Chapter Title

Objectives

- Discuss special element types.
- Learn how to declare attributes in a DTD.
- Learn how to reference external entities.
- Provide tips for good content modeling.
- Explain well-formed documents.
- Explain valid documents.
- Create and use an external DTD.

Special Element Types

There are three more element types that require some explanation. These are special element types that if used, must be declared within the document type declaration.

Empty Elements

Empty elements have no content. If you will recall, there are two ways to represent an empty element.

```
<empty.element></empty.element>
```

or

```
<empty.element/>
```

If you intend to model your content using one or more empty elements, you must declare them as such in the DTD. This is done using the *EMPTY* keyword. The syntax is for defining empty elements is:

```
<!ELEMENT empty.element EMPTY>
```

Tip: You will typically use empty elements only when you are going to associate attributes with that element. We are going to discuss attribute declarations shortly.

Unrestricted Elements

Unrestricted elements can contain any elements that are declared elsewhere in the DTD (either internal or external). You cannot stipulate the order of any elements contained within an unrestricted element.

In the following example, we show a declaration of an empty element (<empty.element>), and an unrestricted element (<unrestricted.element>).

```
<?xml version="1.0"?>
<!DOCTYPE root.element [
   <!ELEMENT empty.element EMPTY>
   <!ELEMENT unrestricted.element ANY>
]>
<root.element>

   <empty.element/>

   <unrestricted.element>
      <empty.element/>
```

```
        <empty.element/>
        <empty.element/>
    </unrestricted.element>

</root.element>
```

File: Chapter 4\SpecialElementTypes.xml

Mixed Content Elements

Mixed content elements are elements that can contain text and/or other elements. This can get complicated in theory, but declaring them is not. For example, let's declare an element called <mixed.content> that can either contain data (PCDATA), or one of three elements called <one>, <two> and <three> respectively. The syntax would be as follows:

```
<!ELEMENT mixed.content (#PCDATA | one | two | three)*>
```

There are a couple points to make note of when declaring mixed content elements. First, the content model has to take the form of a single set of alternatives starting with #PCDATA followed by the element types that can occur. Each element can be declared only once. Second, the asterisk (*) after the closing parenthesis and before the closing > is not a typo. You have to include the asterisk when declaring mixed content elements.

A Few Final Words on Elements

You now have the knowledge to create sophisticated element content models that have structure and integrity. Believe it or not, we have just scratched the surface on the topic of DTD's. Now that you know how to declare elements, let's look at declaring attributes within those elements.

Attribute Declarations

Attributes can be likened to properties of an element. Attributes are specific to an element and must be defined within the start tag of the element that they belong to. We have seen attributes within our employee XML document in the <employee> element.

```
<employee id="A1234">
```

In this case, "id" is an attribute. Like elements, attributes must be declared in the DTD or an error will occur. An attribute declaration uses the <!ATTLIST> tag and has the following syntax:

```
<!ATTLIST element.name attribute.definitions>
```

It is a general practice to keep attribute declarations close the element declarations that they belong to. Also, if multiple attributes are being declared for a single entity, use multiple lines in declaring them. This makes the DTD much more readable and easy to understand. Do not dwell on this code as we are going to explain the syntax in more detail in a moment. Instead, focus on the <!ATTLIST> tag and its association with the <employee> element.

```
. . . (omitted)

<!ELEMENT employee (name, position?)>
   <!ATTLIST employee
            empid CDATA #REQUIRED
            sex   CDATA #REQUIRED>

. . . (omitted)

   <employee empid="123" sex="M">
      <name>John Doe</name>
   </employee>

. . . (omitted)
```

An attribute declaration accomplishes the following:

1. It declares the attribute names and element association.

2. It specifies the attribute type.

3. It can be used to specify a default value for the attribute.

Each attribute declaration consists of a name and type pair.

Attribute Types

There are three types of attributes that you can create. These consist of a string type (similar to #PCDATA elements), a tokenized element, and an enumerated attribute.

String Type

Like the name suggests, this attribute type is for storing string values. Any attribute that is not specifically declared is assumed to be a string type attribute. The syntax for creating a string type attribute is:

```
<!ATTLIST element.name attribute.name CDATA>
```

Notice that the CDATA (character data) string is used when declaring string type attributes while #PCDATA (parseable character data) is used when declaring string type elements. The following example declares an attribute named “empid” that is a string type and is associated with the <employee> element.

```
<!ATTLIST employee empid CDATA>
```

When an <employee> element is created, the “empid” attribute will contain a string value.

```
<employee empid=”A1234”>
```

Tokenized Type

Tokenized attributes are attributes whose value consists of one or more *tokens* that have meaning in XML. There are seven tokens that you can assign to an attribute.

ID

An ID attribute serves as an identifier for an element. This is very similar to the ID attribute in HTML. You may have seen or done something like the following in an HTML document:

```
<DIV ID=”DIV1”>Some text</DIV>
```

The ID attribute in HTML allows you to uniquely identify an HTML element. The same is true in XML, only it must be declared (as all things in XML must). The syntax for declaring an XML attribute as an ID attribute is:

```
<!ATTLIST element.name attribute.name ID #REQUIRED>
```

An ID attribute is very similar to a primary key in a table. While a primary key uniquely identifies a row in a table, an ID attribute uniquely identifies as element. Thus, an ID attribute value must be unique to the XML document. An ID value must also conform to XML naming rules, which we discussed previously.

Tip: You can name an ID attribute any name that you like, but it is standard practice to name the attibute “id” so that it is easy to visually identify.

IDREF

An IDREF attribute is a pointer to an ID attribute value that is declared somewhere else in the document. In relational database terms, this is similar to a foreign key reference. They key is that the IDREF attribute value must match the value of an ID attribute somewhere else in the document. An example is as follows:

```
. . . (omitted)

<!ELEMENT element1 EMPTY>
   <!ATTLIST element1 id ID #REQUIRED>

<!ELEMENT element2 EMPTY>
   <!ATTLIST element2 el1.id IDREF #REQUIRED>

. . . (omitted)

<element1 id=”A1234”/>

<element2 el1.id=”A1234”/>

. . . (omitted)
```

File: Chapter 4\IDAttributes.xml

An IDREF attribute is used when an element needs to refer to a specific instance of another element type. In the above example, <element2> has a direct reference to an instance of an <element1> element by using an IDREF attribute.

Returning to our employee document, let’s add an “id” attribute to the <employee> element so that we can uniquely identify each employee by their unique employee ID.

```
<!ELEMENT employee (name, position, address, phone)>
<!ATTLIST employee id ID #REQUIRED>
```

File: Chapter 4\EmployeesWithDTD.xml

IDREFS

An IDREFS attribute has the same purpose as an IDREF attribute except that it can hold multiple ID values. Each ID value is separated by a space. Extending our previous example, we could declare a new element type (<element3>) that references multiple (one or more) <element1> instances.

```
. . . (omitted)
```

```
<!ELEMENT element1 EMPTY>
   <!ATTLIST element1 id ID #REQUIRED>

<!ELEMENT element3 EMPTY>
   <!ATTLIST element3 el1.ids IDREF #REQUIRED>

. . . (omitted)

<element1 id="A1234"/>
<element1 id="B1234"/>
<element1 id="C1234"/>

. . . (omitted)

<element3 el1.ids="A1234 B1234 C1234"/>

. . . (omitted)
```

File: Chapter 4\IDAttributes.xml

In the above example, <element3> references not one <element1> element, but three.

The IDREFS attribute type provides a one-to-many type of capability. A single element can reference multiple instances of other elements. You may relate this capability to the relational models primary/foreign key concept. This is partly correct, but the XML implementation is very different.

In our example, the referenced elements were of the same type. This is not a requirement in XML. A single IDREFS attribute can not only reference multiple elements, but multiple types of elements as well. This distinction makes it very different from the relational model in that primary/foreign key relationships relate data between only two tables (element types). IDREFS do not have this restriction. This provides more flexibility, but does not provide data integrity as does the relational model.

Warning: Do not make the mistake of thinking that IDREFS are the same as primary/foreign key relationships in the relational model. There is no data integrity associated with IDREFS.

ENTITY

ENTITY attributes are pointers to external entities. So far we have been working with *standalone documents*. Standalone documents have no references to external entities (they are complete and self-contained). The value of an ENTITY attribute must consist of characters that represent an external entity (storage object). The syntax for declaring an ENTITY attribute is:

```
<!ATTLIST element.name ENTITY>
```

We are going to go into detail on referencing external entities later.

ENTITIES

ENTITIES attributes are the same as ENTITY attributes, except they reference multiple external entities. Like IDREFS, the values must be separated by a space. The syntax for declaring an ENTITIES attribute is:

```
<!ATTLIST element.name ENTITIES>
```

NMTOKEN

NMTOKEN attributes have values that are token strings. These token strings can consist of any mixture of name characters. The syntax for NMTOKEN attributes is:

```
<!ATTLIST element.name NMTOKEN>
```

NMTOKENS

NMTOKEN attributes are the same as NMTOKEN attributes, except that they can contain multiple values provided that they are separated by spaces. Tokens are similar to string typess, but the are processed differently. NMTOKENS attribute values are treated each independently by an XML processor and can be queried independently be an application.

```
<?xml version="1.0"?>
<!DOCTYPE tokens [

   <!ELEMENT tokens ( element1+)>

   <!ELEMENT element1 ( #PCDATA )>
      <!ATTLIST element1 attr1 NMTOKEN #IMPLIED>
      <!ATTLIST element1 attr2 NMTOKENS #IMPLIED>
]>
<tokens>

   <element1 attr1="Token1"
             attr2="123_Token1 Token2 Token3">
      Some text here...
   </element1>

</tokens>
```

File: Chapter 4\TokenAttributes.xml

Enumerated Type

Enumerated attributes have values that are part of a list of possible values. These values must be valid name tokens. The syntax is:

```
<!ATTLIST element.name attribute.name
      ( option1 | option 2 | option n )
```

```
        "default value">
```

Let's say that you want to declare an element called <primary> with an attribute called "color". The only valid primary color values are "RED", "GREEN" and "BLUE", and we want to force our DTD to constrain our possible values to these. We also want the default value to be "RED".

Here is the resulting XML document.

```
<?xml version="1.0"?>
<!DOCTYPE colors [

   <!ELEMENT colors ( primary+ )>

   <!ELEMENT primary EMPTY>
      <!ATTLIST primary color ( RED | GREEN | BLUE )
"RED">
]>
<colors>

   <primary color="RED"/>
   <primary color="GREEN"/>
   <primary color="BLUE"/>

   <!-- ERROR!
   <primary color="YELLOW"/>
   -->

</colors>
```

File: Chapter 4\EnumerationAttributes.xml

There is no such thing as an optional attribute in XML. However, using enumeration attributes, you can emulate an optional value. This is accomplished by defining the default value in the <!ATTLIST> declaration as an empty string.

```
<!ATTLIST elem attr (X | Y | Z) "">
```

If you leave the attribute out, the default value, which is an empty string, will be assigned to the attribute. In other words, you do not have to assign it in the element.

Attribute Attributes

There are three keywords in XML that are used to instruct the processor how to treat an attribute. You have already seen us use two of these in our previous examples. These keywords always begin with a # character are used at the end of the attribute declaration.

#REQUIRED

This keyword means that the attribute is required. If it is omitted, the XML processor will generate a fatal error. When we discussed the ID and IDREFS token types, we used the keyword in the example. This is because ID attributes are required if the document is to be validated, which we will discuss in the next chapter.

```
<!ATTLIST element1 id ID #REQUIRED>
```

#IMPLIED

This keyword instructs the XML processor to the application that no value was specified if the attribute was not assigned. We first saw implied attributes in the NMTOKENS example.

```
<!ELEMENT element1 ( #PCDATA )>
   <!ATTLIST element1 attr1 NMTOKEN #IMPLIED>
   <!ATTLIST element1 attr2 NMTOKENS #IMPLIED>
```

If either the "attr1" and "attr2" attributes are left out, the processor reports it the application and assumes that it knows how to deal with it.

#FIXED

This keyword is used to create a text constant. The syntax is:

```
<!ATTLIST element.name attribute.name
          attribute.type #FIXED "default value">
```

Examine the following statement:

```
<!ATTLIST form method CDATA #FIXED "POST">
```

The <form> element now has an attribute called "method". This attribute can only have one value: POST. The result is that the following statement is valid, and any deviation from it will result in the XML processor generating a fatal error.

```
<form method="POST">
```

This is actually a clever example, because if you know HTML, you know that the <form> tag has a "method" attribute, and it must be set to "POST" to function correctly. This is a good example of using one markup language to define another markup language (i.e. XML to define HTML).

The Completed Employee DTD

Now that we have covered many aspects of DTD's, let's go back and review the finished employee XML document. With a complete DTD in place, we can have a content model in place that will provide a structure for all employee information that we may want to add to our document.

```
<?xml version="1.0"?>
<!DOCTYPE employees [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee (name, position, address, phone)>
   <!ATTLIST employee id ID #REQUIRED>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone (main, office*, fax*, mobile*,
home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>
]>

<employees>
    <employee id="A1234">
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
            <main>(714) 555-1000</main>
            <fax>(714) 555-1001</fax>
        </phone>
    </employee>
    <employee id="A2345">
        <name>
            <first>Jane</first>
            <last>Doe</last>
        </name>
        <position>Systems Analyst</position>
        <address>
        </address>
        <phone>
            <main>(714) 555-2000</main>
            <mobile>(949) 555-2200</mobile>
            <home>(949) 555-2220</home>
        </phone>
    </employee>
</employees>
```

File Chapter 4\EmployeesWithDTD.xml

Elements or Attributes?

The question arises: "When should I use an element and when should I use an attribute?" This is going to be a topic that will be debated for a long time. There are really no clear guidelines on this one. Many have tried to throw some basic ones together, but for the most part, they really don't solve the issue. Here are some guidelines that have been used by many. The first three are really the most pragmatic and useful, but the others can shed some light when looking at XML documents authored by others.

- Use whatever is more readable.
- When in doubt, use an element. Elements typically make more sense to others (are more readable) when you give them an XML document.
- Use elements to represent objects, attributes to represent properties of that object.
- Use elements to represent physical aspects. Use attributes to represent intangible, abstract aspects.
- If you are using an XML editor, it may be easier to use elements since some editors have trouble with attributes.

The real answer to the question is: "it depends". It depends on you, the author. It is up to you to create content models that accurately represent the structure of the data that you are trying to capture. It is also up to you to create entities (XML documents) that are readable and maintainable.

There are features in XML that require you to use one over the other. This has not been a discussion about that. What we are discussing now is a preference issue.

Tip: Since there is no "magic bullet" to solving the problem, the best recommendation is to use good common sense and create entities that make sense.

External Entities

External entities are entities (storage objects) that are not part of your document. When we originally talked about XML structure, there were two facets:

- Logical – the logical order of elements in a document.
- Physical – the physical location of content.

These two can be very different. Whenever an XML processor processes a document, it is working with the logical structure as one entity. However, that one logical entity may be physically broken into multiple physical entities (files/documents).

This is possible because you can declare external entities within your XML document and include them so that they are treated like one document. In effect, they become one logical entity, even though they are multiple entities.

The syntax for declaring an external entity is:

```
<!ENTITY entity.name SYSTEM|PUBLIC "URI">
```

Let's break down this syntax. The first item is the <!ENTITY> declaration tag. The next is entity name that you define. The next item is either the SYSTEM or PUBLIC keywords, which we will describe in a moment. Finally, the Uniform Resource Identifier (URI) that points to the external entity is specified. A URI is just like a Uniform Resource Locator (URL), but has more capability. URI's can be fully qualified or relative. The typical structure of a URI is:

```
protocol://login-name:password@host:port/path
```

The protocol is optional and can specify a number of network protocols including Internet protocols. The login and password are optional, and if not specified, anonymous access is attempted. Finally, the path the absolute or relative path to the external entity. For example, if we wanted to include a document from a web site, the following would be valid:

```
<!ENTITY doc1 SYSTEM "http://www.mysrver.com/doc1.xml">
```

Identifiers

Before we show a completed example, let's define the SYSTEM and PUBLIC keywords, or *identifiers*, and what they mean.

SYSTEM

A system identifier is a *Universal Resource Identifier* (URI) that is used to retrieve an external entity (physical storage object). A URI is similar to a

URL, except that URL's are intended for network (Internet) use. For all intents and purposes, the two terms are interchangeable.

A identifier can reference an absolute position or a relative position as is exemplified in the following code fragments:

```
<!-- Absolute reference -->
<!DOCTYPE doc.name SYSTEM
"http://www.server.com/external.dtd">

<!-- Relative reference -->
<!DOCTYPE doc.name SYSTEM "../DTDs/external.dtd">
```

Most of the time you will use SYSTEM identifiers for your external DTD's.

PUBLIC

A public identifier is the officially recorded identifier for an external entity. Officially registered public identifiers are registered by the creator of the DTD with The International Standards Organization (ISO). Public identifier names are assigned by the American Graphic Communication Association (CGA) under the direction of the ISO.

The syntax for a public identifier name is:

```
reg.type // owner // description // language
```

Let's break down all four elements of a public identifier name.

- **reg.type** is a plus sign (+) sign if the owner is registered according to the ISO 9070 standard. It is a minus sign (-) if it is not. Most are not, so the minus sign (-) is more common.

- **owner** is the creator of the identifier (individual or company).

- **description** is a text description of the identifier. Keep this short even though you can have a lot of text here.

- **language** is the two character language code.

An example of a public identifier could be:

```
-//David Harding/My First Public Identifier//EN
```

When an XML processor encounters a PUBLIC identifier, it attempts to generate a location URI. If it can't locate the file, it uses a secondary parameter called a *system literal*, which you have to provide. The following is an example of that.

```
<!ENTITY my.entities PUBLIC
   "-//David Harding//My First Public Identifier//EN",
   "http://www.davidharding.com/public_id.xml">
```

In this example, if the public identifier cannot be located using the public identifier name, the URI is used.

Tip: Public identifier resolution has not been completely solved yet. The method that is currently the defacto standard is the same as that of SGML, which uses and external catalog file called "catalog" that points to the various locations for each possible public identifier.

Since this is not the official standard, and is likely to change, we are going to focus on system identifiers since that is what you are going to use most of (or all of) the time anyway.

Using External Entities

To really see how external entities can be put to use, let's break our employee XML document into multiple pieces. Our goal is to break the physical structure into manageable chunks, while maintaining the logical structure.

Let's start with our completed EmployeesWithDTD.xml document. We are going to break it into three pieces:

- **EmployeeMaster.xml**

  This XML document will contain:

  - the employee content model (DTD) that we have defined,
  - references to the other employee entities, which in turn will contain individual employee information.

- **JohnDoe.xml**

  This is the first employee entity that is referenced by EmployeeMaster.xml. It contains the content (elements and attributes) for John Doe. This content already adheres to the DTD defined in the master document.

- **JaneDoe.xml**

  This is the second employee entity that is referenced by EmployeeMaster.xml. It contains the content (elements and attributes) for Jane Doe. This too adheres to the DTD defined in the master document.

Let's look at the EmployeeMaster.xml file first.

```
<?xml version="1.0"?>
<!DOCTYPE employees [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee (name, position, address, phone)>
   <!ATTLIST employee id ID #REQUIRED>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone (main, office*, fax*, mobile*,
home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>

   <!ENTITY employee1 SYSTEM "JohnDoe.xml">
   <!ENTITY employee2 SYSTEM "JaneDoe.xml">
]>

<employees>

&employee1;
&employee2;

</employees>
```

File: Chapter 4\EmployeeMaster.xml.

The next two documents contain the data for employees John and Jane Doe. Their contents is as follows:

```
<employee id="A1234">
    <name>
        <first>John</first>
        <last>Doe</last>
    </name>
    <position>Programmer</position>
    <address>
        <street>123 Main Street</street>
        <city>Anywhere</city>
        <state>CA</state>
        <zip>92000</zip>
    </address>
    <phone>
        <main>(714) 555-1000</main>
        <fax>(714) 555-1001</fax>
    </phone>
</employee>
```

File: Chapter 4\JohnDoe.xml.

```
<employee id="A2345">
    <name>
        <first>Jane</first>
        <last>Doe</last>
    </name>
    <position>Systems Analyst</position>
    <address>
    </address>
    <phone>
        <main>(714) 555-2000</main>
        <mobile>(949) 555-2200</mobile>
        <home>(949) 555-2220</home>
    </phone>
</employee>
```

File: Chapter 4\JaneDoe.xml

The employee specific files are pretty straightforward. The master document needs some review, though. Let's look at the <!ENTITY> declarations that point to the employee data files. These are located in the DTD (which makes sense being that the DTD is where they need to be declared).

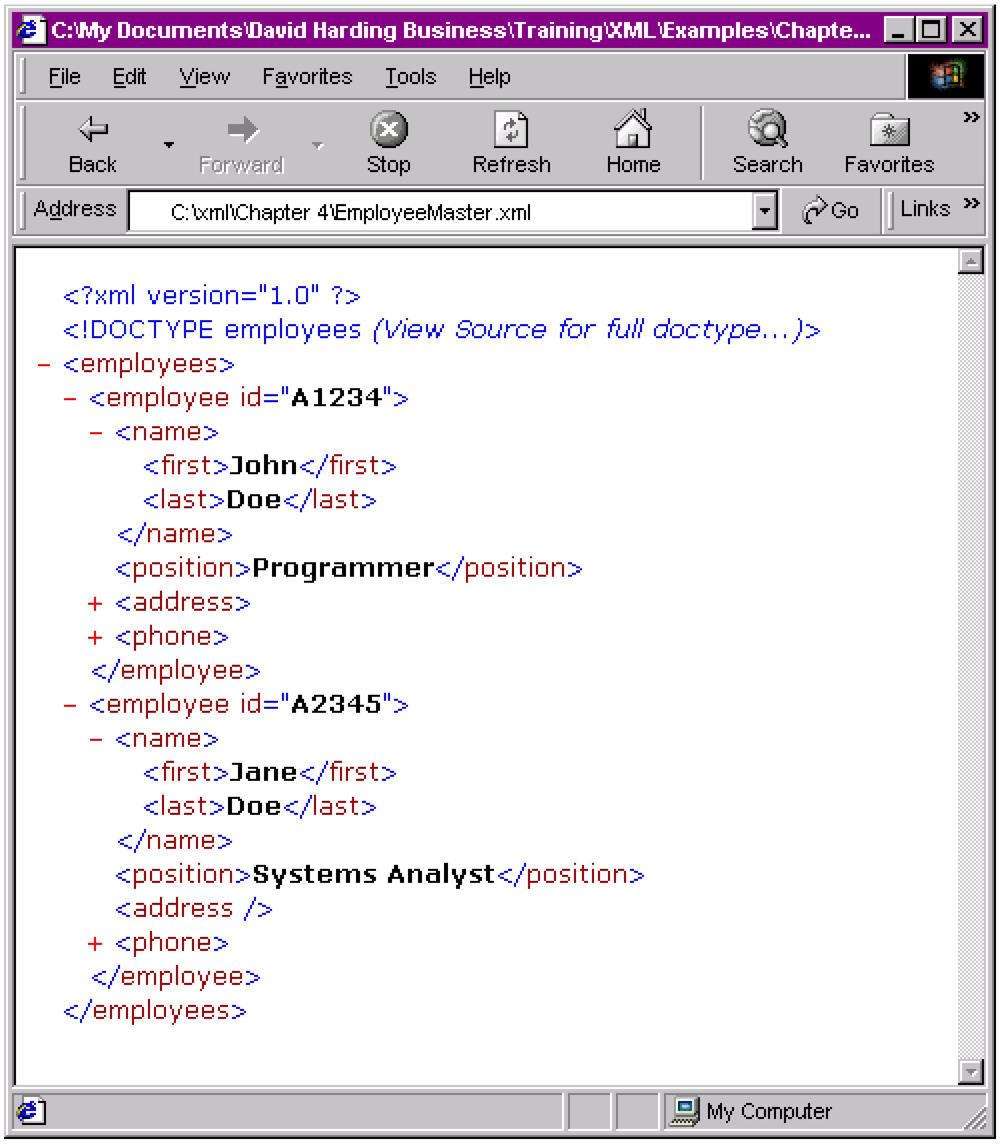
```
<!ENTITY employee1 SYSTEM "JohnDoe.xml">
<!ENTITY employee2 SYSTEM "JaneDoe.xml">
```

The words "employee1" and "employee2" are now system entity references that can be expanded just like character entity references. The difference is that instead of expanding simple text, the XML processor is going to expand the contents of the external entities (documents).

Let's see how this is done within the master document.

```
<employees>
&employee1;
&employee2;
</employees>
```

Within the root element is expanded the contents of both external entities just as if we were expanding an internal entity. When we view the EmployeeMaster.xml document in IE5, the logical content model is intact and treated as a single entity, even though it is physically structured as three distinct entities.


C:\My Documents\David Harding Business\Training\XML\Examples\Chapte...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\xml\Chapter 4\EmployeeMaster.xml
Go Links
My Computer

```
  <?xml version="1.0" ?>
  <!DOCTYPE employees (View Source for full doctype...)>
- <employees>
  - <employee id="A1234">
    - <name>
        <first>John</first>
        <last>Doe</last>
      </name>
      <position>Programmer</position>
    + <address>
    + <phone>
    </employee>
  - <employee id="A2345">
    - <name>
        <first>Jane</first>
        <last>Doe</last>
      </name>
      <position>Systems Analyst</position>
      <address />
    + <phone>
    </employee>
  </employees>
```

File: Chapter 4\EmployeeMaster.xml.

When to use External Entities

The word to keep in mind here is *reusability*. Even though this is not a term often used in the XML community (software developers seem to have the market cornered on it use) it really is the goal.

Using external entities, you can break your content into multiple pieces that can be included in multiple XML documents. This is going to reduce the amount of duplication between documents that may reference the same content.

Tips for Good Content Modeling

As you have seen, you can create rich content models using XML and DTD's. The DTD's that we have created are pretty simple, but as you can imagine, they can get quite complex. Some developers like to take a "less is more" approach to content modeling in XML, but the rule of thumb among the "gurus" is just the opposite. The common statement is, "Always create the richest content model that you can." This may seem strange, but there are some good reasons why.

- It is easier to remove than to add.

  This is a truism in XML and data modeling in general. It is always easier to remove things from a model than to add them later.

- You cannot account for everything.

  Yes, you are a good developer. It doesn't matter. There are going to be things that you forget, your subject matter experts forget (or don't tell you), or data will become available that wasn't before. This being the case, never "paint yourself into a corner" when it comes to content modeling. True, the true "open system" (or content model) is an unrealistic dream, but why make it more difficult an yourself? Add everything that you can, and account for the fact that you are going to miss something anyway.

- You are going to have to support it.

  The reward for building a good content model for your documents is that they are eventually going to become *legacy data.* Most developers cringe when they think of supporting something (even their own creations) years down the road. If you do it right up front, it is going to save you a lot of time and headache later. This is in line with the thought of creating the content model as rich as you can because it will be easier to deal with things that are already in the model than to try and add them later.

- It's small now, but it's going to grow.

  There is a rule in software and data storage. If it starts out as a small system, it is going to grow and become mission critical. We have all created that "stop-gap" application or database that was only supposed to last through a couple of weeks or months and it ended up becoming a workflow process for an entire department. It happens. Account for it. Build the content model right and growth won't be nearly as big an issue.

Well-Formed and Valid

There are two issues that we are going to address in this chapter:

1. The creation of *well-formed* documents

2. The creation of *valid* documents.

“Well-formed” and “valid” are often the most misunderstood words among all of the terminology relating to XML. In this chapter, we are going to discuss what these terms relate to, how you can check to make sure that your documents are both well-formed and valid. Before we get started, let’s put a high-level definition to these terms as they relate to XML.

A well-formed document is one that is *syntactically* correct. It’s elements and attributes are correct as to their declaration.

A valid document is one whose *content* is correct. The values contained within elements and attributes adhere to their respective element and attribute declarations.

In a nutshell, the term “well-formed” has to do with the structure of your content, and the term “valid” has to do with the content itself. In this chapter, we are going to discuss how to create well-formed documents.

Well-Formed Documents

As you have seen so far, you can create very complex content models using XML. If your document is not well-formed, XML processors are not going to be able to use them. The XML specification goes so far as to say that an entity is not an XML document until it is well-formed.

There are rules to document well-formedness, many of which you have already encountered. A document is considered well-formed if:

- It contains one or more elements.
- It has just one element (the root element) that contains all other elements.
- The elements contained within the root are properly nested inside each other (synchronous structures).
- Element names are correctly matched by name and case.
- The names of attributes only appear once in a given element's start tag.
- Attribute values are enclosed in either single or double quotes.
- Attribute values do not reference external entities, either directly or indirectly.
- The replacement text for any entity referenced in an attribute does not contain a less-than (<) character (it can contain the predefined entity &lt;).
- All entities are declared before they are used.
- No entity references contain the name of an unparsed entity.
- The logical and physical structures are properly nested.

If an XML processor encounters a violation against these rules of well-formedness in a document, it will stop processing the document and generate one or more fatal errors.

This is very different than the HTML standard and the way that browsers support HTML. Browsers are very lenient and try to make sense out of everything that it is given. It is rumored that 75% of the code in both Internet Explorer and Netscape Navigator exists to handle bad and/or invalid HTML code!

Checking for Well-Formedness

What is the best (and easiest) way to make sure that your documents are well-formed? There are a number of tools that are available that you can use to check your documents. Best of all, many of the good ones are free! These tools are called non-validating parsers (validation is a whole separate topic we are working towards) and they check your XML syntax. Non-validating parsers are like the old "lint" tool that C/C++ programmers use to pick out the not-so-obvious errors in their source code.

A simple (and free) tool for checking for well-formedness is the Microsoft XML Notepad. Do not use Microsoft Internet Explorer to check your XML documents! Internet Explorer is a browser and by its nature is extremely lenient. On the other hand, XML Notepad provides strict well-formedness checking.

Tip: XML Notepad can be downloaded for free off the Microsoft web site at http://msdn.microsoft.com/xml/notepad/download.asp. This does not mean that it is the best. It just well works for our example.

Unlike many of the non-validating parsers out there, XML Notepad uses a graphical user interface. Most of these tools use a command-line interface. Based on your preferences, you can choose the set of tools that you will add to your XML developers toolkit. Many of them even come with source code so that you can modify them to your liking.

Using XML Notepad

Let's walk through a document that has errors and check it for well-formedness using XML Notepad. This is going to be familiar to software developers who are used to using compilers to catch errors in their source code.

The following is the contents of our flawed document. The lines in bold contain the errors. See if you can identify what the errors are before we use XML Notepad to identify them for us.

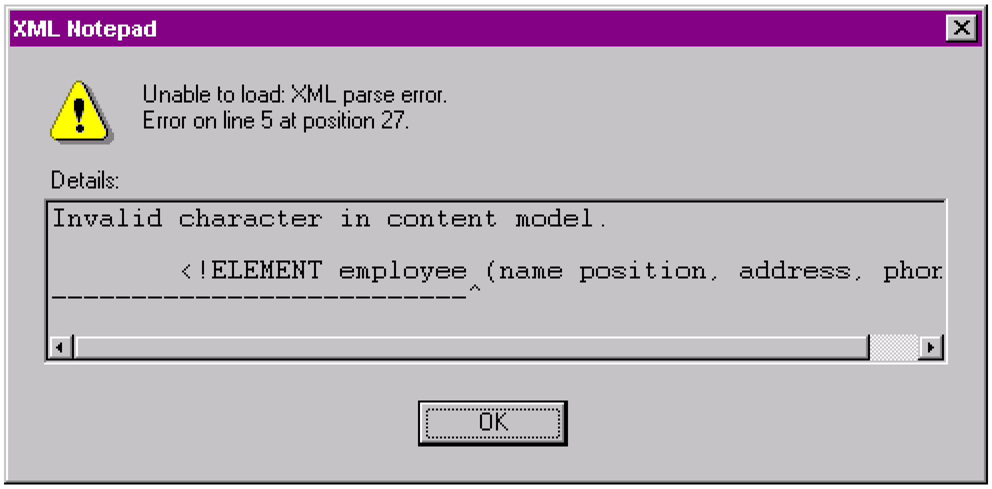
```
<?xml version="1.0"?>
<!DOCTYPE employes [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee (name position, address, phone)>
   <!ATTLIST employee id ID>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>
   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone (main, office*, fax*, mobile*,
home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>
]>

<employees>

    <employee id=A1234>
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
            <main>(714) 555-1000</main>
            <fax>(714) 555-1001</fax>
        </phone>
    </employee>

    <employee id="A2345">
        <name>
            <first>Jane</first>
            <last>Doe</last>
        </name>
        <position>Systems Analyst</position>
        <address>
        </address>
        <phone>
            <main>(714) 555-2000<main>
            <mobile>(949) 555-2200</mobile>
            <home>(949) 555-2220</home>
        </phone>
    </employee>
</employees>
```

File: Chapter 5\Employees_NotWellFormed.xml

Let's open the document in XML Notepad and see how it processes the errors. Like most non-validating parsers, it only reports one error at a time. Do not expect these tools to work like a compiler (e.g. C++, Java, etc.), which gives you a list of errors and/or warnings. This is more of a nuisance than anything, but it does take time to get through a complex document.

> **Tip:** It is often a good idea to check for well-formedness throughout the development of your document instead of creating the entire document then running it through a parser.

When we attempt to open the document the for first time, the error that we get is:



```
Unable to load: XML parse error.
Error on line 5 at position 27.

Invalid character in content model.

        <!ELEMENT employee (name position, address, phor
--------------------------^
```

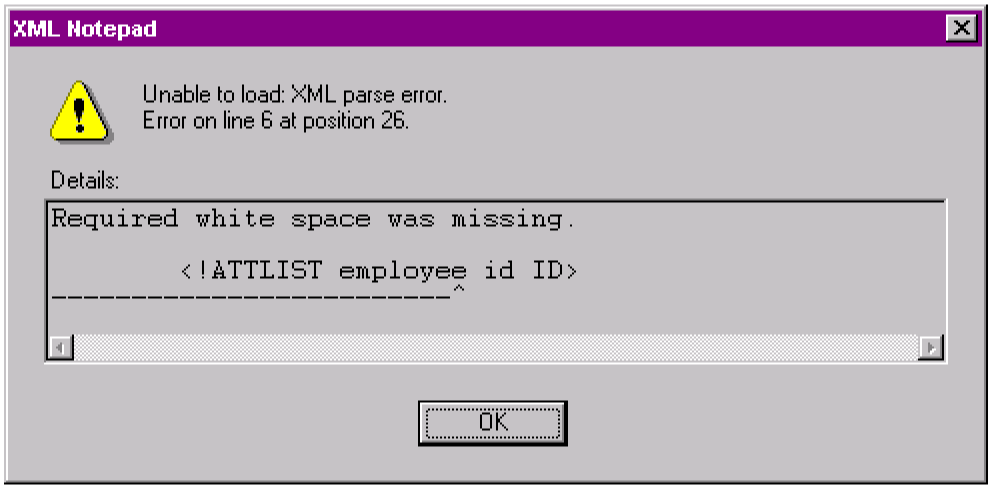
Error 1: Line 5 – missing comma (syntax error).

The first error is not entirely obvious, and this is not uncommon when working with XML parsers. The real error is that there is a comma (,) missing between the "name" and "position" declarations. The parser ran into the error and could not determine how to parse it, so it put us on the first position it could not understand.

To fix the error, we need to change line 5 by adding a comma between "name" and "position".

```
<!ELEMENT employee (name, position, address, phone)>
```

Let's reload the document and get the next error.



```
Unable to load: XML parse error.
Error on line 6 at position 26.

Details:
Required white space was missing.

        <!ATTLIST employee id ID>
-------------------------^
```

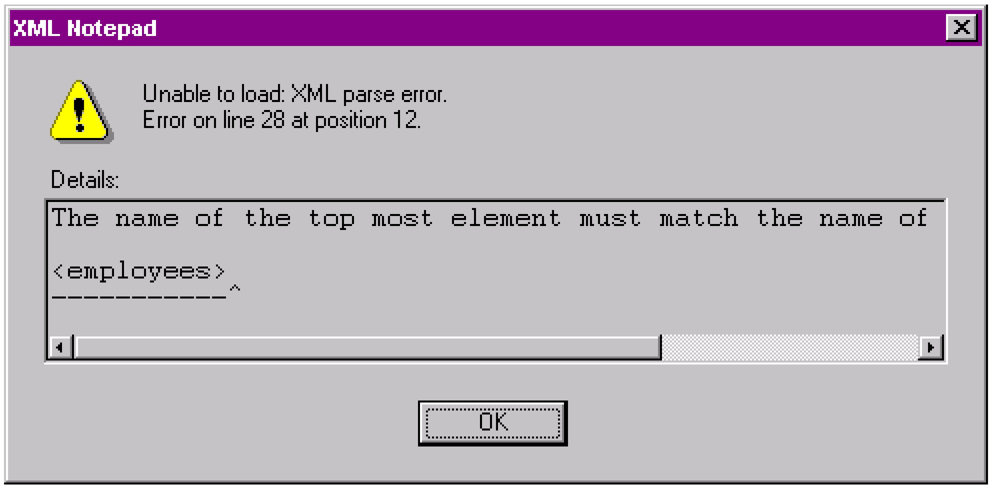
Error #2: Line 6 – missing #REQUIRED declaration (syntax error).

Now this one is somewhat ridiculous. The actual error is that the #REQUIRED attribute is missing. In this example, the error message is not even close to the actual error. However, in XML Notepad's defense, when this same error is run through several other parsers, you get almost the exact same error.

The fix is to add the #REQUIRED attribute to the declaration because ID attributes are required.

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

The next error message is much more descriptive.



```
Unable to load: XML parse error.
Error on line 28 at position 12.

Details:
The name of the top most element must match the name of

<employees>
-----------^
```

Error #3: Line 28 – the root element is named differently than the name specified in the <!DOCTYPE> tag (syntax error).

The full error message reads: "The name of the top most element must match the name of the DOCTYPE declaration."

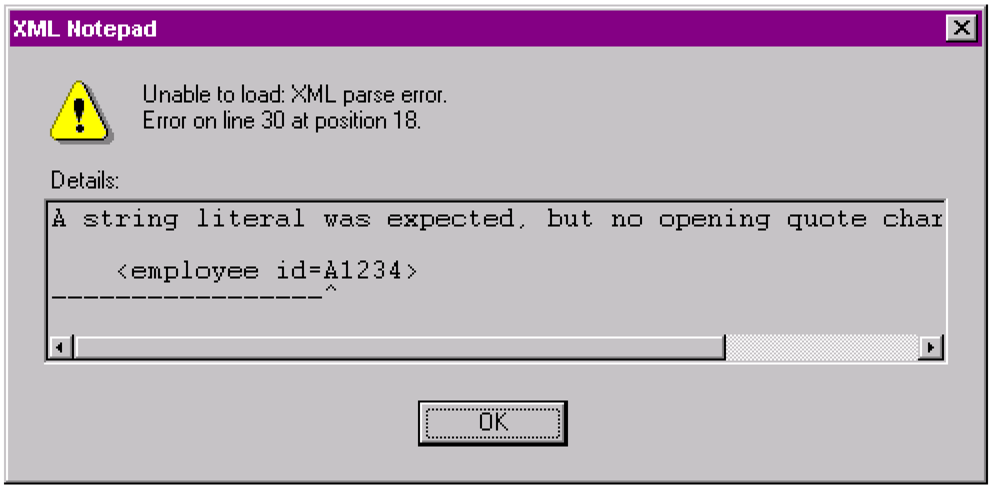
To fix this error, we need to match the name specified in the <!DOCTYPE> tag to the name of the root element.

```
<!DOCTYPE employes [

. . . (omitted)

<employees>
```

In this case, we have misspelled the name in the <!DOCTYPE> tag. We can correct the error by fixing the typo.

```
<!DOCTYPE employees [
```

Let's reload the document and try again. The next error that we encounter will be in the document content. However, this error is still regarding the well-formedness of the document, and is not a violation of a declaration.



```
Unable to load: XML parse error.
Error on line 30 at position 18.

Details:
A string literal was expected, but no opening quote char

    <employee id=A1234>
-----------------^
```

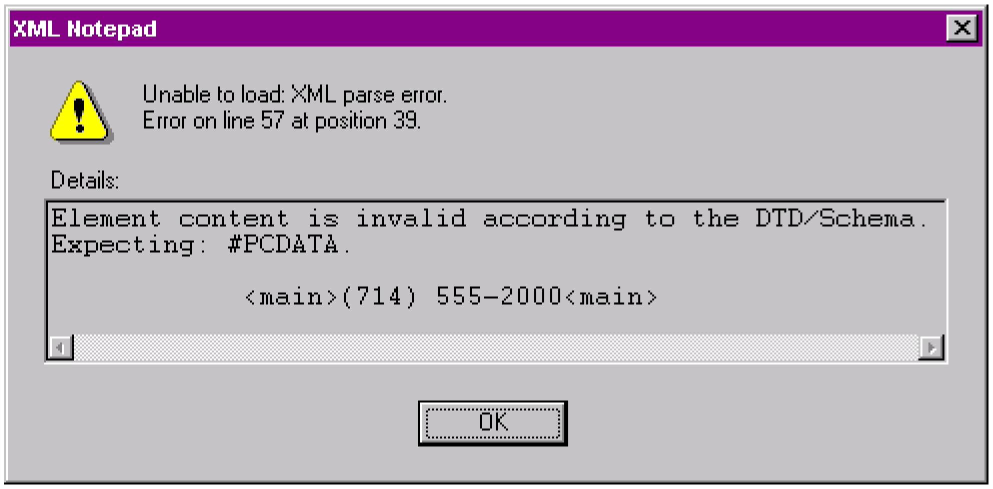
Error #4: Line 30 – missing quotes around the value assigned to the id attribute.

The full error message reads: "A string literal was expected, but no opening quote character was found."

This is indeed a syntax error. All string literals assigned as values to attributes need to be enclosed in quotes. To fix the error, we need to add some quotes.

```
<employee id="A1234">
```

The final error that XML Notepad finds is going to be a bit tricky. The reason being that the error message that we get is not going to give us much of a clue as to what the real error is. This is because XML Notepad attempts to do validity checking as well as checking for well-formedness. This being the case, it is going to report a validity error to us, which it is, but it is due to a lack of well-formedness in the document.



```
Unable to load: XML parse error.
Error on line 57 at position 39.
```

```
Element content is invalid according to the DTD/Schema.
Expecting: #PCDATA.

           <main>(714) 555-2000<main>
```

Error #5: Line 57 – missing terminator tag </main> (syntax error).

The error we received is in fact correct, event though the real error is that the terminator tag for <main> is missing. The XML processor is returning a content error to us. Here is how the XML processor came to this conclusion:

1. The <main> element is declared as type #PCDATA. These means that it can only contain text data. It cannot contain any other elements. This is the rule defined in the DTD.

2. The <main> element is a required element that can only appear once within a <phone> element. This is the way that it is declared within the <phone> element's declaration. The XML processor found the text data immediately following the greater-than symbol (>) in the open tag and started to process the text.

3. The XML processor encountered another <main> element tag. This is a typo; the termination backslash (/) is missing. To the XML processor, however, this was another <main> element. This violates both rules defined previously. Namely, <main> can only appear once and it can only have text data.

This is why we got a content error, and not a syntax error. When you run this example through a purely non-validating parser, which XML Notepad is not, a syntax error is generated regarding the missing terminator tag.

Tips: It is often a good idea to have multiple parsers that you can use during this process. This is very helpful when one parser gives you a less than helpful error message or when you are having trouble understanding what the real error is.

Another tip is to use a separate non-validating parser from your validating parser. This way you can check for well-formedness apart from checking for validity. The last error we encountered is a good example of why it is often good to separate the two.

Choosing a Non-Validating Parser

As was mentioned before, choosing your toolset is a personal matter. There are a number of really good parsers out there to choose from. The following are some resources that you can go to on the Internet to download tools and learn about different resources available.

- The *Expat* non-validating parser is one of the nicest parsers around. It is a command line tool that is very simple to use and basic to understand. It also includes source code. You can download it for free at ftp://ftp.jclark.com/pub/xml/expat.zip.

- The *TclXML* parser is available at http://www.zveno.com/zm.cgi/in-tclxml/. This non-validating parser is cross platform and is very robust.

- Finally, James Tauber's XmlSoftware.com (http://www.xmlsoftware.com/parsers/) is the definitive source for finding both validating and non-validating parsers. This is the place to go to start looking at what is available and putting together your XML tool chest.

Valid Documents

You have seen two types of XML documents thus far: those that have DTD's, and those that do not. XML documents that do have DTD's appear to have very few restrictions. You can add elements and attributes at will without regard to structure as a whole. The only thing that you have to worry about is that the document is syntactically correct. This being the case, you might question the need for DTD's at all. Actually, not having a DTD in your document puts significant restrictions on your document as regards validity.

If you want an XML document to be valid, but do not want to have a DTD associated with it, the following rules must be adhered to:

- All the attribute values in the document must be specified; there can be no default values. This makes sense because default values are specified in the DTD.
- There can be no references to internal entities in the DTD except those that are predefined in the XML specification (i.e. &lt;, &gt, &apos;, &quot;). Again, this makes sense because internal entities are defined in the DTD.
- There can be no attributes that are subject to normalization (contain entity references).
- In elements with content consisting of only elements, there can be no white-space (e.g. space, tab, etc.) between the starting tag and the content of the first element. For example, the following would be illegal:

  ```
  <el1>   <el2>data</el2></el1>
  ```

As you can see, the DTD goes along way to helping XML processors deal with your documents.

Tip: It is very easy to create XML documents without going to the process of creating a DTD. However, the time saved by not creating a DTD will often be spent working around problems/errors that you did not account for.

Make a habit of always using a DTD. Even for simple documents. It makes them easier to create, readable and maintainable. It also makes them easier to process by XML processors.

Going through the process of validating your document is beneficial for a number of reasons. None the least of which is making sure that the content within the document adheres to the structure defined in the DTD.

Validation is Relative

Validating a given document requires an examination of the document's DTD and then checking to make sure that the content contained therein matches that DTD. DTD's can be strict or loose. Also, DTD's don't have much in the way of strict type checking. What is considered valid depends on the content models that you create. How you design your DTD will affect the rules of validation that are applied to the contents of the document. This is a very important point.

Validating parsers are written in simple terms. They understand the content model you define, they understand the element structure contained within the document, and that is it.

Tip: What you may consider valid in an XML document may have a different meaning to a validating parser. It can only compare the DTD to the content that is defined in the document.

Another way to look at is that validation checks the markup, but actually does little or nothing to check what's between the markup (the content), other than look for more markup.

Validating parsers cannot provide much in the way of element content relationship, typos, and data integrity. XML is not a like relational database in that way. There are new additions to the XML standards that are being proposed that address some of these issues, but they are a ways off from being implemented. Vendors such as Microsoft, Netscape and Sun are adding their own extensions to XML support within their products, but these are not portable. Keep this in mind if you decide to use them. These same vendors have done the same thing with HTML, and that is why there are many browser specific web sites on the Internet today. Vendor specific technologies is the antithesis of what XML is all about!

Validating Parsers

Like non-validating parsers, there are a number of validating parsers that are available for free download. As mentioned before, the James Tauber's XmlSoftware.com (http://www.xmlsoftware.com/parsers/) is the first place you may want to go to find a list of parsers that can be downloaded for free.

Final Words on Well-Formedness and Validity

It is easy to create well-formed XML documents if you take the proper measures in the beginning. Creating a DTD to define a concise content model is the first step. Syntactically defining the DTD is not the difficult part. Creating a logical content model that supports the data that you are storing is where it can get difficult. However, this is not a new problem. It is the same for database administrators as it is to application developers. Structuring the data so that it supports all possible cases that your software may face is an art.

In conclusion, always check to make sure that your documents are well-formed and valid. This will allow XML processors to deal with the document in the way that is intended. Get in the habit of always creating a DTD for your documents, no matter how simple they are (remember, it is always the simple ones that grow and become complex). Use a non-validating parser to check to make sure that your document is well-formed. Then use a validating parser to check for validity. It is often a good idea to have a couple of different types of these at your disposal.

Another way to create well-formed and valid documents is to use an XML editor that validates while you are creating the document. There are a few XML editors out there (like Microsoft XML Notepad), but they have a long way to go before they are truly useful. The standard Window's Notepad and the Microsoft Visual InterDev development environment are still the editors of choice for many.

Using External Document Type Definitions

By now you know the importance of the DTD in an XML document. Already, you have enough knowledge about DTD's to create rich content models. However, there are a number of other capabilities that can be exploited within the DTD that are going to be discussed now.

Document Type Declarations

Now that you are going to always implement a DTD, let's talk about the association of a DTD with a XML document. This is done using a *document type declaration* (not to be confused with a document type *definition*). You have already seen and used document type declarations, but it is important to understand the distinction between the declaration and the DTD. The syntax that you have seen so far is:

```
<!DOCTYPE dtd.name [ internal.subset ]>
```

So far, we have created internal DTD's. In other words, the DTD is declared inside of the document. All element, attribute, and entity declarations have been defined in the internal subset of the DTD. Soon we are going to discuss *external* subsets and DTD's.

Standalone Documents

A standalone document is one that does not require any external support and without reference to any other files. In other words, the XML document stands on its own. We can specify that our documents are standalone in the XML processing instruction.

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

The statement *standalone="yes"* means that there are no markup declarations external to the document entity. This does not mean that our document cannot reference external entities (other documents, binary files, etc.) provided that the declarations of the external entities are contained inside the document entity (inside the internal DTD subset). This is a very important point and it needs to be stressed further.

The existence of external entities does not mean that the document is not standalone. As long as these entities are declared within the internal subset of the DTD, the document is still considered to be a standalone document. This is a very important point and may take a while to completely understand. The term “standalone” does not only refer to documents that are physically self-contained.

Tip: All that the “standalone” attribute does is make it explicit as to whether or not the document is a standalone document.

External DTD’s

Using internal DTD’s, you can create sophisticated XML documents that require no external references whatsoever. Considering that a DTD can get rather large and complex, and the same DTD could possibly be used by multiple XML documents, wouldn’t it be nice if you could create a DTD that could be referenced by many XML documents? Think of all the typing (or copying/pasting) that you could save. Well, you can. It is called an *external DTD.*

To an XML processor, there is only one DTD. It consists of the internal DTD subset and the external DTD subset. In other words, the XML processor uses both the internal and external DTD’s (if you have one) and treats them as one. In XML, the internal DTD subset is read before the external DTD subset and so takes precedence.

Associating an external DTD is similar to creating an internal DTD in that it is done within the document type declaration (<!DOCTYPE> tag.). The syntax is:

```
<!DOCTYPE name PUBLIC|SYSTEM external.pointer
[internal.subset]>
```

Notice the new keywords PUBLIC and SYSTEM. These are the same *identifiers* that we used when we declared external entities. Let’s review them again in relation to DTD’s.

SYSTEM

A system identifier is a *Universal Resource Identifier* (URI) that is used to retrieve the DTD. A URI is similar to a URL, except that URL’s are intended for network (Internet) use. For all intents and purposes, the two terms are interchangeable.

A identifier can reference an absolute position or a relative position as is exemplified in the following code fragments:

```
<!-- Absolute reference -->
<!DOCTYPE doc.name SYSTEM
”http://www.server.com/external.dtd”>

<!-- Relative reference -->
```

```
<!DOCTYPE doc.name SYSTEM "../DTDs/external.dtd">
```

Most of the time you will use SYSTEM identifiers for your external DTD's.

PUBLIC

A public identifier is the officially recorded identifier for a DTD. Officially registered public identifiers are registered by the creator of the DTD with The International Standards Organization (ISO). Public identifier names are assigned by the American Graphic Communication Association (CGA) under the direction of the ISO.

The syntax for a public identifier name is:

```
reg.type // owner // description // language
```

Let's break down all four elements of a public identifier name.

- **reg.type** is a plus sign (+) sign if the owner is registered according to the ISO 9070 standard. It is a minus sign (-) if it is not. Most are not, so the minus sign (-) is more common.
- **owner** is the creator of the identifier (individual or company).
- **description** is a text description of the identifier. Keep this short even though you can have a lot of text here.
- **language** is the two character language code.

An example of a public identifier could be:

```
-//David Harding/My First Public Identifier//EN
```

When an XML processor encounters a PUBLIC identifier, it attempts to generate a location URI. If it can't locate the file, it uses a secondary parameter called a *system literal*, which you have to provide. The following is an example of that.

```
<!DOCTYPE david.harding PUBLIC
   "-//David Harding//My First Public Identifier//EN",
   "http://www.davidharding.com/public id.dtd">
```

In this example, if the public identifier cannot be located using the public identifier name, the URI is used.

Using an External DTD

Let's rework our employees document so that it uses an external DTD. The first thing we are going to do is create a new file called "Employees.dtd", which will contain our external subset. This will include the entire DTD that we have created for our employees document up to this point.

```
   <!ELEMENT employees ( employee+ )>
   <!ELEMENT employee (name, position, address, phone)>
   <!ATTLIST employee id ID #REQUIRED>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone (main, office*, fax*, mobile*,
home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>
```

File: Chapter 6\Employees.dtd

Notice that there is no document type declaration (i.e. <!DOCTYPE> tag). There can only be one declaration in an XML document. Remember, external XML that is included in your document is considered part of the same document. In this case, the external and the internal DTD's are treated the same.

Next, let's modify the Employees.xml file so that it references the new external DTD file instead of declaring the content model internally. The new XML document looks like this:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE employees SYSTEM "Employees.dtd" [
<!-- Declare the internal subset here -->
]>

<employees>
    <employee id="A1234">
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
```

```
                <main>(714) 555-1000</main>
                <fax>(714) 555-1001</fax>
            </phone>
        </employee>

        <employee id="A2345">
            <name>
                <first>Jane</first>
                <last>Doe</last>
            </name>
            <position>Systems Analyst</position>
            <address>
            </address>
            <phone>
                <main>(714) 555-2000</main>
                <mobile>(949) 555-2200</mobile>
                <home>(949) 555-2220</home>
            </phone>
        </employee>
    </employees>
```

File: Chapter 6\Employees.xml

There are a number of advantages to using external DTD’s. The most obvious is that you can create rich content models that can be utilized by multiple documents. If the content model changes, you only have to change it one place.

Be careful of multiple declarations when using external DTD’s. When you use an external DTD, you still have the ability to declare things inside the internal subset. If the XML processor finds a duplicate declaration, there are going to be errors. Consider the following example. The our Employee external DTD, the <name> element is declared as containing to child elements: <first> and <last>.

```
<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last  (#PCDATA)>
```

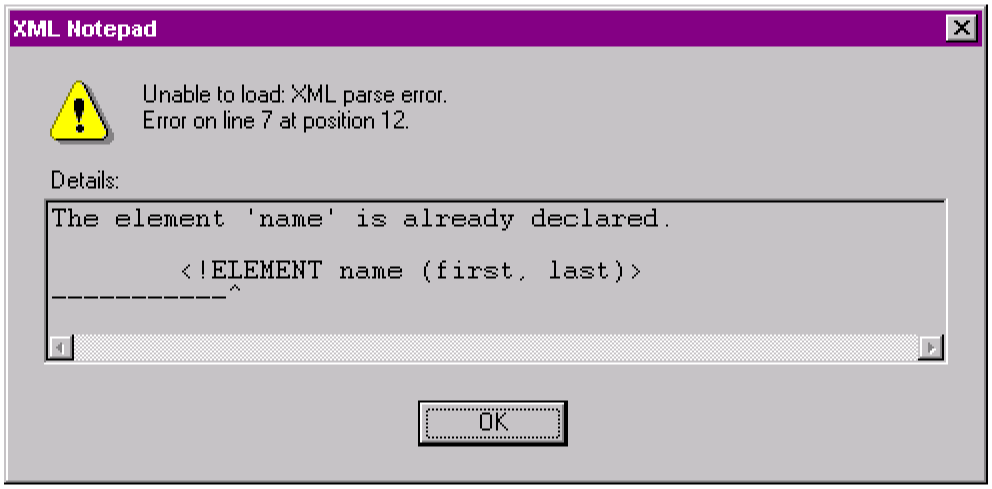
File: Chapter 6\Employees.dtd

What happens if we declare the <name> element again in our internal subset? In the following example, we include the external DTD and redefine the <name> element as a #PCDATA element.

```
<!DOCTYPE employees SYSTEM "Employees.dtd" [
   <!ELEMENT name (#PCDATA)>
]>
```

File: Chapter 6\Employees.xml

Notice what happens when we open the document in XML Notepad.



```
Unable to load: XML parse error.
Error on line 7 at position 12.
```

```
The element 'name' is already declared.

        <!ELEMENT name (first, last)>
-----------^
```

Declaration error in Employees2.xml displayed in XML Notepad.

This may seen like an obvious thing, but when you have an external DTD that is a couple of hundred lines long and an internal subset that is just as descriptive, these types of things happen. Using external DTD's is a great thing, and it is recommended that you consider always using them, but make sure that you understand the external DTD while you are creating your internal subset.

Why Use an External DTD?

External DTD's are a great way to maintain a consistent content model across multiple XML documents. This is an especially useful feature when creating content models that must be standardized across an organization or department. By using external DTD's, you can make sure that all your XML documents remain consistent.

Code Listings

The following are the complete code listings for all of the samples used in this lesson.

Chapter 4\EmployeeMaster.xml

```
<?xml version="1.0"?>
<!DOCTYPE employees [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee (name, position, address, phone)>
   <!ATTLIST employee id ID #REQUIRED>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone
      (main, office*, fax*, mobile*, home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>

   <!ENTITY employee1 SYSTEM "JohnDoe.xml">
   <!ENTITY employee2 SYSTEM "JaneDoe.xml">
]>

<employees>

&employee1;
&employee2;

</employees>
```

Chapter 4\EmployeesWithDTD.xml

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

<!DOCTYPE employees [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee
      (name, position, address, phone)>
   <!ATTLIST employee id ID #REQUIRED>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone
      (main, office*, fax*, mobile*, home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>
]>

<employees>

    <employee id="A1234">
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
            <main>(714) 555-1000</main>
            <fax>(714) 555-1001</fax>
        </phone>
    </employee>

    <employee id="A2345">
        <name>
```

```
                <first>Jane</first>
                <last>Doe</last>
            </name>
            <position>Systems Analyst</position>
            <address>
            </address>
            <phone>
                <main>(714) 555-2000</main>
                <mobile>(949) 555-2200</mobile>
                <home>(949) 555-2220</home>
            </phone>
        </employee>

    </employees>
```

Chapter 4\EnumerationAttributes.xml

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

<!DOCTYPE colors [

   <!ELEMENT colors ( primary+ )>

   <!ELEMENT primary EMPTY>
      <!ATTLIST primary color
            ( RED | GREEN | BLUE ) "RED">
]>

<colors>

   <primary color="RED"/>
   <primary color="GREEN"/>
   <primary color="BLUE"/>

   <!-- ERROR!
   <primary color="YELLOW"/>
   -->

</colors>
```

Chapter 4\IDAttributes.xml

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

<!DOCTYPE id.attribs [

   <!ELEMENT id.attribs
      ( element1*, element2*, element3* )>

   <!ELEMENT element1 EMPTY>
      <!ATTLIST element1 id ID #REQUIRED>

   <!ELEMENT element2 EMPTY>
      <!ATTLIST element2 el1.id IDREF #REQUIRED>

   <!ELEMENT element3 EMPTY>
      <!ATTLIST element3 el1.ids IDREFS #REQUIRED>
]>

<id.attribs>

   <!-- Create three <element1> elements each with
      a unique ID -->
   <element1 id="A1234"/>
   <element1 id="B1234"/>
   <element1 id="C1234"/>

   <!-- Reference a single element -->
   <element2 el1.id="A1234"/>

   <!-- Reference multiple elements -->
   <element3 el1.ids="A1234 B1234 C1234"/>

</id.attribs>
```

Chapter 4\JohnDoe.xml

```xml
<employee id="A1234">
    <name>
        <first>John</first>
        <last>Doe</last>
    </name>
    <position>Programmer</position>
    <address>
        <street>123 Main Street</street>
        <city>Anywhere</city>
        <state>CA</state>
        <zip>92000</zip>
    </address>
    <phone>
        <main>(714) 555-1000</main>
        <fax>(714) 555-1001</fax>
    </phone>
</employee>
```

Chapter 4\JaneDoe.xml

```
<employee id="A2345">
    <name>
        <first>Jane</first>
        <last>Doe</last>
    </name>
    <position>Systems Analyst</position>
    <address>
    </address>
    <phone>
        <main>(714) 555-2000</main>
        <mobile>(949) 555-2200</mobile>
        <home>(949) 555-2220</home>
    </phone>
</employee>
```

Chapter 4\SpecialElementTypes.xml

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

<!DOCTYPE root.element [
   <!ELEMENT empty.element EMPTY>
   <!ELEMENT unrestricted.element ANY>
]>

<root.element>

   <empty.element/>

   <unrestricted.element>
      <empty.element/>
      <empty.element/>
      <empty.element/>
   </unrestricted.element>

</root.element>
```

Chapter 4\TokenAttributes.xml

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

<!DOCTYPE tokens [

   <!ELEMENT tokens ( element1+)>

   <!ELEMENT element1 ( #PCDATA )>
      <!ATTLIST element1 attr1 NMTOKEN #IMPLIED>
      <!ATTLIST element1 attr2 NMTOKENS #IMPLIED>
]>

<tokens>

   <element1 attr1="Token1"
        attr2="123_Token1 Token2 Token3">
      Some text here...
   </element1>

</tokens>
```

Chapter 5\ Employees_NotWellFormed.xml

```
<?xml version="1.0"?>
<!DOCTYPE employes [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee
      (name position, address, phone)>
   <!ATTLIST employee id ID>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address
      (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone (main, office*, fax*, mobile*,
home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>
]>

<employees>

    <employee id=A1234>
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
            <main>(714) 555-1000</main>
            <fax>(714) 555-1001</fax>
        </phone>
    </employee>
    <employee id="A2345">
        <name>
            <first>Jane</first>
            <last>Doe</last>
        </name>
        <position>Systems Analyst</position>
        <address>
        </address>
        <phone>
            <main>(714) 555-2000<main>
            <mobile>(949) 555-2200</mobile>
            <home>(949) 555-2220</home>
        </phone>
    </employee>

</employees>
```

Chapter 5\Employees_WellFormed.xml

```
<?xml version="1.0"?>
<!DOCTYPE employees [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee (name, position, address, phone)>
   <!ATTLIST employee id ID #REQUIRED>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone (main, office*, fax*, mobile*,
home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>
]>

<employees>

    <employee id="A1234">
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
            <main>(714) 555-1000</main>
            <fax>(714) 555-1001</fax>
        </phone>
    </employee>

    <employee id="A2345">
        <name>
            <first>Jane</first>
            <last>Doe</last>
        </name>
        <position>Systems Analyst</position>
        <address>
        </address>
        <phone>
            <main>(714) 555-2000</main>
            <mobile>(949) 555-2200</mobile>
            <home>(949) 555-2220</home>
        </phone>
    </employee>

</employees>
```

Chapter 5\TagError.xml

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

<!DOCTYPE employees [
   <!ELEMENT employees ( employee+ )>

   <!ELEMENT employee (name, position, address, phone)>
   <!ATTLIST employee id ID #REQUIRED>

   <!ELEMENT name (first, last)>
   <!ELEMENT first (#PCDATA)>
   <!ELEMENT last  (#PCDATA)>

   <!ELEMENT position (#PCDATA)>

   <!ELEMENT address (street?, city?, state?, zip?)>
   <!ELEMENT street (#PCDATA)>
   <!ELEMENT city   (#PCDATA)>
   <!ELEMENT state  (#PCDATA)>
   <!ELEMENT zip    (#PCDATA)>

   <!ELEMENT phone
      (main, office*, fax*, mobile*, home*)>
   <!ELEMENT main   (#PCDATA)>
   <!ELEMENT office (#PCDATA)>
   <!ELEMENT fax    (#PCDATA)>
   <!ELEMENT mobile (#PCDATA)>
   <!ELEMENT home   (#PCDATA)>
]>

<employees>

    <employee id="A1234">
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
            <main>(714) 555-1000</main>
            <fax>(714) 555-1001</fax>
        </phone>
    </employee>

    <employee id="A2345">
        <name>
            <first>Jane</first>
            <last>Doe</last>
        </name>
        <position>Systems Analyst</position>
        <address>
        </address>
        <phone>
            <main>(714) 555-2000</main>
            <mobile>(949) 555-2200</mobile>
            <home>(949) 555-2220</home>
        </phone>
    </employee>

</employees>
```

Chapter 6\Employees.dtd

```
<!ELEMENT employees ( employee+ )>

<!ELEMENT employee
   (name, position, address, phone)>
<!ATTLIST employee id ID #REQUIRED>

<!ELEMENT name (first, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last  (#PCDATA)>

<!ELEMENT position (#PCDATA)>

<!ELEMENT address (street?, city?, state?, zip?)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city   (#PCDATA)>
<!ELEMENT state  (#PCDATA)>
<!ELEMENT zip    (#PCDATA)>

<!ELEMENT phone
   (main, office*, fax*, mobile*, home*)>
<!ELEMENT main   (#PCDATA)>
<!ELEMENT office (#PCDATA)>
<!ELEMENT fax    (#PCDATA)>
<!ELEMENT mobile (#PCDATA)>
<!ELEMENT home   (#PCDATA)>
```

Chapter 6\Employees.xml

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

<!DOCTYPE employees SYSTEM "Employees.dtd" [

   <!-- Declare the internal subset here -->

]>

<employees>

    <employee id="A1234">
        <name>
            <first>John</first>
            <last>Doe</last>
        </name>
        <position>Programmer</position>
        <address>
            <street>123 Main Street</street>
            <city>Anywhere</city>
            <state>CA</state>
            <zip>92000</zip>
        </address>
        <phone>
            <main>(714) 555-1000</main>
            <fax>(714) 555-1001</fax>
        </phone>
    </employee>

    <employee id="A2345">
        <name>
            <first>Jane</first>
            <last>Doe</last>
        </name>
        <position>Systems Analyst</position>
        <address>
        </address>
        <phone>
            <main>(714) 555-2000</main>
            <mobile>(949) 555-2200</mobile>
            <home>(949) 555-2220</home>
        </phone>
    </employee>

</employees>
```