Document Object Model (DOM) Level 2 Traversal and Range Specification

Version 1.0

W3C Proposed Recommendation *27 September, 2000*

This version:
http://www.w3.org/TR/2000/PR-DOM-Level-2-Traversal-Range-20000927
(PostScript file, PDF file, plain text, ZIP file)
Latest version:
http://www.w3.org/TR/DOM-Level-2-Traversal-Range
Previous version:
http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510

Editors:
Vidur Apparao, *Netscape Communications Corporation*
Mike Champion, *Arbortext and Software AG*
Joe Kesselman, *IBM*
Jonathan Robie, *Texcel Research and Software AG*
Peter Sharpe, *SoftQuad Software Inc.*

Abstract

This specification defines the Document Object Model Level 2 Traversal and Range, a platform- and language-neutral interface that allows programs and scripts to dynamically traverse and identify a range of content in a document. The Document Object Model Level 2 Traversal and Range builds on the Document Object Model Level 2 Core [DOM Level 2 Core].

The DOM Level 2 Traversal and Range is made of two modules. The two modules contains specialized interfaces dedicaced to traversing the document structure and identify a range in a document.

Status of this document

This is a W3C Proposed Recommendation for review by W3C members and other interested parties. W3C Advisory Committee Members are invited to send formal comments, visible only to the W3C Team, to dom-review@w3.org until October 25, 2000.

Comments on this document are invited and are to be sent to the public mailing list www-dom@w3.org. An archive is available at http://lists.w3.org/Archives/Public/www-dom/.

Publication as a Proposed Recommendation does not imply endorsement by the W3C membership. This is still a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite W3C Proposed Recommendations as other than "work in progress."

This document has been produced as part of the W3C DOM Activity. The authors of this document are the DOM WG members. Different modules of the Document Object Model have different editors.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Table of contents

Expanded Table of Contents

Copyright Notice

W3C Document Copyright Notice and License

W3C Software Copyright Notice and License

Note: This section is a copy of the W3C Software Copyright Notice and License and could be found at http://www.w3.org/Consortium/Legal/copyright-software-19980720

1. Document Object Model Traversal

Editors
Mike Champion, Software AG
Joe Kesselman, IBM
Jonathan Robie, Software AG

1.1. Overview

This chapter describes the optional DOM Level 2 *Traversal* feature. Its `TreeWalker` [p.24] , `NodeIterator` [p.19] , and `NodeFilter` [p.21] interfaces provide easy-to-use, robust, selective traversal of a document's contents. A DOM application can use the `hasFeature` method of the `DOMImplementation` interface to determine whether this feature is supported or not. The feature string for all the interfaces listed in this section is "Traversal" and the version is "2.0".

`NodeIterators` [p.19] and `TreeWalkers` [p.24] are two different ways of representing the nodes of a document subtree and a position within the nodes they present. A `NodeIterator` [p.19] presents a flattened view of the subtree as an ordered sequence of nodes, presented in document order. Because this view is presented without respect to hierarchy, iterators have methods to move forward and backward, but not to move up and down. Conversely, a `TreeWalker` [p.24] maintains the hierarchical relationships of the subtree, allowing navigation of this hierarchy. In general, `TreeWalkers` are better for tasks in which the structure of the document around selected nodes will be manipulated, while `NodeIterators` are better for tasks that focus on the content of each selected node.

`NodeIterators` [p.19] and `TreeWalkers` [p.24] each present a view of a document subtree that may not contain all nodes found in the subtree. In this specification, we refer to this as the *logical view* to distinguish it from the *physical view*, which corresponds to the document subtree per se. When an iterator or `TreeWalker` [p.24] is created, it may be associated with a `NodeFilter` [p.21] , which examines each node and determines whether it should appear in the logical view. In addition, flags may be used to specify which node types should occur in the logical view.

`NodeIterators` [p.19] and `TreeWalkers` [p.24] are dynamic - the logical view changes to reflect changes made to the underlying document. However, they differ in how they respond to those changes. `NodeIterators` [p.19] , which present the nodes sequentially, attempt to maintain their location relative to a position in that sequence when the sequence's contents change. `TreeWalkers` [p.24] , which present the nodes as a filtered tree, maintain their location relative to their current node and remain attached to that node if it is moved to a new context. We will discuss these behaviors in greater detail below.

1.1.1. `NodeIterators`

A `NodeIterator` [p.19] allows the members of a list of nodes to be returned sequentially. In the current DOM interfaces, this list will always consist of the nodes of a subtree, presented in document order. When an iterator is first created, calling its `nextNode()` method returns the first node in the logical view of the subtree; in most cases, this is the root of the subtree. Each successive call advances the `NodeIterator` through the list, returning the next node available in the logical view. When no more nodes are visible, `nextNode()` returns `null`.

`NodeIterators` [p.19] are created using the `createNodeIterator` method found in the `DocumentTraversal` [p.27] interface. When a `NodeIterator` [p.19] is created, flags can be used to determine which node types will be "visible" and which nodes will be "invisible" while traversing the tree; these flags can be combined using the `OR` operator. Nodes that are "invisible" are skipped over by the iterator as though they did not exist.

The following code creates an iterator, then calls a function to print the name of each element:

```
        NodeIterator iter=
     ((DocumentTraversal)document).createNodeIterator(
          root, NodeFilter.SHOW_ELEMENT, null);

        while (Node n = iter.nextNode())
            printMe(n);
```

1.1.1.1. Moving Forward and Backward

`NodeIterators` [p.19] present nodes as an ordered list, and move forward and backward within this list. The iterator's position is always either between two nodes, before the first node, or after the last node. When an iterator is first created, the position is set before the first item. The following diagram shows the list view that an iterator might provide for a particular subtree, with the position indicated by an asterisk '*' :

```
 * A B C D E F G H I
```

Each call to `nextNode()` returns the next node and advances the position. For instance, if we start with the above position, the first call to `nextNode()` returns "A" and advances the iterator:

```
 [A] * B C D E F G H I
```

The position of a `NodeIterator` [p.19] can best be described with respect to the last node returned, which we will call the *reference node*. When an iterator is created, the first node is the reference node, and the iterator is positioned before the reference node. In these diagrams, we use square brackets to indicate the reference node.

A call to `previousNode()` returns the previous node and moves the position backward. For instance, if we start with the `NodeIterator` [p.19] between "A" and "B", it would return "A" and move to the position shown below:

```
 * [A] B C D E F G H I
```

If `nextNode()` is called at the end of a list, or `previousNode()` is called at the beginning of a list, it returns `null` and does not change the position of the iterator. When a `NodeIterator` [p.19] is first created, the reference node is the first node:

```
 * [A] B C D E F G H I
```

1.1.1.2. Robustness

A `NodeIterator` [p.19] may be active while the data structure it navigates is being edited, so an iterator must behave gracefully in the face of change. Additions and removals in the underlying data structure do not invalidate a `NodeIterator`; in fact, a `NodeIterator` is never invalidated unless its `detach()` method is invoked. To make this possible, the iterator uses the reference node to maintain its position. The state of an iterator also depends on whether the iterator is positioned before or after the reference node.

If changes to the iterated list do not remove the reference node, they do not affect the state of the `NodeIterator` [p.19] . For instance, the iterator's state is not affected by inserting new nodes in the vicinity of the iterator or removing nodes other than the reference node. Suppose we start from the following position:

```
A B C [D] * E F G H I
```

Now let's remove "E". The resulting state is:

```
A B C [D] * F G H I
```

If a new node is inserted, the `NodeIterator` [p.19] stays close to the reference node, so if a node is inserted between "D" and "F", it will occur between the iterator and "F":

```
A B C [D] * X F G H I
```

Moving a node is equivalent to a removal followed by an insertion. If we move "I" to the position before "X" the result is:

```
A B C [D] * I X F G H
```

If the reference node is removed from the list being iterated over, a different node is selected as the reference node. If the reference node's position is before that of the `NodeIterator` [p.19] , which is usually the case after `nextNode()` has been called, the nearest node before the iterator is chosen as the new reference node. Suppose we remove the "D" node, starting from the following state:

```
A B C [D] * F G H I
```

The "C" node becomes the new reference node, since it is the nearest node to the `NodeIterator` [p.19] that is before the iterator:

```
A B [C] * F G H I
```

If the reference node is after the `NodeIterator` [p.19] , which is usually the case after `previousNode()` has been called, the nearest node after the iterator is chosen as the new reference node. Suppose we remove "E", starting from the following state:

```
A B C D * [E] F G H I
```

The "F" node becomes the new reference node, since it is the nearest node to the `NodeIterator` [p.19] that is after the iterator:

```
A B C D * [F] G H I
```

As noted above, moving a node is equivalent to a removal followed by an insertion. Suppose we wish to move the "D" node to the end of the list, starting from the following state:

```
A B C [D] * F G H I C
```

The resulting state is as follows:

```
A B [C] * F G H I D
```

One special case arises when the reference node is the last node in the list and the reference node is removed. Suppose we remove node "C", starting from the following state:

```
A B * [C]
```

According to the rules we have given, the new reference node should be the nearest node after the `NodeIterator` [p.19] , but there are no further nodes after "C". The same situation can arise when `previousNode()` has just returned the first node in the list, which is then removed. Hence: If there is no node in the original direction of the reference node, the nearest node in the opposite direction is selected as the reference node:

```
A [B] *
```

If the `NodeIterator` [p.19] is positioned within a block of nodes that is removed, the above rules clearly indicate what is to be done. For instance, suppose "C" is the *parent* [p.73] node of "D", "E", and "F", and we remove "C", starting with the following state:

```
A B C [D] * E F G H I D
```

The resulting state is as follows:

```
A [B] * G H I D
```

Finally, note that removing a `NodeIterator` [p.19] 's `root` node from its *parent* [p.73] does not alter the list being iterated over, and thus does not change the iterator's state.

1.1.1.3. Visibility of Nodes

The underlying data structure that is being iterated may contain nodes that are not part of the logical view, and therefore will not be returned by the `NodeIterator` [p.19] . If nodes that are to be excluded because of the value of the `whatToShow` flag, `nextNode()` returns the next visible node, skipping over the excluded "invisible" nodes. If a `NodeFilter` [p.21] is present, it is applied before returning a node; if the filter does not accept the node, the process is repeated until a node is accepted by the filter and is returned. If no visible nodes are encountered, a `null` is returned and the iterator is positioned at the end of the list. In this case, the reference node is the last node in the list, whether or not it is visible. The same approach is taken, in the opposite direction, for `previousNode()`.

In the following examples, we will use lowercase letters to represent nodes that are in the data structure, but which are not in the logical view. For instance, consider the following list:

```
A [B] * c d E F G
```

A call to `nextNode()` returns E and advances to the following position:

```
A B c d [E] * F G
```

Nodes that are not visible may nevertheless be used as reference nodes if a reference node is removed. Suppose node "E" is removed, started from the state given above. The resulting state is:

```
A B c [d] * F G
```

Suppose a new node "X", which is visible, is inserted before "d". The resulting state is:

```
A B c X [d] * F G
```

Note that a call to `previousNode()` now returns node X. It is important not to skip over invisible nodes when the reference node is removed, because there are cases, like the one just given above, where the wrong results will be returned. When "E" was removed, if the new reference node had been "B" rather than "d", calling `previousNode()` would not return "X".

1.1.2. `NodeFilters`

`NodeFilters` [p.21] allow the user to create objects that "filter out" nodes. Each filter contains a user-written function that looks at a node and determines whether or not it should be presented as part of the traversal's logical view of the document. To use a `NodeFilter` [p.21] , you create a `NodeIterator` [p.19] or a `TreeWalker` [p.24] that uses the filter. The traversal engine applies the filter to each node, and if the filter does not accept the node, traversal skips over the node as though it were not present in the document. `NodeFilters` need not know how to navigate the structure that contains the nodes on which they operate.

Filters will be consulted when a traversal operation is performed, or when a `NodeIterator` [p.19] 's reference node is removed from the subtree being iterated over and it must select a new one. However, the exact timing of these filter calls may vary from one DOM implementation to another. For that reason, `NodeFilters` [p.21] should not attempt to maintain state based on the history of past invocations; the resulting behavior may not be portable.

Similarly, `TreeWalkers` [p.24] and `NodeIterators` [p.19] should behave as if they have no memory of past filter results, and no anticipation of future results. If the conditions a `NodeFilter` [p.21] is examining have changed (e.g., an attribute which it tests has been added or removed) since the last time the traversal logic examined this node, this change in visibility will be discovered only when the next traversal operation is performed. For example: if the filtering for the current node changes from `FILTER_SHOW` to `FILTER_SKIP`, a `TreeWalker` [p.24] will be able to navigate off that node in any direction, but not back to it unless the filtering conditions change again. `NodeFilters` which change during a traversal can be written, but their behavior may be confusing and they should be avoided when possible.

1.1.2.1. Using `NodeFilters`

A `NodeFilter` [p.21] contains one method named `acceptNode()`, which allows a `NodeIterator` [p.19] or `TreeWalker` [p.24] to pass a `Node` to a filter and ask whether it should be present in the logical view. The `acceptNode()` function returns one of three values to state how the `Node` should be treated. If `acceptNode()` returns `FILTER_ACCEPT`, the `Node` will be present in the logical view; if it returns `FILTER_SKIP`, the `Node` will not be present in the logical view, but the children of the `Node` may; if it returns `FILTER_REJECT`, neither the `Node` nor its *descendants* [p.73] will be present in the logical view. Since iterators present nodes as an ordered list, without hierarchy, `FILTER_REJECT` and `FILTER_SKIP` are synonyms for `NodeIterators`, skipping only the single current node.

Consider a filter that accepts the named anchors in an HTML document. In HTML, an HREF can refer to any A element that has a NAME attribute. Here is a `NodeFilter` [p.21] in Java that looks at a node and determines whether it is a named anchor:

```
class NamedAnchorFilter implements NodeFilter
{
 short acceptNode(Node n) {
  if (n.getNodeType()==Node.ELEMENT_NODE) {
   Element e = (Element)n;
   if (! e.getNodeName().equals("A"))
    return FILTER_SKIP;
  if (e.getAttributeNode("NAME") != null)
    return FILTER_ACCEPT;
   }
        return FILTER_SKIP;
  }
}
```

If the above `NodeFilter` [p.21] were to be used only with `NodeIterators` [p.19] , it could have used `FILTER_REJECT` wherever `FILTER_SKIP` is used, and the behavior would not change. For `TreeWalker` [p.24] , though, `FILTER_REJECT` would reject the children of any element that is not a named anchor, and since named anchors are always contained within other elements, this would have meant that no named anchors would be found. `FILTER_SKIP` rejects the given node, but continues to examine the children; therefore, the above filter will work with either a `NodeIterator` [p.19] or a `TreeWalker`.

To use this filter, the user would create an instance of the `NodeFilter` [p.21] and create a `NodeIterator` [p.19] using it:

```
NamedAnchorFilter myFilter = new NamedAnchorFilter();
NodeIterator iter=
     ((DocumentTraversal)document).createNodeIterator(
          node, NodeFilter.SHOW_ELEMENT, myFilter);
```

Note that the use of the `SHOW_ELEMENT` flag is not strictly necessary in this example, since our sample `NodeFilter` [p.21] tests the `nodeType`. However, some implementations of the Traversal interfaces may be able to improve `whatToShow` performance by taking advantage of knowledge of the document’s structure, which makes the use of `SHOW_ELEMENT` worthwhile. Conversely, while we could remove the `nodeType` test from our filter, that would make it dependent upon `whatToShow` to distinguish between

`Elements`, `Attr`'s, and `ProcessingInstructions`.

1.1.2.2. `NodeFilters` and Exceptions

When writing a `NodeFilter` [p.21] , users should avoid writing code that can throw an exception. However, because a DOM implementation can not prevent exceptions from being thrown, it is important that the behavior of filters that throw an exception be well-defined. A `TreeWalker` [p.24] or `NodeIterator` [p.19] does not catch or alter an exception thrown by a filter, but lets it propagate up to the user's code. The following functions may invoke a `NodeFilter`, and may therefore propagate an exception if one is thrown by a filter:

1. `NodeIterator` [p.19] `.nextNode()`
2. `NodeIterator` [p.19] `.previousNode()`
3. `TreeWalker` [p.24] `.firstChild()`
4. `TreeWalker` [p.24] `.lastChild()`
5. `TreeWalker` [p.24] `.nextSibling()`
6. `TreeWalker` [p.24] `.previousSibling()`
7. `TreeWalker` [p.24] `.nextNode()`
8. `TreeWalker` [p.24] `.previousNode()`
9. `TreeWalker` [p.24] `.parentNode()`

1.1.2.3. `NodeFilters` and Document Mutation

Well-designed `NodeFilters` [p.21] should not have to modify the underlying structure of the document. But a DOM implementation can not prevent a user from writing filter code that does alter the document structure. Traversal does not provide any special processing to handle this case. For instance, if a `NodeFilter` [p.21] removes a node from a document, it can still accept the node, which means that the node may be returned by the `NodeIterator` [p.19] or `TreeWalker` [p.24] even though it is no longer in the subtree being traversed. In general, this may lead to inconsistent, confusing results, so we encourage users to write `NodeFilters` that make no changes to document structures. Instead, do your editing in the loop controlled by the traversal object.

1.1.2.4. `NodeFilters` and `whatToShow` flags

`NodeIterator` [p.19] and `TreeWalker` [p.24] apply their `whatToShow` flags before applying filters. If a node is skipped by the active `whatToShow` flags, a `NodeFilter` [p.21] will not be called to evaluate that node. Please note that this behavior is similar to that of `FILTER_SKIP`; children of that node will be considered, and filters may be called to evaluate them. Also note that it will in fact be a "skip" even if the `NodeFilter` would have preferred to reject the entire subtree; if this would cause a problem in your application, consider setting `whatToShow` to `SHOW_ALL` and performing the `nodeType` test inside your filter.

1.1.3. `TreeWalker`

The `TreeWalker` [p.24] interface provides many of the same benefits as the `NodeIterator` [p.19] interface. The main difference between these two interfaces is that the `TreeWalker` presents a tree-oriented view of the nodes in a subtree, rather than the iterator's list-oriented view. In other words, an iterator allows you to move forward or back, but a `TreeWalker` allows you to also move to the *parent* [p.73] of a node, to one of its children, or to a *sibling* [p.73] .

Using a `TreeWalker` [p.24] is quite similar to navigation using the Node directly, and the navigation methods for the two interfaces are analogous. For instance, here is a function that recursively walks over a tree of nodes in document order, taking separate actions when first entering a node and after processing any children:

```
processMe(Node n) {
   nodeStartActions(n);
   for (Node child=n.firstChild();
        child != null;
        child=child.nextSibling()) {
      processMe(child);
   }
   nodeEndActions(n);
}
```

Doing the same thing using a `TreeWalker` [p.24] is quite similar. There is one difference: since navigation on the `TreeWalker` changes the current position, the position at the end of the function has changed. A read/write attribute named `currentNode` allows the current node for a `TreeWalker` to be both queried and set. We will use this to ensure that the position of the `TreeWalker` is restored when this function is completed:

```
processMe(TreeWalker tw) {
   Node n = tw.getCurrentNode();
   nodeStartActions(tw);
   for (Node child=tw.firstChild();
        child!=null;
        child=tw.nextSibling()) {
      processMe(tw);
   }

   tw.setCurrentNode(n);
   nodeEndActions(tw);
}
```

The advantage of using a `TreeWalker` [p.24] instead of direct `Node` navigation is that the `TreeWalker` allows the user to choose an appropriate view of the tree. Flags may be used to show or hide `Comments` or `ProcessingInstructions`; entities may be expanded or shown as `EntityReference` nodes. In addition, `NodeFilters` [p.21] may be used to present a custom view of the tree. Suppose a program needs a view of a document that shows which tables occur in each chapter, listed by chapter. In this view, only the chapter elements and the tables that they contain are seen. The first step is to write an appropriate filter:

```
class TablesInChapters implements NodeFilter {

   short acceptNode(Node n) {
      if (n.getNodeType()==Node.ELEMENT_NODE) {

          if (n.getNodeName().equals("CHAPTER"))
             return FILTER_ACCEPT;

          if (n.getNodeName().equals("TABLE"))
             return FILTER_ACCEPT;

          if (n.getNodeName().equals("SECT1")
              || n.getNodeName().equals("SECT2")
              || n.getNodeName().equals("SECT3")
              || n.getNodeName().equals("SECT4")
              || n.getNodeName().equals("SECT5")
              || n.getNodeName().equals("SECT6")
              || n.getNodeName().equals("SECT7"))
             return FILTER_SKIP;

      }

      return FILTER_REJECT;
        }
}
```

This filter assumes that TABLE elements are contained directly in CHAPTER or SECTn elements. If another kind of element is encountered, it and its children are rejected. If a SECTn element is encountered, it is skipped, but its children are explored to see if they contain any TABLE elements.

Now the program can create an instance of this `NodeFilter` [p.21] , create a `TreeWalker` [p.24] that uses it, and pass this `TreeWalker` to our ProcessMe() function:

```
TablesInChapters tablesInChapters  = new TablesInChapters();
TreeWalker tw  =
     ((DocumentTraversal)document).createTreeWalker(
          root, NodeFilter.SHOW_ELEMENT, tablesInChapters);
processMe(tw);
```

(Again, we’ve chosen to both test the `nodeType` in the filter’s logic and use `SHOW_ELEMENT`, for the reasons discussed in the earlier `NodeIterator` [p.19] example.)

Without making any changes to the above `ProcessMe()` function, it now processes only the CHAPTER and TABLE elements. The programmer can write other filters or set other flags to choose different sets of nodes; if functions use `TreeWalker` [p.24] to navigate, they will support any view of the document defined with a `TreeWalker`.

Note that the structure of a `TreeWalker` [p.24] ’s filtered view of a document may differ significantly from that of the document itself. For example, a `TreeWalker` with only `SHOW_TEXT` specified in its `whatToShow` parameter would present all the `Text` nodes as if they were *siblings* [p.73] of each other yet had no *parent* [p.73] .

1.1.3.1. Robustness

As with `NodeIterators` [p.19] , a `TreeWalker` [p.24] may be active while the data structure it navigates is being edited, and must behave gracefully in the face of change. Additions and removals in the underlying data structure do not invalidate a `TreeWalker`; in fact, a `TreeWalker` is never invalidated.

But a `TreeWalker` [p.24] 's response to these changes is quite different from that of a `NodeIterator` [p.19] . While `NodeIterators` respond to editing by maintaining their position within the list that they are iterating over, `TreeWalkers` will instead remain attached to their `currentNode`. All the `TreeWalker`'s navigation methods operate in terms of the context of the `currentNode` at the time they are invoked, no matter what has happened to, or around, that node since the last time the `TreeWalker` was accessed. This remains true even if the `currentNode` is moved out of its original subtree.

As an example, consider the following document fragment:

```
...
<subtree>
        <twRoot>
                <currentNode/>
                <anotherNode/>
        </twRoot>
</subtree>
...
```

Let's say we have created a `TreeWalker` [p.24] whose `root` node is the <twRoot/> element and whose `currentNode` is the <currentNode/> element. For this illustration, we will assume that all the nodes shown above are accepted by the `TreeWalker`'s `whatToShow` and filter settings.

If we use `removeChild()` to remove the <currentNode/> element from its *parent* [p.73] , that element remains the `TreeWalker` [p.24] 's `currentNode`, even though it is no longer within the `root` node's subtree. We can still use the `TreeWalker` to navigate through any children that the orphaned `currentNode` may have, but are no longer able to navigate outward from the `currentNode` since there is no *parent* [p.73] available.

If we use `insertBefore()` or `appendChild()` to give the <currentNode/> a new *parent* [p.73] , then `TreeWalker` [p.24] navigation will operate from the `currentNode`'s new location. For example, if we inserted the <currentNode/> immediately after the <anotherNode/> element, the `TreeWalker`'s `previousSibling()` operation would move it back to the <anotherNode/>, and calling `parentNode()` would move it up to the <twRoot/>.

If we instead insert the `currentNode` into the <subtree/> element, like so:

```
...
<subtree>
        <currentNode/>
        <twRoot>
                <anotherNode/>
        </twRoot>
</subtree>
...
```

we have moved the `currentNode` out from under the `TreeWalker` [p.24] 's `root` node. This does not invalidate the `TreeWalker`; it may still be used to navigate relative to the `currentNode`. Calling its `parentNode()` operation, for example, would move it to the <subtree/> element, even though that too is outside the original `root` node. However, if the `TreeWalker`'s navigation should take it back into the original `root` node's subtree -- for example, if rather than calling `parentNode()` we called `nextNode()`, moving the `TreeWalker` to the <twRoot/> element -- the `root` node will "recapture" the `TreeWalker`, and prevent it from traversing back out.

This becomes a bit more complicated when filters are in use. Relocation of the `currentNode` -- or explicit selection of a new `currentNode`, or changes in the conditions that the `NodeFilter` [p.21] is basing its decisions on -- can result in a `TreeWalker` [p.24] having a `currentNode` which would not otherwise be visible in the filtered (logical) view of the document. This node can be thought of as a "transient member" of that view. When you ask the `TreeWalker` to navigate off this node the result will be just as if it had been visible, but you may be unable to navigate back to it unless conditions change to make it visible again.

In particular: If the `currentNode` becomes part of a subtree that would otherwise have been Rejected by the filter, that entire subtree may be added as transient members of the logical view. You will be able to navigate within that subtree (subject to all the usual filtering) until you move upward past the Rejected *ancestor* [p.73] . The behavior is as if the Rejected node had only been Skipped (since we somehow wound up inside its subtree) until we leave it; thereafter, standard filtering applies.

1.2. Formal Interface Definition

Interface *NodeIterator* (introduced in **DOM Level 2**)

`Iterators` are used to step through a set of nodes, e.g. the set of nodes in a `NodeList`, the document subtree governed by a particular `Node`, the results of a query, or any other set of nodes. The set of nodes to be iterated is determined by the implementation of the `NodeIterator`. DOM Level 2 specifies a single `NodeIterator` implementation for document-order traversal of a document subtree. Instances of these iterators are created by calling `DocumentTraversal` [p.27] `.createNodeIterator()`.

IDL Definition

```
// Introduced in DOM Level 2:
interface NodeIterator {
  readonly attribute Node             root;
  readonly attribute unsigned long    whatToShow;
  readonly attribute NodeFilter       filter;
  readonly attribute boolean          expandEntityReferences;
  Node               nextNode()
                                        raises(DOMException);
  Node               previousNode()
                                        raises(DOMException);
  void               detach();
};
```

Attributes

`expandEntityReferences` of type `boolean`, readonly

The value of this flag determines whether the children of entity reference nodes are visible to the iterator. If false, they and their *descendants* [p.73] will be rejected. Note that this rejection takes precedence over `whatToShow` and the filter. Also note that this is currently the only situation where `NodeIterators` may reject a complete subtree rather than skipping individual nodes.
To produce a view of the document that has entity references expanded and does not expose the entity reference node itself, use the `whatToShow` flags to hide the entity reference node and set `expandEntityReferences` to true when creating the iterator.
To produce a view of the document that has entity reference nodes but no entity expansion, use the `whatToShow` flags to show the entity reference node and set `expandEntityReferences` to false.

`filter` of type `NodeFilter` [p.21] , readonly

The `NodeFilter` [p.21] used to screen nodes.

`root` of type `Node`, readonly

The root node of the `NodeIterator`, as specified when it was created.

`whatToShow` of type `unsigned long`, readonly

This attribute determines which node types are presented via the iterator. The available set of constants is defined in the `NodeFilter` [p.21] interface. Nodes not accepted by `whatToShow` will be skipped, but their children may still be considered. Note that this skip takes precedence over the filter, if any.

Methods

`detach`

Detaches the `NodeIterator` from the set which it iterated over, releasing any computational resources and placing the iterator in the INVALID state. After `detach` has been invoked, calls to `nextNode` or `previousNode` will raise the exception INVALID_STATE_ERR.

No Parameters
No Return Value
No Exceptions

`nextNode`

Returns the next node in the set and advances the position of the iterator in the set. After a `NodeIterator` is created, the first call to `nextNode()` returns the first node in the set.
Return Value

`Node`	The next `Node` in the set being iterated over, or `null` if there are no more members in that set.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if this method is called after the `detach` method was invoked.

No Parameters

`previousNode`

Returns the previous node in the set and moves the position of the `NodeIterator` backwards in the set.

Return Value

`Node`	The previous `Node` in the set being iterated over, or `null` if there are no more members in that set.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if this method is called after the `detach` method was invoked.

No Parameters

Interface *NodeFilter* (introduced in **DOM Level 2**)

Filters are objects that know how to "filter out" nodes. If a `NodeIterator` [p.19] or `TreeWalker` [p.24] is given a `NodeFilter`, it applies the filter before it returns the next node. If the filter says to accept the node, the traversal logic returns it; otherwise, traversal looks for the next node and pretends that the node that was rejected was not there.

The DOM does not provide any filters. `NodeFilter` is just an interface that users can implement to provide their own filters.

`NodeFilters` do not need to know how to traverse from node to node, nor do they need to know anything about the data structure that is being traversed. This makes it very easy to write filters, since the only thing they have to know how to do is evaluate a single node. One filter may be used with a number of different kinds of traversals, encouraging code reuse.

IDL Definition

```
// Introduced in DOM Level 2:
interface NodeFilter {

  // Constants returned by acceptNode
  const short               FILTER_ACCEPT                  = 1;
  const short               FILTER_REJECT                  = 2;
  const short               FILTER_SKIP                    = 3;


  // Constants for whatToShow
  const unsigned long       SHOW_ALL                       = 0xFFFFFFFF;
  const unsigned long       SHOW_ELEMENT                   = 0x00000001;
  const unsigned long       SHOW_ATTRIBUTE                 = 0x00000002;
  const unsigned long       SHOW_TEXT                      = 0x00000004;
  const unsigned long       SHOW_CDATA_SECTION             = 0x00000008;
  const unsigned long       SHOW_ENTITY_REFERENCE          = 0x00000010;
  const unsigned long       SHOW_ENTITY                    = 0x00000020;
  const unsigned long       SHOW_PROCESSING_INSTRUCTION    = 0x00000040;
  const unsigned long       SHOW_COMMENT                   = 0x00000080;
  const unsigned long       SHOW_DOCUMENT                  = 0x00000100;
  const unsigned long       SHOW_DOCUMENT_TYPE             = 0x00000200;
  const unsigned long       SHOW_DOCUMENT_FRAGMENT         = 0x00000400;
  const unsigned long       SHOW_NOTATION                  = 0x00000800;

  short              acceptNode(in Node n);
};
```

Definition group *Constants returned by acceptNode*

The following constants are returned by the acceptNode() method:

Defined Constants

`FILTER_ACCEPT`
: Accept the node. Navigation methods defined for `NodeIterator` [p.19] or `TreeWalker` [p.24] will return this node.

`FILTER_REJECT`
: Reject the node. Navigation methods defined for `NodeIterator` [p.19] or `TreeWalker` [p.24] will not return this node. For `TreeWalker`, the children of this node will also be rejected. `NodeIterators` treat this as a synonym for `FILTER_SKIP`.

`FILTER_SKIP`
: Skip this single node. Navigation methods defined for `NodeIterator` [p.19] or `TreeWalker` [p.24] will not return this node. For both `NodeIterator` and `TreeWalker`, the children of this node will still be considered.

Definition group *Constants for whatToShow*

These are the available values for the `whatToShow` parameter used in `TreeWalkers` [p.24] and `NodeIterators` [p.19] . They are the same as the set of possible types for `Node`, and their values are derived by using a bit position corresponding to the value of `nodeType` for the equivalent node type. If a bit in `whatToShow` is set false, that will be taken as a request to skip over this type of node; the behavior in that case is similar to that of `FILTER_SKIP`.

Note that if node types greater than 32 are ever introduced, they may not be individually testable via `whatToShow`. If that need should arise, it can be handled by selecting `SHOW_ALL` together with an appropriate `NodeFilter`.

Defined Constants

`SHOW_ALL`
: Show all `Nodes`.

`SHOW_ATTRIBUTE`
: Show `Attr` nodes. This is meaningful only when creating an iterator or tree-walker with an attribute node as its `root`; in this case, it means that the attribute node will appear in the first position of the iteration or traversal. Since attributes are never children of other nodes, they do not appear when traversing over the document tree.

`SHOW_CDATA_SECTION`
: Show `CDATASection` nodes.

`SHOW_COMMENT`
: Show `Comment` nodes.

`SHOW_DOCUMENT`
: Show `Document` nodes.

`SHOW_DOCUMENT_FRAGMENT`
: Show `DocumentFragment` nodes.

`SHOW_DOCUMENT_TYPE`
: Show `DocumentType` nodes.

`SHOW_ELEMENT`
: Show `Element` nodes.

`SHOW_ENTITY`
: Show `Entity` nodes. This is meaningful only when creating an iterator or tree-walker with an `Entity` node as its `root`; in this case, it means that the `Entity` node will appear in the first position of the traversal. Since entities are not part of the document tree, they do not appear when traversing over the document tree.

`SHOW_ENTITY_REFERENCE`
: Show `EntityReference` nodes.

`SHOW_NOTATION`
: Show `Notation` nodes. This is meaningful only when creating an iterator or tree-walker with a `Notation` node as its `root`; in this case, it means that the `Notation` node will appear in the first position of the traversal. Since notations are not part of the document tree, they do not appear when traversing over the document tree.

`SHOW_PROCESSING_INSTRUCTION`
: Show `ProcessingInstruction` nodes.

`SHOW_TEXT`
: Show `Text` nodes.

Methods

`acceptNode`

Test whether a specified node is visible in the logical view of a `TreeWalker` [p.24] or `NodeIterator` [p.19] . This function will be called by the implementation of `TreeWalker` and `NodeIterator`; it is not normally called directly from user code. (Though you could do so if you wanted to use the same filter to guide your own application logic.)

Parameters

`n` of type `Node`

The node to check to see if it passes the filter or not.

Return Value

`short`	a constant to determine whether the node is accepted, rejected, or skipped, as defined above [p.22] .

No Exceptions

Interface *TreeWalker* (introduced in **DOM Level 2**)

`TreeWalker` objects are used to navigate a document tree or subtree using the view of the document defined by their `whatToShow` flags and filter (if any). Any function which performs navigation using a `TreeWalker` will automatically support any view defined by a `TreeWalker`.

Omitting nodes from the logical view of a subtree can result in a structure that is substantially different from the same subtree in the complete, unfiltered document. Nodes that are *siblings* [p.73] in the `TreeWalker` view may be children of different, widely separated nodes in the original view. For instance, consider a `NodeFilter` [p.21] that skips all nodes except for Text nodes and the root node of a document. In the logical view that results, all text nodes will be *siblings* [p.73] and appear as direct children of the root node, no matter how deeply nested the structure of the original document.

IDL Definition

```
// Introduced in DOM Level 2:
interface TreeWalker {
  readonly attribute Node             root;
  readonly attribute unsigned long    whatToShow;
  readonly attribute NodeFilter       filter;
  readonly attribute boolean          expandEntityReferences;
           attribute Node             currentNode;
                                        // raises(DOMException) on setting

  Node               parentNode();
  Node               firstChild();
  Node               lastChild();
  Node               previousSibling();
  Node               nextSibling();
  Node               previousNode();
  Node               nextNode();
};
```

Attributes

`currentNode` of type `Node`

The node at which the `TreeWalker` is currently positioned.
Alterations to the DOM tree may cause the current node to no longer be accepted by the `TreeWalker`'s associated filter. `currentNode` may also be explicitly set to any node, whether or not it is within the subtree specified by the `root` node or would be accepted by the filter and `whatToShow` flags. Further traversal occurs relative to `currentNode` even if it is not part of the current view, by applying the filters in the requested direction; if no traversal is possible, `currentNode` is not changed.

Exceptions on setting

`DOMException`	NOT_SUPPORTED_ERR: Raised if an attempt is made to set `currentNode` to `null`.

`expandEntityReferences` of type `boolean`, readonly

The value of this flag determines whether the children of entity reference nodes are visible to the `TreeWalker`. If false, they and their *descendants* [p.73] will be rejected. Note that this rejection takes precedence over `whatToShow` and the filter, if any.
To produce a view of the document that has entity references expanded and does not expose the entity reference node itself, use the `whatToShow` flags to hide the entity reference node and set `expandEntityReferences` to true when creating the `TreeWalker`. To produce a view of the document that has entity reference nodes but no entity expansion, use the `whatToShow` flags to show the entity reference node and set `expandEntityReferences` to false.

`filter` of type `NodeFilter` [p.21] , readonly

The filter used to screen nodes.

`root` of type `Node`, readonly

The `root` node of the `TreeWalker`, as specified when it was created.

`whatToShow` of type `unsigned long`, readonly

This attribute determines which node types are presented via the `TreeWalker`. The available set of constants is defined in the `NodeFilter` [p.21] interface. Nodes not accepted by `whatToShow` will be skipped, but their children may still be considered. Note that this skip takes precedence over the filter, if any.

Methods

`firstChild`

Moves the `TreeWalker` to the first visible *child* [p.73] of the current node, and returns the new node. If the current node has no visible children, returns `null`, and retains the current node.

Return Value

`Node`	The new node, or `null` if the current node has no visible children in the `TreeWalker`'s logical view.

No Parameters
No Exceptions

`lastChild`

Moves the `TreeWalker` to the last visible *child* [p.73] of the current node, and returns the new node. If the current node has no visible children, returns `null`, and retains the current node.

Return Value

`Node`	The new node, or `null` if the current node has no children in the `TreeWalker`'s logical view.

No Parameters
No Exceptions

`nextNode`

Moves the `TreeWalker` to the next visible node in document order relative to the current node, and returns the new node. If the current node has no next node, or if the search for nextNode attempts to step upward from the `TreeWalker`'s `root` node, returns `null`, and retains the current node.

Return Value

`Node`	The new node, or `null` if the current node has no next node in the `TreeWalker`'s logical view.

No Parameters
No Exceptions

`nextSibling`

Moves the `TreeWalker` to the next *sibling* [p.73] of the current node, and returns the new node. If the current node has no visible next *sibling* [p.73] , returns `null`, and retains the current node.

Return Value

`Node`	The new node, or `null` if the current node has no next *sibling* [p.73] . in the `TreeWalker`'s logical view.

No Parameters
No Exceptions

`parentNode`

Moves to and returns the closest visible *ancestor* [p.73] node of the current node. If the search for `parentNode` attempts to step upward from the `TreeWalker`'s `root` node, or if it fails to find a visible *ancestor* [p.73] node, this method retains the current position and returns `null`.

Return Value

`Node`	The new *parent* [p.73] node, or `null` if the current node has no parent in the `TreeWalker`'s logical view.

No Parameters
No Exceptions

`previousNode`
Moves the `TreeWalker` to the previous visible node in document order relative to the current node, and returns the new node. If the current node has no previous node, or if the search for `previousNode` attempts to step upward from the `TreeWalker`'s `root` node, returns `null`, and retains the current node.
Return Value

`Node`	The new node, or `null` if the current node has no previous node in the `TreeWalker`'s logical view.

No Parameters
No Exceptions

`previousSibling`
Moves the `TreeWalker` to the previous *sibling* [p.73] of the current node, and returns the new node. If the current node has no visible previous *sibling* [p.73] , returns `null`, and retains the current node.
Return Value

`Node`	The new node, or `null` if the current node has no previous *sibling* [p.73] . in the `TreeWalker`'s logical view.

No Parameters
No Exceptions

Interface *DocumentTraversal* (introduced in **DOM Level 2**)

`DocumentTraversal` contains methods that create iterators and tree-walkers to traverse a node and its children in document order (depth first, pre-order traversal, which is equivalent to the order in which the start tags occur in the text representation of the document). In DOMs which support the Traversal feature, `DocumentTraversal` will be implemented by the same objects that implement the Document interface.
IDL Definition

```
// Introduced in DOM Level 2:
interface DocumentTraversal {
  NodeIterator       createNodeIterator(in Node root,
                                        in unsigned long whatToShow,
                                        in NodeFilter filter,
                                        in boolean entityReferenceExpansion)
                                        raises(DOMException);
  TreeWalker         createTreeWalker(in Node root,
                                      in unsigned long whatToShow,
                                      in NodeFilter filter,
                                      in boolean entityReferenceExpansion)
                                        raises(DOMException);
};
```

Methods

`createNodeIterator`

Create a new `NodeIterator` [p.19] over the subtree rooted at the specified node.

Parameters

`root` of type `Node`

The node which will be iterated together with its children. The iterator is initially positioned just before this node. The `whatToShow` flags and the filter, if any, are not considered when setting this position. The root must not be `null`.

`whatToShow` of type `unsigned long`

This flag specifies which node types may appear in the logical view of the tree presented by the iterator. See the description of `NodeFilter` [p.21] for the set of possible `SHOW_` values.
These flags can be combined using `OR`.

`filter` of type `NodeFilter` [p.21]

The `NodeFilter` to be used with this `TreeWalker` [p.24] , or `null` to indicate no filter.

`entityReferenceExpansion` of type `boolean`

The value of this flag determines whether entity reference nodes are expanded.

Return Value

`NodeIterator` [p.19]	The newly created `NodeIterator`.

Exceptions

`DOMException`	NOT_SUPPORTED_ERR: Raised if the specified `root` is `null`.

`createTreeWalker`

Create a new `TreeWalker` [p.24] over the subtree rooted at the specified node.

Parameters

`root` of type `Node`

The node which will serve as the `root` for the `TreeWalker` [p.24] . The `whatToShow` flags and the `NodeFilter` [p.21] are not considered when setting this value; any node type will be accepted as the `root`. The `currentNode` of the `TreeWalker` is initialized to this node, whether or not it is visible. The `root` functions as a stopping point for traversal methods that look upward in the document structure, such as `parentNode` and nextNode. The `root` must not be `null`.

`whatToShow` of type `unsigned long`
This flag specifies which node types may appear in the logical view of the tree presented by the tree-walker. See the description of `NodeFilter` [p.21] for the set of possible SHOW_ values.
These flags can be combined using `OR`.

`filter` of type `NodeFilter` [p.21]
The `NodeFilter` to be used with this `TreeWalker` [p.24] , or `null` to indicate no filter.

`entityReferenceExpansion` of type `boolean`
If this flag is false, the contents of `EntityReference` nodes are not presented in the logical view.

Return Value

`TreeWalker` [p.24]	The newly created `TreeWalker`.

Exceptions

`DOMException`	NOT_SUPPORTED_ERR: Raised if the specified `root` is `null`.

2. Document Object Model Range

Editors
Vidur Apparao, Netscape Communications
Peter Sharpe, SoftQuad Software Inc.

2.1. Introduction

A Range identifies a range of content in a Document, DocumentFragment or Attr. It is contiguous in the sense that it can be characterized as selecting all of the content between a pair of boundary-points.

Note: In a text editor or a word processor, a user can make a selection by pressing down the mouse at one point in a document, moving the mouse to another point, and releasing the mouse. The resulting selection is contiguous and consists of the content between the two points.

The term ’selecting’ does not mean that every Range corresponds to a selection made by a GUI user; however, such a selection can be returned to a DOM user as a Range.

Note: In bidirectional writing (Arabic, Hebrew), a range may correspond to a logical selection that is not necessarily contiguous when displayed. A visually contiguous selection, also used in some cases, may not correspond to a single logical selection, and may therefore have to be represented by more than one range.

The Range interface provides methods for accessing and manipulating the document tree at a higher level than similar methods in the Node interface. The expectation is that each of the methods provided by the Range interface for the insertion, deletion and copying of content can be directly mapped to a series of Node editing operations enabled by DOM Core. In this sense, the Range operations can be viewed as convenience methods that also enable the implementation to optimize common editing patterns.

This chapter describes the Range interface, including methods for creating and moving a Range and methods for manipulating content with Ranges. The feature string for the interfaces listed in this section is "Range" and the version is "2.0".

2.2. Definitions and Notation

2.2.1. Position

This chapter refers to two different representations of a document: the text or source form that includes the document markup and the tree representation similar to the one described in the introduction section of the DOM Level 2 Core [DOM Level 2 Core].

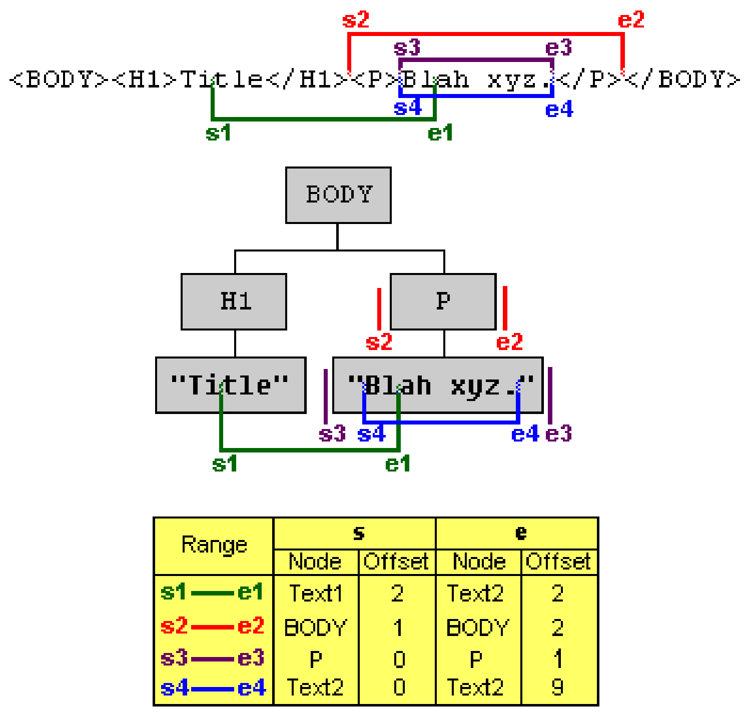
A Range consists of two *boundary-points* corresponding to the start and the end of the Range. A boundary-point’s position in a Document or DocumentFragment tree can be characterized by a node and an offset. The node is called the *container* of the boundary-point and of its position. The container and its ancestors are the *ancestor container*s of the boundary-point and of its position. The offset within the node is called the *offset* of the boundary-point and its position. If the container is an Attr, Document, DocumentFragment, Element or EntityReference node, the offset is between its *child* [p.73] nodes. If the container is a CharacterData, Comment or ProcessingInstruction node, the offset is between the *16-bit units* [p.73] of the UTF-16 encoded string contained by it.

The *boundary-points* [p.31] of a Range must have a common *ancestor container* [p.31] which is either a Document, DocumentFragment or Attr node. That is, the content of a Range must be entirely within the subtree rooted by a single Document, DocumentFragment or Attr Node. This common *ancestor container* [p.31] is known as the *root container* of the Range. The tree rooted by the *root container* [p.32] is known as the Range's *context tree*.

The *container* [p.31] of an *boundary-point* [p.31] of a Range must be an Element, Comment, ProcessingInstruction, EntityReference, CDATASection, Document, DocumentFragment, Attr, or Text node. None of the *ancestor container* [p.31] s of the *boundary-point* [p.31] of a Range can be a DocumentType, Entity or Notation node.

In terms of the text representation of a document, the *boundary-points* [p.31] of a Range can only be on token boundaries. That is, the *boundary-point* [p.31] of the text range cannot be in the middle of a start- or end-tag of an element or within the name of an entity or character reference. A Range locates a contiguous portion of the content of the structure model.

The relationship between locations in a text representation of the document and in the Node tree interface of the DOM is illustrated in the following diagram:



Range	s		e	
	Node	Offset	Node	Offset
s1——e1	Text1	2	Text2	2
s2——e2	BODY	1	BODY	2
s3——e3	P	0	P	1
s4——e4	Text2	0	Text2	9

Range Example

In this diagram, four different Ranges are illustrated. The *boundary-points* [p.31] of each Range are labelled with *s#* (the start of the Range) and *e#* (the end of the Range), where # is the number of the Range. For Range 2, the start is in the BODY element and is immediately after the H1 element and immediately before the P element, so its position is between the H1 and P children of BODY. The *offset* [p.31] of a *boundary-point* [p.31] whose *container* [p.31] is not a CharacterData node is 0 if it is before the first child, 1 if between the first and second child, and so on. So, for the start of the Range 2, the *container* [p.31] is BODY and the *offset* [p.31] is 1. The *offset* [p.31] of a *boundary-point* [p.31] whose *container* [p.31] is a CharacterData node is obtained similarly but using *16-bit unit* [p.73] positions instead. For example, the *boundary-point* [p.31] labelled s1 of the Range 1 has a Text node (the one containing "Title") as its *container* [p.31] and an *offset* [p.31] of 2 since it is between the second and third *16-bit unit* [p.73] .

Notice that the *boundary-point* [p.31] s of Ranges 3 and 4 correspond to the same location in the text representation. An important feature of the Range is that a *boundary-point* [p.31] of a Range can unambiguously represent every position within the document tree.

The *container* [p.31] s and *offset* [p.31] s of the *boundary-point* [p.31] s can be obtained through the following read-only Range attributes:

```
readonly attribute Node startContainer;
readonly attribute long startOffset;
readonly attribute Node endContainer;
readonly attribute long endOffset;
```

If the *boundary-point* [p.31] s of a Range have the same *container* [p.31] s and *offset* [p.31] s, the Range is said to be a *collapsed* Range. (This is often referred to as an insertion point in a user agent.)

2.2.2. Selection and Partial Selection

A node or *16-bit unit* [p.73] unit is said to be *selected* by a Range if it is between the two *boundary-point* [p.31] s of the Range, that is, if the position immediately before the node or 16-bit unit is before the end of the Range and the position immediately after the node or 16-bit unit is after the start of the range. For example, in terms of a text representation of the document, an element would be *selected* [p.33] by a Range if its corresponding start-tag was located after the start of the Range and its end-tag was located before the end of the Range. In the examples in the above diagram, the Range 2 *selects* [p.33] the P node and the Range 3 *selects* [p.33] the text node containing the text "Blah xyz."

A node is said to be *partially selected* by a Range if it is an *ancestor container* [p.31] of exactly one *boundary-point* [p.31] of the Range. For example, consider Range 1 in the above diagram. The element H1 is *partially selected* [p.33] by that Range since the start of the Range is within one of its children.

2.2.3. Notation

Many of the examples in this chapter are illustrated using a text representation of a document. The *boundary-point* [p.31] s of a Range are indicated by displaying the characters (be they markup or data characters) between the two *boundary-point* [p.31] s in bold, as in

```
<FOO>ABC<BAR>DEF</BAR></FOO>
```

When both *boundary-point* [p.31] s are at the same position, they are indicated with a bold caret ('^'), as in

```
<FOO>A^BC<BAR>DEF</BAR></FOO>
```

2.3. Creating a Range

A Range is created by calling the `createRange()` method on the `DocumentRange` [p.53] interface. This interface can be obtained from the object implementing the `Document` interface using binding-specific casting methods.

```
interface DocumentRange {
  Range createRange();
}
```

The initial state of the Range returned from this method is such that both of its *boundary-point* [p.31] s are positioned at the beginning of the corresponding Document, before any content. In other words, the *container* [p.31] of each *boundary-point* [p.31] is the Document node and the offset within that node is 0.

Like some objects created using methods in the Document interface (such as Nodes and DocumentFragments), Ranges created via a particular document instance can select only content associated with that Document, or with DocumentFragments and Attrs for which that Document is the `ownerDocument`. Such Ranges, then, can not be used with other Document instances.

2.4. Changing a Range's Position

A Range's position can be specified by setting the *container* [p.31] and *offset* [p.31] of each boundary-point with the `setStart` and `setEnd` methods.

```
void setStart(in Node parent, in long offset)
                      raises(RangeException);
void setEnd(in Node parent, in long offset)
              raises(RangeException);
```

If one boundary-point of a Range is set to have a *root container* [p.32] other than the current one for the Range, the Range is *collapsed* [p.33] to the new position. This enforces the restriction that both boundary-points of a Range must have the same *root container* [p.32] .

The start position of a Range is guaranteed to never be after the end position. To enforce this restriction, if the start is set to be at a position after the end, the Range is *collapsed* [p.33] to that position. Similarly, if the end is set to be at a position before the start, the Range is *collapsed* [p.33] to that position.

It is also possible to set a Range's position relative to nodes in the tree:

```
  void setStartBefore(in Node node);
                                raises(RangeException);
  void setStartAfter(in Node node);
                         raises(RangeException);
  void setEndBefore(in Node node);
                        raises(RangeException);
  void setEndAfter(in Node node);
                       raises(RangeException);
```

The *parent* [p.73] of the node becomes the *container* [p.31] of the *boundary-point* [p.31] and the Range is subject to the same restrictions as given above in the description of `setStart()`and `setEnd()`.

A Range can be *collapsed* [p.33] to either boundary-point:

```
  void collapse(in boolean toStart);
```

Passing `TRUE` as the parameter `toStart` will *collapse* [p.33] the Range to its start, `FALSE` to its end.

Testing whether a Range is *collapsed* [p.33] can be done by examining the `collapsed` attribute:

```
  readonly attribute boolean collapsed;
```

The following methods can be used to make a Range select the contents of a node or the node itself.

```
  void selectNode(in Node n);
  void selectNodeContents(in Node n);
```

The following examples demonstrate the operation of the methods `selectNode` and `selectNodeContents`:

```
Before:
  ^<BAR><FOO>A<MOO>B</MOO>C</FOO></BAR>
After Range.selectNodeContents(FOO):
  <BAR><FOO>A<MOO>B</MOO>C</FOO></BAR>
(In this case, FOO is the parent of both boundary-points)
After Range.selectNode(FOO):

<BAR><FOO>A<MOO>B</MOO>C</FOO></BAR>
```

2.5. Comparing Range Boundary-Points

It is possible to compare two Ranges by comparing their boundary-points:

```
  short compareBoundaryPoints(in CompareHow how, in Range sourceRange) raises(RangeException);
```

where `CompareHow` is one of four values: `START_TO_START`, `START_TO_END`, `END_TO_END` and `END_TO_START`. The return value is -1, 0 or 1 depending on whether the corresponding boundary-point of the Range is before, equal to, or after the corresponding boundary-point of `sourceRange`. An exception is thrown if the two Ranges have different *root container* [p.32] s.

The result of comparing two boundary-points (or positions) is specified below. An informal but not always correct specification is that an boundary-point is before, equal to, or after another if it corresponds to a location in a text representation before, equal to, or after the other's corresponding location.

Let A and B be two boundary-points or positions. Then one of the following holds: A is *before* B, A is *equal to* B, or A is *after* B. Which one holds is specified in the following by examining four cases:

In the first case the boundary-points have the same *container* [p.31] . A is *before* B if its *offset* [p.31] is less than the *offset* [p.31] of B, A is *equal to* B if its *offset* [p.31] is equal to the *offset* [p.31] of B, and A is *after* B if its *offset* [p.31] is greater than the *offset* [p.31] of B.

In the second case a child node C of the *container* [p.31] of A is an *ancestor container* [p.31] of B. In this case, A is *before* B if the *offset* [p.31] of A is less than or equal to the index of the child node C and A is *after* B otherwise.

In the third case a child node C of the *container* [p.31] of B is an *ancestor container* [p.31] of A. In this case, A is *before* B if the index of the child node C is less than the *offset* [p.31] of B and A is *after* B otherwise.

In the fourth case, none of three other cases hold: the containers of A and B are *siblings* [p.73] or *descendants* [p.73] of sibling nodes. In this case, A is *before* B if the *container* [p.31] of A is before the *container* [p.31] of B in a pre-order traversal of the Ranges' *context tree* [p.32] and A is *after* B otherwise.

Note that because the same location in a text representation of the document can correspond to two different positions in the DOM tree, it is possible for two boundary-points to not compare equal even though they would be equal in the text representation. For this reason, the informal definition above can sometimes be incorrect.

2.6. Deleting Content with a Range

One can delete the contents selected by a Range with:

```
  void deleteContents();
```

`deleteContents()` deletes all nodes and characters selected by the Range. All other nodes and characters remain in the *context tree* [p.32] of the Range. Some examples of this deletion operation are:

```
(1) <FOO>AB<MOO>CD</MOO>CD</FOO>  -->
<FOO>A^CD</FOO>

(2) <FOO>A<MOO>BC</MOO>DE</FOO>  -->
<FOO>A<MOO>B</MOO>^E</FOO>

(3) <FOO>XY<BAR>ZW</BAR>Q</FOO>  -->
<FOO>X^<BAR>W</BAR>Q</FOO>

(4) <FOO><BAR1>AB</BAR1><BAR2/><BAR3>CD</BAR3></FOO>
-->  <FOO><BAR1>A</BAR1>^<BAR3>D</BAR3>
```

After `deleteContents()` is invoked on a Range, the Range is *collapsed* [p.33] . If no node was *partially selected* [p.33] by the Range, then it is *collapsed* [p.33] to its original start point, as in example (1). If a node was *partially selected* [p.33] by the Range and was an *ancestor container* [p.31] of the start of the Range and no *ancestor* [p.73] of the node satisfies these two conditions, then the Range is collapsed to the position immediately after the node, as in examples (2) and (4). If a node was *partially selected* [p.33] by the Range and was an *ancestor container* [p.31] of the end of the Range and no ancestor of the node satisfies these two conditions, then the Range is collapsed to the position immediately before the node, as in examples (3) and (4).

Note that if deletion of a Range leaves adjacent Text nodes, they are not automatically merged, and empty Text nodes are not automatically removed. Two Text nodes should be joined only if each is the container of one of the boundary-points of a Range whose contents are deleted. To merge adjacent Text nodes, or remove empty text nodes, the `normalize()` method on the `Node` interface should be used.

2.7. Extracting Content

If the contents of a Range need to be extracted rather than deleted, the following method may be used:

```
  DocumentFragment extractContents();
```

The `extractContents()` method removes nodes from the Range's *context tree* [p.32] similarly to the `deleteContents()` method. In addition, it places the deleted contents in a new `DocumentFragment`. The following examples illustrate the contents of the returned DocumentFragment:

```
(1) <FOO>AB<MOO>CD</MOO>CD</FOO>  -->
B<MOO>CD</MOO>

(2) <FOO>A<MOO>BC</MOO>DE</FOO>  -->
<MOO>C<MOO>D

(3) <FOO>XY<BAR>ZW</BAR>Q</FOO>  -->
Y<BAR>Z</BAR>

(4)
<FOO><BAR1>AB</BAR1><BAR2/><BAR3>CD</BAR3></FOO> -->
<BAR1>B</BAR1><BAR2/><BAR3>C</BAR3>
```

It is important to note that nodes that are *partially selected* [p.33] by the Range are cloned. Since part of such a node's contents must remain in the Range's *context tree* [p.32] and part of the contents must be moved to the new DocumentFragment, a clone of the *partially selected* [p.33] node is included in the new DocumentFragment. Note that cloning does not take place for *selected* [p.33] elements; these nodes are moved to the new DocumentFragment.

2.8. Cloning Content

The contents of a Range may be duplicated using the following method:

```
DocumentFragment cloneContents();
```

This method returns a `DocumentFragment` that is similar to the one returned by the method `extractContents()`. However, in this case, the original nodes and character data in the Range are not removed from the Range's *context tree* [p.32] . Instead, all of the nodes and text content within the returned `DocumentFragment` are cloned.

2.9. Inserting Content

A node may be inserted into a Range using the following method:

```
void insertNode(in Node n) raises(RangeException);
```

The `insertNode()` method inserts the specified node into the Range's *context tree* [p.32] . The node is inserted at the start *boundary-point* [p.31] of the Range, without modifying it.

If the start boundary point of the Range is in a `Text` node, the `insertNode` operation splits the `Text` node at the boundary point. If the node to be inserted is also a `Text` node, the resulting adjacent `Text` nodes are not normalized automatically; this operation is left to the application.

The Node passed into this method can be a `DocumentFragment`. In that case, the contents of the `DocumentFragment` are inserted at the start *boundary-point* [p.31] of the Range, but the `DocumentFragment` itself is not. Note that if the Node represents the root of a sub-tree, the entire sub-tree is inserted.

The same rules that apply to the `insertBefore()` method on the Node interface apply here. Specifically, the Node passed in, if it already has a parent, will be removed from its existing position.

2.10. Surrounding Content

The insertion of a single node to subsume the content selected by a Range can be performed with:

```
void surroundContents(in Node newParent);
```

The `surroundContents()` method causes all of the content selected by the Range to be rooted by the specified node. The nodes may not be Attr, Entity, DocumentType, Notation, Document, or DocumentFragment nodes. Calling `surroundContents()` with the Element node FOO in the following examples yields:

```
    Before:
      <BAR>AB<MOO>C</MOO>DE</BAR>

    After surroundContents(FOO):

<BAR>A<FOO>B<MOO>C</MOO>D</FOO>E</BAR>
```

Another way of describing the effect of this method on the Range's *context tree* [p.32] is to decompose it in terms of other operations:

1. Remove the contents selected by the Range with a call to `extractContents()`.
2. Insert the node `newParent` where the Range is collapsed (after the extraction) with `insertNode().`
3. Insert the entire contents of the extracted DocumentFragment into `newParent`. Specifically, invoke the `appendChild()` on `newParent` passing in the DocumentFragment returned as a result of the call to `extractContents()`
4. Select `newParent` and all of its contents with `selectNode()`.

The `surroundContents()` method raises an exception if the Range *partially selects* [p.33] a non-Text node. An example of a Range for which `surroundContents()`raises an exception is:

<FOO>A**B<BAR>C**D</BAR>E</FOO>

If the node `newParent` has any children, those children are removed before its insertion. Also, if the node `newParent` already has a parent, it is removed from the original parent's `childNodes` list.

2.11. Miscellaneous Members

One can clone a Range:

```
Range cloneRange();
```

This creates a new Range which selects exactly the same content as that selected by the Range on which the method `cloneRange` was invoked. No content is affected by this operation.

Because the boundary-points of a Range do not necessarily have the same *container* [p.31] s, use:

```
readonly attribute Node commonAncestorContainer;
```

to get the *ancestor container* [p.31] of both boundary-points that is furthest down from the Range's *root container* [p.32]

One can get a copy of all the character data selected or partially selected by a Range with:

```
DOMString toString();
```

This does nothing more than simply concatenate all the character data selected by the Range. This includes character data in both `Text` and `CDATASection` nodes.

2.12. Range modification under document mutation

As a document is modified, the Ranges within the document need to be updated. For example, if one boundary-point of a Range is within a node and that node is removed from the document, then the Range would be invalid unless it is fixed up in some way. This section describes how Ranges are modified under document mutations so that they remain valid.

There are two general principles which apply to Ranges under document mutation: The first is that all Ranges in a document will remain valid after any mutation operation and the second is that, as much as possible, all Ranges will select the same portion of the document after any mutation operation.

Any mutation of the document tree which affect Ranges can be considered to be a combination of basic deletion and insertion operations. In fact, it can be convenient to think of those operations as being accomplished using the `deleteContents()` and `insertNode()` Range methods and, in the case of Text mutations, the `splitText()` and `normalize()` methods.

2.12.1. Insertions

An insertion occurs at a single point, the insertion point, in the document. For any Range in the document tree, consider each boundary-point. The only case in which the boundary-point will be changed after the insertion is when the boundary-point and the insertion point have the same *container* [p.31] and the *offset* [p.31] of the insertion point is strictly less than the *offset* [p.31] of the Range’s boundary-point. In that case the *offset* [p.31] of the Range’s boundary-point will be increased so that it is between the same nodes or characters as it was before the insertion.

Note that when content is inserted at a boundary-point, it is ambiguous as to where the boundary-point should be repositioned if its relative position is to be maintained. There are two possibilities: at the start or at the end of the newly inserted content. We have chosen that in this case neither the *container* [p.31] nor *offset* [p.31] of the boundary-point is changed. As a result, the boundary-point will be positioned at the start of the newly inserted content.

Examples:

Suppose the Range selects the following:

<P>Abcd efgh X**Y blah i**jkl</P>

Consider the insertion of the text "*inserted text*" at the following positions:

1. Before the ’X’:

<P>Abcd efgh *inserted text*X**Y blah i**jkl</P>

2. After the ’X’:

<P>Abcd efgh X***inserted text*****Y blah i**jkl</P>

3. After the ’Y’:

<P>Abcd efgh X**Y*inserted text* blah i**jkl</P>

4. After the ’h’ in "Y blah":

<P>Abcd efgh X**Y blah*inserted text* i**jkl</P>

2.12.2. Deletions

Any deletion from the document tree can be considered as a sequence of `deleteContents()` operations applied to a minimal set of disjoint Ranges. To specify how a Range is modified under deletions we need only consider what happens to a Range under a single `deleteContents()`operation of another Range. And, in fact, we need only consider what happens to a single boundary-point of the Range since both boundary-points are modified using the same algorithm.

If a boundary-point of the original Range is within the content being deleted, then after the deletion it will be at the same position as the resulting boundary-point of the (now *collapsed* [p.33]) Range used to delete the contents.

If a boundary-point is after the content being deleted then it is not affected by the deletion unless its *container* [p.31] is also the *container* [p.31] of one of the boundary-points of the Range being deleted. If there is such a common *container* [p.31] , then the index of the boundary-point is modified so that the boundary-point maintains its position relative to the content of the *container* [p.31] .

If a boundary-point is before the content being deleted then it is not affected by the deletion at all.

Examples:

In these examples, the Range on which `deleteContents()`is invoked is indicated by the underline.

Example 1.

Before:

```
<P>Abcd efgh The Range ijkl</P>
```

After:

```
<P>Abcd Range ijkl</P>
```

Example 2.

Before:

```
<p>Abcd efgh The Range ijkl</p>
```

After:

```
<p>Abcd ^kl</p>
```

Example 3.

Before:

```
<P>ABCD efgh The <EM>Range</EM> ijkl</P>
```

After:

<P>ABCD <EM>**ange**</EM> ijkl</P>

In this example, the container of the start boundary-point after the deletion is the Text node holding the string "ange".

Example 4.

Before:

<P>Abcd efgh T**he Range i**jkl</P>

After:

<P>Abcd **he Range i**jkl</P>

Example 5.

Before:

<P>Abcd <EM>efgh T**he Range i**j</EM>kl</P>

After:

<P>Abcd ^kl</P>

2.13. Formal Description of the Range Interface

To summarize, the complete, formal description of the `Range` [p.42] interface is given below:

Interface *Range* (introduced in **DOM Level 2**)

IDL Definition

```
// Introduced in DOM Level 2:
interface Range {
  readonly attribute Node             startContainer;
                                        // raises(DOMException) on retrieval

  readonly attribute long             startOffset;
                                        // raises(DOMException) on retrieval

  readonly attribute Node             endContainer;
                                        // raises(DOMException) on retrieval

  readonly attribute long             endOffset;
                                        // raises(DOMException) on retrieval

  readonly attribute boolean          collapsed;
                                        // raises(DOMException) on retrieval

  readonly attribute Node             commonAncestorContainer;
                                        // raises(DOMException) on retrieval
```

```
void                setStart(in Node refNode,
                             in long offset)
                                      raises(RangeException,
                                             DOMException);
void                setEnd(in Node refNode,
                           in long offset)
                                      raises(RangeException,
                                             DOMException);
void                setStartBefore(in Node refNode)
                                      raises(RangeException,
                                             DOMException);
void                setStartAfter(in Node refNode)
                                      raises(RangeException,
                                             DOMException);
void                setEndBefore(in Node refNode)
                                      raises(RangeException,
                                             DOMException);
void                setEndAfter(in Node refNode)
                                      raises(RangeException,
                                             DOMException);
void                collapse(in boolean toStart)
                                      raises(DOMException);
void                selectNode(in Node refNode)
                                      raises(RangeException,
                                             DOMException);
void                selectNodeContents(in Node refNode)
                                      raises(RangeException,
                                             DOMException);

// CompareHow
const unsigned short      START_TO_START                 = 0;
const unsigned short      START_TO_END                   = 1;
const unsigned short      END_TO_END                     = 2;
const unsigned short      END_TO_START                   = 3;

short               compareBoundaryPoints(in unsigned short how,
                                          in Range sourceRange)
                                      raises(DOMException);
void                deleteContents()
                                      raises(DOMException);
DocumentFragment    extractContents()
                                      raises(DOMException);
DocumentFragment    cloneContents()
                                      raises(DOMException);
void                insertNode(in Node newNode)
                                      raises(DOMException,
                                             RangeException);
void                surroundContents(in Node newParent)
                                      raises(DOMException,
                                             RangeException);
Range               cloneRange()
                                      raises(DOMException);
DOMString           toString()
```

```
                                            raises(DOMException);
  void                detach()
                                            raises(DOMException);
};
```

Definition group *CompareHow*

Passed as a parameter to the `compareBoundaryPoints` method.

Defined Constants

`END_TO_END`
: Compare end boundary-point of `sourceRange` to end boundary-point of Range on which `compareBoundaryPoints` is invoked.

`END_TO_START`
: Compare end boundary-point of `sourceRange` to start boundary-point of Range on which `compareBoundaryPoints` is invoked.

`START_TO_END`
: Compare start boundary-point of `sourceRange` to end boundary-point of Range on which `compareBoundaryPoints` is invoked.

`START_TO_START`
: Compare start boundary-point of `sourceRange` to start boundary-point of Range on which `compareBoundaryPoints` is invoked.

Attributes

`collapsed` of type `boolean`, readonly

TRUE if the Range is collapsed

Exceptions on retrieval

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

`commonAncestorContainer` of type `Node`, readonly

The *deepest* [p.73] common *ancestor container* [p.31] of the Range's two boundary-points.

Exceptions on retrieval

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

`endContainer` of type `Node`, readonly

Node within which the Range ends

Exceptions on retrieval

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

`endOffset` of type `long`, readonly

Offset within the ending node of the Range.

Exceptions on retrieval

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

`startContainer` of type `Node`, readonly

Node within which the Range begins

Exceptions on retrieval

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

`startOffset` of type `long`, readonly

Offset within the starting node of the Range.

Exceptions on retrieval

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

Methods

`cloneContents`

Duplicates the contents of a Range

Return Value

`DocumentFragment`	A DocumentFragment that contains content equivalent to this Range.

Exceptions

`DOMException`	HIERARCHY_REQUEST_ERR: Raised if a DocumentType node would be extracted into the new DocumentFragment.
	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Parameters

`cloneRange`

Produces a new Range whose boundary-points are equal to the boundary-points of the Range.

Return Value

`Range` [p.42]	The duplicated Range.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Parameters

`collapse`

Collapse a Range onto one of its boundary-points

Parameters

`toStart` of type `boolean`

If TRUE, collapses the Range onto its start; if FALSE, collapses it onto its end.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`compareBoundaryPoints`

Compare the boundary-points of two Ranges in a document.

Parameters

`how` of type `unsigned short`

`sourceRange` of type `Range` [p.42]

Return Value

`short`	-1, 0 or 1 depending on whether the corresponding boundary-point of the Range is before, equal to, or after the corresponding boundary-point of `sourceRange`.

Exceptions

`DOMException`	WRONG_DOCUMENT_ERR: Raised if the two Ranges are not in the same Document or DocumentFragment. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

`deleteContents`

Removes the contents of a Range from the containing document or document fragment without returning a reference to the removed content.

Exceptions

`DOMException`	NO_MODIFICATION_ALLOWED_ERR: Raised if any portion of the content of the Range is read-only or any of the nodes that contain any of the content of the Range are read-only. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Parameters
No Return Value

`detach`

Called to indicate that the Range is no longer in use and that the implementation may relinquish any resources associated with this Range. Subsequent calls to any methods or attribute getters on this Range will result in a `DOMException` being thrown with an error code of `INVALID_STATE_ERR`.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Parameters
No Return Value

`extractContents`

Moves the contents of a Range from the containing document or document fragment to a new DocumentFragment.

Return Value

`DocumentFragment`	A DocumentFragment containing the extracted contents.

Exceptions

`DOMException`	NO_MODIFICATION_ALLOWED_ERR: Raised if any portion of the content of the Range is read-only or any of the nodes which contain any of the content of the Range are read-only. HIERARCHY_REQUEST_ERR: Raised if a DocumentType node would be extracted into the new DocumentFragment. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Parameters

`insertNode`

Inserts a node into the Document or DocumentFragment at the start of the Range. If the container is a Text node, this will be split at the start of the Range. Adjacent Text nodes will not be automatically merged. If the node to be inserted is a DocumentFragment node, the children will be inserted rather than the DocumentFragment node itself.

Parameters

`newNode` of type `Node`

The node to insert at the start of the Range

Exceptions

`DOMException`	NO_MODIFICATION_ALLOWED_ERR: Raised if an *ancestor container* [p.31] of the start of the Range is read-only. WRONG_DOCUMENT_ERR: Raised if `newNode` and the *container* [p.31] of the start of the Range were not created from the same document. HIERARCHY_REQUEST_ERR: Raised if the *container* [p.31] of the start of the Range is of a type that does not allow children of the type of `newNode` or if `newNode` is an ancestor of the *container* [p.31] . INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.
`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if `newNode` is an Attr, Entity, Notation, or Document node.

No Return Value

`selectNode`

Select a node and its contents

Parameters

`refNode` of type `Node`

The node to select.

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if an ancestor of `refNode` is an Entity, Notation or DocumentType node or if `refNode` is a Document, DocumentFragment, Attr, Entity, or Notation node.
`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`selectNodeContents`

Select the contents within a node

Parameters

`refNode` of type `Node`

Node to select from

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if `refNode` or an ancestor of `refNode` is an Entity, Notation or DocumentType node.
`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`setEnd`

Sets the attributes describing the end of a Range.

Parameters

`refNode` of type `Node`

The `refNode` value. This parameter must be different from `null`.

`offset` of type `long`

The `endOffset` value.

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if `refNode` or an ancestor of `refNode` is an Entity, Notation, or DocumentType node.
`DOMException`	INDEX_SIZE_ERR: Raised if `offset` is negative or greater than the number of child units in `refNode`. Child units are *16-bit units* [p.73] if `refNode` is a CharacterData, Comment or ProcessingInstruction node. Child units are Nodes in all other cases. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`setEndAfter`

Sets the end of a Range to be after a node

Parameters

`refNode` of type `Node`

Range ends after `refNode`.

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode`is not an Attr, Document or DocumentFragment node or if `refNode` is a Document, DocumentFragment, Attr, Entity, or Notation node.
`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`setEndBefore`

Sets the end position to be before a node.

Parameters

`refNode` of type `Node`

Range ends before `refNode`

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode`is not an Attr, Document, or DocumentFragment node or if `refNode`is a Document, DocumentFragment, Attr, Entity, or Notation node.
`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`setStart`

Sets the attributes describing the start of the Range.

Parameters

`refNode` of type `Node`

The `refNode` value. This parameter must be different from `null`.

`offset` of type `long`

The `startOffset` value.

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if `refNode` or an ancestor of `refNode` is an Entity, Notation, or DocumentType node.
`DOMException`	INDEX_SIZE_ERR: Raised if `offset` is negative or greater than the number of child units in `refNode`. Child units are *16-bit units* [p.73] if `refNode` is a CharacterData, Comment or ProcessingInstruction node. Child units are Nodes in all other cases.
	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`setStartAfter`

Sets the start position to be after a node

Parameters

`refNode` of type `Node`

Range starts after `refNode`

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode`is not an Attr, Document, or DocumentFragment node or if `refNode`is a Document, DocumentFragment, Attr, Entity, or Notation node.
`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`setStartBefore`

Sets the start position to be before a node

Parameters

`refNode` of type `Node`

Range starts before `refNode`

Exceptions

`RangeException` [p.54]	INVALID_NODE_TYPE_ERR: Raised if the root container of `refNode`is not an Attr, Document, or DocumentFragment node or if `refNode`is a Document, DocumentFragment, Attr, Entity, or Notation node.
`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Return Value

`surroundContents`

Reparents the contents of the Range to the given node and inserts the node at the position of the start of the Range.

Parameters

`newParent` of type `Node`

The node to surround the contents with.

Exceptions

`DOMException`	NO_MODIFICATION_ALLOWED_ERR: Raised if an *ancestor container* [p.31] of either boundary-point of the Range is read-only. WRONG_DOCUMENT_ERR: Raised if `newParent` and the *container* [p.31] of the start of the Range were not created from the same document. HIERARCHY_REQUEST_ERR: Raised if the *container* [p.31] of the start of the Range is of a type that does not allow children of the type of `newParent` or if `newParent` is an ancestor of the *container* [p.31] or if `node` would end up with a child node of a type not allowed by the type of `node`. INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.
`RangeException` [p.54]	BAD_BOUNDARYPOINTS_ERR: Raised if the Range *partially selects* [p.33] a non-text node. INVALID_NODE_TYPE_ERR: Raised if `node` is an Attr, Entity, DocumentType, Notation, Document, or DocumentFragment node.

No Return Value

`toString`

Returns the contents of a Range as a string. This string contains only the data characters, not any markup.

Return Value

`DOMString`	The contents of the Range.

Exceptions

`DOMException`	INVALID_STATE_ERR: Raised if `detach()` has already been invoked on this object.

No Parameters

Interface *DocumentRange* (introduced in **DOM Level 2**)

IDL Definition

```
// Introduced in DOM Level 2:
interface DocumentRange {
  Range              createRange();
};
```

Methods

`createRange`

This interface can be obtained from the object implementing the `Document` interface using binding-specific casting methods.

Return Value

`Range` [p.42]	The initial state of the Range returned from this method is such that both of its boundary-points are positioned at the beginning of the corresponding Document, before any content. The Range returned can only be used to select content associated with this Document, or with DocumentFragments and Attrs for which this Document is the `ownerDocument`.

No Parameters

No Exceptions

Exception *RangeException* introduced in DOM Level 2

Range operations may throw a `RangeException` [p.54] as specified in their method descriptions.

IDL Definition

```
// Introduced in DOM Level 2:
exception RangeException {
  unsigned short   code;
};
// RangeExceptionCode
const unsigned short      BAD_BOUNDARYPOINTS_ERR         = 1;
const unsigned short      INVALID_NODE_TYPE_ERR          = 2;
```

Definition group *RangeExceptionCode*

An integer indicating the type of error generated.

Defined Constants

`BAD_BOUNDARYPOINTS_ERR`

If the boundary-points of a Range do not meet specific requirements.

`INVALID_NODE_TYPE_ERR`

If the *container* [p.31] of an boundary-point of a Range is being set to either a node of an invalid type or a node with an ancestor of an invalid type.

Appendix A: IDL Definitions

This appendix contains the complete OMG IDL [OMGIDL] for the Level 2 Document Object Model Traversal and Range definitions. The definitions are divided into Traversal [p.55] , and Range [p.56] .

The IDL files are also available as:
http://www.w3.org/TR/2000/PR-DOM-Level-2-Traversal-Range-20000927/idl.zip

A.1: Document Object Model Traversal

traversal.idl:

```
// File: traversal.idl

#ifndef _TRAVERSAL_IDL_
#define _TRAVERSAL_IDL_

#include "dom.idl"

#pragma prefix "dom.w3c.org"
module traversal
{

  typedef dom::Node Node;

  interface NodeFilter;

  // Introduced in DOM Level 2:
  interface NodeIterator {
    readonly attribute Node             root;
    readonly attribute unsigned long    whatToShow;
    readonly attribute NodeFilter       filter;
    readonly attribute boolean          expandEntityReferences;
    Node               nextNode()
                                        raises(dom::DOMException);
    Node               previousNode()
                                        raises(dom::DOMException);
    void               detach();
  };

  // Introduced in DOM Level 2:
  interface NodeFilter {

    // Constants returned by acceptNode
    const short               FILTER_ACCEPT                  = 1;
    const short               FILTER_REJECT                  = 2;
    const short               FILTER_SKIP                    = 3;


    // Constants for whatToShow
    const unsigned long       SHOW_ALL                       = 0xFFFFFFFF;
    const unsigned long       SHOW_ELEMENT                   = 0x00000001;
    const unsigned long       SHOW_ATTRIBUTE                 = 0x00000002;
    const unsigned long       SHOW_TEXT                      = 0x00000004;
    const unsigned long       SHOW_CDATA_SECTION             = 0x00000008;
    const unsigned long       SHOW_ENTITY_REFERENCE          = 0x00000010;
    const unsigned long       SHOW_ENTITY                    = 0x00000020;
    const unsigned long       SHOW_PROCESSING_INSTRUCTION    = 0x00000040;
    const unsigned long       SHOW_COMMENT                   = 0x00000080;
    const unsigned long       SHOW_DOCUMENT                  = 0x00000100;
    const unsigned long       SHOW_DOCUMENT_TYPE             = 0x00000200;
    const unsigned long       SHOW_DOCUMENT_FRAGMENT         = 0x00000400;
    const unsigned long       SHOW_NOTATION                  = 0x00000800;

    short              acceptNode(in Node n);
  };

  // Introduced in DOM Level 2:
  interface TreeWalker {
    readonly attribute Node              root;
    readonly attribute unsigned long     whatToShow;
    readonly attribute NodeFilter        filter;
    readonly attribute boolean           expandEntityReferences;
             attribute Node              currentNode;
                                         // raises(dom::DOMException) on setting

    Node               parentNode();
    Node               firstChild();
    Node               lastChild();
    Node               previousSibling();
    Node               nextSibling();
    Node               previousNode();
    Node               nextNode();
  };

  // Introduced in DOM Level 2:
  interface DocumentTraversal {
    NodeIterator       createNodeIterator(in Node root,
                                          in unsigned long whatToShow,
                                          in NodeFilter filter,
                                          in boolean entityReferenceExpansion)
                                        raises(dom::DOMException);
    TreeWalker         createTreeWalker(in Node root,
                                        in unsigned long whatToShow,
                                        in NodeFilter filter,
                                        in boolean entityReferenceExpansion)
                                        raises(dom::DOMException);
  };
};

#endif // _TRAVERSAL_IDL_
```

A.2: Document Object Model Range

ranges.idl:

```
// File: ranges.idl

#ifndef _RANGES_IDL_
#define _RANGES_IDL_

#include "dom.idl"

#pragma prefix "dom.w3c.org"
module ranges
{

  typedef dom::Node Node;
  typedef dom::DocumentFragment DocumentFragment;
  typedef dom::DOMString DOMString;

  // Introduced in DOM Level 2:
  exception RangeException {
    unsigned short   code;
  };
  // RangeExceptionCode
  const unsigned short      BAD_BOUNDARYPOINTS_ERR         = 1;
  const unsigned short      INVALID_NODE_TYPE_ERR          = 2;


  // Introduced in DOM Level 2:
  interface Range {
    readonly attribute Node             startContainer;
                                        // raises(dom::DOMException) on retrieval

    readonly attribute long             startOffset;
                                        // raises(dom::DOMException) on retrieval

    readonly attribute Node             endContainer;
                                        // raises(dom::DOMException) on retrieval

    readonly attribute long             endOffset;
                                        // raises(dom::DOMException) on retrieval

    readonly attribute boolean          collapsed;
                                        // raises(dom::DOMException) on retrieval

    readonly attribute Node             commonAncestorContainer;
                                        // raises(dom::DOMException) on retrieval

    void               setStart(in Node refNode,
                                 in long offset)
                                        raises(RangeException,
                                               dom::DOMException);
    void               setEnd(in Node refNode,
                               in long offset)
                                        raises(RangeException,
                                               dom::DOMException);
    void               setStartBefore(in Node refNode)
                                        raises(RangeException,
                                               dom::DOMException);
    void               setStartAfter(in Node refNode)
                                        raises(RangeException,
                                               dom::DOMException);
    void               setEndBefore(in Node refNode)
                                        raises(RangeException,
                                               dom::DOMException);
    void               setEndAfter(in Node refNode)
                                        raises(RangeException,
                                               dom::DOMException);
    void               collapse(in boolean toStart)
                                        raises(dom::DOMException);
    void               selectNode(in Node refNode)
                                        raises(RangeException,
                                               dom::DOMException);
    void               selectNodeContents(in Node refNode)
                                        raises(RangeException,
                                               dom::DOMException);

    // CompareHow
    const unsigned short      START_TO_START                 = 0;
    const unsigned short      START_TO_END                   = 1;
    const unsigned short      END_TO_END                     = 2;
    const unsigned short      END_TO_START                   = 3;

    short              compareBoundaryPoints(in unsigned short how,
                                             in Range sourceRange)
                                        raises(dom::DOMException);
    void               deleteContents()
                                        raises(dom::DOMException);
    DocumentFragment   extractContents()
                                        raises(dom::DOMException);
    DocumentFragment   cloneContents()
                                        raises(dom::DOMException);
    void               insertNode(in Node newNode)
                                        raises(dom::DOMException,
                                               RangeException);
    void               surroundContents(in Node newParent)
                                        raises(dom::DOMException,
                                               RangeException);
    Range              cloneRange()
                                        raises(dom::DOMException);
    DOMString          toString()
                                        raises(dom::DOMException);
    void               detach()
                                        raises(dom::DOMException);
  };

  // Introduced in DOM Level 2:
  interface DocumentRange {
    Range              createRange();
  };
};

#endif // _RANGES_IDL_
```

Appendix B: Java Language Binding

This appendix contains the complete Java [Java] bindings for the Level 2 Document Object Model Traversal and Range. The definitions are divided into Traversal [p.59] , and Range [p.61] .

The Java files are also available as http://www.w3.org/TR/2000/PR-DOM-Level-2-Traversal-Range-20000927/java-binding.zip

B.1: Document Object Model Traversal

org/w3c/dom/traversal/NodeIterator.java:

```
package org.w3c.dom.traversal;

import org.w3c.dom.Node;
import org.w3c.dom.DOMException;

public interface NodeIterator {
    public Node getRoot();

    public int getWhatToShow();

    public NodeFilter getFilter();

    public boolean getExpandEntityReferences();

    public Node nextNode()
                         throws DOMException;

    public Node previousNode()
                             throws DOMException;

    public void detach();

}
```

org/w3c/dom/traversal/NodeFilter.java:

```
package org.w3c.dom.traversal;

import org.w3c.dom.Node;

public interface NodeFilter {
    // Constants returned by acceptNode
    public static final short FILTER_ACCEPT             = 1;
    public static final short FILTER_REJECT             = 2;
    public static final short FILTER_SKIP               = 3;

    // Constants for whatToShow
    public static final int SHOW_ALL                  = 0xFFFFFFFF;
    public static final int SHOW_ELEMENT              = 0x00000001;
    public static final int SHOW_ATTRIBUTE            = 0x00000002;
    public static final int SHOW_TEXT                 = 0x00000004;
    public static final int SHOW_CDATA_SECTION        = 0x00000008;
    public static final int SHOW_ENTITY_REFERENCE     = 0x00000010;
    public static final int SHOW_ENTITY               = 0x00000020;
    public static final int SHOW_PROCESSING_INSTRUCTION = 0x00000040;
    public static final int SHOW_COMMENT              = 0x00000080;
    public static final int SHOW_DOCUMENT             = 0x00000100;
    public static final int SHOW_DOCUMENT_TYPE        = 0x00000200;
    public static final int SHOW_DOCUMENT_FRAGMENT    = 0x00000400;
    public static final int SHOW_NOTATION             = 0x00000800;

    public short acceptNode(Node n);

}
```

org/w3c/dom/traversal/TreeWalker.java:

```
package org.w3c.dom.traversal;

import org.w3c.dom.Node;
import org.w3c.dom.DOMException;

public interface TreeWalker {
    public Node getRoot();

    public int getWhatToShow();

    public NodeFilter getFilter();

    public boolean getExpandEntityReferences();

    public Node getCurrentNode();
    public void setCurrentNode(Node currentNode)
                        throws DOMException;

    public Node parentNode();

    public Node firstChild();

    public Node lastChild();

    public Node previousSibling();

    public Node nextSibling();

    public Node previousNode();

    public Node nextNode();

}
```

org/w3c/dom/traversal/DocumentTraversal.java:

```
package org.w3c.dom.traversal;

import org.w3c.dom.Node;
import org.w3c.dom.DOMException;

public interface DocumentTraversal {
    public NodeIterator createNodeIterator(Node root, 
                                           int whatToShow, 
                                           NodeFilter filter, 
                                           boolean entityReferenceExpansion)
                                           throws DOMException;

    public TreeWalker createTreeWalker(Node root, 
                                       int whatToShow, 
                                       NodeFilter filter, 
                                       boolean entityReferenceExpansion)
                                       throws DOMException;

}
```

B.2: Document Object Model Range

org/w3c/dom/ranges/RangeException.java:

```
package org.w3c.dom.ranges;

public class RangeException extends RuntimeException {
    public RangeException(short code, String message) {
       super(message);
       this.code = code;
    }
    public short   code;
    // RangeExceptionCode
    public static final short BAD_BOUNDARYPOINTS_ERR    = 1;
    public static final short INVALID_NODE_TYPE_ERR     = 2;

}
```

org/w3c/dom/ranges/Range.java:

```
package org.w3c.dom.ranges;

import org.w3c.dom.Node;
import org.w3c.dom.DocumentFragment;
import org.w3c.dom.DOMException;

public interface Range {
    public Node getStartContainer()
                                       throws DOMException;

    public int getStartOffset()
                                       throws DOMException;
```

```
public Node getEndContainer()
                                   throws DOMException;

public int getEndOffset()
                                   throws DOMException;

public boolean getCollapsed()
                                   throws DOMException;

public Node getCommonAncestorContainer()
                                   throws DOMException;

public void setStart(Node refNode,
                     int offset)
                     throws RangeException, DOMException;

public void setEnd(Node refNode,
                   int offset)
                   throws RangeException, DOMException;

public void setStartBefore(Node refNode)
                           throws RangeException, DOMException;

public void setStartAfter(Node refNode)
                          throws RangeException, DOMException;

public void setEndBefore(Node refNode)
                         throws RangeException, DOMException;

public void setEndAfter(Node refNode)
                        throws RangeException, DOMException;

public void collapse(boolean toStart)
                     throws DOMException;

public void selectNode(Node refNode)
                       throws RangeException, DOMException;

public void selectNodeContents(Node refNode)
                               throws RangeException, DOMException;

// CompareHow
public static final short START_TO_START            = 0;
public static final short START_TO_END              = 1;
public static final short END_TO_END                = 2;
public static final short END_TO_START              = 3;

public short compareBoundaryPoints(short how,
                                   Range sourceRange)
                                   throws DOMException;

public void deleteContents()
                           throws DOMException;

public DocumentFragment extractContents()
                                        throws DOMException;
```

```
    public DocumentFragment cloneContents()
                                          throws DOMException;

    public void insertNode(Node newNode)
                           throws DOMException, RangeException;

    public void surroundContents(Node newParent)
                                 throws DOMException, RangeException;

    public Range cloneRange()
                            throws DOMException;

    public String toString()
                           throws DOMException;

    public void detach()
                       throws DOMException;

}
```

org/w3c/dom/ranges/DocumentRange.java:

```
package org.w3c.dom.ranges;

public interface DocumentRange {
    public Range createRange();

}
```

org/w3c/dom/ranges/DocumentRange.java:

Appendix C: ECMA Script Language Binding

This appendix contains the complete ECMA Script [ECMAScript] binding for the Level 2 Document Object Model Traversal and Range definitions. The definitions are divided into Traversal [p.65] , and Range [p.67] .

Note: Exceptions handling is only supported by ECMAScript implementation compliant with the Standard ECMA-262 3rd. Edition ([ECMAScript]).

C.1: Document Object Model Traversal

Object **NodeIterator**

The **NodeIterator** object has the following properties:

root

This read-only property is of type **Node**.

whatToShow

This read-only property is of type **int**.

filter

This read-only property is of type **NodeFilter**.

expandEntityReferences

This read-only property is of type **boolean**.

The **NodeIterator** object has the following methods:

nextNode()

This method returns a **Node**.
This method can raise a **DOMException**.

previousNode()

This method returns a **Node**.
This method can raise a **DOMException**.

detach()

This method has no return value.

Class **NodeFilter**

The **NodeFilter** class has the following constants:

NodeFilter.FILTER_ACCEPT

This constant is of type **short** and its value is **1**.

NodeFilter.FILTER_REJECT

This constant is of type **short** and its value is **2**.

NodeFilter.FILTER_SKIP

This constant is of type **short** and its value is **3**.

NodeFilter.SHOW_ALL

This constant is of type **int** and its value is **0xFFFFFFFF**.

NodeFilter.SHOW_ELEMENT

This constant is of type **int** and its value is **0x00000001**.

NodeFilter.SHOW_ATTRIBUTE

This constant is of type **int** and its value is **0x00000002**.

NodeFilter.SHOW_TEXT
This constant is of type **int** and its value is **0x00000004**.
NodeFilter.SHOW_CDATA_SECTION
This constant is of type **int** and its value is **0x00000008**.
NodeFilter.SHOW_ENTITY_REFERENCE
This constant is of type **int** and its value is **0x00000010**.
NodeFilter.SHOW_ENTITY
This constant is of type **int** and its value is **0x00000020**.
NodeFilter.SHOW_PROCESSING_INSTRUCTION
This constant is of type **int** and its value is **0x00000040**.
NodeFilter.SHOW_COMMENT
This constant is of type **int** and its value is **0x00000080**.
NodeFilter.SHOW_DOCUMENT
This constant is of type **int** and its value is **0x00000100**.
NodeFilter.SHOW_DOCUMENT_TYPE
This constant is of type **int** and its value is **0x00000200**.
NodeFilter.SHOW_DOCUMENT_FRAGMENT
This constant is of type **int** and its value is **0x00000400**.
NodeFilter.SHOW_NOTATION
This constant is of type **int** and its value is **0x00000800**.

Object **NodeFilter**
This is an ECMAScript function reference. This method returns a **short**. The parameter is of type **Node**.

Object **TreeWalker**
The **TreeWalker** object has the following properties:
root
This read-only property is of type **Node**.
whatToShow
This read-only property is of type **int**.
filter
This read-only property is of type **NodeFilter**.
expandEntityReferences
This read-only property is of type **boolean**.
currentNode
This property is of type **Node** and can raise a **DOMException** on setting.

The **TreeWalker** object has the following methods:
parentNode()
This method returns a **Node**.
firstChild()
This method returns a **Node**.
lastChild()
This method returns a **Node**.
previousSibling()
This method returns a **Node**.
nextSibling()
This method returns a **Node**.

previousNode()
This method returns a **Node**.
nextNode()
This method returns a **Node**.
Object **DocumentTraversal**
The **DocumentTraversal** object has the following methods:
createNodeIterator(root, whatToShow, filter, entityReferenceExpansion)
This method returns a **NodeIterator**.
The **root** parameter is of type **Node**.
The **whatToShow** parameter is of type **int**.
The **filter** parameter is of type **NodeFilter**.
The **entityReferenceExpansion** parameter is of type **boolean**.
This method can raise a **DOMException**.
createTreeWalker(root, whatToShow, filter, entityReferenceExpansion)
This method returns a **TreeWalker**.
The **root** parameter is of type **Node**.
The **whatToShow** parameter is of type **int**.
The **filter** parameter is of type **NodeFilter**.
The **entityReferenceExpansion** parameter is of type **boolean**.
This method can raise a **DOMException**.

C.2: Document Object Model Range

Class **Range**
The **Range** class has the following constants:
Range.START_TO_START
This constant is of type **short** and its value is **0**.
Range.START_TO_END
This constant is of type **short** and its value is **1**.
Range.END_TO_END
This constant is of type **short** and its value is **2**.
Range.END_TO_START
This constant is of type **short** and its value is **3**.
Object **Range**
The **Range** object has the following properties:
startContainer
This read-only property is of type **Node** and can raise a **DOMException** on retrieval.
startOffset
This read-only property is of type **long** and can raise a **DOMException** on retrieval.
endContainer
This read-only property is of type **Node** and can raise a **DOMException** on retrieval.
endOffset
This read-only property is of type **long** and can raise a **DOMException** on retrieval.
collapsed
This read-only property is of type **boolean** and can raise a **DOMException** on retrieval.

commonAncestorContainer
This read-only property is of type **Node** and can raise a **DOMException** on retrieval.

The **Range** object has the following methods:

setStart(refNode, offset)
This method has no return value.
The **refNode** parameter is of type **Node**.
The **offset** parameter is of type **long**.
This method can raise a **RangeException** or a **DOMException**.

setEnd(refNode, offset)
This method has no return value.
The **refNode** parameter is of type **Node**.
The **offset** parameter is of type **long**.
This method can raise a **RangeException** or a **DOMException**.

setStartBefore(refNode)
This method has no return value.
The **refNode** parameter is of type **Node**.
This method can raise a **RangeException** or a **DOMException**.

setStartAfter(refNode)
This method has no return value.
The **refNode** parameter is of type **Node**.
This method can raise a **RangeException** or a **DOMException**.

setEndBefore(refNode)
This method has no return value.
The **refNode** parameter is of type **Node**.
This method can raise a **RangeException** or a **DOMException**.

setEndAfter(refNode)
This method has no return value.
The **refNode** parameter is of type **Node**.
This method can raise a **RangeException** or a **DOMException**.

collapse(toStart)
This method has no return value.
The **toStart** parameter is of type **boolean**.
This method can raise a **DOMException**.

selectNode(refNode)
This method has no return value.
The **refNode** parameter is of type **Node**.
This method can raise a **RangeException** or a **DOMException**.

selectNodeContents(refNode)
This method has no return value.
The **refNode** parameter is of type **Node**.
This method can raise a **RangeException** or a **DOMException**.

compareBoundaryPoints(how, sourceRange)
This method returns a **short**.
The **how** parameter is of type **short**.
The **sourceRange** parameter is of type **Range**.
This method can raise a **DOMException**.

deleteContents()
This method has no return value.
This method can raise a **DOMException**.
extractContents()
This method returns a **DocumentFragment**.
This method can raise a **DOMException**.
cloneContents()
This method returns a **DocumentFragment**.
This method can raise a **DOMException**.
insertNode(newNode)
This method has no return value.
The **newNode** parameter is of type **Node**.
This method can raise a **DOMException** or a **RangeException**.
surroundContents(newParent)
This method has no return value.
The **newParent** parameter is of type **Node**.
This method can raise a **DOMException** or a **RangeException**.
cloneRange()
This method returns a **Range**.
This method can raise a **DOMException**.
toString()
This method returns a **String**.
This method can raise a **DOMException**.
detach()
This method has no return value.
This method can raise a **DOMException**.

Object **DocumentRange**
The **DocumentRange** object has the following methods:
createRange()
This method returns a **Range**.

Class **RangeException**
The **RangeException** class has the following constants:
RangeException.BAD_BOUNDARYPOINTS_ERR
This constant is of type **short** and its value is **1**.
RangeException.INVALID_NODE_TYPE_ERR
This constant is of type **short** and its value is **2**.

Exception **RangeException**
The **RangeException** object has the following properties:
code
This property is of type **unsigned short**.

C.2: Document Object Model Range

Appendix D: Acknowledgements

Many people contributed to this specification, including members of the DOM Working Group and the DOM Interest Group. We especially thank the following:

Lauren Wood (SoftQuad Software Inc., *chair*), Andrew Watson (Object Management Group), Andy Heninger (IBM), Arnaud Le Hors (W3C and IBM), Ben Chang (Oracle), Bill Smith (Sun), Bill Shea (Merrill Lynch), Bob Sutor (IBM), Chris Lovett (Microsoft), Chris Wilson (Microsoft), David Brownell (Sun), David Singer (IBM), Don Park (invited), Eric Vasilik (Microsoft), Gavin Nicol (INSO), Ian Jacobs (W3C), James Clark (invited), James Davidson (Sun), Jared Sorensen (Novell), Joe Kesselman (IBM), Joe Lapp (webMethods), Joe Marini (Macromedia), Johnny Stenback (Netscape), Jonathan Marsh (Microsoft), Jonathan Robie (Texcel Research and Software AG), Kim Adamson-Sharpe (SoftQuad Software Inc.), Laurence Cable (Sun), Mark Davis (IBM), Mark Scardina (Oracle), Martin Dürst (W3C), Mick Goulish (Software AG), Mike Champion (Arbortext and Software AG), Miles Sabin (Cromwell Media), Patti Lutsky (Arbortext), Paul Grosso (Arbortext), Peter Sharpe (SoftQuad Software Inc.), Phil Karlton (Netscape), Philippe Le Hégaret (W3C, *W3C team contact*), Ramesh Lekshmynarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home and Netscape), Rich Rollman (Microsoft), Rick Gessner (Netscape), Scott Isaacs (Microsoft), Sharon Adler (INSO), Steve Byrne (JavaSoft), Tim Bray (invited), Tom Pixley (Netscape), Vidur Apparao (Netscape), Vinod Anupam (Lucent).

Thanks to all those who have helped to improve this specification by sending suggestions and corrections.

D.1: Production Systems

This specification was written in XML. The HTML, OMG IDL, Java and ECMA Script bindings were all produced automatically.

Thanks to Joe English, author of cost, which was used as the basis for producing DOM Level 1. Thanks also to Gavin Nicol, who wrote the scripts which run on top of cost. Arnaud Le Hors and Philippe Le Hégaret maintained the scripts.

For DOM Level 2, we used Xerces as the basis DOM implementation and wish to thank the authors. Philippe Le Hégaret and Arnaud Le Hors wrote the Java programs which are the DOM application.

Thanks also to Jan Kärrman, author of html2ps, which we use in creating the PostScript version of the specification.

Glossary

Editors

Arnaud Le Hors, W3C and IBM
Lauren Wood, SoftQuad Software Inc.
Robert S. Sutor, IBM Research (for DOM Level 1)

Several of the following term definitions have been borrowed or modified from similar definitions in other W3C or standards documents. See the links within the definitions for more information.

16-bit unit
: The base unit of a `DOMString`. This indicates that indexing on a `DOMString` occurs in units of 16 bits. This must not be misunderstood to mean that a `DOMString` can store arbitrary 16-bit units. A `DOMString` is a character string encoded in UTF-16; this means that the restrictions of UTF-16 as well as the other relevant restrictions on character strings must be maintained. A single character, for example in the form of a numeric character reference, may correspond to one or two 16-bit units. For more information, see [Unicode] and [ISO/IEC 10646].

ancestor
: An *ancestor* node of any node A is any node above A in a tree model of a document, where "above" means "toward the root."

child
: A *child* is an immediate *descendant* node of a node.

deepest
: The *deepest* element is that element which is furthest from the root or document element in a tree model of the document.

descendant
: A *descendant* node of any node A is any node below A in a tree model of a document, where "above" means "toward the root."

parent
: A *parent* is an immediate *ancestor* node of a node.

sibling
: Two nodes are *siblings* if and only if they have the same *parent* node.

tokenized
: The description given to various information items (for example, attribute values of various types, but not including the StringType CDATA) after having been processed by the XML processor. The process includes stripping leading and trailing white space, and replacing multiple space characters by one. See the definition of tokenized type.

References

For the latest version of any W3C specification please consult the list of W3C Technical Reports available at http://www.w3.org/TR.

F.1: Normative references

DOM Level 2 Core
W3C (World Wide Web Consortium) Document Object Model Level 2 Core Specification, September 2000. Available at http://www.w3.org/TR/2000/PR-DOM-Level-2-Core-20000927

ECMAScript
ECMA (European Computer Manufacturers Association) ECMAScript Language Specification. Available at http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

ISO/IEC 10646
ISO (International Organization for Standardization). ISO/IEC 10646-1:2000 (E). Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization.

Java
Sun Microsystems Inc. The Java Language Specification, James Gosling, Bill Joy, and Guy Steele, September 1996. Available at http://java.sun.com/docs/books/jls

OMGIDL
OMG (Object Management Group) IDL (Interface Definition Language) defined in The Common Object Request Broker: Architecture and Specification, version 2.3.1, October 1999. Available from http://www.omg.org/

Unicode
The Unicode Consortium. The Unicode Standard, Version 3.0., February 2000. Available at http://www.unicode.org/unicode/standard/versions/Unicode3.0.html.

Index