



W3C DOM4

W3C Recommendation 19 November 2015

This version:

<http://www.w3.org/TR/2015/REC-dom-20151119/>

Latest published version:

<http://www.w3.org/TR/dom/>

Implementation report:

<http://w3c.github.io/test-results/dom/details.html>

Bug tracker:

[file a bug](#) ([open bugs](#), [old bugs](#))

Previous version:

<http://www.w3.org/TR/2015/PR-dom-20151006/>

Editors:

[Anne van Kesteren](#), [Mozilla](#) (Upstream WHATWG version)

Aryeh Gregor, [Mozilla](#) (Upstream WHATWG version)

Ms2ger, [Mozilla](#) (Upstream WHATWG version)

[Alex Russell](#), [Google](#)

[Robin Berjon](#), [W3C](#)

Please check the [errata](#) for any errors or issues reported since publication.

See also [translations](#).

[Copyright](#) © 2015 [W3C](#)® ([MIT](#), [ERCIM](#), [Keio](#), [Beihang](#)). W3C [liability](#), [trademark](#) and [document use](#) rules apply.

Abstract

DOM defines a platform-neutral model for events and node trees. DOM4 adds [Mutation Observers](#) as a replacement for [Mutation Events](#).

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the [W3C technical reports index](http://www.w3.org/TR/) at <http://www.w3.org/TR/>.

This document is published as a snapshot of the [DOM Living Standard](#) with the intent of keeping the differences from the original to a strict minimum, and only through subsetting (only things that are not implemented were removed for this publication).



WARNING

Implementers should take heed of the [old bugs list](#) in general, but more particularly of these two bugs that adversely affect interoperability (most particularly of the [Document interface](#)):

- [Bug 19431: Namespace of elements made via .createElement\(\) in XML documents must be null](#)
- [Bug 22960: Document, XMLDocument, HTMLDocument, oh my](#)

This document was published by the [HTML Working Group](#) as a Recommendation.

If you wish to make comments regarding this document in a manner that is tracked by the W3C, please submit them via using [our public issues list](#). If you cannot do this then you can also e-mail feedback to www-dom@w3.org ([subscribe](#), [archives](#)), and arrangements will be made to transpose the comments to our public bug database. All feedback is welcome.

No changes were done since the previous publication.

An extensive [test suite for this specification](#) is available. Please see the Working Group's [implementation report](#).

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document was produced by a group operating under the [5 February 2004 W3C Patent Policy](#). W3C maintains a [public list of any patent disclosures](#) made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains [Essential Claim\(s\)](#) must disclose the information in accordance with [section 6 of the W3C Patent Policy](#).

This document is governed by the [14 October 2005 W3C Process Document](#).

Table of Contents

[Goals](#)

[1 Conformance](#)

[1.1 Dependencies](#)

[1.2 Extensibility](#)

[2 Terminology](#)

[2.1 Trees](#)

[2.2 Strings](#)

[2.3 Ordered sets](#)

[2.4 Namespaces](#)

[3 Events](#)

[3.1 Introduction to "DOM Events"](#)

[3.2 Interface Event](#)

[3.3 Interface CustomEvent](#)

[3.4 Constructing events](#)

[3.5 Defining event interfaces](#)

[3.6 Interface EventTarget](#)

[3.7 Dispatching events](#)

[3.8 Firing events](#)

[4 Nodes](#)

[4.1 Introduction to "The DOM"](#)

[4.2 Node tree](#)

[4.2.1 Mutation algorithms](#)

[4.2.2 Interface NonElementParentNode](#)

[4.2.3 Interface ParentNode](#)

[4.2.4 Interface NonDocumentTypeChildNode](#)

[4.2.5 Interface ChildNode](#)

[4.2.6 Old-style collections: NodeList and HTMLCollection](#)

[4.2.6.1 Interface NodeList](#)

[4.2.6.2 Interface HTMLCollection](#)

[4.3 Mutation observers](#)

[4.3.1 Interface MutationObserver](#)

[4.3.2 Queuing a mutation record](#)

[4.3.3 Interface MutationRecord](#)

[4.3.4 Garbage collection](#)

[4.4 Interface Node](#)

[4.5 Interface Document](#)

[4.5.1 Interface DOMImplementation](#)

[4.6 Interface DocumentFragment](#)

[4.7 Interface DocumentType](#)

[4.8 Interface Element](#)

[4.8.1 Interface Attr](#)

[4.9 Interface CharacterData](#)

[4.10 Interface Text](#)

[4.11 Interface ProcessingInstruction](#)

[4.12 Interface Comment](#)

[5 Ranges](#)

[5.1 Introduction to "DOM Ranges"](#)

[5.2 Interface Range](#)

[6 Traversal](#)

[6.1 Interface NodeIterator](#)

[6.2 Interface TreeWalker](#)

[6.3 Interface NodeFilter](#)

[7 Sets](#)

[7.1 Interface DOMTokenList](#)

[7.2 Interface DOMSettableTokenList](#)

[8 Historical](#)

[8.1 DOM Events](#)

[8.2 DOM Core](#)

[8.3 DOM Ranges](#)

[8.4 DOM Traversal](#)

[A Exceptions and Errors](#)

[A.1 Exceptions](#)

[A.2 Interface DOMError](#)

[A.3 Error names](#)

[B CSS Concepts](#)

[References](#)

[Acknowledgments](#)

Goals

This specification standardizes the DOM. It does so as follows:

1. By consolidating *DOM Level 3 Core* [\[DOM3CORE\]](#), *Element Traversal* [\[ELEMENTTRAVERSAL\]](#), *Selectors API Level 2* [\[SELECTORSAPI\]](#), the "DOM Event Architecture" and "Basic Event Interfaces" chapters of *UI Events* [\[UIEVENTS\]](#) (specific type of events do not belong in the DOM Standard), and *DOM Level 2 Traversal and Range* [\[DOM2TR\]](#), and:
 - Aligning them with the JavaScript ecosystem where possible.
 - Aligning them with existing implementations.
 - Simplifying them as much as possible.
2. By moving features from the HTML Standard [\[HTML\]](#) that ake more sense to be specified as part of the DOM Standard.
3. By defining a replacement for the "Mutation Events" and "Mutation Name Event Types" chapters of *UI Events Specification (formerly DOM Level 3 Events)* [\[UIEVENTS\]](#) as the old model was problematic.

Note: The old model is expected to be removed from implementations in due course.

4. By defining new features that simplify common DOM operations.

1 Conformance

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. For readability, these words do not appear in all uppercase letters in this specification. [\[RFC2119\]](#)

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and terminate these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc.) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

User agents may impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

When a method or an attribute is said to call another method or attribute, the user agent must invoke its internal API for that attribute or method so that e.g. the author can't change the behavior by overriding attributes or methods with custom properties or functions in JavaScript.

Unless otherwise stated, string comparisons are done in a [case-sensitive](#) manner.

1.1 Dependencies

The IDL fragments in this specification must be interpreted as required for conforming IDL fragments, as described in the Web IDL specification. [\[WEBIDL\]](#)

Some of the terms used in this specification are defined in *Encoding*, *Selectors*, *Web IDL*, *XML*, and *Namespaces in XML*. [\[ENCODING\]](#) [\[SELECTORS\]](#) [\[WEBIDL\]](#) [\[XML\]](#) [\[XMLNS\]](#)

1.2 Extensibility

Vendor-specific proprietary extensions to this specification are strongly discouraged. Authors must not use such extensions, as doing so reduces interoperability and fragments the user base, allowing only users of specific user agents to access the content in question.

When extensions are needed, the DOM Standard can be updated accordingly, or a new standard can be written that hooks into the provided extensibility hooks for *applicable specifications*.

2 Terminology

The term *context object* means the object on which the algorithm, attribute getter, attribute setter, or method being discussed was called. When the [context object](#) is unambiguous, the term can be omitted.

2.1 Trees

A *tree* is a finite hierarchical tree structure. In *tree order* is preorder, depth-first traversal of a [tree](#).

An object that *participates* in a [tree](#) has a *parent*, which is either another object or null, and an ordered list of zero or more *child* objects. An object *A* whose [parent](#) is object *B* is a [child](#) of *B*.

The *root* of an object is itself, if its [parent](#) is null, or else it is the [root](#) of its [parent](#).

An object *A* is called a *descendant* of an object *B*, if either *A* is a [child](#) of *B* or *A* is a [child](#) of an object *C* that is a [descendant](#) of *B*.

An *inclusive descendant* is an object or one of its [descendants](#).

An object *A* is called an *ancestor* of an object *B* if and only if *B* is a [descendant](#) of *A*.

An *inclusive ancestor* is an object or one of its [ancestors](#).

An object *A* is called a *sibling* of an object *B*, if and only if *B* and *A* share the same non-null [parent](#).

An object *A* is *preceding* an object *B* if *A* and *B* are in the same [tree](#) and *A* comes before *B* in [tree order](#).

An object *A* is *following* an object *B* if *A* and *B* are in the same [tree](#) and *A* comes after *B* in [tree order](#).

The *first child* of an object is its first [child](#) or null if it has no [children](#).

The *last child* of an object is its last [child](#) or null if it has no [children](#).

The *previous sibling* of an object is its first [preceding sibling](#) or null if it has no [preceding sibling](#).

The *next sibling* of an object is its first [following sibling](#) or null if it has no [following sibling](#).

The *index* of an object is its number of [preceding siblings](#).

2.2 Strings

Comparing two strings in a *case-sensitive* manner means comparing them exactly, code point for code point.

Comparing two strings in a *ASCII case-insensitive* manner means comparing them exactly, code point for code point, except that the characters in the range U+0041 to U+005A (i.e.

LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z), inclusive, and the corresponding characters in the range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z), inclusive, are considered to also match.

Converting a string to ASCII uppercase means replacing all characters in the range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z), inclusive, with the corresponding characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z).

Converting a string to ASCII lowercase means replacing all characters in the range U+0041 to U+005A (i.e. LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z), inclusive, with the corresponding characters in the range U+0061 to U+007A (i.e. LATIN SMALL LETTER A to LATIN SMALL LETTER Z).

A string *pattern* is a *prefix match* for a string *s* when *pattern* is not longer than *s* and truncating *s* to *pattern*'s length leaves the two strings as matches of each other.

2.3 Ordered sets

The *ordered set parser* takes a string *input* and then runs these steps:

1. Let *position* be a pointer into *input*, initially pointing at the start of the string.
2. Let *tokens* be an ordered set of tokens, initially empty.
3. [Skip ASCII whitespace](#).
4. While *position* is not past the end of *input*:
 1. [Collect a code point sequence](#) of code points that are not [ASCII whitespace](#).
 2. If the collected string is not in *tokens*, append the collected string to *tokens*.
 3. [Skip ASCII whitespace](#).
5. Return *tokens*.

To *collect a code point sequence* of code points, run these steps:

1. Let *input* and *position* be the same variables as those of the same name in the algorithm that invoked these steps.
2. Let *result* be the empty string.
3. While *position* does not point past the end of *input* and the code point at *position* is one of *code points*, append that code point to the end of *result* and advance *position* to the next code point in *input*.
4. Return *result*.

To *skip ASCII whitespace* means to [collect a code point sequence](#) of [ASCII whitespace](#) and discard the return value.

The *ordered set serializer* takes a *set* and returns the concatenation of the strings in *set*, separated from each other by U+0020.

2.4 Namespaces

The *HTML namespace* is <http://www.w3.org/1999/xhtml>.

The *XML namespace* is <http://www.w3.org/XML/1998/namespace>.

The *XMLNS namespace* is <http://www.w3.org/2000/xmlns/>.

3 Events

3.1 Introduction to "DOM Events"

Throughout the web platform [events](#) are [dispatched](#) to objects to signal an occurrence, such as network activity or user interaction. These objects implement the [EventTarget](#) interface and can therefore add [event listeners](#) to observe [events](#) by calling [addEventListener\(\)](#):

```
obj.addEventListener("load", imgFetched)

function imgFetched(ev) {
  // great success
}
```

[Event listeners](#) can be removed by utilizing the [removeEventListener\(\)](#) method, passing the same arguments.

[Events](#) are objects too and implement the [Event](#) interface (or a derived interface). In the example above `ev` is the [event](#). It is passed as argument to [event listener](#)'s **callback** (typically a JavaScript Function as shown above). [Event listeners](#) key off the [event](#)'s [type](#) attribute value ("load" in the above example). The [event](#)'s [target](#) attribute value returns the object to which the [event](#) was [dispatched](#) (*obj* above).

Now while typically [events](#) are [dispatched](#) by the user agent as the result of user interaction or the completion of some task, applications can [dispatch events](#) themselves, commonly known as synthetic events:

```
// add an appropriate event listener
obj.addEventListener("cat", function(e) { process(e.detail) })

// create and dispatch the event
var event = new CustomEvent("cat", {"detail":{"hazcheeseburger":true}})
obj.dispatchEvent(event)
```

Apart from signaling, [events](#) are sometimes also used to let an application control what happens next in an operation. For instance as part of form submission an [event](#) whose [type](#) attribute value is "submit" is [dispatched](#). If this [event](#)'s [preventDefault\(\)](#) method is invoked, form submission will be terminated. Applications who wish to make use of this functionality through [events dispatched](#) by the application (synthetic events) can make use of the return value of the [dispatchEvent\(\)](#) method:

```
if(obj.dispatchEvent(event)) {
  // event was not canceled, time for some magic
}
```

When an [event](#) is [dispatched](#) to an object that [participates](#) in a [tree](#) (e.g. an [element](#)), it can reach [event listeners](#) on that object's [ancestors](#) too. First all object's [ancestor event listeners](#) whose **capture** variable is set to true are invoked, in [tree order](#). Second, object's own [event listeners](#) are invoked. And finally, and only if [event](#)'s [bubbles](#) attribute value is true, object's [ancestor event listeners](#) are invoked again, but now in reverse [tree order](#).

Lets look at an example of how [events](#) work in a [tree](#):

```

<!doctype html>
<html>
  <head>
    <title>Boring example</title>
  </head>
  <body>
    <p>Hello <span id=x>world</span>!</p>
    <script>
      function test(e) {
        debug(e.target, e.currentTarget, e.eventPhase)
      }
      document.addEventListener("hey", test, true)
      document.body.addEventListener("hey", test)
      var ev = new Event("hey", {bubbles:true})
      document.getElementById("x").dispatchEvent(ev)
    </script>
  </body>
</html>

```

The debug function will be invoked twice. Each time the [events](#)'s [target](#) attribute value will be the span [element](#). The first time [currentTarget](#) attribute's value will be the [document](#), the second time the body [element](#). [eventPhase](#) attribute's value switches from [CAPTURING_PHASE](#) to [BUBBLING_PHASE](#). If an [event listener](#) was registered for the span [element](#), [eventPhase](#) attribute's value would have been [AT_TARGET](#).

3.2 Interface [Event](#)

IDL	<pre> [Constructor(DOMString type, optional EventInit eventInitDict), Exposed=(Window,Worker)] interface Event { readonly attribute DOMString type; readonly attribute EventTarget? target; readonly attribute EventTarget? currentTarget; const unsigned short NONE = 0; const unsigned short CAPTURING_PHASE = 1; const unsigned short AT_TARGET = 2; const unsigned short BUBBLING_PHASE = 3; readonly attribute unsigned short eventPhase; void stopPropagation(); void stopImmediatePropagation(); readonly attribute boolean bubbles; readonly attribute boolean cancelable; void preventDefault(); readonly attribute boolean defaultPrevented; [Unforgeable] readonly attribute boolean isTrusted; readonly attribute DOMTimeStamp timeStamp; void initEvent(DOMString type, boolean <i>bubbles</i>, boolean <i>cancelable</i>); }; dictionary EventInit { boolean bubbles = false; boolean cancelable = false; }; </pre>
------------	--

An *event* allows for signaling that something has occurred. E.g. that an image has completed downloading. It is represented by the [Event](#) interface or an interface that inherits

from the `Event` interface.

This box is non-normative. Implementation requirements are given below this box.
`event = new Event(type [, eventInitDict])`

Returns a new *event* whose `type` attribute value is set to *type*. The optional *eventInitDict* argument allows for setting the `bubbles` and `cancelable` attributes via object members of the same name.

***event* . type**

Returns the type of *event*, e.g. "click", "hashchange", or "submit".

***event* . target**

Returns the object to which *event* is [dispatched](#).

***event* . currentTarget**

Returns the object whose [event listener](#)'s **callback** is currently being invoked.

***event* . eventPhase**

Returns the [event](#)'s phase, which is one of `NONE`, `CAPTURING_PHASE`, `AT_TARGET`, and `BUBBLING_PHASE`.

***event* . stopPropagation()**

When [dispatched](#) in a [tree](#), invoking this method prevents *event* from reaching any objects other than the current object.

***event* . stopImmediatePropagation()**

Invoking this method prevents *event* from reaching any registered [event listeners](#) after the current one finishes running and, when [dispatched](#) in a [tree](#), also prevents *event* from reaching any other objects.

***event* . bubbles**

Returns true or false depending on how *event* was initialized. True if *event*'s goes through its `target` attribute value's [ancestors](#) in reverse [tree order](#), and false otherwise.

***event* . cancelable**

Returns true or false depending on how *event* was initialized. Its return value does not always carry meaning, but true can indicate that part of the operation during which *event* was [dispatched](#), can be canceled by invoking the [preventDefault\(\)](#) method.

***event* . preventDefault()**

If invoked when the `cancelable` attribute value is true, signals to the operation that caused *event* to be [dispatched](#) that it needs to be canceled.

***event* . defaultPrevented**

Returns true if [preventDefault\(\)](#) was invoked while the `cancelable` attribute value is true, and false otherwise.

***event* . isTrusted**

Returns true if *event* was [dispatched](#) by the user agent, and false otherwise.

***event* . timeStamp**

Returns the creation time of *event* as the number of milliseconds that passed since 00:00:00 UTC on 1 January 1970.

The *type* attribute must return the value it was initialized to. When an [event](#) is created the attribute must be initialized to the empty string.

The *target* and *currentTarget* attributes must return the values they were initialized to. When an [event](#) is created the attributes must be initialized to null.

The *eventPhase* attribute must return the value it was initialized to, which must be one of the following:

***NONE* (numeric value 0)**

[Events](#) not currently [dispatched](#) are in this phase.

***CAPTURING_PHASE* (numeric value 1)**

When an [event](#) is [dispatched](#) to an object that [participates](#) in a [tree](#) it will be in this phase before it reaches its [target](#) attribute value.

***AT_TARGET* (numeric value 2)**

When an [event](#) is [dispatched](#) it will be in this phase on its [target](#) attribute value.

***BUBBLING_PHASE* (numeric value 3)**

When an [event](#) is [dispatched](#) to an object that [participates](#) in a [tree](#) it will be in this phase after it reaches its [target](#) attribute value.

Initially the attribute must be initialized to [NONE](#).

Each [event](#) has the following associated flags that are all initially unset:

- *stop propagation flag*
- *stop immediate propagation flag*
- *canceled flag*
- *initialized flag*
- *dispatch flag*

The *stopPropagation()* method must set the [stop propagation flag](#).

The *stopImmediatePropagation()* method must set both the [stop propagation flag](#) and [stop immediate propagation flag](#).

The *bubbles* and *cancelable* attributes must return the values they were initialized to.

The *preventDefault()* method must set the [canceled flag](#) if the [cancelable](#) attribute value is true.

The *defaultPrevented* attribute must return true if the [canceled flag](#) is set and false otherwise.

The *isTrusted* attribute must return the value it was initialized to. When an [event](#) is created the attribute must be initialized to false.

The *timeStamp* attribute must return the value it was initialized to. When an [event](#) is created the attribute must be initialized to the number of milliseconds that have passed since 00:00:00 UTC on 1 January 1970, ignoring leap seconds.

To *initialize* an *event*, with *type*, *bubbles*, and *cancelable*, run these steps:

1. Set the [initialized flag](#).
2. Unset the [stop propagation flag](#), [stop immediate propagation flag](#), and [canceled flag](#).
3. Set the [isTrusted](#) attribute to false.
4. Set the [target](#) attribute to null.
5. Set the [type](#) attribute to *type*.
6. Set the [bubbles](#) attribute to *bubbles*.
7. Set the [cancelable](#) attribute to *cancelable*.

The *initEvent(type, bubbles, cancelable)* method, when invoked, must run these steps:

1. If [context object](#)'s [dispatch flag](#) is set, terminate these steps.
2. [Initialize](#) the [context object](#) with *type*, *bubbles*, and *cancelable*.

Note: As [events](#) have constructors [initEvent\(\)](#) is superfluous. However, it has to be supported for legacy content.

3.3 Interface [CustomEvent](#)

IDL `[Constructor(DOMString type, optional CustomEventInit eventInitDict),
Exposed=(Window,Worker)]
interface CustomEvent : Event {
 readonly attribute any detail;

 void initCustomEvent(DOMString type, boolean bubbles, boolean cancelable,
 any detail);
};

dictionary CustomEventInit : EventInit {
 any detail = null;
};`

[Events](#) using the [CustomEvent](#) interface can be used to carry custom data.

This box is non-normative. Implementation requirements are given below this box.

`event = new CustomEvent(type [, eventInitDict])`

Works analogously to the constructor for [Event](#) except that the optional *eventInitDict* argument now allows for setting the *detail* attribute too.

`event . detail`

Returns any custom data *event* was created with. Typically used for synthetic events.

The *detail* attribute must return the value it was initialized to.

The *initCustomEvent(type, bubbles, cancelable, detail)* method must, when invoked, run these steps:

1. If [context object](#)'s [dispatch flag](#) is set, terminate these steps.
2. [Initialize](#) the [context object](#) with *type*, *bubbles*, and *cancelable*.
3. Set [context object](#)'s [detail](#) attribute to *detail*.

3.4 Constructing events

When a *constructor* of the [Event](#) interface, or of an interface that inherits from the [Event](#) interface, is invoked, these steps must be run:

1. Create an [event](#) that uses the interface the constructor was invoked upon.
2. Set its [initialized flag](#).
3. Initialize the [type](#) attribute to the *type* argument.
4. If there is an *eventInitDict* argument then for each [dictionary member](#) defined therein find the attribute on [event](#) whose [identifier](#) matches the key of the [dictionary member](#) and then set the attribute to the value of that [dictionary member](#).
5. Return the [event](#).

3.5 Defining event interfaces

In general, when defining a new interface that inherits from [Event](#) please always ask feedback from the WHATWG or the W3C www-dom@w3.org mailing list.

The [CustomEvent](#) interface can be used as starting point. However, do not introduce any *init*Event()* methods as they are redundant with constructors. Interfaces that inherit from the [Event](#) interface that have such a method only have it for historical reasons.

3.6 Interface [EventTarget](#)

IDL	<pre>[Exposed=(Window,Worker)] interface EventTarget { void addEventListener(DOMString type, EventListener? callback, optional boolean <i>capture</i> = false); void removeEventListener(DOMString type, EventListener? callback, optional</pre>
------------	--


```

boolean capture = false);
    boolean dispatchEvent(Event event);
};

callback interface EventListener {
    void handleEvent(Event event);
};

```

EventTarget is an object to which an event is dispatched when something has occurred. Each EventTarget has an associated list of event listeners.

An *event listener* associates a callback with a specific event. Each event listener consists of a **type** (of the event), **callback**, and **capture** variable.

Note: The callback is named EventListener for historical reasons. As can be seen from the definition above, an event listener is a more broad concept.

This box is non-normative. Implementation requirements are given below this box.

target . addEventListener(type, callback [, capture = false])

Appends an event listener for events whose type attribute value is *type*. The **callback** argument sets the **callback** that will be invoked when the event is dispatched. When set to true, the *capture* argument prevents **callback** from being invoked if the event's eventPhase attribute value is BUBBLING_PHASE. When false, **callback** will not be invoked when event's eventPhase attribute value is CAPTURING_PHASE. Either way, **callback** will be invoked when event's eventPhase attribute value is AT_TARGET.

The event listener is appended to *target*'s list of event listeners and is not appended if it is a duplicate, i.e. having the same **type**, **callback**, and **capture** values.

target . removeEventListener(type, callback [, capture = false])

Remove the event listener in *target*'s list of event listeners with the same *type*, **callback**, and **capture**.

target . dispatchEvent(event)

Dispatches a synthetic event *event* to *target* and returns true if either *event*'s cancelable attribute value is false or its preventDefault() method was not invoked, and false otherwise.

The addEventListener(type, callback, capture) method must run these steps:

1. If *callback* is null, terminate these steps.
2. Append an event listener to the associated list of event listeners with **type** set to *type*, **callback** set to *callback*, and **capture** set to *capture*, unless there already is an event listener in that list with the same **type**, **callback**, and **capture**.

The removeEventListener(type, callback, capture) method must run these steps:

1. Remove an event listener from the associated list of event listeners, whose **type** is *name*, **callback** is *callback*, and **capture** is *capture*.

The `dispatchEvent(event)` method must run these steps:

1. If `event`'s [dispatch flag](#) is set, or if its [initialized flag](#) is not set, [throw](#) an `"InvalidStateError"` exception.
2. Initialize `event`'s [isTrusted](#) attribute to false.
3. [Dispatch](#) the `event` and return the value that returns.

3.7 Dispatching events

To *dispatch* an [event](#) to a given object, with an optional *target override*, run these steps:

1. Let `event` be the [event](#) that is dispatched.
2. Set `event`'s [dispatch flag](#).
3. Initialize `event`'s [target](#) attribute to *target override*, if it is given, and the object to which `event` is dispatched otherwise.
4. If `event`'s [target](#) attribute value is [participating](#) in a [tree](#), let *event path* be a static ordered list of all its [ancestors](#) in [tree order](#), and let *event path* be the empty list otherwise.
5. Initialize `event`'s [eventPhase](#) attribute to `CAPTURING_PHASE`.
6. For each object in *event path*, [invoke](#) its [event listeners](#) with event `event`, as long as `event`'s [stop propagation flag](#) is unset.
7. Initialize `event`'s [eventPhase](#) attribute to `AT_TARGET`.
8. [Invoke](#) the [event listeners](#) of `event`'s [target](#) attribute value with `event`, if `event`'s [stop propagation flag](#) is unset.
9. If `event`'s [bubbles](#) attribute value is true, run these substeps:
 1. Reverse the order of *event path*.
 2. Initialize `event`'s [eventPhase](#) attribute to `BUBBLING_PHASE`.
 3. For each object in *event path*, [invoke](#) its [event listeners](#), with event `event` as long as `event`'s [stop propagation flag](#) is unset.
10. Unset `event`'s [dispatch flag](#).
11. Initialize `event`'s [eventPhase](#) attribute to `NONE`.
12. Initialize `event`'s [currentTarget](#) attribute to null.
13. Return false if `event`'s [canceled flag](#) is set, and true otherwise.

To *invoke* the [event listeners](#) for an object with an event run these steps:

1. Let `event` be the [event](#) for which the [event listeners](#) are invoked.

2. Let *listeners* be a copy of the [event listeners](#) associated with the object for which these steps are run.
3. Initialize *event*'s [currentTarget](#) attribute to the object for which these steps are run.
4. Then run these substeps for each [event listener](#) in *listeners*:
 1. If *event*'s [stop immediate propagation flag](#) is set, terminate the [invoke](#) algorithm.
 2. Let *listener* be the [event listener](#).
 3. If *event*'s [type](#) attribute value is not *listener*'s **type**, terminate these substeps (and run them for the next [event listener](#)).
 4. If *event*'s [eventPhase](#) attribute value is [CAPTURING_PHASE](#) and *listener*'s **capture** is false, terminate these substeps (and run them for the next [event listener](#)).
 5. If *event*'s [eventPhase](#) attribute value is [BUBBLING_PHASE](#) and *listener*'s **capture** is true, terminate these substeps (and run them for the next [event listener](#)).
 6. Call *listener*'s **callback**'s `handleEvent`, with the event passed to this algorithm as the first argument and *event*'s [currentTarget](#) attribute value as [callback this value](#). If this throws any exception, [report the exception](#).

3.8 Firing events

To *fire an event named e* means that a new [event](#) using the [Event](#) interface, with its [type](#) attribute initialized to *e*, and its [isTrusted](#) attribute initialized to true, is to be [dispatched](#) to the given object.

Note: *Fire in the context of DOM is short for creating, initializing, and [dispatching an event](#). [Fire an event](#) makes that process easier to write down. If the [event](#) needs its `bubbles` or `cancelable` attribute initialized, one could write "[fire an event](#) named `submit` with its `cancelable` attribute initialized to true".*

4 Nodes

4.1 Introduction to "The DOM"

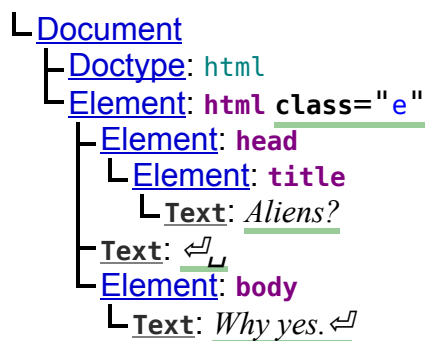
In its original sense, "The DOM" is an API for accessing and manipulating documents (in particular, HTML and XML documents). In this specification, the term "document" is used for any markup-based resource, ranging from short static documents to long essays or reports with rich multimedia, as well as to fully-fledged interactive applications.

These documents are presented as a [node tree](#). Some of the [nodes](#) in the [tree](#) can have [children](#), while others are always leaves.

To illustrate, consider this HTML document:

```
<!DOCTYPE html>
<html class=e>
  <head><title>Aliens?</title></head>
  <body>Why yes.</body>
</html>
```

It is represented as follows:



Note that, due to the magic that is [HTML parsing](#), not all [ASCII whitespace](#) were turned into [Text nodes](#), but the general concept is clear. Markup goes in, a [tree](#) of [nodes](#) comes out.

Note: The most excellent [Live DOM Viewer](#) can be used to explore this matter in more detail.

4.2 Node tree

Objects implementing the [Document](#), [DocumentFragment](#), [DocumentType](#), [Element](#), [Text](#), [ProcessingInstruction](#), or [Comment](#) interface (simply called *nodes*) [participate](#) in a [tree](#), simply named the *node tree*.

A [node tree](#) is constrained as follows, expressed as a relationship between the type of [node](#) and its allowed [children](#):

Document

In [tree order](#):

1. Zero or more nodes each of which is either [ProcessingInstruction](#) or [Comment](#).
2. Optionally one [DocumentType](#) node.

3. Zero or more nodes each of which is either ProcessingInstruction or Comment.
4. Optionally one Element node.
5. Zero or more nodes each of which is either ProcessingInstruction or Comment.

DocumentFragment

Element

Zero or more nodes each of which is one of Element, ProcessingInstruction, Comment, or Text.

DocumentType

Text

ProcessingInstruction

Comment

None.

The *length* of a node depends on *node*:

↪ DocumentType

Zero.

↪ Text

↪ ProcessingInstruction

↪ Comment

Its length attribute value.

↪ **Any other node**

Its number of children.

A node is considered *empty* if its length is zero.

4.2.1 Mutation algorithms

To ensure *pre-insertion validity* of a *node* into a *parent* before a *child*, run these steps:

1. If *parent* is not a Document, DocumentFragment, or Element node, throw a "HierarchyRequestError".
2. If *node* is a host-including inclusive ancestor of *parent*, throw a "HierarchyRequestError".
3. If *child* is not null and its parent is not *parent*, throw a "NotFoundError" exception.
4. If *node* is not a DocumentFragment, DocumentType, Element, Text, ProcessingInstruction, or Comment node, throw a "HierarchyRequestError".
5. If either *node* is a Text node and *parent* is a document, or *node* is a doctype and *parent* is not a document, throw a "HierarchyRequestError".
6. If *parent* is a document, and any of the statements below, switched on *node*, are true, throw a "HierarchyRequestError".

↪ DocumentFragment node

If *node* has more than one element child or has a Text node child.

Otherwise, if *node* has one [element child](#) and either *parent* has an [element child](#), *child* is a [doctype](#), or *child* is not null and a [doctype](#) is [following child](#).

↪ [element](#)

parent has an [element child](#), *child* is a [doctype](#), or *child* is not null and a [doctype](#) is [following child](#).

↪ [doctype](#)

parent has a [doctype child](#), an [element](#) is [preceding child](#), or *child* is null and *parent* has an [element child](#).

To pre-insert a *node* into a *parent* before a *child*, run these steps:

1. [Ensure pre-insertion validity](#) of *node* into *parent* before *child*.
2. Let *reference child* be *child*.
3. If *reference child* is *node*, set it to *node*'s [next sibling](#).
4. [Adopt](#) *node* into *parent*'s [node document](#).
5. [Insert](#) *node* into *parent* before *reference child*.
6. Return *node*.

[Specifications](#) may define *insertion steps* for all or some [nodes](#). The algorithm is passed *newNode* as indicated in the [insert](#) algorithm below.

To *insert* a *node* into a *parent* before a *child* with an optional *suppress observers flag*, run these steps:

1. Let *count* be the number of [children](#) of *node* if it is a [DocumentFragment node](#), and one otherwise.
2. If *child* is non-null, run these substeps:
 1. For each [range](#) whose [start node](#) is *parent* and [start offset](#) is greater than *child*'s [index](#), increase its [start offset](#) by *count*.
 2. For each [range](#) whose [end node](#) is *parent* and [end offset](#) is greater than *child*'s [index](#), increase its [end offset](#) by *count*.
3. Let *nodes* be *node*'s [children](#) if *node* is a [DocumentFragment node](#), and a list containing solely *node* otherwise.
4. If *node* is a [DocumentFragment node](#), [queue a mutation record](#) of "childList" for *node* with removedNodes *nodes*.

Note: This step intentionally does not pay attention to the suppress observers flag.

5. If *node* is a [DocumentFragment node](#), [remove](#) its [children](#) with the *suppress observers flag* set.

6. If *suppress observers flag* is unset, [queue a mutation record](#) of "childList" for *parent* with addedNodes *nodes*, nextSibling *child*, and previousSibling *child*'s [previous sibling](#) or *parent*'s [last child](#) if *child* is null.
7. For each *newNode* in *nodes*, in [tree order](#), run these substeps:
 1. Insert *newNode* into *parent* before *child* or at the end of *parent* if *child* is null.
 2. Run the [insertion steps](#) with *newNode*.

To *append* a *node* to a *parent*, [pre-insert](#) *node* into *parent* before null.

To *replace* a *child* with *node* within a *parent*, run these steps:

1. If *parent* is not a [Document](#), [DocumentFragment](#), or [Element](#) [node](#), [throw](#) a "[HierarchyRequestError](#)".
2. If *node* is a [host-including inclusive ancestor](#) of *parent*, [throw](#) a "[HierarchyRequestError](#)".
3. If *child*'s [parent](#) is not *parent*, [throw](#) a "[NotFoundError](#)" exception.
4. If *node* is not a [DocumentFragment](#), [DocumentType](#), [Element](#), [Text](#), [ProcessingInstruction](#), or [Comment](#) [node](#), [throw](#) a "[HierarchyRequestError](#)".
5. If either *node* is a [Text](#) [node](#) and *parent* is a [document](#), or *node* is a [doctype](#) and *parent* is not a [document](#), [throw](#) a "[HierarchyRequestError](#)".
6. If *parent* is a [document](#), and any of the statements below, switched on *node*, are true, [throw](#) a "[HierarchyRequestError](#)".
 - ↪ [DocumentFragment](#) [node](#)
 If *node* has more than one [element child](#) or has a [Text](#) [node child](#).

 Otherwise, if *node* has one [element child](#) and either *parent* has an [element child](#) that is not *child* or a [doctype](#) is [following](#) *child*.
 - ↪ [element](#)
parent has an [element child](#) that is not *child* or a [doctype](#) is [following](#) *child*.
 - ↪ [doctype](#)
parent has a [doctype child](#) that is not *child*, or an [element](#) is [preceding](#) *child*.

Note: The above statements differ from the [pre-insert algorithm](#).

7. Let *reference child* be *child*'s [next sibling](#).
8. If *reference child* is *node*, set it to *node*'s [next sibling](#).
9. [Adopt](#) *node* into *parent*'s [node document](#).
10. [Remove](#) *child* from its *parent* with the *suppress observers flag* set.
11. [Insert](#) *node* into *parent* before *reference child* with the *suppress observers flag* set.

12. Let *nodes* be *node*'s [children](#) if *node* is a [DocumentFragment node](#), and a list containing solely *node* otherwise.
13. [Queue a mutation record](#) of "childList" for target *parent* with addedNodes *nodes*, removedNodes a list solely containing *child*, nextSibling *reference child*, and previousSibling *child*'s [previous sibling](#).
14. Return *child*.

To *replace all* with a *node* within a *parent*, run these steps:

1. If *node* is not null, [adopt](#) *node* into *parent*'s [node document](#).
2. Let *removedNodes* be *parent*'s [children](#).
3. Let *addedNodes* be the empty list if *node* is null, *node*'s [children](#) if *node* is a [DocumentFragment node](#), and a list containing *node* otherwise.
4. [Remove](#) all *parent*'s [children](#), in [tree order](#), with the *suppress observers flag* set.
5. If *node* is not null, [insert](#) *node* into *parent* before null with the *suppress observers flag* set.
6. [Queue a mutation record](#) of "childList" for *parent* with addedNodes *addedNodes* and removedNodes *removedNodes*.

Note: This algorithm does not make any checks with regards to the [node tree constraints](#). Specification authors need to use it wisely.

To *pre-remove* a *child* from a *parent*, run these steps:

1. If *child*'s [parent](#) is not *parent*, [throw](#) a "NotFoundError" exception.
2. [Remove](#) *child* from *parent*.
3. Return *child*.

[Specifications](#) may define *removing steps* for all or some [nodes](#). The algorithm is passed *removedNode*, *oldParent*, and *oldPreviousSibling*, as indicated in the [remove](#) algorithm below.

To *remove* a *node* from a *parent* with an optional *suppress observers flag* set, run these steps:

1. Let *index* be *node*'s [index](#).
2. For each [range](#) whose [start node](#) is an [inclusive descendant](#) of *node*, set its [start](#) to (*parent*, *index*).
3. For each [range](#) whose [end node](#) is an [inclusive descendant](#) of *node*, set its [end](#) to (*parent*, *index*).
4. For each [range](#) whose [start node](#) is *parent* and [start offset](#) is greater than *index*, decrease its [start offset](#) by one.

5. For each [range](#) whose [end node](#) is *parent* and [end offset](#) is greater than *index*, decrease its [end offset](#) by one.
6. Let *oldPreviousSibling* be *node*'s [previous sibling](#).
7. If *suppress observers flag* is unset, [queue a mutation record](#) of "childList" for *parent* with removedNodes a list solely containing *node*, nextSibling *node*'s [next sibling](#), and previousSibling *oldPreviousSibling*.
8. For each [ancestor](#) *ancestor* of *node*, if *ancestor* has any [registered observers](#) whose **options**'s subtree is true, then for each such [registered observer](#) *registered*, append a [transient registered observer](#) whose **observer** and **options** are identical to those of *registered* and **source** which is *registered* to *node*'s list of [registered observers](#).
9. Remove *node* from its *parent*.
10. Run the [removing steps](#) with *node*, *parent*, and *oldPreviousSibling*.

4.2.2 Interface [NonElementParentNode](#)

Note: The [getElementById\(\)](#) method is not on [elements](#) for compatibility with older versions of jQuery. If a time comes where that version of jQuery has disappeared, we might be able to support it.

IDL

```
[NoInterfaceObject,
  Exposed=Window]
interface NonElementParentNode {
  Element? getElementById(DOMString elementId);
};
Document implements NonElementParentNode;
DocumentFragment implements NonElementParentNode;
```

This box is non-normative. Implementation requirements are given below this box.

node . [getElementById](#)(*elementId*)

Returns the first [element](#) within *node*'s [descendants](#) whose [ID](#) is *elementId*.

The [getElementById\(*elementId*\)](#) method must return the first [element](#), in [tree order](#), within [context object](#)'s [descendants](#), whose [ID](#) is *elementId*, and null if there is no such [element](#) otherwise.

4.2.3 Interface [ParentNode](#)

The *mutation method macro*:

1. Let *node* be null.
2. Replace each string in *nodes* with a [Text node](#) whose [data](#) is the string value.
3. If *nodes* contains more than one [node](#), set *node* to a new [DocumentFragment](#) and [append](#) each [node](#) in *nodes* to it. Rethrow any exceptions.

Otherwise, set *node* to the single [node](#) *nodes* contains.

IDL

```
[NoInterfaceObject,
 Exposed=Window]
interface ParentNode {
  [SameObject] readonly attribute HTMLCollection children;
  readonly attribute Element? firstElementChild;
  readonly attribute Element? lastElementChild;
  readonly attribute unsigned long childElementCount;

  Element? querySelector(DOMString selectors);
  [NewObject] NodeList querySelectorAll(DOMString selectors);
};
Document implements ParentNode;
DocumentFragment implements ParentNode;
Element implements ParentNode;
```

This box is non-normative. Implementation requirements are given below this box.

collection = node . children

Returns the [child elements](#).

element = node . firstElementChild

Returns the first [child](#) that is an [element](#), and null otherwise.

element = node . lastElementChild

Returns the last [child](#) that is an [element](#), and null otherwise.

node . querySelector(selectors)

Returns the first [element](#) that is a [descendant](#) of *node* that matches *selectors*.

node . querySelectorAll(selectors)

Returns all [element descendants](#) of *node* that match *selectors*.

The *children* attribute must return an [HTMLCollection](#) [collection](#) rooted at the [context object](#) matching only [element children](#).

The *firstElementChild* attribute must return the first [child](#) that is an [element](#), and null otherwise.

The *lastElementChild* attribute must return the last [child](#) that is an [element](#), and null otherwise.

The *childElementCount* attribute must return the number of [children](#) of the [context object](#) that are [elements](#).

To *match a relative selectors string* *relativeSelectors* against a *set*, run these steps:

1. Let *s* be the result of [parse a relative selector](#) from *relativeSelectors* against *set*.
[\[SELECTORS\]](#)
2. If *s* is failure, [throw](#) a "SyntaxError".

3. Return the result of [evaluate a selector](#) *s* using [:scope elements](#) set. [\[SELECTORS\]](#)

To *scope-match* a *selectors string* *selectors* against a *node*, run these steps:

1. Let *s* be the result of [parse a selector](#) *selectors*. [\[SELECTORS\]](#)
2. If *s* is failure, [throw](#) a "SyntaxError".
3. Return the result of [evaluate a selector](#) *s* against *node*'s [root](#) using [scoping root](#) *node* and scoping method [scope-filtered](#). [\[SELECTORS\]](#).

The *querySelector(selectors)* method must return the first result of running [scope-match a selectors string](#) *selectors* against the [context object](#), and null if the result is an empty list otherwise.

The *querySelectorAll(selectors)* method must return the [static](#) result of running [scope-match a selectors string](#) *selectors* against the [context object](#).

4.2.4 Interface [NonDocumentTypeChildNode](#)

Note: The [previousElementSibling](#) and [nextElementSibling](#) attributes have been removed from [DocumentType](#) nodes for compatibility reasons. If these additions are deemed compatible enough in the future, they could be reinstated.

IDL

```
[NoInterfaceObject,
 Exposed=Window]
interface NonDocumentTypeChildNode {
  readonly attribute Element? previousElementSibling;
  readonly attribute Element? nextElementSibling;
};
Element implements NonDocumentTypeChildNode;
CharacterData implements NonDocumentTypeChildNode;
```

This box is non-normative. Implementation requirements are given below this box.

element = node . previousElementSibling

Returns the first [preceding sibling](#) that is an [element](#), and null otherwise.

element = node . nextElementSibling

Returns the first [following sibling](#) that is an [element](#), and null otherwise.

The *previousElementSibling* attribute must return the first [preceding sibling](#) that is an [element](#), and null otherwise.

The *nextElementSibling* attribute must return the first [following sibling](#) that is an [element](#), and null otherwise.

4.2.5 Interface [ChildNode](#)

IDL

```
[NoInterfaceObject,
 Exposed=Window]
```

```

interface ChildNode {
    void remove();
};
DocumentType implements ChildNode;
Element implements ChildNode;
CharacterData implements ChildNode;

```

This box is non-normative. Implementation requirements are given below this box.

`node . remove()`

Removes *node*.

The `remove()` method must run these steps:

1. If the [context object](#) does not have a [parent](#), terminate these steps.
2. **Remove** the [context object](#) from the [context object's](#) [parent](#).

4.2.6 Old-style collections: [NodeList](#) and [HTMLCollection](#)

A *collection* is an object that represents a lists of DOM nodes. A [collection](#) can be either *live* or *static*. Unless otherwise stated, a [collection](#) must be [live](#).

If a [collection](#) is [live](#), then the attributes and methods on that object must operate on the actual underlying data, not a snapshot of the data.

When a [collection](#) is created, a filter and a root are associated with it.

The [collection](#) then *represents* a view of the subtree rooted at the [collection's](#) root, containing only nodes that match the given filter. The view is linear. In the absence of specific requirements to the contrary, the nodes within the [collection](#) must be sorted in [tree order](#).

4.2.6.1 Interface [NodeList](#)

A [NodeList](#) object is a [collection](#) of [nodes](#).

IDL

```

[Exposed=Window]
interface NodeList {
    getter Node? item(unsigned long index);
    readonly attribute unsigned long length;
    iterable<Node>;
};

```

This box is non-normative. Implementation requirements are given below this box.

`collection . length`

Returns the number of [nodes](#) in the [collection](#).

`element = collection . item(index)`

`element = collection[index]`

Returns the [node](#) with index *index* from the [collection](#). The [nodes](#) are sorted in [tree order](#).

The object's [supported property indices](#) are the numbers in the range zero to one less than the number of nodes [represented by the collection](#). If there are no such elements, then there are no [supported property indices](#).

The *length* attribute must return the number of nodes [represented by the collection](#).

The *item(index)* method must return the *index*th node in the [collection](#). If there is no *index*th node in the [collection](#), then the method must return null.

4.2.6.2 Interface *HTMLCollection*

```
IDL [Exposed=Window]
interface HTMLCollection {
  readonly attribute unsigned long length;
  getter Element? item(unsigned long index);
  getter Element? namedItem(DOMString name);
};
```

An *HTMLCollection* object is a [collection](#) of [elements](#).

Note: *Elements is the better solution for representing a [collection](#) of [elements](#). *HTMLCollection* is an historical artifact we cannot rid the web of.*

This box is non-normative. Implementation requirements are given below this box.

collection . length

Returns the number of [elements](#) in the [collection](#).

element = collection . item(index)
element = collection[index]

Returns the [element](#) with index *index* from the [collection](#). The [elements](#) are sorted in [tree order](#).

element = collection . namedItem(name)
element = collection[name]

Returns the first [element](#) with [ID](#) or name *name* from the collection.

The object's [supported property indices](#) are the numbers in the range zero to one less than the number of nodes [represented by the collection](#). If there are no such elements, then there are no [supported property indices](#).

The *length* attribute must return the number of nodes [represented by the collection](#).

The *item(index)* method must return the *index*th [element](#) in the [collection](#). If there is no *index*th [element](#) in the [collection](#), then the method must return null.

The [supported property names](#), all [unenumerable](#), are the values from the list returned by these steps:

1. Let *result* be an empty list.
2. For each *element* [represented by the collection](#), in [tree order](#), run these substeps:
 1. If *element* has an [ID](#) which is neither the empty string nor is in *result*, append *element's ID* to *result*.
 2. If *element* is in the [HTML namespace](#) and [has](#) a [name attribute](#) whose [value](#) is neither the empty string nor is in *result*, append *element's name attribute value* to *result*.
3. Return *result*.

The *namedItem(key)* method must run these steps:

1. If *key* is the empty string, return null.
2. Return the first [element](#) in the [collection](#) for which at least one of the following is true:
 - it has an [ID](#) which is *key*.
 - it [has](#) a [name attribute](#) whose [value](#) is *key*;
 or null if there is no such [element](#).

4.3 Mutation observers

Each [unit of related similar-origin browsing contexts](#) has a *mutation observer compound microtask queued flag* and an associated list of [MutationObserver](#) objects which is initially empty. [\[HTML\]](#)

To *queue a mutation observer compound microtask*, run these steps:

1. If [mutation observer compound microtask queued flag](#) is set, terminate these steps.
2. Set [mutation observer compound microtask queued flag](#).
3. [Queue](#) a [compound microtask](#) to [notify mutation observers](#).

To *notify mutation observers*, run these steps:

1. Unset [mutation observer compound microtask queued flag](#).
2. Let *notify list* be a copy of [unit of related similar-origin browsing contexts](#)'s list of [MutationObserver](#) objects.
3. For each [MutationObserver](#) object *mo* in *notify list*, [execute a compound microtask subtask](#) to run these steps: [\[HTML\]](#)
 1. Let *queue* be a copy of *mo's* [record queue](#).
 2. Empty *mo's* [record queue](#).
 3. Remove all [transient registered observers](#) whose **observer** is *mo*.

4. If *queue* is non-empty, call *mo*'s [callback](#) with *queue* as first argument, and *mo* (itself) as second argument and [callback this value](#). If this throws an exception, [report the exception](#).

Each [node](#) has an associated list of [registered observers](#).

A *registered observer* consists of an **observer** (a [MutationObserver](#) object) and **options** (a [MutationObserverInit](#) dictionary). A *transient registered observer* is a specific type of [registered observer](#) that has a **source** which is a [registered observer](#).

4.3.1 Interface [MutationObserver](#)

IDL

```
[Constructor(MutationCallback callback)]
interface MutationObserver {
  void observe(Node target, MutationObserverInit options);
  void disconnect();
  sequence<MutationRecord> takeRecords();
};

callback MutationCallback = void (sequence<MutationRecord> mutations,
MutationObserver observer);

dictionary MutationObserverInit {
  boolean childList = false;
  boolean attributes;
  boolean characterData;
  boolean subtree = false;
  boolean attributeOldValue;
  boolean characterDataOldValue;
  sequence<DOMString> attributeFilter;
};
```

A [MutationObserver](#) object can be used to observe mutations to the [tree](#) of [nodes](#).

Each [MutationObserver](#) object has these associated concepts:

- A *callback* set on creation.
- A list of [nodes](#) on which it is a [registered observer](#)'s **observer** that is initially empty.
- A list of [MutationRecord](#) objects called the *record queue* that is initially empty.

This box is non-normative. Implementation requirements are given below this box.

observer = new [MutationObserver](#)(*callback*)

Constructs a [MutationObserver](#) object and sets its [callback](#) to *callback*. The *callback* is invoked with a list of [MutationRecord](#) objects as first argument and the constructed [MutationObserver](#) object as second argument. It is invoked after [nodes](#) registered with the [observe\(\)](#) method, are mutated.

observer . [observe](#)(*target*, *options*)

Instructs the user agent to observe a given *target* (a [node](#)) and report any mutations based on the criteria given by *options* (an object).

The *options* argument allows for setting mutation observation options via object members. These are the object members that can be used:

childList

Set to true if mutations to *target's* [children](#) are to be observed.

attributes

Set to true if mutations to *target's* [attributes](#) are to be observed. Can be omitted if `attributeOldValue` and/or `attributeFilter` is specified.

characterData

Set to true if mutations to *target's* [data](#) are to be observed. Can be omitted if `characterDataOldValue` is specified.

subtree

Set to true if mutations to not just *target*, but also *target's* [descendants](#) are to be observed.

attributeOldValue

Set to true if `attributes` is true or omitted and *target's* [attribute value](#) before the mutation needs to be recorded.

characterDataOldValue

Set to true if `characterData` is set to true or omitted and *target's* [data](#) before the mutation needs to be recorded.

attributeFilter

Set to a list of [attribute local names](#) (without [namespace](#)) if not all [attribute mutations](#) need to be observed and `attributes` is true or omitted.

***observer* . disconnect()**

Stops *observer* from observing any mutations. Until the [observe\(\)](#) method is used again, *observer's* [callback](#) will not be invoked.

***observer* . takeRecords()**

Empties the [record queue](#) and returns what was in there.

The *MutationObserver(callback)* constructor must create a new [MutationObserver](#) object with [callback](#) set to *callback*, append it to the [unit of related similar-origin browsing contexts's](#) list of [MutationObserver](#) objects, and then return it.

The *observe(target, options)* method, when invoked, must run these steps:

1. If either *options's* `attributeOldValue` or `attributeFilter` is present and *options's* `attributes` is omitted, set *options's* `attributes` to true.
2. If *options's* `characterDataOldValue` is present and *options's* `characterData` is omitted, set *options's* `characterData` to true.
3. If none of *options's* `childList` `attributes`, and `characterData` is true, [throw](#) a `TypeError`.

4. If *options'* `attributeOldValue` is true and *options'* `attributes` is false, [throw](#) a JavaScript `TypeError`.
5. If *options'* `attributeFilter` is present and *options'* `attributes` is false, [throw](#) a JavaScript `TypeError`.
6. If *options'* `characterDataOldValue` is true and *options'* `characterData` is false, [throw](#) a JavaScript `TypeError`.
7. For each [registered observer](#) *registered* in *target's* list of [registered observers](#) whose **observer** is the [context object](#):
 1. Remove all [transient registered observers](#) whose **source** is *registered*.
 2. Replace *registered's* **options** with *options*.
8. Otherwise, add a new [registered observer](#) to *target's* list of [registered observers](#) with the [context object](#) as the **observer** and *options* as the **options**, and add *target* to [context object's](#) list of [nodes](#) on which it is registered.

The `disconnect()` method must, for each [node](#) *node* in the [context object's](#) list of [nodes](#), remove any [registered observer](#) on *node* for which the [context object](#) is the **observer**, and also empty [context object's](#) [record queue](#).

The `takeRecords()` method must return a copy of the [record queue](#) and then empty the [record queue](#).

4.3.2 Queuing a mutation record

To *queue a mutation record* of *type* for *target* with one or more of (depends on *type*) name *name*, namespace *namespace*, oldValue *oldValue*, addedNodes *addedNodes*, removedNodes *removedNodes*, previousSibling *previousSibling*, and nextSibling *nextSibling*, run these steps:

1. Let *interested observers* be an initially empty set of [MutationObserver](#) objects optionally paired with a string.
2. Let *nodes* be the [inclusive ancestors](#) of *target*.
3. Then, for each *node* in *nodes*, and then for each *registered observer* (with *registered observer's* **options** as *options*) in *node's* list of [registered observers](#):
 1. If *node* is not *target* and *options's* `subtree` is false, continue.
 2. If *type* is "attributes" and *options's* `attributes` is not true, continue.
 3. If *type* is "attributes", *options's* `attributeFilter` is present, and either *options's* `attributeFilter` does not contain *name* or *namespace* is non-null, continue.
 4. If *type* is "characterData" and *options's* `characterData` is not true, continue.
 5. If *type* is "childList" and *options's* `childList` is false, continue.
 6. If *registered observer's* **observer** is not in *interested observers*, append *registered observer's* **observer** to *interested observers*.

7. If either *type* is "attributes" and *options*'s *attributeOldValue* is true, or *type* is "characterData" and *options*'s *characterDataOldValue* is true, set the paired string of *registered observer*'s **observer** in *interested observers* to *oldValue*.
4. Then, for each *observer* in *interested observers*:
 1. Let *record* be a new *MutationRecord* object with its *type* set to *type* and *target* set to *target*.
 2. If *name* and *namespace* are given, set *record*'s *attributeName* to *name*, and *record*'s *attributeNamespace* to *namespace*.
 3. If *addedNodes* is given, set *record*'s *addedNodes* to *addedNodes*.
 4. If *removedNodes* is given, set *record*'s *removedNodes* to *removedNodes*,
 5. If *previousSibling* is given, set *record*'s *previousSibling* to *previousSibling*.
 6. If *nextSibling* is given, set *record*'s *nextSibling* to *nextSibling*.
 7. If *observer* has a paired string, set *record*'s *oldValue* to *observer*'s paired string.
 8. Append *record* to *observer*'s *record queue*.
5. [Queue a mutation observer compound microtask](#).

4.3.3 Interface *MutationRecord*

IDL

```
[Exposed=Window]
interface MutationRecord {
  readonly attribute DOMString type;
  readonly attribute Node target;
  [SameObject] readonly attribute NodeList addedNodes;
  [SameObject] readonly attribute NodeList removedNodes;
  readonly attribute Node? previousSibling;
  readonly attribute Node? nextSibling;
  readonly attribute DOMString? attributeName;
  readonly attribute DOMString? attributeNamespace;
  readonly attribute DOMString? oldValue;
};
```

This box is non-normative. Implementation requirements are given below this box.

record . *type*

Returns "attributes" if it was an [attribute](#) mutation. "characterData" if it was a mutation to a [CharacterData node](#). And "childList" if it was a mutation to the [tree of nodes](#).

record . *target*

Returns the [node](#) the mutation affected, depending on the *type*. For "attributes", it is the [element](#) whose [attribute](#) changed. For "characterData", it is the [CharacterData node](#). For "childList", it is the [node](#) whose [children](#) changed.

record . *addedNodes*

record . *removedNodes*

Return the [nodes](#) added and removed respectively.

record . previousSibling

record . nextSibling

Return the [previous](#) and [next sibling](#) respectively of the added or removed [nodes](#), and null otherwise.

record . attributeName

Returns the [local name](#) of the changed [attribute](#), and null otherwise.

record . attributeNamespace

Returns the [namespace](#) of the changed [attribute](#), and null otherwise.

record . oldValue

The return value depends on [type](#). For "attributes", it is the [value](#) of the changed [attribute](#) before the change. For "characterData", it is the [data](#) of the changed [node](#) before the change. For "childList", it is null.

The *type* and *target* attributes must return the values they were initialized to.

The *addedNodes* and *removedNodes* attributes must return the values they were initialized to. Unless stated otherwise, when a [MutationRecord](#) object is created, they must both be initialized to an empty [NodeList](#).

The *previousSibling*, *nextSibling*, *attributeName*, *attributeNamespace*, and *oldValue* attributes must return the values they were initialized to. Unless stated otherwise, when a [MutationRecord](#) object is created, they must be initialized to null.

4.3.4 Garbage collection

[Nodes](#) have a strong reference to [registered observers](#) in their list of [registered observers](#).

[Registered observers](#) in a [node](#)'s list of [registered observers](#) have a weak reference to the [node](#).

4.4 Interface [Node](#)

IDL

```
[Exposed=Window]
interface Node : EventTarget {
  const unsigned short ELEMENT_NODE = 1;
  const unsigned short ATTRIBUTE_NODE = 2; // historical
  const unsigned short TEXT_NODE = 3;
  const unsigned short CDATA_SECTION_NODE = 4; // historical
  const unsigned short ENTITY_REFERENCE_NODE = 5; // historical
  const unsigned short ENTITY_NODE = 6; // historical
  const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short COMMENT_NODE = 8;
  const unsigned short DOCUMENT_NODE = 9;
  const unsigned short DOCUMENT_TYPE_NODE = 10;
  const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
  const unsigned short NOTATION_NODE = 12; // historical
  readonly attribute unsigned short nodeType;
  readonly attribute DOMString nodeName;

  readonly attribute DOMString? baseURI;
```

```

readonly attribute Document? ownerDocument;
readonly attribute Node? parentNode;
readonly attribute Element? parentElement;
boolean hasChildNodes();
[SameObject] readonly attribute NodeList childNodes;
readonly attribute Node? firstChild;
readonly attribute Node? lastChild;
readonly attribute Node? previousSibling;
readonly attribute Node? nextSibling;

        attribute DOMString? nodeValue;
        attribute DOMString? textContent;
void normalize();

[NewObject] Node cloneNode(optional boolean deep = false);
boolean isEqualNode(Node? node);

const unsigned short DOCUMENT_POSITION_DISCONNECTED = 0x01;
const unsigned short DOCUMENT_POSITION_PRECEDING = 0x02;
const unsigned short DOCUMENT_POSITION_FOLLOWING = 0x04;
const unsigned short DOCUMENT_POSITION_CONTAINS = 0x08;
const unsigned short DOCUMENT_POSITION_CONTAINED_BY = 0x10;
const unsigned short DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 0x20;
unsigned short compareDocumentPosition(Node other);
boolean contains(Node? other);

DOMString? lookupPrefix(DOMString? namespace);
DOMString? lookupNamespaceURI(DOMString? prefix);
boolean isDefaultNamespace(DOMString? namespace);

Node insertBefore(Node node, Node? child);
Node appendChild(Node node);
Node replaceChild(Node node, Node child);
Node removeChild(Node child);
};

```

Note: *Node is an abstract interface and does not exist as [node](#). It is used by all [nodes](#) ([Document](#), [DocumentFragment](#), [DocumentType](#), [Element](#), [Text](#), [ProcessingInstruction](#), and [Comment](#)).*

Each [node](#) has an associated *node document*, set upon creation, that is a [document](#).

Note: *A [node's node document](#) can be changed by the [adopt](#) algorithm.*

Each [node](#) also has an associated *base URL*.

Note: *Other specifications define the value of the [base URL](#) and its observable behavior. This specification solely defines the concept and the [baseURI](#) attribute.*

This box is non-normative. Implementation requirements are given below this box.

node . nodeType

Returns the type of *node*, represented by a number from the following list:

Node . ELEMENT_NODE (1)

node is an [element](#).

Node . TEXT_NODE (3)

node is a [Text node](#).

Node . PROCESSING_INSTRUCTION_NODE (7)

node is a [ProcessingInstruction node](#).

Node . COMMENT_NODE (8)

node is a [Comment node](#).

Node . DOCUMENT_NODE (9)

node is a [document](#).

Node . DOCUMENT_TYPE_NODE (10)

node is a [doctype](#).

Node . DOCUMENT_FRAGMENT_NODE (11)

node is a [DocumentFragment node](#).

node . nodeName

Returns a string appropriate for the type of *node*, as follows:

Element

Its [tagName](#) attribute value.

Text

"#text".

ProcessingInstruction

Its [target](#).

Comment

"#comment".

Document

"#document".

DocumentType

Its [name](#).

DocumentFragment

"#document - fragment".

The *nodeType* attribute must return the type of the node, which must be one of the following:

- *ELEMENT_NODE* (1);

- `TEXT_NODE` (3);
- `PROCESSING_INSTRUCTION_NODE` (7);
- `COMMENT_NODE` (8);
- `DOCUMENT_NODE` (9);
- `DOCUMENT_TYPE_NODE` (10);
- `DOCUMENT_FRAGMENT_NODE` (11).

The `nodeName` attribute must return the following, depending on the [context object](#):

↪ **Element**
Its `tagName` attribute value.

↪ **Text**
"#text".

↪ **ProcessingInstruction**
Its `target`.

↪ **Comment**
"#comment".

↪ **Document**
"#document".

↪ **DocumentType**
Its `name`.

↪ **DocumentFragment**
"#document-fragment".

This box is non-normative. Implementation requirements are given below this box.

`node . baseURI`
Returns the [base URL](#).

The `baseURI` attribute must return the associated [base URL](#).

This box is non-normative. Implementation requirements are given below this box.

`node . ownerDocument`
Returns the [node document](#).
Returns null for [documents](#).

`node . parentNode`
Returns the [parent](#).

node* . *parentElement

Returns the [parent element](#).

node* . *hasChildNodes()

Returns whether *node* has [children](#).

node* . *childNodes

Returns the [children](#).

node* . *firstChild

Returns the [first child](#).

node* . *lastChild

Returns the [last child](#).

node* . *previousSibling

Returns the [previous sibling](#).

node* . *nextSibling

Returns the [next sibling](#).

The *ownerDocument* attribute must run these steps:

1. If the [context object](#) is a [document](#), return null.
2. Return the [node document](#).

The [node document](#) of a [document](#) is that [document](#) itself.

All [nodes](#) have a [document](#) at all times.

The *parentNode* attribute must return the [parent](#).

The *parentElement* attribute must return the [parent element](#).

The *hasChildNodes()* method must return true if the [context object](#) has [children](#), and false otherwise.

The *childNodes* attribute must return a [NodeList](#) rooted at the [context object](#) matching only [children](#).

The *firstChild* attribute must return the [first child](#).

The *lastChild* attribute must return the [last child](#).

The *previousSibling* attribute must return the [previous sibling](#).

The *nextSibling* attribute must return the [next sibling](#).

The *nodeValue* attribute must return the following, depending on the [context object](#):

- ↪ Text
- ↪ Comment
- ↪ ProcessingInstruction
The [context object](#)'s [data](#).
- ↪ **Any other node**
Null.

The *nodeValue* attribute must, on setting, if the new value is null, act as if it was the empty string instead, and then do as described below, depending on the [context object](#):

- ↪ Text
- ↪ Comment
- ↪ ProcessingInstruction
[Replace data](#) with node [context object](#), offset 0, count [length](#) attribute value, and data new value.
- ↪ **Any other node**
Do nothing.

The *textContent* attribute must return the following, depending on the [context object](#):

- ↪ DocumentFragment
- ↪ Element
The concatenation of [data](#) of all the Text [node descendants](#) of the [context object](#), in [tree order](#).
- ↪ Text
- ↪ ProcessingInstruction
- ↪ Comment
The [context object](#)'s [data](#).
- ↪ **Any other node**
Null.

The *textContent* attribute must, on setting, if the new value is null, act as if it was the empty string instead, and then do as described below, depending on the [context object](#):

- ↪ DocumentFragment
- ↪ Element
 1. Let *node* be null.
 2. If new value is not the empty string, set *node* to a new Text [node](#) whose [data](#) is new value.
 3. [Replace all](#) with *node* within the [context object](#).
- ↪ Text
- ↪ ProcessingInstruction
- ↪ Comment
[Replace data](#) with node [context object](#), offset 0, count [length](#) attribute value, and data new value.

↪ **Any other node**
Do nothing.

This box is non-normative. Implementation requirements are given below this box.
`node . normalize()`

Removes empty Text nodes and concatenates the data of remaining contiguous Text nodes into the first of their nodes.

The `normalize()` method must run these steps:

For each Text node descendant of the context object:

1. Let *node* be the Text node descendant.
 2. Let *length* be *node*'s length attribute value.
 3. If *length* is zero, remove *node* and continue with the next Text node, if any.
 4. Let *data* be the concatenation of the data of *node*'s contiguous Text nodes (excluding itself), in tree order.
 5. Replace data with *node*, offset *length*, count 0, and data *data*.
 6. Let *current node* be *node*'s next sibling.
 7. While *current node* is a Text node:
 1. For each range whose start node is *current node*, add *length* to its start offset and set its start node to *node*.
 2. For each range whose end node is *current node*, add *length* to its end offset and set its end node to *node*.
 3. For each range whose start node is *current node*'s parent and start offset is *current node*'s index, set its start node to *node* and its start offset to *length*.
 4. For each range whose end node is *current node*'s parent and end offset is *current node*'s index, set its end node to *node* and its end offset to *length*.
 5. Add *current node*'s length attribute value to *length*.
 6. Set *current node* to its next sibling.
 8. Remove *node*'s contiguous Text nodes (excluding itself), in tree order.
-

This box is non-normative. Implementation requirements are given below this box.
`node . cloneNode([deep = false])`

Returns a copy of *node*. If *deep* is true, the copy also includes the *node*'s descendants.

`node . isEqualNode(other)`

Returns whether *node* and *other* have the same properties.

[Specifications](#) may define *cloning steps* for all or some [nodes](#). The algorithm is passed *copy*, *node*, *document*, and optionally a *clone children flag*, as indicated in the [clone](#) algorithm.

Note: HTML defines *cloning steps* for *script* and *input* elements. SVG ought to do the same for its *script* elements, but does not call this out at the moment.

To clone a *node*, optionally with a *document* and a *clone children flag*, run these steps:

1. If *document* is not given, let *document* be *node*'s [node document](#).
2. Let *copy* be a [node](#) that implements the same interfaces as *node*.
3. If *copy* is a [document](#), set its [node document](#) and *document* to *copy*.

Otherwise, set *copy*'s [node document](#) to *document*.

4. Copy the following from *node* to *copy*, depending on the type of *node*:

↪ **Document**

Its [encoding](#), [content type](#), [URL](#), its mode ([quirks mode](#), [limited quirks mode](#), or [no-quirks mode](#)), and its type ([XML document](#) or [HTML document](#)).

↪ **DocumentType**

Its [name](#), [public ID](#), and [system ID](#).

↪ **Element**

Its [namespace](#), [namespace prefix](#), [local name](#), and its [attribute list](#).

↪ **Text**

↪ **Comment**

Its [data](#).

↪ **ProcessingInstruction**

Its [target](#) and [data](#).

↪ **Any other node**

—

5. Run any [cloning steps](#) defined for *node* in [other applicable specifications](#) and pass *copy*, *node*, *document* and the *clone children flag* if set, as parameters.
6. If the *clone children flag* is set, [clone](#) all the [children](#) of *node* and append them to *copy*, with *document* as specified and the *clone children flag* being set.
7. Return *copy*.

The `cloneNode(deep)` method must return a [clone](#) of the [context object](#), with the *clone children flag* set if *deep* is true.

A [node](#) *A* equals a [node](#) *B* if all of the following conditions are true:

- *A* and *B*'s [nodeType](#) attribute value is identical.
- The following are also equal, depending on *A*:
 - ↪ **[DocumentType](#)**
Its [name](#), [public ID](#), and [system ID](#).
 - ↪ **[Element](#)**
Its [namespace](#), [namespace prefix](#), [local name](#), and its number of [attributes](#) in its [attribute list](#).
 - ↪ **[ProcessingInstruction](#)**
Its [target](#) and [data](#).
 - ↪ **[Text](#)**
 - ↪ **[Comment](#)**
Its [data](#).
 - ↪ **Any other node**
—
- If *A* is an [element](#), each [attribute](#) in its [attribute list](#) has an [attribute](#) with the same [namespace](#), [local name](#), and [value](#) in *B*'s [attribute list](#).
- *A* and *B* have the same number of [children](#).
- Each [child](#) of *A* [equals](#) the [child](#) of *B* at the identical [index](#).

The `isEqualNode(node)` method must return true if *node* is not null and [context object equals node](#), and false otherwise.

This box is non-normative. Implementation requirements are given below this box.

`node . compareDocumentPosition(other)`

Returns a bitmask indicating the position of *other* relative to *node*. These are the bits that can be set:

Node . DOCUMENT_POSITION_DISCONNECTED (1)

Set when *node* and *other* are not in the same [tree](#).

Node . DOCUMENT_POSITION_PRECEDING (2)

Set when *other* is [preceding node](#).

Node . DOCUMENT_POSITION_FOLLOWING (4)

Set when *other* is [following node](#).

Node . DOCUMENT_POSITION_CONTAINS (8)

Set when *other* is an [ancestor](#) of *node*.

Node . DOCUMENT_POSITION_CONTAINED_BY (16, 10 in hexadecimal)

Set when *other* is a [descendant](#) of *node*.

***node* . contains(*other*)**

Returns true if *other* is an [inclusive descendant](#) of *node*, and false otherwise.

These are the constants `compareDocumentPosition()` returns as mask:

- `DOCUMENT_POSITION_DISCONNECTED` (1);
- `DOCUMENT_POSITION_PRECEDING` (2);
- `DOCUMENT_POSITION_FOLLOWING` (4);
- `DOCUMENT_POSITION_CONTAINS` (8);
- `DOCUMENT_POSITION_CONTAINED_BY` (16, 10 in hexadecimal);
- `DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC` (32, 20 in hexadecimal).

The `compareDocumentPosition(other)` method must run these steps:

1. Let *reference* be the [context object](#).
2. If *other* and *reference* are the same object, return zero.
3. If *other* and *reference* are not in the same [tree](#), return the result of adding `DOCUMENT_POSITION_DISCONNECTED`, `DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC`, and either `DOCUMENT_POSITION_PRECEDING` or `DOCUMENT_POSITION_FOLLOWING`, with the constraint that this is to be consistent, together.

Note: Whether to return `DOCUMENT_POSITION_PRECEDING` or `DOCUMENT_POSITION_FOLLOWING` is typically implemented via pointer comparison. In JavaScript implementations `Math.random()` can be used.

4. If *other* is an [ancestor](#) of *reference*, return the result of adding `DOCUMENT_POSITION_CONTAINS` to `DOCUMENT_POSITION_PRECEDING`.
5. If *other* is a [descendant](#) of *reference*, return the result of adding `DOCUMENT_POSITION_CONTAINED_BY` to `DOCUMENT_POSITION_FOLLOWING`.
6. If *other* is [preceding](#) *reference* return `DOCUMENT_POSITION_PRECEDING`.
7. Return `DOCUMENT_POSITION_FOLLOWING`.

The `contains(other)` method must return true if *other* is an [inclusive descendant](#) of the [context object](#), and false otherwise (including when *other* is null).

To locate a namespace prefix for an *element* using *namespace* run these steps:

1. If *element*'s [namespace](#) is *namespace* and its [namespace prefix](#) is not null, return its [namespace prefix](#).
2. If, *element* has an [attribute](#) whose [namespace prefix](#) is "xmlns" and [value](#) is *namespace*, then return *element*'s first such [attribute](#)'s [local name](#).
3. If *element*'s [parent element](#) is not null, return the result of running [locate a namespace prefix](#) on that [element](#) using *namespace*. Otherwise, return null.

To locate a namespace for a *node* using *prefix* depends on *node*:

↪ Element

1. If its [namespace](#) is not null and its [namespace prefix](#) is *prefix*, return [namespace](#).
2. If it [has](#) an [attribute](#) whose [namespace](#) is the [XMLNS namespace](#), [namespace prefix](#) is "xmlns" and [local name](#) is *prefix*, or if *prefix* is null and it [has](#) an [attribute](#) whose [namespace](#) is the [XMLNS namespace](#), [namespace prefix](#) is null and [local name](#) is "xmlns":
 1. Let *value* be its [value](#) if it is not the empty string, and null otherwise.
 2. Return *value*.
3. If its [parent element](#) is null, return null.
4. Return the result of running [locate a namespace](#) on its [parent element](#) using *prefix*.

↪ Document

1. If its [document element](#) is null, return null.
2. Return the result of running [locate a namespace](#) on its [document element](#) using *prefix*.

↪ DocumentType

↪ DocumentFragment

Return null.

↪ Any other node

1. If its [parent element](#) is null, return null.
2. Return the result of running [locate a namespace](#) on its [parent element](#) using *prefix*.

The *lookupPrefix(namespace)* method must run these steps:

1. If *namespace* is null or the empty string, return null.
2. Otherwise it depends on the [context object](#):

↪ Element

Return the result of [locating a namespace prefix](#) for the node using *namespace*.

↪ Document

Return the result of [locating a namespace prefix](#) for its [document element](#), if that is not null, and null otherwise.

↪ DocumentType

↪ DocumentFragment

Return null.

↪ Any other node

Return the result of [locating a namespace prefix](#) for its [parent element](#), or if that is null, null.

The `lookupNamespaceURI(prefix)` method must run these steps:

1. If *prefix* is the empty string, set it to null.
2. Return the result of running [locate a namespace](#) for the [context object](#) using *prefix*.

The `isDefaultNamespace(namespace)` method must run these steps:

1. If *namespace* is the empty string, set it to null.
 2. Let *defaultNamespace* be the result of running [locate a namespace](#) for the [context object](#) using null.
 3. Return true if *defaultNamespace* is the same as *namespace*, and false otherwise.
-

The `insertBefore(node, child)` method must return the result of [pre-inserting](#) *node* into the [context object](#) before *child*.

The `appendChild(node)` method must return the result of [appending](#) *node* to the [context object](#).

The `replaceChild(node, child)` method must return the result of [replacing](#) *child* with *node* within the [context object](#).

The `removeChild(child)` method must return the result of [pre-removing](#) *child* from the [context object](#).

The *list of elements with local name localName* for a [node](#) *root* is the [HTMLCollection](#) returned by the following algorithm:

1. If *localName* is "*" (U+002A), return a [HTMLCollection](#) rooted at *root*, whose filter matches only [elements](#).
2. Otherwise, if *root*'s [node document](#) is an [HTML document](#), return a [HTMLCollection](#) rooted at *root*, whose filter matches the following [descendant elements](#):
 - Whose [namespace](#) is the [HTML namespace](#) and whose [local name](#) is *localName* [converted to ASCII lowercase](#).
 - Whose [namespace](#) is *not* the [HTML namespace](#) and whose [local name](#) is *localName*.
3. Otherwise, return a [HTMLCollection](#) rooted at *root*, whose filter matches [descendant elements](#) whose [local name](#) is *localName*.

When invoked with the same argument, the same [HTMLCollection](#) object may be returned as returned by an earlier call.

The *list of elements with namespace namespace and local name localName* for a [node](#) *root* is the [HTMLCollection](#) returned by the following algorithm:

1. If *namespace* is the empty string, set it to null.
2. If both *namespace* and *localName* are "*" (U+002A), return a [HTMLCollection](#) rooted at *root*, whose filter matches [descendant elements](#).
3. Otherwise, if *namespace* is "*" (U+002A), return a [HTMLCollection](#) rooted at *root*, whose filter matches [descendant elements](#) whose [local name](#) is *localName*.
4. Otherwise, if *localName* is "*" (U+002A), return a [HTMLCollection](#) rooted at *root*, whose filter matches [descendant elements](#) whose [namespace](#) is *namespace*.
5. Otherwise, return a [HTMLCollection](#) rooted at *root*, whose filter matches [descendant elements](#) whose [namespace](#) is *namespace* and [local name](#) is *localName*.

When invoked with the same arguments, the same [HTMLCollection](#) object may be returned as returned by an earlier call.

The *list of elements with class names classNames* for a [node root](#) is the [HTMLCollection](#) returned by the following algorithm:

1. Let *classes* be the result of running the [ordered set parser](#) on *classNames*.
2. If *classes* is the empty set, return an empty [HTMLCollection](#).
3. Return a [HTMLCollection](#) rooted at *root*, whose filter matches [descendant elements](#) that have all their [classes](#) in *classes*.

The comparisons for the [classes](#) must be done in an [ASCII case-insensitive](#) manner if *root's node document* is in [quirks mode](#), and in a [case-sensitive](#) manner otherwise.

When invoked with the same argument, the same [HTMLCollection](#) object may be returned as returned by an earlier call.

4.5 Interface [Document](#)

IDL `[Constructor,
Exposed=Window]
interface Document : Node {
 [SameObject] readonly attribute DOMImplementation implementation;
 readonly attribute DOMString URL;
 readonly attribute DOMString documentURI;
 readonly attribute DOMString origin;
 readonly attribute DOMString compatMode;
 readonly attribute DOMString characterSet;
 readonly attribute DOMString contentType;

 readonly attribute DocumentType? doctype;
 readonly attribute Element? documentElement;
 HTMLCollection getElementsByTagName(DOMString localName);
 HTMLCollection getElementsByTagNameNS(DOMString? namespace, DOMString
localName);
 HTMLCollection getElementsByClassName(DOMString classNames);

 [NewObject] Element createElement(DOMString localName);
 [NewObject] Element createElementNS(DOMString? namespace, DOMString
qualifiedName);
 [NewObject] DocumentFragment createDocumentFragment();
 [NewObject] Text createTextNode(DOMString data);`

```

[NewObject] Comment createComment(DOMString data);
[NewObject] ProcessingInstruction createProcessingInstruction(DOMString
target, DOMString data);

[NewObject] Node importNode(Node node, optional boolean deep = false);
Node adoptNode(Node node);

[NewObject] Event createEvent(DOMString interface);

[NewObject] Range createRange();

// NodeFilter.SHOW_ALL = 0xFFFFFFFF
[NewObject] NodeIterator createNodeIterator(Node root, optional unsigned
long whatToShow = 0xFFFFFFFF, optional NodeFilter? filter = null);
[NewObject] TreeWalker createTreeWalker(Node root, optional unsigned long
whatToShow = 0xFFFFFFFF, optional NodeFilter? filter = null);
};

[Exposed=Window]
interface XMLDocument : Document {};

```

Document [nodes](#) are simply known as *documents*.

Each [document](#) has an associated *encoding*, *content type*, and *URL*. [\[ENCODING\]](#) [\[URL\]](#)

Unless stated otherwise, a [document](#)'s [encoding](#) is the `utf-8` [encoding](#), its [content type](#) is "application/xml", and its [URL](#) is "about:blank".

Unless stated otherwise, a [document](#)'s [origin](#) is a globally unique identifier and its [effective script origin](#) is an [alias](#) of that [origin](#). [\[HTML\]](#)

A [document](#) is assumed to be an *XML document* unless it is flagged as being an *HTML document*. Whether a [document](#) is an [HTML document](#) or an [XML document](#) affects the behavior of certain APIs.

A [document](#) is always set to one of three modes: *no-quirks mode*, the default; *quirks mode*, used typically for legacy documents; and *limited-quirks mode*. Unless stated otherwise, a [document](#) must be in [no-quirks mode](#).

Note: The mode is only ever changed from the default if the [document](#) is created by the [HTML parser](#), based on the presence, absence, or value of the *DOCTYPE* string. [\[HTML\]](#)

Note: [No-quirks mode](#) was originally known as "standards mode" and [limited-quirks mode](#) was once known as "almost standards mode". They have been renamed because their details are now defined by standards. (And because Ian Hickson vetoed their original names on the basis that they are nonsensical.)

This box is non-normative. Implementation requirements are given below this box.

`document = new Document()`

Returns a new [document](#).

`document . implementation`

Returns *document's* [DOMImplementation](#) object.

***document* . URL**

***document* . documentURI**

Returns *document's* [URL](#).

***document* . origin**

Returns *document's* [origin](#).

***document* . compatMode**

Returns the string "CSS1Compat" if *document* is in [no-quirks mode](#) or [limited-quirks mode](#), and "BackCompat", if *document* is in [quirks mode](#).

***document* . characterSet**

Returns *document's* [encoding](#).

***document* . contentType**

Returns *document's* [content type](#).

The *Document()* constructor must return a new [document](#) whose [origin](#) is an [alias](#) to the [origin](#) of the global object's associated [document](#), and [effective script origin](#) is an [alias](#) to the [effective script origin](#) of the global object's associated [document](#). [HTML]

Note: Unlike [createDocument\(\)](#) this constructor does not return an [XMLDocument object](#), but a [document](#) ([Document object](#)).

The *implementation* attribute must return the [DOMImplementation](#) object that is associated with the [document](#).

The *URL* and *documentURI* attributes must return the [URL](#).

The *origin* attribute must return the [Unicode serialization](#) of [context object's](#) [origin](#).

The *compatMode* attribute must return "BackCompat" if the [context object](#) is in [quirks mode](#), and "CSS1Compat" otherwise.

The *characterSet* attribute's getter and *inputEncoding* attribute's getter must run these steps:

1. Let *name* be [encoding's](#) [name](#).
2. If *name* is in the first column in the table below, set *name* to the value of the second column on the same row:

Name	Compatibility name
utf-8	"UTF-8"
ibm866	"IBM866"
iso-8859-2	"ISO-8859-2"
iso-8859-3	"ISO-8859-3"

iso-8859-4	"ISO-8859-4"
iso-8859-5	"ISO-8859-5"
iso-8859-6	"ISO-8859-6"
iso-8859-7	"ISO-8859-7"
iso-8859-8	"ISO-8859-8"
iso-8859-8-i	"ISO-8859-8-I"
iso-8859-10	"ISO-8859-10"
iso-8859-13	"ISO-8859-13"
iso-8859-14	"ISO-8859-14"
iso-8859-15	"ISO-8859-15"
iso-8859-16	"ISO-8859-16"
koi8-r	"KOI8-R"
koi8-u	"KOI8-U"
gbk	"GBK"
big5	"Big5"
euc-jp	"EUC-JP"
iso-2022-jp	"ISO-2022-JP"
shift_jis	"Shift_JIS"
euc-kr	"EUC-KR"
utf-16be	"UTF-16BE"
utf-16le	"UTF-16LE"

3. Return *name*.

The *contentType* attribute must return the [content type](#).

This box is non-normative. Implementation requirements are given below this box.
document.doctype

Returns the [doctype](#) or null if there is none.

document.documentElement

Returns the [document element](#).

collection = document.getElementsByTagName(localName)

If *localName* is "*" returns a [HTMLCollection](#) of all [descendant elements](#).

Otherwise, returns a [HTMLCollection](#) of all [descendant elements](#) whose [local name](#) is *localName*. (Matches case-insensitively against [elements](#) in the [HTML namespace](#) within an [HTML document](#).)

collection = document.getElementsByTagNameNS(namespace, localName)

If *namespace* and *localName* are "*" returns a [HTMLCollection](#) of all [descendant elements](#).

If only *namespace* is "*" returns a [HTMLCollection](#) of all [descendant elements](#) whose [local name](#) is *localName*.

If only *localName* is "*" returns a [HTMLCollection](#) of all [descendant elements](#) whose [namespace](#) is *namespace*.

Otherwise, returns a [HTMLCollection](#) of all [descendant elements](#) whose [namespace](#) is *namespace* and [local name](#) is *localName*.

***collection* = *document* . *getElementsByClassName*(*classes*)**

***collection* = *element* . *getElementsByClassName*(*classes*)**

Returns a [HTMLCollection](#) of the [elements](#) in the object on which the method was invoked (a [document](#) or an [element](#)) that have all the classes given by *classes*.

The *classes* argument is interpreted as a space-separated list of classes.

The *doctype* attribute must return the [child](#) of the [document](#) that is a [doctype](#), and null otherwise.

The *documentElement* attribute must return the [document element](#).

The *getElementsByTagName*(*localName*) method must return the [list of elements with local name *localName*](#) for the [context object](#).

Note: Thus, in an HTML document, document.getElementsByTagName("F00") will match F00 elements that are not in the HTML namespace, and foo elements that are in the HTML namespace, but not F00 elements that are in the HTML namespace.

The *getElementsByTagNameNS*(*namespace*, *localName*) method must return the [list of elements with namespace *namespace* and local name *localName*](#) for the [context object](#).

The *getElementsByClassName*(*classNames*) method must return the [list of elements with class names *classNames*](#) for the [context object](#).

Given the following XHTML fragment:

```
<div id="example">
  <p id="p1" class="aaa bbb"/>
  <p id="p2" class="aaa ccc"/>
  <p id="p3" class="bbb ccc"/>
</div>
```

A call to `document.getElementById('example').getElementsByClassName('aaa')` would return a [HTMLCollection](#) with the two paragraphs p1 and p2 in it.

A call to `getElementsByClassName('ccc bbb')` would only return one node, however, namely p3. A call to `document.getElementById('example').getElementsByClassName('bbb ccc')` would return the same thing.

A call to `getElementsByClassName('aaa,bbb')` would return no nodes; none of the elements above are in the `aaa,bbb` class.

This box is non-normative. Implementation requirements are given below this box.
element = document . createElement(*localName*)

Returns an [element](#) in the [HTML namespace](#) [see [bug 19431](#)] with *localName* as [local name](#). (In an [HTML document](#) *localName* is lowercased.)

If *localName* does not match the [Name](#) production an "[InvalidCharacterError](#)" exception will be thrown.

***element* = *document* . createElementNS(*namespace*, *qualifiedName*)**

Returns an [element](#) with [namespace](#) *namespace*. Its [namespace prefix](#) will be everything before ":" (U+003E) in *qualifiedName* or null. Its [local name](#) will be everything after ":" (U+003E) in *qualifiedName* or *qualifiedName*.

If *localName* does not match the [Name](#) production an "[InvalidCharacterError](#)" exception will be thrown.

If one of the following conditions is true a "[NamespaceError](#)" exception will be thrown:

- *localName* does not match the [QName](#) production.
- [Namespace prefix](#) is not null and *namespace* is the empty string.
- [Namespace prefix](#) is "xml" and *namespace* is not the [XML namespace](#).
- *qualifiedName* or [namespace prefix](#) is "xmlns" and *namespace* is not the [XMLNS namespace](#).
- *namespace* is the [XMLNS namespace](#) and neither *qualifiedName* nor [namespace prefix](#) is "xmlns".

***documentFragment* = *document* . createDocumentFragment()**

Returns a [DocumentFragment](#) [node](#).

***text* = *document* . createTextNode(*data*)**

Returns a [Text](#) [node](#) whose [data](#) is *data*.

***comment* = *document* . createComment(*data*)**

Returns a [Comment](#) [node](#) whose [data](#) is *data*.

***processingInstruction* = *document* . createProcessingInstruction(*target*, *data*)**

Returns a [ProcessingInstruction](#) [node](#) whose [target](#) is *target* and [data](#) is *data*.

If *target* does not match the [Name](#) production an "[InvalidCharacterError](#)" exception will be thrown.

If *data* contains ">" an "[InvalidCharacterError](#)" exception will be thrown.

The *element* interface for any *name* and *namespace* is [Element](#), unless stated otherwise.

Note: The HTML Standard will e.g. define that for *html* and the [HTML namespace](#), the [HTMLHtmlElement](#) interface is used. [\[HTML\]](#)

The *createElement(localName)* method must run the these steps:

1. If *localName* does not match the Name production, [throw](#) an "InvalidCharacterError" exception.
2. If the [context object](#) is an [HTML document](#), let *localName* be [converted to ASCII lowercase](#).
3. Let *interface* be the [element interface](#) for *localName* and the [HTML namespace](#).
4. Return a new [element](#) that implements *interface*, with no attributes, [namespace](#) set to the [HTML namespace](#) [see [bug 19431](#)], [local name](#) set to *localName*, and [node document](#) set to the [context object](#).

The `createElementNS(namespace, qualifiedName)` method must run these steps:

1. If *namespace* is the empty string, set it to null.
2. If *qualifiedName* does not match the Name production, [throw](#) an "InvalidCharacterError" exception.
3. If *qualifiedName* does not match the QName production, [throw](#) a "NamespaceError" exception.
4. If *qualifiedName* contains a ":" (U+003E), then split the string on it and let *prefix* be the part before and *localName* the part after. Otherwise, let *prefix* be null and *localName* be *qualifiedName*.
5. If *prefix* is not null and *namespace* is null, [throw](#) a "NamespaceError" exception.
6. If *prefix* is "xml" and *namespace* is not the [XML namespace](#), [throw](#) a "NamespaceError" exception.
7. If *qualifiedName* or *prefix* is "xmlns" and *namespace* is not the [XMLNS namespace](#), [throw](#) a "NamespaceError" exception.
8. If *namespace* is the [XMLNS namespace](#) and neither *qualifiedName* nor *prefix* is "xmlns", [throw](#) a "NamespaceError" exception.
9. Let *interface* be the [element interface](#) for *localName* and *namespace*.
10. Return a new [element](#) that implements *interface*, with no attributes, [namespace](#) set to *namespace*, [namespace prefix](#) set to *prefix*, [local name](#) set to *localName*, and [node document](#) set to the [context object](#).

The `createDocumentFragment()` method must return a new DocumentFragment [node](#) with its [node document](#) set to the [context object](#).

The `createTextNode(data)` method must return a new Text [node](#) with its [data](#) set to *data* and [node document](#) set to the [context object](#).

Note: No check is performed that data consists of characters that match the Char production.

The `createComment(data)` method must return a new Comment [node](#) with its [data](#) set to *data* and [node document](#) set to the [context object](#).

Note: No check is performed that data consists of characters that match the Char production or that it contains two adjacent hyphens or ends with a hyphen.

The `createProcessingInstruction(target, data)` method must run these steps:

1. If *target* does not match the Name production, throw an `"InvalidCharacterError"` exception.
2. If *data* contains the string ">", throw an `"InvalidCharacterError"` exception.
3. Return a new ProcessingInstruction node, with target set to *target*, data set to *data*, and node document set to the context object.

Note: No check is performed that target contains "xml" or ":", or that data contains characters that match the Char production.

This box is non-normative. Implementation requirements are given below this box.

`clone = document . importNode(node [, deep = false])`

Returns a copy of *node*. If *deep* is true, the copy also includes the *node*'s descendants.

If *node* is a document throws a `"NotSupportedError"` exception.

`node = document . adoptNode(node)`

Moves *node* from another document and returns it.

If *node* is a document throws a `"NotSupportedError"` exception.

The `importNode(node, deep)` method must run these steps:

1. If *node* is a document, throw a `"NotSupportedError"` exception.
2. Return a clone of *node*, with context object and the *clone children flag* set if *deep* is true.

Specifications may define *adopting steps* for all or some nodes. The algorithm is passed *node* and *oldDocument*, as indicated in the adopt algorithm.

To *adopt* a *node* into a *document*, run these steps:

1. Let *oldDocument* be *node*'s node document.
2. If *node*'s parent is not null, remove *node* from its parent.
3. Set *node*'s inclusive descendants's node document to *document*.
4. Run any adopting steps defined for *node* in other applicable specifications and pass *node* and *oldDocument* as parameters.

The `adoptNode(node)` method must run these steps:

1. If *node* is a [document](#), [throw](#) a "`NotSupportedError`" exception.
 2. [Adopt](#) *node* into the [context object](#).
 3. Return *node*.
-

The `createEvent(interface)` method must run these steps:

1. Let *constructor* be null.
2. If *interface* is an [ASCII case-insensitive](#) match for any of the strings in the first column in the following table, set *constructor* to the interface in the second column on the same row as the matching string:

String	Interface	Notes
"customevent"	CustomEvent	
"event"	Event	
"events"	Event	
"htmlevents"	Event	
"keyboardevent"	KeyboardEvent	[UIEVENTS]
"keyevents"	KeyboardEvent	[UIEVENTS]
"messageevent"	MessageEvent	[HTML]
"mouseevent"	MouseEvent	[UIEVENTS]
"mouseevents"	MouseEvent	[UIEVENTS]
"touchevent"	TouchEvent	[TOUCHEVENTS]
"uievent"	UIEvent	[UIEVENTS]
"uievents"	UIEvent	[UIEVENTS]

3. If *constructor* is null, [throw](#) a "`NotSupportedError`".
4. Let *event* be the result of [invoking](#) the initial value of *constructor* with the empty string as argument.
5. Unset *event*'s [initialized flag](#).
6. Return *event*.

Note: [Event constructors](#) can be used instead.

The `createRange()` method must return a new [range](#) with ([context object](#), 0) as its [start](#) and [end](#).

Note: The [Range\(\)](#) constructor can be used instead.

The `createNodeIterator(root, whatToShow, filter)` method must run these steps:

1. Create a NodeIterator object.
2. Set root and initialize the referenceNode attribute to the *root* argument.
3. Initialize the pointerBeforeReferenceNode attribute to true.
4. Set whatToShow to the *whatToShow* argument.
5. Set filter to *filter*.
6. Return the newly created NodeIterator object.

The *createTreeWalker(root, whatToShow, filter)* method must run these steps:

1. Create a TreeWalker object.
2. Set root and initialize the currentNode attribute to the *root* argument.
3. Set whatToShow to the *whatToShow* argument.
4. Set filter to *filter*.
5. Return the newly created TreeWalker object.

4.5.1 Interface DOMImplementation

User agents must create a DOMImplementation object whenever a document is created and associate it with that document.

IDL

```
[Exposed=Window]
interface DOMImplementation {
  [NewObject] DocumentType createDocumentType(DOMString qualifiedName,
DOMString publicId, DOMString systemId);
  [NewObject] XMLDocument createDocument(DOMString? namespace,
[TreatNullAs=EmptyString] DOMString qualifiedName, optional DocumentType?
doctype = null);
  [NewObject] Document createHTMLDocument(optional DOMString title);

  boolean hasFeature(); // useless; always returns true
};
```

This box is non-normative. Implementation requirements are given below this box.
`doctype = document . implementation . createDocumentType(qualifiedName,
publicId, systemId)`

Returns a doctype, with the given *qualifiedName*, *publicId*, and *systemId*. If *qualifiedName* does not match the Name production, an "InvalidCharacterError" exception is thrown, and if it does not match the QName production, a "NamespaceError" exception is thrown.

`doc = document . implementation . createDocument(namespace, qualifiedName [,
doctype = null])`

Returns an XMLDocument [see [bug 22960](#)], with a document element whose local name is *qualifiedName* and whose namespace is *namespace* (unless *qualifiedName* is the empty string), and with *doctype*, if it is given, as its doctype.

This method throws the same exceptions as the [createElementNS](#) method, when invoked with the same arguments.

```
doc = document . implementation . createHTMLDocument([title])
```

Returns a [document](#), with a basic [tree](#) already constructed including a [title](#) element, unless the *title* argument is omitted.

The *createDocumentType*(*qualifiedName*, *publicId*, *systemId*) method must run these steps:

1. If *qualifiedName* does not match the [Name](#) production, [throw](#) an ["InvalidCharacterError"](#) exception.
2. If *qualifiedName* does not match the [QName](#) production, [throw](#) a ["NamespaceError"](#) exception.
3. Return a new [doctype](#), with *qualifiedName* as its [name](#), *publicId* as its [public ID](#), and *systemId* as its [system ID](#), and with its [node document](#) set to the associated [document](#) of the [context object](#).

Note: No check is performed that *publicId* matches the [PublicChar](#) production or that *systemId* does not contain both a `'` and `"`.

The *createDocument*(*namespace*, *qualifiedName*, *doctype*) method must run these steps:

1. Let *document* be a new [XMLDocument](#) [see [bug 22960](#)].

Note: This method creates an [XMLDocument](#) rather than a normal [document](#). They are identical except for the addition of the [Load\(\)](#) method deployed content relies upon. [\[HTML\]](#)

2. Let *element* be null.
3. If *qualifiedName* is not the empty string, set *element* to the result of invoking the [createElementNS\(\)](#) method with the arguments *namespace* and *qualifiedName* on *document*. Rethrow any exceptions.
4. If *doctype* is not null, [append](#) *doctype* to *document*.
5. If *element* is not null, [append](#) *element* to *document*.
6. *document*'s [origin](#) is an [alias](#) to the [origin](#) of the [context object](#)'s associated [document](#), and *document*'s [effective script origin](#) is an [alias](#) to the [effective script origin](#) of the [context object](#)'s associated [document](#). [\[HTML\]](#)
7. Return *document*.

The *createHTMLDocument*(*title*) method must run these steps:

1. Let *doc* be a new [document](#) that is an [HTML document](#).
2. Set *doc*'s [content type](#) to "text/html".

3. Create a [doctype](#), with "html" as its [name](#) and with its [node document](#) set to *doc*. [Append](#) the newly created node to *doc*.
4. Create an html element in the [HTML namespace](#), and [append](#) it to *doc*.
5. Create a head element in the [HTML namespace](#), and [append](#) it to the html element created in the previous step.
6. If the *title* argument is not omitted:
 1. Create a title element in the [HTML namespace](#), and [append](#) it to the head element created in the previous step.
 2. Create a [Text node](#), set its [data](#) to *title* (which could be the empty string), and [append](#) it to the title element created in the previous step.
7. Create a body element in the [HTML namespace](#), and [append](#) it to the html element created in the earlier step.
8. *doc*'s [origin](#) is an [alias](#) to the [origin](#) of the [context object](#)'s associated [document](#), and *doc*'s [effective script origin](#) is an [alias](#) to the [effective script origin](#) of the [context object](#)'s associated [document](#). [\[HTML\]](#)
9. Return *doc*.

The *hasFeature()* method must return true.

Note: *hasFeature()* originally would report whether the user agent claimed to support a given DOM feature, but experience proved it was not nearly as reliable or granular as simply checking whether the desired objects, attributes, or methods existed. As such, it should no longer be used, but continues to exist (and simply returns true) so that old pages don't stop working.

4.6 Interface [DocumentFragment](#)

IDL `[Constructor,
Exposed=Window]
interface DocumentFragment : Node {
};`

A [DocumentFragment node](#) can have an associated [element](#) named *host*.

An object *A* is a *host-including inclusive ancestor* of an object *B*, if either *A* is an [inclusive ancestor](#) of *B*, or if *B*'s [root](#) has an associated [host](#) and *A* is a [host-including inclusive ancestor](#) of *B*'s [root](#)'s [host](#).

Note: The [DocumentFragment node](#)'s [host](#) concept is useful for HTML's [template element](#) and the [ShadowRoot](#) object and impacts the [pre-insert](#) and [replace](#) algorithms.

This box is non-normative. Implementation requirements are given below this box.

```
tree = new DocumentFragment()
```

Returns a new [DocumentFragment](#) [node](#).

The `DocumentFragment()` constructor must return a new [DocumentFragment](#) [node](#) whose [node document](#) is the global object's associated [document](#).

4.7 Interface [DocumentType](#)

```
IDL [Exposed=Window]
interface DocumentType : Node {
    readonly attribute DOMString name;
    readonly attribute DOMString publicId;
    readonly attribute DOMString systemId;
};
```

[DocumentType](#) [nodes](#) are simply known as *doctype*s.

[Doctype](#)s have an associated *name*, *public ID*, and *system ID*.

When a [doctype](#) is created, its [name](#) is always given. Unless explicitly given when a [doctype](#) is created, its [public ID](#) and [system ID](#) are the empty string.

The *name* attribute must return the [name](#).

The *publicId* attribute must return the [public ID](#).

The *systemId* attribute must return the [system ID](#).

4.8 Interface [Element](#)

```
IDL [Exposed=Window]
interface Element : Node {
    readonly attribute DOMString? namespaceURI;
    readonly attribute DOMString? prefix;
    readonly attribute DOMString localName;
    readonly attribute DOMString tagName;

    attribute DOMString id;
    attribute DOMString className;
    [SameObject] readonly attribute DOMTokenList classList;

    [SameObject] readonly attribute NamedNodeMap attributes;
    DOMString? getAttribute(DOMString name);
    DOMString? getAttributeNS(DOMString? namespace, DOMString localName);
    void setAttribute(DOMString name, DOMString value);
    void setAttributeNS(DOMString? namespace, DOMString name, DOMString
value);
    void removeAttribute(DOMString name);
    void removeAttributeNS(DOMString? namespace, DOMString localName);
    boolean hasAttribute(DOMString name);
    boolean hasAttributeNS(DOMString? namespace, DOMString localName);

    HTMLCollection getElementsByTagName(DOMString localName);
    HTMLCollection getElementsByTagNameNS(DOMString? namespace, DOMString
localName);
```

```
HTMLCollection getElementsByClassName(DOMString classNames);  
};
```

Element [nodes](#) are simply known as *elements*.

[Elements](#) have an associated *namespace*, *namespace prefix*, and *local name*. When an [element](#) is created, its [local name](#) is always given. Unless explicitly given when an [element](#) is created, its [namespace](#) and [namespace prefix](#) are null.

[Elements](#) also have an ordered *attribute list*. Unless explicitly given when an [element](#) is created, its [attribute list](#) is empty. An [element](#) has an [attribute](#) A if A is in its [attribute list](#).

[Applicable specifications](#) and this specification (can) use the hooks an *attribute is set*, an *attribute is changed*, an *attribute is added*, and an *attribute is removed*, for further processing of the [attribute's value](#).

To *get an attribute* for an [element](#) *element* using a *localName* and optionally a *namespace*, run these steps:

1. If *namespace* is not given, set it to null.
2. Return the [value](#) of the [attribute](#) in *element's attribute list* whose [namespace](#) is *namespace* and [local name](#) is *localName*, if it has one, and null otherwise.

To *set an attribute* for an [element](#) *element* using a *localName* and *value*, and optionally a *name*, *prefix*, and *namespace*, run these steps:

1. If *name* is not given, set it to *localName*.
2. If *prefix* is not given, set it to null.
3. If *namespace* is not given, set it to null.
4. Let *attribute* be the [attribute](#) in *element's attribute list* whose [namespace](#) is *namespace* and whose [local name](#) is *localName*, or null if there is no such [attribute](#).
5. If *attribute* is null, create an [attribute](#) whose [local name](#) is *localName*, [value](#) is *value*, [name](#) is *name*, [namespace](#) is *namespace*, and [namespace prefix](#) is *prefix*, and then [append](#) this [attribute](#) to *element* and terminate these steps.
6. [Change](#) *attribute* from *element* to *value*.

To *change an attribute* *attribute* from an [element](#) *element* to *value*, run these steps:

1. [Queue a mutation record](#) of "attributes" for *element* with name *attribute's local name*, namespace *attribute's namespace*, and oldValue *attribute's value*.
2. Set *attribute's value* to *value*.
3. An [attribute is set](#) and an [attribute is changed](#).

To *append an attribute* *attribute* to an [element](#) *element*, run these steps:

1. [Queue a mutation record](#) of "attributes" for *element* with name *attribute's local name*, namespace *attribute's namespace*, and oldValue null.

2. Append the *attribute* to the *element*'s [attribute list](#).
3. An [attribute is set](#) and an [attribute is added](#).

To *remove* an [attribute](#) *attribute* from an [element](#) *element*, run these steps:

1. [Queue a mutation record](#) of "attributes" for *element* with name *attribute*'s [local name](#), namespace *attribute*'s [namespace](#), and oldValue *attribute*'s [value](#).
2. Remove *attribute* from the *element*'s [attribute list](#).
3. An [attribute is removed](#).

[Elements](#) can have an associated *unique identifier (ID)* and have an associated `DOMTokenList` object. The `DOMTokenList` object's associated [attribute](#)'s [local name](#) is `class` and its associated ordered set of tokens is called the [element](#)'s *classes*.

Note: Historically [elements](#) could have multiple identifiers e.g. by using the HTML *id* [attribute](#) and a DTD. This specification makes *ID* a concept of the DOM and allows for only one per [element](#), given by an [id attribute](#).

Either when an [element](#) is created that [has](#) an [id attribute](#) whose [value](#) is not the empty string or when an [element](#)'s [id attribute](#) is [set](#) to a [value](#) other than the empty string, set the [element](#)'s *ID* to the new [value](#).

When an [element](#)'s [id attribute](#) is [removed](#) or [set](#) to the empty string, unset the [element](#)'s *ID*.

Either when an [element](#) is created that [has](#) a [class attribute](#) or when an [element](#)'s [class attribute](#) is [set](#), set the [element](#)'s *classes* to the new [value](#), [parsed](#).

When an [element](#)'s [class attribute](#) is [removed](#), set the [element](#)'s *classes* to the empty set.

Note: While this specification defines user agent processing requirements for *id* and *class* [attributes](#) on any [element](#), it makes no claims as to whether using them is conforming or not.

A [node](#)'s [parent](#) of type `Element` is known as a *parent element*. If the [node](#) has a [parent](#) of a different type, its [parent element](#) is null.

The *document element* of a [document](#) is the [element](#) whose [parent](#) is that [document](#), if it exists, and null otherwise.

Note: Per the [node tree constraints](#), there can only be one such [element](#).

When an [element](#) or one of its [ancestors](#) is the [document element](#), it is *in a document*.

This box is non-normative. Implementation requirements are given below this box.
`namespace = element.namespaceURI`

Returns the [namespace](#).

prefix = *element* . *prefix*

Returns the [namespace prefix](#).

localName = *element* . *localName*

Returns the [local name](#).

qualifiedName = *element* . *tagName*

If [namespace prefix](#) is not null, returns the concatenation of [namespace prefix](#), ":", and [local name](#). Otherwise it returns the [local name](#). (The return value is upcased in an [HTML document](#).)

The *namespaceURI* attribute must return the [context object](#)'s [namespace](#).

The *prefix* attribute must return the [context object](#)'s [namespace prefix](#).

The *localName* attribute must return the [context object](#)'s [local name](#).

The *tagName* attribute must run these steps:

1. If [context object](#)'s [namespace prefix](#) is not null, let *qualified name* be its [namespace prefix](#), followed by a ":" (U+003A), followed by its [local name](#). Otherwise, let *qualified name* be its [local name](#).
 2. If the [context object](#) is in the [HTML namespace](#) and its [node document](#) is an [HTML document](#), let *qualified name* be [converted to ASCII uppercase](#).
 3. Return *qualified name*.
-

Some IDL attributes are defined to *reflect* a particular content attribute of a given name. This means that on getting, these steps must be run:

1. [Get an attribute](#) for the [context object](#) using content attribute's name and let *value* be the result.
2. If *value* is null, return the empty string.
3. Return *value*.

On setting, [set an attribute](#) for the [context object](#) using the name of the attribute and the given value.

The *id* attribute must [reflect](#) the "id" content attribute.

The *className* attribute must [reflect](#) the "class" content attribute.

The *classList* attribute must return the associated [DOMTokenList](#) object representing the [context object](#)'s [classes](#).

The *attributes* attribute must return a [NamedNodeMap](#).

The `getAttribute(name)` method must run these steps:

1. If the [context object](#) is in the [HTML namespace](#) and its [node document](#) is an [HTML document](#), let *name* be [converted to ASCII lowercase](#).
2. Return the [value](#) of the first [attribute](#) in the [context object](#)'s [attribute list](#) whose [name](#) is *name*, and null otherwise.

The `getAttributeNS(namespace, localName)` method must return the following steps:

1. If *namespace* is the empty string, set it to null.
2. Return [getting an attribute](#) for the [context object](#) using *localName* and *namespace*.

The `setAttribute(name, value)` method must run these steps:

1. If *name* does not match the [Name](#) production in XML, [throw](#) an `"InvalidCharacterError"` exception.
2. If the [context object](#) is in the [HTML namespace](#) and its [node document](#) is an [HTML document](#), let *name* be [converted to ASCII lowercase](#).
3. Let *attribute* be the first [attribute](#) in the [context object](#)'s [attribute list](#) whose [name](#) is *name*, or null if there is no such [attribute](#).
4. If *attribute* is null, create an [attribute](#) whose [local name](#) is *name* and [value](#) is *value*, and then [append](#) this [attribute](#) to the [context object](#) and terminate these steps.
5. [Change](#) *attribute* from [context object](#) to *value*.

The `setAttributeNS(namespace, name, value)` method must run these steps:

1. If *namespace* is the empty string, set it to null.
2. If *name* does not match the [Name](#) production in XML, [throw](#) an `"InvalidCharacterError"` exception.
3. If *name* does not match the [QName](#) production in Namespaces in XML, [throw](#) a `"NamespaceError"` exception.
4. If *name* contains a ":" (U+003E), then split the string on it and let *prefix* be the part before and *localName* the part after. Otherwise, let *prefix* be null and *localName* be *name*.
5. If *prefix* is not null and *namespace* is null, [throw](#) a `"NamespaceError"` exception.
6. If *prefix* is "xml" and *namespace* is not the [XML namespace](#), [throw](#) a `"NamespaceError"` exception.
7. If *name* or *prefix* is "xmlns" and *namespace* is not the [XMLNS namespace](#), [throw](#) a `"NamespaceError"` exception.
8. If *namespace* is the [XMLNS namespace](#) and neither *name* nor *prefix* is "xmlns", [throw](#) a `"NamespaceError"` exception.

9. [Set an attribute](#) for the [context object](#) using *localName*, *value*, and also *name*, *prefix*, and *namespace*.

The *removeAttribute(name)* method must run these steps:

1. If the [context object](#) is in the [HTML namespace](#) and its [node document](#) is an [HTML document](#), let *name* be [converted to ASCII lowercase](#).
2. [Remove](#) the first [attribute](#) from the [context object](#) whose [name](#) is *name*, if any.

The *removeAttributeNS(namespace, localName)* method must return the following steps:

1. If *namespace* is the empty string, set it to null.
2. [Remove](#) the [attribute](#) from the [context object](#) whose [namespace](#) is *namespace* and [local name](#) is *localName*, if any.

The *hasAttribute(name)* method must run these steps:

1. If the [context object](#) is in the [HTML namespace](#) and its [node document](#) is an [HTML document](#), let *name* be [converted to ASCII lowercase](#).
2. Return true if the [context object has](#) an [attribute](#) whose [name](#) is *name*, and false otherwise.

The *hasAttributeNS(namespace, localName)* method must run these steps:

1. If *namespace* is the empty string, set it to null.
2. Return true if the [context object has](#) an [attribute](#) whose [namespace](#) is *namespace* and [local name](#) is *localName*, and false otherwise.

The *getElementsByTagName(localName)* method must return the [list of elements with local name localName](#) for the [context object](#).

The *getElementsByTagNameNS(namespace, localName)* method must return the [list of elements with namespace namespace and local name localName](#) for the [context object](#).

The *getElementsByClassName(classNames)* method must return the [list of elements with class names classNames](#) for the [context object](#).

4.8.1 Interface *Attr*

IDL

```
[Exposed=Window]
interface Attr {
  readonly attribute DOMString? namespaceURI;
  readonly attribute DOMString? prefix;
  readonly attribute DOMString localName;
  readonly attribute DOMString name;
  attribute DOMString value;

  readonly attribute boolean specified; // useless; always returns true
};
```


Attr objects are simply known as *attributes*. They are sometimes referred to as *content attributes* to avoid confusion with IDL attributes.

Attributes have a *namespace* (null or a non-empty string), *namespace prefix* (null or a non-empty string), *local name* (a non-empty string), *name* (a non-empty string), *value* (a string), and *element* (null or an element).

Note: If designed today they would just have a name and value.

When an attribute is created, its local name and value are always given. Unless explicitly given when an attribute is created, its name is identical to its local name, and its namespace and namespace prefix are null.

An *A attribute* is an attribute whose local name is *A* and whose namespace and namespace prefix are null.

The *namespaceURI* attribute must return the namespace.

The *prefix* attribute must return the namespace prefix.

The *localName* attribute must return the local name.

The *name* attribute's getter must return the name.

The *value* attribute's getter and *textContent* attribute's getter must both return the value.

Setting the value attribute must change value to the new value.

The value attribute's setter and textContent attribute's setter must both run these steps:

1. If context object's element is null, set context object's value to the given value.
2. Otherwise, change the context object from context object's element to the given value.

Unlike node's textContent, no special null handling is required.

The *specified* attribute must return true.

4.9 Interface CharacterData

```
IDL [Exposed=Window]
interface CharacterData : Node {
    [TreatNullAs=EmptyString] attribute DOMString data;
    readonly attribute unsigned long length;
    DOMString substringData(unsigned long offset, unsigned long count);
    void appendData(DOMString data);
    void insertData(unsigned long offset, DOMString data);
    void deleteData(unsigned long offset, unsigned long count);
    void replaceData(unsigned long offset, unsigned long count, DOMString
data);
};
```

Note: CharacterData is an abstract interface and does not exist as node. It is used by Text, Comment, and ProcessingInstruction nodes.

Each [node](#) inheriting from the [CharacterData](#) interface has an associated mutable string called *data*.

To *replace data* of node *node* with offset *offset*, count *count*, and data *data*, run these steps:

1. Let *length* be *node*'s [length](#) attribute value.
2. If *offset* is greater than *length*, [throw](#) an "[IndexSizeError](#)" exception.
3. If *offset* plus *count* is greater than *length* let *count* be *length* minus *offset*.
4. [Queue a mutation record](#) of "[characterData](#)" for *node* with oldValue *node*'s [data](#).
5. Insert *data* into *node*'s [data](#) after *offset* [code units](#).
6. Let *delete offset* be *offset* plus the number of [code units](#) in *data*.
7. Starting from *delete offset* [code units](#), remove *count* [code units](#) from *node*'s [data](#).
8. For each [range](#) whose [start node](#) is *node* and [start offset](#) is greater than *offset* but less than or equal to *offset* plus *count*, set its [start offset](#) to *offset*.
9. For each [range](#) whose [end node](#) is *node* and [end offset](#) is greater than *offset* but less than or equal to *offset* plus *count*, set its [end offset](#) to *offset*.
10. For each [range](#) whose [start node](#) is *node* and [start offset](#) is greater than *offset* plus *count*, increase its [start offset](#) by the number of [code units](#) in *data*, then decrease it by *count*.
11. For each [range](#) whose [end node](#) is *node* and [end offset](#) is greater than *offset* plus *count*, increase its [end offset](#) by the number of [code units](#) in *data*, then decrease it by *count*.

To *substring data* with node *node*, offset *offset*, and count *count*, run these steps:

1. Let *length* be *node*'s [length](#) attribute value.
2. If *offset* is greater than *length*, [throw](#) an "[IndexSizeError](#)" exception.
3. If *offset* plus *count* is greater than *length*, return a string whose value is the [code units](#) from the *offset*th [code unit](#) to the end of *node*'s [data](#), and then terminate these steps.
4. Return a string whose value is the [code units](#) from the *offset*th [code unit](#) to the *offset+count*th [code unit](#) in *node*'s [data](#).

The *data* attribute must return [data](#), and on setting, must [replace data](#) with node [context object](#) offset 0, count [length](#) attribute value, and data new value.

The *length* attribute must return the number of [code units](#) in [data](#).

The *substringData(offset, count)* method must [substring data](#) with node [context object](#), offset *offset*, and count *count*.

The *appendData(data)* method must [replace data](#) with node [context object](#), offset [length](#) attribute value, count 0, and data *data*.

The `insertData(offset, data)` method must [replace data](#) with node [context object](#), offset `offset`, count 0, and data `data`.

The `deleteData(offset, count)` method must [replace data](#) with node [context object](#), offset `offset`, count `count`, and data the empty string.

The `replaceData(offset, count, data)` method must [replace data](#) with node [context object](#), offset `offset`, count `count`, and data `data`.

4.10 Interface [Text](#)

IDL `[Constructor(optional DOMString data = ""),
Exposed=Window]
interface Text : CharacterData {
 [NewObject] Text splitText(unsigned long offset);
 readonly attribute DOMString wholeText;
};`

This box is non-normative. Implementation requirements are given below this box.
`text = new Text([data = ""])`

Returns a new [Text node](#) whose [data](#) is `data`.

`text . splitText(offset)`

Splits [data](#) at the given `offset` and returns the remainder as [Text node](#).

`text . wholeText`

Returns the combined [data](#) of all direct [Text node siblings](#).

The `Text(data)` constructor must return a new [Text node](#) whose [data](#) is `data` and [node document](#) is the global object's associated [document](#).

To *split* a [Text node](#) with offset `offset`, run these steps:

1. Let *length* be *node*'s `length` attribute value.
2. If `offset` is greater than *length*, [throw](#) an "`IndexSizeError`" exception.
3. Let *count* be *length* minus `offset`.
4. Let *new data* be the result of [substringing data](#) with node *node*, offset `offset`, and count *count*.
5. Let *new node* be a new [Text node](#), with the same [node document](#) as *node*. Set *new node*'s [data](#) to *new data*.
6. Let *parent* be *node*'s [parent](#).
7. If *parent* is not null, run these substeps:
 1. [Insert](#) *new node* into *parent* before *node*'s [next sibling](#).

2. For each [range](#) whose [start node](#) is *node* and [start offset](#) is greater than *offset*, set its [start node](#) to *new node* and decrease its [start offset](#) by *offset*.
3. For each [range](#) whose [end node](#) is *node* and [end offset](#) is greater than *offset*, set its [end node](#) to *new node* and decrease its [end offset](#) by *offset*.
4. For each [range](#) whose [start node](#) is *parent* and [start offset](#) is equal to the [index](#) of *node* + 1, increase its [start offset](#) by one.
5. For each [range](#) whose [end node](#) is *parent* and [end offset](#) is equal to the [index](#) of *node* + 1, increase its [end offset](#) by one.
8. [Replace data](#) with node *node*, offset *offset*, count *count*, and data the empty string.
9. If *parent* is null, run these substeps:
 1. For each [range](#) whose [start node](#) is *node* and [start offset](#) is greater than *offset*, set its [start offset](#) to *offset*.
 2. For each [range](#) whose [end node](#) is *node* and [end offset](#) is greater than *offset*, set its [end offset](#) to *offset*.
10. Return *new node*.

The *splitText(offset)* method must [split](#) the [context object](#) with offset *offset*.

The *contiguous Text nodes* of a node are the node itself, the [previous sibling Text](#) node (if any) and its [contiguous Text nodes](#), and the [next sibling Text](#) node (if any) and its [contiguous Text nodes](#), avoiding any duplicates.

The *wholeText* attribute must return a concatenation of the [data](#) of the [contiguous Text nodes](#) of the [context object](#), in [tree order](#).

4.11 Interface [ProcessingInstruction](#)

IDL [Exposed=Window]
 interface *ProcessingInstruction* : [CharacterData](#) {
 readonly attribute DOMString [target](#);
 };

[ProcessingInstruction nodes](#) have an associated *target*.

The *target* attribute must return the [target](#).

4.12 Interface [Comment](#)

IDL [[Constructor](#)(optional DOMString *data* = ""),
 Exposed=Window]
 interface *Comment* : [CharacterData](#) {
 };

This box is non-normative. Implementation requirements are given below this box.
`comment = new Comment([data = ""])`

Returns a new [Comment node](#) whose [data](#) is *data*.

The *Comment(data)* constructor must return a new *Comment* [node](#) whose [data](#) is *data* and [node document](#) is the global object's associated [document](#).

5 Ranges

5.1 Introduction to "DOM Ranges"

A [Range](#) object ([range](#)) represents a sequence of content within a [node tree](#). Each [range](#) has a [start](#) and an [end](#) which are [boundary points](#). A [boundary point](#) is a tuple consisting of a [node](#) and a non-negative numeric [offset](#). So in other words, a [range](#) represents a piece of content within a [node tree](#) between two [boundary points](#).

[Ranges](#) are frequently used in editing for selecting and copying content.

```
L Element: p
├── Element: img src="insanity-wolf" alt="Little-endian BOM; decode as big-endian!"
├── Text: CSS 2.1 syndata is
├── Element: em
│   └── Text: awesome
└── Text: !
```

In the [node tree](#) above, a [range](#) can be used to represent the sequence “syndata is awes”. Assuming *p* is assigned to the *p* [element](#), and *em* to the *em* [element](#), this would be done as follows:

```
var range = new Range(),
    firstText = p.childNodes[1],
    secondText = em.firstChild
range.setStart(firstText, 9) // do not forget the leading space
range.setEnd(secondText, 4)
// range now stringifies to the aforementioned quote
```

Note: [Attributes](#) such as *src* and *alt* in the [node tree](#) above cannot be represented by a [range](#). The [ranges](#) concept is only useful for [nodes](#).

[Ranges](#) are affected by mutations to the [node tree](#). Such mutations will not invalidate a [range](#) and will try to ensure that the [range](#) still represents the same piece of content. Necessarily, a [range](#) might itself be modified as part of the mutation to the [node tree](#) when e.g. part of the content it represents is mutated.

Note: See the [insert](#) and [remove](#) algorithms, the [normalize\(\)](#) method, and the [replace data](#) and [split](#) algorithms for the hairy details.

5.2 Interface [Range](#)

IDL

```
[Constructor,
 Exposed=Window]
interface Range {
  readonly attribute Node startContainer;
  readonly attribute unsigned long startOffset;
  readonly attribute Node endContainer;
  readonly attribute unsigned long endOffset;
  readonly attribute boolean collapsed;
  readonly attribute Node commonAncestorContainer;

  void setStart(Node node, unsigned long offset);
```

```

void setEnd(Node node, unsigned long offset);
void setStartBefore(Node node);
void setStartAfter(Node node);
void setEndBefore(Node node);
void setEndAfter(Node node);
void collapse(optional boolean toStart = false);
void selectNode(Node node);
void selectNodeContents(Node node);

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(unsigned short how, Range sourceRange);

void deleteContents();
[NewObject] DocumentFragment extractContents();
[NewObject] DocumentFragment cloneContents();
void insertNode(Node node);
void surroundContents(Node newParent);

[NewObject] Range cloneRange();
void detach();

boolean isPointInRange(Node node, unsigned long offset);
short comparePoint(Node node, unsigned long offset);

boolean intersectsNode(Node node);

stringifier;
};

```

Range objects are simply known as *ranges*.

A *boundary point* is a (node, offset) tuple, where offset is a non-negative integer.

Note: Generally speaking, a boundary point's offset will be between zero and the boundary point's node length, inclusive. Algorithms that modify a tree (in particular the insert, remove, replace data, and split algorithms) also modify ranges associated with that tree.

If the two nodes of boundary points (*node A*, *offset A*) and (*node B*, *offset B*) have the same root, the *position* of the first relative to the second is either *before*, *equal*, or *after*, as returned by the following algorithm:

1. If *node A* is the same as *node B*, return equal if *offset A* is the same as *offset B*, before if *offset A* is less than *offset B*, and after if *offset A* is greater than *offset B*.
2. If *node A* is following *node B*, compute the position of (*node B*, *offset B*) relative to (*node A*, *offset A*). If it is before, return after. If it is after, return before.
3. If *node A* is an ancestor of *node B*:
 1. Let *child* equal *node B*.
 2. While *child* is not a child of *node A*, set *child* to its parent.
 3. If the index of *child* is less than *offset A*, return after.

4. Return [before](#).

Each [range](#) has two associated [boundary points](#) — a *start* and *end*.

For convenience, *start node* is [start](#)'s [node](#), *start offset* is [start](#)'s [offset](#), *end node* is [end](#)'s [node](#), and *end offset* is [end](#)'s [offset](#).

The root of a [range](#) is the [root](#) of its [start node](#).

A [node](#) is contained in a [range](#) if [node](#)'s [root](#) is the same as [range](#)'s [root](#), and ([node](#), 0) is [after](#) [range](#)'s [start](#), and ([node](#), [length](#) of [node](#)) is [before](#) [range](#)'s [end](#).

A [node](#) is partially contained in a [range](#) if it is an [inclusive ancestor](#) of the [range](#)'s [start node](#) but not its [end node](#), or vice versa.

Some facts to better understand these definitions:

- The content that one would think of as being within the [range](#) consists of all [contained nodes](#), plus possibly some of the contents of the [start node](#) and [end node](#) if those are [Text](#), [ProcessingInstruction](#), Or [Comment nodes](#).
- The [nodes](#) that are contained in a [range](#) will generally not be contiguous, because the [parent](#) of a [contained node](#) will not always be [contained](#).
- However, the [descendants](#) of a [contained node](#) are [contained](#), and if two [siblings](#) are [contained](#), so are any [siblings](#) that lie between them.
- The first [contained node](#) (if there are any) will always be after the [start node](#), and the last [contained node](#) will always be equal to or before the [end node](#)'s last [descendant](#).
- The [start node](#) and [end node](#) of a [range](#) are never [contained](#) within it.
- There exists a partially contained [node](#) if and only if the [start node](#) and [end node](#) are different.
- The [commonAncestorContainer](#) attribute value is neither [contained](#) nor [partially contained](#).
- If the [start node](#) is an [ancestor](#) of the [end node](#), the common [inclusive ancestor](#) will be the [start node](#). Exactly one of its [children](#) will be [partially contained](#), and a [child](#) will be [contained](#) if and only if it [precedes](#) the [partially contained child](#). If the [end node](#) is an [ancestor](#) of the [start node](#), the opposite holds.
- If the [start node](#) is not an [inclusive ancestor](#) of the [end node](#), nor vice versa, the common [inclusive ancestor](#) will be distinct from both of them. Exactly two of its [children](#) will be [partially contained](#), and a [child](#) will be contained if and only if it lies between those two.

This box is non-normative. Implementation requirements are given below this box.
`range = new Range()`

Returns a new [range](#).

The `Range()` constructor must return a new [range](#) with (global object's associated [document](#), 0) as its [start](#) and [end](#).

This box is non-normative. Implementation requirements are given below this box.
`node = range.startContainer`

Returns *range*'s [start node](#).

`offset = range.startOffset`

Returns *range*'s [start offset](#).

`node = range.endContainer`

Returns *range*'s [end node](#).

`offset = range.endOffset`

Returns *range*'s [end offset](#).

`collapsed = range.collapsed`

Returns true if *range*'s [start](#) and [end](#) are the same, and false otherwise.

`container = range.commonAncestorContainer`

Returns the [node](#), furthest away from the [document](#), that is an [ancestor](#) of both *range*'s [start node](#) and [end node](#).

The `startContainer` attribute must return the [start node](#).

The `startOffset` attribute must return the [start offset](#).

The `endContainer` attribute must return the [end node](#).

The `endOffset` attribute must return the [end offset](#).

The `collapsed` attribute must return true if [start](#) is the same as [end](#), and false otherwise.

The `commonAncestorContainer` attribute must run these steps:

1. Let *container* be [start node](#).
2. While *container* is not an [inclusive ancestor](#) of [end node](#), let *container* be *container*'s [parent](#).
3. Return *container*.

To set the start or end of a range to a [boundary point](#) (*node*, *offset*), run these steps:

1. If *node* is a [doctype](#), [throw](#) an "[InvalidNodeTypeError](#)" exception.
 2. If *offset* is greater than *node*'s [length](#), [throw](#) an "[IndexSizeError](#)" exception.
 3. Let *bp* be the [boundary point](#) (*node*, *offset*).
 4. **↔ If these steps were invoked as "set the start"**
 1. If *bp* is [after](#) the *range*'s [end](#), or if *range*'s [root](#) is not equal to *node*'s [root](#), set *range*'s [end](#) to *bp*.
 2. Set *range*'s [start](#) to *bp*.
- ↔ If these steps were invoked as "set the end"**
1. If *bp* is [before](#) the *range*'s [start](#), or if *range*'s [root](#) is not equal to *node*'s [root](#), set *range*'s [start](#) to *bp*.
 2. Set *range*'s [end](#) to *bp*.

The *setStart*(*node*, *offset*) method must [set the start](#) of the [context object](#) to [boundary point](#) (*node*, *offset*).

The *setEnd*(*node*, *offset*) method must [set the end](#) of the [context object](#) to [boundary point](#) (*node*, *offset*).

The *setStartBefore*(*node*) method must run these steps:

1. Let *parent* be *node*'s [parent](#).
2. If *parent* is null, [throw](#) an "[InvalidNodeTypeError](#)" exception.
3. [Set the start](#) of the [context object](#) to [boundary point](#) (*parent*, *node*'s [index](#)).

The *setStartAfter*(*node*) method must run these steps:

1. Let *parent* be *node*'s [parent](#).
2. If *parent* is null, [throw](#) an "[InvalidNodeTypeError](#)" exception.
3. [Set the start](#) of the [context object](#) to [boundary point](#) (*parent*, *node*'s [index](#) plus one).

The *setEndBefore*(*node*) method must run these steps:

1. Let *parent* be *node*'s [parent](#).
2. If *parent* is null, [throw](#) an "[InvalidNodeTypeError](#)" exception.
3. [Set the end](#) of the [context object](#) to [boundary point](#) (*parent*, *node*'s [index](#)).

The *setEndAfter*(*node*) method must run these steps:

1. Let *parent* be *node*'s [parent](#).
2. If *parent* is null, [throw](#) an "[InvalidNodeTypeError](#)" exception.
3. [Set the end](#) of the [context object](#) to [boundary point](#) (*parent*, *node*'s [index](#) plus one).

The `collapse(toStart)` method, when invoked, must if `toStart` is true, set `end` to `start`, and set `start` to `end` otherwise.

To select a `node` within a `range`, run these steps:

1. Let `parent` be `node`'s `parent`.
2. If `parent` is null, `throw` an `"InvalidNodeTypeError"`.
3. Let `index` be `node`'s `index`.
4. Set `range`'s `start` to `boundary point` (`parent`, `index`).
5. Set `range`'s `end` to `boundary point` (`parent`, `index` plus one).

The `selectNode(node)` method must `select` `node` within `context object`.

The `selectNodeContents(node)` method must run these steps:

1. If `node` is a `doctype`, `throw` an `"InvalidNodeTypeError"`.
2. Let `length` be the `length` of `node`.
3. Set `start` to the `boundary point` (`node`, 0).
4. Set `end` to the `boundary point` (`node`, `length`).

The `compareBoundaryPoints(how, sourceRange)` method must run these steps:

1. If `how` is not one of
 - `START_TO_START`,
 - `START_TO_END`,
 - `END_TO_END`, and
 - `END_TO_START`,`throw` a `"NotSupportedError"` exception.
2. If `context object`'s `root` is not the same as `sourceRange`'s `root`, `throw` a `"WrongDocumentError"` exception.
3. If `how` is:
 - ↪ `START_TO_START`:
Let *this point* be the `context object`'s `start`.
Let *other point* be `sourceRange`'s `start`.
 - ↪ `START_TO_END`:
Let *this point* be the `context object`'s `end`.
Let *other point* be `sourceRange`'s `start`.

↪ **END_TO_END:**

Let *this point* be the [context object](#)'s [end](#).

Let *other point* be *sourceRange*'s [end](#).

↪ **END_TO_START:**

Let *this point* be the [context object](#)'s [start](#).

Let *other point* be *sourceRange*'s [end](#).

4. If the [position](#) of *this point* relative to *other point* is

↪ [before](#)

Return -1.

↪ [equal](#)

Return 0.

↪ [after](#)

Return 1.

The *deleteContents()* method must run these steps:

1. If [start](#) equals [end](#), terminate these steps.
2. Let *original start node*, *original start offset*, *original end node*, and *original end offset* be the [context object](#)'s [start node](#), [start offset](#), [end node](#), and [end offset](#), respectively.
3. If *original start node* and *original end node* are the same, and they are a [Text](#), [ProcessingInstruction](#), or [Comment node](#), [replace data](#) with node *original start node*, offset *original start offset*, count *original end offset* minus *original start offset*, and data the empty string, and then terminate these steps.
4. Let *nodes to remove* be a list of all the [nodes](#) that are [contained](#) in the [context object](#), in [tree order](#), omitting any [node](#) whose [parent](#) is also [contained](#) in the [context object](#).
5. If *original start node* is an [inclusive ancestor](#) of *original end node*, set *new node* to *original start node* and *new offset* to *original start offset*.
6. Otherwise:
 1. Let *reference node* equal *original start node*.
 2. While *reference node*'s [parent](#) is not null and is not an [inclusive ancestor](#) of *original end node*, set *reference node* to its [parent](#).
 3. Set *new node* to the [parent](#) of *reference node*, and *new offset* to one plus the [index](#) of *reference node*.

Note: If *reference node*'s [parent](#) were null, it would be the [root](#) of the [context object](#), so would be an [inclusive ancestor](#) of *original end node*, and we could not reach this point.

7. If *original start node* is a [Text](#), [ProcessingInstruction](#), or [Comment node](#), [replace data](#) with node *original start node*, offset *original start offset*, count *original start node*'s [length](#) minus *original start offset*, data the empty string.

8. For each *node* in *nodes to remove*, in [tree order](#), [remove](#) *node* from its [parent](#).
9. If *original end node* is a [Text](#), [ProcessingInstruction](#), or [Comment node](#), [replace data](#) with *node original end node*, offset 0, count *original end offset* and data the empty string.
10. Set [start](#) and [end](#) to (*new node*, *new offset*).

To extract a [range](#) *range*, run these steps:

1. Let *fragment* be a new [DocumentFragment node](#) whose [node document](#) is *range's start node's node document.*
2. If *range's start* equals its [end](#), return *fragment*.
3. Let *original start node*, *original start offset*, *original end node*, and *original end offset* be *range's start node*, [start offset](#), [end node](#), and [end offset](#), respectively.
4. If *original start node* equals *original end node*, and they are a [Text](#), [ProcessingInstruction](#), or [Comment node](#):
 1. Let *clone* be a [clone](#) of *original start node*.
 2. Set the [data](#) of *clone* to the result of [substringing data](#) with *node original start node*, offset *original start offset*, and count *original end offset* minus *original start offset*.
 3. [Append](#) *clone* to *fragment*.
 4. [Replace data](#) with *node original start node*, offset *original start offset*, count *original end offset* minus *original start offset*, and data the empty string.
 5. Return *fragment*.
5. Let *common ancestor* be *original start node*.
6. While *common ancestor* is not an [inclusive ancestor](#) of *original end node*, set *common ancestor* to its own [parent](#).
7. Let *first partially contained child* be null.
8. If *original start node* is not an [inclusive ancestor](#) of *original end node*, set *first partially contained child* to the first [child](#) of *common ancestor* that is [partially contained](#) in *range*.
9. Let *last partially contained child* be null.
10. If *original end node* is not an [inclusive ancestor](#) of *original start node*, set *last partially contained child* to the last [child](#) of *common ancestor* that is [partially contained](#) in *range*.

Note: These variable assignments do actually always make sense. For instance, if *original start node* is not an [inclusive ancestor](#) of *original end node*, *original start node* is itself [partially contained](#) in *range*, and so are all its [ancestors](#) up until a [child](#) of *common ancestor*. *common ancestor* cannot be *original start node*, because it has to be an

inclusive ancestor of original end node. The other case is similar. Also, notice that the two children will never be equal if both are defined.

11. Let *contained children* be a list of all children of common ancestor that are contained in range, in tree order.
12. If any member of *contained children* is a doctype, throw a "HierarchyRequestError" exception.

Note: We do not have to worry about the first or last partially contained node, because a doctype can never be partially contained. It cannot be a boundary point of a range, and it cannot be the ancestor of anything.

13. If *original start node* is an inclusive ancestor of *original end node*, set *new node* to *original start node* and *new offset* to *original start offset*.
14. Otherwise:
 1. Let *reference node* equal *original start node*.
 2. While *reference node*'s parent is not null and is not an inclusive ancestor of *original end node*, set *reference node* to its parent.
 3. Set *new node* to the parent of *reference node*, and *new offset* to one plus *reference node*'s index.

Note: If *reference node*'s parent is null, it would be the root of range, so would be an inclusive ancestor of original end node, and we could not reach this point.

15. If *first partially contained child* is a Text, ProcessingInstruction, or Comment node:

Note: In this case, first partially contained child is original start node.

1. Let *clone* be a clone of *original start node*.
 2. Set the data of *clone* to the result of substringing data with node *original start node*, offset *original start offset*, and count *original start node*'s length minus *original start offset*.
 3. Append *clone* to *fragment*.
 4. Replace data with node *original start node*, offset *original start offset*, count *original start node*'s length minus *original start offset*, and data the empty string.
16. Otherwise, if *first partially contained child* is not null:
 1. Let *clone* be a clone of *first partially contained child*.
 2. Append *clone* to *fragment*.
 3. Let *subrange* be a new range whose start is (*original start node*, *original start offset*) and whose end is (*first partially contained child*, *first partially contained*

child's [length](#)).

4. Let *subfragment* be the result of [extracting](#) *subrange*.
5. [Append](#) *subfragment* to *clone*.
17. For each *contained child* in *contained children*, [append](#) *contained child* to *fragment*.
18. If *last partially contained child* is a [Text](#), [ProcessingInstruction](#), Or [Comment](#) [node](#):

Note: In this case, last partially contained child is original end node.

1. Let *clone* be a [clone](#) of *original end node*.
2. Set the [data](#) of *clone* to the result of [substringing data](#) with node *original end node*, offset 0, and count *original end offset*.
3. [Append](#) *clone* to *fragment*.
4. [Replace data](#) with node *original end node*, offset 0, count *original end offset*, and data the empty string.
19. Otherwise, if *last partially contained child* is not null:
 1. Let *clone* be a [clone](#) of *last partially contained child*.
 2. [Append](#) *clone* to *fragment*.
 3. Let *subrange* be a new [range](#) whose [start](#) is (*last partially contained child*, 0) and whose [end](#) is (*original end node*, *original end offset*).
 4. Let *subfragment* be the result of [extracting](#) *subrange*.
 5. [Append](#) *subfragment* to *clone*.
20. Set *range's* [start](#) and [end](#) to (*new node*, *new offset*).
21. Return *fragment*.

The *extractContents()* method must return the result of [extracting context object](#).

To *clone* a [range](#) *range*, run these steps:

1. Let *fragment* be a new [DocumentFragment](#) [node](#) whose [node document](#) is *range's* [start node's](#) [node document](#).
2. If *range's* [start](#) equals its [end](#), return *fragment*.
3. Let *original start node*, *original start offset*, *original end node*, and *original end offset* be *range's* [start node](#), [start offset](#), [end node](#), and [end offset](#), respectively.
4. If *original start node* equals *original end node*, and they are a [Text](#), [ProcessingInstruction](#), Or [Comment](#) [node](#):
 1. Let *clone* be a [clone](#) of *original start node*.

2. Set the [data](#) of *clone* to the result of [substringing data](#) with node *original start node*, offset *original start offset*, and count *original end offset* minus *original start offset*.
3. [Append](#) *clone* to *fragment*.
4. Return *fragment*.
5. Let *common ancestor* be *original start node*.
6. While *common ancestor* is not an [inclusive ancestor](#) of *original end node*, set *common ancestor* to its own [parent](#).
7. Let *first partially contained child* be null.
8. If *original start node* is not an [inclusive ancestor](#) of *original end node*, set *first partially contained child* to the first [child](#) of *common ancestor* that is [partially contained](#) in *range*.
9. Let *last partially contained child* be null.
10. If *original end node* is not an [inclusive ancestor](#) of *original start node*, set *last partially contained child* to the last [child](#) of *common ancestor* that is [partially contained](#) in *range*.

Note: These variable assignments do actually always make sense. For instance, if *original start node* is not an [inclusive ancestor](#) of *original end node*, *original start node* is itself [partially contained](#) in *range*, and so are all its [ancestors](#) up until a [child](#) of *common ancestor*. *common ancestor* cannot be *original start node*, because it has to be an [inclusive ancestor](#) of *original end node*. The other case is similar. Also, notice that the two [children](#) will never be equal if both are defined.

11. Let *contained children* be a list of all [children](#) of *common ancestor* that are [contained](#) in *range*, in [tree order](#).
12. If any member of *contained children* is a [doctype](#), [throw](#) a "[HierarchyRequestError](#)" exception.

Note: We do not have to worry about the first or last partially contained node, because a [doctype](#) can never be partially contained. It cannot be a boundary point of a range, and it cannot be the ancestor of anything.

13. If *first partially contained child* is a [Text](#), [ProcessingInstruction](#), Or [Comment](#) [node](#):

Note: In this case, first partially contained child is *original start node*.

1. Let *clone* be a [clone](#) of *original start node*.
2. Set the [data](#) of *clone* to the result of [substringing data](#) with node *original start node*, offset *original start offset*, and count *original start node's length* minus *original start offset*.

3. [Append](#) *clone* to *fragment*.
14. Otherwise, if *first partially contained child* is not null:
 1. Let *clone* be a [clone](#) of *first partially contained child*.
 2. [Append](#) *clone* to *fragment*.
 3. Let *subrange* be a new [range](#) whose [start](#) is (*original start node*, *original start offset*) and whose [end](#) is (*first partially contained child*, *first partially contained child's length*).
 4. Let *subfragment* be the result of [cloning](#) *subrange*.
 5. [Append](#) *subfragment* to *clone*.
15. For each *contained child* in *contained children*:
 1. Let *clone* be a [clone](#) of *contained child* with the *clone children flag* set.
 2. [Append](#) *clone* to *fragment*.
16. If *last partially contained child* is a [Text](#), [ProcessingInstruction](#), Or [Comment](#) [node](#):

Note: In this case, last partially contained child is original end node.

 1. Let *clone* be a [clone](#) of *original end node*.
 2. Set the [data](#) of *clone* to the result of [substringing data](#) with node *original end node*, offset 0, and count *original end offset*.
 3. [Append](#) *clone* to *fragment*.
17. Otherwise, if *last partially contained child* is not null:
 1. Let *clone* be a [clone](#) of *last partially contained child*.
 2. [Append](#) *clone* to *fragment*.
 3. Let *subrange* be a new [range](#) whose [start](#) is (*last partially contained child*, 0) and whose [end](#) is (*original end node*, *original end offset*).
 4. Let *subfragment* be the result of [cloning](#) *subrange*.
 5. [Append](#) *subfragment* to *clone*.
18. Return *fragment*.

The `cloneContents()` method must return the result of [cloning context object](#).

To insert a [node](#) *node* into a [range](#) *range*, run these steps:

1. If *range's start node* is either a [ProcessingInstruction](#) or [Comment](#) [node](#), or a [Text](#) [node](#) whose [parent](#) is null, [throw](#) an "[HierarchyRequestError](#)" exception.
2. Let *referenceNode* be null.

3. If *range*'s [start node](#) is a [Text node](#), set *referenceNode* to that [Text node](#).
4. Otherwise, set *referenceNode* to the [child](#) of [start node](#) whose [index](#) is [start offset](#), and null if there is no such [child](#).
5. Let *parent* be *range*'s [start node](#) if *referenceNode* is null, and *referenceNode*'s [parent](#) otherwise.
6. [Ensure pre-insertion validity](#) of *node* into *parent* before *referenceNode*.
7. If *range*'s [start node](#) is a [Text node](#), [split](#) it with offset *range*'s [start offset](#), set *referenceNode* to the result, and set *parent* to *referenceNode*'s [parent](#).
8. If *node* equals *referenceNode*, set *referenceNode* to its [next sibling](#).
9. If *node*'s [parent](#) is not null, [remove](#) *node* from its [parent](#).
10. Let *newOffset* be *parent*'s [length](#) if *referenceNode* is null, and *referenceNode*'s [index](#) otherwise.
11. Increase *newOffset* by *node*'s [length](#) if *node* is a [DocumentFragment node](#), and one otherwise.
12. [Pre-insert](#) *node* into *parent* before *referenceNode*.
13. If *range*'s [start](#) and [end](#) are the same, set *range*'s [end](#) to (*parent*, *newOffset*).

The *insertNode(node)* method must [insert](#) *node* into [context object](#).

The *surroundContents(newParent)* method must run these steps:

1. If a non-[Text node](#) is [partially contained](#) in the [context object](#), [throw](#) an "[InvalidStateError](#)" exception.
2. If *newParent* is a [Document](#), [DocumentType](#), Or [DocumentFragment node](#), [throw](#) an "[InvalidNodeTypeError](#)" exception.
3. Let *fragment* be the result of [extracting context object](#).
4. If *newParent* has [children](#), [replace all](#) with null within *newParent*.
5. [Insert](#) *newParent* into [context object](#).
6. [Append](#) *fragment* to *newParent*.
7. [Select](#) *newParent* within [context object](#).

The *cloneRange()* method must return a new [range](#) with the same [start](#) and [end](#) as the [context object](#).

The *detach()* method must do nothing. ■ **Note: Its functionality (disabling a [Range object](#)) was removed, but the method itself is preserved for compatibility.**

This box is non-normative. Implementation requirements are given below this box.
~~**position = range.comparePoint(parent, offset)**~~

Returns -1 if the point is before the range, 0 if the point is in the range, and 1 if the point is after the range.

***intersects* = *range* . intersectsNode(*node*)**

Returns whether *range* intersects *node*.

The *isPointInRange*(*node*, *offset*) must run these steps:

1. If *node*'s [root](#) is different from the [context object](#)'s [root](#), return false.
2. If *node* is a [doctype](#), [throw](#) an "[InvalidNodeTypeError](#)" exception.
3. If *offset* is greater than *node*'s [length](#), [throw](#) an "[IndexSizeError](#)" exception.
4. If (*node*, *offset*) is [before start](#) or [after end](#), return false.
5. Return true.

The *comparePoint*(*node*, *offset*) method must run these steps:

1. If *node*'s [root](#) is different from the [context object](#)'s [root](#), [throw](#) a "[WrongDocumentError](#)" exception.
 2. If *node* is a [doctype](#), [throw](#) an "[InvalidNodeTypeError](#)" exception.
 3. If *offset* is greater than *node*'s [length](#), [throw](#) an "[IndexSizeError](#)" exception.
 4. If (*node*, *offset*) is [before start](#), return -1.
 5. If (*node*, *offset*) is [after end](#), return 1.
 6. Return 0.
-

The *intersectsNode*(*node*) method must run these steps:

1. If *node*'s [root](#) is different from the [context object](#)'s [root](#), return false.
 2. Let *parent* be *node*'s [parent](#).
 3. If *parent* is null, return true.
 4. Let *offset* be *node*'s [index](#).
 5. If (*parent*, *offset*) is [before end](#) and (*parent*, *offset* + 1) is [after start](#), return true.
 6. Return false.
-

The *stringifier* must run these steps:

1. Let *s* be the empty string.

2. If [start node](#) equals [end node](#), and it is a [Text node](#), return the substring of that [Text node](#)'s [data](#) beginning at [start offset](#) and ending at [end offset](#).
 3. If [start node](#) is a [Text node](#), append to s the substring of that [node](#)'s [data](#) from the [start offset](#) until the end.
 4. Append to s the concatenation, in [tree order](#), of the [data](#) of all [Text nodes](#) that are [contained](#) in the [context object](#).
 5. If [end node](#) is a [Text node](#), append to s the substring of that [node](#)'s [data](#) from its start until the [end offset](#).
 6. Return s.
-

Note: The [createContextualFragment\(\)](#), [getClientRects\(\)](#), and [getBoundingClientRect\(\)](#) methods are defined in other specifications.
[\[DOMPS\]](#) [\[CSSOMVIEW\]](#)

6 Traversal

NodeIterator and TreeWalker objects can be used to filter and traverse [node trees](#).

Each NodeIterator and TreeWalker object also has an associated *root node*, *whatToShow* bitmask, and *filter* callback.

To *filter node* run these steps:

1. Let n be *node*'s nodeType attribute value minus 1.
2. If the n^{th} bit (where 0 is the least significant bit) of [whatToShow](#) is not set, return FILTER_SKIP.
3. If [filter](#) is null, return FILTER_ACCEPT.
4. Let *result* be the return value of calling [filter](#)'s `acceptNode` with *node* as argument. Rethrow any exceptions.
5. Return *result*.

6.1 Interface NodeIterator

IDL

```
[Exposed=Window]
interface NodeIterator {
    [SameObject] readonly attribute Node root;
    readonly attribute Node referenceNode;
    readonly attribute boolean pointerBeforeReferenceNode;
    readonly attribute unsigned long whatToShow;
    readonly attribute NodeFilter? filter;

    Node? nextNode();
    Node? previousNode();

    void detach();
};
```

Note: NodeIterator objects can be created using the [createNodeIterator\(\)](#) method.

Each NodeIterator object has an associated *iterator collection*, which is a [collection](#) rooted at [root](#), whose filter matches any [node](#).

Each NodeIterator object has these [removing steps](#) with *oldNode*, *oldParent*, and *oldPreviousSibling*:

1. If *oldNode* is not an [inclusive ancestor](#) of the referenceNode attribute value, terminate these steps.
2. If the pointerBeforeReferenceNode attribute value is true, run these substeps:
 1. Let *nextSibling* be *oldPreviousSibling*'s [next sibling](#), if *oldPreviousSibling* is non-null, and *oldParent*'s [first child](#) otherwise.

2. If *nextSibling* is non-null, set the referenceNode attribute to *nextSibling* and terminate these steps.
3. Let *next* be the first node following *oldParent*.
4. If *next* is not an inclusive ancestor of root, set the referenceNode attribute to *next* and terminate these steps.
5. Otherwise, set the pointerBeforeReferenceNode attribute to false.

Note: Steps are not terminated here.

3. Set the referenceNode attribute to first node preceding *oldPreviousSibling*, if *oldPreviousSibling* is non-null, and to *oldParent* otherwise.

Note: As mentioned earlier NodeIterator objects have an associated root node, whatToShow bitmask, and filter callback as well.

The *root* attribute must return root.

The *referenceNode* and *pointerBeforeReferenceNode* attributes must return what they were initialized to.

The *whatToShow* attribute must return whatToShow.

The *filter* attribute must return filter.

To *traverse* in direction *direction* run these steps:

1. Let *node* be the value of the referenceNode attribute.
2. Let *before node* be the value of the pointerBeforeReferenceNode attribute.
3. Run these substeps:
 1. ↪ **If direction is next**

If *before node* is false, let *node* be the first node following *node* in the iterator collection. If there is no such node return null.

If *before node* is true, set it to false.
 2. ↪ **If direction is previous**

If *before node* is true, let *node* be the first node preceding *node* in the iterator collection. If there is no such node return null.

If *before node* is false, set it to true.
 2. Filter *node* and let *result* be the return value.
 3. If *result* is FILTER_ACCEPT, go to the next step in the overall set of steps.

Otherwise, run these substeps again.

4. Set the referenceNode attribute to *node*, set the pointerBeforeReferenceNode attribute to *before node*, and return *node*.

The *nextNode()* method must traverse in direction next.

The *previousNode()* method must traverse in direction previous.

The *detach()* method must do nothing. ■ **Note: Its functionality (disabling a NodeIterator object) was removed, but the method itself is preserved for compatibility.**

6.2 Interface TreeWalker

```
IDL
[Exposed=Window]
interface TreeWalker {
  [SameObject] readonly attribute Node root;
  readonly attribute unsigned long whatToShow;
  readonly attribute NodeFilter? filter;
  attribute Node currentNode;

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

Note: TreeWalker objects can be created using the createTreeWalker() method.

Note: As mentioned earlier TreeWalker objects have an associated root node, whatToShow bitmask, and filter callback.

The *root* attribute must return root.

The *whatToShow* attribute must return whatToShow.

The *filter* attribute must return filter.

The *currentNode* attribute must return what it was initialized to.

Setting the currentNode attribute must set it to the new value.

The *parentNode()* method must run these steps:

1. Let *node* be the value of the currentNode attribute.
2. While *node* is not null and is not root, run these substeps:
 1. Let *node* be *node*'s parent.
 2. If *node* is not null and filtering *node* returns FILTER_ACCEPT, then set the currentNode attribute to *node*, return *node*.

3. Return null.

To *traverse children* of type *type*, run these steps:

1. Let *node* be the value of the currentNode attribute.
2. Set *node* to *node*'s [first child](#) if *type* is first, and *node*'s [last child](#) if *type* is last.
3. *Main*: While *node* is not null, run these substeps:
 1. [Filter](#) *node* and let *result* be the return value.
 2. If *result* is FILTER_ACCEPT, then set the currentNode attribute to *node* and return *node*.
 3. If *result* is FILTER_SKIP, run these subsubsteps:
 1. Let *child* be *node*'s [first child](#) if *type* is first, and *node*'s [last child](#) if *type* is last.
 2. If *child* is not null, set *node* to *child* and goto [Main](#).
 4. While *node* is not null, run these subsubsteps:
 1. Let *sibling* be *node*'s [next sibling](#) if *type* is first, and *node*'s [previous sibling](#) if *type* is last.
 2. If *sibling* is not null, set *node* to *sibling* and goto [Main](#).
 3. Let *parent* be *node*'s [parent](#).
 4. If *parent* is null, *parent* is [root](#), or *parent* is currentNode attribute's value, return null.
 5. Otherwise, set *node* to *parent*.
4. Return null.

The *firstChild()* method must [traverse children](#) of type first.

The *lastChild()* method must [traverse children](#) of type last.

To *traverse siblings* of type *type* run these steps:

1. Let *node* be the value of the currentNode attribute.
2. If *node* is [root](#), return null.
3. Run these substeps:
 1. Let *sibling* be *node*'s [next sibling](#) if *type* is next, and *node*'s [previous sibling](#) if *type* is previous.
 2. While *sibling* is not null, run these subsubsteps:
 1. Set *node* to *sibling*.

2. [Filter](#) *node* and let *result* be the return value.
3. If *result* is [FILTER_ACCEPT](#), then set the [currentNode](#) attribute to *node* and return *node*.
4. Set *sibling* to *node*'s [first child](#) if *type* is next, and *node*'s [last child](#) if *type* is previous.
5. If *result* is [FILTER_REJECT](#) or *sibling* is null, then set *sibling* to *node*'s [next sibling](#) if *type* is next, and *node*'s [previous sibling](#) if *type* is previous.
3. Set *node* to its [parent](#).
4. If *node* is null or is [root](#), return null.
5. [Filter](#) *node* and if the return value is [FILTER_ACCEPT](#), then return null.
6. Run these substeps again.

The *nextSibling()* method must [traverse siblings](#) of type next.

The *previousSibling()* method must [traverse siblings](#) of type previous.

The *previousNode()* method must run these steps:

1. Let *node* be the value of the [currentNode](#) attribute.
2. While *node* is not [root](#), run these substeps:
 1. Let *sibling* be the [previous sibling](#) of *node*.
 2. While *sibling* is not null, run these subsubsteps:
 1. Set *node* to *sibling*.
 2. [Filter](#) *node* and let *result* be the return value.
 3. While *result* is not [FILTER_REJECT](#) and *node* has a [child](#), set *node* to its [last child](#) and then [filter](#) *node* and set *result* to the return value.
 4. If *result* is [FILTER_ACCEPT](#), then set the [currentNode](#) attribute to *node* and return *node*.
 5. Set *sibling* to the [previous sibling](#) of *node*.
 3. If *node* is [root](#) or *node*'s [parent](#) is null, return null.
 4. Set *node* to its [parent](#).
 5. [Filter](#) *node* and if the return value is [FILTER_ACCEPT](#), then set the [currentNode](#) attribute to *node* and return *node*.
3. Return null.

The *nextNode()* method must run these steps:

1. Let *node* be the value of the [currentNode](#) attribute.

2. Let *result* be `FILTER_ACCEPT`.
3. Run these substeps:
 1. While *result* is not `FILTER_REJECT` and *node* has a [child](#), run these subsubsteps:
 1. Set *node* to its [first child](#).
 2. [Filter](#) *node* and set *result* to the return value.
 3. If *result* is `FILTER_ACCEPT`, then set the `currentNode` attribute to *node* and return *node*.
 2. If a [node](#) is [following](#) *node* and is not [following root](#), set *node* to the first such [node](#).

Otherwise, return null.

 3. [Filter](#) *node* and set *result* to the return value.
 4. If *result* is `FILTER_ACCEPT`, then set the `currentNode` attribute to *node* and return *node*.
 5. Run these substeps again.

6.3 Interface [NodeFilter](#)

IDL

```
[Exposed=Window]
callback interface NodeFilter {
    // Constants for acceptNode()
    const unsigned short FILTER_ACCEPT = 1;
    const unsigned short FILTER_REJECT = 2;
    const unsigned short FILTER_SKIP = 3;

    // Constants for whatToShow
    const unsigned long SHOW_ALL = 0xFFFFFFFF;
    const unsigned long SHOW_ELEMENT = 0x1;
    const unsigned long SHOW_ATTRIBUTE = 0x2; // historical
    const unsigned long SHOW_TEXT = 0x4;
    const unsigned long SHOW_CDATA_SECTION = 0x8; // historical
    const unsigned long SHOW_ENTITY_REFERENCE = 0x10; // historical
    const unsigned long SHOW_ENTITY = 0x20; // historical
    const unsigned long SHOW_PROCESSING_INSTRUCTION = 0x40;
    const unsigned long SHOW_COMMENT = 0x80;
    const unsigned long SHOW_DOCUMENT = 0x100;
    const unsigned long SHOW_DOCUMENT_TYPE = 0x200;
    const unsigned long SHOW_DOCUMENT_FRAGMENT = 0x400;
    const unsigned long SHOW_NOTATION = 0x800; // historical

    unsigned short acceptNode(Node node);
};
```

[NodeFilter](#) objects can be used as [filter](#) callback and provide constants for the [whatToShow](#) bitmask.

Note: It is typically implemented as a JavaScript function.

These constants can be used as callback return value:

- *FILTER_ACCEPT* (1);
- *FILTER_REJECT* (2);
- *FILTER_SKIP* (3).

These constants can be used for the [whatToShow](#) bitmask:

- *SHOW_ALL* (4294967295, FFFFFFFF in hexadecimal);
- *SHOW_ELEMENT* (1);
- *SHOW_TEXT* (4);
- *SHOW_PROCESSING_INSTRUCTION* (64, 40 in hexadecimal);
- *SHOW_COMMENT* (128, 80 in hexadecimal);
- *SHOW_DOCUMENT* (256, 100 in hexadecimal);
- *SHOW_DOCUMENT_TYPE* (512, 200 in hexadecimal);
- *SHOW_DOCUMENT_FRAGMENT* (1024, 400 in hexadecimal).

7 Sets

Note: Yes, the names [*DOMTokenList*](#) and [*DOMSettableTokenList*](#) are unfortunate legacy mishaps.

7.1 Interface [*DOMTokenList*](#)

IDL

```
interface DOMTokenList {  
  readonly attribute unsigned long length;  
  getter DOMString? item(unsigned long index);  
  boolean contains(DOMString token);  
  void add(DOMString... tokens);  
  void remove(DOMString... tokens);  
  boolean toggle(DOMString token, optional boolean force);  
  stringifier;  
  iterable<DOMString>;  
};
```

A [*DOMTokenList*](#) object has an associated ordered set of *tokens*, which is initially empty.

A [*DOMTokenList*](#) object also has an associated [*element*](#) and an [*attribute*](#)'s [*local name*](#).

A [*DOMTokenList*](#) object's *update steps* are:

1. If there is no associated [*attribute*](#) (when the object is a [*DOMSettableTokenList*](#)), terminate these steps.
2. [*Set an attribute*](#) for the associated [*element*](#) using associated [*attribute*](#)'s [*local name*](#) and the result of running the [*ordered set serializer*](#) for [*tokens*](#).

This box is non-normative. Implementation requirements are given below this box.

***tokenList* . length**

Returns the number of tokens.

***tokenList* . item(*index*)**

***tokenList*[*index*]**

Returns the token with index *index*.

***tokenList* . contains(*token*)**

Returns true if *token* is present, and false otherwise.

Throws a "[*SyntaxError*](#)" exception if *token* is the empty string.

Throws an "[*InvalidCharacterError*](#)" exception if *token* contains any [*ASCII whitespace*](#).

***tokenList* . add(*tokens*...)**

Adds all arguments passed, except those already present.

Throws a "[*SyntaxError*](#)" exception if one of the arguments is the empty string.

Throws an "[*InvalidCharacterError*](#)" exception if one of the arguments contains any [*ASCII whitespace*](#).

`tokenList . remove(tokens...)`

Removes arguments passed, if they are present.

Throws a "[SyntaxError](#)" exception if one of the arguments is the empty string.

Throws an "[InvalidCharacterError](#)" exception if one of the arguments contains any [ASCII whitespace](#).

`tokenList . toggle(token [, force])`

If *force* is not given, "toggles" *token*, removing it if it's present and adding it if it's not. If *force* is true, adds *token* (same as [add\(\)](#)). If *force* is false, removes *token* (same as [remove\(\)](#)).

Returns true if *token* is now present, and false otherwise.

Throws a "[SyntaxError](#)" exception if *token* is empty.

Throws an "[InvalidCharacterError](#)" exception if *token* contains any spaces.

The *length* attribute must return the number of tokens in the [tokens](#).

The object's [supported property indices](#) are the numbers in the range zero to the number of tokens in [tokens](#) minus one, unless [tokens](#) is empty, in which case there are no [supported property indices](#).

The *item(index)* method must run these steps:

1. If *index* is equal to or greater than the number of tokens in [tokens](#), return null.
2. Return the *index*th token in [tokens](#).

The *contains(token)* method must run these steps:

1. If *token* is the empty string, then [throw](#) a "[SyntaxError](#)" exception.
2. If *token* contains any [ASCII whitespace](#), then [throw](#) an "[InvalidCharacterError](#)" exception.
3. Return true if *token* is in [tokens](#), and false otherwise.

The *add(tokens...)* method must run these steps:

1. If one of *tokens* is the empty string, [throw](#) a "[SyntaxError](#)" exception.
2. If one of *tokens* contains any [ASCII whitespace](#), then [throw](#) an "[InvalidCharacterError](#)" exception.
3. For each *token* in *tokens*, in given order, that is not in [tokens](#), append *token* to [tokens](#).
4. Run the [update steps](#).

The *remove(tokens...)* method must run these steps:

1. If one of *tokens* is the empty string, [throw](#) a "[SyntaxError](#)" exception.

2. If one of *tokens* contains any [ASCII whitespace](#), then [throw](#) an `"InvalidCharacterError"` exception.
3. For each *token* in *tokens*, remove *token* from [tokens](#).
4. Run the [update steps](#).

The *toggle(token, force)* method must run these steps:

1. If *token* is the empty string, [throw](#) a `"SyntaxError"` exception.
2. If *token* contains any [ASCII whitespace](#), [throw](#) an `"InvalidCharacterError"` exception.
3. If *token* is in [tokens](#), run these substeps:
 1. If *force* is either not passed or is false, then remove *token* from [tokens](#), run the [update steps](#), and return false.
 2. Otherwise, return true.
4. Otherwise, run these substeps:
 1. If *force* is passed and is false, return false.
 2. Otherwise, append *token* to [tokens](#), run the [update steps](#), and return true.

The *stringifier* must return the result of the [ordered set serializer](#) for [tokens](#).

7.2 Interface [DOMSettableTokenList](#)

```
IDL interface DOMSettableTokenList : DOMTokenList {
    attribute DOMString value;
};
```

A [DOMSettableTokenList](#) object is equivalent to a [DOMTokenList](#) object without an associated [attribute](#).

This box is non-normative. Implementation requirements are given below this box.

tokenList . value

Returns the associated set as string.

Can be set, to change the associated set via a string.

The *value* attribute must return the result of the [ordered set serializer](#) for [tokens](#).

Setting the *value* attribute must run the [ordered set parser](#) for the given value and set [tokens](#) to the result.

8 Historical

As explained in [goals](#) this specification is a significant revision of various DOM specifications. This section attempts to enumerate the changes.

8.1 DOM Events

These are the changes made to the features described in the "DOM Event Architecture", "Basic Event Interfaces", "Mutation Events", and "Mutation Name Event Types" chapters of *UI Events Specification (formerly DOM Level 3 Events)*. [\[UIEVENTS\]](#)

- Events have constructors now.
- Removes *MutationEvent*, and *MutationNameEvent*.
- Fire is no longer synonymous with dispatch, but includes initializing an event.
- The propagation and canceled flags are unset when invoking `initEvent()` rather than after dispatch.

8.2 DOM Core

These are the changes made to the features described in *DOM Level 3 Core*.

DOMString and *DOMTimeStamp* are now defined in Web IDL.

Node now inherits from *EventTarget*.

Nodes are implicitly *adopted* across *document* boundaries.

Doctypes now always have a *node document* and can be moved across *document* boundaries.

ProcessingInstruction now inherits from *CharacterData*.

attributes moved from *Node* to *Element*.

namespaceURI, prefix, and localName moved from *Node* to *Element* and *Attr*.

The remainder of interfaces and interface members listed in this section were removed to simplify the DOM platform. Implementations conforming to this specification will not support them.

⚠Warning! It is not yet clear if it would be web-compatible to remove all the following features. The editors welcome any data showing that some of these features should be reintroduced.

Interfaces:

- *CDATASection*
- *DOMConfiguration*
- *DOMErrorHandler*
- *DOMImplementationList*
- *DOMImplementationSource*
- *DOMLocator*

- *DOMObject*
- *DOMStringList*
- *DOMUserData*
- *Entity*
- *EntityReference*
- *NameList*
- *Notation*
- *TypeInfo*
- *UserDataHandler*

Interface members:

Node

- *hasAttributes()*
- *attributes*
- *namespaceURI*
- *prefix*
- *localName*
- *isSupported*
- *getFeature()*
- *getUserData()*
- *setUserData()*
- *isSameNode()*

Document

- *createCDATASection()*
- *createAttribute()*
- *createAttributeNS()*
- *createEntityReference()*
- *inputEncoding*
- *xmlEncoding*
- *xmlStandalone*
- *xmlVersion*
- *strictErrorChecking*
- *domConfig*
- *normalizeDocument()*
- *renameNode()*

DOMImplementation

- *getFeature()*

Attr

No longer inherits from Node and therefore completely changed.

Element

- *getAttributeNode()*
- *getAttributeNodeNS()*
- *setAttributeNode()*
- *removeAttributeNode()*
- *schemaTypeInfo*
- *setIdAttribute()*
- *setIdAttributeNS()*
- *setIdAttributeNode()*

DocumentType

- *entities*
- *notations*
- *internalSubset*

Text

- *isElementContentWhitespace*
- *replaceWholeText()*

8.3 DOM Ranges

These are the changes made to the features described in the "Document Object Model Range" chapter of *DOM Level 2 Traversal and Range*.

- *RangeException* has been removed.
- *Range* objects can now be moved between [documents](#) and used on [nodes](#) that are not [in a document](#).
- A wild *Range()* constructor appeared.
- New methods *comparePoint()*, *intersectsNode()*, and *isPointInRange()* have been added.
- *detach()* is now a no-op.
- *toString()* is now defined through IDL.

8.4 DOM Traversal

These are the changes made to the features described in the "Document Object Model Traversal" chapter of *DOM Level 2 Traversal and Range*.

- *createNodeIterator()* and *createTreeWalker()* now have optional arguments and lack a fourth argument which is no longer relevant given entity references never made it into the DOM.
- The *expandEntityReferences* attribute has been removed from the *NodeIterator* and *TreeWalker* interfaces for the aforementioned reason.
- The *referenceNode* and *pointerBeforeReferenceNode* attributes have been added to *NodeIterator* objects to align with proprietary extensions of implementations.
- *nextNode()* and *previousNode()* now throw when invoked from a *NodeFilter* to align with user agents.
- *detach()* is now a no-op.

A. Exceptions and Errors

A.1 Exceptions

⚠Warning! This entire section was moved out into [IDL](#). Refer to [\[Web IDL\]](#) for latest definition.

An *exception* is a type of object that represents an error and which can be thrown or treated as a first class value by implementations. Web IDL does not allow exceptions to be defined, but instead has a number of pre-defined exceptions that specifications can reference and throw in their definition of operations, attributes, and so on. Exceptions have an *error name*, a `DOMString`, which is the type of error the exception represents, and a *message*, which is an optional, user agent-defined value that provides human readable details of the error.

One kind of exception is a *DOMException*, which is an exception that encapsulates a name and an *optional integer code*, for compatibility with historically defined exceptions in the DOM.

For a [DOMException](#), the [error name](#) must be one of the names listed in the [error names table](#) below. The table also indicates the [DOMException](#)'s integer code for that error name, if it has one.

Exceptions can be *created* by providing its [error name](#). Exceptions can also be *thrown*, by providing the same details required to [create](#) one.

A.2 Interface DOMError

IDL

```
[Constructor(DOMString name, optional DOMString message = "")]  
interface DOMError {  
    readonly attribute DOMString name;  
    readonly attribute DOMString message;  
};
```

⚠Warning! [DOMError](#) is deprecated and MUST NOT be used. See [Error in WebIDL](#) instead.

This interface is intended for historical purpose only. As with exceptions, the [error names table](#) is used.

The `DOMError(name, message)` constructor must return a new [DOMError](#) object whose [name](#) attribute is initialized to *name* and whose [message](#) attribute is initialized to *message*.

The *name* attribute must return the value it was initialized to.

The *message* attribute must return the value it was initialized to.

The value of the [message](#) will typically be implementation-dependent and for informational purposes only.

A *name DOMError* means a `DOMError` object whose `name` attribute is initialized to *name* and whose `message` attribute is initialized to a helpful implementation-dependent message that explains the error.

A.3 Error names

The *error names table* below lists all the allowed error names, a description, and legacy `code` exception field values (when the error name is used for [throwing](#) an exception).

⚠Warning! This entire section was moved out into [WebIDL](#). Refer to [\[Web IDL\]](#) for latest definition.

Name	Description	Legacy <code>code</code> exception field value (if any)
" <i>IndexSizeError</i> "	The index is not in the allowed range.	<code>INDEX_SIZE_ERR</code> (1)
" <i>HierarchyRequestError</i> "	The operation would yield an incorrect node tree .	<code>HIERARCHY_REQUEST_ERR</code> (3)
" <i>WrongDocumentError</i> "	The object is in the wrong document .	<code>WRONG_DOCUMENT_ERR</code> (4)
" <i>InvalidCharacterError</i> "	The string contains invalid characters.	<code>INVALID_CHARACTER_ERR</code> (5)
" <i>NoModificationAllowedError</i> "	The object can not be modified.	<code>NO_MODIFICATION_ALLOWED_ERR</code> (7)
" <i>NotFoundError</i> "	The object can not be found here.	<code>NOT_FOUND_ERR</code> (8)
" <i>NotSupportedError</i> "	The operation is not supported.	<code>NOT_SUPPORTED_ERR</code> (9)
" <i>InvalidStateError</i> "	The object is in an invalid state.	<code>INVALID_STATE_ERR</code> (11)
" <i>SyntaxError</i> "	The string did not match the expected pattern.	<code>SYNTAX_ERR</code> (12)
" <i>InvalidModificationError</i> "	The object can not be modified in this way.	<code>INVALID_MODIFICATION_ERR</code> (13)
" <i>NamespaceError</i> "	The operation is not allowed by <i>Namespaces in XML</i> . [XMLNS]	<code>NAMESPACE_ERR</code> (14)
" <i>InvalidAccessError</i> "	The object does not support the operation or argument.	<code>INVALID_ACCESS_ERR</code> (15)
" <i>SecurityError</i> "	The operation is insecure.	<code>SECURITY_ERR</code> (18)
" <i>NetworkError</i> "	A network error occurred.	<code>NETWORK_ERR</code> (19)
" <i>AbortError</i> "	The operation was aborted.	<code>ABORT_ERR</code> (20)
" <i>URLMismatchError</i> "	The given URL does not match another URL.	<code>URL_MISMATCH_ERR</code> (21)
" <i>QuotaExceededError</i> "	The quota has been exceeded.	<code>QUOTA_EXCEEDED_ERR</code> (22)
" <i>TimeoutError</i> "	The operation timed out.	<code>TIMEOUT_ERR</code> (23)

Name	Description	Legacy <u>code</u> exception field value (if any)
<i>"InvalidNodeTypeError"</i>	The supplied node is incorrect or has an incorrect ancestor for this operation.	<i>INVALID_NODE_TYPE_ERR</i> (24)
<i>"DataCloneError"</i>	The object can not be cloned.	<i>DATA_CLONE_ERR</i> (25)
<i>"EncodingError"</i>	The encoding operation (either encoded or decoding) failed.	—
<i>"NotReadableError"</i>	The I/O read operation failed.	

B. CSS Concepts

Note: This section contains concepts introduced by the Selectors Level 4 and will be removed in the future once the Selectors Level 4 is updated. Please refer to [\[SELECTORS\]](#) for up-to-date definitions.

scoping root

Some host applications may choose to scope selectors to a particular subtree or fragment of the document. The root of the scoping subtree is called the scoping root.

:scope elements

The root of the scoping subtree, the scoping root, may be either a true element (the scoping element) or a virtual one (such as a DocumentFragment).

scope-filtered

When scoping, a selector matches an element only if the element is within the scope, even if other components of the selector are outside the scope. (A scoping element is considered to be in scope.)

parse a relative selector

The method to parse a relative selector from a string source, against :scope elements refs. It returns either a complex selector list, or failure.

evaluate a selector

The method to evaluate a selector against a set of elements.

parse a selector

The method to parse a selector from a string source. It returns either a complex selector list, or failure.

References

[CSSOMVIEW]

(Non-normative)

- [Document Object Model \(DOM\) Level 2 Style](#), C. Wilson, P. Le Hégarret, V. Apparao. W3C.
- [CSSOM View Module](#), S. Pieters, G. Adams. W3C.

[DOM2TR]

(Non-normative) [Document Object Model \(DOM\) Level 2 Traversal and Range Specification](#), Joe Kesselman, Jonathan Robie, Mike Champion et al.. W3C.

[DOM3CORE]

(Non-normative) [Document Object Model \(DOM\) Level 3 Core Specification](#), Arnaud Le Hors, Philippe Le Hégarret, Lauren Wood et al.. W3C.

[DOMPS]

(Non-normative) [DOM Parsing and Serialization](#), T. Leithead. Work in Progress. W3C.

[ELEMENTTRAVERSAL]

(Non-normative) [Element Traversal Specification](#), Doug Schepers. W3C.

[ENCODING]

[Encoding](#), A. van Kesteren, J. Bell, A. Phillips. W3C.

[HTML]

[HTML5](#), R. Berjon, T. Leithead, E. Doyle Navara, E. O'Connor, S. Pfeiffer. W3C.

[RFC2119]

[Key words for use in RFCs to Indicate Requirement Levels](#), Scott Bradner. IETF.

[SELECTORS]

[Selectors Level 4](#), E. Etemad, T. Atkins. W3C.

[SELECTORSAPI]

(Non-normative) [Selectors API Level 2](#), Lachlan Hunt. W3C.

[TOUCHEVENTS]

[Touch Events](#), Doug Schepers, Sangwhan Moon, Matt Brubeck, and Arthur Barstow. W3C.

[UIEVENTS]

[UI Events \(formerly DOM Level 3 Events\)](#), Gary Kacmarcik and Travis Leithead. W3C.

[URL]

(Non-normative)

Note: URLs can be used in numerous different manners, in many differing contexts. For the purpose of producing strict URLs one may wish to consider [\[RFC3986\]](#) [\[RFC3987\]](#). The W3C URL specification defines the term URL, various algorithms for dealing with URLs, and an API for constructing, parsing, and resolving URLs. Developers of Web browsers are advised to keep abreast of the latest URL developments by tracking the progress of <https://url.spec.whatwg.org/>. We expect that the W3C URL draft will evolve along the

Recommendation track as the community converges on a definition of URL processing.

- [URL](#), Anne van Kesteren. WHATWG.
- [RFC3986](#), T. Berners-Lee; R. Fielding; L. Masinter. IETF.
- [RFC3987](#), M. Duerst, M. Suignard. IETF.

[WEBIDL]

[Web IDL Level 1](#), Cameron McCormack, Boris Zbarsky, Yves Lafon, Travis Leithead. W3C.

[XML]

[Extensible Markup Language](#), Tim Bray, Jean Paoli, C. M. Sperberg-McQueen et al.. W3C.

[XMLNS]

[Namespaces in XML](#), Tim Bray, Dave Hollander, Andrew Layman et al.. W3C.

Acknowledgments

There have been a lot of people that have helped make DOM more interoperable over the years and thereby furthered the goals of this standard. Likewise many people have helped making this standard what it is today.

With that, many thanks to Adam Klein, Adrian Bateman, Alex Russell, Arkadiusz Michalski, Arnaud Le Hors, Arun Ranganathan, Björn Höhrmann, Boris Zbarsky, Brandon Slade, Brandon Wallace, Brian Kardell, Cameron McCormack, Christophe Dumez, Daniel Glazman, David Bruant, David Flanagan, David Håsäther, Dethe Elza, Dimitri Glazkov, Domenic Denicola, Dominic Cooney, Dominique Hazaël-Massieux, Don Jordan, Doug Schepers, Erik Arvidsson, Gavin Nicol, Geoffrey Sneddon, Glenn Maynard, Harald Alvestrand, Henri Sivonen, Ian Hickson, Igor Bukanov, Jacob Rossi, Jake Archibald, Jake Verbaten, James Graham, James Robinson, Jens Lindström, João Eiras, Joe Kesselman, Jonas Sicking, Jonathan Robie, Joshua Bell, Justin Summerlin, 呂康豪 (Kang-Hao Lu), Kevin Sweeney, Lachlan Hunt, Lauren Wood, Manish Goregaokar, Manish Tripathi, Marcos Caceres, Mark Miller, Mats Palmgren, Mounir Lamouri, MichaelTM Smith, Mike Champion, Ojan Vafai, Olli Pettay, Ondřej Žára, Peter Sharpe, Philip Jägenstedt, Philippe Le Hégaré, Rafael Weinstein, Rick Waldron, Robbert Broersma, Robin Berjon, Rune F. Halvorsen, Ryosuke Niwa, Seo Sanghyeon, Shiki Okasaka, Simon Pieters, Steve Byrne, Stig Halvorsen, Tab Atkins, Timo Tijhof, Tom Pixley, Travis Leithead, Vidur Apparao, Warren He, Yehuda Katz, Yoichi Osato, and Zack Weinberg for being awesome!

This standard is written by [Anne van Kesteren](#) ([Mozilla](#), annevk@annevk.nl) with substantial contributions from Aryeh Gregor ([Mozilla](#), ayg@aryeh.name) and Ms2ger ([Mozilla](#), ms2ger@gmail.com).

Per [CC0](#), to the extent possible under law, the WHATWG editors have waived all copyright and related or neighboring rights to the [WHATWG work](#).