



CHAPTER 23

org.w3c.dom

Package **org.w3c.dom**

Java 1.4

This package defines the Java binding to the core and XML modules of the DOM Level 2 API defined by the World Wide Web Consortium (W3C). DOM stands for Document Object Model, and the DOM API defines a way to represent an XML document as a tree of nodes. It includes methods that allow document trees to be traversed, examined, modified, and built from scratch. **Node** is the central interface of the package. All nodes in a document tree implement this interface, and it defines the basic methods for traversing and modifying the tree of nodes. Most of the other interfaces in the package are extensions of **Node** that represent specific types of XML content. The most important and commonly used of these sub-interfaces are **Document**, **Element** and **Text**. A **Document** object serves as the root of the document tree and defines methods for searching the tree for elements with a specified tag name or ID attribute. The **Element** interface represents an XML element or tag and has methods for manipulating the element's attributes. The **Text** interface represents a run of plain text within an **Element**, and has methods for querying or altering that text. **NodeList** and **DOMImplementation** do not extend **Node** but are also important interfaces.

Interfaces:

```
public interface Attr extends Node;  
public interface CDATASection extends Text;  
public interface CharacterData extends Node;  
public interface Comment extends CharacterData;  
public interface Document extends Node;  
public interface DocumentFragment extends Node;  
public interface DocumentType extends Node;  
public interface DOMImplementation;  
public interface Element extends Node;  
public interface Entity extends Node;  
public interface EntityReference extends Node;  
public interface NamedNodeMap;
```

Package *org.w3c.dom*

```
public interface Node;  
public interface NodeList;  
public interface Notation extends Node;  
public interface ProcessingInstruction extends Node;  
public interface Text extends CharacterData;
```

Exception:

public class **DOMException** extends RuntimeException;

Attr

Java 1.4

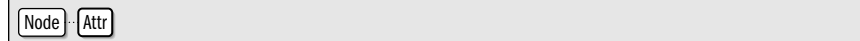
org.w3c.dom

An **Attr** object represents an attribute of an **Element** node. **Attr** objects are associated with **Element** nodes, but are not directly part of the document tree: the **getParentNode()** method of an **Attr** object always returns null. Use **getOwnerElement()** to determine which **Element** an **Attr** is part of. You can obtain an **Attr** object by calling the **getAttributeNode()** method of **Element**, or you can obtain a **NamedNodeMap** of all **Attr** objects for an element with the **getAttributes()** method of **Node**.

getName() returns the name of the attribute. **getValue()** returns the attribute value as a string. **getSpecified()** returns **true** if the attribute was explicitly specified in the source document through a call to **setValue()**, and returns **false** if the attribute represents a default obtained from a DTD or other schema.

XML allows attributes to contain text and entity references. The **getValue()** method returns the attribute value as a single string. If you want to know the precise composition of the attribute however, you can examine the children of the **Attr** node: they may consist of **Text** and/or **EntityReference** nodes.

In most cases the easiest way to work with attributes is with the **getAttribute()** and **setAttribute()** methods of the **Element** interface. These methods avoid the use of **Attr** nodes altogether.



```
public interface Attr extends Node {  
    // Public Instance Methods  
    public abstract String getName();  
    public abstract org.w3c.dom.Element getOwnerElement();  
    public abstract boolean getSpecified();  
    public abstract String getValue();  
    public abstract void setValue(String value) throws DOMException;  
}
```

Passed To: javax.imageio.metadata.IIOMetadataNode.{removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}, org.w3c.dom.Element.{removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}

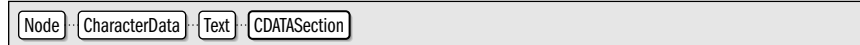
Returned By: javax.imageio.metadata.IIOMetadataNode.{getAttributeNode(), getAttributeNodeNS(), removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}, org.w3c.dom.Document.{createAttribute(), createAttributeNS()}, org.w3c.dom.Element.{getAttributeNode(), getAttributeNodeNS(), removeAttributeNode(), setAttributeNode(), setAttributeNodeNS()}

CDATASection

Java 1.4

org.w3c.dom

This interface represents a CDATA section in an XML document. **CDATASection** is a sub-interface of **Text** and does not define any methods of its own. The content of the CDATA section is available through the **getNodeValue()** method inherited from **Node**, or through the **getData()** method inherited from **CharacterData**. Although **CDATASection** nodes can often be treated in the same way as **Text** nodes, note that the **Node.normalize()** method does not merge adjacent CDATA sections.



```
public interface CDATASection extends Text {
}
```

Returned By: org.w3c.dom.Document.createCDATASection()

CharacterData

Java 1.4

org.w3c.dom

This interface is a generic one that is extended by **Text**, **CDATASection** (which extends **Text**) and **Comment**. Any node in a document tree that implements **CharacterData** also implements one of these more specific types. This interface exists simply to group the string manipulation methods that these text-related node types all share.

The **CharacterData** interface defines a mutable string. **getData()** returns the “character data” as a **String** object, and **setData()** allows it to be set from a **String** object. **getLength()** returns the number of characters of character data, and **substringData()** returns just the specified portion of the data as a string. The **appendData()**, **deleteData()**, **insertData()**, and **replaceData()** methods mutate the data by appending a string to the end, deleting region, inserting a string at the specified location, and replacing a region with a specified string.



```
public interface CharacterData extends Node {
// Public Instance Methods
    public abstract void appendData(String arg) throws DOMException;
    public abstract void deleteData(int offset, int count) throws DOMException;
    public abstract String getData() throws DOMException;
    public abstract int getLength();
    public abstract void insertData(int offset, String arg) throws DOMException;
    public abstract void replaceData(int offset, int count, String arg) throws DOMException;
    public abstract void setData(String data) throws DOMException;
    public abstract String substringData(int offset, int count) throws DOMException;
}
```

Implementations: Comment, Text

Comment

Java 1.4

org.w3c.dom

A **Comment** node represents a comment in an XML document. The content of the comment (i.e., the text between **<!--** and **-->**) is available with the **getData()** method inher-

Comment

ited from `CharacterData`, or through the `getNodeValue()` method inherited from `Node`. This content may be manipulated using the various methods inherited from `CharacterData`.



```
public interface Comment extends CharacterData {  
}
```

Returned By: `org.w3c.dom.Document.createComment()`

Document

Java 1.4

org.w3c.dom

This interface represents a DOM document, and an object that implements this interface serves as the root of a DOM document tree. Most of the methods defined by the `Document` interface are “factory methods” that are used to create various types of nodes that can be inserted into this document. Note that there are two versions of the methods for creating attributes and elements. The methods with “NS” in their name are namespace-aware and require the attribute or element name to be specified as a combination of a namespace URI and a local name. You’ll notice that throughout the DOM API, methods with “NS” in their names are namespace-aware. Other important methods include the following.

`getElementsByTagName()` and its namespace-aware variant `getElementsByTagNameNS()` search the document tree for `Element` nodes that have the specified tag name and return a `NodeList` containing those matching nodes. The `Element` interface defines methods by the same names that search only within the sub-tree defined by an `Element`.

`getElementById()` is a related method that searches the document tree for a single element with the specified unique value for an ID attribute. This is useful when you use an ID attribute to uniquely identify certain tags within an XML document. Note that this method does not search for attributes that are named “id” or “ID”. It searches for attributes whose XML type (as declared in the document’s DTD) is ID. Such attributes are often named “id”, but this is not required.

An XML document must have a single root element. `getDocumentElement()` returns this `Element` object. Note, however that this does not mean that a `Document` node has only one child. It must have exactly one child that is an `Element`, but it can also have other children such as `Comment` and `ProcessingInstruction` nodes. The `getDoctype()` method returns the `DocumentType` object (or null if there isn’t one) that represents the document’s DTD. `getImplementation()` returns the `DOMImplementation` object that represents the DOM implementation that created this document tree.



```
public interface Document extends Node {  
// Public Instance Methods  
public abstract Attr createAttribute(String name) throws DOMException;  
public abstract Attr createAttributeNS(String namespaceURI, String qualifiedName) throws DOMException;  
public abstract CDATASection createCDATASection(String data) throws DOMException;  
public abstract Comment createComment(String data);  
public abstract DocumentFragment createDocumentFragment();  
public abstract org.w3c.dom.Element createElement(String tagName) throws DOMException;  
public abstract org.w3c.dom.Element createElementNS(String namespaceURI, String qualifiedName)  
throws DOMException;  
public abstract EntityReference createEntityReference(String name) throws DOMException;  
public abstract ProcessingInstruction createProcessingInstruction(String target, String data)  
throws DOMException;
```

```

public abstract Text createTextNode(String data);
public abstract DocumentType getDoctype();
public abstract org.w3c.dom.Element getDocumentElement();
public abstract org.w3c.dom.Element getElementById(String elementId);
public abstract NodeList getElementsByName(String tagName);
public abstract NodeList getElementsByNameNS(String namespaceURI, String localName);
public abstract DOMImplementation getImplementation();
public abstract Node importNode(Node importedNode, boolean deep) throws DOMException;
}

```

Implementations: org.w3c.dom.html.HTMLDocument

Returned By: javax.imageio.metadata.IIOMetadataNode.getOwnerDocument(),
 javax.xml.parsers.DocumentBuilder.{newDocument(), parse()}, DOMImplementation.createDocument(),
 Node.getOwnerDocument(), org.w3c.dom.html.HTMLFrameElement.getContentDocument(),
 org.w3c.dom.html.HTMLIFrameElement.getContentDocument(),
 org.w3c.dom.html.HTMLObjectElement.getContentDocument()

DocumentFragment

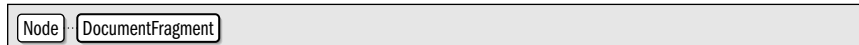
Java 1.4

org.w3c.dom

The DocumentFragment interface represents a portion—or fragment—of a document. More specifically, it represents one or more adjacent document nodes, and all of the descendants of each. DocumentFragment nodes are never part of a document tree, and getParentNode() always returns null. Although a DocumentFragment does not have a parent, it can have children, and you can use the inherited Node methods to add child nodes (or delete or replace them) to a DocumentFragment.

DocumentFragment nodes exhibit a special behavior that makes them quite useful: when a request is made to insert a DocumentFragment into a document tree, it is not the DocumentFragment node itself that is inserted, but each of the children of the DocumentFragment instead. This makes DocumentFragment useful as a temporary placeholder for a sequence of nodes that you wish to insert, all at once, into a document.

You can create a new, empty, DocumentFragment to work with by calling the createDocumentFragment() method of the desired Document.



```

public interface DocumentFragment extends Node {
}

```

Returned By: org.w3c.dom.Document.createDocumentFragment()

DocumentType

Java 1.4

org.w3c.dom

This interface represents the Document Type Declaration (DTD) of a document. Because the DTD is not part of the document itself, a DocumentType object is not part of DOM document tree, even though it extends the Node interface. If a Document has a DTD, then you may obtain the DocumentType object that represents it by calling the getDoctype() method of the Document object.

getName(), getPublicId(), getSystemId(), and getInternalSubset() all return strings (or null) that contain the name, public identifier, system identifier, and internal subset of the document type. getEntities() returns a read-only NamedNodeMap that represents the a name-to-value mapping for all internal and external general entities declared by the DTD. You can use this NamedNodeMap to lookup an Entity object by name. Similarly, getNotations()

DocumentType

returns a read-only `NamedNodeMap` that allows you to look up a `Notation` object declared in the DTD by name.

`DocumentType` does not provide access to the bulk of a DTD, which usually consists of element and attribute declarations. Future versions of the DOM API may provide more details.

| | |
|------|--------------|
| Node | DocumentType |
|------|--------------|

```
public interface DocumentType extends Node {  
    // Property Accessor Methods (by property name)  
    public abstract NamedNodeMap getEntities();  
    public abstract String getInternalSubset();  
    public abstract String getName();  
    public abstract NamedNodeMap getNotations();  
    public abstract String getPublicId();  
    public abstract String getSystemId();  
}
```

Passed To: `DOMImplementation.createDocument()`

Returned By: `org.w3c.dom.Document.getDoctype()`, `DOMImplementation.createDocumentType()`

DOMException

Java 1.4

`org.w3c.dom`

serializable unchecked

An instance of this class is thrown whenever an exception is raised by the DOM API. Unlike many Java APIs, the DOM API does not define specialized subclasses to define different categories of exceptions. Instead, a more specific exception type is specified by the public field `code`. The value of this field will be one of the constants defined by this class, which have the following meanings:

INDEX_SIZE_ERR

Indicates an out-of-bounds error for an array or string index.

DOMSTRING_SIZE_ERR

Indicates that a requested text is too big to fit into a `String` object. Exceptions of this type are intended for DOM implementations for other languages and should not occur in Java.

HIERARCHY_REQUEST_ERR

Indicates that an attempt was made to illegally place a node somewhere in the document tree hierarchy.

WRONG_DOCUMENT_ERR

Indicates that an attempt was made to use a node with a document that is different than the document that created the node.

INVALID_CHARACTER_ERR

Indicates that an illegal character is used (for example, in an element name).

NO_DATA_ALLOWED_ERR

Not currently used.

NO_MODIFICATION_ALLOWED_ERR

Indicates that an attempt was made to modify a node that is read-only and does not allow modifications. Entity, `EntityReference`, and `Notation` nodes, and all of their descendants, are read-only.

NOT_FOUND_ERR

Indicates that a node was not found where it was expected to be.

NOT_SUPPORTED_ERR

Indicates that a method or property is not supported in the current DOM implementation.

INUSE_ATTRIBUTE_ERR

Indicates that an attempt was made to associate an **Attr** with an **Element** when that **Attr** node was already associated with a different **Element** node.

INVALID_STATE_ERR

Indicates that an attempt was made to use an object that is not yet, or is no longer, in a state that allows such use.

SYNTAX_ERR

Indicates that a specified string contains a syntax error. Exceptions of this type are not raised by the core module of the DOM API described here.

INVALID_MODIFICATION_ERR

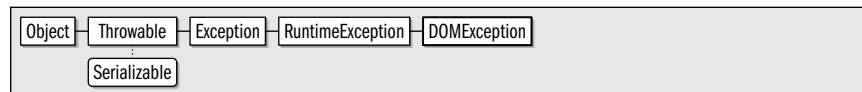
Exceptions of this type are not raised by the core module of the DOM API described here.

NAMESPACE_ERR

Indicates an error involving element or attribute namespaces.

INVALID_ACCESS_ERR

Indicates that an attempt was made to access an object in a way that is not supported by the implementation.



```

public class DOMException extends RuntimeException {
// Public Constructors
    public DOMException(short code, String message);
// Public Constants
    public static final short DOMSTRING_SIZE_ERR;           =2
    public static final short HIERARCHY_REQUEST_ERR;       =3
    public static final short INDEX_SIZE_ERR;              =1
    public static final short INUSE_ATTRIBUTE_ERR;          =10
    public static final short INVALID_ACCESS_ERR;          =15
    public static final short INVALID_CHARACTER_ERR;       =5
    public static final short INVALID_MODIFICATION_ERR;    =13
    public static final short INVALID_STATE_ERR;           =11
    public static final short NAMESPACE_ERR;               =14
    public static final short NO_DATA_ALLOWED_ERR;         =6
    public static final short NO_MODIFICATION_ALLOWED_ERR; =7
    public static final short NOT_FOUND_ERR;               =8
    public static final short NOT_SUPPORTED_ERR;           =9
    public static final short SYNTAX_ERR;                  =12
    public static final short WRONG_DOCUMENT_ERR;         =4
// Public Instance Fields
    public short code;
}
  
```

DOMException

Thrown By: Too many methods to list.

DOMImplementation

Java 1.4

org.w3c.dom

This interface defines methods that are global to an implementation of the DOM rather than specific to a particular `Document` object. Obtain a reference to the `DOMImplementation` object that represents your implementation by calling the `getImplementation()` method of any `Document` object. `createDocument()` returns a new, empty `Document` object which you can populate with nodes that you create using the `create` methods defined by the `Document` interface.

`hasFeature()` allows you to test whether your DOM implementation supports a specified version of a named feature, or module, of the DOM standard. This method should return `true` when you pass the feature name “core” and the version “1.0”, or when you pass the feature names “core” or “xml” and the version “2.0”. The DOM standard includes a number of optional modules, but the Java platform has not adopted the sub-packages of this package that define the API for those optional modules, and therefore the DOM implementation bundled with a Java implementation is not likely to support those modules.

The `javax.xml.parsers.DocumentBuilder` class provides another way to obtain the `DOMImplementation` object by calling its `getDOMImplementation()` object. It also defines a shortcut `newDocument()` method for creating empty `Document` objects to populate.

```
public interface DOMImplementation {  
    // Public Instance Methods  
    public abstract org.w3c.dom.Document createDocument(String namespaceURI, String qualifiedName,  
                                                         DocumentType doctype) throws DOMException;  
    public abstract DocumentType createDocumentType(String qualifiedName, String publicId, String systemId)  
        throws DOMException;  
    public abstract boolean hasFeature(String feature, String version);  
}
```

Implementations: `org.w3c.dom.css.DOMImplementationCSS`,
`org.w3c.dom.html.HTMLDOMImplementation`

Returned By: `javax.xml.parsers.DocumentBuilder.getDOMImplementation()`,
`org.w3c.dom.Document.getImplementation()`

Element

Java 1.4

org.w3c.dom

This interface represents an element (or tag) in an XML document. `getTagName()` returns the tagname of the element, including the namespace prefix if there is one. When working with namespaces, you will probably prefer to use the namespace-aware methods defined by the `Node` interface. Use `getNamespaceURI()` to get the namespace URI of the element, and use `getLocalName()` to the local name of the element within that namespace. You can also use `getPrefix()` to query the namespace prefix, or `setPrefix()` to change the namespace prefix (this does not change the namespace URI).

`Element` defines a `getElementsByTagName()` method and a corresponding namespace-aware `getElementsByTagNameNS()` method, which behave just like the methods of the same names on the `Document` object, except that they search for named elements only within the subtree rooted at this `Element`.

The remaining methods of the `Element` interface are for querying and setting attribute values, testing the existence of an attribute, and removing an attribute from the `Element`. There are a confusing number of methods to perform these four basic attribute

operations. If an attribute-related method has “NS” in its name, then it is namespace-aware. If it has “Node” in its name, then it works with `Attr` objects rather than with the simpler string representation of the attribute value. Attributes in XML documents may contain entity references. If your document includes entity references in attribute values, then you may need to use the `Attr` interface because the expansion of such an entity reference can result in a sub-tree of nodes beneath the `Attr` object. Whenever possible, however, it is much easier to work with the methods that treat attribute values as plain strings. Note also that in addition to the attribute methods defined by the `Element` interface you can also obtain a `NamedNodeMap` of `Attr` objects with the `getAttributes()` method of the `Node` interface.

Finally, note that `getAttribute()` and related methods and `hasAttribute()` and related methods return the value of or test for the existence of both explicitly specified attributes, and also attributes for which a default value is specified in the document DTD. If you need to determine whether an attribute was explicitly specified in the document, obtain its `Attr` object, and use its `getSpecified()` method.

| | |
|------|---------|
| Node | Element |
|------|---------|

```

public interface Element extends Node {
    // Public Instance Methods
    public abstract String getAttribute(String name);
    public abstract Attr getAttributeNode(String name);
    public abstract Attr getAttributeNodeNS(String namespaceURI, String localName);
    public abstract String getAttributeNS(String namespaceURI, String localName);
    public abstract NodeList getElementsByName(String name);
    public abstract NodeList getElementsByNameNS(String namespaceURI, String localName);
    public abstract String getTagName();
    public abstract boolean hasAttribute(String name);
    public abstract boolean hasAttributeNS(String namespaceURI, String localName);
    public abstract void removeAttribute(String name) throws DOMException;
    public abstract Attr removeAttributeNode(Attr oldAttr) throws DOMException;
    public abstract void removeAttributeNS(String namespaceURI, String localName) throws DOMException;
    public abstract void setAttribute(String name, String value) throws DOMException;
    public abstract Attr setAttributeNode(Attr newAttr) throws DOMException;
    public abstract Attr setAttributeNodeNS(Attr newAttr) throws DOMException;
    public abstract void setAttributeNS(String namespaceURI, String qualifiedName, String value)
        throws DOMException;
}

```

Implementations: javax.imageio.metadata.IIOMetadataNode, org.w3c.dom.html.HTMLElement

Passed To: org.w3c.dom.css.DocumentCSS.getOverrideStyle(),
org.w3c.dom.css.ViewCSS.getComputedStyle()

Returned By: Attr.getOwnerElement(), org.w3c.dom.Document.{createElement(),
createElementNS(), getDocumentElement(), getElementById() }

Entity

Java 1.4

org.w3c.dom

This interface represents an entity defined in an XML DTD. The name of the entity is specified by the `getNodeName()` method inherited from the `Node` interface. The entity content is represented by the child nodes of the `Entity` node. The methods defined by this interface return the public identifier and system identifier for external entities, and the notation name for unparsed entities. Note that `Entity` nodes and their children are not part of the document tree (and the `getParentNode()` method of an `Entity` always returns

Entity

null). Instead, a document may contain one or more references to an entity. See the `EntityReference` interface.

Entities are defined in the DTD (document type definition) of a document, either as part of an external DTD file, or as part of an “internal subset” that defines local entities that are specific to the current document. The `DocumentType` interface has a `getEntities()` method that returns a `NamedNodeMap` mapping entity names to `Entity` nodes. This is the only way to obtain an `Entity` object; because they are part of the DTD, `Entity` nodes never appear within the document tree itself.

`Entity` nodes and all descendants of an `Entity` node are read-only and cannot be edited or modified in any way.

Node

Entity

```
public interface Entity extends Node {  
    // Public Instance Methods  
    public abstract String getNotationName();  
    public abstract String getPublicId();  
    public abstract String getSystemId();  
}
```

EntityReference

Java 1.4

`org.w3c.dom`

This interface represents a reference from an XML document to an entity defined in the document's DTD. Character entities and predefined entities such as `<` are always expanded in XML documents and do not create `EntityReference` nodes. Note also that some XML parsers expand all entity references. Documents created by such parsers do not contain `EntityReference` nodes.

This interface defines no methods of its own. The `getNodeName()` method of the `Node` interface provides the name of the referenced entity. The `getEntities()` method of the `DocumentType` interface provides a way to look up the `Entity` object associated with that name. Note however, that the `DocumentType` may not contain an `Entity` with the specified name (because, for example, nonvalidating XML parsers are not required to parse the external subset of the DTD.) In this case, the `EntityReference` is a reference to a named entity whose content is not known, and it has no children. On the other hand, if the `DocumentType` does contain an `Entity` node with the specified name, then the child nodes of the `EntityReference` are a copy of the child nodes of the `Entity`, and represent the expansion of the entity. (The children of an `EntityReference` may not be an exact copy of the children of an `Entity` if the entity's expansion includes namespace prefixes that are not bound to namespace URIs.)

Like `Entity` nodes, `EntityReference` nodes and their descendants are read-only and cannot be edited or modified.

Node

EntityReference

```
public interface EntityReference extends Node {  
}
```

Returned By: `org.w3c.dom.Document.createEntityReference()`

NamedNodeMap

Java 1.4

`org.w3c.dom`

The `NamedNodeMap` interface defines a collection of nodes that may be looked up by name or by namespace URI and local name. It is unrelated to the `java.util.Map` interface.

Use `getNamedItem()` to look for and return a node whose `getNodeName()` method returns the specified value. Use `getNamedItemNS()` to look for and return a node whose `getNamespaceURI()` and `getLocalName()` methods return the specified values. A `NamedNodeMap` is a mapping from names to nodes, and does not order the nodes in any particular way. Nevertheless, it does impose an arbitrary ordering on the nodes and allow them to be looked up by index. Use `getLength()` to find out how many nodes are contained in the `NamedNodeMap`, and use `item()` to obtain the `Node` object at a specified index.

If a `NamedNodeMap` is not read-only, you can use `removeNamedItem()` and `removeNamedItemNS()` to remove a named node from the map, and you can use `setNamedItem()` and `setNamedItemNS()` to add a node to the map, mapping to it from its name or its namespace URI and local name.

`NamedNodeMap` objects are live, which means that they immediately reflect any changes to the document tree. For example, if you obtain a `NamedNodeMap` that represents the attributes of an element, and then add a new attribute to that element, the new attribute is automatically available through the `NamedNodeMap`: you do not need to obtain a new `NamedNodeMap` to get the modified set of attributes.

`NamedNodeMap` is returned only by relatively obscure methods of the DOM API. The most notable use is as the return value of the `getAttributes()` method of `Node`. It is usually easier to work with attributes through the methods of the `Element` interface, however. Two methods of `DocumentType` also return read-only `NamedNodeMap` objects.

```
public interface NamedNodeMap {
    // Public Instance Methods
    public abstract int getLength();
    public abstract Node getNamedItem(String name);
    public abstract Node getNamedItemNS(String namespaceURI, String localName);
    public abstract Node item(int index);
    public abstract Node removeNamedItem(String name) throws DOMException;
    public abstract Node removeNamedItemNS(String namespaceURI, String localName) throws DOMException;
    public abstract Node setNamedItem(Node arg) throws DOMException;
    public abstract Node setNamedItemNS(Node arg) throws DOMException;
}
```

Returned By: `javax.imageio.metadata.IIOMetadataNode.getAttributes()`,
`DocumentType.getEntities()`, `getNotations()`, `Node.getAttributes()`

Node

Java 1.4

`org.w3c.dom`

All objects in a DOM document tree (including the `Document` object itself) implement the `Node` interface, which provides basic methods for traversing and manipulating the tree.

`getParentNode()` and `getChildNodes()` allow you to traverse up and down the document tree. You can enumerate the children of a given node by looping through the elements of the `NodeList` returned by `getChildNodes()`, or by using `getFirstChild()` and `getNextSibling()` (or `getLastChild()` and `getPreviousSibling()` to loop backwards). It is sometimes useful to call `hasChildNodes()` to determine whether a node has children or not. `getOwnerDocument()` returns the `Document` node of which the node is a descendant or with which it is associated. It provides a quick way to jump to the root of the document tree.

Several methods allow you to add children to a tree or alter the list of children. `appendChild()` adds a new child node at the end of this node's list of children. `insertChild()` inserts a node into this node's list of children, placing it immediately before a specified child node. `removeChild()` removes the specified node from this node's list of children. `replaceChild()` replaces one child node of this node with another node. For all of these

Node

methods, if the node to be appended or inserted is already part of the document tree, it is first removed from its current parent. Use `cloneNode()` to produce a copy of this node. Pass `true` if you want all descendants of this node to be cloned as well.

Every object in a document tree implements the `Node` interface, but also implements a more specialized subinterface, such as `Element` or `Text`. The `getNodeType()` method provides an easy way to determine which subinterface a node implements: the return value is one of the `_NODE` constants defined by this class. You might use the return value of `getNodeType()` in a `switch` statement, for example, to determine how to process a node of unknown type.

`getNodeName()` and `getNodeValue()` provide additional information about a node, but the interpretation of the strings they return depends on the node type as shown in the table below. Note that subinterfaces typically define specialized methods (such as the `getTagName()` method of `Element` and the `getData()` method of `Text`) for obtaining this same information. Note also that unless a node is read-only, you can use `setNodeValue()` to alter the value associated with the node.

| Node type | Node name | Node value |
|--|-----------------------------------|-------------------------|
| <code>ELEMENT_NODE</code> | The element's tag name | <code>null</code> |
| <code>ATTRIBUTE_NODE</code> | The attribute name | The attribute value |
| <code>TEXT_NODE</code> | "#text" | The text of the node |
| <code>CDATA_SECTION_NODE</code> | "#cdata-section" | The text of the node |
| <code>ENTITY_REFERENCE_NODE</code> | The name of the referenced entity | <code>null</code> |
| <code>ENTITY_NODE</code> | The entity name | <code>null</code> |
| <code>PROCESSING_INSTRUCTION_NODE</code> | The target of the PI | The remainder of the PI |
| <code>COMMENT_NODE</code> | "#comment" | The text of the comment |
| <code>DOCUMENT_NODE</code> | "#document" | <code>null</code> |
| <code>DOCUMENT_TYPE_NODE</code> | The document type name | <code>null</code> |
| <code>DOCUMENT_FRAGMENT_NODE</code> | "#document-fragment" | <code>null</code> |
| <code>NOTATION_NODE</code> | The notation name | <code>null</code> |

In documents that use namespaces, the `getNodeName()` method of a `Element` or `Attr` node returns the qualified node name, which may include a namespace prefix. In documents that use namespaces, you may prefer to use the namespace-aware methods `getNamespaceURI()`, `getLocalName()` and `getPrefix()`.

`Element` nodes may have a list of attributes, and the `Element` interface defines a number of methods for working with these attributes. In addition, however, `Node` defines the `hasAttributes()` method to determine if a node has any attributes. If it does, they can be retrieved with `getAttributes()`.

Text content in an XML document is represented by `Text` nodes, which have methods for manipulating that textual content. The `Node` interface defines a `normalize()` method which has the specialized purpose of normalizing all descendants of a node by deleting empty `Text` nodes and coalescing adjacent `Text` nodes into a single combined node. Document trees usually start off in this normalized form, but modifications to the tree may result in nonnormalized documents.

Most of the other interfaces in this package extend `Node`. `Document`, `Element` and `Text` are the most commonly used.

```

public interface Node {
// Public Constants
    public static final short ATTRIBUTE_NODE;           =2
    public static final short CDATA_SECTION_NODE;       =4
    public static final short COMMENT_NODE;             =8
    public static final short DOCUMENT_FRAGMENT_NODE;   =11
    public static final short DOCUMENT_NODE;           =9
    public static final short DOCUMENT_TYPE_NODE;      =10
    public static final short ELEMENT_NODE;            =1
    public static final short ENTITY_NODE;             =6
    public static final short ENTITY_REFERENCE_NODE;    =5
    public static final short NOTATION_NODE;           =12
    public static final short PROCESSING_INSTRUCTION_NODE; =7
    public static final short TEXT_NODE;               =3
// Property Accessor Methods (by property name)
    public abstract NamedNodeMap getAttributes();
    public abstract NodeList getChildNodes();
    public abstract Node getFirstChild();
    public abstract Node getLastChild();
    public abstract String getLocalName();
    public abstract String getNamespaceURI();
    public abstract Node getNextSibling();
    public abstract String getNodeName();
    public abstract short getNodeType();
    public abstract String getNodeValue() throws DOMException;
    public abstract void setNodeValue(String nodeValue) throws DOMException;
    public abstract org.w3c.dom.Document getOwnerDocument();
    public abstract Node getParentNode();
    public abstract String getPrefix();
    public abstract void setPrefix(String prefix) throws DOMException;
    public abstract Node getPreviousSibling();
// Public Instance Methods
    public abstract Node appendChild(Node newChild) throws DOMException;
    public abstract Node cloneNode(boolean deep);
    public abstract boolean hasAttributes();
    public abstract boolean hasChildNodes();
    public abstract Node insertBefore(Node newChild, Node refChild) throws DOMException;
    public abstract boolean isSupported(String feature, String version);
    public abstract void normalize();
    public abstract Node removeChild(Node oldChild) throws DOMException;
    public abstract Node replaceChild(Node newChild, Node oldChild) throws DOMException;
}

```

Implementations: Attr, CharacterData, org.w3c.dom.Document, DocumentFragment, DocumentType, org.w3c.dom.Element, org.w3c.dom.Entity, EntityReference, Notation, ProcessingInstruction

Passed To: Too many methods to list.

Returned By: Too many methods to list.

Type Of: javax.imageio.metadata.IIOInvalidTreeException.offendingNode

NodeList

Java 1.4

org.w3c.dom

This interface represents a read-only ordered collection of nodes that can be iterated through. `getLength()` returns the number of nodes in the list, and `item()` returns the Node

NodeList

at a specified index in the list (the index of the first node is 0). The elements of a `NodeList` are always valid `Node` objects: a `NodeList` never contains null elements.

Note that `NodeList` objects are live—they are not static but immediately reflect changes to the document tree. For example, if you have a `NodeList` that represents the children of a specific node, and you then delete one of those children, the child will be removed from your `NodeList`. Be careful when looping through the elements of a `NodeList` if the body of your loop makes changes to the document tree (such as deleting nodes) that may affect the contents of the `NodeList`!

```
public interface NodeList {  
    // Public Instance Methods  
    public abstract int getLength();  
    public abstract Node item(int index);  
}
```

Implementations: `javax.imageio.metadata.IIOMetadataNode`

Returned By: `javax.imageio.metadata.IIOMetadataNode.getChildren()`,
`getElementsByTagName()`, `getElementsByTagNameNS()`,
`org.w3c.dom.Document.getElementsByTagName()`, `getElementsByTagNameNS()`,
`org.w3c.dom.Element.getElementsByTagName()`, `getElementsByTagNameNS()`, `Node.getChildren()`,
`org.w3c.dom.html.HTMLDocument.getElementsByName()`

Notation

Java 1.4

`org.w3c.dom`

This interface represents a notation declared in the DTD of an XML document. In XML notations are used to specify the format of an unparsed entity or to formally declare a processing instruction target.

The `getNodeName()` method of the `Node` interface returns the name of the notation. `getSystemId()` and `getPublicId()` return the system identifier and the public identifier specified in the notation declaration. The `getNotations()` method of the `DocumentType` interface returns a `NamedNodeMap` of `Notation` objects declared in the DTD and provides a way to look up `Notation` objects by notation name.

Because notations appear in the DTD and not the document itself, `Notation` nodes are never part of the document tree, and the `getParentNode()` method always returns null. Similarly, since XML notation declarations never have any content, a `Notation` node never has children and `getChildren()` always returns null. Notation objects are read-only and cannot be modified in any way.

Node Notation

```
public interface Notation extends Node {  
    // Public Instance Methods  
    public abstract String getPublicId();  
    public abstract String getSystemId();  
}
```

ProcessingInstruction

Java 1.4

`org.w3c.dom`

This interface represents an XML processing instruction (or PI) which specifies an arbitrary string of data to a named target processor. The `getTarget()` and `getData()` methods return the target and data portions of a PI, and these values can also be obtained using the `getNodeName()` and `getNodeValue()` methods of the `Node` interface. You can alter the

data portion of a PI with `setData()` or with the `setNodeValue()` method of `Node`. ProcessingInstruction nodes never have children.

Node ProcessingInstruction

```
public interface ProcessingInstruction extends Node {
// Public Instance Methods
    public abstract String getData();
    public abstract String getTarget();
    public abstract void setData(String data) throws DOMException;
}
```

Returned By: `org.w3c.dom.Document.createProcessingInstruction()`

Text

Java 1.4

`org.w3c.dom`

A `Text` node represents a run of plain text that does not contain any XML markup. Plain text appears within XML elements and attributes, and `Text` nodes typically appear as children of `Element` and `Attr` nodes. `Text` nodes inherit from `CharacterData`, and the textual content of a `Text` node is available through the `getData()` method inherited from `CharacterData` or through the `getNodeValue()` method inherited from `Node`.

`Text` nodes may be manipulated using any of the methods inherited from `CharacterData`. The `Text` interface defines one method of its own: `splitText()` splits a `Text` node at the specified character position. The method changes the original node so that it contains only the text up to the specified position. Then it creates a new `Text` node that contains the text from the specified position on and inserts that new node into the document tree immediately after the original one. The `Node.normalize()` method reverses this process by deleting empty `Text` nodes and merging adjacent `Text` nodes into a single node.

`Text` nodes never have children.

Node CharacterData Text

```
public interface Text extends CharacterData {
// Public Instance Methods
    public abstract Text splitText(int offset) throws DOMException;
}
```

Implementations: `CDATASection`

Returned By: `org.w3c.dom.Document.createTextNode()`, `Text.splitText()`