The DOM `Node` interface is an abstract base class upon which many other DOM API objects are based, thus letting those object types to be used similarly and often interchangeably. As an abstract class, there is no such thing as a plain `Node` object. All objects that implement `Node` functionality are based on one of its subclasses. Most notable are `Document`, `Element`, and `DocumentFragment`.

In addition, every kind of DOM node is represented by an interface based on `Node`. These include `Attr`, `CharacterData` (which `Text`, `Comment`, and `CDATASection` are all based on), `ProcessingInstruction`, `DocumentType`, `Notation`, `Entity`, and `EntityReference`.

In some cases, a particular feature of the base `Node` interface may not apply to one of its child interfaces; in that case, the inheriting node may return `null` or throw an exception, depending on circumstances. For example, attempting to add children to a node type that cannot have children will throw an exception.

Properties

In addition to the properties below, `Node` inherits properties from its parent, `EventTarget`.

`Node.baseURI` | Read only

Returns a `DOMString` representing the base URL of the document containing the `Node`.

`Node.baseURIObject` | Read only

(Not available to web content.) The `nsIURI` object representing the base URI for the element.

`Node.childNodes` | Read only

Returns a live `NodeList` containing all the children of this node. `NodeList` being live means that if the children of the `Node` change, the `NodeList` object is automatically updated.

`Node.firstChild` | Read only

Returns a `Node` representing the first direct child node of the node, or `null` if the node has no child.

`Node.isConnected` | Read only

A boolean indicating whether or not the Node is connected (directly or indirectly) to the context object, e.g. the `Document` object in the case of the normal DOM, or the `ShadowRoot` in the case of a shadow DOM.

`Node.lastChild` | Read only

Returns a `Node` representing the last direct child node of the node, or `null` if the node has no child.

`Node.nextSibling` | Read only

Returns a `Node` representing the next node in the tree, or `null` if there isn't such node.

`Node.nodeName` | Read only

Returns a `DOMString` containing the name of the `Node`. The structure of the name will differ with the node type. E.g. An `HTMLElement` will contain the name of the corresponding tag, like `'audio'` for an `HTMLAudioElement`, a `Text` node will have the `'#text'` string, or a `Document` node will have the `'#document'` string.

`Node.nodeType` | Read only

Returns an `unsigned short` representing the type of the node. Possible values are:

Name	Value
`ELEMENT_NODE`	`1`
`ATTRIBUTE_NODE`	`2`
`TEXT_NODE`	`3`
`CDATA_SECTION_NODE`	`4`
`ENTITY_REFERENCE_NODE`	`5`
`ENTITY_NODE`	`6`
`PROCESSING_INSTRUCTION_NODE`	`7`
`COMMENT_NODE`	`8`
`DOCUMENT_NODE`	`9`
`DOCUMENT_TYPE_NODE`	`10`
`DOCUMENT_FRAGMENT_NODE`	`11`
`NOTATION_NODE`	`12`

`Node.nodeValue`

Returns / Sets the value of the current node.

`Node.ownerDocument` | Read only

Returns the `Document` that this node belongs to. If the node is itself a document, returns `null`.

`Node.parentNode` | Read only

Returns a `Node` that is the parent of this node. If there is no such node, like if this node is the top of the tree or if doesn't participate in a tree, this property returns `null`.

`Node.parentElement` | Read only

Returns an `Element` that is the parent of this node. If the node has no parent, or if that parent is not an `Element`, this property returns `null`.

`Node.previousSibling` | Read only

Returns a `Node` representing the previous node in the tree, or `null` if there isn't such node.

`Node.textContent`

Returns / Sets the textual content of an element and all its descendants.

Obsolete properties

`Node.localName` | Read only

Returns a `DOMString` representing the local part of the qualified name of an element.

> **Note:** In Firefox 3.5 and earlier, the property upper-cases the local name for HTML elements (but not XHTML elements). In later versions, this does not happen, so the property is in lower case for both HTML and XHTML.

`Node.namespaceURI` | Read only

The namespace URI of this node, or `null` if it is no namespace.

> **Note:** In Firefox 3.5 and earlier, HTML elements are in no namespace. In later versions, HTML elements are in the `http://www.w3.org/1999/xhtml/` namespace in both HTML and XML trees.

`Node.nodePrincipal` | Obsolete since Gecko 46

A `nsIPrincipal` representing the node principal.

`Node.prefix` | Read only

Is a `DOMString` representing the namespace prefix of the node, or `null` if no prefix is specified.

`Node.rootNode` | Read only

Returns a `Node` object representing the topmost node in the tree, or the current node if it's the topmost node in the tree. This has been replaced by `Node.getRootNode()`.

Methods

In addition to the properties below, `Node` inherits methods from its parent, `EventTarget`.

`Node.appendChild(childNode)`
Adds the specified `childNode` argument as the last child to the current node.
If the argument referenced an existing node on the DOM tree, the node will be detached from its current position and attached at the new position.

`Node.cloneNode()`
Clone a `Node`, and optionally, all of its contents. By default, it clones the content of the node.

`Node.compareDocumentPosition()`
Compares the position of the current node against another node in any other document.

`Node.contains()`
Returns a `Boolean` value indicating whether or not a node is a descendant of the calling node.

`Node.getBoxQuads()`
Returns a list of the node's CSS boxes relative to another node.

`Node.getRootNode()`
Returns the context object's root which optionally includes the shadow root if it is available.

`Node.hasChildNodes()`
Returns a `Boolean` indicating whether or not the element has any child nodes.

`Node.insertBefore()`
Inserts a `Node` before the reference node as a child of a specified parent node.

`Node.isDefaultNamespace()`
Accepts a namespace URI as an argument and returns a `Boolean` with a value of `true` if the namespace is the default namespace on the given node or `false` if not.

`Node.isEqualNode()`
Returns a `Boolean` which indicates whether or not two nodes are of the same type and all their defining data points match.

`Node.isSameNode()`
Returns a `Boolean` value indicating whether or not the two nodes are the same (that is, they reference the same object).

`Node.lookupPrefix()`
Returns a `DOMString` containing the prefix for a given namespace URI, if present, and `null` if not. When multiple prefixes are possible, the result is implementation-dependent.

`Node.lookupNamespaceURI()`
Accepts a prefix and returns the namespace URI associated with it on the given node if found (and `null` if not). Supplying `null` for the prefix will return the default namespace.

`Node.normalize()`
Clean up all the text nodes under this element (merge adjacent, remove empty).

`Node.removeChild()`
Removes a child node from the current element, which must be a child of the current node.

`Node.replaceChild()`
Replaces one child `Node` of the current one with the second one given in parameter.

Obsolete methods

`Node.getUserData()`
Allows a user to get some `DOMUserData` from the node.

`Node.hasAttributes()`
Returns a `Boolean` indicating if the element has any attributes, or not.

`Node.isSupported()`
Returns a `Boolean` flag containing the result of a test whether the DOM implementation implements a specific feature and this feature is supported by the specific node.

`Node.setUserData()`
Allows a user to attach, or remove, `DOMUserData` to the node.

Examples

Remove all children nested within a node

```
function removeAllChildren(element) {
  while (element.firstChild) {
    element.removeChild(element.firstChild)
  }
}
```

Sample usage

```
/* ... an alternative to document.body.innerHTML = "" ... */
removeAllChildren(document.body)
```

Recurse through child nodes

The following function recursively calls a callback function for each node contained by a root node (including the root itself):

```
function eachNode(rootNode, callback) {
    if (!callback) {
        const nodes = []
        eachNode(rootNode, function(node) {
            nodes.push(node)
        })
        return nodes
    }

    if (false === callback(rootNode)) {
        return false
    }

    if (rootNode.hasChildNodes()) {
        const nodes = rootNode.childNodes
        for (let i = 0, l = nodes.length; i < l; ++i) {
            if (false === eachNode(nodes[i], callback)) {
                return
            }
        }
    }
}
```

Syntax

```
eachNode(rootNode, callback)
```

Description

Recursively calls a function for each descendant node of *rootNode* (including the root itself).

If *callback* is omitted, the function returns an `Array` instead, which contains *rootNode* and all nodes contained within.

If *callback* is provided, and it returns `Boolean false` when called, the current recursion level is aborted, and the function resumes execution at the last parent's level. This can be used to abort loops once a node has been found (such as searching for a text node which contains a certain string).

Parameters

rootNode
The `Node` object whose descendants will be recursed through.

callback | Optional
An optional callback function that receives a `Node` as its only argument. If omitted, `eachNode` returns an `Array` of every node contained within *rootNode* (including the root itself).

Sample usage

The following example prints the `textContent` properties of each `<span>` tag in a `<div>` element named `"box"`:

```
<div id="box">
  <span>Foo</span>
  <span>Bar</span>
  <span>Baz</span>
</div>
```

```
const box = document.getElementById("box")
eachNode(box, function(node) {
  if (null != node.textContent) {
    console.log(node.textContent)
  }
})
```

The above will result in the following strings printing to the user's console:

```
"\n\t", "Foo", "\n\t", "Bar", "\n\t", "Baz"
```

Note: Whitespace forms part of a `Text` node, meaning indentation and newlines form separate `Text` between the `Element` nodes.

Realistic usage

The following demonstrates a real-world use of the `eachNode()` function: searching for text on a web-page.

We use a wrapper function named `grep` to do the searching:

```
function grep(parentNode, pattern) {
    const matches = []
    let endScan = false

    eachNode(parentNode, function(node){
        if (endScan) {
            return false
        }

        // Ignore anything which isn't a text node
        if (node.nodeType !== Node.TEXT_NODE) {
            return
        }

        if (typeof pattern === "string") {
            if (-1 !== node.textContent.indexOf(pattern)) {
                matches.push(node)
            }
        }
        else if (pattern.test(node.textContent)) {
            if (!pattern.global) {
                endScan = true
                matches = node
            }
            else {
                matches.push(node)
            }
```

```
            }
        })

        return matches
    }
```

For example, to find `Text` nodes that contain typos:

```
    const typos = ["teh", "adn", "btu", "adress", "youre", "msitakes"]
    const pattern = new RegExp("\\b(" + typos.join("|") + ")\\b", "gi")
    const mistakes = grep(document.body, pattern)
    console.log(mistakes)
```