Systems for Design and Implementation

2015-2016

Course 5

Contents

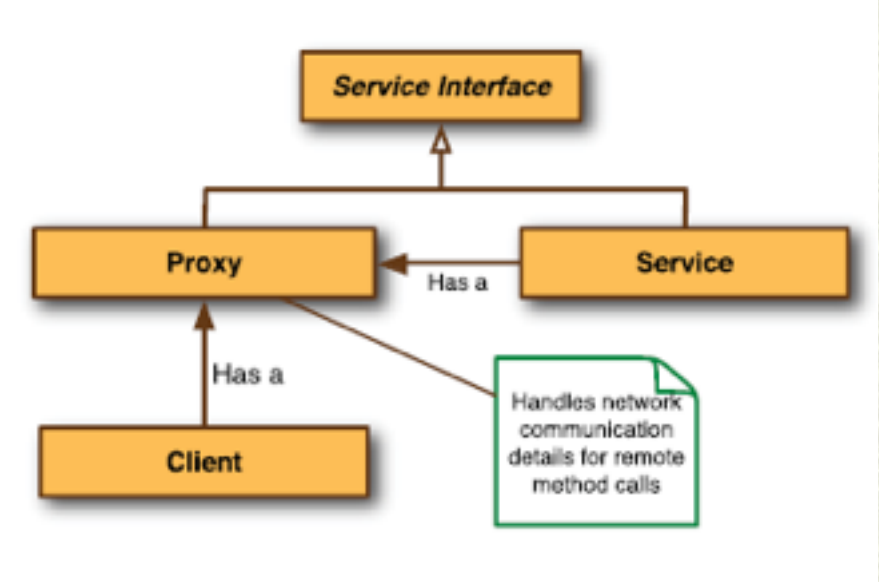
- Spring Remoting
- XML documents
- DTDs
- XML Schema
- C# API for XML processing

Spring Remoting

- Spring supports remoting for several different RPC models, including Remote Method Invocation (RMI), Caucho's Hessian and Burlap, and Spring's own HTTP invoker.
- In all models, services can be configured into the application as Spring-managed beans.
- This is accomplished using a proxy factory bean that enables you to wire remote services into properties of your other beans as if they were local objects.
- The client makes calls to the proxy as if the proxy were providing the service functionality.
- The proxy communicates with the remote service on behalf of the client. It handles the details of connecting and making remote calls to the remote service.

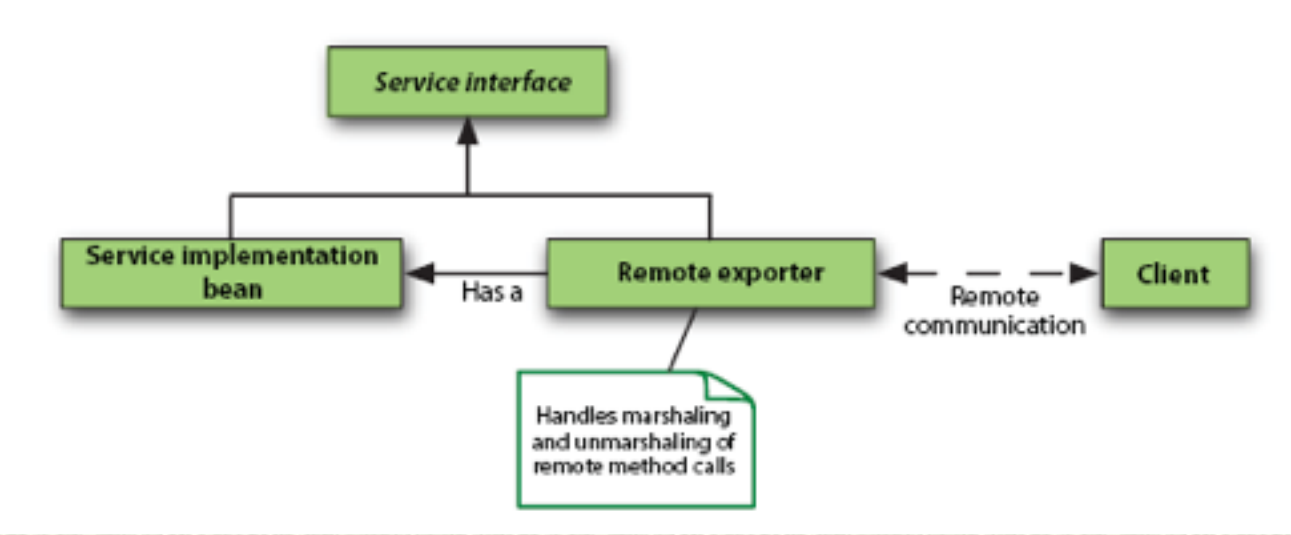
Spring Remoting

- In Spring, remote services are proxied so that they can be wired into client code as if they were any other Spring bean.



Spring Remoting

- Spring-managed beans can be exported as remote services using remote exporters.
- If the call to the remote service results in a `java.rmi.RemoteException`, the proxy handles that exception and rethrows it as an unchecked `RemoteAccessException`.
- Rethrowing a `RemoteAccessException` makes it optional for the client to handle the exception.



Spring Remoting

Step to publish a remote service using RMI:

1. Create the service interface to extend `java.rmi.Remote`.
2. Write the service implementation class with methods that throw `java.rmi.RemoteException`.
3. Run the RMI compiler (`rmic`) to produce client stub classes.
4. Start an RMI registry to host the services.
5. Register the service in the RMI registry.

`RemoteException` and `MalformedURLExceptions` are thrown very often, even though these exceptions usually indicate a fatal error that cannot be recovered from try-catch block.

Configuring an RMI Service in Spring

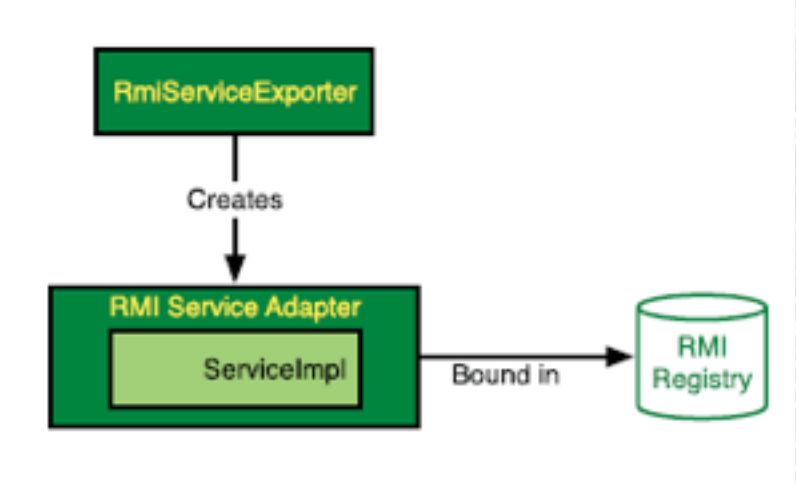
- Spring provides an easier way to publish RMI services.
- You just have to write a POJO that performs the functionality of the service. You do not have to write RMI-specific classes with methods that throw `RemoteException`.

```
public interface ITransformer {
    String transform(String text);
}
import java.util.Date;

public class TransformerImpl implements ITransformer{
    public String transform(String text) {
        System.out.println("Method called "+text);
        return text.toUpperCase()+(new Date());
    }
}
```

Configuring an RMI Service in Spring

- `RmiServiceExporter` exports any Spring-managed bean as an RMI service. It works by wrapping the bean in an adapter class. The adapter class is then bound to the RMI registry and proxies requests to the service class.



Configuring an RMI Service in Spring

```
//remote service configuration
//spring-server.xml
<bean id="transformerService"
     class="transformer.services.impl.TransformerImpl"/>

 <bean
     class="org.springframework.remoting.rmi.RmiServiceExporter">

   <property name="serviceName" value="TransformerService"/>
   <property name="service" ref="transformerService"/>
   <property name="serviceInterface"
     value="transformer.services.ITransformer"/>
   <property name="servicePort" value="1099"/>
 </bean>
```

Configuring an RMI Service in Spring

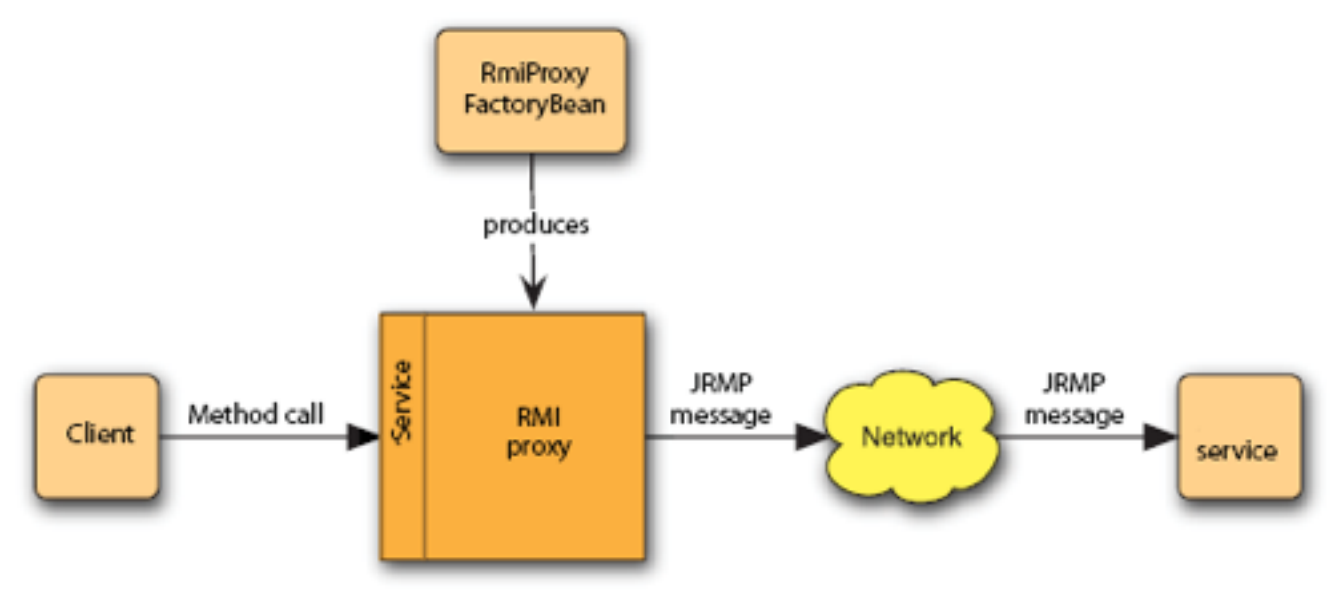
- The `serviceName` property names the RMI service.
- The `serviceInterface` property specifies the interface that the service implements.
- By default `RmiServiceExporter` attempts to bind to a RMI registry on port 1099 of the local machine. If no RMI registry is found at that port, `RmiServiceExporter` will start one.
- If you want to bind to a RMI registry at a different port or host, you can specify so with the `registryPort` and `registryHost` properties.

Wiring an RMI service

- Traditionally, RMI clients must use the RMI API's `Naming` class to look up a service from the RMI registry.
- Spring's `RmiProxyFactoryBean` is a factory bean that creates a proxy to an RMI service.
- The object can be injected in any other bean that needs a reference to the remote service.
- The client code does not know that it is dealing with an RMI service.
- The proxy catches any `RemoteExceptions` that may be thrown by the service and rethrows them as unchecked exceptions that may be safely ignored.

Wiring an RMI service

- `RmiProxyFactoryBean` produces a proxy object that talks to remote RMI services on behalf of the client. The client talks to the proxy through the service's interface as if the remote service were just a local POJO.



Wiring an RMI service

```
//spring-client.xml
<bean id="transformerService"
   class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://127.0.0.1:1099/
     TransformerService"/>
    <property name="serviceInterface"
     value="transformer.services.ITransformer"/>
   </bean>
```

- The URL of the service is set through `RmiProxyFactoryBean's serviceUrl` property.
- The interface that the service provides is specified with the `serviceInterface` property.

Starting the server

```
 //StartServer.java
public class StartServer {
    public static void main(String[] args) {
        ApplicationContext factory = new
     ClassPathXmlApplicationContext("classpath:spring-
     server.xml");
    }
}
```

Running the client

```
 //StartClient.java
public class StartClient {
    public static void main(String[] args) {
        ApplicationContext factory = new
     ClassPathXmlApplicationContext("classpath:spring-
     client.xml");
        ITransformer
     transformerServ=(ITransformer)factory.getBean("transformerSer
     vice");
        String text="Ana are mere";
        System.out.println("Sending text = " + text);
        String response=transformerServ.transform(text);
        System.out.println("Received response = " + response);

    }
}
```

XML

- XML documents (Web Programming)
- Well-formed XML documents
- Valid XML documents
- XML parsing
- API for XML

Well formed XML documents

- Non-empty elements are delimited by both a start-tag and an end-tag.

```
<message> … </message>
<text> …</text->
```

- Empty elements may be marked with an empty-element (self-closing) tag, such as `<text />`. This is equal to `<text></text>`.
- All attribute values are quoted with either single (') or double (") quotes. Single quotes close a single quote and double quotes close a double quote.
- Tags may be nested but must not overlap. Each non-root element must be completely contained in another element.

```
<message> <text> </message> </text>
```

- The document complies with its declared character encoding.
- Element names are case-sensitive.

```
<message> ... </message>
<Message> ... </message>
  <SENDER> ... </senDer>
```

XML Documents Validation

```
<message sender="ana" receiver="mihai">
        <text> Hello! </text>
</message>

<message>
        <text> Hello! </text>
        <sender> ana </sender>
        <receiver> mihai </receiver>
</message>
<message>
        <sender> ana </sender>
        <receiver> mihai </receiver>
        <text> Hello! </text>
</message>
```

DTD, XML Schema, RELAX NG

Document Type Definition (DTD)

- It is the oldest schema format for XML.
- A DTD specifies the kinds of tags that can be included in an XML document, and their order.

```
//message.dtd.
<?xml version="1.0"?>
<!ELEMENT message (sender, receiver, text)>
<!ELEMENT sender (#PCDATA)>
<!ELEMENT receiver (#PCDATA)>
<!ELEMENT text (#PCDATA)>
<message>
        <text> Hello! </text>
        <sender> ana </sender>
        <receiver> mihai </receiver>
</message>
```

DTDs

- Markup declarations are used to declare which elements types, attribute lists, entities and notations are allowed in the structure of the corresponding class of XML documents.
- *Element Type Declaration*: defines an element and its possible content. A valid XML document contains only elements that are defined in the DTD.
- An element's content is specified by some keywords and characters:
  - **EMPTY** for no content
  - **ANY** for any content
  - , for orders
  - | for alternatives ("either...or")
  - ( ) for groups
  - * for any number (zero or more)
  - + for at least once (one or more)
  - ? mark for optional (zero or one)
  - If there is no *, + or ?, the element must occur exactly one time.
  - #**PCDATA** for parsed character data

DTDs

- Attributes in DTDs

```
<!ELEMENT book (#PCDATA)>
<!ATTLIST book
     title CDATA #REQUIRED
     date CDATA #IMPLIED
     author CDATA "unknown"
>
```

- Attribute's type:
  - `(value1 | value2 | ...)` A list of values separated by vertical bars
  - `CDATA` Unparsed character data (a text string)
  - `ID` A name that no other ID attribute shares
  - `IDREF` A reference to an ID defined elsewhere in the document
  - `IDREFS` A space-separated list containing one or more ID references
  - `ENTITY, etc`

DTDs

- Attribute-Specification Parameters:
  - **#REQUIRED** The attribute value must be specified in the document.
  - **#IMPLIED** The value does not need to be specified in the document. If it isn't, the application will have a default value it uses.
  - "**defaultValue**" The default value to use if a value is not specified in the document.
  - **#FIXED** "**fixedValue**" The value to use. If the document specifies any value at all, it must be the same.
- Entities:

```
<!ENTITY chat "XYZ Chat">
<!ENTITY copyr "&#169;"> <!-- Copyright Symbol -->
<message>
    <text> &chat; &copyr; abcdef </text>
</message>
<!-- XYZ Chat © abcdef-->
```

DTDs

- DOCTYPE is used to include a DTD in an XML document:
  - Internal DTD: defined in the same document

```
<?xml version="1.0" standalone="yes" ?>
<!DOCTYPE message [
    <!ELEMENT message (text)>
    <!ELEMENT text (#PCDATA)>
]>
<message><text> Hello.</text></message>
```

  - External DTD: defined in another file (private, public)
    - Private:

```
<!DOCTYPE message SYSTEM "message.dtd">
```

    - Public:

```
<!DOCTYPE root_element PUBLIC "DTD_name" "DTD_location">
```

The "DTD_location"(relative or absolute URL) is used to find the public DTD if it cannot be located by the "DTD_name".

DTDs

- An XML document may contain two DOCTYPE declarations: an internal and an external declaration:

```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE document SYSTEM "message.dtd" [
  <!ELEMENT sender (#PCDATA)>
  <!ELEMENT text (#PCDATA)>
]>
```

- Conflicts between the name of the elements may appear.

  Eg. `<text>` appears in `message.dtd` and the internal dtd declaration

Namespaces in DTDs

- Are used to solve name conflicts between elements from different DTDs.
- Declaring a namespace:

```
<!ELEMENT title (#PCDATA)>
<!ATTLIST title
     xmlns CDATA #FIXED "http://www.xyz.com/book"
>
```

The attribute should be defined for every element in the DTD.

- Referencing a namespace:

```
<title xmlns="http://www.xyz.com/book">
     Overview
</title>
```

- Using a prefix:

```
<SL:title xmlns:SL='http:/www.xyz.com/book'>
     Overview
</SL:title>
```

The prefix can be used for all contained tags.

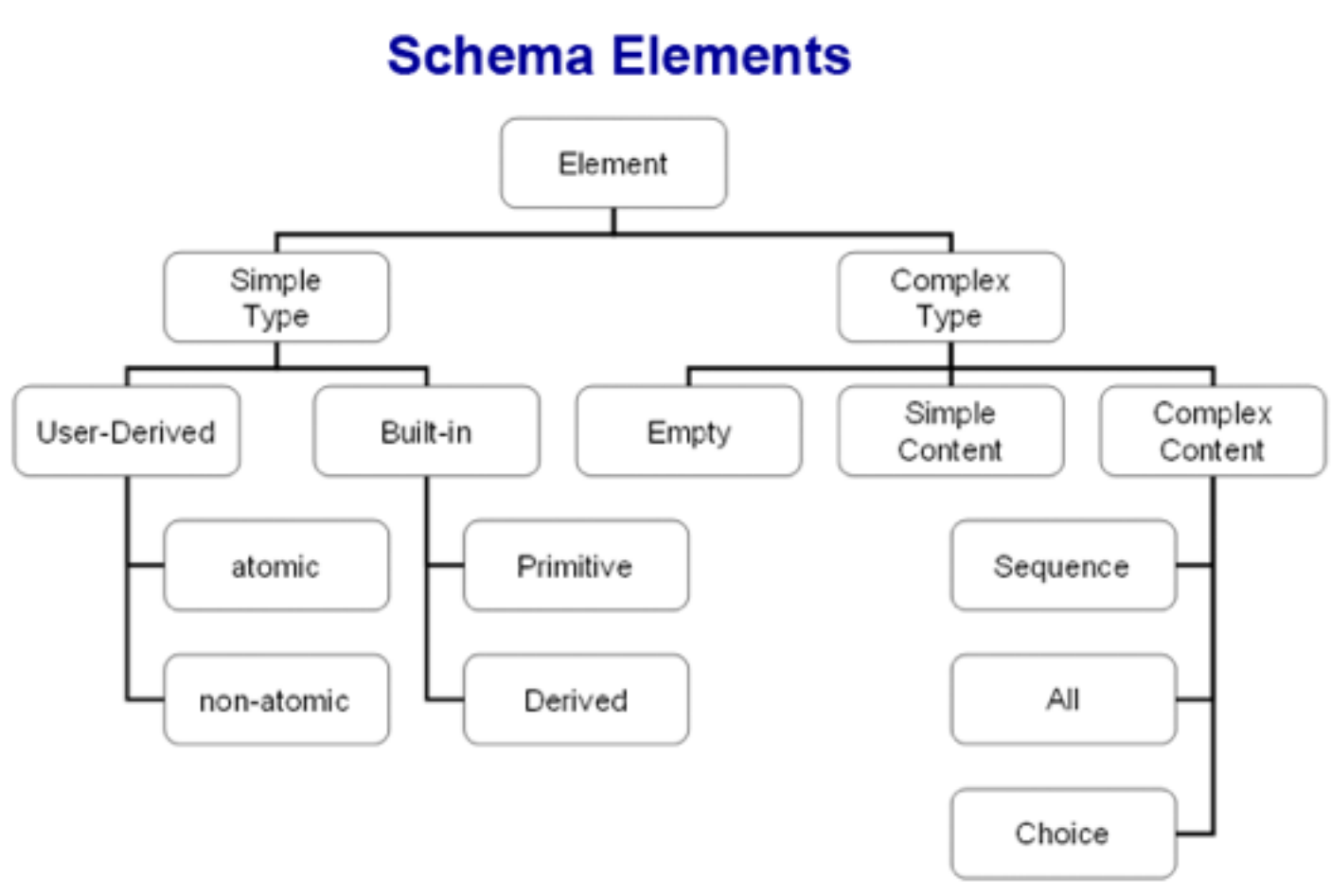
Disadvantages of DTDs

- They are not written in XML syntax, which means you have to learn a new syntax in order to write them.
- There is no support for namespaces.
- There are no constraints imposed on the kind of character data allowed, so datatyping is not possible.
- There is minimal support for code modularity and none for inheritance.
- Large DTDs are hard to read and maintain.
- There are no default values for elements and attribute defaults must be specified when they are declared.

XML Schema

- XML Schema is an XML-based language used to create XML-based languages and data models.
- An XML schema defines:
  - element and attribute names for a class of XML documents,
  - the structure of the XML documents,
  - the type of content that each element can hold.
- XML documents that attempt to adhere to an XML schema are said to be instances of that schema.

XML Schema Elements



XML Schema Elements

1. Elements can be of simple type or complex type.
2. Simple type elements can only contain text. They can not have child elements or attributes.
3. All the built-in types are simple types (e.g, xs:string).
4. Schema authors can derive simple types by restricting another simple type.
5. Simple types can be atomic (e.g, strings and integers) or non-atomic (e.g, lists).
6. Complex-type elements can contain child elements and attributes as well as text.
7. By default, complex-type elements have complex content, meaning that they have child elements.
8. Complex-type elements can be limited to having simple content, meaning they only contain text. They are different from simple type elements in that they have attributes.
9. Complex types can be limited to having no content, meaning they are empty, but they may have attributes.
10. Complex types may have mixed content - a combination of text and child elements.

XML Schema Example

```
<!-- Person.xsd -->
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="FirstName" type="xs:string" />
        <xs:element name="LastName" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="Person.xsd">
    <FirstName> Ion </FirstName>
    <LastName>Vasilescu</LastName>
</Person>
```

Simple Type Elements

- Simple-type elements have no children or attributes.
- XML Schema specifies 44 built-in types, 19 of which are primitive.
  - Primitive: **`string`**, **`boolean`**, **`decimal`**, **`float`**, **`double`**, **`dateTime`**, **`time`**, **`date`**, etc.
  - Derived built-in: **`ID`**, **`IDREF`**, **`IDREFS`**, **`Name`**, **`long`**, **`int`**, **`short`**, **`byte`**, etc.
- A simple-type element is defined using the **`type`** attribute.

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="FirstName" type="xs:string"/>
</xs:schema>
```

User-derived Simple Types

- A schema author can derive a new simple type using the `<xs:simpleType>` element.
- This simple type can then be used in the same way the built-in simple types are.
- Simple types are derived by restricting built-in simple types or other user-derived simple types.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:simpleType name="Password">
  <xs:restriction base="xs:string">
   <xs:length value="8"/>
  </xs:restriction>
 </xs:simpleType>
<xs:element name="PW" type="Password"/>
</xs:schema>
```

- Restrictions: `length, minLength, maxLength,totalDigits, fractionDigits,` etc.

Complex Type Elements

- Complex-type elements have attributes, child elements, or some combination of the two.
- It is necessary to specify that a complex-type element is a complex type, using the **`xs:complexType`** element.

```
<xs:element name="ElementName">
 <xs:complexType>
  <!--Content Model -->
 </xs:complexType>
</xs:element>
```

- Content models are used to indicate the structure and order in which child elements can appear within their parent element.
  - **`xs:sequence`** - the elements must appear in the order specified.
  - **`xs:all`** - the elements must appear, but the order is not important.
  - **`xs:choice`** - only one of the elements can appear.

Complex Type Elements

```
<xs:element name="ElementName">
 <xs:complexType>
  <xs:choice>
   <xs:element name="Child1" type="xs:string"/>
   <xs:element name="Child2">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="AA" type="xs:string"/>
      <xs:element name="BB" type="xs:boolean"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="Child3" type="xs:decimal"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Complex Type Elements

- By default, elements that are declared locally must appear once and only once within their parent.
- This constraint can be changed using the `minOccurs` and `maxOccurs` attributes. The default value of each of these attributes is 1.

```
<xs:element name="Title" minOccurs="0"/>
<xs:element name="Chapter" type="xs:string" maxOccurs="unbounded"/>
```

- A complex type element may contain both child elements and character text.

```
<xs:element name="ElementName">
 <xs:complexType mixed="true">
  <xs:sequence>
   <xs:element name="Child1" type="xs:string"/>
   <xs:element name="Child2" type="xs:int"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Attributes

- Only complex-type elements can contain attributes.
- Attributes are of simple types.
- Attribute values can be restricted.
- Attributes can have default values. To specify a default value, the **`default`** attribute of the **`xs:attribute`** element is used.
- Attribute values can be **`fixed`**, meaning that, if they appear in the instance document, they must contain a specified value.
- By default, attributes are optional, but they can be required by setting the **`use`** attribute of **`xs:attribute`** to **`required`**.
- Empty element: is an element that has no content, but it may have attributes.

```
<xs:element name="HomePage">
     <xs:complexType>
      <xs:attribute name="URL" type="xs:anyURI“ use=“required”/>
     </xs:complexType>
</xs:element>

<HomePage URL="http://www.xyz.com"/>
```

Elements with simple content

- An element with simple content is one that only contains character data.
- If such an element contains one or more attributes, then it is a complex-type element.
- Elements with simple content and attributes are declared using the `xs:simpleContent` element and then extending the element with the `xs:extension` element, which must specify the type of simple content contained.

```
<xs:element name="ElementName">
 <xs:complexType>
  <xs:simpleContent>
   <xs:extension base="xs:string">
    <xs:attribute name="AttName" type="xs:boolean"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>
</xs:element>
```

Elements with Complex Content

- Elements that have child elements are said to contain complex content.
- Attributes for such elements are declared after the element's model group.

```
<xs:element name="Name">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="FirstName" type="xs:string"/>
       <xs:element name="LastName" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="Pseudonym" type="xs:boolean"/>
      <xs:attribute name="HomePage" type="xs:anyURI"/>
     </xs:complexType>
</xs:element>
<Name Pseudonym="true" HomePage="http://www.xyz.com">
  <FirstName>Ion</FirstName>
  <LastName>Vasilescu</LastName>
 </Name>
```

Namespaces

- Namespaces are used to group elements and attributes that relate to each other in some special way.
- Namespaces are held in a unique URI (Uniform Resource Identifier).
- To populate a namespace with an XML schema, set the `targetNamespace` attribute of the `xs:schema` element to a URI.
- You must also include a `xmlns` attribute, so that global elements declared in the target namespace can be referenced within the schema.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://xyz.com/Person"
  xmlns="http://xyz.com/Person“>
<?xml version="1.0"?>
<Person  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://xyz.com/Person"
 xsi:schemaLocation="http://xyz.com/Person Person.xsd">
 <Name>
  <FirstName>Ion</FirstName>
  <LastName>Vasilescu</LastName>
 </Name>
</Person>
```

Namespaces

```
<?xml version="1.0"?>
<art:Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:art="http://www.xyz.com/Person"
 xsi:schemaLocation="http://www.xyz.com/Person Person.xsd">
 <art:Name>
  <art:FirstName>Ion</art:FirstName>
  <art:LastName>Vasilescu</art:LastName>
 </art:Name>
</art:Person>
```

Remark: Requires that the `elementFormDefault` and `attributeFormDefault` attributes in the `xs:schema` element are set to `"qualified"`

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns="http://www.xyz.com/Person"
     targetNamespace="http://www.xyz.com/Person"
     elementFormDefault="qualified"
     attributeFormDefault="qualified">
…</xs:schema>
```

XML Schema

- Advantages of XML Schemas:
  - Offer extensive and robust support for data types. The data types can be extended and customized.
  - Users can define their own data type.
  - Uses XML syntax.
  - Extensible.
  - Self-documenting.
  - Provide namespace.
- Disadvantages of XML Schemas:
  - XML Schema is complicated.
  - XML Schema cannot require a specific root element.
  - When describing mixed content, the character data cannot be constrained in any way.
  - Element defaults can only be character data.

XML Parsing

- SAX, the Simple API for XML, is an event-driven parser.
  - It reads the XML document sequentially moving from one element to the next.
  - It cannot go back to already parsed elements.
  - It is a light-weight parser.
  - It cannot be used to modify the XML document.
- DOM Document Object Model
  - It describes an XML document as a tree-like structure, with every XML element being a node in the tree.
  - A DOM-based parser reads the entire document, and forms the corresponding document tree in memory.
  - It can be used to construct new XML documents or to modify existing XML documents.

C# XML

- The **`System.Xml`** namespace comprises the following namespaces and core classes:
  - **`System.Xml.*`**
    - **`XmlReader`** and **`XmlWriter`** high-performance, forward-only cursors for reading or writing an XML stream (SAX)
    - **`XmlDocument`** represents an XML document in a W3C-style DOM
  - **`System.Xml.XPath`** infrastructure and API (XPathNavigator) for XPath, a string-based language for querying XML
  - **`System.Xml.XmlSchema`** infrastructure and API for (W3C) XSD schemas
  - **`System.Xml.Xsl`** infrastructure and API (XslCompiledTransform) for performing (W3C) XSLT transformations of XML
  - **`System.Xml.Serialization`** supports the serialization of classes to and from XML
  - **`System.Xml.Linq - (`**LINQ to XML) a lightweight LINQ- friendly XML document object model, plus a set of supplementary query operators

C# XMLReader

- `XmlReader` is a high-performance class for reading an XML stream in a low-level, forward-only manner.
- Instantiating an XmlReader:

```
XmlReader reader = XmlReader.Create ("person.xml");
XmlReader reader = XmlReader.Create (new System.IO.StringReader
   (myString));
```

- `XmlReaderSettings` class is used to control parsing and validation options.
  - Properties

```
IgnoreComments: bool // Skip over comment nodes
IgnoreProcessingInstructions: bool //Skip over processing
   instructions
IgnoreWhitespace: bool  // Skip over whitespace?

XmlReaderSettings settings = new XmlReaderSettings(  );
settings.IgnoreWhitespace = true;
using (XmlReader reader = XmlReader.Create ("person.xml", settings))
```

XMLReader

- The units of an XML stream are XML nodes. The reader traverses the stream in textual (depth-first) order. The `Depth` property of the reader returns the current depth of the cursor.
- The easiest way to read from an `XmlReader` is to call `Read`.
- It advances to the next node in the XML stream
- The first call to `Read` positions the cursor at the first node.
- When `Read` returns false, it means the cursor has advanced past the last node, at which point the `XmlReader` should be closed.

```
XmlReaderSettings settings = new XmlReaderSettings(  );
settings.IgnoreWhitespace = true;
using (XmlReader reader = XmlReader.Create("person.xml", settings))
  while (reader.Read(  ))  {
        Console.WriteLine (reader.NodeType);
  }
```

XMLReader

- `NodeType:XmlNodeType`:
  - `None, XmlDeclaration, Element, EndElement, Text, Attribute Comment, Entity, EndEntity, EntityReference, ProcessingInstruction, CDATA Document, DocumentType, DocumentFragment, Whitespace`, etc.
- `XmlReader` has two string properties that provide access to a node's content: `Name` and `Value`. Depending on the node type, either `Name` or `Value` (or both) is populated.
- Other methods:
  - `ReadStartElement`
  - `ReadEndElement`
  - `ReadElementContentAsString`
  - `ReadAttributeValue`
  - Etc.

XMLReader

- XmlReader provides an indexer that gives direct (random) access to an element's attributes—by name or position.

```
<customer id="123" status="archived"/>
```

```
Console.WriteLine (reader ["id"]);                // 123
Console.WriteLine (reader ["status"]);            // archived
Console.WriteLine (reader ["bogus"] == null);     // True
```

- Using the indexer is equivalent to calling `GetAttribute` method.

```
Reader.GetAttribute(0);
Reader.GetAttribute("id");
```

- The return type is a `string` (`null` if the attribute is not found).

XMLReader

- The **`XmlReader`** class can provide validation of a DTD or an XML Schema when parsing the document
- The validation settings are specified with the **`ValidationType.DTD`** or **`ValidationType.Schema`** properties on the **`XmlReaderSettings`** class.
- The default setting for both properties is **`ValidationType.None`** so the **`XmlReader`** does not validate.

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationEventHandler += new
   ValidationEventHandler(anValidationEventHandler);

settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add(namespace, XmlReader.Create("schema.xsd"));
using (XmlReader xmlValidatingReader = XmlReader.Create("file.xml",
   settings))
{
    while (xmlValidatingReader.Read()) { }
}
```

XMLWriter

- `XmlWriter` is a forward-only writer of an XML stream.
- The design of `XmlWriter` is symmetrical to `XmlReader`.

```
XmlWriterSettings settings = new XmlWriterSettings(  );
settings.Indent = true;
using (XmlWriter writer = XmlWriter.Create ("foo.xml", settings)){
  writer.WriteStartElement ("customer");
  writer.WriteAttributeString ("id", "1");
 writer.WriteAttributeString ("status", "archived");
  writer.WriteElementString ("firstname", "Jim");
  writer.WriteElementString ("lastname"," Bo");
  writer.WriteEndElement(  );
}
<?xml version="1.0" encoding="utf-8" ?>
 <customer id="1" status="archived">
   <firstname>Jim</firstname>
   <lastname>Bo</lastname>
</customer>
```

XMLDocument

- `XmlDocument` is an in-memory representation of an XML document.
- The base type for all objects in an `XmlDocument` tree is `XmlNode`.
  - Derived types: `XmlDocument`, `XmlDocumentFragment`, `XmlEntity`, `XmlNotation`, `XmlLinkedNode`
  - `XmlLinkedNode` exposes `NextSibling` and `PreviousSibling` properties and is an abstract base for the following subtypes:
    - `XmlCharacterData`
    - `XmlDeclaration`
    - `XmlDocumentType`
    - `XmlElement`
    - `XmlEntityReference`
    - `XmlProcesingInstruction`

XMLDocument

- To load an `XmlDocument` from an existing source, an `XmlDocument` is instantiated and then the methods `Load` or `LoadXml` are called:
  - `Load` accepts a filename, `Stream`, `TextReader`, or `XmlReader`.
  - `LoadXml` accepts a literal XML string.
- To save a document, the method `Save` is called with a filename, `Stream`, `TextWriter`, or `XmlWriter`:

```
XmlDocument doc = new XmlDocument(  );
doc.Load ("customer1.xml");
doc.Save ("customer2.xml");
```

- To traverse an XML document, the following properties from XmlElement can be used:
  - `Attributes, FirstChild, ChildNodes, InnerText, LastChild, IsEmpty`
- To create and add new nodes:
  - Call one of the `CreateYYY` methods on the `XmlDocument`, such as `CreateElement`.
  - Add the new node into the tree by calling `AppendChild, PrependChild, InsertBefore,` or `InsertAfter` on the desired parent node.

Traversing using XMLDocument

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>B </lastname>
</customer>

XmlDocument doc = new XmlDocument( );
doc.Load ("customer.xml");
Console.WriteLine(doc.DocumentElement.ChildNodes[0].InnerText); //Jim
Console.WriteLine (doc.DocumentElement.ChildNodes[1].InnerText); //B
```

- GetAttribute
- GetElementsByTagName

Creating an XMLDocument

```
XmlDocument doc = new XmlDocument (  );
doc.AppendChild (doc.CreateXmlDeclaration ("1.0", null, "yes"));
XmlAttribute id     = doc.CreateAttribute ("id");
XmlAttribute status = doc.CreateAttribute ("status");
id.Value     = "123";
status.Value = "archived";
XmlElement firstname = doc.CreateElement ("firstname");
XmlElement lastname  = doc.CreateElement ("lastname");
firstname.AppendChild (doc.CreateTextNode ("Jim"));
lastname.AppendChild  (doc.CreateTextNode ("B"));

XmlElement customer = doc.CreateElement ("customer");
customer.Attributes.Append (id);
customer.Attributes.Append (status);
customer.AppendChild (lastname);
customer.AppendChild (firstname);
doc.AppendChild (customer);
```

Validating an XmlDocument

```
XmlReaderSettings settings = new XmlReaderSettings(  );
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

XmlDocument xmlDoc = new XmlDocument(  );
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try {
    xmlDoc.Load (r);

  }catch (XmlSchemaValidationException ex) { ... }
```

XmlSerialization

- The .NET Framework provides a dedicated XML serialization engine called **`XmlSerializer`** in the **`System.Xml.Serialization`** namespace.
- It is suitable for serializing .NET types to XML file.
- To use **`XmlSerializer`**, you instantiate it and call **`Serialize`** or **`Deserialize`** with a **`Stream`** and object instance.
- **`XmlSerializer`** can serialize types without any attributes.
- By default, it serializes all public fields and properties on a type.
- Members can be excluded from serialization with the **`XmlIgnore`** attribute.
- By default, fields and properties serialize to an XML element.
- You can request an XML attribute to be used instead, using **`XmlAttribute`** attribute.
- Other attributes: **`XmlElement`**, **`XmlRoot`**, **`XmlArray`**, **`XmlArrayItem`**

XmlSerialization

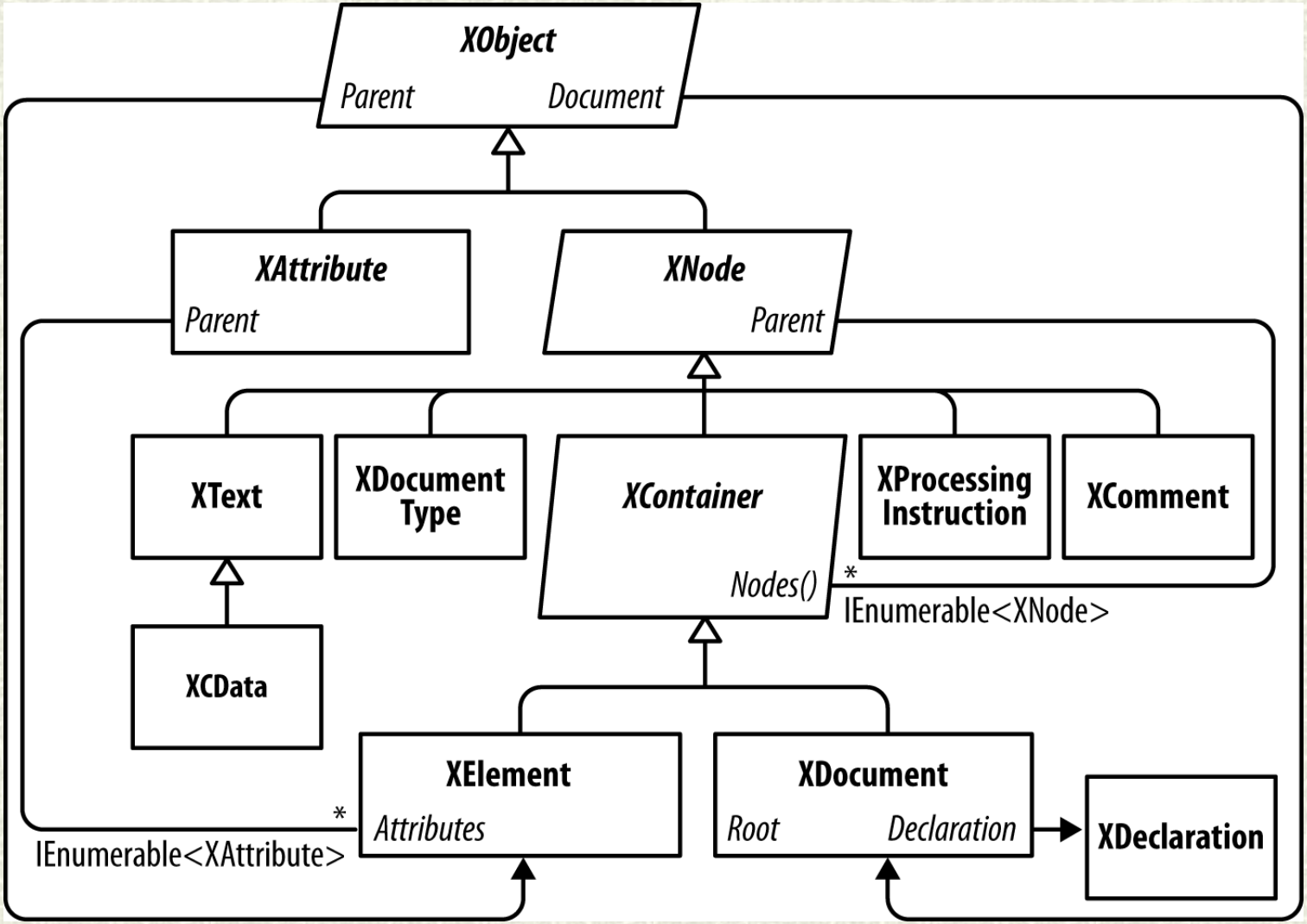
```
public class Person {
  [XmlElement ("FirstName")] public string name;
  [XmlAttribute ("Age")] public int age;}
Person p = new Person(  );
p.name = "Stacey"; p.age = 30;
XmlSerializer xs = new XmlSerializer (typeof (Person));
using (Stream s = File.Create ("person.xml"))
  xs.Serialize (s, p);
Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) xs.Deserialize (s);
Console.WriteLine (p2.Name + " " + p2.Age);   // Stacey 30

<?xml version="1.0"?>
<Person Age="30" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <FirstName>Stacey</FirstName>
</Person>
```

LINQ to XML

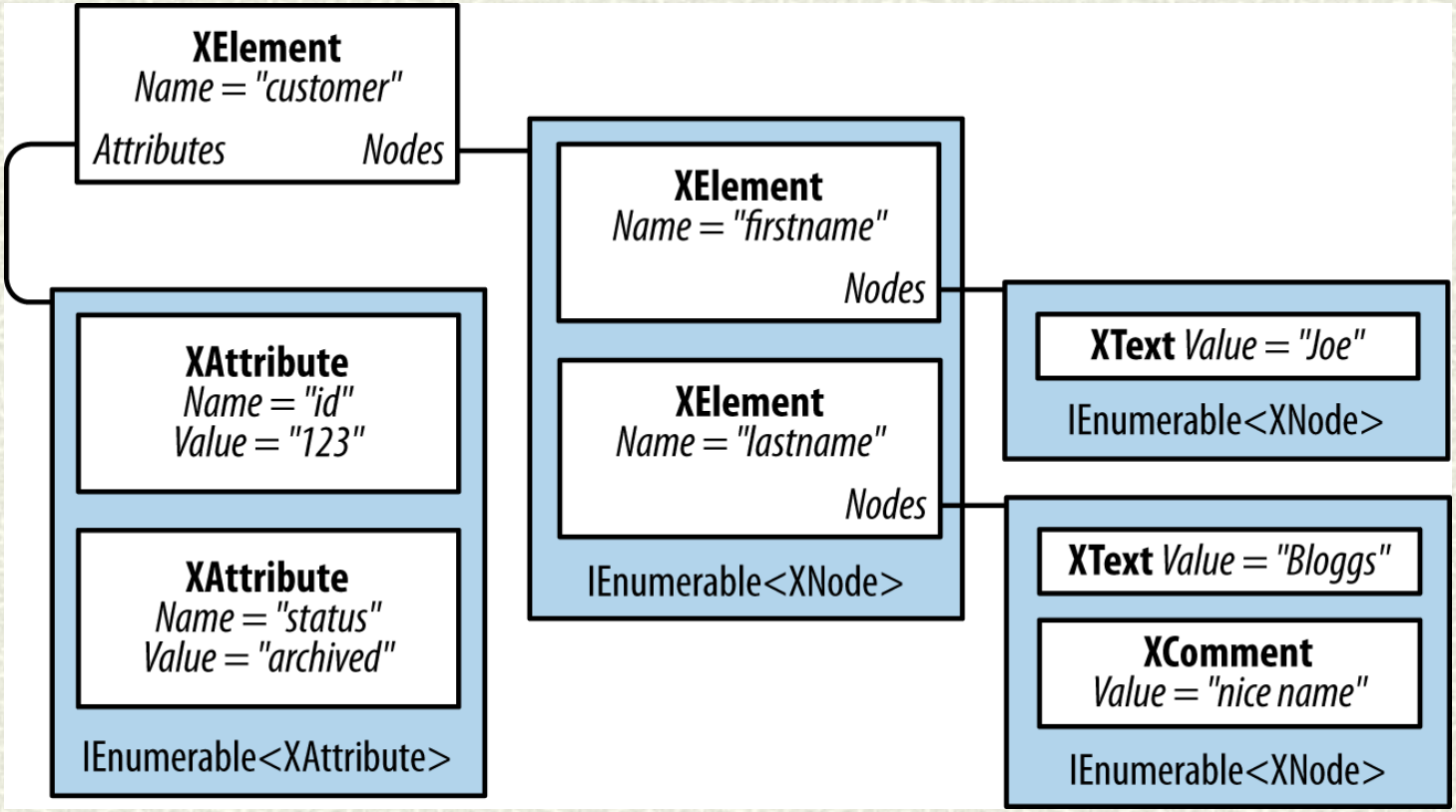
- Since .NET Framework 3.5
- A lightweight façade over the low-level `XmlReader` and `XmlWriter` classes
- It comprises a lightweight LINQ- friendly XML document object model, and a set of query operators:
  - X-DOM (an XML DOM)
  - ~ 10 supplementary query operators
- The X-DOM consists of types like: `XDocument`, `XElement`, and `XAttribute`.
- The X-DOM types are not tied to LINQ.
- Loading, instantiating, updating, and saving an X-DOM can be done without writing a LINQ query.

X-DOM Types


XObject
Parent
Document
XAttribute
Parent
XNode
Parent
XText
XDocument Type
XContainer
Nodes()
XProcessing Instruction
XComment
*
IEnumerable<XNode>
XCData
XElement
Attributes
XDocument
Root
Declaration
XDeclaration
*
IEnumerable<XAttribute>

XDOM Example

```
<customer id='123' status='archived'>
     <firstname>Joe</firstname>
      <lastname>Bloggs<!--nice name--></lastname>
 </customer>
```



XDOM -Loading and Parsing

- `XElement` and `XDocument` provide methods to build an X-DOM tree from an existing source:
  - `Load` builds an X-DOM from a file, URI, Stream, TextReader, or XmlReader.
  - `Parse` builds an X-DOM from a string.

```
XDocument fromWeb = XDocument.Load ("http://xyz.com/file.xml");
XElement fromFile = XElement.Load (@"PathTo\file.xml");
XElement config = XElement.Parse (
            @"<configuration>
               <client enabled='true'>
                  <timeout>30</timeout>
               </client>
             </configuration>");
```

Building an XDOM

- An X-DOM tree can be manually created by instantiating objects and adding them to a parent via XContainer's `Add` method.

```
XElement lastName = new XElement ("lastname", "Bloggs");
  lastName.Add (new XComment ("nice name"));
  XElement customer = new XElement ("customer");
  customer.Add (new XAttribute ("id", 123));
  customer.Add (new XElement ("firstname", "Joe"));
  customer.Add (lastName);
 Console.WriteLine (customer.ToString());
```

```
//Output
<customer id="123">
      <firstname>Joe</firstname>
      <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

Functional Construction

- Code resembles the shape of the XML

```
XElement customer =
    new XElement ("customer", new XAttribute ("id", 123),
      new XElement ("firstname", "joe"),
      new XElement ("lastname", "bloggs",
        new XComment ("nice name")
      )
    );
```

//Output

```
<customer id="123">
    <firstname>Joe</firstname>
    <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

Navigating and Querying

```
var bench = new XElement ("bench",
                  new XElement ("toolbox",
                    new XElement ("handtool", "Hammer"),
                    new XElement ("handtool", "Rasp")
                  ),
                  new XElement ("toolbox",
                    new XElement ("handtool", "Saw"),
                    new XElement ("powertool", "Nailgun")
                  ),
                  new XComment ("Be careful with the nailgun")
            );


IEnumerable<string> query = from toolbox in bench.Elements()
                            from tool in toolbox.Elements()
                               where tool.Name == "handtool"
                                select tool.Value;
     //Result
{ "Hammer", "Rasp", "Saw" }
```

Navigating and Querying

```
var bench = new XElement ("bench",
                  new XElement ("toolbox",
                    new XElement ("handtool", "Hammer"),
                    new XElement ("handtool", "Rasp")
                  ),
                  new XElement ("toolbox",
                    new XElement ("handtool", "Saw"),
                    new XElement ("powertool", "Nailgun")
                  ),
                  new XComment ("Be careful with the nailgun")
            );


int x = bench.Elements ("toolbox").Count(); // 2

from tool in bench.Elements ("toolbox").Elements ("handtool")
   select tool.Value.ToUpper();

//{ "HAMMER", "RASP", "SAW" }
```

Retrieving a single element

- The method `Element` returns the first matching element of the given name.

```
XElement settings = XElement.Load ("databaseSettings.xml");

string cx = settings.Element ("database").
   Element("connectString").Value;
```

- `Element` is equivalent to calling `Elements()` and then applying LINQ's `FirstOrDefault` query operator.
- `Element` returns null if the requested element doesn't exist.

Reference:

Joseph Albahari & Ben Albahari, *C# 6.0 in a Nutshell*, O'Reilly, 2016