Computers are not intelligent. They only think they are.
—Unknown

One thing XML aims to solve is human error. Because of XML's structure and rigidity as a language, there isn't much room for error on the part of XML developers. If you've ever encountered an error at the bank (in their favor!), you can no doubt appreciate the significance of errors in critical computer systems. XML is rapidly being integrated into all kinds of computer systems, including financial systems used by banks. The rigidity of XML as a markup language will no doubt make these systems more robust. The facet of XML that allows errors to be detected is the schema, which is a construct that allows XML developers to define the format and structure of XML data.

This hour introduces you to schemas, including the two major types that are used to define data for XML documents. The first of these schema types, DTDs, is examined in detail in this hour, while the latter is saved for a later lesson. This hour explores the inner workings of DTDs and shows you how to create DTDs from scratch.

In this hour, you'll learn

- How XML allows you to create custom markup languages
- The role of schemas in XML data modeling
- The difference between the types of XML schemas
- What constitutes valid and well-formed documents
- How to declare elements and attributes in a DTD
- How to create and use a DTD for a custom markup language

### Creating Your Own Markup Languages

Before you get too far into this hour, I have to make a little confession. When you create an XML document, you aren't really using XML to code the document. Instead, you are using a markup language that was created in XML. In other words, XML is used to create
markup languages that are then used to create XML documents. The term “XML document” is even a little misleading because the type of the document is really determined by the specific markup language used. So, as an example, if I were to create my very own markup language called MML (Michael’s Markup Language), then the documents I create would be considered MML documents, and I would use MML to code those documents. Generally speaking, the documents are still XML documents because MML is an XML-based markup language, but you would refer to the documents as MML documents.

The point of this discussion is not to split hairs regarding the terminology used to describe XML documents. It is intended to help clarify the point that XML is a technology that enables the creation of custom markup languages. If you’re coming from the world of HTML, you probably think in terms of there being only one markup language—HTML. In the XML world, there are thousands of different markup languages, with each of them applicable to a different type of data. As an XML developer, you have the option of using an existing markup language that someone else created using XML, or you can create your own. An XML-based markup language can be as formal as XHTML, the version of HTML that adheres to the rules of XML, or as informal as my simple Tall Tales trivia language.

When you create your own markup language, you are basically establishing which elements (tags) and attributes are used to create documents in that language. Not only is it important to fully describe the different elements and attributes, but you must also describe how they relate to each other. For example, if you are creating a markup language to keep track of sports information so that you can chart your local softball league, you might use tags such as `<schedule>`, `<game>`, `<team>`, `<player>`, and so on. Examples of attributes for the `player` element might include `name`, `hits`, `rbi`, and so on.

By the Way

Just in case you’re thinking of creating your own sports markup language, I might be able to save you some time by directing you to SportsML (Sports Markup Language). This markup language has elements and attributes similar to the ones I described for your hypothetical softball markup language, except SportsML is much broader and covers many different sports. For more information regarding SportsML, please visit the SportsML web site at http://www.sportsml.org/.

The question you might now be asking yourself is how exactly do you create a markup language? In other words, how do you specify the set of elements and attributes for a markup language, along with how they relate to each other? Although you could certainly create sports XML documents using your own elements and attributes, there really needs to be a set of rules somewhere that establishes the format and structure
of documents created in the language. This set of rules is known as the schema for a markup language. A schema describes the exact elements and attributes that are available within a given markup language, along with which attributes are associated with which elements and the relationships between the elements. You can think of a schema as a legal contract between the person who created the markup language and the person who will create documents using that language.

**By the Way**

Although I describe a schema as a “legal contract,” in reality there is nothing legal about schemas. The point is that schemas are very exacting and thorough, and leave nothing to chance in terms of describing the makeup of XML documents—this degree of exacting thoroughness is what we all look for in an ideal legal contract.

**Schemas and XML Data Modeling**

The process of creating a schema for an XML document is known as data modeling because it involves resolving a class of data into elements and attributes that can be used to describe the data in an XML document. Once a data model (schema) is in place for a particular class of data, you can create structured XML documents that adhere to the model. The real importance of schemas is that they allow XML documents to be validated for accuracy. This simply means that a schema allows an XML developer (or an application) to process a document and see if it adheres to the set of constraints laid out in the schema. If not, you know the document could prove to be problematic. A valid XML document is kind of like a stamp of approval that declares the document suitable for use in an XML application. You learn how to validate your own XML documents in Hour 8, “Validating XML Documents.”

To help clarify the role schemas play in XML, let’s consider a practical real-world analogy. Pretend you just met a friend whom you haven’t seen in years and she gives you her email address so that you can get in touch with her later. However, she lists her email address as lucy*stalefishlabs.com. You know that all email addresses consist of a name followed by an “at” symbol (@), followed by a domain name, which means that something is wrong with her email address. The name@domainname format of email addresses is actually a simple schema—you used this schema to “validate” your friend’s email address and determine that it is in error. The obvious fix is to replace the asterisk (*) in her email address with an “at” symbol (@).

You now understand in the simplest of terms how schemas are used to determine the validity of XML documents, but you don’t entirely know why. The main reason schemas are used in XML is to allow machine validation of document structure. In the invalid email example, you were easily able to see a problem because you knew that email addresses can’t have asterisks in them. But how would an email
application be able to make this determination? The developer of the application would have to write specific code to make sure that email addresses are structured to follow a given syntax, such as the name and domain name being separated by an “at” symbol. Whereas an email application developer writes code to check the validity of an email address, an XML document creator uses a schema. This schema can then be used by XML applications to ensure that documents are valid; schemas provide a mechanism to facilitate the process of validating XML documents.

When it comes to creating schemas, there are two primary approaches you can take:

- Document Type Definitions (DTDs)
- XML Schemas (XSDs)

These two schema approaches represent different technologies that make it possible to describe the data model of an XML-based markup language. The next two sections explain each approach in more detail.

**Document Type Definitions (DTDs)**

Warning, I’m about to roll out a new acronym! The new acronym I want to introduce you to now is **DTD**, which stands for **Document Type Definition**. DTDs represent the original approach of creating a schema for XML documents. I say “original approach” because DTDs did not originate with XML; DTDs originated with XML’s predecessor, SGML (Standard General Markup Language). DTDs made their way into XML because it eased the transition from SGML to XML—many SGML tools existed that could be used for XML. Things have changed since the early days of XML, however, and now there is a more powerful approach to establishing schemas than DTDs. Even so, DTDs are still in use so it’s important for you to understand how they work.

*By the Way*

You learn about the more powerful XML Schema approach in the next section, “XML Schema.”

The main drawback to DTDs is that they are based upon a somewhat cryptic language. XML provides a highly structured approach to formatting data, so why should you have to learn an entirely new language to describe XML schemas? I don’t have a good answer to this question except to say that DTDs are a carryover from XML’s beginnings and they still play a role in some XML applications, so you should learn how to use them. The good news is that DTDs are actually quite simple for describing most XML-based markup languages. This is due to the fact that the DTD language is extremely compact, which is why it has a cryptic appearance. Rather than continue to describe DTDs in words, let’s just look at an example in Listing 3.1.
I warned you it was kind of cryptic. However, if you take a moment to read through the DTD code you can actually start to make some sense of it. You might even recognize that this DTD is for the Tall Tales trivia document that you saw in the previous hour. By studying the code, you can see that the word `ELEMENT` precedes each element that can be used in a TTML (Tall Tales Markup Language) document. Also the attributes for the `tt` element are listed after the word `ATTLIST` (line 4); in this case there is only one attribute, `answer` (line 5). Also notice that the three possible values of the `answer` attribute (`a`, `b`, and `c`) are listed out beside the attribute (line 5). Although there are a few strange looking pieces of information in this DTD, such as the `!` at the beginning of each line and `(#PCDATA)` following each element, it’s pretty apparent that DTDs aren’t overly complex.

You learn a great deal more about DTDs later in this hour, so I won’t go into more detail just yet. Instead, we’ll move on and learn about the other approach to data modeling that uses a syntax that should be very familiar to you.

**XML Schema (XSDs)**

XML Schema replaces DTDs with a more powerful and intuitive approach to creating schemas for XML-based markup languages. Schemas created using XML Schema are coded in the XSD (XML Schema Definition) language, and are therefore referred to as XSDs. XML Schema and the XSD language were created by the W3C (World Wide Web Consortium), and represent a considerably more powerful and flexible approach to schemas than DTDs. The idea behind XML Schema is to use XML as the basis for creating schemas. So, instead of using the special DTD language to create a schema, you can use familiar XML elements and attributes that are defined in the XSD language.

An XSD is very similar in purpose to a DTD in that it is used to establish the schema of a class of XML documents. Similar to DTDs, XSDs describe elements and their
HOUR 3: Defining Data with DTD Schemas

content models so that documents can be validated. However, XSDs go several steps beyond DTDs by allowing you to associate data types with elements. In a DTD, element content is pretty much limited to text. An XSD is more flexible in that it can set the data type of elements to specific types, such as integer numbers and dates.

Of course, the most compelling aspect of XSDs is the fact that they are based upon an XML vocabulary (XSD). This means that you create an XSD as an XML document. So, the familiar tag/attribute approach to encoding XML documents is all you need to know to code an XSD document. You still have to learn the specific elements and attributes that comprise the XSD language, but it isn’t too terribly difficult to learn. To give you an example, the code in Listing 3.2 is for an XSD that describes the familiar Tall Tales document.

LISTING 3.2 An XSD Document That Serves as a Schema for the Tall Tales XML Document

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
<xsd:element name="talltales" minOccurs="1" maxOccurs="1">
  <xsd:complexType>
    <xsd:element name="tt">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="question" type="xsd:string" maxOccurs="1" />
          <xsd:element name="a" type="xsd:string" maxOccurs="1" />
          <xsd:element name="b" type="xsd:string" maxOccurs="1" />
          <xsd:element name="c" type="xsd:string" maxOccurs="1" />
        </xsd:sequence>
        <xsd:attribute name="answer" type="answerType" use="required" />
      </xsd:complexType>
    </xsd:element>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

As you can see, XSDs aren’t nearly as compact as DTDs, and can be more difficult to understand initially. The reason for this is because XSDs are considerably more powerful and flexible than DTDs, and with advanced features comes complexity. You learn all about creating XSDs in Hour 7, “Using XML Schema XSDs,” after which this code will make complete sense to you. For now, the main thing to understand
Comparing Schema Technologies

is that XML Schema allows you to use XML code to model data in a more detailed manner than DTDs. For example, in an XSD you can specify exactly the number of times an element is allowed to appear when nested.

Another schema technology exists that helps to simplify the complexities associated with XSDs. I'm referring to RELAX NG, which many people consider to be more powerful, more concise, and easier to use than XML Schema. RELAX NG doesn't yet enjoy the widespread support of DTDs and XSDs but that scenario is likely to change. You learn more about RELAX NG and how it compares to XML Schema in Hour 7, “Using XML Schema.”

Comparing Schema Technologies

Although you haven’t really learned much about the inner workings of DTDs and XSDs, you do understand that they represent two of the fundamental approaches to modeling XML document data. I alluded earlier that DTDs aren’t as powerful as XSDs but there are advantages to DTDs when you consider that they have a long history and also are easier to create. On the other hand, XSDs offer considerable technical advantages over DTDs when you consider how much control they give you when nailing down the specifics of an XML data model. I’d like to take a moment to cover the specific differences between DTDs and XSDs so that you can better understand when to use each. Following is a list of the major differences between DTDs and XSDs:

▶ DTDs are coded using a special language, whereas XSDs are coded in XML.
▶ XSDs can be processed just like any other XML documents.
▶ XSDs support a variety of data types (integer numbers, floating point numbers, Booleans [true/false], dates, and so on), whereas DTDs treat all data as strings or lists of strings.
▶ XSDs present an open-ended data model, which allows you to extend custom markup languages and establish complex relationships between elements without invalidating documents. DTDs employ a closed data model that doesn’t allow for much in the way of extensibility.

This list of differences obviously tips the scales in favor of XSDs. This is not just a matter of me being enamored with XML Schema. Keep in mind that XML Schema was created long after DTDs, so it stands to reason that it would address many of the weaknesses found in DTDs. Knowing this, you might think that it’s a no-brainer choosing XSDs over DTDs to describe the data model for all of your XML documents. Although XSDs represent a more modern approach to XML data modeling, DTDs aren’t quite dead yet.
The one thing DTDs have going for them is the fact that they’ve been around for so long and are so widely accepted. If you think an inferior technology has never been able to survive solely on its longevity and widespread acceptance, think again. Look no further than your VCR to see such a technology in action; Betamax was known to be a more powerful technology but somehow the VHS standard stuck and the better technology lost. Even so, DTDs are being supplanted by XML Schema and other schema standards such as the newer RELAX NG schema technology.

One compelling argument for understanding DTDs is that many XML-based markup languages are still expressed as DTDs. So, if you plan on using any existing XML languages, you may need to be familiar with DTDs in order to understand how they work. The other point to keep in mind is that DTDs are actually very good at describing the structure of XML documents. The fact that XSDs do a better job is not enough to discount DTDs entirely.

The Importance of Document Validation

Before digging deeper into DTDs, I’d like to address a topic that has been already been touched on a few times: document validation. I’ve already mentioned that the primary purpose of schemas is to provide a means of validating XML documents. What I didn’t point out is that an XML document doesn’t necessarily have to be valid in order to be useful. So, there is a little more to the document validation story than I’ve let on.

In addition to document validity, XML establishes the concept of a well-formed document, which is a document that meets all of the general language rules of XML. A well-formed document follows the five XML commandments you learned about in the previous hour, but it doesn’t necessarily qualify as being a valid document. The reason is because a well-formed document doesn’t have to be associated with a schema. Looking at XML documents in this light, it becomes apparent that there are two different degrees of “correctness” with respect to XML documents. The first degree is met by well-formed documents, which must meet the strict language requirements of XML. Taking things a step further leads us to valid documents, which are well-formed documents that also adhere to a schema of some sort (DTD or XSD, for example).

A valid document is always a well-formed document, but the reverse is not always true.

The terms valid and well-formed are extremely important to XML and are used throughout the remainder of the book. Whether or not a document needs to be
valid is up to the particular application in which it is being used. However, all XML
documents must be well-formed or you will undoubtedly encounter errors when
the document is processed. To help keep valid and well-formed documents clear,
remember that well-formed documents must adhere to the general XML rules,
whereas valid documents must also adhere to a schema. You learn how to use DTDs
and XSDs to validate XML documents in Hour 8.

**DTD Construction Basics**

You already know that DTDs represent the original schema technique supported by
XML, and are tightly integrated with the documents they describe. Therefore, to
understand how DTDs are constructed, you must first understand how they relate
to XML documents. In order to use a DTD with a document, you must somehow
associate the DTD with the document. This association is carried out through the
document type declaration, which must be placed at the beginning of an XML
document just after the XML declaration:

```xml
<?xml version="1.0"?>
<!DOCTYPE talltales SYSTEM "talltales.dtd">
```

The second line in this code is the document type declaration for the Tall Tales
document you saw earlier. The main thing to note about this code is how the Tall
Tales DTD is specified.

The terminology surrounding DTDs and document type declarations is admittedly
confusing, so allow me to clarify that DTD stands for document type definition and
contains the actual description of a markup language. A document type declaration
is a line of code in a document that identifies the DTD. So, the big distinction here is
that the definition (DTD) describes your markup language, whereas the declaration
associates it with a document. Got it? Let’s move on!

A DTD describes vital information about the structure of a document using markup declarations, which are lines of code that describe elements, attributes, and the relationship between them. The following types of markup declarations may be used within a DTD:

- The elements allowed in the document
- The attributes that may be assigned to each element
- Entities that are allowed in the document
- Notations that are allowed for use with external entities

Elements and attributes you know about, but the last two markup declarations relate
to entirely new territory. Don’t worry because you learn more about entities and
notations in the next hour, “Digging Deeper into XML Documents.” For now, it's important to understand that the markup declarations within a DTD serve a vital role in allowing documents to be validated against the DTD.

When associating a DTD with a document, there are two approaches the document type declaration can take:

- It can directly include markup declarations in the document that form the internal DTD.
- It can reference external markup declarations that form the external DTD.

These two approaches to declaring a DTD reveal that there are two parts to a DTD: an internal part and an external part. When you refer to the DTD for a document, you are actually referring to the internal and external DTDs taken together. The reason for breaking the DTD into two parts has to do with flexibility. The external DTD typically describes the general structure for a class of documents, whereas the internal DTD is specific to a given document. XML gives preference to the internal DTD, which means you can use it to override declarations in the external DTD.

If you have any experience with CSS (Cascading Style Sheets) in web design, you may recognize a similar structure in DTDs where the internal DTD overrides the external DTD. In CSS style sheets, local styles override any external style sheets.

### Breaking Down a DTD

The following code shows the general syntax of a document type declaration:

```
<!DOCTYPE RootElem SYSTEM ExternalDTDRef [InternalDTDDecl]>
```

You could argue that it isn’t necessary to understand the inner workings of DTDs in order to use XML, and to some extent that is true. In fact, you don’t necessarily have to know anything about schemas to do interesting things with XML. However, it’s impossible to truly understand the XML technology without having a firm grasp on what constitutes an XML-based markup language. And, of course, XML-based markup languages are described using DTDs and other types of schemas.

The external DTD is referenced by ExternalDTDRef, which is the URI (Uniform Resource Identifier) of a file containing the external DTD. The internal DTD corresponds to InternalDTDDecl and is declared between square brackets ([ ]). In addition to the internal and external DTDs, another very important piece of information is mentioned in the document type declaration: the root element. RootElem identifies the root element of the document class in the document type declaration syntax. The word SYSTEM
indicates that the DTD is located in an external file. Following is an example of a document type declaration that uses both an internal and external DTD:

```
<!DOCTYPE talltales SYSTEM "TallTales.dtd">
<!ELEMENT question (#PCDATA)>
```

A URI (Uniform Resource Identifier) is a more general form of a URL (Uniform Resource Locator) that allows you to identify network resources other than files. URLs should be familiar to you as they are used to represent the addresses of web pages.

This code shows how you might create a document type declaration for the Tall Tales trivia sample document. The root element of the document is `talltales`, which means that all documents of this type must have their content housed within the `talltales` element. The document type declaration references an external DTD stored in the file `TallTales.dtd`. Additionally, an element named `question` is declared as part of the internal DTD. Remember that internal markup declarations always override external declarations of the same name if such declarations exist. It isn’t always necessary to use an internal DTD if the external DTD sufficiently describes a language, which is often the case.

The document type declaration must appear after the XML declaration but before the first element (tag) in a document.

In the previous hour you learned about the XML declaration, which must appear at the beginning of a document and indicates what version of XML is being used. The XML declaration can also contain additional pieces of information that relate to DTDs. I’m referring to the standalone status and character encoding of a document. The standalone status of a document determines whether or not a document relies on any external information sources, such as an external DTD. You can explicitly set the standalone status of a document using the standalone document declaration, which looks like an attribute of the XML declaration:

```
<?xml version="1.0" standalone="no"?>
```

You learn about the character encoding of a document in the next hour, “Digging Deeper into XML Documents.”

A value of yes for standalone indicates that a document is standalone and therefore doesn’t rely on external information sources. A value of no indicates that a document is not standalone and therefore may rely on external information sources. Documents
that rely on an external DTD for validation can't be considered standalone, and must have `standalone` set to `no`. For this reason, `no` is the default value for `standalone`.

**Pondering Elements and Attributes**

The primary components described in a DTD are elements and attributes. Elements and attributes are very important because they establish the logical structure of XML documents. You can think of an element as a logical unit of information, whereas an attribute is a characteristic of that information. This is an important distinction because there is often confusion over when to model information using an element versus using an attribute.

A useful approach to take when assessing the best way to model information is to consider the type of the information and how it will be used. Attributes provide tighter constraints on information, which can be very helpful. More specifically, attributes can be constrained against a predefined list of possible values and can also have default values. Element content is very unconstrained and is better suited for storing long strings of text and other child elements. Consider the following list of advantages that attributes offer over elements:

- Attributes can be constrained against a predefined list of enumerated values.
- Attributes can have default values.
- Attributes have data types, although admittedly somewhat limited.
- Attributes are very concise.

Attributes don't solve every problem, however. In fact, they are limited in several key respects. Following are the major disadvantages associated with attributes:

- Attributes can't store long strings of text.
- Attributes can't contain nested information.
- Whitespace can't be ignored in an attribute value.

Given that attributes are simpler and more concise than elements, it's reasonable that you should use attributes over child elements whenever possible. Fortunately, the decision to use child elements is made fairly straightforward by the limitations of attributes: if a piece of information is a long string of text, requires nested information within it, or requires whitespace to be ignored, you'll want to place it in an element. Otherwise, an attribute is probably your best choice. Of course, regardless of how well your document data maps to attributes, it must have at least a root element.
Digging Deeper into Elements

To declare an element in a DTD, you must use an element declaration, which takes the following form:

```xml
<!ELEMENT ElementName Type>
```

The name of the element determines the name of the tag(s) used to mark up the element in a document and corresponds to `ElementName` in the element declaration. This name must be unique within the context of a DTD. The type of the element is specified in `Type`; XML supports four different types of elements, which are determined by the content contained within the element:

- **Empty**—The element doesn’t contain any content (it can still contain attributes).
- **Element-only**—The element only contains child elements.
- **Mixed**—The element contains a combination of child elements and character data.
- **Any**—The element contains any content allowed by the DTD.

The name of an element must not contain the ampersand character (`&`) or begin with the sequence of letters `X`, `M`, and `L` in any case combination (`XML`, `xml`, `XML`, and so on).

The next few sections explore the different element types in more detail.

Peeking Inside Empty Elements

An empty element is an element that doesn’t contain any element content. An empty element can still contain information, but it must do so using attributes. Empty elements are declared using the following form:

```xml
<!ELEMENT ElementName EMPTY>
```

Following is an example of declaring an empty element using this form:

```xml
<!ELEMENT clothing EMPTY>
```

After an empty element is defined in a DTD, you can use it in a document in one of two ways:

- With a start tag/end tag pair
- With an empty tag
Following is an example of an empty element defined using a start tag/end tag pair:

```xml
<clothing></clothing>
```

Notice that no content appears between the tags; if any content did appear, even a single space, the document would be invalid. A more concise approach to defining empty elements is to use an empty tag. Following is an example of using an empty tag to define an empty element:

```xml
<clothing />
```

As you can see, an empty tag is somewhat of a combination of a start tag and end tag. In addition to being more concise, empty tags help to make it clear in a document that an element is empty and therefore can’t contain content. Remember that empty elements can still contain information in attributes. Following is an example of how you might use a few attributes with an empty element:

```xml
<clothing type="t-shirt" color="navy" size="xl" />
```

**Housing Children with Element-Only Elements**

An element-only element contains nothing but child elements. In other words, no text content is stored within an element-only element. An element is declared element-only by simply listing the child elements that can appear within the element, which is known as the element’s content model. Following is the form expected for declaring an element’s content model:

```xml
<!ELEMENT ElementName ContentModel>
```

The content model is specified using a combination of special element declaration symbols and child element names. The symbols describe the relationship between child elements and the container element. Within the content model, child elements are grouped into sequences or choice groups using parentheses (`()`). A sequence of child elements indicates the order of the elements, whereas a choice group indicates alternate possibilities for how the elements are used. Child elements within a sequence are separated by commas (`,`), whereas elements in a choice group are separated by pipes (`|`). Following are the different element declaration symbols used to establish the content model of elements:

- Parentheses (`()`)—Encloses a sequence or choice group of child elements
- Comma (`,`)—Separates the items in a sequence, which establishes the order in which they must appear
- Pipe (`|`)—Separates the items in a choice group of alternatives
Digging Deeper into Elements

No symbol—Indicates that a child element must appear exactly once

Question mark (?)—Indicates that a child element must appear exactly once or not at all

Plus sign (+)—Indicates that a child element must appear at least once

Asterisk (*)—Indicates that a child element can appear any number of times

Although I could use several paragraphs to try and explain these element declaration symbols, I think an example is much more explanatory. Following is the declaration for an element named resume that might be used in a resume markup language:

```xml
<!ELEMENT resume (intro, (education | experience)+, hobbies?, references*)>
```

The `resume` element is pretty interesting because it demonstrates the usage of every element declaration symbol. The `resume` element is element-only, which is evident by the fact that it contains only child elements. The first child element is `intro`, which must appear exactly once within the `resume` element; this is because no symbols are used with it. The `education` or `experience` elements must then appear at least once, which is indicated by the plus sign just outside of the parentheses. Within the parentheses, the `education` element must appear exactly once, whereas the `experience` element must appear at least once but can also appear multiple times. The idea is to allow you to list part of your education followed by any relevant work experience; you may have worked multiple jobs following a single block of education. The `hobby` element can appear exactly once or not at all; all of your hobbies must be listed within a single `hobby` element. Finally, the `references` element can appear any number of times.

To get a practical feel for how element-only elements affect XML documents in the real world, take a look at the following line of code from the DTD for the RSS language:

```xml
<!ELEMENT item (title | link | description)*>
```

RSS is an XML-based language that is used to syndicate web sites so that applications and other web sites can obtain brief descriptions of new articles and other content. For example, Sports Illustrated (http://sportsillustrated.cnn.com/) offers RSS news feeds for all major sports. Other web sites can tap into these feeds or you can use an RSS aggregator application such as FeedDemon (http://www.feeddemon.com/), which allows you to view news feeds much like you view email in an email client.

Anyway, getting back to the RSS DTD, the previous line of code reveals that the `item` element can contain any combination of `title`, `link`, or `description` elements as long as there is no more than one of each. To see how this DTD
translates into real XML code, check out the following excerpt from a real Sports Illustrated NFL news feed:

```xml
<item>
  <title>Titans’ trio of young WRs showing promise</title>
  <description>Read full story for latest details.</description>
</item>
```

To view the XML code for a Sports Illustrated news feed for yourself, visit the main Sports Illustrated page, scroll down to the bottom, and click the small XML logo. A list of news feeds appears—click any of them to open the RSS document containing the feed data.

As this code reveals, the `<item>` element contains exactly one each of `<title>`, `<link>`, and `<description>` elements. To see how this code affects a real web page, check out the syndicated NFL news feed on my web site in Figure 3.1.

The figure reveals how the Tennessee Titans news story shown in the previous RSS code is syndicated and visible as an RSS news feed on my web site.
news feeds and create your own in Hour 24, “Syndicating the Web with RSS News Feeds.”

**Combining Content and Children with Mixed Elements**

*Mixed elements* are elements that contain both character data (text) and child elements. The simplest mixed element is also known as a *text-only element* because it contains only character data. Text-only elements are declared using the following form:

```xml
<!ELEMENT ElementName (#PCDATA)>
```

The content model for text-only elements consists solely of the symbol `#PCDATA` contained within parentheses, which indicates that an element contains Parsed Character DATA. Following is an example of a simple text-only element declaration:

```xml
<!ELEMENT hobbies (#PCDATA)>
```

By the Way

The word “parsed” in Parsed Character DATA (PCDATA) refers to the fact that PCDATA within a document is processed (parsed) by an XML application. Most of the text content in an XML document is considered PCDATA, including character entities for example. The parsing process involves stripping out extraneous whitespace and, in the case of character entities, replacing entities with the appropriate text. The alternative to PCDATA is CDATA (Character DATA), which is text that isn’t processed by an XML application. Later in the book you learn when it is useful to include CDATA in a document.

This element might be used to mark up a list of hobbies in an XML document that helps to describe you to other people. Following is an example of how you might define the `hobbies` element in a document:

```xml
<hobbies>juggling, unicycling, tight-rope walking</hobbies>
```

Speaking of examples, the following code shows the `title`, `link`, and `description` elements from the RSS DTD that you learned about in the previous section:

```xml
<!ELEMENT title (#PCDATA)>
<!ELEMENT link (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

As you can see, these elements are all defined as text-only elements, which makes sense given the sample RSS data you saw in the previous section.

In reality, a text-only element is just a mixed element that doesn’t contain any child elements. Mixed elements that contain both text data and child elements are declared very much like element-only elements, with the addition of a few subtle
requirements. More specifically, the content model for a mixed element must contain
a repeating choice list of the following form:

```xml
<!ELEMENT ElementName (#PCDATA | ElementList)*>  
```

If this looks a little confusing at first, don’t worry. Let’s break it down. The
symbol `#PCDATA` at the start of the choice list indicates that the mixed element
can contain character data. The remainder of the choice list contains child
elements, and resembles a regular element-only content model. Additional
`#PCDATA` symbols may be interspersed throughout the content model to indicate
that character data appears throughout the child elements. An asterisk (*) must
appear at the end of the content model’s choice list to indicate that the entire
choice group is optional—this is a requirement of mixed elements. Also,
although a mixed element declaration constrains the type of child elements
allowed, it doesn’t constrain the order of the elements or the number of times
they may appear.

**By the Way**

In the content model for a mixed element, the character data (#PCDATA) must
always be specified first in the choice group, and the choice group itself must be
declared as repeating by following it with an asterisk (*).

Although mixed elements provide a considerable amount of flexibility, they lack the
structure of element-only elements. So, with the exception of text-only elements,
you’re better off using element-only elements instead of mixed elements whenever
possible; you can often get the equivalent of a mixed element by declaring attributes
on a text-only element.

**Keeping Your Options Open with Any Elements**

The *any* element is the most flexible element of all because it has virtually no
structure. The any element is declared using the symbol `ANY`, and earns its name
by being able to contain any declared element types, character data, or a mixture
of both. You can think of the any element as a mixed element with a wide-open
content model. Following is the form used to declare an any element:

```xml
<!ELEMENT ElementName ANY>
```

Not surprisingly, the lack of structure associated with the any element makes it
something you should avoid at all costs in a production DTD. I mention a produc-
tion (completed) DTD because the any element is typically used only during the
development of a DTD for testing purposes.
Putting Attributes to Work

Attributes go hand in hand with elements and are incredibly important to the
construction of DTDs. Attributes are used to specify additional information about
an element. More specifically, attributes are used to form a name/value pair that
somehow describes a particular property of an element. Attributes are declared in
a DTD using attribute list declarations, which take the following form:

```xml
<!ATTLIST ElementName AttrName AttrType Default>
```

This form reveals that an attribute has a name (AttrName) and a type (AttrType),
as well as a default value (Default). The default value for an attribute refers to
either a value or a symbol that indicates the use of the attribute. There are four dif-
ferent types of default values you can specify for an attribute in Default:

- `#REQUIRED`—The attribute is required.
- `#IMPLIED`—The attribute is optional.
- `#FIXED` value—The attribute has a fixed value.
- `default`—The default value of the attribute.

The `#REQUIRED` value identifies a required attribute, which means the attribute must be
set if you use the element. The `#IMPLIED` value identifies an optional attribute, which
means the attribute is optional when using the element. The `#FIXED` attribute is used
to assign a fixed value to an attribute, effectively making the attribute a constant
piece of information; you must provide the fixed attribute value after the `#FIXED`
symbol when declaring the attribute. The last option for declaring attribute defaults is
to simply list the default value for an attribute; an attribute will assume its default
value if it isn't explicitly set in the element. Following is an example of an attribute
list for an element that specifies the units for a duration of time:

```xml
<!ELEMENT distance (#PCDATA)>
<!ATTLIST distance units (miles | kilometers | laps) "miles">
```

In this example, the element is named `distance` and its only attribute is named `units`.
The `units` attribute can only be set to one of three possible values: miles, kilometers,
or laps. The default value of the `units` attribute is `miles`, which means that if you
don't explicitly set the attribute it will automatically take on a value of `miles`.

Although attribute lists don’t have to be declared in a particular place within a
DTD, it is common practice to place them immediately below the declaration for
the element to which they belong.
In addition to the default value of an attribute value, you must also specify the type of the attribute in the attribute list declaration. There are 10 different attribute types, which follow:

- **CDATA**—Unparsed character data
- **Enumerated**—A series of string values
- **NOTATION**—A notation declared somewhere else in the DTD
- **ENTITY**—An external binary entity
- **ENTITIES**—Multiple external binary entities separated by whitespace
- **ID**—A unique identifier
- **IDREF**—A reference to an ID declared somewhere else in the DTD
- **IDREFS**—Multiple references to IDs declared somewhere else in the document
- **NMTOKEN**—A name consisting of XML token characters (letters, numbers, periods, dashes, colons, and underscores)
- **NMTOKENS**—Multiple names consisting of XML token characters

To help in understanding these attribute types, it's possible to classify them into three groups: string, enumerated, and tokenized.

**String attributes** are the most commonly used attributes and are represented by the CDATA type. The CDATA type indicates that an attribute contains a simple string of text. Following is an example of declaring a simple CDATA attribute that must be defined in the education element:

```xml
<!ATTLIST education school CDATA #REQUIRED>
```

In this example, the school a person attended is a required character data attribute of the education element. If you wanted to make the school attribute optional, you could use the #IMPLIED symbol:

```xml
<!ATTLIST education school CDATA #IMPLIED>
```

**Enumerated attributes** are constrained to a list of predefined strings of text. The enumerated type is similar to the CDATA type except the acceptable attribute values must come from a list that is provided in the attribute list declaration. Following is an example of how you might provide an enumerated attribute for specifying the type of degree earned as part of the education element:

```xml
<!ATTLIST education degree (associate | bachelors | masters | doctorate) "bachelors">
```
When using the degree attribute in a document, you are required to select a value from the enumerated list. If you don’t use the attribute at all, it will assume the default value of bachelors.

Tokenized attributes are processed as tokens by an XML application, which means the application converts all contiguous whitespace to a single space character and eliminates all leading and trailing whitespace. In addition to eliminating the majority of whitespace in a tokenized attribute value, the XML application also validates the value of a tokenized attribute based upon the declared attribute type: ENTITY, ENTITIES, ID, IDREF, IDREFS, NMTOKEN, or NMTOKENS.

A token is the smallest piece of information capable of being processed by an XML application. A tokenized attribute is simply an attribute that is processed into tokens by an XML application, which has the effect of eliminating extraneous whitespace (space characters, newline characters, and so on). Contrast this with a string attribute, which goes unprocessed, and therefore retains all of its whitespace.

The ENTITY and ENTITIES types are used to reference entities, which you learn about in the next hour. As an example, images are typically referenced as binary entities, in which case you use an ENTITY attribute value to associate an image with an element type:

```xml
<!ATTLIST photo image ENTITY #IMPLIED>
```

The ENTITIES type is similar to ENTITY but it allows you to specify multiple entities. The ID, IDREF, and IDREFS attribute types all relate to unique identifiers. The ID type is a unique identifier that can be used to uniquely identify an element within a document:

```xml
<!ATTLIST part id ID #REQUIRED>
```

Only one attribute of type ID may be assigned to a given element type. The NMTOKEN and NMTOKENS attribute types are used to specify attributes containing name token values. A name token value consists of a single name, which means that it can’t contain whitespace. More specifically, a name token value can consist of alphanumeric characters in addition to the following characters: ., -, _, and :

**Working with Multiple Attributes**

I’ve only shown you example of individual attributes thus far, but you’ll likely create elements that rely on several attributes. You can list all of the attributes for an element in a single attribute list by listing the attributes one after the next within the
attribute list declaration. Following is an example of declaring multiple attributes
within a single attribute list:

```xml
<!ELEMENT photo (image, format)>
<!ATTLIST photo
  image ENTITY #IMPLIED
  photo format NOTATION (gif | jpeg) #REQUIRED
>
```

This example shows how the two attributes of the photo element, image and photo,
are declared in a single attribute list declaration.

## A Complete DTD Example

Admittedly, this hour has thrown a great deal of information at you, most of which
is quite technical. But there’s a method to the madness, and now it’s time to see some
of the payoff. To help you get some perspective on how elements and attributes fit
into a DTD for a new custom markup language, let’s work through the design of
a DTD for a sports training markup language. This markup language, which we’ll
call ETML (Endurance Training Markup Language), might come in handy if you
even decide to compete in a marathon or triathlon—it models training data related to
endurance sports such as running, swimming, and cycling. The following are the
major pieces of information that are associated with each individual training session:

- Date—The date and time of the training session
- Type—The type of training session (running, swimming, cycling, and so on)
- Heart rate—The average heart rate sustained during the training session
- Duration—The duration of the training session
- Distance—The distance covered in the training session (measured in miles or
  kilometers)
- Location—The location of the training session
- Comments—General comments about the training session

Knowing that all of this information must be accounted for within a training session
element, can you determine which ones would be better suited as child elements and
which would be better suited as attributes? There really is no correct answer but there
are a few logical reasons you might separate some of the information into elements
and some into attributes. The following is how I would organize this information:

- Attributes—Date, Type, Heart rate
- Child elements—Duration, Distance, Location, Comments
The date, type, and heart rate for a training session are particularly well suited for attributes because they all involve short, simple values. The type attribute goes a step further because you can use an enumerated list of predefined values (running, cycling, and so on). The duration and distance of a session could really go either way in terms of being modeled by an element or an attribute. However, by modeling them as elements you leave room for each of them to have attributes that allow you to specify additional information such as the exact units of measure. The location and comments potentially contain descriptive text, and therefore are also better suited as child elements.

A golden rule of XML design is that the more constraints you can impose on a document, the more structured its content will be. In other words, try to create schemas that leave little to chance in terms of how elements and attributes are intended to be used.

By the Way

With the conceptual design of the DTD in place, you’re ready to dive into the code. Listing 3.3 contains the code for the ETML DTD, which is stored in the file etml.dtd.

**LISTING 3.3** The etml.dtd DTD That Is Used to Validate ETML Documents

```
1: <!ELEMENT trainlog (session)>  
2:  
3: <!ELEMENT session (duration, distance, location, comments)>  
4: <!ATTLIST session  
5:   date CDATA #IMPLIED  
6:   type (running | swimming | cycling) "running"  
7:   heartrate CDATA #IMPLIED  
8: >  
9:  
10: <!ELEMENT duration (#PCDATA)>  
11: <!ATTLIST duration  
12:   units (seconds | minutes | hours) "minutes"  
13: >  
14:  
15: <!ELEMENT distance (#PCDATA)>  
16: <!ATTLIST distance  
17:   units (miles | kilometers | laps) "miles"  
18: >  
19:  
20: <!ELEMENT location (#PCDATA)>  
21:  
22: <!ELEMENT comments (#PCDATA)>  
```

You should be able to apply what you’ve learned throughout this hour to understanding the ETML DTD. All of the elements and attributes in the DTD flow from the conceptual design that you just completed. The trainlog element (line 1) is the root element for ETML documents and contains session elements for each training session.
session. Each session element consists of duration, distance, location, and comments child elements (line 3) and date, type, and heartrate attributes (lines 4–7). Notice that the type attribute of the session element (line 6) and the units attributes of the duration and distance elements (lines 12 and 17) are constrained to lists of enumerated values.

Of course, no DTD is really complete without an XML document to demonstrate its usefulness. Listing 3.4 shows a sample document that is coded in ETML.

LISTING 3.4 The Training Log Sample ETML Document

```
<?xml version="1.0"?>
<!DOCTYPE trainlog SYSTEM "etml.dtd">
<trainlog>
  <session date="11/19/05" type="running" heartrate="158">
    <duration units="minutes">50</duration>
    <distance units="miles">5.5</distance>
    <location>Warner Park</location>
    <comments>Mid-morning run, a little winded throughout.</comments>
  </session>
  <session date="11/21/05" type="cycling" heartrate="153">
    <duration units="hours">1.5</duration>
    <distance units="miles">26.4</distance>
    <location>Natchez Trace Parkway</location>
    <comments>Hilly ride, felt strong as an ox.</comments>
  </session>
  <session date="11/24/05" type="running" heartrate="156">
    <duration units="hours">2.5</duration>
    <distance units="miles">16.8</distance>
    <location>Warner Park</location>
    <comments>Afternoon run, felt reasonably strong.</comments>
  </session>
</trainlog>
```

As you can see, this document strictly adheres to the ETML DTD both in terms of the elements it defines as well as the nesting of the elements. The DTD is specified in the document type declaration, which clearly references the file etml.dtd (line 2). Another couple of aspects of the document to pay attention to are the type and units attributes (lines 5, 12, and 19), which adhere to the lists of available choices defined in the DTD. Keep in mind that even though only three training sessions are included in the document, the DTD allows you to include as many as you want. So if you’re feeling energetic, go sign up for a marathon and start logging away training sessions in your new markup language!

By the Way

Hour 8 shows you how to validate an XML document against a DTD.
Summary

XML is a markup language that allows you to create other markup languages. The process of establishing the structure of XML-based markup languages is known as data modeling. The resulting data model for a custom markup language is defined in a construct known as a schema. Two primary approaches exist for creating schemas for XML languages: DTDs and XSDs. There are advantages and disadvantages to each approach, but they are both suitable for most XML applications.

In this hour you gained some important insight into document validation and found out what it means for a document to be well formed. You also explored the details of the DTD approach to defining XML schemas, and found out how DTDs are responsible for describing the structure and format of a class of XML documents. You saw that a DTD consists of markup declarations that determine the rules for a custom markup language. In addition to providing a formal set of rules for a markup language, DTDs form a critical part of XML in that they provide a means of validating documents for accuracy. The hour culminated with the creation of a complete DTD for the Endurance Training Markup Language (ETML). You also saw a sample XML document created in ETML.

Q&A

Q. I don’t plan on creating any of my own custom markup languages. Do I still need to know the details of DTDs and XSDs?

A. Yes. In many cases, markup languages aren’t documented too well, which means the DTD or XSD for a language may be all you have to go by when it comes to figuring out the format of documents.

Q. Would I ever want to create both a DTD and an XSD for a custom markup language?

A. Probably not. The only reason you might do this is if you create a DTD initially for convenience, and later replace it with an XSD as your needs change and as you migrate to tools that demand support for XML Schema (XSDs).

Q. Why would you ever want to use an internal DTD?

A. Because an internal DTD is declared directly in an XML document and not in an external file, it applies only to that particular document. Therefore, you should only use an internal DTD for markup declarations that apply to a specific document. Otherwise, all markup declarations should be placed in an external DTD so that they can be reused with other documents.
Q. *Why would you use an attribute over an element when designing a DTD?*

A. Attributes provide tighter constraints on data, can be constrained against a predefined list of possible values, and can have default values. Element content is much less constrained, and is better suited for housing long strings of text and other child elements. A golden rule of XML design is that the more constraints you can impose on a document, the more structured its content will be. Knowing this, you should attempt to fit data into attributes whenever possible.

**Workshop**

The Workshop is designed to help you anticipate possible questions, review what you’ve learned, and begin learning how to put your knowledge into practice.

**Quiz**

1. What is the purpose of a schema?

2. If you wanted to create a schema for a markup language with specific data types, such as integer numbers and dates, which schema technology would you need to use?

3. What is the difference between valid and well-formed documents?

**Quiz Answers**

1. A schema describes the exact elements and attributes that are available within a given markup language, along with which attributes are associated with which elements and the relationships between the elements.

2. XML Schema is the schema technology of choice for schemas that require specific data types such as integer numbers and dates.

3. Well-formed documents must adhere to the general XML rules, whereas valid documents must also adhere to a schema.
Exercises

1. Modify the ETML DTD so that it includes a new attribute of the session element that stores a rating for the training session—the rating indicates how well you felt during the session on a scale of 1 to 10. Design the new rating attribute so that it is constrained to a list of numeric values between 1 and 10.

2. Modify the trainlog.xml document so that it takes advantage of the new rating attribute.