Back in Hour 3, “Defining Data with DTD Schemas,” you learned how to describe a custom markup language using a DTD. Although DTDs can certainly get the job done, a more modern alternative is available for describing XML-based markup languages. This alternative is called XML Schema, and schemas developed using XML Schema are known as XSDs. Getting back to the Scott Adams quote, some XML developers have argued that maybe there is a shred of Dilbert ridiculousness to XML Schema given the complexity of the resulting code involved in creating an XSD. The W3C (World Wide Web Consortium) initially set out to improve upon DTDs by developing a schema technology that was based on XML. The end result is known as the XML Schema Definition Language, or XSD, which is now an official W3C standard. XSD schemas are used similarly to DTDs in that they provide a means of defining a custom markup language and validating XML documents. However, XSDs are considerably more powerful than DTDs and give you much finer control over the design of markup languages. As with many technologies, power and flexibility adds complexity, so you’ll find that XSD schemas are generally a bit larger and more complex than DTDs. This hour introduces you to XML Schema and shows you how to create XSD schemas that can be used to validate your own documents.

In this hour, you’ll learn

- The basics of XML Schema
- How to use elements and attributes in XSD schemas
- How to work with simple and complex data types
- How to build a complete XSD schema and use it to validate a document

XML Schema Construction Basics

In a DTD you lay out the elements and attributes that can be used to describe a particular type of data. Similar to a DTD, XML Schema allows you to create markup languages by carefully describing the elements and attributes that can be used to code information. Unlike DTDs, schemas created with XML Schema are coded in XML, which makes them more
consistent in terms of keeping everything in the XML domain; if you recall, DTDs use their own cryptic language. The language used to describe markup languages in XML Schema is XSD. Schemas created in this language are often referred to simply as XSDs.

The XSD language is an XML-based language, which means you use XML elements and attributes to describe the structure of your own custom markup languages. This means that XSD itself was created in XML. Although this might seem confusing at first, keep in mind that it is necessary for there to be a means of validating XSD documents, which means the XSD language must be spelled out in terms of XML. More specifically, the elements and attributes in the XSD language are described in none other than a DTD. This is because it isn’t exactly possible to use XSD to describe the XSD schema. Admittedly, this is a “chicken and egg” kind of problem because we’re talking about creating a schema for a schema language that is in turn used to create schemas. Which one comes first? To be honest, it really doesn’t matter. Rather than confuse you further, I’d rather push on and learn how an XSD document comes together. The main point here is that XSD is an XML-based markup language, similar in many ways to any other custom markup language you might create.

Because XSD schema documents are really just XML documents, you must include the familiar XML declaration at the start of them:

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

After entering the XML declaration, you’re ready to start coding the XSD document. All of the elements and attributes in XSD are part of what is known as a namespace, which if you recall from Hour 5, “Putting Namespaces to Use,” is essentially a grouping of elements and attributes that guarantees uniqueness in their names. You typically assign a namespace a prefix that is used throughout a document to reference elements and attributes within the namespace. In order to reference XSD elements and attributes, you must first declare the XSD namespace in the root element of the XSD document. The prefix of the XSD namespace is typically set to xsd, which means that all XSD elements and attributes are preceded by the prefix xsd and a colon (:). The root element of XSD documents is named xsd:schema. Following is an example of how you declare the XSD namespace in the xsd:schema element:

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

In this code, the xmlns:xsd attribute is used to set the XSD namespace, which is a standard URI made available by the W3C. This means that you must precede each element and attribute name with xsd:. So, to recap, the general structure of an XSD schema document has the following form:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
</xsd:schema>
```
Of course, this code has no content within the root element, so it isn’t doing much. However, it lays the groundwork for the basis of all XSD schema documents.

**XSD Data Types**

The XSD language is defined by the elements and attributes that can be used within it, as well as their relationship to one another. At the heart of XSD are data types, which determine the type of data that can be represented by a particular piece of markup code. For example, numeric data in XSD is coded differently than text data and therefore has an associated data type that is used when creating a schema with XSD. There are two different general types of data used in XSDs: simple data and complex data. *Simple data* corresponds to basic pieces of information such as numbers, strings of text, dates, times, lists, and so on. *Complex data*, on the other hand, represents more involved information such as mixed elements and sequences of elements. Generally speaking, complex data types are built upon simple data types.

Simple data types can be used with both elements and attributes and provide a means of describing the exact nature of a piece of information. The `<xsd:element>` element is used to create elements of a simple type, whereas the `<xsd:attribute>` element is used to create attributes. Following are a few examples of each:

```xml
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="occupation" type="xsd:string"/>
<xsd:attribute name="birthdate" type="xsd:date"/>
<xsd:attribute name="weight" type="xsd:integer"/>
```

Although these examples show how simple data types enter the picture with elements and attributes, they don’t reveal the relationship between elements and attributes, which is critical in any XSD document. These relationships are established by complex data types, which are capable of detailing the content models of elements. Following is an example of how simple data types can be used within a complex type to describe the content model of an element named `person`:

```xml
<xsd:element name="person">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="occupation" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexContent>
</xsd:element>
```
Keep in mind that this XSD code describes a custom markup language that is used to create XML documents. In order to fully understand how the schema code works, it's a good idea to take a look at what XML code might look like that adheres to the schema. Following is an example of some XML document data that follows the data structure laid out in the prior XSD schema code:

```
<person birthdate="1969-10-28" weight="160">
  <name>Milton James</name>
  <title>Mr.</title>
  <occupation>mayor</occupation>
</person>
```

This code should look much more familiar to you as it is basic XML code with custom elements and attributes. It doesn’t take too much analysis to see that this code adheres to the XSD schema code you just saw. For example, the `person` element includes two attributes, `birthdate` and `weight`, as well as three child elements: `name`, `title`, and `occupation`. Unlike a DTD, the schema is able to carefully describe the data type of each element and attribute. For example, the `birthdate` attribute is a date (xsd:date), not just a string that happens to store a date, and the `weight` attribute is an integer number (xsd:integer).

**XSD Schemas and XML Documents**

You now have a basic knowledge of how a schema is used to establish a markup language that in turn is used to create XML documents. What you don’t know is how a schema is actually associated with such documents. If you recall, a DTD is associated with a document by way of a document type declaration. XSDs don’t rely on a document type declaration and instead use a special attribute called `noNamespaceSchemaLocation`. To associate a schema with an XML document for validation purposes, you set this attribute of the root element to the location of the schema document. However, in order to use this attribute you must first declare the namespace to which it belongs. Following is how this is accomplished in XML code:

```
<contacts xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="contacts.xsd">
  <person birthdate="1969-10-28" weight="160">
    <name>Milton James</name>
    <title>Mr.</title>
    <occupation>mayor</occupation>
  </person>
</contacts>
```

There is also a `schemaLocation` attribute for referencing a schema that has its own namespace. This is useful if you want to explicitly reference elements using a prefix for the schema. You find out more about this attribute later in the lesson.
This code shows how to declare the appropriate namespace and then set the noNamespaceSchemaLocation attribute for the schema document. Assuming the schema for the contacts document is located in the file named contacts.xsd, this XML document is ready for validation. This brings up an important point regarding schema documents—they are coded in XML but they are stored in files with a .xsd extension. This makes it possible to determine quickly if a file is an XSD schema.

Many XML documents are stored in files with extensions other than .xml. Although .xml is certainly a suitable extension for any XML document, it is generally better to use the more specific extension dictated by the markup language, assuming that such an extension exists. As an example, in the previous hour you worked with SVG documents that were stored in files with a .svg extension.

By the way

Working with Simple Types

XSD includes several different simple data types, or simple types, that make it possible to model a wide range of data in XML documents. These types can be classified according to the kind of data they represent. Following are the major categories of simple data types supported in the XSD language, along with the specific XSD elements associated with each category:

- String types—xsd:string
- Boolean types—xsd:boolean
- Number types—xsd:integer, xsd:decimal, xsd:float, xsd:double
- Date and time types—xsd:time, xsd:timeInstant, xsd:duration, xsd:date, xsd:month, xsd:year, xsd:century, xsd:recurringDate, xsd:recurringDay
- Custom types—xsd:simpleType

These simple types are typically used to create elements and attributes in a schema document. In order to create an element based upon a simple type, you must use the xsd:element element, which has two primary attributes used to describe the element: name and type. The name attribute is used to set the element name, which is the name that appears within angle brackets (&lt;&gt;) when you use the element in XML code. The type attribute determines the type of the element and can be set to a simple or complex type. Following are the element examples you saw a little earlier in the hour that make use of the xsd:string simple type:

<xsd:element name="name" type="xsd:string"/>
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="occupation" type="xsd:string"/>
Attributes are created in much the same manner as elements and even rely on the same two attributes, name and type. However, you create an attribute using the xsd:attribute element. Following are the attribute examples you saw earlier that use the xsd:date and xsd:integer simple types:

```
<xsd:attribute name="birthdate" type="xsd:date"/>
<xsd:attribute name="weight" type="xsd:integer"/>
```

Now that you understand how simple types enter the picture with elements and attributes, you’re ready to learn more about the types themselves.

**The String Type**

The string type represents a string of text and is represented in the type attribute by the xsd:string value. The string type is probably the most commonly used type in XSD. Following is an example of how to use the xsd:string value to create a string element:

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

In an XML document, this element might be used like this:

```
<name>Milton James</name>
```

**The Boolean Type**

The Boolean type represents a true/false or yes/no value and is represented in the type attribute by the xsd:boolean value. When using a Boolean type in an XML document, you can set it to true or false, or 1 or 0, respectively. Following is an example of an attribute that is a Boolean type:

```
<xsd:attribute name="retired" type="xsd:boolean"/>
```

In an XML document, this attribute might be used like this:

```
<person retired="false">
  <name>Milton James</name>
</person>
```

**Number Types**

Number types are used in XSD to describe elements or attributes with numeric values. The following number types are available for use in schemas to represent numeric information:

- `xsd:integer`—Integer numbers (with no fractional part); for example, 3
- `xsd:decimal`—Decimal numbers (with a fractional part); for example, 3.14
Working with Simple Types

- **xsd:float**—Single precision (32-bit) floating point numbers; for example, 6.022E23
- **xsd:double**—Double precision (64-bit) floating point numbers; same as float but for considerably more precise numbers

If you'd like to exert exacting control over the sign of integer numbers, you might consider using one of these additional numeric types: **xsd:positiveInteger**, **xsd:negativeInteger**, **xsd:nonPositiveInteger**, or **xsd:nonNegativeInteger**. The latter two types are zero-inclusive, whereas the first two don't include zero.

To create an element or attribute for a numeric piece of information, you simply select the appropriate number type in the XSD. Following is an example of a couple of attributes that are number types:

```xml
<xsd:attribute name="height" type="xsd:decimal"/>
<xsd:attribute name="weight" type="xsd:integer"/>
```

In an XML document, this attribute might be used like this:

```xml
<person height="5.75" weight="160">
    <name>Milton James</name>
</person>
```

### Date and Time Types

XSD includes support for date and time types, which is very useful when it comes to modeling such information. Following are the different date and time types that are supported in XSD:

- **xsd:time**—A time of day; for example, 4:40 p.m.
- **xsd:timeInstant**—An instant in time; for example, 4:40 p.m. on August 24, 1970
- **xsd:duration**—A length of time; for example, 3 hours and 15 minutes
- **xsd:date**—A day in time; for example, August 24, 1970
- **xsd:month**—A month in time; for example, August, 1970
- **xsd:year**—A year in time; for example, 1970
- **xsd:century**—A century; for example, 20th century
- **xsd:recurringDate**—A date without regard for the year; for example, August 24
- **xsd:recurringDay**—A day of the month without regard for the month or year; for example, the 24th of the month
To create an element or attribute for a date or time, you must select the appropriate date or time type in the XSD. Following is an example of an attribute that is a date type:

```xml
<xsd:attribute name="birthdate" type="xsd:date"/>
```

This attribute is of type `xsd:date`, which means that it can be used in XML documents to store a day in time, such as October 28, 1969. You don't just set the `birthdate` attribute to `October 28, 1969`, however. Dates and times are actually considered highly formatted pieces of information, so you must enter them according to predefined formats set forth by the XSD language. The format for the `xsd:date` type is `ccyy–mm–dd`, where `cc` is the century (19), `yy` is the year (69), `mm` is the month (10), and `dd` is the day (28). The following code shows how you would specify this date in the `birthdate` attribute using the CCYY-MM-DD format:

```xml
<person birthdate="1969-10-28" height="5.75" weight="160">
  <name>Milton James</name>
</person>
```

Other date and time types use similar formats. For example, the `xsd:month` type uses the format `ccyy–mm`, `xsd:year` uses `ccyy`, and `xsd:century` uses the succinct format `cc`. The `xsd:recurringDate` type uses `– mm-dd` to format recurring dates, whereas the `xsd:recurringDay` type uses `---dd`. Following is an example of the `xsd:recurringDate` type so that you can see how the dashes fit into things:

```xml
<person birthday="–10--28" height="5.75" weight="160">
  <name>Milton James</name>
</person>
```

In this example, an attribute named `birthday` is used instead of `birthdate`, with the idea being that a birthday is simply a day and month without a birth year (a birth date implies a specific year). Notice that an extra dash appears at the beginning of the `birthday` attribute value to serve as a placeholder for the intentionally missing year.

The remaining time types are `xsd:duration`, `xsd:time`, and `xsd:timeInstant`. The `xsd:duration` type uses an interesting format to represent a length of time—to specify a value of type `xsd:duration` you must enter the length of time according to the format `PyyYmmMddDTthHmmsSsS`. The `P` in the format indicates the period portion of the value, which consists of the year (`yy`), month (`mm`), and day (`dd`). The `T` in the format begins the optional time portion of the value and consists of hours (`hh`), minutes (`mm`), and seconds (`ss`). You can precede a time duration value with a minus sign (`-`) to indicate that the duration of time goes in the reverse direction (back in time). Following is an example of how you would use this format to code the time duration value 3 years, 4 months, 2 days, 13 hours, 27 minutes, and 11 seconds:

```xml
<worldrecord duration="P3Y4M2DT13H27M11S"/>
</worldrecord>
The \texttt{xsd:time} type adheres to the format \texttt{hh:mm:ss.sss}. In addition to specifying the hours (hh), minutes (mm), and seconds (ss.sss) of the time, you may also enter a plus (+) or minus (-) sign followed by hh:mm to indicate the offset of the time from Universal Time (UTC). As an example, the U.S. Central Standard Time zone is six hours behind UTC time, so you would need to indicate that in an \texttt{xsd:time} value that is in Central Standard Time (CST). Following is an example of a CST time:

\begin{verbatim}
<meeting start="15:30:00-06:00">
</meeting>
\end{verbatim}

UTC stands for Coordinated Universal Time and is the same as Greenwich Mean Time (GMT). UTC time is set for London, England, and therefore must be adjusted for any other time zones. Other time zones are adjusted by adding or subtracting time from UTC time. For example, U.S. Pacific Standard Time (PST) is UTC – 8, whereas Japan is UTC + 9.

Notice in the code that the hours in the time are entered in 24-hour form, also known as “military time,” meaning that there is no a.m. or p.m. involved. The time specified in this example is 3:30 p.m. CST.

The \texttt{xsd:timeInstant} type follows the type \texttt{ccyy–mm–ddThh:mm:ss.sss} and is essentially an \texttt{xsd:time} type with the year, month, and day tacked on. As an example, the previous \texttt{xsd:time} type could be coded as a \texttt{xsd:timeInstant} type with the following code:

\begin{verbatim}
<meeting start="2002-02-23T15:30:00-06:00">
</meeting>
\end{verbatim}

The \texttt{xsd:timeInstant} type follows the type \texttt{ccyy–mm–ddThh:mm:ss.sss} and is essentially an \texttt{xsd:time} type with the year, month, and day tacked on. As an example, the previous \texttt{xsd:time} type could be coded as a \texttt{xsd:timeInstant} type with the following code:

\begin{verbatim}
<meeting start="2002-02-23T15:30:00-06:00">
</meeting>
\end{verbatim}

### By the Way

UTC stands for Coordinated Universal Time and is the same as Greenwich Mean Time (GMT). UTC time is set for London, England, and therefore must be adjusted for any other time zones. Other time zones are adjusted by adding or subtracting time from UTC time. For example, U.S. Pacific Standard Time (PST) is UTC – 8, whereas Japan is UTC + 9.

### Custom Types

One of the neatest things about XSD is how it allows you to cook up your own custom data types. Custom data types allow you to refine simple data types to meet your own needs. For example, you can limit the range of numbers for a number type, or constrain a string type to a list of possible strings. Regardless of how you customize a type, you always begin with the \texttt{xsd:simpleType} element, which is used to create custom simple types. Most of the time your custom types will represent a constraint of a simple type, in which case you'll also need to use the \texttt{xsd:restriction} element. The \texttt{xsd:restriction} element supports a type named \texttt{base} that refers to the base type you are customizing. Following is the general structure of a custom simple type:

\begin{verbatim}
<xsd:simpleType name="onetotenType">
  <xsd:restriction base="xsd:integer">
    <!-- Restriction content here -->
  </xsd:restriction>
</xsd:simpleType>
\end{verbatim}
This code merely sets up the type to be created; the actual restrictions on the custom type are identified using one of several different elements. To constrain the range of values a number may have, you use one of the following elements:

- `xsd:minInclusive`—Minimum number allowed
- `xsd:minExclusive`—One less than the minimum number allowed
- `xsd:maxInclusive`—The maximum number allowed
- `xsd:maxExclusive`—One greater than the maximum number allowed

These types allow you to set lower and upper ranges on numeric values. Following is an example of how you would limit a numeric value to a range of 1 to 10:

```xml
<xsd:simpleType name="onetotenType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1"/>
    <xsd:maxInclusive value="10"/>
  </xsd:restriction>
</xsd:simpleType>
```

It's important to note that this code only establishes a custom type named `onetotenType`; it doesn't actually create an element or attribute of that type. In order to create an element or attribute of a custom type, you must specify the type name in the `type` attribute of the `xsd:element` or `xsd:attribute` element:

```xml
<xsd:element name="rating" type="onetotenType"/>
```

Although this approach works fine, if you plan on using a custom type with only a single element or attribute, you may want to declare the type directly within the element or attribute, like this:

```xml
<xsd:element name="rating">
  <xsd:simpleType>
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="1"/>
      <xsd:maxInclusive value="10"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

In addition to controlling the bounds of simple types, it is also possible to control the length of them. For example, you might want to limit the size of a string of text. To do so, you would use one of the following elements:

- `xsd:length`—The exact number of characters
- `xsd:minlength`—The minimum number of characters
- `xsd:maxlength`—The maximum number of characters
Because the xsd:length element specifies the exact length, you can’t use it with the xsd: minlength or xsd:maxlength elements. However, you can use the xsd: minlength and xsd:maxlength elements together to set the bounds of a string’s length. Following is an example of how you might control the length of a string type:

```xml
<xsd:element name="password">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="8"/>
      <xsd:maxLength value="12"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

In this example, a password element is created that must have at least 8 characters but no more than 12. This shows how to control the length of strings, but it is also possible to control the length of numbers. More specifically, you can use the xsd:precision and xsd:scale elements to control how many digits appear to the left or right of a decimal point; this is known as the precision of a number. The xsd:precision element determines how many total digits are allowed in a number, whereas xsd:scale determines how many of those digits appear to the right of the decimal point. So, if you wanted to allow monetary values up to $9999.00 with two decimal places, you would use the following code:

```xml
<xsd:element name="balance">
  <xsd:simpleType>
    <xsd:restriction base="xsd:decimal">
      <xsd:precision value="6"/>
      <xsd:scale value="2"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Keep in mind that the xsd:precision and xsd:scale elements set the maximum allowable number of digits for the total number and to the right of the decimal place, which means that all of the following examples are valid for the balance element:

```xml
<balance>3.14</balance>
<balance>12.95</balance>
<balance>1.1</balance>
<balance>524.78</balance>
```

One other customization I’d like to mention at this point has to do with default and fixed values. In the event that an element or attribute isn’t specified in a document, you may want to declare a default value that is assumed. You may also want to limit an element or attribute so that it can have only one possible value, which is known as a fixed value. Default and fixed values are established with the default and fixed
attributes of the xsd:element and xsd:attribute elements. Following are a few examples of default and fixed elements and attributes:

```xml
<xsd:element name="balance" type="xsd:decimal" default="0.0"/>
<xsd:element name="pi" type="xsd:decimal" fixed="3.14"/>
<xsd:attribute name="expired" type="xsd:boolean" default="false"/>
<xsd:attribute name="title" type="xsd:string" fixed="Mr."/>
```

The balance element has a default value of 0.0, which means it will assume this value if it isn’t used in a document. The same thing goes for the expired attribute, which assumes the default value of false if it goes unused. The pi element is fixed at the value 3.14, which means if it is used it must be set to that value. Similarly, the title attribute must be set to Mr. if it is used. Notice that none of the examples are defined as having both default and fixed values; that’s because you aren’t allowed to define both a default and a fixed value for any single element or attribute.

In addition to customizing simple types as you’ve seen thus far, you can also do some other interesting things with custom types. The next few sections explore the following data types, which are considered slightly more advanced custom types:

- Enumerated types
- List types
- Patterned types

## Enumerated Types

Enumerated types are used to constrain the set of possible values for a simple type and can be applied to any of the simple types except the Boolean type. To create an enumerated type, you use the xsd:enumeration element to identify each of the possible values. These values are listed within an xsd:restriction element, which identifies the base type. As an example, consider an element named team that represents the name of an NHL hockey team. Following is an example of how you might code this element with the help of enumerated types:

```xml
<xsd:element name="team">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Nashville Predators"/>
      <xsd:enumeration value="Detroit Red Wings"/>
      <xsd:enumeration value="St. Louis Blues"/>
      <xsd:enumeration value="Chicago Blackhawks"/>
      <xsd:enumeration value="Columbus Blue Jackets"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```
This code obviously doesn’t include every NHL team, but you get the idea. The important thing to note is that the schema won’t allow an XML developer to use any value for the team element other than those listed here. So, if you were creating a fantasy hockey data service that allowed people to access hockey data on a team-by-team basis, they would only be able to choose from your predefined list of teams. Enumerated types therefore provide a very effective means of tightly defining data that is limited to a set of predefined possibilities.

**List Types**

Whereas enumerated types force an XML developer to use a value from a predefined set of values, list types allow an XML developer to provide multiple values for a given element. The xsd:list element is used to create list types, which are useful any time you need to allow for a list of information. As an example, you might want to create an element that stores rainfall totals for each month of the year as part of an XML-based weather application. Following is code that carries out this function:

```xml
<xsd:element name="rainfall">
  <xsd:simpleType>
    <xsd:list base="xsd:decimal">
      <xsd:length value="12"/>
    </xsd:list>
  </xsd:simpleType>
</xsd:element>
```

This code allows you to list exactly 12 decimal numbers, separated by white space. Following is an example of what the XML code might look like for the rainfall element:

```xml
<rainfall>1.25 2.0 3.0 4.25 3.75 1.5 0.25 0.75 1.25 1.75 2.0 2.25</rainfall>
```

If you wanted to be a little more flexible and not require exactly 12 items in the list, you could use the xsd:minLength and xsd:maxLength elements to set minimum and maximum bounds on the list. You can also create a completely unbounded list by using the xsd:list element by itself, like this:

```xml
<xsd:element name="cities">
  <xsd:simpleType>
    <xsd:list base="xsd:string"/>
  </xsd:simpleType>
</xsd:element>
```

**Patterned Types**

Patterned types are undoubtedly the trickiest of all custom types, but they are also the most powerful in many ways. Patterned types allow you to use a regular expression to establish a pattern that tightly controls the format of a simple type. A regular expression is a coded pattern using a special language that describes an arrangement
of letters, numbers, and symbols. The regular expression language employed by XSD is fairly complex, so I won’t attempt a complete examination of it. Instead, I’d like to focus on the basics and allow you to investigate it further on your own if you decide you’d like to become a regular expression guru. Getting back to patterned types, you create a patterned type using the <xsd:pattern> element.

The <xsd:pattern> element requires an attribute named value that contains the regular expression for the pattern. Following are the building blocks of a regular expression pattern:

- . — Any character
- \d — Any digit
- \D — Any nondigit
- \s — Any white space
- \S — Any nonwhite space
- x? — One x or none at all
- x+ — One or more x’s
- x* — Any number of x’s
- (xy) — Groups x and y together
- x|y — x or y
- [xyz] — One of x, y, or z
- [x-y] — In the range x to y
- x{n} — n number of x’s in a row
- x{n,m} — At least n number of x’s but no more than m

See, I told you regular expressions are kind of tricky. Actually, these regular expression symbols and patterns aren’t too difficult to understand when you see them in context, so let’s take a look at a few examples. First off, how about a phone number? A standard U.S. phone number including area code is of the form xxx-xxx-xxxx. In terms of patterned types and regular expressions, this results in the following code:

```xml
<xsd:element name="phonenum">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d\d\d-\d\d\d-\d\d\d\d"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```
As you can see, the phonenum element is described by a pattern that consists of sequences of digits separated by hyphens. Although this pattern works fine, it’s important to note that regular expressions are extremely flexible, often offering more than one solution to a given problem. For example, the following xsd:pattern element also works for a phone number:

```xml
<xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
```

In this example a phone number is described using curly braces to indicate how many decimal numbers can appear at each position in the pattern. The code \d{3} indicates that there should be exactly three decimal numbers, whereas \d{4} indicates exactly four decimal numbers.

Let’s now consider a slightly more advanced regular expression pattern such as a pizza order. Our pizza order pattern must have the form s-c-t+t+t+, where s is the size (small, medium or large), c is the crust (thin or deep), and each t is an optional topping (sausage, pepperoni, mushroom, peppers, onions, and anchovies) in addition to cheese, which is assumed. Following is how this pizza order pattern resolves into an XSD regular expression pattern:

```xml
<xsd:element name="pizza">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="(small|medium|large)-(thin|deep)-(sausage+)?(pepperoni+)?(mushroom+)?(peppers+)?(onions+)?(anchovies+)?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Following is an example of how you might code a pizza element based upon this pattern:

```xml
<pizza>medium-deep-sausage+mushroom+</pizza>
```

Obviously, there is a great deal more that can be done with regular expression patterns. Hopefully this is enough information to get you going in the right direction with patterned types.

---

**Digging into Complex Types**

Complex data types represent a step up from simple types because they allow you to do more interesting things such as define the content model of elements. Complex types effectively build upon simple types, so your knowledge of simple types will come in quite handy as you work with complex types. All complex types are created using the xsd:complexType element. This element includes an attribute named name that is used to name a complex type. You can also declare a complex type directly within an element, in which case it doesn’t require a name.
Complex types can be broken down into four major classifications, as follows:

- Empty elements
- Element-only elements
- Mixed elements
- Sequences and choices

The next few sections explore these different complex types in detail.

**Empty Elements**

Empty elements contain no text content or child elements but are capable of having attributes. In fact, attributes are the only way to associate information with empty elements. You create empty elements using the `xsd:complexType` element in conjunction with the `xsd:complexContent` element. Following is an example of how you create an empty element:

```
<xsd:element name="automobile">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="xsd:anyType">
        <xsd:attribute name="vin" type="xsd:string"/>
        <xsd:attribute name="year" type="xsd:year"/>
        <xsd:attribute name="make" type="xsd:string"/>
        <xsd:attribute name="model" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Although this may seem like a lot of work to simply create an empty element with a few attributes, it is necessary. The `xsd:complexType` and `xsd:complexContent` elements are necessary to establish that this is a complex type, whereas the `xsd:extension` element is used to declare that there is no specific base type (`xsd:anyType`) for the element. Finally, the attributes for the element are created using the familiar `xsd:attribute` element. Following is an example of how you would use the automobile element in an XML document:

```
<automobile vin="SALHY1245SA61555" year="1995" make="Land Rover" model="Range Rover"/>
```

**Element-Only Elements**

Element-only elements are elements that contain only child elements with no text content. They can also contain attributes, of course, but no text content is allowed.
within an element-only element. To create an element-only element, you simply use the xsd:complexType element. Following is an example of an element-only element that contains a single child element:

```xml
<xsd:element name="assets">
  <xsd:complexType>
    <xsd:element name="automobile" type="automobileType"/>
  </xsd:complexType>
</xsd:element>
```

This code presents a new wrinkle because the child element of assets is declared as type automobileType. This kind of named complex type is created much like the named simple types you saw earlier in the hour. Another approach to creating an element-only element involves coding the element as a named type. Following is an example of how you might code the automobileType named complex data type:

```xml
<xsd:complexType name="automobileType">
  <xsd:complexContent>
    <xsd:extension base="xsd:anyType">
      <xsd:attribute name="vin" type="xsd:string"/>
      <xsd:attribute name="year" type="xsd:year"/>
      <xsd:attribute name="make" type="xsd:string"/>
      <xsd:attribute name="model" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

This is the same empty complex type you saw in the previous section, except in this case it has been created as a named type with additional attributes. Following is an example of XML code that uses the assets element, automobile element, and automobileType complex type:

```xml
<assets>
  <automobile vin="SALHV12458A661555" year="1995"
               make="Land Rover" model="Range Rover"/>
</assets>
```

You might be wondering exactly how useful the assets element is because it can contain only a single automobile element. In reality, practically all element-only elements are capable of storing multiple child elements, sometimes of different types. However, in order to allow for multiple child elements you must use a special construct known as a sequence. You learn about sequences a little later in this hour in the section titled “Sequences and Choices.”

## Mixed Elements

Mixed elements contain both text and child elements and are the most flexible of all elements. Text-only elements are considered a type of mixed element and can contain only text with no child elements. You create text-only elements using the
HOUR 7: Using XML Schema

xsd:complexType element in conjunction with the xsd:simpleContent element. Following is an example of a text-only element:

```xml
<xsd:element name="distance">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="units" type="xsd:string" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

The `distance` element can be used to store a distance traveled and is capable of using different units of measure to give meaning to the numeric content it stores. The actual distance is located in the element's content, whereas the units are determined by the `units` attribute, which is a string. It's important to notice the extra `use` attribute, which is set to `required`. This attribute setting makes the `units` attribute a requirement of the `distance` element, which means you must assign a value to the `units` attribute. Following is an example of how the `distance` element and `units` attribute might be used in an XML document:

```xml
<distance units="miles">12.5</distance>
```

Although text-only elements are certainly useful in their own right, there are some situations where it is necessary to have the utmost freedom in coding element content, and that freedom comes with the mixed element. Mixed elements are created similarly to other complex types but with the addition of the `xsd:mixed` attribute. Keep in mind that mixed types allow for text and child element content, as well as attributes. Following is an example of a mixed type:

```xml
<xsd:element name="message">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="emph" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="to" type="xsd:string" use="required"/>
    <xsd:attribute name="from" type="xsd:string" use="required"/>
    <xsd:attribute name="timestamp" type="xsd:timeInstant" use="required"/>
  </xsd:complexType>
</xsd:element>
```

In this example, a mixed element is created that can contain text, an `emph` element, and three attributes. Admittedly, I skipped ahead a little by placing the `emph` child element in a sequence, but that will be cleared up in the next section. Following is an example of how the `message` element might be used in an XML document:

```xml
<message to="you" from="me" timestamp="2001-03-14T2:45:00">I hope you return soon. I've <emph>really</emph> missed you!</message>
```
In this example the *emph* child element is used to add emphasis to the word “really” in the message.

**Sequences and Choices**

One powerful aspect of complex types is the ability to organize elements into sequences and choices. A *sequence* is a list of child elements that must appear in a particular order, whereas a *choice* is a list of child elements from which only one must be used. You create a sequence with the *xsd:sequence* element, which houses the elements that comprise the sequence. Following is an example of creating a sequence:

```xml
<xsd:element name="quiz">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="question" type="xsd:string">
      </xsd:element>
      <xsd:element name="answer" type="xsd:string">
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

In this example, the quiz element contains two child elements, question and answer, that must appear in the order specified. By default, a sequence can occur only once within an element. However, you can use the *xsd:minOccurs* and *xsd:maxOccurs* attributes to allow the sequence to occur multiple times. For example, if you wanted to allow the quiz element to contain up to 20 question/answer pairs, you would code it like this:

```xml
<xsd:element name="quiz">
  <xsd:complexType>
    <xsd:sequence minOccurs="1" maxOccurs="20">
      <xsd:element name="question" type="xsd:string">
      </xsd:element>
      <xsd:element name="answer" type="xsd:string">
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

You can set the *maxOccurs* attribute to *unbounded* to allow for an unlimited number of sequences. The *maxOccurs* attribute can also be used with individual elements to control the number of times they can occur.

Following is an example of how you might use the quiz element in an XML document:

```xml
<quiz>
  <question>What does XML stand for?</question>
  <answer>Extensible Markup Language</answer>
  <question>Who is responsible for overseeing XML?</question>
  <answer>World Wide Web Consortium (W3C)</answer>
  <question>What is the latest version of XML?</question>
  <answer>1.0</answer>
</quiz>
```
If you want to allow an element to contain one of a series of optional elements, you can use a choice. A choice allows you to list several child elements and/or sequences, with only one of them allowed for use in any given element. Choices are created with the `xsd:choice` element, which contains the list of choice elements. Following is an example of a choice:

```xml
<xsd:element name="id">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="ssnum" type="xsd:string">
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="birthdate" type="xsd:date"/>
        </xsd:sequence>
      </xsd:element>
      <xsd:element name="licensenum" type="xsd:string"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

In this example, an element named `id` is created that allows three different approaches to providing identification: social security number, name and birth date, or driver's license number. The choice is what makes it possible for the element to accept only one of the approaches. Notice that a sequence is used with the name and birth date approach because it involves two child elements. Following is an example of a few `id` elements that use each of the different choice approaches:

```xml
<id>
  <ssnum>123-89-4567</ssnum>
</id>
<id>
  <name>Milton James</name>
  <birthdate>1969-10-28</birthdate>
</id>
<id>
  <licensenum>12348765</licensenum>
</id>
```

If you're looking to create content models with little structure, you might consider using the `xsd:all` type, which is used to create complex types that can hold any number of elements in any order. The `xsd:all` element is used much like a sequence except that the child elements within it can appear any number of times and in any order.

By the Way

One last topic worth covering before moving on to a complete XSD example has to do with how data types are referenced. With the exception of the root element, which is automatically referenced in an XSD, global document components must be
referenced in order to actually appear as part of a document’s architecture. You should consider using a global component when you have an element or attribute that appears repeatedly within other elements. In most of the examples you’ve seen, the components have been declared locally, which means they are automatically referenced within the context that they appear. However, consider an element, such as the following one, which is declared globally:

```xml
<xsd:element name="password">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:minLength value="8"/>
            <xsd:maxLength value="12"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Although this element has been declared and is ready for use, it doesn’t actually appear within the structure of an XSD until you reference it. You reference elements using the `ref` attribute, which applies to both elements and attributes. Following is an example of how the `password` element might be referenced within another element:

```xml
<xsd:element name="login">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="userid" type="xsd:string"/>
            <xsd:element ref="password"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

In this example the `userid` element is created and used locally, whereas the `password` element is referenced from the previous global element declaration. Whether or not you use elements and attributes locally or globally primarily has to do with how valuable they are outside of a specific context; if an element or attribute is used only in a single location then you might as well simplify things and keep it local. Otherwise, you should consider making it a global component and then referencing it wherever it is needed using the `ref` attribute.

The difference between local and global elements has to do with how they are created, which determines how you can use them. An element (`userid` in the previous example) declared within another element is considered local to that element, and can only be used within that element. A global element (`password` in the previous example) is declared by itself and can be referenced from any other element.
Namespaces and XSD Schemas

Although I didn't mention it back in Hour 5, namespaces actually play an extremely important role in XSD schemas. I didn't want to digress too much in that hour and sidetrack you since I hadn't really gotten into the details of XSD schemas just yet. For this reason, it's now worth tackling the topic of namespaces as they relate to XSD schemas to clarify exactly how namespaces impact XSDs.

The xsd Prefix

The first thing to understand about namespaces and schemas is that there is nothing magical about the prefix xsd. The prefix xsd is used with the XSD schema as a means of referencing elements and attributes that are used to construct schemas for your own custom markup languages. For example, following is the namespace declaration for the etml.xsd example schema document, which you will learn about in just a moment:

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

This code shows how the prefix xsd is used to declare the XSD schema explicitly. Now that you understand how prefixes work with explicit namespace declarations, you know that this prefix could be named anything you want. Of course, there is no reason to deviate from xsd since it has become somewhat of a standard among XML developers, but I wanted to point out that there is nothing hardwired into XML when it comes to namespace prefixes.

Referencing Schema Documents

In addition to providing a means of referencing the schema of a schema document, namespaces also play an important role in documents that rely on an XSD schema for validation. If this sounds confusing, I think a quick explanation will clear things up. In order to identify the physical schema document for a document, you must use a special attribute and assign the location of the schema document to it. There are two attributes you can use to accomplish this task:

- `schemaLocation`—Locates a schema and its associated namespace
- `noNamespaceSchemaLocation`—Locates a schema with no namespace

These attributes are standard attributes that are located in a namespace named http://www.w3.org/2001/XMLSchema-instance. In order to properly reference either of these attributes, you must first explicitly declare the namespace in which they are
located. It is standard to use the xsi prefix for this namespace, as the following attribute assignment shows:

```xml
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

With this namespace declared, you can now use one of the schema location attributes to reference the physical schema document. Following is an example of how this task is carried out for the training log example document, which is based on the etml.xsd schema:

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

In this example the `noNamespaceSchemaLocation` attribute is used because you don’t care about associating the ETML schema with a namespace. If, however, you wanted to associate it with a namespace, you would use the `schemaLocation` attribute instead:

```xml
<trainlog
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.michaelmorrison.com/ns/etml etml.xsd">
```

Notice in the `schemaLocation` attribute that two pieces of information are provided: the namespace for the schema and the location of the schema document. The `schemaLocation` attribute is useful whenever you are working with a schema and you want to associate it with a namespace. It’s important to understand that this sample code doesn’t actually establish a schema prefix for the ETML document. Instead, it simply establishes that the etml.xsd schema document is associated with the ETML namespace. To establish prefix for the ETML tags and attributes, you must declare the ETML namespace, as shown in this code:

```xml
<trainlog
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:etml="http://www.michaelmorrison.com/ns/etml"
  xsi:schemaLocation="http://www.michaelmorrison.com/ns/etml etml.xsd">
```

Now the prefix `etml` can be used to reference tags and attributes as part of the ETML namespace, as in `etml:distance`.

**A Complete XML Schema Example**

You’ve covered an awful lot of territory in this hour and hopefully have a pretty good understanding of the XSD language and how it is used to create XSD schemas. To help pull together everything that you’ve learned, it might be helpful for you to see a complete example. If you recall, in Hour 3 you constructed a DTD for a sports
HOUR 7: Using XML Schema

training markup language known as ETML. Listing 7.1 contains the XSD equivalent for this markup language, which puts to use many of the XSD construction techniques you've learned about throughout this hour.

LISTING 7.1 The etml.xsd XSD Schema Used to Validate ETML Documents

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="trainlog">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="session" type="sessionType" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="sessionType">
    <xsd:sequence>
      <xsd:element name="duration" type="xsd:duration"/>
      <xsd:element name="distance" type="distanceType"/>
      <xsd:element name="location" type="xsd:string"/>
      <xsd:element name="comments" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="date" type="xsd:date" use="required"/>
    <xsd:attribute name="type" type="typeType" use="required"/>
    <xsd:attribute name="heartrate" type="xsd:positiveInteger"/>
  </xsd:complexType>

  <xsd:complexType name="distanceType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="units" type="unitsType" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:simpleType name="typeType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="running"/>
      <xsd:enumeration value="swimming"/>
      <xsd:enumeration value="cycling"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="unitsType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="miles"/>
      <xsd:enumeration value="kilometers"/>
      <xsd:enumeration value="laps"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```
Admittedly, this is considerably more code than the ETML DTD that you saw in Hour 3. However, you have to consider the fact that XSDs provide a more exacting approach to data modeling by incorporating rich data types. A quick study of the XSD code for ETML reveals that this schema does a much better job of modeling ETML data than its DTD counterpart. This is primarily due to the data typing features of XSD. Additionally, because XSD is an XML-based language, the code should be a little more familiar to you than the more cryptic code used in DTDs.

The `trainlog` element is described first in the XSD as containing a sequence of `session` elements (lines 4–11). The `sessionType` data type is created to represent `session` elements (line 13) and contains child elements that store the duration, distance, location, and comments for a training session (lines 15–18). The `sessionType` data type also includes several attributes that store the date, type, and heart rate for the training session (lines 21–23). The remaining `distanceType` (line 26), `typeType` (line 34), and `unitsType` (line 42) data types model the remaining content in ETML documents.

Of course, no schema would be complete without an example XML document that puts it through its paces. Listing 7.2 contains the training log document, modified slightly to accommodate the needs of the XSD schema.

LISTING 7.2 The Training Log Example ETML Document

```xml
1: <?xml version="1.0"?>
2:
3: <trainlog
4:   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5:   xsi:noNamespaceSchemaLocation="etml.xsd">
6:   <session date="2005-11-19" type="running" heartrate="158">
7:     <duration>PT45M</duration>
8:     <distance units="miles">5.5</distance>
9:     <location>Warner Park</location>
10:     <comments>Mid-morning run, a little winded throughout.</comments>
11:   </session>
12:
13:   <session date="2005-11-21" type="cycling" heartrate="153">
14:     <duration>PT2H30M</duration>
15:     <distance units="miles">37.0</distance>
16:     <location>Natchez Trace Parkway</location>
17:     <comments>Hilly ride, felt strong as an ox.</comments>
18:   </session>
19:
20:   <session date="2005-11-24" type="running" heartrate="156">
21:     <duration>PT1H30M</duration>
22:     <distance units="miles">8.5</distance>
23:     <location>Warner Park</location>
24:     <comments>Afternoon run, felt reasonably strong.</comments>
25:   </session>
26: </trainlog>
```

Other than including the standard `noNamespaceSchemaLocation` attribute to identify the XSD schema document (line 5), the changes to the training log document have to
do with the stricter data typing features of XSDs. For example, the date attributes and
duration elements conform to the xsd:date and xsd:duration simple types (lines 6,
7, 14, 15, 21, and 22). Beyond those changes, the document is the same as the one
you saw in Hour 3 with the DTD version of the ETML schema. This version of the doc-
ument, however, is considered valid with respect to the ETML XSD, whereas the previ-
ous version is considered valid with respect to the ETML DTD.

**RELAX NG and the Future of XML Schema**

This hour has painstakingly led you down the path of crafting schemas as XSDs as opposed to using the older DTD technology. Yet another schema technology has emerged that is worth addressing before sending you on your way with XML Schema. I'm referring to RELAX NG (pronounced “relaxing”), which is a schema technology with many of the benefits of XML Schema minus the verbose coding required of XSDs. You can think of RELAX NG as somewhat of a hybrid schema technology that falls somewhere between DTDs and XSDs. RELAX NG schemas represent a dramatic improvement over DTDs in terms of power and flexibility, yet they can be expressed in a compact format that requires less coding complexity than XML Schema. As an example of the difference between RELAX NG and XML Schema, the RELAX NG lan-
guage consists of 28 elements, whereas XML Schema consists of 42 elements.

So have you wasted an hour of your life learning about an overly complex XML schema technology? The answer is a resounding no. XML Schema is still the reign-
ing successor to DTDs, and enjoys considerably more widespread support than RELAX NG. However, given the shorter learning curve and simpler syntax, I would be remiss if I didn’t at least mention that RELAX NG is another viable option in terms of developing your own schemas. In fact, RELAX NG allows you to do a few things that aren't even possible in XML Schema. But like I said, RELAX NG is a relatively new technology that has yet to establish itself as an industry-wide XML schema technology. Although I certainly encourage you to learn more about RELAX NG, it’s important for you to know how to create schemas as XSDs for the immediate future.

It’s tough to speculate about the future of XML Schema and whether or not RELAX NG stands a realistic chance of upending it as the de facto standard for expressing XML schemas. For now, your best bet is to become fluent in XML Schema while pos-
sibly starting to get acquainted with RELAX NG if you have a desire to look forward and explore other schema options. The good news is that you’ve now learned about two different schema technologies, so if RELAX NG ever catches on in a big way you shouldn’t have too much trouble learning a third technology. To find out more about RELAX NG, visit the RELAX NG web site at http://www.relaxng.org/.
Summary

Although DTDs certainly represent a technology that is sufficient for modeling XML document data, XML Schema provides a much more advanced alternative that is rapidly replacing DTDs for defining XML schemas. XSD schemas are constructed using the XSD markup language, which includes elements and attributes for describing the structure of custom XML-based markup languages. This means that you create XSD schema documents in the same manner as you create any other XML document, which makes XSD schemas immediately more accessible to XML developers than DTDs.

This hour explored the inner workings of XSD schemas and taught you the fundamental skills necessary to design and create them. After explaining the different types of XSD data and how to use each of them, the hour showed you a complete example schema created in XSD. Admittedly, XSD is a fairly complex topic that is a bit tough to absorb all at once. Much of the complexity can be eliminated if you rely on a schema tool such as those mentioned in this lesson.

Q&A

Q. Are there any simple types other than the ones mentioned in this hour?
A. Yes. In addition to the simple types you learned about in this hour, there are a few other types such as binary and uriReference that I didn’t mention. These types were left out of the lesson primarily for the sake of brevity, as it is difficult to cover the entirety of the XSD language in a single hour. Besides, the aim of this hour is not to make you an XSD expert but to give you the essential knowledge required to design and create XSD schemas.

Q. How do I find out more about regular expressions and how they are used to create patterned types?
A. You can learn an immense amount about regular expressions from the book Sams Teach Yourself Regular Expressions in 10 Minutes, by Ben Forta. Or check out Stephen Ramsay’s online regular expression tutorial, which is located at http://etext.lib.virginia.edu/services/helpsheets/unix/regex.html. This tutorial is hosted by the University of Virginia’s Electronic Text Center.

Q. I still don’t quite understand the distinction between local and global components in an XSD schema. What gives?
A. Elements and attributes are considered global if they are declared directly below the xsd:schema element. If they are declared anywhere else, they are considered local. Global elements and attributes are used differently in that they must be referenced in order to factor into a schema. Contrast this with local elements and attributes, which are automatically referenced simply by virtue of being local.
HOUR 7: Using XML Schema

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 file extension is used to identify XSD schema documents?
2. What two primary attributes are used with the xsd:element element to describe an element in an XSD schema?
3. What simple type would you use to represent a recurring monthly bill in an XSD schema?
4. How do you control how many times an element may appear within an XSD schema?

Quiz Answers

1. XSD schema documents are stored with a file extension of .xsd.
2. The two primary attributes used with the xsd:element element to describe an element in an XSD schema are name and type.
3. Both the xsd:recurringDate and xsd:recurringDay types are sufficient to store the date of a recurring monthly bill. The xsd:recurringDate type would be better if you cared about knowing the month of the bill, whereas the xsd:recurringDay type would work fine if you were interested only in the day of the month.
4. The xsd:minOccurs and xsd:maxOccurs attributes allow you to control how many times an element appears within an XSD schema.

Exercises

1. Using the pets.xml document from Hour 1, create a pets.xsd schema document that uses XSD to describe a virtual pet markup language.
2. Modify the pets.xml document so that it can be validated against the new pets.xsd schema document.