

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Chapter 4. XML

We've described the general characteristics of SOA and now approach the practical details. At the core of SOA is *Extensible Markup Language (XML)*, a set of widely accepted rules for organizing data in a text format. In particular, each message exchanged with a Web service is in a format that conforms to those rules.

XML is described at the Web site of the World Wide Web Consortium (W3C), the standard's sponsoring organization. To give you a sense of what XML provides, we occasionally quote from the W3C recommendation *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, which is available at <http://www.w3.org/TR>.

We omit many complexities that are described in the specification, as well as on the Web and in textbooks. A user interface will stand between you and those documents anyway, so your development work may not require you to access XML documents a lot.

But don't be fooled. The user interface lets you work quickly and reduces the tedium of many tasks, but it doesn't separate you completely from the underlying technology. If you understand SOA at the level of the XML-based files, you can

- respond to error conditions with greater skill
- understand the implications of user-interface settings that pertain to XML
- review or even change the XML files directly ("Oh, that's what's happening!")
- avoid dependence on tools provided by a particular vendor

Excellence often requires you to know details that are below the surface, and in any case, broccoli is good for you.

Introduction to XML

Listing 4.1 shows an example of an XML document that holds details on a request for insurance coverage.

Listing 4.1. Sample XML document

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- policy request -->
<Insured CustomerID="5">
  <CarPolicy PolicyType="Auto">
    <Vehicle Category="Sedan">
      <Make>Honda</Make>
      <Model>Accord</Model>
    </Vehicle>
    <Vehicle Category="Sport">
      <Make>Ford</Make>
      <Model>Mustang</Model>
    </Vehicle>
  </CarPolicy>
  <CarPolicy PolicyType="Antique">
    <Vehicle Category="Sport">
      <Make>Triumph</Make>
      <Model>Spitfire</Model>
    </Vehicle>
  </CarPolicy>
</Insured>
```

When you create an XML document, you use a *vocabulary*, which is a set of terms that are organized in a way that reflects the data used in your business. Each tag name (such as `Vehicle`) is a reserved word only in the narrow context of your application, company, or industry.

Data that conforms to an XML format is handled by an *XML processor*, also called an *XML engine*. The processor is software that

- is invoked by other software
- identifies units of information in the XML-formatted input
- provides that information for some purpose such as
 - to display the data, as occurs when you launch an XML file with a recent browser
 - to convert the data to another format, as occurs when an SOA runtime product invokes a SOAP engine
 - to construct a runtime service implementation, as occurs when you invoke a Business Process Execution Language (BPEL) engine

XML benefits the business developer in several ways:

- XML allows development of specialized vocabularies that reduce the cost of business-to-business interaction. Widely used vocabularies include ACORD (for the insurance industry) and IFX (for the financial-services industry).
- You can process data with ease because of the many software products that support XML. A product might provide text searches and editing capability or might transform text from one XML vocabulary into another.
- XML provides mechanisms for validating the data, including technologies such as Document Type Definition (DTD), XML Schema Definition (XSD), RELAX NG, and Schematron. This book focuses on XSD because in many cases an SOA runtime product uses that technology to validate transmitted data.
- An XML document is clear to the human reader. The content is simple text rather than a stream of binary characters and is relatively *self-documenting*, showing data values as well as relationships. In our example, a customer is requesting two policies, and each policy includes details on specific cars.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

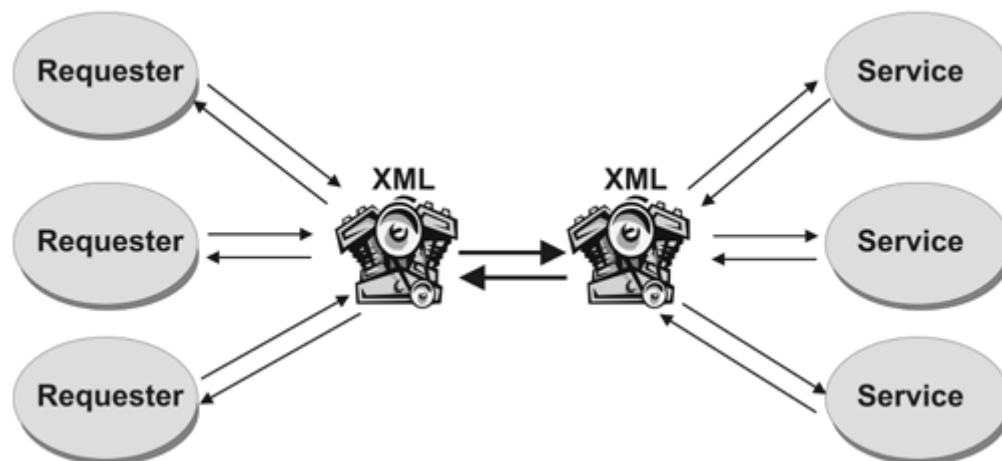
No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

XML As Used in SOA

XML makes possible the implementations that were anticipated during the years when SOA was just a subject for academic review:

- As you develop a service
 - you're likely to use XML to describe the service interface
 - you may use XML to create the logic itself (as you'll see in [Chapter 7](#), when we describe BPEL)
 - in relation to Web services, you may access the transmitted data using technologies such as XPath, which require knowledge of the XML-based message format
- After you develop a service, you may use XML to store the service contract in a registry.
- At configuration time, you may use XML to configure the service, even to the extent of assigning different values to variables inside the service implementation. That degree of configurability is shown in [Chapter 9](#), when we describe Service Component Architecture.
- At Web-service run time, the transmitted data is simply text in an XML format, and XML engines convert the data at the transmission endpoints, as [Figure 4.1](#) illustrates.

Figure 4.1. XML engines at Web-service run time



At a transmitting endpoint, an XML processor converts a message into the format required for transmission. At the receiving endpoint, another XML processor converts the message from the transmission medium into the local format.

In the absence of a universal format like that provided by XML, an SOA runtime product would need to act in one of two ways to support Web services. The product could handle data conversions as needed to transform data from each endpoint-specific format directly to every other one. With this approach, the addition of a new kind of endpoint would require significant updates to the runtime product. Alternatively, the product would restrict the computer languages used in the participating endpoints.

XML has a cost. Use of text rather than a binary format slows processing and requires more disk space. An SOA runtime product can reduce the cost of transmission by compressing each XML-based message, but time is required for data conversion in any case.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Structure of an XML Document

Let's look again at the XML document shown earlier ([Listing 4.1](#)).

Our document is composed of a set of tagged constructs, with each tag bounded by angle brackets (< and >). The first construct is called the *XML declaration*. This line identifies the XML version, specifies an *encoding* (a subject far afield from most business development), and includes an initial and final question mark (?).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

If the XML declaration is missing, the document must conform to XML 1.0. For details on encoding, see the Web site <http://skew.org/xml/tutorial>.

Our document also includes a *comment*, which is text that an XML processor can ignore and whose purpose is usually to clarify (to a human reader) some aspect of the file. A comment is characterized by specific initial and final characters, as shown here.

```
<!-- policy request -->
```

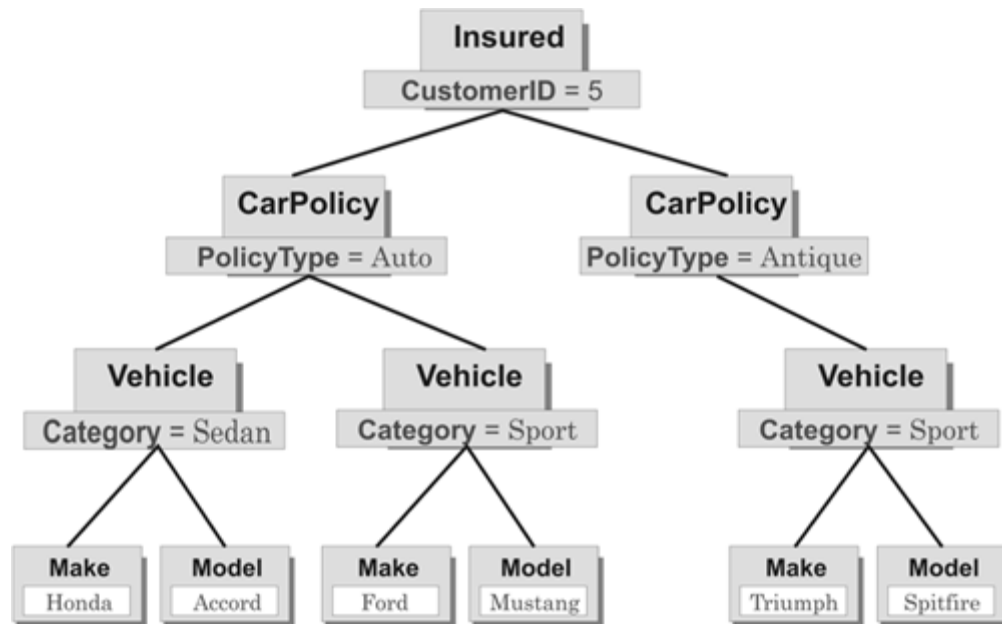
Most important (as noted in the XML specification), the document includes "one or more elements" that represent the data of interest. In most cases, an element is "delimited by start-tags and end-tags"; as shown here, the end-tag precedes the element name with a virgule, or forward slash (/).

```
<Insured>
...
</Insured>
```

Each XML document must have a root element (such as [Insured](#)), which includes a set of other elements in a tree structure. [Figure 4.2](#) shows the tree structure for the document under review.

Figure 4.2. XML tree structure

[\[View full size image\]](#)



An important characteristic of XML is that each superior, or *parent*, element includes its immediately subordinate, or *child*, elements completely. For example, a parent element cannot include a start-tag of a child element without including the corresponding end-tag. Child elements can themselves have children, and we use the phrase *descendants* to refer to all elements that are within the start and end tags of a parent element, at any level of nesting.

Most elements include *content*: child elements, a literal value (as is in the `Make` element — `<Make>Honda</Make>`), or both, as in the following example of *mixed content*:

```

<Vehicle>Cool!
  <Make>Triumph</Make>
  <Model>Spitfire</Model>
</Vehicle>

```

An element also can include *attributes*, which are name-value pairs that provide additional information associated with the element. You specify attributes in the start-tag of the element. Each attribute is composed of a name, an equal sign (=), and a single- or double-quoted string. To separate one attribute from the next, you use white space (carriage returns, spaces, or tabs). Here's an example of an element with two attributes.

```

<Insured CustomerID="5" Status="In Review">

```

Although attributes are part of the element, they are not considered content. This distinction has a practical effect when you work with XPath, as described in [Chapter 6](#).

Note that each identifier in the XML document is case-sensitive. An element named `vehicle` is distinct from one named `VEHICLE`.

You can organize your data in various ways, specifying elements in some cases and attributes in others. Within the `Insured` element, for example, we might add an `Options` element to describe the insurance coverage in effect when a car is disabled:

```

<Options>
  <TemporaryRental MaximumDays="10"/>
  <Towing/>
</Options>

```

The `TemporaryRental` element tells whether the insurance company pays for a customer's vehicle rental; the `MaximumDays` attribute indicates the maximum number of days that are covered; and the `Towing` element indicates whether the insurance company pays for use of a towing service.

An element (such as `TemporaryRental`) with no content is said to be *empty*. It can have a start and end tag or (as shown) can have a single tag with an ending virgule.

A lack of content in an element can mean that the data is elsewhere. A value for the `Towing` element might be assigned as a default in the XML Schema definition (as described later) or in the software that accesses the XML processor.

Later in this book, we mention *processing instructions (PIs)*, which are XML statements that are used by the XML processor or by the software that invokes the processor. Here's an example.

```
<? HandleThis how="somehow" ?>
```

The PI includes a question mark within each angle bracket and specifies a *PI target*, which is a name that identifies the instruction. The PI's content is the set of parameters and related values. In this example, the PI target is `HandleThis`, the parameter is `how`, and the value of the parameter is `somehow`.

Last, you may want the XML processor to ignore characters that are otherwise meaningful in a technical sense. For example, content that appears to be an element start-tag (such as `<You&Me>`) might be a string to be printed. To hide such content, place it in a character-data (*CDATA*) section, as shown here.

```
<![CDATA[ <You&Me> ]]>
```

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Namespaces

We now explain *namespaces*, which are identifiers that place element and attribute names in categories. If you're reading only to get a sense of SOA, you can skim this section, but you may have reasons to read straight through:

- Knowledge of namespaces can help you avoid feeling intimidated by the seemingly complex data at the top of most XML documents.
- A developer needs to know the subject at least occasionally.
- The material on XML Schema is best understood by someone with prior knowledge of namespaces.
- Later chapters in this book refer to namespaces.

Purpose of Namespaces

Namespaces are categories for ensuring that names in an XML document are unique in that document. With namespaces, a developer can combine different XML vocabularies without fear that identical names in different vocabularies will be handled in the same way. An XML processor, for example, might use a namespace to distinguish the rules for validating a name that is used multiple times for different purposes. Similarly, code that you write to review a complex XML-based message might need to identify the namespace associated with a name.

Before we tell you how to specify a namespace, consider an example that includes two instances of the element name `Title`. One instance refers to a book, and one refers to a person.

```
<TitleTypes>
  <Title xmlns="BookTitleNamespace">
    <Name>Oliver Twist</Name>
  </Title>
  <Title>Duke of Windsor</Title>
</TitleTypes>
```

If an element name is in a namespace, the XML processor interprets the name in a way that includes the namespace identifier. We might express the first instance of the name `Title` as follows:

```
{BookTitleNamespace}Title
```

An element or attribute name can be outside any namespace, as is true of the second instance of `Title`. In the absence of a namespace, the XML processor interprets the name without referring to a namespace identifier. We might express that second instance simply as `Title`.

The implication of a namespace depends solely on the software that processes a given XML document. An XML processor might use a namespace identifier to determine which of several versions of an XML vocabulary is being used or to determine which XML Schema validates a given element.

Namespace Identifiers

A namespace is usually a character string that is formatted as if to refer to a Web-based resource such as a Web site

or file. But a namespace may be any Universal Resource Identifier (URI) — that is, any unique value that conforms to the URI rules defined by the Internet Engineering Task Force (IETF). Here are some examples of namespace URIs:

```
http://www.w3.org/2001/XMLSchema
f81d4fae-7jan-11d0-a765-00a0c91e6bf6
ISBN:0452010624
```

Later, we show how to declare a namespace.

One kind of URI is a Uniform Resource Locator (URL), which can be used as a namespace identifier. The specific URL may refer to a Web site that describes the purpose of the namespace or of the XML document. In most cases, however, the XML processor does *not* access a Web site, and the URI often does not refer to a Web site at all.

The namespace URI is just an identifier, with no meaning other than to place a set of names in a category. Often, your organization's Internet domain name (such as www.w3.org) is appropriately included to ensure uniqueness.

Namespace Qualification and Declarations

An XML-based name is said to be *namespace-qualified* if the XML associates the name with a namespace. A name that is not associated with a namespace is not in a default namespace; instead, the XML processor interprets the name as being outside any namespace.

A name can be namespace-qualified in either of two ways. Each way requires you to first declare a namespace by using the identifier `xmlns`, which is an abbreviation for *XML namespace*.

You declare a *non-default namespace* by following `xmlns` with a colon (:) and a second identifier of your choice (for instance, `target`), as in the following start-tag.

```
<Insured
xmlns:target="http://www.IBM.com/HighlightInsurance">
```

Later in the XML document, you can use that second identifier as a prefix to indicate that an element or attribute name (such as `Model`) is in a non-default namespace.

```
<target:Model>Spitfire</target:Model>
```

You declare a *default namespace* by specifying `xmlns` without a subsequent colon, as in the following start-tag.

```
<Insured xmlns="http://www.IBM.com/software">
```

In the absence of other namespace syntax, the element name (`Insured`, in this case) is in the default namespace, as is the name of each element descended from that element.

The start-tag of the XML root element often includes one or more namespace declarations, as shown here.

```
<wsdl:definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.transunion.com/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

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

Namespace Rules

Before we review an example, consider the following rules:

- Any namespace declaration is *in scope* (that is, available) in the element in which the declaration appears and in the element's descendant elements.
- An element name that is in the scope of a default namespace and is not prefixed is in the default namespace.
- An attribute name can never be in a default namespace:
 - If the attribute name has a prefix, the name is in the namespace referenced by that prefix.
 - If the attribute name is not prefixed, the name is outside any namespace. You can never use a default namespace to reference an attribute.
- A default namespace can be overridden for a given element and for all the descendant elements. In the following example, the element name `CarPolicy` is in the `PolicySpace` namespace; and the element names `Vehicle`, `Make`, and `Model` are in the `VehicleSpace` namespace.

```
<CarPolicy xmlns="PolicySpace" PolicyType="Auto">
  <Vehicle xmlns="VehicleSpace" Category="Sedan">
    <Make>Honda</Make>
    <Model>Accord</Model>
  </Vehicle>
</CarPolicy>
```

- A default namespace can be removed from scope for a given element and for all the descendant elements. In the following example, the names `Vehicle`, `Make`, and `Model` are not in any namespace because the default namespace is set to an empty string ("").

```
<Vehicle xmlns="" Category="Sedan">
  <Make>Honda</Make>
  <Model>Accord</Model>
</Vehicle>
```

- Multiple non-default namespaces can be available for a given element and for all descendant elements, and those namespaces are also available for the attributes in those elements. In the following example, the element names `Order` and `Customer` are in `OrderSpace`, while the element names `Company` and `Buyer` are in `CustSpace`, as is the attribute `Number`.

```
<ord:Order xmlns:ord="OrderSpace">
  <ord:Customer xmlns:cust="CustSpace">
    <cust:Company cust:Number="123">
      ABC
    </cust:Company>
    <cust:Buyer>
      Smith
    </cust:Buyer>
  </ord:Customer>
</ord:Order>
```

Namespace Example and Terminology

Consider the XML shown in [Listing 4.2](#).

Listing 4.2. Sample XML with namespaces

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- an order -->
<Order xmlns="http://www.ibm.com/software"
  xmlns:prod="http://www.ibm.com/next">
  <Customer CustID="IBM" Code="1234">
    <prod:Product prod:ID="0123">
      <prod:Type>Blackboard</prod:Type>
      <prod:Quantity>2</prod:Quantity>
    </prod:Product>
    <prod:Product prod:ID="0456">
      <prod:Type>Whiteboard</prod:Type>
      <prod:Quantity>3</prod:Quantity>
    </prod:Product>
    <prod:Product prod:ID="0789"/>
  </Customer>
</Order>
```

In this example two element names, `Order` and `Customer`, are in the default namespace, <http://www.ibm.com/software>. The element names `Product`, `Type`, and `Quantity` are in the namespace <http://www.ibm.com/next>. Given the prefix in front of the attribute name `ID` in the `Product` element, the name `ID` is also in that second namespace. The names of the `Customer` element's `CustID` and `Code` attributes are not in a namespace.

The following terminology is in use:

- The name of a namespace (such as <http://www.ibm.com/software>) is the *namespace URI*.
- The element or attribute name can include a prefix and a colon (as in `prod:Quantity`). A name in that form is called a *qualified name*, or *QName*, and the identifier that follows the colon is called a *local name*. If a prefix is not in use, neither is the colon, and the QName and local name are identical.
- An XML identifier (such as a local name) that has no colon is sometimes called an *NCName*. (The *NC* comes from the phrase *no colon*.)

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

XML Schema

The next sections introduce XML Schema definitions, or XSDs. You use an XML Schema to describe the values permitted in an XML document and the relationships among the document's elements. For details about the XML Schema definition language, see the Web site <http://www.w3.org/TR/xmlschema-0>.

Data Type

To understand an XSD, you first need to understand the notion of *data type*. A data type is an identifier that specifies a set of values along with operations able to act on those values. In most computer languages, for example, an integer can include digits and can be mathematically combined with other numbers.

In general, we use data types to validate user input, to avoid runtime errors, and to increase the efficiency of runtime code. Data types also allow a meaningful categorization of data, so that, for instance:

- Each use of an employee ID is based on a data type named Employee-ID.
- Each employee record has the same hierarchy, with data types such as Employee-ID, Name, and Age.
- A known set of validations is in use when a data type such as Name (always containing Last-Name and First-Name) is used in different ways, whether for an employee record or a customer-contact record.

Data types are described in various ways. With some simplification, we can characterize XML Schema types as fitting into one of four basic categories: primitive, derived, simple, and complex.

A *primitive type* is a data type that does not include another and is not derived from another. Examples include Boolean, decimal, and string.

A *derived type* is based on a primitive type and is provided by XML Schema. The type nmtoken, for example, is a subset of a string, with no spaces anywhere.

A *simple type* can be a primitive type or a derived type, but it also can be derived from another simple type by a developer. We want to emphasize a particular use of types in business development, so we'll use the phrase *simple type* to refer only to a developer-derived simple type. If you read XML specifications, however, be aware that the phrase "simple type" can include primitive and derived types.

A simple type retains every operation of the type on which the simple type is based but adds data restrictions that reflect business rules. Given a base type of string, for example, you might create a simple type called Employee-ID and allow only the letters A through J followed by five digits. The meaningfulness of the name Employee-ID helps developers and business analysts to think and communicate clearly. In addition, type checking at different points in the development cycle can ensure that all employee IDs conform to the restrictions specified in the definition of the type.

A *complex type* is a data type that is composed of other data types. An example might be called Employee-Record, which includes a simple type called Employee-ID, a complex type called Name, an integer called Age, and so on. Any data type in the composition may be primitive, derived, simple, or complex. When you create a complex type, you give names even to the primitive and derived types that are included in that complex type. As always, meaningful names are helpful.

A type doesn't contain values but identifies what values are possible. An employee-record type, for example, doesn't refer to a specific employee, but expresses the format needed to describe an employee. A *variable*, in contrast, contains data that fulfills the type requirements. A variable of an employee-record type, for example, can hold the employee ID, name, and age of a specific employee. The variable is said to be an *instance* of the type.

As we stated earlier, a data type allows specific values and operations. A string, for example, can be concatenated with a string, but not with an integer. In relation to a given value, however, you can sometimes ignore or override those restrictions. You can concatenate the string "The year was " with the integer 2000, for example, but only if the integer is first converted to a string. Conversions may happen automatically or by the developer's *cast*, which is a directive that converts a value from one type to another.

Computer languages are sometimes categorized as either *weakly typed*, to the extent that data-type conversions happen automatically, or *strongly typed*, to the extent that the conversions happen only as a result of an explicit cast. A weakly typed language requires less discipline at development time; but a strongly typed language allows for faster processing at run time because less type checking is required then.

Purpose of an XML Schema

You author an XML Schema (sometimes called an XML Schema definition, or XSD) to describe the values that are allowable in each element and attribute in an XML document and to characterize the relationship of one element to another. In short, an XSD describes an XML vocabulary.

In SOA, the XSD has two primary purposes:

- to tell SOA-related tools (often in an integrated development environment) how to construct the code that operates "behind the scenes" to make possible a data exchange between the requester and a specific service
- to assign validation rules that restrict the kind of data accepted by an endpoint at run time

The XSDs for a particular service must be available at each endpoint of a transmission. In the case of a Web service, an XSD is often embedded in a Web Services Description Language (WSDL) file, as described in [Chapter 5](#). The XSD usually is not transferred with the business data.

Structure of an XML Schema

An XML Schema specifies a *content model*, which is an allowable set of *content* (names and types) for elements, along with equivalent details on the attributes of each element. An XML stream that conforms to the rules established in the XML Schema is called an *instance document*, and we can refer to the elements and attributes in that stream as *instance elements* and *instance attributes*, respectively.

As the example in [Listing 4.3](#) shows, an XML Schema is itself written in XML.

Listing 4.3. Sample XML Schema definition

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:target="http://www.IBM.com/HighlightInsurance"
  targetNamespace="http://www.IBM.com/HighlightInsurance"
  elementFormDefault="unqualified">
  <annotation>
    <documentation xml:lang="en">
      An example XML Schema Definition (XSD).
    </documentation>
  </annotation>
  <element name="Insured">
    <complexType>
      <choice minOccurs="1" maxOccurs="unbounded">
        <element name="CarPolicy">
          <complexType>
            <choice maxOccurs="unbounded">
              <element name="Vehicle"
                type="target:VehicleType"/>
              <element name="Driver"
                type="target:DriverType"/>
            </choice>
            <attribute name="PolicyType"
              type="string" default="Auto"/>
          </complexType>
        </element>
        <element name="HomePolicy" type="target:HomePolicyType"/>
      </choice>
      <attribute name="CustomerID" type="string" use="required"/>
    </complexType>
  </element>

  <complexType name="VehicleType">
    <group ref="target:VehicleGroup"/>
    <attribute name="VIN" use="required">
      <simpleType>
        <restriction base="string">
```

```

        <minLength value="4"/>
        <maxLength value="26"/>
    </restriction>
</simpleType>
</attribute>
<attribute name="Category" type="string"/>
</complexType>

<group name="VehicleGroup">
    <sequence>
        <element name="Make" type="string"/>
        <element name="Model" type="string"/>
    </sequence>
</group>

<!-- The complex types DriverType and HomePolicyType are not shown -->
</schema>

```

Global and Local Types

Any data types that are immediate children of the `schema` element are *global*, which means you can use the type name when assigning characteristics to elements or attributes anywhere in the Schema. An example of a global type is `VehicleType`.

```

<schema>
    <complexType name="VehicleType">
        .
        .
    </complexType>
</schema>

```

Other data types you declare are *local*, which means that they affect only their parent, in which case a type name is unnecessary.

```

<element name="Insured">
    <complexType>
        .
        .
    </complexType>

```

A data type that has no name is sometimes called an *anonymous type*.

Simple and Complex Types

A global or local type can be simple or complex. A *simple type* is a data type that indicates the allowable text content for an element or attribute, as in the following lines.

```

<attribute name="VIN" use="required">
    <simpleType>
        <restriction base="string">
            <minLength value="4"/>
            <maxLength value="26"/>
        </restriction>
    </simpleType>
</attribute>

```

Here, the subordinate `restriction` element includes two kinds of details:

- A *base type*, which is the type from which the simple type will be derived. The base type is a global simple type, a derived type, or a primitive type.
- A set of *facets* (that is, characteristics) and related values. In this case, a restriction is in place for a minimum number of characters (`minLength`) and a maximum number of characters (`maxLength`).

Instead of a `restriction` element, we could have used a `list` element (to allow content to be a series of values of the base type) or a `union` element (to allow content to represent any of several base types).

A *complex type* is a data type that includes elements, attributes, or both, as in the following lines.

```
<element name="CarPolicy">
  <complexType>
    <choice maxOccurs="unbounded">
      <element name="Vehicle" type="target:VehicleType"/>
      <element name="Driver" type="string"/>
    </choice>
    <attribute name="PolicyType"
              type="string" default="Auto"/>
  </complexType>
</element>
```

Here, the subordinate `choice` element shows that

- any number of child elements are valid, as indicated by the `maxOccurs` attribute setting
- a given child element can be called `Vehicle` (of type `VehicleType`) or `Driver` (a string)

Instance attributes are optional by default. `CarPolicy` has an optional `PolicyType` attribute, for example, and the default value of `PolicyType` is *Auto*.

`VehicleType` is another complex type, which includes a group of elements (type `VehicleGroup`, as described in the next section) and the attributes `VIN` and `Category`.

```
<complexType name="VehicleType">
  <group ref="target:VehicleGroup"/>
  <attribute name="VIN" use="required">
    .
  </attribute>
  <attribute name="Category" type="string"/>
</complexType>
```

Groups

A *group declaration* is essentially a data type that specifies a list of elements. An example of a group declaration is `VehicleGroup`.

```
<group name="VehicleGroup">
  <sequence>
    <element name="Make" type="string"/>
    <element name="Model" type="string"/>
  </sequence>
</group>
```

Instance elements are required unless the minimum-occurrence (`minOccurs`) attribute is set to 0. (The default value of `minOccurs` is 1.) The element `VehicleGroup`, for example, indicates that each child instance element (`Make` and

`Model`) is required.

Sequencing

A complex type or group might include a *sequencing element*:

- The `sequence` element means that the instance elements must be in the specified order.
- The `all` element means that the instance elements can be in any order.
- The `choice` element means that the instance document can include a subset of elements.

Let's look at two examples.

`Insured` (the root element of the XML instance document) must include at least one `CarPolicy` or `HomePolicy` instance element and can include any number of those elements in any combination or order.

```
<element name="Insured">
  <complexType>
    <choice minOccurs="1" maxOccurs="unbounded">
      <element name="CarPolicy">
        <complexType>
          .
          .
        </complexType>
      </element>
      <element name="HomePolicy"
        type="target:HomePolicyType"/>
    </choice>
    <attribute name="CustomerID" type="string"
      use="required"/>
  </complexType>
</element>
```

Incidentally, the `Insured` instance element also must include a `CustomerID` attribute, as indicated by the value of `use`.

As shown in `VehicleGroup`, the `Make` instance element must precede `Model`.

```
<group name="VehicleGroup">
  <sequence>
    <element name="Make" type="string"/>
    <element name="Model" type="string"/>
  </sequence>
</group>
```

Simple and Complex Content

You can derive a complex type from an existing (base) type. The derived type will have characteristics of the base type

- with *extensions*, which are added attributes, content, or both
- with *restrictions*, which are exclusions of the existing attributes, content, or both

For example, you can use the `simpleContent` Schema element to add attributes to an instance element that has text content. Consider the `Options` element, which we described earlier.


```
<Options>
  <TemporaryRental MaximumDays="10"/>
  <Towing/>
</Options>
```

Here is a related XML Schema element.

```
<element name="Options">
  <all>
    <element name="TemporaryRental" type="TemporaryRentalType"/>
    <element name="Towing" type="boolean" default="true"/>
  </all>
</element>

<complexType name="TemporaryRentalType" default="true">
  <simpleContent>
    <extension base="boolean">
      <attribute name="MaximumDays"
        type="integer" default="10"/>
      <attribute name="MaxDollarPerDay"
        type="decimal" default="25.99"/>
    </extension>
  </simpleContent>
</complexType>
```

Within the `TemporaryRentalType` element, the `simpleContent` element specifies that any instance element based on that type has Boolean text (value *true* or *false*, with a default) and can include the optional attributes `MaximumDays` (which takes an integer value) and `MaxDollarPerDay` (which takes a decimal value). Each attribute value has a default, too.

You might use the element `simpleContent` to create a type that restricts aspects of an existing complex type, which itself has simple content. Here's an example.

```
<complexType name="PremiumRentalType" default="true">
  <simpleContent>
    <restriction base="TemporaryRentalType">
      <attribute name="MaximumDays" use="required"/>
      <attribute name="MaxDollarPerDay" use="prohibited"/>
    </extension>
  </simpleContent>
</complexType>
```

Given the new definition, you could allow an instance element (of whatever name) and base that element on the complex type `PremiumRentalType`. Only one attribute is valid, and it's required. The modified `MaxDollarPerDay` attribute is not valid in the instance element because, in the Schema attribute definition, the value of `use` is *prohibited*.

You also can use the `complexContent` Schema element

- to extend a complex type — for example, to add elements or attributes
- to restrict a complex type — for example, to remove elements or attributes, to change element characteristics such as the minimum number of occurrences, or to change attribute characteristics

Schemas and Namespaces

An XML processor uses namespace details to process an instance document, and the primary source of the namespace details can be either the instance document or the related Schema. Different SOA implementations

handle the issue differently, but in any case, settings in the Schema indicate which source to use:

- If the source is the Schema, less namespace information appears in the instance document. In this case, the instance document is easier to understand and maintain, especially when the Schema organization is complicated. Also, the Schema author has the flexibility to merge or split Schemas, with less chance that the reorganization will mean changes to the existing XML instance documents.
- If the source of namespace details is the XML instance document, that document shows namespace information to a wider audience, and the XML processor may run faster.

In the `schema` element of our sample XML Schema definition, we set the attribute `elementFormDefault` to *unqualified*, which means that most instance elements are not qualified with a namespace name. Instead, the Schema is the source of namespace details for every element other than the root element. The *unqualified* setting is the default.

The `schema` element includes other namespace details, too:

- The namespace <http://www.w3.org/2001/XMLSchema> is the default namespace, but this namespace is often associated with the prefix `xs` or `xsd`, as shown later in a WSDL file. That namespace refers to identifiers such as `attribute`, which is the name of an element in the XML Schema itself.
- The attribute `targetNamespace` specifies the target namespace, which is used as a category for each name you're adding. The target namespace includes the names of the elements and attributes that will be allowed in your XML instance document and includes the names of any types you create.
- As you can see, the name of the target namespace is also in a second namespace declaration. The purpose of that second declaration is to allow references to any data-type information that you (as the author of the XML Schema) specify in the Schema. The following lines, for example, reference the namespace that is identified by the prefix `target` and points to `VehicleType`, which is a data type in the Schema.

```
<element name="Vehicle"
         type="target:VehicleType"/>
```

Aside from the types you create, a set of XML Schema types is available in the default namespace (<http://www.w3.org/2001/XMLSchema>). For that reason, the use of `string` in the following declaration is valid in our example.

```
<element name="Make" type="string"/>
```

Last, the `documentation` element in our example includes the prefix `xml`, which refers to a namespace that is defined by the XML specification. The language of the `documentation` element is American English, as indicated by the following entry.

```
xml:lang="en-US"
```

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Data-type Reuse

To organize your Schemas in a way that benefits your company over time, store the reusable types in a set of Schema files that other Schemas can access. The types must be global. A Schema gains access to the reusable types by either of two declarations: `include` or `import`.

The `include` declaration is the preferred choice when the Schema being accessed either has no target namespace or has the same target namespace as the accessing Schema. The `import` declaration is necessary when the accessed Schema has a target namespace that differs from the target namespace of the accessing Schema.

User name:

Book: SOA for the Business Developer: Concepts, BPEL, and SCA

No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Instance Document

The Schema described earlier validates the instance document shown in [Listing 4.4](#).

Listing 4.4. Sample XML instance document

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<root:Insured
  xmlns:root="http://www.highlight.com/Insurance"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.ibm.com/HighlightInsurance Insurance.xsd"
  CustomerID="5">
  <CarPolicy PolicyType="Auto">
    <Vehicle VIN="A123">
      <Make>Honda</Make>
      <Model>Accord</Model>
    </Vehicle>
    <Vehicle VIN="A456">
      <Make>Ford</Make>
      <Model>Mustang</Model>
    </Vehicle>
  </CarPolicy>
  <CarPolicy PolicyType="Antique Auto">
    <Vehicle VIN="B321">
      <Make>Triumph</Make>
      <Model>Spitfire</Model>
    </Vehicle>
    <Vehicle VIN="B654">
      <Make>Buick</Make>
      <Model>Skylark</Model>
    </Vehicle>
    <Vehicle VIN="B987">
      <Make>Porsche</Make>
      <Model>Speedster</Model>
    </Vehicle>
  </CarPolicy>
</root:Insured>
```

We qualify `Insured` with the target namespace because that name is defined in a global element of the XML Schema. We cannot qualify the other names without causing a Schema validation error, because the Schema element `elementFormDefault` is set to *unqualified*.

After we declare a non-default namespace and use it to qualify `Insured`, we use the prefix `xsi` to specify an XML-related namespace and to set the attribute `schemaLocation`.

```
xsi:schemaLocation=
  "http://www.ibm.com/HighlightInsurance Insurance.xsd"
```

The setting of `schemaLocation` repeats the target namespace and tells the location of the XML Schema. The attribute setting is only a hint to the XML processor, which may be configured to use a different Schema.