Lesson: Document Object Model

This lesson presents the Document Object Model (DOM). A DOM is a standard tree structure, where each node contains one of the components from an XML structure. The two most common types of nodes are element nodes and text nodes. Using DOM functions lets you create nodes, remove nodes, change their contents, and traverse the node hierarchy.

The examples in this lesson demonstrate how to parse an existing XML file to construct a DOM, display and inspect the DOM hierarchy, and explore the syntax of namespaces. It also shows how to create a DOM from scratch, and see how to use some of the implementation-specific features in Sun's JAXP implementation to convert an existing data set to XML.

When to Use DOM

The Document Object Model standard is, above all, designed for documents (for example, articles and books). In addition, the JAXP 1.4.2 implementation supports XML Schema, something that can be an important consideration for any given application.

On the other hand, if you are dealing with simple data structures and if XML Schema is not a big part of your plans, then you may find that one of the more object-oriented standards, such as JDOM or dom4j, is better suited for your purpose.

From the start, DOM was intended to be language-neutral. Because it was designed for use with languages such as C and Perl, DOM does not take advantage of Java's object-oriented features. That fact, in addition to the distinction between documents and data, also helps to account for the ways in which processing a DOM differs from processing a JDOM or dom4j structure.

In this section, we will examine the differences between the models underlying those standards to help you choose the one that is most appropriate for your application.

Documents Versus Data

The major point of departure between the document model used in DOM and the data model used in JDOM or dom4j lies in:

- The kind of node that exists in the hierarchy
- The capacity for mixed content

It is the difference in what constitutes a "node" in the data hierarchy that primarily accounts for the differences in programming with these two models. However, the capacity for mixed content, more than anything else, accounts for the difference in how the standards define a node. So we start by examining DOM's mixed-content model.

Mixed-Content Model

Text and elements can be freely intermixed in a DOM hierarchy. That kind of structure is called mixed content in the DOM model.

Mixed content occurs frequently in documents. For example, suppose you wanted to represent this structure:

```
<sentence>This is an <bold>important</bold> idea.</sentence>
```

The hierarchy of DOM nodes would look something like this, where each line represents one node:

```
ELEMENT: sentence
   + TEXT: This is an
   + ELEMENT: bold
     + TEXT: important
   + TEXT: idea.
```

Note that the sentence element contains text, followed by a sub-element, followed by additional text. It is the intermixing of text and elements that defines the mixed-content model.

Types of Nodes

To provide the capacity for mixed content, DOM nodes are inherently very simple. In the foregoing example, the "content" of the first element (its value) simply identifies the kind of node it is.

First-time users of a DOM are usually thrown by this fact. After navigating to the `<sentence>` node, they ask for the node's "content", and expect to get something useful. Instead, all they can find is the name of the element, `sentence`.

Note - The DOM Node API defines `nodeValue()`, `nodeType()`, and `nodeName()` methods. For the first element node, `nodeName()` returns `sentence`, while `nodeValue()` returns null. For the first text node, `nodeName()` returns `#text`, and `nodeValue()` returns "`This is an `". The important point is that the **value** of an element is not the same as its **content**.

In the example above, what does it mean to ask for the "text" of the sentence? Any of the following could be reasonable, depending on your application:

- This is an
- This is an idea.
- This is an important idea.
- This is an <bold>important</bold> idea.

A Simpler Model

With DOM, you are free to create the semantics you need. However, you are also required to do the processing necessary to implement those semantics. Standards such as JDOM and dom4j, on the other hand, make it easier to do simple things, because each node in the hierarchy is an object.

Although JDOM and dom4j make allowances for elements having mixed content, they are not primarily designed for such situations. Instead, they are targeted for applications where the XML structure contains data.

The elements in a data structure typically contain either text or other elements, but not both. For example, here is some XML that represents a simple address book:

```
<addressbook>
    <entry>
        <name>Fred</name>
        <email>fred@home</email>
    </entry>
      ...
</addressbook>
```

Note - For very simple XML data structures like this one, you could also use the regular-expression package (`java.util.regex`) built into the Java platform in version 1.4.

In JDOM and dom4j, after you navigate to an element that contains text, you invoke a method such as `text()` to get its content. When processing a DOM, though, you must inspect the list of sub-elements to "put together" the text of the node, as you saw earlier - even if that list contains only one item (a TEXT node).

So for simple data structures such as the address book, you can save yourself a bit of work by using JDOM or dom4j. It may make sense to use one of those models even when the data is technically "mixed" but there is always one (and only one) segment of text for a given node.

Here is an example of that kind of structure, which would also be easily processed in JDOM or

dom4j:

```
<addressbook>
    <entry>Fred
        <email>fred@home</email>
    </entry>
      ...
</addressbook>
```

Here, each entry has a bit of identifying text, followed by other elements. With this structure, the program could navigate to an entry, invoke `text()` to find out whom it belongs to, and process the `<email>` sub-element if it is at the correct node.

Increasing the Complexity

But for you to get a full understanding of the kind of processing you need to do when searching or manipulating a DOM, it is important to know the kinds of nodes that a DOM can conceivably contain.

Here is an example that illustrates this point. It is a representation of this data:

```
<sentence>
    The &projectName; <![CDATA[<i>project</i>]]> is
    <?editor: red><bold>important</bold><?editor: normal>.
</sentence>
```

This sentence contains an **entity reference** - a pointer to an entity that is defined elsewhere. In this case, the entity contains the name of the project. The example also contains a CDATA section (uninterpreted data, like `<pre>` data in HTML) as well as **processing instructions** (`<?...?>`), which in this case tell the editor which color to use when rendering the text.

Here is the DOM structure for that data. It is representative of the kind of structure that a robust application should be prepared to handle:

```
+ ELEMENT: sentence
       + TEXT: The
       + ENTITY REF: projectName
        + COMMENT:
        The latest name we are using
        + TEXT: Eagle
       + CDATA: <i>project</i>
       + TEXT: is
       + PI: editor: red
       + ELEMENT: bold
          + TEXT: important
       + PI: editor: normal
```

This example depicts the kinds of nodes that may occur in a DOM. Although your application may be able to ignore most of them most of the time, a truly robust implementation needs to recognize and deal with each of them.

Similarly, the process of navigating to a node involves processing sub-elements, ignoring the ones you are not interested in and inspecting the ones you are, until you find the node you are interested in.

A program that works on fixed, internally generated data can afford to make simplifying assumptions: that processing instructions, comments, CDATA nodes, and entity references will not exist in the data structure. But truly robust applications that work on a variety of data - especially data coming from the outside world - must be prepared to deal with all possible XML entities.

(A "simple" application will work only as long as the input data contains the simplified XML structures it expects. But there are no validation mechanisms to ensure that more complex structures will not exist. After all, XML was specifically designed to allow them.)

To be more robust, a DOM application must do these things:

1. When searching for an element:
    a. Ignore comments, attributes, and processing instructions.
    b. Allow for the possibility that sub-elements do not occur in the expected order.
    c. Skip over TEXT nodes that contain ignorable white space, if not validating.
2. When extracting text for a node:
    a. Extract text from CDATA nodes as well as text nodes.
    b. Ignore comments, attributes, and processing instructions when gathering the text.
    c. If an entity reference node or another element node is encountered, recurse (that is, apply the text-extraction procedure to all sub-nodes).

Note - From JAXP 1.2 onwards, the JAXP parser does not insert entity reference nodes into the DOM. Instead, it inserts a `TEXT` node containing the contents of the reference. The JAXP 1.1 parser, which was built into version 1.4 of the Java platform Standard Edition (the Java SE platform), on the other hand, does insert entity reference nodes. So a robust implementation that is parser-independent needs to be prepared to handle entity reference nodes.

Of course, many applications will not have to worry about such things, because the kind of data they see will be strictly controlled. But if the data can come from a variety of external sources, then the application will probably need to take these possibilities into account.

The code you need to carry out these functions is given near the end of this lesson in Searching for Nodesand Obtaining Node Content. Right now, the goal is simply to determine whether DOM is suitable for your application.

Choosing Your Model

As you can see, when you are using DOM, even a simple operation such as getting the text from a node can take a bit of programming. So if your programs handle simple data structures, then JDOM, dom4j, or even the 1.4 regular-expression package (`java.util.regex`) may be more appropriate for your needs.

For fully-fledged documents and complex applications, on the other hand, DOM gives you a lot of flexibility. And if you need to use XML Schema, then again DOM is the way to go - for now, at least.

If you process both documents and data in the applications you develop, then DOM may still be your best choice. After all, after you have written the code to examine and process a DOM structure, it is fairly easy to customize it for a specific purpose. So choosing to do everything in DOM means that you will only have to deal with one set of APIs, rather than two.

In addition, the DOM standard is a codified standard for an in-memory document model. It is powerful and robust, and it has many implementations. That is a significant decision-making factor for many large installations, particularly for large-scale applications that need to minimize costs resulting from API changes.

Finally, even though the text in an address book may not permit bold, italics, colors, and font sizes today, one day you may want to handle these things. Because DOM will handle virtually anything you throw at it, choosing DOM makes it easier to future-proof your application.

Reading XML Data into a DOM

In this section, you will construct a Document Object Model by reading in an existing XML file.

Note - In Extensible Stylesheet Language Transformations, you will see how to write out a DOM as an XML file. (You will also see how to convert an existing data file into XML with relative ease.)

Creating the Program

The Document Object Model provides APIs that let you create, modify, delete, and rearrange nodes. Before you try to create a DOM, it is helpful to understand how a DOM is structured. This series of examples will make DOM internals visible via a sample program called `DOMEcho`, which you will find in the directory *`INSTALL_DIR`*`/jaxp-`*`version`*`/samples/dom` after you have installed the JAXP API.

Create the Skeleton

First, build a simple program to read an XML document into a DOM and then write it back out again.

Start with the normal basic logic for an application, and check to make sure that an argument has been supplied on the command line:

```
public class DOMEcho {

    static final String outputEncoding = "UTF-8";

    private static void usage() {
        // ...
    }

    public static void main(String[] args) throws Exception {
        String filename = null;

        for (int i = 0; i < args.length; i++) {
            if (...) {
                // ...
            }
            else {
                filename = args[i];
                if (i != args.length - 1) {
                    usage();
                }
            }
        }

        if (filename == null) {
            usage();
        }
    }
}
```

This code performs all the basic set up operations. All output for `DOMEcho` uses UTF-8 encoding. The `usage()` method that is called if no argument is specified simply tells you what arguments `DOMEcho` expects, so the code is not shown here. A `filename` string is also declared, which will

be the name of the XML file to be parsed into a DOM by `DOMEcho`.

Import the Required Classes

In this section, all the classes are individually named so you that can see where each class comes from, in case you want to reference the API documentation. In the sample file, the import statements are made with the shorter form, such as `javax.xml.parsers.*`.

These are the JAXP APIs used by `DOMEcho`:

```
package dom;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
```

These classes are for the exceptions that can be thrown when the XML document is parsed:

```
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.*
```

These classes read the sample XML file and manage output:

```
import java.io.File;
import java.io.OutputStreamWriter;
import java.io.Printwriter;
```

Finally, import the W3C definitions for a DOM, DOM exceptions, entities and nodes:

```
import org.w3c.dom.Document;
import org.w3c.dom.DocumentType;
import org.w3c.dom.Entity;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
```

Handle Errors

Next, add the error-handling logic. The most important point is that a JAXP-conformant document builder is required to report SAX exceptions when it has trouble parsing an XML document. The DOM parser does not have to actually use a SAX parser internally, but because the SAX standard is already there, it makes sense to use it for reporting errors. As a result, the error-handling code for DOM applications is very similar to that for SAX applications:

```
private static class MyErrorHandler implements ErrorHandler {

    private PrintWriter out;

    MyErrorHandler(PrintWriter out) {
        this.out = out;
    }

    private String getParseExceptionInfo(SAXParseException spe) {
        String systemId = spe.getSystemId();
        if (systemId == null) {
            systemId = "null";
        }

        String info = "URI=" + systemId + " Line=" + spe.getLineNumber() +
                      ": " + spe.getMessage();
        return info;
    }
```

```
    public void warning(SAXParseException spe) throws SAXException {
        out.println("Warning: " + getParseExceptionInfo(spe));
    }

    public void error(SAXParseException spe) throws SAXException {
        String message = "Error: " + getParseExceptionInfo(spe);
        throw new SAXException(message);
    }

    public void fatalError(SAXParseException spe) throws SAXException {
        String message = "Fatal Error: " + getParseExceptionInfo(spe);
        throw new SAXException(message);
    }
}
```

As you can see, the `DomEcho` class's error handler generates its output using `PrintWriter` instances.

Instantiate the Factory

Next, add the following code to the `main()` method, to obtain an instance of a factory that can give us a document builder.

```
public static void main(String[] args) throws Exception {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

    // ...
}
```

Get a Parser and Parse the File

Now, add the following code to `main()` to get an instance of a builder, and use it to parse the specified file.

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(new File(filename));
```

The file being parsed is provided by the `filename` variable that was declared at the beginning of the `main()` method, which is passed to `DOMEcho` as an argument when the program is run.

Configuring the Factory

By default, the factory returns a non-validating parser that knows nothing about name spaces. To get a validating parser, or one that understands name spaces (or both), you can configure the factory to set either or both of those options using the following code.

```
public static void main(String[] args) throws Exception {

    String filename = null;
    boolean dtdValidate = false;
    boolean xsdValidate = false;
    String schemaSource = null;

    for (int i = 0; i < args.length; i++) {
```

```
        if (args[i].equals("-dtd"))  {
            dtdValidate = true;
        }
        else if (args[i].equals("-xsd")) {
            xsdValidate = true;
        }
        else if (args[i].equals("-xsdss")) {
            if (i == args.length - 1) {
                usage();
            }
            xsdValidate = true;
            schemaSource = args[++i];
        }
        else {
            filename = args[i];
            if (i != args.length - 1) {
                usage();
            }
        }
    }

    if (filename == null) {
        usage();
    }

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

    dbf.setNamespaceAware(true);
    dbf.setValidating(dtdValidate || xsdValidate);

    // ...

    DocumentBuilder db = dbf.newDocumentBuilder();
    Document doc = db.parse(new File(filename));
}
```

As you can see, command line arguments are set up so that you can inform `DOMEcho` to perform validation against either a DTD or an XML Schema, and the factory is configured to be name space aware and to perform whichever type of validation the user specifies.

Note - JAXP-conformant parsers are not required to support all combinations of those options, even though the reference parser does. If you specify an invalid combination of options, the factory generates a `ParserConfigurationException` when you attempt to obtain a parser instance.

More information about how to use name spaces and validation is provided in Validating with XML Schema, in which the code that is missing from the above extract will be described.

Handling Validation Errors

The default response to a validation error, as dictated by the SAX standard, is to do nothing. The JAXP standard requires throwing SAX exceptions, so you use exactly the same error-handling mechanisms as you use for a SAX application. In particular, you use the `DocumentBuilder` class's `setErrorHandler` method to supply it with an object that implements the SAX `ErrorHandler` interface.

Note - `DocumentBuilder` also has a `setEntityResolver` method you can use.

The following code configures the document builder to use the error handler defined in Handle Errors.

```
DocumentBuilder db = dbf.newDocumentBuilder();
OutputStreamWriter errorWriter = new OutputStreamWriter(System.err,
                                                 outputEncoding);
db.setErrorHandler(new MyErrorHandler (new PrintWriter(errorWriter, true)));
Document doc = db.parse(new File(filename));
```

The code you have seen so far has set up the document builder, and configured it to perform validation upon request. Error handling is also in place. However, `DOMEcho` does not do anything yet. In the next section, you will see how to display the DOM structure and begin to explore it. For example, you will see what entity references and CDATA sections look like in the DOM. And perhaps most importantly, you will see how text nodes (which contain the actual data) reside under element nodes in a DOM.

Displaying the DOM Nodes

To create or manipulate a DOM, it helps to have a clear idea of how the nodes in a DOM are structured. This section of the tutorial exposes the internal structure of a DOM, so that you can see what it contains. The `DOMEcho` example does this by echoing the DOM nodes, and then printing them out onscreen, with the appropriate indentation to make the node hierarchy apparent. The specification of these node types can be found in the DOM Level 2 Core Specification, under the specification for `Node`. Table 3-1 below is adapted from that specification.

Table 3-1 Node Types

Node	nodeName	nodeValue	Attributes
`Attr`	Name of attribute	Value of attribute	null
`CDATASection`	`#cdata-section`	Content of the CDATA section	null
`Comment`	`#comment`	Content of the comment	null
`Document`	`#document`	null	null
`DocumentFragment`	`#documentFragment`	null	null
`DocumentType`	Document Type name	null	null
`Element`	Tag name	null	null
`Entity`	Entity name	null	null

`EntityReference`	Name of entity referenced	null	null
`Notation`	Notation name	null	null
`ProcessingInstruction`	Target	Entire content excluding the target	null
`Text`	`#text`	Content of the text node	null

The information in this table is extremely useful; you will need it when working with a DOM, because all these types are intermixed in a DOM tree.

Obtaining Node Type Information

The DOM node element type information is obtained by calling the various methods of the `org.w3c.dom.Node` class. The node attributes by exposed by `DOMEcho` are echoed by the following code.

```
private void printlnCommon(Node n) {
    out.print(" nodeName=\"" + n.getNodeName() + "\"");

    String val = n.getNamespaceURI();
    if (val != null) {
        out.print(" uri=\"" + val + "\"");
    }

    val = n.getPrefix();

    if (val != null) {
        out.print(" pre=\"" + val + "\"");
    }

    val = n.getLocalName();
    if (val != null) {
        out.print(" local=\"" + val + "\"");
    }

    val = n.getNodeValue();
    if (val != null) {
        out.print(" nodeValue=");
        if (val.trim().equals("")) {
            // Whitespace
            out.print("[WS]");
        }
        else {
            out.print("\"" + n.getNodeValue() + "\"");
        }
    }
    out.println();
}
```

Every DOM node has at least a type, a name, and a value, which might or might not be empty. In the example above, the `Node` interface's `getNamespaceURI()`, `getPrefix()`, `getLocalName()`, and `getNodeValue()` methods return and print the echoed node's namespace URI, namespace prefix, local qualified name and value. Note that the `trim()` method is called on the value returned by `getNodeValue()` to establish whether the node's value is empty white space and print a message accordingly.

For the full list of `Node` methods and the different information they return, see the API documentation for `Node`.

Next, a method is defined to set the indentation for the nodes when they are printed, so that the node hierarchy will be easily visible.

```
private void outputIndentation() {
    for (int i = 0; i < indent; i++) {
        out.print(basicIndent);
    }
}
```

The `basicIndent` constant to define the basic unit of indentation used when `DOMEcho` displays the node tree hierarchy, is defined by adding the following highlighted lines to the `DOMEcho` constructor class.

```
public class DOMEcho {
    static final String outputEncoding = "UTF-8";

    private PrintWriter out;
    private int indent = 0;
    private final String basicIndent = "  ";

    DOMEcho(PrintWriter out) {
        this.out = out;
    }
}
```

As was the case with the error handler defined in Handle Errors, the `DOMEcho` program will create its output as `PrintWriter` instances.

Lexical Controls

Lexical information is the information you need to reconstruct the original syntax of an XML document. Preserving lexical information is important in editing applications, where you want to save a document that is an accurate reflection of the original-complete with comments, entity references, and any CDATA sections it may have included at the outset.

Most applications, however, are concerned only with the content of the XML structures. They can afford to ignore comments, and they do not care whether data was coded in a CDATA section or as plain text, or whether it included an entity reference. For such applications, a minimum of lexical information is desirable, because it simplifies the number and kind of DOM nodes that the application must be prepared to examine.

The following `DocumentBuilderFactory` methods give you control over the lexical information you see in the DOM.

`setCoalescing()`

To convert `CDATA` nodes to `Text` nodes and append to an adjacent `Text` node (if any).

`setExpandEntityReferences()`

To expand entity reference nodes.

`setIgnoringComments()`

To ignore comments.

`setIgnoringElementContentWhitespace()`

To ignore whitespace that is not a significant part of element content.

The default values for all these properties is false, which preserves all the lexical information necessary to reconstruct the incoming document in its original form. Setting them to true lets you construct the simplest possible DOM so that the application can focus on the data's semantic content without having to worry about lexical syntax details. Table 3-2 summarizes the effects of the settings.

Table 3-2 Lexical Control Settings

API	Preserve Lexical Info	Focus on Content
`setCoalescing()`	False	True
`setExpandEntityReferences()`	False	True
`setIgnoringComments()`	False	True
`setIgnoringElementContentWhitespace()`	False	True

The implementation of these methods in the main method of the `DomEcho` example is shown below.

```
// ...

dbf.setIgnoringComments(ignoreComments);
dbf.setIgnoringElementContentWhitespace(ignoreWhitespace);
dbf.setCoalescing(putCDATAIntoText);
dbf.setExpandEntityReferences(!createEntityRefs);

// ...
```

The boolean variables `ignoreComments`, `ignoreWhitespace`, `putCDATAIntoText`, and `createEntityRefs` are declared at the beginning of the main method code, and they are set by command line arguments when `DomEcho` is run.

```
public static void main(String[] args) throws Exception {
    // ...

    boolean ignoreWhitespace = false;
    boolean ignoreComments = false;
    boolean putCDATAIntoText = false;
```

```
    boolean createEntityRefs = false;

    for (int i = 0; i < args.length; i++) {
        if (...) {  // Validation arguments here
           // ...
        }
        else if (args[i].equals("-ws")) {
            ignoreWhitespace = true;
        }
        else if (args[i].startsWith("-co")) {
            ignoreComments = true;
        }
        else if (args[i].startsWith("-cd")) {
            putCDATAIntoText = true;
        }
        else if (args[i].startsWith("-e")) {
            createEntityRefs = true;

            // ...
        }
        else {
            filename = args[i];

            // Must be last arg
            if (i != args.length - 1) {
                usage();
            }
        }
    }

    // ...
}
```

Printing DOM Tree Nodes

The `DomEcho` application allows you to see the structure of a DOM, and demonstrates what nodes make up the DOM and how they are arranged. Generally, the vast majority of nodes in a DOM tree will be `Element` and `Text` nodes.

Note - Text nodes exist **under** element nodes in a DOM, and data is always stored in text nodes. Perhaps the most common error in DOM processing is to navigate to an element node and expect it to contain the data that is stored in that element. Not so! Even the simplest element node has a text node under it that contains the data.

The code to print out the DOM tree nodes with the appropriate indentation is shown below.

```
private void echo(Node n) {
    outputIndentation();
    int type = n.getNodeType();

    switch (type) {
        case Node.ATTRIBUTE_NODE:
            out.print("ATTR:");
            printlnCommon(n);
            break;

        case Node.CDATA_SECTION_NODE:
```

```
            out.print("CDATA:");
            printlnCommon(n);
            break;

        case Node.COMMENT_NODE:
            out.print("COMM:");
            printlnCommon(n);
            break;

        case Node.DOCUMENT_FRAGMENT_NODE:
            out.print("DOC_FRAG:");
            printlnCommon(n);
            break;

        case Node.DOCUMENT_NODE:
            out.print("DOC:");
            printlnCommon(n);
            break;

        case Node.DOCUMENT_TYPE_NODE:
            out.print("DOC_TYPE:");
            printlnCommon(n);
            NamedNodeMap nodeMap = ((DocumentType)n).getEntities();
            indent += 2;
            for (int i = 0; i < nodeMap.getLength(); i++) {
                Entity entity = (Entity)nodeMap.item(i);
                echo(entity);
            }
            indent -= 2;
            break;

        case Node.ELEMENT_NODE:
            out.print("ELEM:");
            printlnCommon(n);

            NamedNodeMap atts = n.getAttributes();
            indent += 2;
            for (int i = 0; i < atts.getLength(); i++) {
                Node att = atts.item(i);
                echo(att);
            }
            indent -= 2;
            break;

        case Node.ENTITY_NODE:
            out.print("ENT:");
            printlnCommon(n);
            break;

        case Node.ENTITY_REFERENCE_NODE:
            out.print("ENT_REF:");
            printlnCommon(n);
            break;

        case Node.NOTATION_NODE:
            out.print("NOTATION:");
            printlnCommon(n);
            break;

        case Node.PROCESSING_INSTRUCTION_NODE:
            out.print("PROC_INST:");
            printlnCommon(n);
            break;
```

```
            case Node.TEXT_NODE:
                out.print("TEXT:");
                printlnCommon(n);
                break;

            default:
                out.print("UNSUPPORTED NODE: " + type);
                printlnCommon(n);
                break;
        }

        indent++;
        for (Node child = n.getFirstChild(); child != null;
             child = child.getNextSibling()) {
            echo(child);
        }
        indent--;
    }
```

This code first of all uses switch statements to print out the different node types and any possible child nodes, with the appropriate indentation.

Node attributes are not included as children in the DOM hierarchy. They are instead obtained via the `Node` interface's `getAttributes` method.

The `DocType` interface is an extension of `w3c.org.dom.Node`. It defines the `getEntities` method, which you use to obtain `Entity` nodes - the nodes that define entities. Like `Attribute` nodes, `Entity` nodes do not appear as children of DOM nodes.

Node Operations

This section takes a quick look at some of the operations you might want to apply to a DOM.

- Creating nodes
- Traversing nodes
- Searching for nodes
- Obtaining node content
- Creating attributes
- Removing and changing nodes
- Inserting nodes

Creating Nodes

You can create different types nodes using the methods of the `Document` interface. For example, `createElement`, `createComment`, `createCDATAsection`, `createTextNode`, and so on. The full list of methods for creating different nodes is provided in the API documentation for `org.w3c.dom.Document`.

Traversing Nodes

The `org.w3c.dom.Node` interface defines a number of methods you can use to traverse nodes, including `getFirstChild`, `getLastChild`, `getNextSibling`, `getPreviousSibling`, and `getParentNode`. Those operations are sufficient to get from anywhere in the tree to any

other location in the tree.

Searching for Nodes

When you are searching for a node with a particular name, there is a bit more to take into account. Although it is tempting to get the first child and inspect it to see whether it is the right one, the search must account for the fact that the first child in the sub-list could be a comment or a processing instruction. If the XML data has not been validated, it could even be a text node containing ignorable whitespace.

In essence, you need to look through the list of child nodes, ignoring the ones that are of no concern and examining the ones you care about. Here is an example of the kind of routine you need to write when searching for nodes in a DOM hierarchy. It is presented here in its entirety (complete with comments) so that you can use it as a template in your applications.

```
/**
 * Find the named subnode in a node's sublist.
 * <ul>
 * <li>Ignores comments and processing instructions.
 * <li>Ignores TEXT nodes (likely to exist and contain
 *         ignorable whitespace, if not validating.
 * <li>Ignores CDATA nodes and EntityRef nodes.
 * <li>Examines element nodes to find one with
 *        the specified name.
 * </ul>
 * @param name  the tag name for the element to find
 * @param node  the element node to start searching from
 * @return the Node found
 */
public Node findSubNode(String name, Node node) {
    if (node.getNodeType() != Node.ELEMENT_NODE) {
        System.err.println("Error: Search node not of element type");
        System.exit(22);
    }

    if (! node.hasChildNodes()) return null;

    NodeList list = node.getChildNodes();
    for (int i=0; i < list.getLength(); i++) {
        Node subnode = list.item(i);
        if (subnode.getNodeType() == Node.ELEMENT_NODE) {
           if (subnode.getNodeName().equals(name))
               return subnode;
        }
    }
    return null;
}
```

For a deeper explanation of this code, see Increasing the Complexity in When to Use DOM. Note, too, that you can use APIs described in Lexical Controls to modify the kind of DOM the parser constructs. The nice thing about this code, though, is that it will work for almost any DOM.

Obtaining Node Content

When you want to get the text that a node contains, you again need to look through the list of child nodes, ignoring entries that are of no concern and accumulating the text you find in `TEXT` nodes, `CDATA` nodes, and `EntityRef` nodes. Here is an example of the kind of routine you can use for that process.

```
/**
```

```
  * Return the text that a node contains. This routine:
  * <ul>
  * <li>Ignores comments and processing instructions.
  * <li>Concatenates TEXT nodes, CDATA nodes, and the results of
  *     recursively processing EntityRef nodes.
  * <li>Ignores any element nodes in the sublist.
  *     (Other possible options are to recurse into element
  *      sublists or throw an exception.)
  * </ul>
  * @param    node  a  DOM node
  * @return   a String representing its contents
  */
public String getText(Node node) {
    StringBuffer result = new StringBuffer();
    if (! node.hasChildNodes()) return "";

    NodeList list = node.getChildNodes();
    for (int i=0; i < list.getLength(); i++) {
        Node subnode = list.item(i);
        if (subnode.getNodeType() == Node.TEXT_NODE) {
            result.append(subnode.getNodeValue());
        }
        else if (subnode.getNodeType() == Node.CDATA_SECTION_NODE) {
            result.append(subnode.getNodeValue());
        }
        else if (subnode.getNodeType() == Node.ENTITY_REFERENCE_NODE) {
            // Recurse into the subtree for text
            // (and ignore comments)
            result.append(getText(subnode));
        }
    }

    return result.toString();
}
```

For a deeper explanation of this code, see Increasing the Complexity in When to Use DOM. Again, you can simplify this code by using the APIs described in Lexical Controls to modify the kind of DOM the parser constructs. But the nice thing about this code is that it will work for almost any DOM.

Creating Attributes

The `org.w3c.dom.Element` interface, which extends Node, defines a `setAttribute` operation, which adds an attribute to that node. (A better name from the Java platform standpoint would have been `addAttribute`. The attribute is not a property of the class, and a new object is created.) You can also use the `Document`'s `createAttribute` operation to create an instance of `Attribute` and then use the `setAttributeNode` method to add it.

Removing and Changing Nodes

To remove a node, you use its parent Node's `removeChild` method. To change it, you can use either the parent node's `replaceChild` operation or the node's `setNodeValue` operation.

Inserting Nodes

The important thing to remember when creating new nodes is that when you create an element node, the only data you specify is a name. In effect, that node gives you a hook to hang things on. You hang an item on the hook by adding to its list of child nodes. For example, you might add a text

node, a CDATA node, or an attribute node. As you build, keep in mind the structure you have seen in this tutorial. Remember: Each node in the hierarchy is extremely simple, containing only one data element.

Running the DOMEcho Sample

To run the DOMEcho sample, follow the steps below.

1. **Navigate to the `samples` directory.**`% cd` *`install-dir`*`/jaxp-1_4_2-`*`release-date`*`/samples.`
2. **Compile the example class.**`% javac dom/*`
3. **Run the `DOMEcho` program on an XML file.**

   Choose one of the XML files in the `data` directory and run the `DOMEcho` program on it. Here, we have chosen to run the program on the file `personal-schema.xml`.

   ```
   % java dom/DOMEcho data/personal-schema.xml
   ```

   The XML file `personal-schema.xml` contains the personnel files for a small company. When you run the `DOMEcho` program on it, you should see the following output.

   ```
   DOC: nodeName="#document"
    ELEM: nodeName="personnel"
          local="personnel"
    TEXT: nodeName="#text"
          nodeValue=[WS]
    ELEM: nodeName="person"
          local="person"
    ATTR: nodeName="id"
          local="id"
          nodeValue="Big.Boss"
    TEXT: nodeName="#text"
          nodeValue=[WS]
    ELEM: nodeName="name"
          local="name"
    ELEM: nodeName="family"
          local="family"
    TEXT: nodeName="#text"
          nodeValue="Boss"
    TEXT: nodeName="#text"
          nodeValue=[WS]
    ELEM: nodeName="given"
          local="given"
    TEXT: nodeName="#text"
          nodeValue="Big"
    TEXT: nodeName="#text"
          nodeValue=[WS]
    ELEM: nodeName="email"
          local="email"
    TEXT: nodeName="#text"
          nodeValue="chief@foo.example.com"
    TEXT: nodeName="#text"
          nodeValue=[WS]
    ELEM: nodeName="link"
          local="link"
    ATTR: nodeName="subordinates"
          local="subordinates"
          nodeValue="one.worker two.worker
                     three.worker four.worker
                     five.worker"
    TEXT: nodeName="#text"
   ```

```
       nodeValue=[WS]
 TEXT: nodeName="#text"
       nodeValue=[WS]
 ELEM: nodeName="person"
       local="person"
 ATTR: nodeName="id"
       local="id"
       nodeValue="one.worker"
 TEXT: nodeName="#text"
       nodeValue=[WS]
 ELEM: nodeName="name"
       local="name"
 ELEM: nodeName="family"
       local="family"
 TEXT: nodeName="#text"
       nodeValue="Worker"
 TEXT: nodeName="#text"
       nodeValue=[WS]
 ELEM: nodeName="given"
       local="given"
 TEXT: nodeName="#text"
       nodeValue="One"
 TEXT: nodeName="#text"
       nodeValue=[WS]
 ELEM: nodeName="email"
       local="email"
 TEXT: nodeName="#text"
       nodeValue="one@foo.example.com"
 TEXT: nodeName="#text"
       nodeValue=[WS]
 ELEM: nodeName="link"
       local="link"
 ATTR: nodeName="manager"
       local="manager"
       nodeValue="Big.Boss"
 TEXT: nodeName="#text"
       nodeValue=[WS]

[...]
```

As you can see, `DOMEcho` prints out all the nodes for the different elements in the document, with the correct indentation to show the node hierarchy.

Validating with XML Schema

This section looks at the process of XML Schema validation. Although a full treatment of XML Schema is beyond the scope of this tutorial, this section shows you the steps you take to validate an XML document using an XML Schema definition. (To learn more about XML Schema, you can review the online tutorial, XML Schema Part 0: Primer. At the end of this section, you will also learn how to use an XML Schema definition to validate a document that contains elements from multiple namespaces.

Overview of the Validation Process

To be notified of validation errors in an XML document, the following must be true:

- The factory must configured, and the appropriate error handler set.
- The document must be associated with at least one schema, and possibly more.

Configuring the `DocumentBuilder` Factory

It is helpful to start by defining the constants you will use when configuring the factory. These are the same constants you define when using XML Schema for SAX parsing, and they are declared at the beginning of the `DOMEcho` example program.

```
static final String JAXP_SCHEMA_LANGUAGE =
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
static final String W3C_XML_SCHEMA =
    "http://www.w3.org/2001/XMLSchema";
```

Next, you configure `DocumentBuilderFactory` to generate a namespace-aware, validating parser that uses XML Schema. This is done by calling the `setValidating` method on the `DocumentBuilderFactory` instance `dbf`, that was created in Instantiate the Factory.

```
// ...

dbf.setNamespaceAware(true);
dbf.setValidating(dtdValidate || xsdValidate);

if (xsdValidate) {
    try {
        dbf.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
    }
    catch (IllegalArgumentException x) {
        System.err.println("Error: JAXP DocumentBuilderFactory attribute "
                           + "not recognized: " + JAXP_SCHEMA_LANGUAGE);
        System.err.println("Check to see if parser conforms to JAXP 1.2 spec.");
        System.exit(1);
    }
}

// ...
```

Because JAXP-compliant parsers are not namespace-aware by default, it is necessary to set the property for schema validation to work. You also set a factory attribute to specify the parser language to use. (For SAX parsing, on the other hand, you set a property on the parser generated by the factory).

Associating a Document with a Schema

Now that the program is ready to validate with an XML Schema definition, it is necessary only to ensure that the XML document is associated with (at least) one. There are two ways to do that:

- With a schema declaration in the XML document
- By specifying the schema(s) to use in the application

Note - When the application specifies the schema(s) to use, it overrides any schema declarations in the document.

To specify the schema definition in the document, you would create XML like this:

```
<documentRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:noNamespaceSchemaLocation='YourSchemaDefinition.xsd'> [...]
```

The first attribute defines the XML namespace (`xmlns`) prefix, `xsi`, which stands for "XML Schema instance." The second line specifies the schema to use for elements in the document that do not have a namespace prefix-that is, for the elements you typically define in any simple, uncomplicated XML document. (You will see how to deal with multiple namespaces in the next section.)

You can also specify the schema file in the application, which is the case for `DOMEcho`.

```
static final String JAXP_SCHEMA_SOURCE =
    "http://java.sun.com/xml/jaxp/properties/schemaSource";

// ...

dbf.setValidating(dtdValidate || xsdValidate);
if (xsdValidate) {
    // ...
}

if (schemaSource != null) {
    dbf.setAttribute(JAXP_SCHEMA_SOURCE, new File(schemaSource));
}
```

Here, too, there are mechanisms at your disposal that will let you specify multiple schemas. We will take a look at those next.

Validating with Multiple Namespaces

Namespaces let you combine elements that serve different purposes in the same document without having to worry about overlapping names.

Note - The material discussed in this section also applies to validating when using the SAX parser. You are seeing it here, because at this point you have learned enough about namespaces for the discussion to make sense.

To contrive an example, consider an XML data set that keeps track of personnel data. The data set may include information from a tax declaration form as well as information from the employee's hiring form, with both elements named `form` in their respective schemas.

If a prefix is defined for the tax namespace, and another prefix defined for the hiring namespace, then the personnel data could include segments like the following.

```
<employee id="...">
  <name>....</name>
  <tax:form>
     ...w2 tax form data...
  </tax:form>
  <hiring:form>
     ...employment history, etc....
  </hiring:form>
</employee>
```

The contents of the `tax:form` element would obviously be different from the contents of the `hiring:form` element and would have to be validated differently.

Note, too, that in this example there is a default namespace that the unqualified element names `employee` and `name` belong to. For the document to be properly validated, the schema for that

namespace must be declared, as well as the schemas for the `tax` and `hiring` namespaces.

Note - The default namespace is actually a specific namespace. It is defined as the "namespace that has no name." So you cannot simply use one namespace as your default this week, and another namespace as the default later. This "unnamed namespace" (or "null namespace") is like the number zero. It does not have any value to speak of (no name), but it is still precisely defined. So a namespace that does have a name can never be used as the default namespace.

When parsed, each element in the data set will be validated against the appropriate schema, as long as those schemas have been declared. Again, the schemas can be declared either as part of the XML data set or in the program. (It is also possible to mix the declarations. In general, though, it is a good idea to keep all the declarations together in one place.)

Declaring the Schemas in the XML Data Set

To declare the schemas to use for the preceding example in the data set, the XML code would look something like the following.

```
<documentRoot
  xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
    "employeeDatabase.xsd"
  xsi:schemaLocation=
  "http://www.irs.gov.example.com/
   fullpath/w2TaxForm.xsd
   http://www.ourcompany.example.com/
   relpath/hiringForm.xsd"
  xmlns:tax=
    "http://www.irs.gov.example.com/"
  xmlns:hiring=
    "http://www.ourcompany.example.com/"
>
```

The `noNamespaceSchemaLocation` declaration is something you have seen before, as are the last two entries, which define the namespace prefixes `tax` and `hiring`. What is new is the entry in the middle, which defines the locations of the schemas to use for each namespace referenced in the document.

The `xsi:schemaLocation` declaration consists of entry pairs, where the first entry in each pair is a fully qualified URI that specifies the namespace, and the second entry contains a full path or a relative path to the schema definition. In general, fully qualified paths are recommended. In that way, only one copy of the schema will tend to exist.

Note that you cannot use the namespace prefixes when defining the schema locations. The `xsi:schemaLocation` declaration understands only namespace names and not prefixes.

Declaring the Schemas in the Application

To declare the equivalent schemas in the application, the code would look something like the following.

```
static final String employeeSchema = "employeeDatabase.xsd";
static final String taxSchema = "w2TaxForm.xsd";
static final String hiringSchema = "hiringForm.xsd";
```

```
static final String[] schemas = {
    employeeSchema,
    taxSchema,
    hiringSchema,
};

static final String JAXP_SCHEMA_SOURCE =
    "http://java.sun.com/xml/jaxp/properties/schemaSource";

// ...

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance()

// ...

factory.setAttribute(JAXP_SCHEMA_SOURCE, schemas);
```

Here, the array of strings that points to the schema definitions (`.xsd` files) is passed as the argument to the `factory.setAttribute` method. Note the differences from when you were declaring the schemas to use as part of the XML data set.

- There is no special declaration for the default (unnamed) schema.
- You do not specify the namespace name. Instead, you only give pointers to the `.xsd` files.

To make the namespace assignments, the parser reads the `.xsd` files, and finds in them the name of the target namespace they apply to. Because the files are specified with URIs, the parser can use an `EntityResolver` (if one has been defined) to find a local copy of the schema.

If the schema definition does not define a target namespace, then it applies to the default (unnamed, or null) namespace. So, in our example, you would expect to see these target namespace declarations in the schemas:

- A string that points to the URI of the schema
- An `InputStream` with the contents of the schema
- A SAX `InputSource`
- A File
- An array of Objects, each of which is one of the types defined here

An array of Objects can be used only when the schema language has the ability to assemble a schema at runtime. Also, when an array of Objects is passed it is illegal to have two schemas that share the same namespace.

Running the `DOMEcho` Sample With Schema Validation

To run the `DOMEcho` sample with schema validation, follow the steps below.

1. **Navigate to the `samples` directory.**`% cd` *`install-dir`*`/jaxp-1_4_2-`*`release-date`*`/samples.`
2. **Compile the example class, using the class path you have just set.**`% javac dom/*`
3. **Run the `DOMEcho` program on an XML file, specifying schema validation.**

   Choose one of the XML files in the `data` directory and run the `DOMEcho` program on it with the `-xsd` option specified. Here, we have chosen to run the program on the file `personal-schema.xml`.

   ```
   % java dom/DOMEcho -xsd data/personal-schema.xml
   ```

As you saw in Configuring the Factory, the `-xsd` option tells `DOMEcho` to perform validation against the XML schema that is defined in the `personal-schema.xml` file. In this case, the schema is the file `personal.xsd`, which is also located in the `sample/data` directory.

4. **Open `personal-schema.xml` in a text editor and delete the schema declaration.**

   Remove the following from the opening `<personnel>` tag.

   ```
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation='personal.xsd'
   ```

   Do not forget to save the file.

5. **Run `DOMEcho` again, specifying the `-xsd` option once more.**`% java dom/DOMEcho -xsd data/personal-schema.xml`

   This time, you will see a stream of errors.

6. **Run `DOMEcho` one more time, this time specifying the `-xsdss` option and specifying the schema definition file.**

   As you saw in Configuring the Factory, the `-xsdss` option tells `DOMEcho` to perform validation against an XML schema definition that is specified when the program is run. Once again, use the file `personal.xsd`.

   ```
   % java dom/DOMEcho -xsdss data/personal.xsd data/personal-schema.xml
   ```

   You will see the same output as before, meaning that the XML file has been successfully validated against the schema.

Further Information

For further information on the W3C Document Object Model (DOM), see The DOM standard page.

For more information on schema-based validation mechanisms, see the following.

- The W3C standard validation mechanism, XML Schema
- RELAX NG's regular-expression based validation mechanism
- Schematron's assertion-based validation mechanism