию используется значение `empty string`, то есть пустая строка.

Свойство `show-destination` регулирует порядок отображения ресурса, на который указывает ссылка. Возможные значения этого свойства определяются следующим образом:

```
replace | new
```

Значение по умолчанию `replace` замещает текущий документ новым ресурсом, а значение `new` отображает его в новом окне.

Свойство `starting-state` позволяет устанавливать начальное состояние отображаемого объекта. Определяется оно при помощи следующих значений:

```
show | hide
```

Значение по умолчанию `show` заставляет браузер автоматически отображать этот объект. Значение `hide` оставляет его изначально в скрытом состоянии.

Свойство `target-stylesheet` позволяет указывать стилевую таблицу, которая будет применяться для отображения того или иного документа. Применяется следующий набор возможных значений:

```
use-normal-stylesheet | <uri-specification>
```

По умолчанию используется значение `use-normal-stylesheet`.

Свойство `blank-or-not-blank` предназначено для мастер-страниц. Оно указывает, будет эта страница пустой или нет. Используется следующий набор значений:

```
blank | not-blank | any | inherit
```

По умолчанию выбрано значение `any`.

Свойство `column-count` задает количество колонок в теле страницы. Применяется следующий набор возможных значений:

```
<number> | inherit
```

Данному свойству по умолчанию присваивается значение 1.

Свойство `column-gap` объявляет ширину промежутка между отдельными колонками. Может иметь следующие значения:

```
<length> | <percentage> | inherit
```

Если значение не указано, то используется `12.0pt`.

Свойство `force-page-count` управляет нумерацией последовательности страниц. Используется следующий набор возможных значений:

```
auto | even | odd | end-on-even | end-on-odd | no-force | inherit
```

По умолчанию выбрано значение `auto`.

Начальный номер последовательности страниц указывается при помощи значения `initial-page-number`. Применяется следующий набор значений:

```
auto | auto-odd | auto-even | <number> | inherit
```

"Умолчальное" значение — `auto`.

Свойство `maximum-repeats` указывает максимальное количество повторов страниц. Используется следующий набор возможных значений:

```
<number> | no-limit | inherit
```

Второе из них является значением по умолчанию.

Свойство `media-usage` указывает тип отображения содержимого документа. Мы можем разбить его на страницы или ограничиться отображением содержимого в непрерывном виде. Выбор режима производится из списка следующих возможных значений:

```
auto | paginate | bounded-in-one-dimension | unbounded
```

По умолчанию свойство принимает значение `auto`.

Свойство `page-height` позволяет устанавливать высоту страниц. Для этого используется следующий набор возможных значений:

```
auto | indefinite | <length> | inherit
```

Значение по умолчанию — `auto`.

Положение мастер-страниц задается при помощи свойства `page-position`. Используется следующий набор значений:

```
first | last | rest | any | inherit
```

Если ни одно из ключевых слов не указано, применяется значение `any`.

Ширина одиночной мастер-страницы задается при помощи свойства `page-width`. Возможные значения этого свойства объявляются следующим образом:

```
auto | indefinite | <length> | inherit
```

По умолчанию ширина устанавливается автоматически (`auto`).

Свойство `region-name` объявляет наименование и, соответственно, тип специализированной области отображения. Используется следующее множество значений:

```
xsl-region-body | xsl-region-start | xsl-region-end |
↳xsl-region-before | xsl-region-after | xsl-before-float-separator |
↳xsl-footnote-separator | <name>
```

Свойство `border-collapse` заимствовано из CSS level 2 и определяет выбор используемой модели границ в табличных объектах. Применяется уже знакомый нам набор возможных значений:

```
collapse | separate | inherit
```

По умолчанию используется значение `collapse`.

Свойство `caption-side` указывает, с какой стороны таблицы будет отображаться ее заголовок. Для этого используются следующие значения:

```
before | after | start | end | top | bottom | left | right | inherit
```

По умолчанию выбрано значение `before`.

Свойство `column-number` указывает номер колонки для текущего объекта форматирования. В качестве значений данного свойства могут использоваться целые положительные числа. Отсчет ведется от единицы.

Свойство `column-width` задает ширину столбца таблицы. Оно обладает следующим набором возможных значений:

```
<length> | <percentage>
```

Свойство `empty-cells` регулирует порядок отображения пустых ячеек таблицы. Используется следующий набор возможных значений:

```
show | hide | inherit
```

Значение `show`, выбранное по умолчанию, заставляет браузер отображать пустые ячейки. А значение `hide` указывает, что эти ячейки отображать не следует.

Свойство `ends-row` по сути своей — индикатор. Оно указывает, является или нет данная ячейка последней в строке таблицы. Соответственно применяются булевы значения:

```
true | false
```

По умолчанию используется значение `false`.

Свойство `number-columns-repeated` задает количество столбцов таблицы с одинаковым оформлением. В качестве значения используется обычное целое положительное число.

Свойство `number-columns-spanned` указывает количество соседних столбцов или ячеек одной строки, объединенных вместе. В качестве значения используется обычное целое положительное число.

Свойство `number-rows-spanned` указывает количество соседних ячеек одного столбца, объединенных вместе. В качестве значения может выступать обычное целое положительное число.

Свойство `starts-row` является индикатором, и указывает, что данная ячейка является первой в строке таблицы. Свойство обладает следующим набором возможных значений:

```
true | false
```

По умолчанию выбирается значение `false`.

Свойство `table-layout`, заимствованное из CSS level 2, позволяет выбирать модель раскладки таблицы. Применяется знакомый нам набор возможных значений:

```
auto | fixed | inherit
```

По умолчанию выбирается значение `auto`.

Свойство `direction` задает направление распространения текста вдоль горизонтальной оси. Имеет следующий набор значений:

```
ltr | rtl | inherit
```

По умолчанию используется значение `ltr`, то есть, пишем мы слева направо.

Свойство `glyph-orientation-horizontal` назначает угол поворота символа относительно горизонтали. Оно обладает следующим набором значений:

```
<angle> | inherit
```

По умолчанию выбрано значение `0deg`. Мы можем использовать только углы, кратные 90 градусам.

Свойство `glyph-orientation-vertical` задает угол поворота символа, отсчитываемый от вертикальной оси. Свойство обладает следующим набором значений:

```
auto | <angle> | inherit
```

По умолчанию применяется значение `auto`. Разрешено использовать только углы, кратные 90 градусам.

Свойство `text-depth` определяет так называемую "глубину" текста, то есть расстояние между обычной и доминантной базовой линией. Свойство может иметь следующие значения:

```
use-font-metrics | <length> | inherit
```

По умолчанию используется значение `use-font-metrics`.

Свойство `writing-mode` регулирует порядок заполнения текстом блока объекта форматирования. Имеется следующий набор возможных значений:

```
lr-tb | rl-tb | tb-rl | lr | rl | tb | inherit
```

По умолчанию используется значение `lr-tb`, то есть текст развертывается слева направо, и строки идут сверху вниз.

Свойство `src` задает URI графического файла для объекта форматирования, включающего внешний файл. В качестве значения данного свойства выступает конкретный URI.

Свойство `visibility` регулирует видимость того или иного объекта форматирования. Свойство позаимствовано из CSS level 2, поэтому унаследовало следующий набор возможных значений:

```
visible | hidden | collapse | inherit
```

По умолчанию выбирается значение `visible`.

Свойство `z-index` позволяет задавать z-координату (очередность отображения) объекта форматирования. Оно обладает следующим набором значений:

```
auto | <integer> | inherit
```

По умолчанию используется значение `auto`.

Свойство `background` является агрегатным и позволяет сразу указывать основные характеристики фона. Для этого применяется следующее множество возможных значений:

```
[<background-color> || <background-image> || <background-repeat> ||
↳<background-attachment> || <background-position> ]] | inherit
```

Свойство `background-position` задает расположение фонового изображения. Значения этого свойства объявлены в спецификации так:

```
[ [<percentage> | <length> ]{1,2} | [ [top | center | bottom] ||
↳[left | center | right] ] ] | inherit
```

По умолчанию используется значение `0% 0%`.

Свойство `border` регулирует вид границы объекта форматирования. Обладает следующим множеством возможных значений:

```
[ <border-width> || <border-style> || <color> ] | inherit
```

Соответственно, XSL наследует и все остальные свойства границ, отступов, полей и фона из CSS level 2.

Свойство `font` позволяет указывать все необходимые свойства применяемого шрифта. Достигается это при помощи обширного набора возможных значений:

```
[ [ <font-style> || <font-variant> || <font-weight> ]? <font-size>
↳[ / <line-height>]? <font-family> ] | caption | icon | menu |
↳message-box | small-caption | status-bar | inherit
```

Тип позиционирования элемента форматирования задается свойством `position`. Оно обладает следующим набором возможных значений:

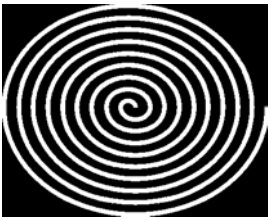
```
static | relative | absolute | fixed | inherit
```

Размеры страниц задаются при помощи свойства `size`. Возможные его значения определяются так:

```
<length>{1,2} | auto | landscape | portrait | inherit
```

На этом мы завершаем обзор основных свойств объектов форматирования XSL. А значит, мы заканчиваем рассмотрение всего языка XSL.

Приложение 1



Официальная спецификация XML

```
[1]  document ::=  prolog element Misc*
[2]  Char ::=  #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD]
| [#x10000-#x10FFFF] /*  any Unicode character, excluding the
surrogate blocks, FFFE, and FFFF. */
[3]  S ::=  (#x20 | #x9 | #xD | #xA)+
[4]  NameChar ::=  Letter | Digit | '.' | '-' | '_' | ':'
| CombiningChar | Extender
[5]  Name ::=  (Letter | '_' | ':') (NameChar)*
[6]  Names ::=  Name (S Name)*
[7]  Nmtoken ::=  (NameChar)+
[8]  Nmtokens ::=  Nmtoken (S Nmtoken)*
[9]  EntityValue ::=  '"' ([^%&"] | PEReference | Reference)* '"'
   |  "'" ([^%&'] | PEReference | Reference)* "'"
[10]  AttValue ::=  '"' ([^<&"] | Reference)* '"'
   |  "'" ([^<&'] | Reference)* "'"
[11]  SystemLiteral ::=  ('"' [^"]* '"') | ("'" [^']* "'")
[12]  PubidLiteral ::=  '"' PubidChar* '"' | "'" (PubidChar - "'")*
"'"
[13]  PubidChar ::=  #x20 | #xD | #xA | [a-zA-Z0-9] | [-
'()+,./:=?;!*#@$_%]
[14]  CharData ::=  [^<&]* - ([^<&]* ']]>' [^<&]*)
[15]  Comment ::=  '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
[16]  PI ::=  '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17]  PITarget ::=  Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
[18]  CDSect ::=  CDStart CData CDEnd
[19]  CDStart ::=  '<![CDATA['
[20]  CData ::=  (Char* - (Char* ']]>' Char*))
[21]  CDEnd ::=  ']]>'
[22]  prolog ::=  XMLDecl? Misc* (doctypedecl Misc*)?
[23]  XMLDecl ::=  '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24]  VersionInfo ::=  S 'version' Eq (' VersionNum ' | " VersionNum
")
[25]  Eq ::=  S? '=' S?
[26]  VersionNum ::=  ([a-zA-Z0-9_.:] | '-')+
[27]  Misc ::=  Comment | PI |  S
[28]  doctypedecl ::=  '<!DOCTYPE' S Name (S ExternalID)? S? ('['
(markupdecl | PEReference | S)* ']' S?)? '>' [  VC: Root Element
Type ]
```

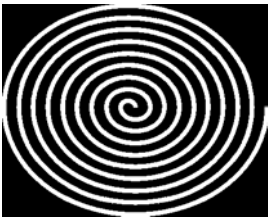
```
[29]  markupdecl ::=  elementdecl | AttlistDecl | EntityDecl
| NotationDecl | PI | Comment  [  VC: Proper Declaration/PE
Nesting ]
     [  WFC: PEs in Internal Subset ]
[30]  extSubset ::=  TextDecl? extSubsetDecl
[31]  extSubsetDecl ::=  ( markupdecl | conditionalSect
| PEReference | S )*
[32]  SDDecl ::=  S 'standalone' Eq (("'" ('yes' | 'no') "'") | ('"'
('yes' | 'no') '"'))  [  VC: Standalone Document Declaration ]
[33]  LanguageID ::=  Langcode ('-' Subcode)*
[34]  Langcode ::=  ISO639Code |  IanaCode |  UserCode
[35]  ISO639Code ::=  ([a-z] | [A-Z]) ([a-z] | [A-Z])
[36]  IanaCode ::=  ('i' | 'I') '-' ([a-z] | [A-Z])+
[37]  UserCode ::=  ('x' | 'X') '-' ([a-z] | [A-Z])+
[38]  Subcode ::=  ([a-z] | [A-Z])+
[39]  element ::=  EmptyElemTag
    | STag content ETag [  WFC: Element Type Match ]
     [  VC: Element Valid ]
[40]  STag ::=  '<' Name (S Attribute)* S? '>' [  WFC: Unique Att
Spec ]
[41]  Attribute ::=  Name Eq AttValue [  VC: Attribute Value Type ]
     [  WFC: No External Entity References ]
     [  WFC: No < in Attribute Values ]
[42]  ETag ::=  '</' Name S? '>'
[43]  content ::=  (element | CharData | Reference | CDSect | PI
| Comment)*
[44]  EmptyElemTag ::=  '<' Name (S Attribute)* S? '/>' [  WFC:
Unique Att Spec ]
[45]  elementdecl ::=  '<!ELEMENT' S Name S contentspec S? '>' [
VC: Unique Element Type Declaration ]
[46]  contentspec ::=  'EMPTY' | 'ANY' | Mixed | children
[47]  children ::=  (choice | seq) ('?' | '*' | '+')?
[48]  cp ::=  (Name | choice | seq) ('?' | '*' | '+')?
[49]  choice ::=  '(' S? cp ( S? '|' S? cp )* S? ')' [  VC: Proper
Group/PE Nesting ]
[50]  seq ::=  '(' S? cp ( S? ',' S? cp )* S? ')' [  VC: Proper
Group/PE Nesting ]
[51]  Mixed ::=  '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
    | '(' S? '#PCDATA' S? ')'  [  VC: Proper Group/PE Nesting ]
     [  VC: No Duplicate Types ]
[52]  AttlistDecl ::=  '<!ATTLIST' S Name AttDef* S? '>'
[53]  AttDef ::=  S Name S AttType S DefaultDecl
[54]  AttType ::=  StringType | TokenizedType | EnumeratedType
[55]  StringType ::=  'CDATA'
[56]  TokenizedType ::=  'ID' [  VC: ID ]
     [  VC: One ID per Element Type ]
     [  VC: ID Attribute Default ]
    | 'IDREF' [  VC: IDREF ]
    | 'IDREFS' [  VC: IDREF ]
    | 'ENTITY' [  VC: Entity Name ]
    | 'ENTITIES' [  VC: Entity Name ]
    | 'NMTOKEN' [  VC: Name Token ]
```

```
    | 'NMTOKENS' [  VC: Name Token ]
[57]  EnumeratedType ::=  NotationType | Enumeration
[58]  NotationType ::=  'NOTATION' S '(' S? Name (S? '|' S? Name)*
S? ')'  [  VC: Notation Attributes ]
[59]  Enumeration ::=  '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')' [
VC: Enumeration ]
[60]  DefaultDecl ::=  '#REQUIRED' | '#IMPLIED'
    | (('#FIXED' S)? AttValue) [  VC: Required Attribute ]
    [  VC: Attribute Default Legal ]
    [  WFC: No < in Attribute Values ]
    [  VC: Fixed Attribute Default ]
[61]  conditionalSect ::=  includeSect | ignoreSect
[62]  includeSect ::=  '<![' S? 'INCLUDE' S? '[' extSubsetDecl ']]>'
[63]  ignoreSect ::=  '<![' S? 'IGNORE' S? '[' ignoreSectContents*
']]>'
[64]  ignoreSectContents ::=  Ignore ('<![' ignoreSectContents ']]>'
Ignore)*
[65]  Ignore ::=  Char* - (Char* ('<![' | ']]>') Char*)
[66]  CharRef ::=  '&#' [0-9]+ ';'
    | '&#x' [0-9a-fA-F]+ ';' [  WFC: Legal Character ]
[67]  Reference ::=  EntityRef | CharRef
[68]  EntityRef ::=  '&' Name ';' [  WFC: Entity Declared ]
    [  VC: Entity Declared ]
    [  WFC: Parsed Entity ]
    [  WFC: No Recursion ]
[69]  PEReference ::=  '%' Name ';' [  VC: Entity Declared ]
    [  WFC: No Recursion ]
    [  WFC: In DTD ]
[70]  EntityDecl ::=  GEDecl | PEDecl
[71]  GEDecl ::=  '<!ENTITY' S Name S EntityDef S? '>'
[72]  PEDecl ::=  '<!ENTITY' S '%' S Name S PEDef S? '>'
[73]  EntityDef ::=  EntityValue | (ExternalID NDataDecl?)
[74]  PEDef ::=  EntityValue | ExternalID
[75]  ExternalID ::=  'SYSTEM' S SystemLiteral
    | 'PUBLIC' S PubidLiteral S SystemLiteral
[76]  NDataDecl ::=  S 'NDATA' S Name [  VC: Notation Declared ]
[77]  TextDecl ::=  '<?xml' VersionInfo? EncodingDecl S? '?>'
[78]  extParsedEnt ::=  TextDecl? content
[79]  extPE ::=  TextDecl? extSubsetDecl
[80]  EncodingDecl ::=  S 'encoding' Eq ('"' EncName '"' |  "'"
EncName "'" )
[81]  EncName ::=  [A-Za-z] ([A-Za-z0-9._] | '-')* /*  Encoding name
contains only Latin characters */
[82]  NotationDecl ::=  '<!NOTATION' S Name S (ExternalID |
PublicID) S? '>'
[83]  PublicID ::=  'PUBLIC' S PubidLiteral
[84]  Letter ::=  BaseChar | Ideographic
[85]  BaseChar ::=  [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-
#x00D6] | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
| [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E] | [#x0180-
#x01C3] | [#x01CD-#x01F0] | [#x01F4-#x01F5] | [#x01FA-#x0217]
| [#x0250-#x02A8] | [#x02BB-#x02C1] | #x0386 | [#x0388-#x038A]
```

```
| #x038C | [#x038E-#x03A1] | [#x03A3-#x03CE] | [#x03D0-#x03D6]
| #x03DA | #x03DC | #x03DE | #x03E0 | [#x03E2-#x03F3] | [#x0401-
#x040C] | [#x040E-#x044F] | [#x0451-#x045C] | [#x045E-#x0481]
| [#x0490-#x04C4] | [#x04C7-#x04C8] | [#x04CB-#x04CC] | [#x04D0-
#x04EB] | [#x04EE-#x04F5] | [#x04F8-#x04F9] | [#x0531-#x0556]
| #x0559 | [#x0561-#x0586] | [#x05D0-#x05EA] | [#x05F0-#x05F2]
| [#x0621-#x063A] | [#x0641-#x064A] | [#x0671-#x06B7] | [#x06BA-
#x06BE] | [#x06C0-#x06CE] | [#x06D0-#x06D3] | #x06D5 | [#x06E5-
#x06E6] | [#x0905-#x0939] | #x093D | [#x0958-#x0961] | [#x0985-
#x098C] | [#x098F-#x0990] | [#x0993-#x09A8] | [#x09AA-#x09B0]
| #x09B2 | [#x09B6-#x09B9] | [#x09DC-#x09DD] | [#x09DF-#x09E1]
| [#x09F0-#x09F1] | [#x0A05-#x0A0A] | [#x0A0F-#x0A10] | [#x0A13-
#x0A28] | [#x0A2A-#x0A30] | [#x0A32-#x0A33] | [#x0A35-#x0A36]
| [#x0A38-#x0A39] | [#x0A59-#x0A5C] | #x0A5E | [#x0A72-#x0A74]
| [#x0A85-#x0A8B] | #x0A8D | [#x0A8F-#x0A91] | [#x0A93-#x0AA8]
| [#x0AAA-#x0AB0] | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9] | #x0ABD
| #x0AE0 | [#x0B05-#x0B0C] | [#x0B0F-#x0B10] | [#x0B13-#x0B28]
| [#x0B2A-#x0B30] | [#x0B32-#x0B33] | [#x0B36-#x0B39] | #x0B3D
| [#x0B5C-#x0B5D] | [#x0B5F-#x0B61] | [#x0B85-#x0B8A] | [#x0B8E-
#x0B90] | [#x0B92-#x0B95] | [#x0B99-#x0B9A] | #x0B9C | [#x0B9E-
#x0B9F] | [#x0BA3-#x0BA4] | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5]
| [#x0BB7-#x0BB9] | [#x0C05-#x0C0C] | [#x0C0E-#x0C10] | [#x0C12-
#x0C28] | [#x0C2A-#x0C33] | [#x0C35-#x0C39] | [#x0C60-#x0C61]
| [#x0C85-#x0C8C] | [#x0C8E-#x0C90] | [#x0C92-#x0CA8] | [#x0CAA-
#x0CB3] | [#x0CB5-#x0CB9] | #x0CDE | [#x0CE0-#x0CE1] | [#x0D05-
#x0D0C] | [#x0D0E-#x0D10] | [#x0D12-#x0D28] | [#x0D2A-#x0D39]
| [#x0D60-#x0D61] | [#x0E01-#x0E2E] | #x0E30 | [#x0E32-#x0E33]
| [#x0E40-#x0E45] | [#x0E81-#x0E82] | #x0E84 | [#x0E87-#x0E88]
| #x0E8A | #x0E8D | [#x0E94-#x0E97] | [#x0E99-#x0E9F] | [#x0EA1-
#x0EA3] | #x0EA5 | #x0EA7 | [#x0EAA-#x0EAB] | [#x0EAD-#x0EAE]
| #x0EB0 | [#x0EB2-#x0EB3] | #x0EBD | [#x0EC0-#x0EC4] | [#x0F40-
#x0F47] | [#x0F49-#x0F69] | [#x10A0-#x10C5] | [#x10D0-#x10F6]
| #x1100 | [#x1102-#x1103] | [#x1105-#x1107] | #x1109 | [#x110B-
#x110C] | [#x110E-#x1112] | #x113C | #x113E | #x1140 | #x114C
| #x114E | #x1150 | [#x1154-#x1155] | #x1159 | [#x115F-#x1161]
| #x1163 | #x1165 | #x1167 | #x1169 | [#x116D-#x116E] | [#x1172-
#x1173] | #x1175 | #x119E | #x11A8 | #x11AB | [#x11AE-#x11AF]
| [#x11B7-#x11B8] | #x11BA | [#x11BC-#x11C2] | #x11EB | #x11F0
| #x11F9 | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9] | [#x1F00-#x1F15]
| [#x1F18-#x1F1D] | [#x1F20-#x1F45] | [#x1F48-#x1F4D] | [#x1F50-
#x1F57] | #x1F59 | #x1F5B | #x1F5D | [#x1F5F-#x1F7D] | [#x1F80-
#x1FB4] | [#x1FB6-#x1FBC] | #x1FBE | [#x1FC2-#x1FC4] | [#x1FC6-
#x1FCC] | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB] | [#x1FE0-#x1FEC]
| [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC] | #x2126 | [#x212A-#x212B]
| #x212E | [#x2180-#x2182] | [#x3041-#x3094] | [#x30A1-#x30FA]
| [#x3105-#x312C] | [#xAC00-#xD7A3]
[86]  Ideographic ::=  [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]
[87]  CombiningChar ::=  [#x0300-#x0345] | [#x0360-#x0361]
| [#x0483-#x0486] | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-
```

```
↪#x05BD] | #x05BF | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652]
↪| #x0670 | [#x06D6-#x06DC] | [#x06DD-#x06DF] | [#x06E0-#x06E4]
↪| [#x06E7-#x06E8] | [#x06EA-#x06ED] | [#x0901-#x0903] | #x093C
↪| [#x093E-#x094C] | #x094D | [#x0951-#x0954] | [#x0962-#x0963]
↪| [#x0981-#x0983] | #x09BC | #x09BE | #x09BF | [#x09C0-#x09C4]
↪| [#x09C7-#x09C8] | [#x09CB-#x09CD] | #x09D7 | [#x09E2-#x09E3]
↪| #x0A02 | #x0A3C | #x0A3E | #x0A3F | [#x0A40-#x0A42] | [#x0A47-
↪#x0A48] | [#x0A4B-#x0A4D] | [#x0A70-#x0A71] | [#x0A81-#x0A83]
↪| #x0ABC | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9] | [#x0ACB-#x0ACD]
↪| [#x0B01-#x0B03] | #x0B3C | [#x0B3E-#x0B43] | [#x0B47-#x0B48]
↪| [#x0B4B-#x0B4D] | [#x0B56-#x0B57] | [#x0B82-#x0B83] | [#x0BBE-
↪#x0BC2] | [#x0BC6-#x0BC8] | [#x0BCA-#x0BCD] | #x0BD7 | [#x0C01-
↪#x0C03] | [#x0C3E-#x0C44] | [#x0C46-#x0C48] | [#x0C4A-#x0C4D]
↪| [#x0C55-#x0C56] | [#x0C82-#x0C83] | [#x0CBE-#x0CC4] | [#x0CC6-
↪#x0CC8] | [#x0CCA-#x0CCD] | [#x0CD5-#x0CD6] | [#x0D02-#x0D03]
↪| [#x0D3E-#x0D43] | [#x0D46-#x0D48] | [#x0D4A-#x0D4D] | #x0D57
↪| #x0E31 | [#x0E34-#x0E3A] | [#x0E47-#x0E4E] | #x0EB1 | [#x0EB4-
↪#x0EB9] | [#x0EBB-#x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-#x0F19]
↪| #x0F35 | #x0F37 | #x0F39 | #x0F3E | #x0F3F | [#x0F71-#x0F84]
↪| [#x0F86-#x0F8B] | [#x0F90-#x0F95] | #x0F97 | [#x0F99-#x0FAD]
↪| [#x0FB1-#x0FB7] | #x0FB9 | [#x20D0-#x20DC] | #x20E1 | [#x302A-
↪#x302F] | #x3099 | #x309A
[88]  Digit ::=  [#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9]
↪| [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F] | [#x0AE6-
↪#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF] | [#x0C66-#x0C6F]
↪| [#x0CE6-#x0CEF] | [#x0D66-#x0D6F] | [#x0E50-#x0E59] | [#x0ED0-
↪#x0ED9] | [#x0F20-#x0F29]
[89]  Extender ::=  #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640
↪| #x0E46 | #x0EC6 | #x3005 | [#x3031-#x3035] | [#x309D-#x309E]
↪| [#x30FC-#x30FE]
```

Приложение 2



Официальная спецификация XML Schema

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- XML Schema schema for XML Schemas: Part 1: Structures -->
<!DOCTYPE schema PUBLIC "-//W3C//DTD XMLSCHEMA 200010//EN"
"XMLSchema.dtd" [
<!--
     keep this schema XML1.0 DTD valid
  -->
<!ATTLIST element xmlns:x CDATA #IMPLIED>
<!-- provide ID type information even for parsers which only read
↳the internal subset -->
<!ATTLIST schema          id  ID  #IMPLIED>
<!ATTLIST complexType     id  ID  #IMPLIED>
<!ATTLIST complexContent  id  ID  #IMPLIED>
<!ATTLIST simpleContent   id  ID  #IMPLIED>
<!ATTLIST extension       id  ID  #IMPLIED>
<!ATTLIST element         id  ID  #IMPLIED>
<!ATTLIST group           id  ID  #IMPLIED>
<!ATTLIST all             id  ID  #IMPLIED>
<!ATTLIST choice          id  ID  #IMPLIED>
<!ATTLIST sequence        id  ID  #IMPLIED>
<!ATTLIST any             id  ID  #IMPLIED>
<!ATTLIST anyAttribute    id  ID  #IMPLIED>
<!ATTLIST attribute       id  ID  #IMPLIED>
<!ATTLIST attributeGroup  id  ID  #IMPLIED>
<!ATTLIST unique          id  ID  #IMPLIED>
<!ATTLIST key             id  ID  #IMPLIED>
<!ATTLIST keyref          id  ID  #IMPLIED>
<!ATTLIST selector        id  ID  #IMPLIED>
```

```
<!ATTLIST field           id  ID  #IMPLIED>
<!ATTLIST include         id  ID  #IMPLIED>
<!ATTLIST import          id  ID  #IMPLIED>
<!ATTLIST redefine        id  ID  #IMPLIED>
<!ATTLIST notation        id  ID  #IMPLIED>
]>
<schema targetNamespace="http://www.w3.org/2000/10/XMLSchema"
↳blockDefault="#all" elementFormDefault="qualified" version="Id:
↳XMLSchema.xsd,v 1.26 2000/10/23 08:58:09 ht Exp "
↳xmlns="http://www.w3.org/2000/10/XMLSchema">


 <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/2000/CR-xmlschema-1-
↳20001024/structures.html">
   The schema corresponding to this document is normative,
   with respect to the syntactic constraints it expresses in the
   XML Schema language.  The documentation (within
↳&lt;documentation> elements)
   below, is not normative, but rather highlights important aspects
of the W3C Recommendation of which this is a part</documentation>
 </annotation>


 <annotation>
   <documentation xml:lang="en">
   The simpleType element and all of its members are defined
   in datatypes.xsd</documentation>
 </annotation>


 <include schemaLocation="datatypes.xsd"/>


 <import namespace="http://www.w3.org/XML/1998/namespace"
         schemaLocation="http://www.w3.org/2000/10/xml.xsd">
   <annotation>
     <documentation xml:lang="en">
       Get access to the xml: attribute groups for xml:lang
       as declared on 'documentation' below
     </documentation>
   </annotation>
 </import>
```

```
<complexType name="openAttrs">
  <annotation>
    <documentation xml:lang="en">
      This type is extended by almost all schema types
      to allow attributes from other namespaces to be
      added to user schemas.
    </documentation>
  </annotation>
  <complexContent>
    <restriction base="anyType">
      <anyAttribute namespace="##other" processContents="lax"/>
    </restriction>
  </complexContent>
</complexType>

<complexType name="annotated">
  <annotation>
    <documentation xml:lang="en">
      This type is extended by all types which allow annotation
      other than &lt;schema&gt; itself
    </documentation>
  </annotation>
  <complexContent>
    <extension base="openAttrs">
      <sequence>
       <element ref="annotation" minOccurs="0"/>
      </sequence>
      <attribute name="id" type="ID"/>
    </extension>
  </complexContent>
</complexType>

<element name="schemaTop" abstract="true" type="annotated">
 <annotation>
  <documentation xml:lang="en">
  This abstract element defines an substitution group over the
  elements which occur freely at the top level of schemas.  These
↳are:
  simpleType, complexType, element, attribute, attributeGroup,
↳group, notation
```

```
   All of their types are based on the "annotated" type by
↳extension.</documentation>
  </annotation>
 </element>

 <element name="redefinable" abstract="true"
↳substitutionGroup="schemaTop">
  <annotation>
   <documentation xml:lang="en">
   This abstract element defines a substitution group for the
   elements which can self-redefine (see &lt;redefine>
↳below).</documentation>
  </annotation>
 </element>

 <simpleType name="formChoice">
  <annotation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <restriction base="NMTOKEN">
   <enumeration value="qualified"/>
   <enumeration value="unqualified"/>
  </restriction>
 </simpleType>

 <element name="schema" id="schema">
  <annotation>
    <documentation xml:lang="en"
source="http://www.w3.org/TR/xmlschema-1/#element-schema"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="openAttrs">
     <sequence>
      <choice minOccurs="0" maxOccurs="unbounded">
       <element ref="include"/>
       <element ref="import"/>
       <element ref="redefine"/>
       <element ref="annotation"/>
```

```
      </choice>
      <sequence minOccurs="0" maxOccurs="unbounded">
       <element ref="schemaTop"/>
       <element ref="annotation" minOccurs="0"
↳maxOccurs="unbounded"/>
      </sequence>
     </sequence>
     <attribute name="targetNamespace" type="uriReference"/>
     <attribute name="version" type="token"/>
     <attribute name="finalDefault" type="derivationSet"
↳use="default" value=""/>
     <attribute name="blockDefault" type="blockSet" use="default"
↳value=""/>
     <attribute name="attributeFormDefault" type="formChoice"
↳use="default" value="unqualified"/>
     <attribute name="elementFormDefault" type="formChoice"
↳use="default" value="unqualified"/>
     <attribute name="id" type="ID"/>
    </extension>
   </complexContent>
  </complexType>

  <key name="element">
   <selector xpath="element"/>
   <field xpath="@name"/>
  </key>

  <key name="attribute">
   <selector xpath="attribute"/>
   <field xpath="@name"/>
  </key>

  <key name="type">
   <selector xpath="complexType|simpleType"/>
   <field xpath="@name"/>
  </key>

  <key name="group">
   <selector xpath="group"/>
   <field xpath="@name"/>
  </key>
```

```
   <key name="attributeGroup">
   <selector xpath="attributeGroup"/>
   <field xpath="@name"/>
  </key>

  <key name="notation">
   <selector xpath="notation"/>
   <field xpath="@name"/>
  </key>

  <key name="identityConstraint">
   <selector xpath=".//key|.//unique|.//keyref"/>
   <field xpath="@name"/>
  </key>

 </element>

 <simpleType name="allNNI">
  <annotation><documentation xml:lang="en">
   for maxOccurs</documentation></annotation>
  <union memberTypes="nonNegativeInteger">
   <simpleType>
    <restriction base="NMTOKEN">
     <enumeration value="unbounded"/>
    </restriction>
   </simpleType>
  </union>
 </simpleType>

 <attributeGroup name="occurs">
  <annotation><documentation xml:lang="en">
   for all particles</documentation></annotation>
  <attribute name="minOccurs" type="nonNegativeInteger"
↳use="default" value="1"/>
  <attribute name="maxOccurs" type="allNNI" use="default"
↳value="1"/>
 </attributeGroup>

 <attributeGroup name="defRef">
  <annotation><documentation xml:lang="en">
```

```
   for element, group and attributeGroup,
   which both define and reference</documentation></annotation>
  <attribute name="name" type="NCName"/>
  <attribute name="ref" type="QName"/>
</attributeGroup>

<group name="typeDefParticle">
 <annotation>
   <documentation xml:lang="en">
  'complexType' uses this</documentation></annotation>
 <choice>
  <element name="group" type="groupRef"/>
  <element ref="all"/>
  <element ref="choice"/>
  <element ref="sequence"/>
 </choice>
</group>

<group name="groupDefParticle">
 <annotation>
   <documentation xml:lang="en">
  'topLevelGroup' uses this</documentation></annotation>
 <choice>
  <element ref="all"/>
  <element ref="choice"/>
  <element ref="sequence"/>
 </choice>
</group>

<group name="nestedParticle">
 <choice>
  <element name="element" type="localElement"/>
  <element name="group" type="groupRef"/>
  <element ref="choice"/>
  <element ref="sequence"/>
  <element ref="any"/>
 </choice>
</group>

<group name="particle">
 <choice>
```

```
   <element name="element" type="localElement"/>
   <element name="group" type="groupRef"/>
   <element ref="all"/>
   <element ref="choice"/>
   <element ref="sequence"/>
   <element ref="any"/>
  </choice>
 </group>

 <complexType name="attribute">
  <complexContent>
   <extension base="annotated">
    <sequence>
     <element name="simpleType" minOccurs="0"
↳type="localSimpleType"/>
    </sequence>
    <attributeGroup ref="defRef"/>
    <attribute name="type" type="QName"/>
    <attribute name="use" use="default" value="optional">
     <simpleType>
      <restriction base="NMTOKEN">
       <enumeration value="prohibited"/>
       <enumeration value="optional"/>
       <enumeration value="required"/>
       <enumeration value="default"/>
       <enumeration value="fixed"/>
      </restriction>
     </simpleType>
    </attribute>
    <attribute name="value" use="optional" type="string"/>
    <attribute name="form" type="formChoice"/>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="topLevelAttribute">
  <complexContent>
   <restriction base="attribute">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
```

```
     <element name="simpleType" minOccurs="0"
↳type="localSimpleType"/>
    </sequence>
    <attribute name="ref" use="prohibited"/>
    <attribute name="form" use="prohibited"/>
    <attribute name="use" use="prohibited"/>
    <attribute name="name" use="required" type="NCName"/>
   </restriction>
  </complexContent>
 </complexType>

 <group name="attrDecls">
  <sequence>
   <choice minOccurs="0" maxOccurs="unbounded">
    <element name="attribute" type="attribute"/>
    <element name="attributeGroup" type="attributeGroupRef"/>
   </choice>
   <element ref="anyAttribute" minOccurs="0"/>
  </sequence>
 </group>

 <element name="anyAttribute" type="wildcard" id="anyAttribute">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-anyAttribute"/>
  </annotation>
 </element>

 <group name="complexTypeModel">
  <choice>
      <element ref="simpleContent"/>
      <element ref="complexContent"/>
      <sequence>
       <annotation>
        <documentation xml:lang="en">
   This branch is short for
   &lt;complexContent>
   &lt;restriction base="anyType">
   ...
   &lt;/restriction>
```

```
   &lt;/complexContent></documentation>
       </annotation>
       <group ref="typeDefParticle" minOccurs="0"/>
       <group ref="attrDecls"/>
      </sequence>
  </choice>
 </group>

 <complexType name="complexType" abstract="true">
  <complexContent>
   <extension base="annotated">
    <group ref="complexTypeModel"/>
    <attribute name="name" type="NCName">
     <annotation>
      <documentation xml:lang="en">
      Will be restricted to required or forbidden</documentation>
     </annotation>
    </attribute>
    <attribute name="mixed" type="boolean" use="default"
↳value="false">
     <annotation>
      <documentation xml:lang="en">
      Not allowed if simpleContent child is chosen.
      May be overriden by setting on complexContent
↳child.</documentation>
    </annotation>
    </attribute>
    <attribute name="abstract" type="boolean" use="default"
↳value="false"/>
    <attribute name="final" type="derivationSet"/>
    <attribute name="block" type="derivationSet" use="default"
↳value=""/>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="topLevelComplexType">
  <complexContent>
   <restriction base="complexType">
    <sequence>
```

```
    <element ref="annotation" minOccurs="0"/>
    <group ref="complexTypeModel"/>
   </sequence>
   <attribute name="name" type="NCName" use="required"/>
  </restriction>
 </complexContent>
</complexType>

<complexType name="localComplexType">
 <complexContent>
  <restriction base="complexType">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="complexTypeModel"/>
   </sequence>
   <attribute name="name" use="prohibited"/>
  </restriction>
 </complexContent>
</complexType>

<complexType name="restrictionType">
 <complexContent>
  <extension base="annotated">
   <sequence>
    <choice>
     <group ref="typeDefParticle" minOccurs="0"/>
     <group ref="simpleRestrictionModel" minOccurs="0"/>
    </choice>
    <group ref="attrDecls"/>
   </sequence>
   <attribute name="base" type="QName" use="required"/>
  </extension>
 </complexContent>
</complexType>

<complexType name="complexRestrictionType">
 <complexContent>
  <restriction base="restrictionType">
   <sequence>
```

```
     <element ref="annotation" minOccurs="0"/>
     <group ref="typeDefParticle" minOccurs="0"/>
     <group ref="attrDecls"/>
    </sequence>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="extensionType">
  <complexContent>
   <extension base="annotated">
    <sequence>
     <group ref="typeDefParticle" minOccurs="0"/>
     <group ref="attrDecls"/>
    </sequence>
    <attribute name="base" type="QName" use="required"/>
   </extension>
  </complexContent>
 </complexType>

 <element name="complexContent" id="complexContent">
  <annotation>
   <documentation xml:lang="en"
source="http://www.w3.org/TR/xmlschema-1/#element-
complexContent"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="annotated">
     <choice>
      <element name="restriction" type="complexRestrictionType"/>
      <element name="extension" type="extensionType"/>
     </choice>
     <attribute name="mixed" type="boolean">
      <annotation>
       <documentation xml:lang="en">
       Overrides any setting on complexType parent.</documentation>
      </annotation>
    </attribute>
    </extension>
```

```
   </complexContent>
  </complexType>
 </element>

 <complexType name="simpleRestrictionType">
  <complexContent>
   <restriction base="restrictionType">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <group ref="simpleRestrictionModel" minOccurs="0"/>
     <group ref="attrDecls"/>
    </sequence>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="simpleExtensionType">
  <complexContent>
   <restriction base="extensionType">
    <sequence>
     <annotation>
      <documentation xml:lang="en">
      No typeDefParticle group reference</documentation>
     </annotation>
     <element ref="annotation" minOccurs="0"/>
     <group ref="attrDecls"/>
    </sequence>
   </restriction>
  </complexContent>
 </complexType>

 <element name="simpleContent" id="simpleContent">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-simpleContent"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="annotated">
     <choice>
```

```
      <element name="restriction" type="simpleRestrictionType"/>
      <element name="extension" type="simpleExtensionType"/>
     </choice>
    </extension>
   </complexContent>
  </complexType>
 </element>

 <element name="complexType" substitutionGroup="redefinable"
↳type="topLevelComplexType" id="complexType">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-complexType"/>
  </annotation>
 </element>

 <simpleType name="derivationControl">
  <annotation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <restriction base="NMTOKEN">
   <enumeration value="substitution"/>
   <enumeration value="extension"/>
   <enumeration value="restriction"/>
  </restriction>
 </simpleType>

 <simpleType name="reducedDerivationControl">
  <annotation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <restriction base="derivationControl">
   <enumeration value="extension"/>
   <enumeration value="restriction"/>
  </restriction>
 </simpleType>

 <simpleType name="blockSet">
  <annotation>
   <documentation xml:lang="en">
```

```
   #all or (possibly empty) subset of {substitution, extension,
   restriction}</documentation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <union>
   <simpleType>
    <restriction base="token">
     <enumeration value="#all"/>
    </restriction>
   </simpleType>
   <simpleType>
    <list itemType="derivationControl"/>
   </simpleType>
  </union>
 </simpleType>

 <simpleType name="derivationSet">
  <annotation>
   <documentation xml:lang="en">
   #all or (possibly empty) subset of {extension,
⮷restriction}</documentation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <union>
   <simpleType>
    <restriction base="token">
     <enumeration value="#all"/>
    </restriction>
   </simpleType>
   <simpleType>
    <list itemType="reducedDerivationControl"/>
   </simpleType>
  </union>
 </simpleType>

 <complexType name="element" abstract="true">
  <annotation>
   <documentation xml:lang="en">
```

```
   The element element can be used either
   at the toplevel to define an element-type binding globally,
   or within a content model to either reference a globally-defined
   element or type or declare an element-type binding locally.
   The ref form is not allowed at the top level.</documentation>
  </annotation>

  <complexContent>
   <extension base="annotated">
    <sequence>
     <choice minOccurs="0">
      <element name="simpleType" type="localSimpleType"/>
      <element name="complexType" type="localComplexType"/>
     </choice>
     <element ref="identityConstraint" minOccurs="0"
↳maxOccurs="unbounded"/>
    </sequence>
    <attributeGroup ref="defRef"/>
    <attribute name="type" type="QName"/>
    <attribute name="substitutionGroup" type="QName"/>
    <attributeGroup ref="occurs"/>
    <attribute name="default" type="string"/>
    <attribute name="fixed" type="string"/>
    <attribute name="nullable" type="boolean" use="default"
↳value="false"/>
    <attribute name="abstract" type="boolean" use="default"
↳value="false"/>
    <attribute name="final" type="derivationSet" use="default"
↳value=""/>
    <attribute name="block" type="blockSet" use="default" value=""/>
    <attribute name="form" type="formChoice"/>
   </extension>
  </complexContent>
 </complexType>

 <complexType name="topLevelElement">
  <complexContent>
   <restriction base="element">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <choice minOccurs="0">
```

```
      <element name="simpleType" type="localSimpleType"/>
      <element name="complexType" type="localComplexType"/>
     </choice>
     <element ref="identityConstraint" minOccurs="0"
↳maxOccurs="unbounded"/>
    </sequence>
    <attribute name="ref" use="prohibited"/>
    <attribute name="form" use="prohibited"/>
    <attribute name="minOccurs" use="prohibited"/>
    <attribute name="maxOccurs" use="prohibited"/>
    <attribute name="name" use="required" type="NCName"/>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="localElement">
  <complexContent>
   <restriction base="element">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <choice minOccurs="0">
      <element name="simpleType" type="localSimpleType"/>
      <element name="complexType" type="localComplexType"/>
     </choice>
     <element ref="identityConstraint" minOccurs="0"
↳maxOccurs="unbounded"/>
    </sequence>
    <attribute name="substitutionGroup" use="prohibited"/>
    <attribute name="final" use="prohibited"/>
   </restriction>
  </complexContent>
 </complexType>

 <element name="element" type="topLevelElement"
↳substitutionGroup="schemaTop" id="element">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-element"/>
  </annotation>
 </element>
```

```
<complexType name="group" abstract="true">
 <annotation>
  <documentation xml:lang="en">
  group type for explicit groups, named top-level groups and
  group references</documentation>
 </annotation>
 <complexContent>
  <extension base="annotated">
   <group ref="particle" minOccurs="0" maxOccurs="unbounded"/>
   <attributeGroup ref="defRef"/>
   <attributeGroup ref="occurs"/>
  </extension>
 </complexContent>
</complexType>

<complexType name="realGroup">
 <complexContent>
  <restriction base="group">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="groupDefParticle" minOccurs="0" maxOccurs="1"/>
   </sequence>
  </restriction>
 </complexContent>
</complexType>

<complexType name="namedGroup">
 <complexContent>
  <restriction base="realGroup">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="groupDefParticle" minOccurs="1" maxOccurs="1"/>
   </sequence>
   <attribute name="name" use="required" type="NCName"/>
   <attribute name="ref" use="prohibited"/>
   <attribute name="minOccurs" use="prohibited"/>
   <attribute name="maxOccurs" use="prohibited"/>
  </restriction>
 </complexContent>
</complexType>
```

```
 <complexType name="groupRef">
  <complexContent>
   <restriction base="realGroup">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
    </sequence>
    <attribute name="ref" use="required" type="QName"/>
    <attribute name="name" use="prohibited"/>
   </restriction>
  </complexContent>
 </complexType>

 <complexType name="explicitGroup">
  <annotation>
   <documentation xml:lang="en">
   group type for the three kinds of group</documentation>
  </annotation>
  <complexContent>
   <restriction base="group">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
     <group ref="nestedParticle" minOccurs="0"
↳maxOccurs="unbounded"/>
    </sequence>
    <attribute name="name" type="NCName" use="prohibited"/>
    <attribute name="ref" type="QName" use="prohibited"/>
   </restriction>
  </complexContent>
 </complexType>

 <element name="all" id="all">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-all"/>
  </annotation>
  <complexType>
   <annotation>
    <documentation xml:lang="en">
   Only elements allowed inside</documentation>
   </annotation>
```

```
   <complexContent>
    <restriction base="explicitGroup">
     <sequence>
      <element ref="annotation" minOccurs="0"/>
      <element name="element" minOccurs="0" maxOccurs="unbounded">
       <complexType>
        <annotation>
         <documentation xml:lang="en">restricted
↳max/min</documentation>
        </annotation>
        <complexContent>
         <restriction base="localElement">
          <sequence>
           <element ref="annotation" minOccurs="0"/>
           <choice minOccurs="0">
            <element name="simpleType" type="localSimpleType"/>
            <element name="complexType" type="localComplexType"/>
           </choice>
           <element ref="identityConstraint" minOccurs="0"
↳maxOccurs="unbounded"/>
          </sequence>
          <attribute name="minOccurs" use="default" value="1">
           <simpleType>
            <restriction base="nonNegativeInteger">
             <enumeration value="0"/>
             <enumeration value="1"/>
            </restriction>
           </simpleType>
          </attribute>
          <attribute name="maxOccurs" use="default" value="1">
           <simpleType>
            <restriction base="allNNI">
             <enumeration value="0"/>
             <enumeration value="1"/>
            </restriction>
           </simpleType>
          </attribute>
         </restriction>
        </complexContent>
       </complexType>
```

```
      </element>
     </sequence>
     <attribute name="minOccurs" use="default" value="1">
      <simpleType>
       <restriction base="nonNegativeInteger">
        <enumeration value="1"/>
       </restriction>
      </simpleType>
     </attribute>
     <attribute name="maxOccurs" use="default" value="1">
      <simpleType>
       <restriction base="allNNI">
        <enumeration value="1"/>
       </restriction>
      </simpleType>
     </attribute>
    </restriction>
   </complexContent>
  </complexType>
 </element>

 <element name="choice" type="explicitGroup" id="choice">
  <annotation>
   <documentation xml:lang="en"
source="http://www.w3.org/TR/xmlschema-1/#element-choice"/>
  </annotation>
 </element>

 <element name="sequence" type="explicitGroup" id="sequence">
  <annotation>
   <documentation xml:lang="en"
source="http://www.w3.org/TR/xmlschema-1/#element-sequence"/>
  </annotation>
 </element>

 <element name="group" substitutionGroup="redefinable"
type="namedGroup" id="group">
  <annotation>
   <documentation xml:lang="en"
source="http://www.w3.org/TR/xmlschema-1/#element-group"/>
```

```
  </annotation>
 </element>

 <complexType name="wildcard">
  <complexContent>
   <extension base="annotated">
    <attribute name="namespace" type="namespaceList" use="default"
↳value="##any"/>
    <attribute name="processContents" use="default" value="strict">
     <simpleType>
      <restriction base="NMTOKEN">
       <enumeration value="skip"/>
       <enumeration value="lax"/>
       <enumeration value="strict"/>
      </restriction>
     </simpleType>
    </attribute>
   </extension>
  </complexContent>
 </complexType>

 <element name="any" id="any">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-any"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="wildcard">
     <attributeGroup ref="occurs"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

  <annotation>
   <documentation xml:lang="en">
   simple type for the value of the 'namespace' attr of
   'any' and 'anyAttribute'</documentation>
  </annotation>
```

```
  <annotation>
   <documentation xml:lang="en">
   Value is
          ##any      - - any non-conflicting WFXML/attribute at all

          ##other    - - any non-conflicting WFXML/attribute from
                           namespace other than targetNS

          ##local    - - any unqualified non-conflicting
↳WFXML/attribute

          one or     - - any non-conflicting WFXML/attribute from
              more URI        the listed namespaces
              references
              (space separated)

    ##targetNamespace or ##local may appear in the above list, to
        refer to the targetNamespace of the enclosing
        schema or an absent targetNamespace
↳respectively</documentation>
  </annotation>

 <simpleType name="namespaceList">
  <annotation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <union>
   <simpleType>
    <restriction base="token">
     <enumeration value="##any"/>
     <enumeration value="##other"/>
    </restriction>
   </simpleType>
   <simpleType>
    <list>
     <simpleType>
      <union memberTypes="uriReference">
       <simpleType>
        <restriction base="token">
```

```
        <enumeration value="##targetNamespace"/>
        <enumeration value="##local"/>
       </restriction>
      </simpleType>
     </union>
    </simpleType>
   </list>
  </simpleType>
 </union>
</simpleType>

<element name="attribute" substitutionGroup="schemaTop"
↳type="topLevelAttribute" id="attribute">
 <annotation>
  <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-attribute"/>
 </annotation>
</element>

<complexType name="attributeGroup" abstract="true">
 <complexContent>
  <extension base="annotated">
   <group ref="attrDecls"/>
   <attributeGroup ref="defRef"/>
  </extension>
 </complexContent>
</complexType>

<complexType name="namedAttributeGroup">
 <complexContent>
  <restriction base="attributeGroup">
   <sequence>
    <element ref="annotation" minOccurs="0"/>
    <group ref="attrDecls"/>
   </sequence>
   <attribute name="name" use="required" type="NCName"/>
   <attribute name="ref" use="prohibited"/>
  </restriction>
 </complexContent>
</complexType>
```

```
 <complexType name="attributeGroupRef">
  <complexContent>
   <restriction base="attributeGroup">
    <sequence>
     <element ref="annotation" minOccurs="0"/>
    </sequence>
    <attribute name="ref" use="required" type="QName"/>
    <attribute name="name" use="prohibited"/>
   </restriction>
  </complexContent>
 </complexType>

 <element name="attributeGroup" type="namedAttributeGroup"
↳substitutionGroup="redefinable" id="attributeGroup">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-
↳attributeGroup"/>
  </annotation>
 </element>

 <element name="include" id="include">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-include"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="annotated">
     <attribute name="schemaLocation" type="uriReference"
↳use="required"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

 <element name="redefine" id="redefine">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-redefine"/>
  </annotation>
```

```
  <complexType>
   <complexContent>
    <extension base="openAttrs">
     <choice minOccurs="0" maxOccurs="unbounded">
      <element ref="annotation"/>
      <element ref="redefinable"/>
     </choice>
     <attribute name="schemaLocation" type="uriReference"
↳use="required"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

 <element name="import" id="import">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-import"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="annotated">
     <attribute name="namespace" type="uriReference"/>
     <attribute name="schemaLocation" type="uriReference"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

 <simpleType name="XPathExprApprox">
  <annotation>
   <documentation xml:lang="en">
   An XPath expression</documentation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <restriction base="string">
   <pattern value="(/|//|\.|\.\.|:|::|\||(\c-[.:/|])+)+">
    <annotation>
     <documentation xml:lang="en">
```

```
   A VERY permissive definition, probably not even
↳right</documentation>
    </annotation>
   </pattern>
  </restriction>
 </simpleType>

 <complexType name="XPathSpec">
  <complexContent>
   <extension base="annotated">
     <attribute name="xpath" type="XPathExprApprox"/>
   </extension>
  </complexContent>
 </complexType>

 <element name="selector" type="XPathSpec" id="selector">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-selector"/>
  </annotation>
 </element>

 <element name="field" id="field" type="XPathSpec">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-field"/>
  </annotation>
 </element>

 <complexType name="keybase">
  <complexContent>
   <extension base="annotated">
    <sequence>
     <element ref="selector"/>
     <element ref="field" minOccurs="1" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
   </extension>
  </complexContent>
```

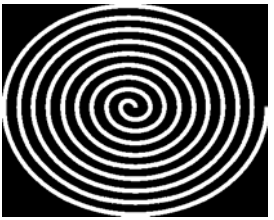
```
 </complexType>
 <element name="identityConstraint" type="keybase" abstract="true"
↳id="identityConstraint">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-
↳identityConstraint"/>
  </annotation>
 </element>


 <element name="unique" substitutionGroup="identityConstraint"
↳id="unique">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-unique"/>
  </annotation>
 </element>
 <element name="key" substitutionGroup="identityConstraint"
↳id="key">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-key"/>
  </annotation>
 </element>
 <element name="keyref" substitutionGroup="identityConstraint"
↳id="keyref">
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-keyref"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="keybase">
     <attribute name="refer" type="QName" use="required"/>
    </extension>
   </complexContent>
  </complexType>
 </element>


 <element name="notation" substitutionGroup="schemaTop"
↳id="notation">
```

```
  <annotation>
   <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-notation"/>
  </annotation>
  <complexType>
   <complexContent>
    <extension base="annotated">
     <attribute name="name" type="NCName" use="required"/>
     <attribute name="public" type="public" use="required"/>
     <attribute name="system" type="uriReference"/>
    </extension>
   </complexContent>
  </complexType>
 </element>

 <simpleType name="public">
  <annotation>
   <documentation xml:lang="en">
   A public identifier, per ISO 8879</documentation>
   <documentation xml:lang="en">
   A utility type, not for public use</documentation>
  </annotation>
  <restriction base="token"/>
 </simpleType>

 <element name="appinfo" id="appinfo">
   <annotation>
     <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-appinfo"/>
   </annotation>
   <complexType mixed="true">
     <sequence minOccurs="0" maxOccurs="unbounded">
       <any processContents="lax"/>
     </sequence>
     <attribute name="source" type="uriReference"/>
   </complexType>
 </element>
```

```
 <element name="documentation"
↳xmlns:x="http://www.w3.org/XML/1998/namespace" id="documentation">
   <annotation>
     <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-documentation"/>
   </annotation>
   <complexType mixed="true">
     <sequence minOccurs="0" maxOccurs="unbounded">
       <any processContents="lax"/>
     </sequence>
     <attribute name="source" type="uriReference"/>
   <attribute ref="x:lang"/>
   </complexType>
 </element>

 <element name="annotation" id="annotation">
   <annotation>
     <documentation xml:lang="en"
↳source="http://www.w3.org/TR/xmlschema-1/#element-annotation"/>
   </annotation>
   <complexType>
     <choice minOccurs="0" maxOccurs="unbounded">
       <element ref="appinfo"/>
       <element ref="documentation"/>
     </choice>
   </complexType>
 </element>

 <annotation>
  <documentation xml:lang="en">
   notations for use within XML Schema schemas</documentation>
 </annotation>

 <notation name="XMLSchemaStructures" public="structures"
↳system="http://www.w3.org/2000/08/XMLSchema.xsd"/>
 <notation name="XML" public="REC-xml-19980210"
↳system="http://www.w3.org/TR/1998/REC-xml-19980210"/>

 <complexType name="anyType" mixed="true">
```

```
  <annotation>
   <documentation xml:lang="en">
   Not the real urType, but as close an approximation as we can
   get in the XML representation</documentation>
  </annotation>
  <sequence>
   <any minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <anyAttribute/>
 </complexType>
</schema>
```

Приложение 3



Официальная спецификация WML

```
<!--
Wireless Markup Language (WML) Document Type Definition.
Copyright Wireless Application Protocol Forum Ltd., 1998,1999.
                      All rights reserved.
WML is an XML language.  Typical usage:
   <?xml version="1.0"?>
   <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
          "http://www.wapforum.org/DTD/wml_1.1.xml">
   <wml>
   ...
   </wml>
Terms and conditions of use are available from the Wireless
Application Protocol Forum Ltd. web site at
http://www.wapforum.org/docs/copyright.htm.
-->
<!ENTITY % length  "CDATA">    <!-- [0-9]+ for pixels or [0-9]+"%" for
                                    percentage length -->
<!ENTITY % vdata   "CDATA">    <!-- attribute value possibly containing
                                    variable references -->
<!ENTITY % HREF    "%vdata;">  <!-- URI, URL or URN designating a hy-
per↳text node. May contain variable references -->
<!ENTITY % boolean      "(true|false)">
<!ENTITY % number  "NMTOKEN">  <!-- a number, with format [0-9]+ -->
<!ENTITY % coreattrs  "id    ID     #IMPLIED
                       class CDATA  #IMPLIED">
<!ENTITY % emph    "em | strong | b | i | u | big | small">
<!ENTITY % layout  "br">
<!ENTITY % text     "#PCDATA | %emph;">
<!-- flow covers "card-level" elements, such as text and images -->
```

```
<!ENTITY % flow     "%text; | %layout; | img | anchor | a | table">
<!-- Task types -->
<!ENTITY % task   "go | prev | noop | refresh">
<!-- Navigation and event elements -->
<!ENTITY % navelmts "do | onevent">
<!--================ Decks and Cards ================-->
<!ELEMENT wml ( head?, template?, card+ )>
<!ATTLIST wml
  xml:lang        NMTOKEN        #IMPLIED
  %coreattrs;
  >
<!-- card intrinsic events -->
<!ENTITY % cardev
 "onenterforward  %HREF;          #IMPLIED
  onenterbackward %HREF;          #IMPLIED
  ontimer         %HREF;          #IMPLIED"
  >
<!-- card field types -->
<!ENTITY % fields  "%flow; | input | select | fieldset">
<!ELEMENT card (onevent*, timer?, (do | p)*)>
<!ATTLIST card
  title           %vdata;       #IMPLIED
  newcontext      %boolean;     "false"
  ordered         %boolean;     "true"
  xml:lang        NMTOKEN       #IMPLIED
  %cardev;
  %coreattrs;
  >
<!--================ Event Bindings ================-->
<!ELEMENT do (%task;)>
<!ATTLIST do
  type          CDATA       #REQUIRED
  label         %vdata;     #IMPLIED
  name          NMTOKEN     #IMPLIED
  optional      %boolean;   "false"
  xml:lang      NMTOKEN     #IMPLIED
  %coreattrs;
  >
<!ELEMENT onevent (%task;)>
```

```
<!ATTLIST onevent
  type         CDATA       #REQUIRED
  %coreattrs;
  >
<!--================ Deck-level declarations ================-->
<!ELEMENT head ( access | meta )+>
<!ATTLIST head
  %coreattrs;
  >
<!ELEMENT template (%navelmts;)*>
<!ATTLIST template
  %cardev;
  %coreattrs;
  >
<!ELEMENT access EMPTY>
<!ATTLIST access
  domain       CDATA       #IMPLIED
  path         CDATA       #IMPLIED
  %coreattrs;
  >
<!ELEMENT meta EMPTY>
<!ATTLIST meta
  http-equiv     CDATA      #IMPLIED
  name           CDATA      #IMPLIED
  forua          %boolean;  #IMPLIED
  content        CDATA      #REQUIRED
  scheme         CDATA      #IMPLIED
  %coreattrs;
  >
<!--================ Tasks ================-->
<!ELEMENT go (postfield | setvar)*>
<!ATTLIST go
  href            %HREF;           #REQUIRED
  sendreferer     %boolean;       "false"
  method          (post|get)      "get"
  accept-charset  CDATA           #IMPLIED
  %coreattrs;
  >
<!ELEMENT prev (setvar)*>
```

```
<!ATTLIST prev
  %coreattrs;
  >
<!ELEMENT refresh (setvar)*>
<!ATTLIST refresh
  %coreattrs;
  >
<!ELEMENT noop EMPTY>
<!ATTLIST noop
  %coreattrs;
  >
<!--================ postfield ================-->
<!ELEMENT postfield EMPTY>
<!ATTLIST postfield
  name         %vdata;           #REQUIRED
  value        %vdata;           #REQUIRED
  %coreattrs;
  >
<!--================ variables ================-->
<!ELEMENT setvar EMPTY>
<!ATTLIST setvar
  name         %vdata;           #REQUIRED
  value        %vdata;           #REQUIRED
  %coreattrs;
  >
<!--================ Card Fields ================-->
<!ELEMENT select (optgroup|option)+>
<!ATTLIST select
  title        %vdata;           #IMPLIED
  name         NMTOKEN           #IMPLIED
  value        %vdata;           #IMPLIED
  iname        NMTOKEN           #IMPLIED
  ivalue       %vdata;           #IMPLIED
  multiple     %boolean;         "false"
  tabindex     %number;          #IMPLIED
  xml:lang     NMTOKEN           #IMPLIED
  %coreattrs;
  >
<!ELEMENT optgroup (optgroup|option)+ >
```

```
<!ATTLIST optgroup
  title      %vdata;    #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >
<!ELEMENT option (#PCDATA | onevent)*>
<!ATTLIST option
  value      %vdata;    #IMPLIED
  title      %vdata;    #IMPLIED
  onpick     %HREF;     #IMPLIED
  xml:lang   NMTOKEN    #IMPLIED
  %coreattrs;
  >
<!ELEMENT input EMPTY>
<!ATTLIST input
  name       NMTOKEN           #REQUIRED
  type       (text|password)   "text"
  value      %vdata;           #IMPLIED
  format     CDATA             #IMPLIED
  emptyok    %boolean;         "false"
  size       %number;          #IMPLIED
  maxlength  %number;          #IMPLIED
  tabindex   %number;          #IMPLIED
  title      %vdata;           #IMPLIED
  xml:lang   NMTOKEN           #IMPLIED
  %coreattrs;
  >
<!ELEMENT fieldset (%fields; | do)* >
<!ATTLIST fieldset
  title           %vdata;      #IMPLIED
  xml:lang        NMTOKEN      #IMPLIED
  %coreattrs;
  >
<!ELEMENT timer EMPTY>
<!ATTLIST timer
  name       NMTOKEN           #IMPLIED
  value      %vdata;           #REQUIRED
  %coreattrs;
  >
```

```
<!--================ Images ================-->
<!ENTITY % IAlign "(top|middle|bottom)" >
<!ELEMENT img EMPTY>
<!ATTLIST img
  alt        %vdata;      #REQUIRED
  src        %HREF;       #REQUIRED
  localsrc   %vdata;      #IMPLIED
  vspace     %length;     "0"
  hspace     %length;     "0"
  align      %IAlign;     "bottom"
  height     %length;     #IMPLIED
  width      %length;     #IMPLIED
  xml:lang   NMTOKEN      #IMPLIED
  %coreattrs;
  >
<!--================ Anchor ================-->
<!ELEMENT anchor ( #PCDATA | br | img | go | prev | refresh )*>
<!ATTLIST anchor
  title        %vdata;      #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >
<!ELEMENT a ( #PCDATA | br | img )*>
<!ATTLIST a
  href         %HREF;       #REQUIRED
  title        %vdata;      #IMPLIED
  xml:lang     NMTOKEN      #IMPLIED
  %coreattrs;
  >
<!--================ Tables ================-->
<!ELEMENT table (tr)+>
<!ATTLIST table
  title       %vdata;    #IMPLIED
  align       CDATA      #IMPLIED
  columns     %number;   #REQUIRED
  xml:lang    NMTOKEN    #IMPLIED
  %coreattrs;
  >
<!ELEMENT tr (td)+>
```

```
<!ATTLIST tr
  %coreattrs;
  >
<!ELEMENT td ( %text; | %layout; | img | anchor | a )*>
<!ATTLIST td
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!--================ Text layout and line breaks ================-->
<!ELEMENT em     (%flow;)*>
<!ATTLIST em
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT strong (%flow;)*>
<!ATTLIST strong
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT b      (%flow;)*>
<!ATTLIST b
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT i      (%flow;)*>
<!ATTLIST i
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT u      (%flow;)*>
<!ATTLIST u
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ELEMENT big    (%flow;)*>
<!ATTLIST big
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
```

```
<!ELEMENT small  (%flow;)*>
<!ATTLIST small
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ENTITY % TAlign "(left|right|center)">
<!ENTITY % WrapMode "(wrap|nowrap)" >
<!ELEMENT p (%fields; | do)*>
<!ATTLIST p
  align       %TAlign;     "left"
  mode        %WrapMode;   #IMPLIED
  xml:lang    NMTOKEN      #IMPLIED
  %coreattrs;
  >
<!ELEMENT br EMPTY>
<!ATTLIST br
  xml:lang        NMTOKEN       #IMPLIED
  %coreattrs;
  >
<!ENTITY quot  "&#34;">     <!-- quotation mark -->
<!ENTITY amp   "&#38;#38;"> <!-- ampersand -->
<!ENTITY apos  "&#39;">     <!-- apostrophe -->
<!ENTITY lt    "&#38;#60;"> <!-- less than -->
<!ENTITY gt    "&#62;">     <!-- greater than -->
<!ENTITY nbsp  " ">    <!-- non-breaking space -->
<!ENTITY shy   "&#173;">    <!-- soft hyphen (discretionary hyphen) -->
<!--
Copyright Wireless Application Protocol Forum Ltd., 1998,1999.
                    All rights reserved.
-->
```

Предметный указатель