DOM

Living Standard — Last Updated 29 June 2020

Participate:

GitHub whatwg/dom (new issue, open issues)
IRC: #whatwg on Freenode

Commits:

GitHub whatwg/dom/commits
Snapshot as of this commit
@thedomstandard

Tests:

web-platform-tests dom/ (ongoing work)

Translations **(non-normative):**

日本語

Abstract

DOM defines a platform-neutral model for events, aborting activities, and node trees.

Table of Contents

Goals

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

1. By consolidating *DOM Level 3 Core* [DOM-Level-3-Core], *Element Traversal* [ELEMENTTRAVERSAL], *Selectors API Level 2* [SELECTORS-API2], the "DOM Event Architecture" and "Basic Event Interfaces" chapters of *DOM Level 3 Events* [uievents-20031107] (specific types of events do not belong in the DOM Standard), and *DOM Level 2 Traversal and Range* [DOM-Level-2-Traversal-Range], 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 make 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 *DOM Level 3 Events* [uievents-20031107] 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. Infrastructure §

This specification depends on the Infra Standard. [INFRA]

Some of the terms used in this specification are defined in *Encoding*, *Selectors*, *Web IDL*, *XML*, and *Namespaces in XML*. [ENCODING] [SELECTORS4] [WEBIDL] [XML] [XML-NAMES]

The term **context object** is an alias for this.

Note

Usage of context object is deprecated in favor of this.

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**.

1.1. 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 null or an object, and has **children**, which is an ordered set of 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. The root of a tree is any object participating in that tree whose parent is null.

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 **inclusive sibling** is an object or one of its siblings.

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, or 0 if it has none.

1.2. Ordered sets §

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

1. Let *inputTokens* be the result of splitting *input* on ASCII whitespace.
2. Let *tokens* be a new ordered set.
3. For each *token* in *inputTokens*, append *token* to *tokens*.
4. Return *tokens*.

The **ordered set serializer** takes a *set* and returns the concatenation of *set* using U+0020 SPACE.

1.3. 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*. [SELECTORS4]
2. If *s* is failure, then throw a "`SyntaxError`" `DOMException`.
3. Return the result of match a selector against a tree with *s* and *node*'s root using scoping root *node*. [SELECTORS4].

Note

Support for namespaces within selectors is not planned and will not be added.

1.4. Namespaces §

To **validate** a *qualifiedName*, throw an "`InvalidCharacterError`" `DOMException` if *qualifiedName* does not match the `Name` or `QName` production.

To **validate and extract** a *namespace* and *qualifiedName*, run these steps:

1. If *namespace* is the empty string, set it to null.
2. Validate *qualifiedName*.
3. Let *prefix* be null.
4. Let *localName* be *qualifiedName*.
5. If *qualifiedName* contains a "`:`" (U+003E), then split the string on it and set *prefix* to the part before and *localName* to the part after.
6. If *prefix* is non-null and *namespace* is null, then throw a "`NamespaceError`" `DOMException`.
7. If *prefix* is "`xml`" and *namespace* is not the XML namespace, then throw a "`NamespaceError`" `DOMException`.
8. If either *qualifiedName* or *prefix* is "`xmlns`" and *namespace* is not the XMLNS namespace, then throw a "`NamespaceError`" `DOMException`.
9. If *namespace* is the XMLNS namespace and neither *qualifiedName* nor *prefix* is "`xmlns`", then throw a "`NamespaceError`" `DOMException`.
10. Return *namespace*, *prefix*, and *localName*.

2. Events §

2.1. 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. *ev* is passed as an argument to the 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).

Although events are typically dispatched by the user agent as the result of user interaction or the completion of some task, applications can dispatch events themselves by using what are 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.

Let's 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, {capture: 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 event'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`.

2.2. Interface `Event` §

IDL
```
[Exposed=(Window,Worker,AudioWorklet)]
interface Event {
  constructor(DOMString type, optional EventInit eventInitDict = {});

  readonly attribute DOMString type;
  readonly attribute EventTarget? target;
  readonly attribute EventTarget? srcElement; // historical
  readonly attribute EventTarget? currentTarget;
  sequence<EventTarget> composedPath();

  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();
           attribute boolean cancelBubble; // historical alias of .stopPropagation
  void stopImmediatePropagation();

  readonly attribute boolean bubbles;
  readonly attribute boolean cancelable;
           attribute boolean returnValue;  // historical
  void preventDefault();
  readonly attribute boolean defaultPrevented;
  readonly attribute boolean composed;

  [LegacyUnforgeable] readonly attribute boolean isTrusted;
  readonly attribute DOMHighResTimeStamp timeStamp;

  void initEvent(DOMString type, optional boolean bubbles = false, optional boolean cancelable =
false); // historical
};

dictionary EventInit {
  boolean bubbles = false;
  boolean cancelable = false;
  boolean composed = false;
};
```

An `Event` object is simply named an **event**. It allows for signaling that something has occurred, e.g., that an image has completed downloading.

A **potential event target** is null or an `EventTarget` object.

An event has an associated **target** (a potential event target). Unless stated otherwise it is null.

An event has an associated **relatedTarget** (a potential event target). Unless stated otherwise it is null.

Note

Other specifications use relatedTarget to define a `relatedTarget` *attribute. [UIEVENTS]*

An event has an associated **touch target list** (a list of zero or more potential event targets). Unless stated otherwise it is the empty list.

Note

The touch target list is for the exclusive use of defining the `TouchEvent` *interface and related interfaces. [TOUCH-EVENTS]*

An event has an associated **path**. A path is a list of structs. Each struct consists of an **invocation target** (an `EventTarget` object), an **invocation-target-in-shadow-tree** (a boolean), a **shadow-adjusted target** (a potential event target), a **relatedTarget** (a potential event target), a **touch target list** (a list of potential event targets), a **root-of-closed-tree** (a boolean), and a **slot-in-closed-tree** (a boolean). A path is initially the empty list.

For web developers (non-normative)

event `= new Event(`***type*** `[,` ***eventInitDict***`])`

Returns a new *event* whose `type` attribute value is set to *type*. The *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 (its target).

event `. currentTarget`

Returns the object whose event listener's callback is currently being invoked.

event `. composedPath()`

Returns the invocation target objects of *event*'s path (objects on which listeners will be invoked), except for any nodes in shadow trees of which the shadow root's mode is "`closed`" that are not reachable from *event*'s `currentTarget`.

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* goes through its target'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, and while executing a listener for the *event* with `passive` set to false, signals to the operation that caused *event* to be dispatched that it needs to be canceled.

event . `defaultPrevented`

Returns true if `preventDefault()` was invoked successfully to indicate cancelation, and false otherwise.

event . `composed`

Returns true or false depending on how *event* was initialized. True if *event* invokes listeners past a `ShadowRoot` node that is the root of its target, and false otherwise.

event . `isTrusted`

Returns true if *event* was dispatched by the user agent, and false otherwise.

event . `timeStamp`

Returns the *event*'s timestamp as the number of milliseconds measured relative to the time origin.

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`** attribute's getter, when invoked, must return this's target.

The **`srcElement`** attribute's getter, when invoked, must return this's target.

The **`currentTarget`** attribute must return the value it was initialized to. When an event is created the attribute must be initialized to null.

The **`composedPath()`** method, when invoked, must run these steps:

1. Let *composedPath* be an empty list.
2. Let *path* be this's path.
3. If *path* is empty, then return *composedPath*.
4. Let *currentTarget* be this's `currentTarget` attribute value.
5. Append *currentTarget* to *composedPath*.
6. Let *currentTargetIndex* be 0.
7. Let *currentTargetHiddenSubtreeLevel* be 0.
8. Let *index* be *path*'s size − 1.
9. While *index* is greater than or equal to 0:
    1. If *path*[*index*]'s root-of-closed-tree is true, then increase *currentTargetHiddenSubtreeLevel* by 1.
    2. If *path*[*index*]'s invocation target is *currentTarget*, then set *currentTargetIndex* to *index* and break.
    3. If *path*[*index*]'s slot-in-closed-tree is true, then decrease *currentTargetHiddenSubtreeLevel* by 1.
    4. Decrease *index* by 1.
10. Let *currentHiddenLevel* and *maxHiddenLevel* be *currentTargetHiddenSubtreeLevel*.
11. Set *index* to *currentTargetIndex* − 1.
12. While *index* is greater than or equal to 0:
    1. If *path*[*index*]'s root-of-closed-tree is true, then increase *currentHiddenLevel* by 1.
    2. If *currentHiddenLevel* is less than or equal to *maxHiddenLevel*, then prepend *path*[*index*]'s invocation target to *composedPath*.
    3. If *path*[*index*]'s slot-in-closed-tree is true, then:

            1. Decrease *currentHiddenLevel* by 1.

            2. If *currentHiddenLevel* is less than *maxHiddenLevel*, then set *maxHiddenLevel* to *currentHiddenLevel*.

        4. Decrease *index* by 1.

13. Set *currentHiddenLevel* and *maxHiddenLevel* to *currentTargetHiddenSubtreeLevel*.

14. Set *index* to *currentTargetIndex* + 1.

15. While *index* is less than *path*’s size:

    1. If *path*[*index*]'s slot-in-closed-tree is true, then increase *currentHiddenLevel* by 1.

    2. If *currentHiddenLevel* is less than or equal to *maxHiddenLevel*, then append *path*[*index*]'s invocation target to *composedPath*.

    3. If *path*[*index*]'s root-of-closed-tree is true, then:

        1. Decrease *currentHiddenLevel* by 1.

        2. If *currentHiddenLevel* is less than *maxHiddenLevel*, then set *maxHiddenLevel* to *currentHiddenLevel*.

    4. Increase *index* by 1.

16. Return *composedPath*.

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.

`AT_TARGET` **(numeric value 2)**
When an event is dispatched it will be in this phase on its target.

`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.

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**
- **in passive listener flag**
- **composed flag**
- **initialized flag**
- **dispatch flag**

The **`stopPropagation()`** method, when invoked, must set this’s stop propagation flag.

The **`cancelBubble`** attribute’s getter, when invoked, must return true if this’s stop propagation flag is set, and false otherwise.

The `cancelBubble` attribute's setter, when invoked, must set this's stop propagation flag if the given value is true, and do nothing otherwise.

The **`stopImmediatePropagation()`** method, when invoked, must set this's stop propagation flag and this's stop immediate propagation flag.

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

To **set the canceled flag**, given an event *event*, if *event*'s `cancelable` attribute value is true and *event*'s in passive listener flag is unset, then set *event*'s canceled flag, and do nothing otherwise.

The **`returnValue`** attribute's getter, when invoked, must return false if this's canceled flag is set, and true otherwise.

The `returnValue` attribute's setter, when invoked, must set the canceled flag with this if the given value is false, and do nothing otherwise.

The **`preventDefault()`** method, when invoked, must set the canceled flag with this.

Note

There are scenarios where invoking `preventDefault()` has no effect. User agents are encouraged to log the precise cause in a developer console, to aid debugging.

The **`defaultPrevented`** attribute's getter, when invoked, must return true if this's canceled flag is set, and false otherwise.

The **`composed`** attribute's getter, when invoked, must return true if this's composed 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.

Note

`isTrusted` is a convenience that indicates whether an event is dispatched by the user agent (as opposed to using `dispatchEvent()`). The sole legacy exception is `click()`, which causes the user agent to dispatch an event whose `isTrusted` attribute is initialized to false.

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

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

1. Set *event*'s initialized flag.
2. Unset *event*'s stop propagation flag, stop immediate propagation flag, and canceled flag.
3. Set *event*'s `isTrusted` attribute to false.
4. Set *event*'s target to null.
5. Set *event*'s `type` attribute to *type*.
6. Set *event*'s `bubbles` attribute to *bubbles*.
7. Set *event*'s `cancelable` attribute to *cancelable*.

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

1. If this's dispatch flag is set, then return.

2. Initialize this with *type*, *bubbles*, and *cancelable*.

Note

As events have constructors `initEvent()` is redundant and incapable of setting `composed`. It has to be supported for legacy content.

2.3. Legacy extensions to the `Window` interface §

IDL
```
partial interface Window {
  [Replaceable] readonly attribute any event; // historical
};
```

Each `Window` object has an associated **current event** (undefined or an `Event` object). Unless stated otherwise it is undefined.

The **`event`** attribute's getter, when invoked, must return this's current event.

Note

Web developers are strongly encouraged to instead rely on the `Event` object passed to event listeners, as that will result in more portable code. This attribute is not available in workers or worklets, and is inaccurate for events dispatched in shadow trees.

2.4. Interface `CustomEvent` §

IDL
```
[Exposed=(Window,Worker)]
interface CustomEvent : Event {
  constructor(DOMString type, optional CustomEventInit eventInitDict = {});

  readonly attribute any detail;

  void initCustomEvent(DOMString type, optional boolean bubbles = false, optional boolean cancelable =
false, optional any detail = null); // historical
};

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

Events using the `CustomEvent` interface can be used to carry custom data.

For web developers (non-normative)

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

Works analogously to the constructor for `Event` except that the *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 this’s dispatch flag is set, then return.
2. Initialize this with *type*, *bubbles*, and *cancelable*.
3. Set this’s `detail` attribute to *detail*.

2.5. Constructing events §

Specifications may define **event constructing steps** for all or some events. The algorithm is passed an *event* as indicated in the inner event creation steps.

Note

This construct can be used by `Event` subclasses that have a more complex structure than a simple 1:1 mapping between their initializing dictionary members and IDL attributes.

When a **constructor** of the `Event` interface, or of an interface that inherits from the `Event` interface, is invoked, these steps must be run, given the arguments *type* and *eventInitDict*:

1. Let *event* be the result of running the inner event creation steps with this interface, null, now, and *eventInitDict*.
2. Initialize *event*’s `type` attribute to *type*.
3. Return *event*.

To **create an event** using *eventInterface*, which must be either `Event` or an interface that inherits from it, and optionally given a Realm *realm*, run these steps:

1. If *realm* is not given, then set it to null.
2. Let *dictionary* be the result of converting the JavaScript value undefined to the dictionary type accepted by *eventInterface*’s constructor. (This dictionary type will either be `EventInit` or a dictionary that inherits from it.)

   This does not work if members are required; see whatwg/dom#600.

3. Let *event* be the result of running the inner event creation steps with *eventInterface*, *realm*, the time of the occurrence that the event is signaling, and *dictionary*.

   ¶ Example

   In macOS the time of the occurrence for input actions is available via the `timestamp` property of `NSEvent` objects.

4. Initialize *event*’s `isTrusted` attribute to true.
5. Return *event*.

Note

Create an event is meant to be used by other specifications which need to separately create and dispatch events, instead of simply firing them. It ensures the event’s attributes are initialized to the correct defaults.

The **inner event creation steps**, given an *interface*, *realm*, *time*, and *dictionary*, are as follows:

1. Let *event* be the result of creating a new object using *eventInterface*. If *realm* is non-null, then use that Realm; otherwise, use the default behavior defined in Web IDL.

   As of the time of this writing Web IDL does not yet define any default behavior; see heycam/webidl#135.

2. Set *event*'s initialized flag.

3. Initialize *event*'s `timeStamp` attribute to a `DOMHighResTimeStamp` representing the high resolution time from the time origin to *time*.

   ⚠ Warning!
   User agents should set a minimum resolution of event's `timeStamp` attribute to 5 microseconds following the existing clock resolution recommendation. [HR-TIME]

4. For each *member* → *value* in *dictionary*, if *event* has an attribute whose identifier is *member*, then initialize that attribute to *value*.

5. Run the event constructing steps with *event*.

6. Return *event*.

2.6. Defining event interfaces §

In general, when defining a new interface that inherits from `Event` please always ask feedback from the WHATWG or the W3C WebApps WG community.

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.

2.7. Interface `EventTarget` §

IDL
```
[Exposed=(Window,Worker,AudioWorklet)]
interface EventTarget {
  constructor();

  void addEventListener(DOMString type, EventListener? callback, optional (AddEventListenerOptions or
boolean) options = {});
  void removeEventListener(DOMString type, EventListener? callback, optional (EventListenerOptions or
boolean) options = {});
  boolean dispatchEvent(Event event);
};

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

dictionary EventListenerOptions {
  boolean capture = false;
};

dictionary AddEventListenerOptions : EventListenerOptions {
  boolean passive = false;
  boolean once = false;
};
```

An `EventTarget` object represents a target to which an event can be dispatched when something has occurred.

Each `EventTarget` object has an associated **event listener list** (a list of zero or more event listeners). It is initially the empty list.

An **event listener** can be used to observe a specific event and consists of:

- **type** (a string)
- **callback** (null or an `EventListener` object)
- **capture** (a boolean, initially false)
- **passive** (a boolean, initially false)
- **once** (a boolean, initially false)
- **removed** (a boolean for bookkeeping purposes, initially false)

Note

Although callback is an `EventListener` object, an event listener is a broader concept as can be seen above.

Each `EventTarget` object also has an associated **get the parent** algorithm, which takes an event *event*, and returns an `EventTarget` object. Unless specified otherwise it returns null.

Note

Nodes, shadow roots, and documents override the get the parent algorithm.

Each `EventTarget` object can have an associated **activation behavior** algorithm. The activation behavior algorithm is passed an event, as indicated in the dispatch algorithm.

Note

This exists because user agents perform certain actions for certain `EventTarget` objects, e.g., the `area` element, in response to synthetic `MouseEvent` events whose `type` attribute is `click`*. Web compatibility prevented it from being removed and it is now the enshrined way of defining an activation of something. [HTML]*

Each `EventTarget` object that has activation behavior, can additionally have both (not either) a **legacy-pre-activation behavior** algorithm and a **legacy-canceled-activation behavior** algorithm.

Note

These algorithms only exist for checkbox and radio `input` elements and are not to be used for anything else. [HTML]

For web developers (non-normative)

target **= new EventTarget();**

Creates a new `EventTarget` object, which can be used by developers to dispatch and listen for events.

target **. addEventListener(*type*, *callback* [, *options*])**

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.

The *options* argument sets listener-specific options. For compatibility this can be a boolean, in which case the method behaves exactly as if the value was specified as *options*'s `capture`.

When set to true, *options*'s `capture` prevents callback from being invoked when the event's `eventPhase` attribute value is `BUBBLING_PHASE`. When false (or not present), callback will not be invoked when event's `eventPhase` attribute value is `CAPTURING_PHASE`. Either way, callback will be invoked if event's `eventPhase` attribute value is `AT_TARGET`.

When set to true, *options*'s `passive` indicates that the callback will not cancel the event by invoking `preventDefault()`. This is used to enable performance optimizations described in § 2.8 Observing event listeners.

When set to true, *options*'s `once` indicates that the callback will only be invoked once after which the event listener will be removed.

The event listener is appended to *target*'s event listener list and is not appended if it has the same type, callback, and capture.

target **. removeEventListener(*type*, *callback* [, *options*])**

Removes the event listener in *target*'s event listener list with the same *type*, *callback*, and *options*.

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.

To **flatten** *options*, run these steps:

1. If *options* is a boolean, then return *options*.
2. Return *options*'s `capture`.

To **flatten more** *options*, run these steps:

1. Let *capture* be the result of flattening *options*.
2. Let *once* and *passive* be false.
3. If *options* is a dictionary, then set *passive* to *options*'s `passive` and *once* to *options*'s `once`.
4. Return *capture*, *passive*, and *once*.

The **`EventTarget()`** constructor, when invoked, must return a new `EventTarget`.

Note

Because of the defaults stated elsewhere, the returned `EventTarget`'s get the parent algorithm will return null, and it will have no activation behavior, legacy-pre-activation behavior, or legacy-canceled-activation behavior.

Note

In the future we could allow custom get the parent algorithms. Let us know if this would be useful for your programs. For now, all author-created `EventTarget`s do not participate in a tree structure.

To **add an event listener**, given an `EventTarget` object *eventTarget* and an event listener *listener*, run these steps:

1. If *eventTarget* is a `ServiceWorkerGlobalScope` object, its service worker's script resource's has ever been evaluated flag is set, and *listener*'s type matches the `type` attribute value of any of the service worker events, then report a warning to the console that this might not give the expected results. [SERVICE-WORKERS]
2. If *listener*'s callback is null, then return.
3. If *eventTarget*'s event listener list does not contain an event listener whose type is *listener*'s type, callback is *listener*'s callback, and capture is *listener*'s capture, then append *listener* to *eventTarget*'s event listener list.

Note

The add an event listener concept exists to ensure event handlers use the same code path. [HTML]

The **`addEventListener(`*`type`*`, `*`callback`*`, `*`options`*`)`** method, when invoked, must run these steps:

1. Let *capture*, *passive*, and *once* be the result of flattening more *options*.
2. Add an event listener with this and an event listener whose type is *type*, callback is *callback*, capture is *capture*, passive is *passive*, and once is *once*.

To **remove an event listener**, given an `EventTarget` object *eventTarget* and an event listener *listener*, run these steps:

1. If *eventTarget* is a `ServiceWorkerGlobalScope` object and its service worker's set of event types to handle contains *type*, then report a warning to the console that this might not give the expected results. [SERVICE-WORKERS]
2. Set *listener*'s removed to true and remove *listener* from *eventTarget*'s event listener list.

Note

HTML needs this to define event handlers. [HTML]

To **remove all event listeners**, given an `EventTarget` object *eventTarget*, for each *listener* of *eventTarget*'s event listener list, remove an event listener with *eventTarget* and *listener*.

Note

HTML needs this to define `document.open()`. *[HTML]*

The **`removeEventListener(type, callback, options)`** method, when invoked, must run these steps:

1. Let *capture* be the result of flattening *options*.
2. If this's event listener list contains an event listener whose type is *type*, callback is *callback*, and capture is *capture*, then remove an event listener with this and that event listener.

Note

The event listener list will not contain multiple event listeners with equal type, callback, *and* capture, *as add an event listener prevents that.*

The **`dispatchEvent(event)`** method, when invoked, must run these steps:

1. If *event*'s dispatch flag is set, or if its initialized flag is not set, then throw an "`InvalidStateError`" `DOMException`.
2. Initialize *event*'s `isTrusted` attribute to false.
3. Return the result of dispatching *event* to this.

2.8. Observing event listeners §

In general, developers do not expect the presence of an event listener to be observable. The impact of an event listener is determined by its callback. That is, a developer adding a no-op event listener would not expect it to have any side effects.

Unfortunately, some event APIs have been designed such that implementing them efficiently requires observing event listeners. This can make the presence of listeners observable in that even empty listeners can have a dramatic performance impact on the behavior of the application. For example, touch and wheel events which can be used to block asynchronous scrolling. In some cases this problem can be mitigated by specifying the event to be `cancelable` only when there is at least one non-`passive` listener. For example, non-`passive` `TouchEvent` listeners must block scrolling, but if all listeners are `passive` then scrolling can be allowed to start in parallel by making the `TouchEvent` uncancelable (so that calls to `preventDefault()` are ignored). So code dispatching an event is able to observe the absence of non-`passive` listeners, and use that to clear the `cancelable` property of the event being dispatched.

Ideally, any new event APIs are defined such that they do not need this property (use public-script-coord@w3.org for discussion).

2.9. Dispatching events §

To **dispatch** an *event* to a *target*, with an optional *legacy target override flag* and an optional *legacyOutputDidListenersThrowFlag*, run these steps:

1. Set *event*'s dispatch flag.
2. Let *targetOverride* be *target*, if *legacy target override flag* is not given, and *target*'s associated `Document` otherwise. [HTML]

   Note

   legacy target override flag *is only used by HTML and only when* target *is a* `Window` *object.*

3. Let *activationTarget* be null.
4. Let *relatedTarget* be the result of retargeting *event*'s relatedTarget against *target*.
5. If *target* is not *relatedTarget* or *target* is *event*'s relatedTarget, then:

1. Let *touchTargets* be a new list.
2. For each *touchTarget* of *event*'s touch target list, append the result of retargeting *touchTarget* against *target* to *touchTargets*.
3. Append to an event path with *event*, *target*, *targetOverride*, *relatedTarget*, *touchTargets*, and false.
4. Let *isActivationEvent* be true, if *event* is a `MouseEvent` object and *event*'s `type` attribute is "`click`", and false otherwise.
5. If *isActivationEvent* is true and *target* has activation behavior, then set *activationTarget* to *target*.
6. Let *slottable* be *target*, if *target* is a slottable and is assigned, and null otherwise.
7. Let *slot-in-closed-tree* be false.
8. Let *parent* be the result of invoking *target*'s get the parent with *event.*
9. While *parent* is non-null:
    1. If *slottable* is non-null:
        1. Assert: *parent* is a slot.
        2. Set *slottable* to null.
        3. If *parent*'s root is a shadow root whose mode is "`closed`", then set *slot-in-closed-tree* to true.
    2. If *parent* is a slottable and is assigned, then set *slottable* to *parent*.
    3. Let *relatedTarget* be the result of retargeting *event*'s relatedTarget against *parent*.
    4. Let *touchTargets* be a new list.
    5. For each *touchTarget* of *event*'s touch target list, append the result of retargeting *touchTarget* against *parent* to *touchTargets*.
    6. If *parent* is a `Window` object, or *parent* is a node and *target*'s root is a shadow-including inclusive ancestor of *parent*, then:
        1. If *isActivationEvent* is true, *event*'s `bubbles` attribute is true, *activationTarget* is null, and *parent* has activation behavior, then set *activationTarget* to *parent*.
        2. Append to an event path with *event*, *parent*, null, *relatedTarget*, *touchTargets*, and *slot-in-closed-tree*.
    7. Otherwise, if *parent* is *relatedTarget*, then set *parent* to null.
    8. Otherwise, set *target* to *parent* and then:
        1. If *isActivationEvent* is true, *activationTarget* is null, and *target* has activation behavior, then set *activationTarget* to *target*.
        2. Append to an event path with *event*, *parent*, *target*, *relatedTarget*, *touchTargets*, and *slot-in-closed-tree*.
    9. If *parent* is non-null, then set *parent* to the result of invoking *parent*'s get the parent with *event*.
    10. Set *slot-in-closed-tree* to false.
10. Let *clearTargetsStruct* be the last struct in *event*'s path whose shadow-adjusted target is non-null.
11. Let *clearTargets* be true if *clearTargetsStruct*'s shadow-adjusted target, *clearTargetsStruct*'s relatedTarget, or an `EventTarget` object in *clearTargetsStruct*'s touch target list is a node and its root is a shadow root, and false otherwise.
12. If *activationTarget* is non-null and *activationTarget* has legacy-pre-activation behavior, then run *activationTarget*'s legacy-pre-activation behavior.
13. For each *struct* in *event*'s path, in reverse order:
    1. If *struct*'s shadow-adjusted target is non-null, then set *event*'s `eventPhase` attribute to `AT_TARGET`.

        2. Otherwise, set *event*'s `eventPhase` attribute to `CAPTURING_PHASE`.

        3. Invoke with *struct*, *event*, "`capturing`", and *legacyOutputDidListenersThrowFlag* if given.

    14. For each *struct* in *event*'s path:

        1. If *struct*'s shadow-adjusted target is non-null, then set *event*'s `eventPhase` attribute to `AT_TARGET`.

        2. Otherwise:

            1. If *event*'s `bubbles` attribute is false, then continue.

            2. Set *event*'s `eventPhase` attribute to `BUBBLING_PHASE`.

        3. Invoke with *struct*, *event*, "`bubbling`", and *legacyOutputDidListenersThrowFlag* if given.

6. Set *event*'s `eventPhase` attribute to `NONE`.

7. Set *event*'s `currentTarget` attribute to null.

8. Set *event*'s path to the empty list.

9. Unset *event*'s dispatch flag, stop propagation flag, and stop immediate propagation flag.

10. If *clearTargets*, then:

    1. Set *event*'s target to null.

    2. Set *event*'s relatedTarget to null.

    3. Set *event*'s touch target list to the empty list.

11. If *activationTarget* is non-null, then:

    1. If *event*'s canceled flag is unset, then run *activationTarget*'s activation behavior with *event*.

    2. Otherwise, if *activationTarget* has legacy-canceled-activation behavior, then run *activationTarget*'s legacy-canceled-activation behavior.

12. Return false if *event*'s canceled flag is set, and true otherwise.

To **append to an event path**, given an *event*, *invocationTarget*, *shadowAdjustedTarget*, *relatedTarget*, *touchTargets*, and a *slot-in-closed-tree*, run these steps:

1. Let *invocationTargetInShadowTree* be false.

2. If *invocationTarget* is a node and its root is a shadow root, then set *invocationTargetInShadowTree* to true.

3. Let *root-of-closed-tree* be false.

4. If *invocationTarget* is a shadow root whose mode is "`closed`", then set *root-of-closed-tree* to true.

5. Append a new struct to *event*'s path whose invocation target is *invocationTarget*, invocation-target-in-shadow-tree is *invocationTargetInShadowTree*, shadow-adjusted target is *shadowAdjustedTarget*, relatedTarget is *relatedTarget*, touch target list is *touchTargets*, root-of-closed-tree is *root-of-closed-tree*, and slot-in-closed-tree is *slot-in-closed-tree*.

To **invoke**, given a *struct*, *event*, *phase*, and an optional *legacyOutputDidListenersThrowFlag*, run these steps:

1. Set *event*'s target to the shadow-adjusted target of the last struct in *event*'s path, that is either *struct* or preceding *struct*, whose shadow-adjusted target is non-null.

2. Set *event*'s relatedTarget to *struct*'s relatedTarget.

3. Set *event*'s touch target list to *struct*'s touch target list.

4. If *event*'s stop propagation flag is set, then return.

5. Initialize *event*'s `currentTarget` attribute to *struct*'s invocation target.

6. Let *listeners* be a clone of *event*'s `currentTarget` attribute value's event listener list.

   Note

   This avoids event listeners added after this point from being run. Note that removal still has an effect due to the removed field.

7. Let *invocationTargetInShadowTree* be *struct*'s invocation-target-in-shadow-tree.

8. Let *found* be the result of running inner invoke with *event*, *listeners*, *phase*, *invocationTargetInShadowTree*, and *legacyOutputDidListenersThrowFlag* if given.

9. If *found* is false and *event*'s `isTrusted` attribute is true, then:

    1. Let *originalEventType* be *event*'s `type` attribute value.

    2. If *event*'s `type` attribute value is a match for any of the strings in the first column in the following table, set *event*'s `type` attribute value to the string in the second column on the same row as the matching string, and return otherwise.

Event type	Legacy event type
"`animationend`"	"`webkitAnimationEnd`"
"`animationiteration`"	"`webkitAnimationIteration`"
"`animationstart`"	"`webkitAnimationStart`"
"`transitionend`"	"`webkitTransitionEnd`"

    3. Inner invoke with *event*, *listeners*, *phase*, *invocationTargetInShadowTree*, and *legacyOutputDidListenersThrowFlag* if given.

    4. Set *event*'s `type` attribute value to *originalEventType*.

To **inner invoke**, given an *event*, *listeners*, *phase*, *invocationTargetInShadowTree*, and an optional *legacyOutputDidListenersThrowFlag*, run these steps:

1. Let *found* be false.

2. For each *listener* in *listeners*, whose removed is false:

    1. If *event*'s `type` attribute value is not *listener*'s type, then continue.

    2. Set *found* to true.

    3. If *phase* is "`capturing`" and *listener*'s capture is false, then continue.

    4. If *phase* is "`bubbling`" and *listener*'s capture is true, then continue.

    5. If *listener*'s once is true, then remove *listener* from *event*'s `currentTarget` attribute value's event listener list.

    6. Let *global* be *listener* callback's associated Realm's global object.

    7. Let *currentEvent* be undefined.

    8. If *global* is a `Window` object, then:

        1. Set *currentEvent* to *global*'s current event.

        2. If *invocationTargetInShadowTree* is false, then set *global*'s current event to *event*.

    9. If *listener*'s passive is true, then set *event*'s in passive listener flag.

    10. Call a user object's operation with *listener*'s callback, "`handleEvent`", « *event* », and *event*'s `currentTarget` attribute value. If this throws an exception, then:

        1. Report the exception.

        2. Set *legacyOutputDidListenersThrowFlag* if given.

           Note
           The legacyOutputDidListenersThrowFlag *is only used by Indexed Database API. [INDEXEDDB]*

    11. Unset *event*'s in passive listener flag.

    12. If *global* is a `Window` object, then set *global*'s current event to *currentEvent*.

    13. If *event*'s stop immediate propagation flag is set, then return *found*.

3. Return *found*.

2.10. Firing events §

To **fire an event** named *e* at *target*, optionally using an *eventConstructor*, with a description of how IDL attributes are to be initialized, and a *legacy target override flag*, run these steps:

1. If *eventConstructor* is not given, then let *eventConstructor* be `Event`.

2. Let *event* be the result of creating an event given *eventConstructor*, in the relevant Realm of *target*.

3. Initialize *event*'s `type` attribute to *e*.

4. Initialize any other IDL attributes of *event* as described in the invocation of this algorithm.

   Note
   This also allows for the `isTrusted` *attribute to be set to false.*

5. Return the result of dispatching *event* at *target*, with *legacy target override flag* set if set.

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.

Example
If the event needs its `bubbles` or `cancelable` attribute initialized, one could write "fire an event named `submit` at *target* with its `cancelable` attribute initialized to true".

Or, when a custom constructor is needed, "fire an event named `click` at *target* using `MouseEvent` with its `detail` attribute initialized to 1".

Occasionally the return value is important:

1. Let *doAction* be the result of firing an event named `like` at *target*.

2. If *doAction* is true, then …

2.11. Action versus occurrence §

An event signifies an occurrence, not an action. Phrased differently, it represents a notification from an algorithm and can be used to influence the future course of that algorithm (e.g., through invoking `preventDefault()`). Events must not be used as actions or initiators that cause some algorithm to start running. That is not what they are for.

Note

This is called out here specifically because previous iterations of the DOM had a concept of "default actions" associated with events that gave folks all the wrong ideas. Events do not represent or cause actions, they can only be used to influence an ongoing one.

3. Aborting ongoing activities §

Though promises do not have a built-in aborting mechanism, many APIs using them require abort semantics. `AbortController` is meant to support these requirements by providing an `abort()` method that toggles the state of a corresponding `AbortSignal` object. The API which wishes to support aborting can accept an `AbortSignal` object, and use its state to determine how to proceed.

APIs that rely upon `AbortController` are encouraged to respond to `abort()` by rejecting any unsettled promise with a new "`AbortError`" `DOMException`.

Example

A hypothetical `doAmazingness({ ... })` method could accept an `AbortSignal` object in order to support aborting as follows:

```
const controller = new AbortController();
const signal = controller.signal;

startSpinner();

doAmazingness({ ..., signal })
  .then(result => ...)
  .catch(err => {
    if (err.name == 'AbortError') return;
    showUserErrorMessage();
  })
  .then(() => stopSpinner());

// …

controller.abort();
```

`doAmazingness` could be implemented as follows:

```
function doAmazingness({signal}) {
  if (signal.aborted) {
    return Promise.reject(new DOMException('Aborted', 'AbortError'));
  }

  return new Promise((resolve, reject) => {
    // Begin doing amazingness, and call resolve(result) when done.
    // But also, watch for signals:
    signal.addEventListener('abort', () => {
      // Stop doing amazingness, and:
      reject(new DOMException('Aborted', 'AbortError'));
    });
  });
}
```

APIs that require more granular control could extend both `AbortController` and `AbortSignal` objects according to their needs.

3.1. Interface `AbortController` §

IDL

```
[Exposed=(Window,Worker)]
interface AbortController {
  constructor();
  [SameObject] readonly attribute AbortSignal signal;

  void abort();
};
```

For web developers (non-normative)

***controller* = new AbortController()**

Returns a new *controller* whose `signal` is set to a newly created `AbortSignal` object.

***controller* . signal**

Returns the `AbortSignal` object associated with this object.

***controller* . abort()**

Invoking this method will set this object's `AbortSignal`'s aborted flag and signal to any observers that the associated activity is to be aborted.

An `AbortController` object has an associated **signal** (an `AbortSignal` object).

The **`AbortController()`** constructor, when invoked, must run these steps:

1. Let *signal* be a new `AbortSignal` object.
2. Let *controller* be a new `AbortController` object whose signal is *signal*.
3. Return *controller*.

The **`signal`** attribute's getter, when invoked, must return this's signal.

The **`abort()`** method, when invoked, must signal abort on this's signal.

3.2. Interface `AbortSignal` §

IDL
```
[Exposed=(Window,Worker)]
interface AbortSignal : EventTarget {
  readonly attribute boolean aborted;

  attribute EventHandler onabort;
};
```

For web developers (non-normative)

***signal* . aborted**

Returns true if this `AbortSignal`'s `AbortController` has signaled to abort, and false otherwise.

An `AbortSignal` object has an associated **aborted flag**. It is unset unless specified otherwise.

An `AbortSignal` object has associated **abort algorithms**, which is a set of algorithms which are to be executed when its aborted flag is set. Unless specified otherwise, its value is the empty set.

To **add** an algorithm *algorithm* to an `AbortSignal` object *signal*, run these steps:

1. If *signal*'s aborted flag is set, then return.
2. Append *algorithm* to *signal*'s abort algorithms.

To **remove** an algorithm *algorithm* from an `AbortSignal` *signal*, remove *algorithm* from *signal*'s abort algorithms.

Note

The abort algorithms enable APIs with complex requirements to react in a reasonable way to `abort()`. For example, a given API's aborted flag might need to be propagated to a cross-thread environment, such as a service worker.

The **`aborted`** attribute's getter, when invoked, must return true if this's aborted flag is set, and false otherwise.

The **`onabort`** attribute is an event handler IDL attribute for the **`onabort`** event handler, whose event handler event type is **`abort`**.

Note

Changes to an `AbortSignal` object represent the wishes of the corresponding `AbortController` object, but an API observing the `AbortSignal` object can chose to ignore them. For instance, if the operation has already completed.

To **signal abort**, given a `AbortSignal` object *signal*, run these steps:

1. If *signal*'s aborted flag is set, then return.

2. Set *signal*'s aborted flag.

3. For each *algorithm* in *signal*'s abort algorithms: run *algorithm*.

4. Empty *signal*'s abort algorithms.

5. Fire an event named `abort` at *signal*.

A *followingSignal* (an `AbortSignal`) is made to **follow** a *parentSignal* (an `AbortSignal`) by running these steps:

1. If *followingSignal*'s aborted flag is set, then return.

2. If *parentSignal*'s aborted flag is set, then signal abort on *followingSignal*.

3. Otherwise, add the following abort steps to *parentSignal*:

    1. Signal abort on *followingSignal*.

3.3. Using `AbortController` and `AbortSignal` objects in APIs §

Any web platform API using promises to represent operations that can be aborted must adhere to the following:

- Accept `AbortSignal` objects through a `signal` dictionary member.
- Convey that the operation got aborted by rejecting the promise with an "`AbortError`" `DOMException`.
- Reject immediately if the `AbortSignal`'s aborted flag is already set, otherwise:
- Use the abort algorithms mechanism to observe changes to the `AbortSignal` object and do so in a manner that does not lead to clashes with other observers.

Example

The steps for a promise-returning method `doAmazingness(options)` could be as follows:

1. Let *p* be a new promise.

2. If *options*' `signal` member is present, then:

    1. If *options*' `signal`'s aborted flag is set, then reject *p* with an "`AbortError`" `DOMException` and return *p*.

    2. Add the following abort steps to *options*' `signal`:

        1. Stop doing amazing things.

        2. Reject *p* with an "`AbortError`" `DOMException`.

3. Run these steps in parallel:
    1. Let *amazingResult* be the result of doing some amazing things.
    2. Resolve *p* with *amazingResult*.
4. Return *p*.

APIs not using promises should still adhere to the above as much as possible.

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.

Each such document is represented as a node tree. Some of the nodes in a 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:

```
└Document
  ├Doctype: html
  └Element: html class="e"
    ├Element: head
    │ └Element: title
    │   └Text: Aliens?
    ├Text: ⏎␣
    └Element: body
      └Text: Why yes.⏎
```

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 §

`Document`, `DocumentType`, `DocumentFragment`, `Element`, `Text`, `ProcessingInstruction`, and `Comment` objects (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 `ProcessingInstruction` or `Comment`.
2. Optionally one `DocumentType` node.
3. Zero or more nodes each of which is `ProcessingInstruction` or `Comment`.
4. Optionally one `Element` node.
5. Zero or more nodes each of which is `ProcessingInstruction` or `Comment`.

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

`DocumentType`
`Text`
`ProcessingInstruction`
`Comment`
None.

To determine the **length** of a node *node*, switch on *node*:

↪ `DocumentType`
Zero.

↪ `Text`
↪ `ProcessingInstruction`
↪ `Comment`
Its data's length.

↪ **Any other node**
Its number of children.

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

4.2.1. Document tree §

A **document tree** is a node tree whose root is a document.

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 be only one such element.

An element is **in a document tree** if its root is a document.

An element is **in a document** if it is in a document tree. Note *The term in a document is no longer supposed to be used. It indicates that the standard using it has not been updated to account for shadow trees.*

4.2.2. Shadow tree §

A **shadow tree** is a node tree whose root is a shadow root.

A shadow root is always attached to another node tree through its host. A shadow tree is therefore never alone. The node tree of a shadow root's host is sometimes referred to as the **light tree**.

Note
A shadow tree's corresponding light tree can be a shadow tree itself.

An element is **connected** if its shadow-including root is a document.

4.2.2.1. Slots §

A shadow tree contains zero or more elements that are **slots**.

Note

A slot can only be created through HTML's `slot` element.

A slot has an associated **name** (a string). Unless stated otherwise it is the empty string.

Use these attribute change steps to update a slot's name:

1. If *element* is a slot, *localName* is `name`, and *namespace* is null, then:
    1. If *value* is *oldValue*, then return.
    2. If *value* is null and *oldValue* is the empty string, then return.
    3. If *value* is the empty string and *oldValue* is null, then return.
    4. If *value* is null or the empty string, then set *element*'s name to the empty string.
    5. Otherwise, set *element*'s name to *value*.
    6. Run assign slottables for a tree with *element*'s root.

Note

The first slot in a shadow tree, in tree order, whose name is the empty string, is sometimes known as the "default slot".

A slot has an associated **assigned nodes** (a list of slottables). Unless stated otherwise it is empty.

4.2.2.2. Slottables §

`Element` and `Text` nodes are **slottables**.

Note

A slot can be a slottable.

A slottable has an associated **name** (a string). Unless stated otherwise it is the empty string.

Use these attribute change steps to update a slottable's name:

1. If *localName* is `slot` and *namespace* is null, then:
    1. If *value* is *oldValue*, then return.
    2. If *value* is null and *oldValue* is the empty string, then return.
    3. If *value* is the empty string and *oldValue* is null, then return.
    4. If *value* is null or the empty string, then set *element*'s name to the empty string.
    5. Otherwise, set *element*'s name to *value*.
    6. If *element* is assigned, then run assign slottables for *element*'s assigned slot.
    7. Run assign a slot for *element*.

A slottable has an associated **assigned slot** (null or a slot). Unless stated otherwise it is null. A slottable is **assigned** if its assigned slot is non-null.

4.2.2.3. Finding slots and slottables §

To **find a slot** for a given slottable *slottable* and an optional *open flag* (unset unless stated otherwise), run these steps:

1. If *slottable*'s parent is null, then return null.
2. Let *shadow* be *slottable*'s parent's shadow root.
3. If *shadow* is null, then return null.
4. If the *open flag* is set and *shadow*'s mode is *not* "`open`", then return null.
5. Return the first slot in tree order in *shadow*'s descendants whose name is *slottable*'s name, if any, and null otherwise.

To **find slottables** for a given slot *slot*, run these steps:

1. Let *result* be an empty list.
2. If *slot*'s root is not a shadow root, then return *result*.
3. Let *host* be *slot*'s root's host.
4. For each slottable child of *host*, *slottable*, in tree order:
    1. Let *foundSlot* be the result of finding a slot given *slottable*.
    2. If *foundSlot* is *slot*, then append *slottable* to *result*.
5. Return *result*.

To **find flattened slottables** for a given slot *slot*, run these steps:

1. Let *result* be an empty list.
2. If *slot*'s root is not a shadow root, then return *result*.
3. Let *slottables* be the result of finding slottables given *slot*.
4. If *slottables* is the empty list, then append each slottable child of *slot*, in tree order, to *slottables*.
5. For each *node* in *slottables*:
    1. If *node* is a slot whose root is a shadow root, then:
        1. Let *temporaryResult* be the result of finding flattened slottables given *node*.
        2. Append each slottable in *temporaryResult*, in order, to *result*.
    2. Otherwise, append *node* to *result*.
6. Return *result*.

4.2.2.4. Assigning slottables and slots §

To **assign slottables** for a slot *slot*, run these steps:

1. Let *slottables* be the result of finding slottables for *slot*.
2. If *slottables* and *slot*'s assigned nodes are not identical, then run signal a slot change for *slot*.
3. Set *slot*'s assigned nodes to *slottables*.
4. For each *slottable* in *slottables*, set *slottable*'s assigned slot to *slot*.

To **assign slottables for a tree**, given a node *root*, run assign slottables for each slot *slot* in *root*'s inclusive descendants, in tree order.

To **assign a slot**, given a slottable *slottable*, run these steps:

1. Let *slot* be the result of finding a slot with *slottable*.
2. If *slot* is non-null, then run assign slottables for *slot*.

4.2.2.5. Signaling slot change §

Each similar-origin window agent has **signal slots** (a set of slots), which is initially empty. [HTML]

To **signal a slot change**, for a slot *slot*, run these steps:

1. Append *slot* to *slot*'s relevant agent's signal slots.
2. Queue a mutation observer microtask.

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

   `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 non-null and a doctype is following *child*.

   element
   : *parent* has an element child, *child* is a doctype, or *child* is non-null and a doctype is following *child*.

   doctype
   : *parent* has a doctype child, *child* is non-null and 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 *referenceChild* be *child*.
3. If *referenceChild* is *node*, then set *referenceChild* to *node*'s next sibling.

4. Insert *node* into *parent* before *referenceChild*.
5. Return *node*.

Specifications may define **insertion steps** for all or some nodes. The algorithm is passed *insertedNode*, as indicated in the insert algorithm below.

Specifications may define **children changed steps** for all or some nodes. The algorithm is passed no argument and is called from insert, remove, and replace data.

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

1. Let *nodes* be *node*'s children, if *node* is a `DocumentFragment` node; otherwise « *node* ».
2. Let *count* be *nodes*'s size.
3. If *count* is 0, then return.
4. If *node* is a `DocumentFragment` node, then:
    1. Remove its children with the *suppress observers flag* set.
    2. Queue a tree mutation record for *node* with « », *nodes*, null, and null.

       Note
       This step intentionally does not pay attention to the suppress observers flag.
5. If *child* is non-null, then:
    1. For each live range whose start node is *parent* and start offset is greater than *child*'s index, increase its start offset by *count*.
    2. For each live range whose end node is *parent* and end offset is greater than *child*'s index, increase its end offset by *count*.
6. Let *previousSibling* be *child*'s previous sibling or *parent*'s last child if *child* is null.
7. For each *node* in *nodes*, in tree order:
    1. Adopt *node* into *parent*'s node document.
    2. If *child* is null, then append *node* to *parent*'s children.
    3. Otherwise, insert *node* into *parent*'s children before *child*'s index.
    4. If *parent* is a shadow host and *node* is a slottable, then assign a slot for *node*.
    5. If *parent*'s root is a shadow root, and *parent* is a slot whose assigned nodes is the empty list, then run signal a slot change for *parent*.
    6. Run assign slottables for a tree with *node*'s root.
    7. For each shadow-including inclusive descendant *inclusiveDescendant* of *node*, in shadow-including tree order:
        1. Run the insertion steps with *inclusiveDescendant*.
        2. If *inclusiveDescendant* is connected, then:
            1. If *inclusiveDescendant* is custom, then enqueue a custom element callback reaction with *inclusiveDescendant*, callback name "`connectedCallback`", and an empty argument list.
            2. Otherwise, try to upgrade *inclusiveDescendant*.

            Note
            If this successfully upgrades inclusiveDescendant, *its* `connectedCallback` *will be enqueued automatically during the upgrade an element algorithm.*

8. If *suppress observers flag* is unset, then queue a tree mutation record for *parent* with *nodes*, « », *previousSibling*, and *child*.
9. Run the children changed steps for *parent*.

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

   `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 *referenceChild* be *child*'s next sibling.
8. If *referenceChild* is *node*, then set *referenceChild* to *node*'s next sibling.
9. Let *previousSibling* be *child*'s previous sibling.
10. Let *removedNodes* be the empty set.
11. If *child*'s parent is non-null, then:
    1. Set *removedNodes* to « *child* ».
    2. Remove *child* with the *suppress observers flag* set.

    Note

    The above can only be false if child *is* node*.*

12. Let *nodes* be *node*'s children if *node* is a `DocumentFragment` node; otherwise « *node* ».
13. Insert *node* into *parent* before *referenceChild* with the *suppress observers flag* set.
14. Queue a tree mutation record for *parent* with *nodes*, *removedNodes*, *previousSibling*, and *referenceChild*.
15. Return *child*.

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

1. Let *removedNodes* be *parent*’s children.
2. Let *addedNodes* be the empty set.
3. If *node* is a `DocumentFragment` node, then set *addedNodes* to *node*’s children.
4. Otherwise, if *node* is non-null, set *addedNodes* to « *node* ».
5. Remove all *parent*’s children, in tree order, with the *suppress observers flag* set.
6. If *node* is non-null, then insert *node* into *parent* before null with the *suppress observers flag* set.
7. If either *addedNodes* or *removedNodes* is not empty, then queue a tree mutation record for *parent* with *addedNodes*, *removedNodes*, null, and null.

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*, then throw a "`NotFoundError`" `DOMException`.
2. Remove *child*.
3. Return *child*.

Specifications may define **removing steps** for all or some nodes. The algorithm is passed *removedNode*, and optionally *oldParent*, as indicated in the remove algorithm below.

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

1. Let *parent* be *node*’s parent
2. Assert: *parent* is non-null.
3. Let *index* be *node*’s index.
4. For each live range whose start node is an inclusive descendant of *node*, set its start to (*parent*, *index*).
5. For each live range whose end node is an inclusive descendant of *node*, set its end to (*parent*, *index*).
6. For each live range whose start node is *parent* and start offset is greater than *index*, decrease its start offset by 1.
7. For each live range whose end node is *parent* and end offset is greater than *index*, decrease its end offset by 1.
8. For each `NodeIterator` object *iterator* whose root’s node document is *node*’s node document, run the `NodeIterator` pre-removing steps given *node* and *iterator*.
9. Let *oldPreviousSibling* be *node*’s previous sibling.
10. Let *oldNextSibling* be *node*’s next sibling.
11. Remove *node* from its *parent*’s children.
12. If *node* is assigned, then run assign slottables for *node*’s assigned slot.
13. If *parent*’s root is a shadow root, and *parent* is a slot whose assigned nodes is the empty list, then run signal a slot change for *parent*.
14. If *node* has an inclusive descendant that is a slot, then:
    1. Run assign slottables for a tree with *parent*’s root.
    2. Run assign slottables for a tree with *node*.
15. Run the removing steps with *node* and *parent*.

16. Let *isParentConnected* be *parent*'s connected.

17. If *node* is custom and *isParentConnected* is true, then enqueue a custom element callback reaction with *node*, callback name "`disconnectedCallback`", and an empty argument list.

    Note

    It is intentional for now that custom elements do not get parent *passed. This might change in the future if there is a need.*

18. For each shadow-including descendant *descendant* of *node*, in shadow-including tree order, then:

    1. Run the removing steps with *descendant*.

    2. If *descendant* is custom and *isParentConnected* is true, then enqueue a custom element callback reaction with *descendant*, callback name "`disconnectedCallback`", and an empty argument list.

19. For each inclusive ancestor *inclusiveAncestor* of *parent*, and then for each *registered* of *inclusiveAncestor*'s registered observer list, if *registered*'s options's `subtree` is true, then append a new transient registered observer whose observer is *registered*'s observer, options is *registered*'s options, and source is *registered* to *node*'s registered observer list.

20. If *suppress observers flag* is unset, then queue a tree mutation record for *parent* with « », « *node* », *oldPreviousSibling*, and *oldNextSibling*.

21. Run the children changed steps for *parent*.

4.2.4. Mixin `NonElementParentNode` §

Note

Web compatibility prevents the `getElementById()` method from being exposed on elements (and therefore on `ParentNode`).

IDL
```
interface mixin NonElementParentNode {
  Element? getElementById(DOMString elementId);
};
Document includes NonElementParentNode;
DocumentFragment includes NonElementParentNode;
```

For web developers (non-normative)

***node* . `getElementById(`*`elementId`*`)`**

Returns the first element within *node*'s descendants whose ID is *elementId*.

The **`getElementById(`*`elementId`*`)`** method, when invoked, must return the first element, in tree order, within this's descendants, whose ID is *elementId*, and null if there is no such element otherwise.

4.2.5. Mixin `DocumentOrShadowRoot` §

IDL
```
interface mixin DocumentOrShadowRoot {
};
Document includes DocumentOrShadowRoot;
ShadowRoot includes DocumentOrShadowRoot;
```

Note

The `DocumentOrShadowRoot` mixin is expected to be used by other standards that want to define APIs shared between documents and shadow roots.

4.2.6. Mixin ParentNode §

To **convert nodes into a node**, given *nodes* and *document*, run these steps:

1. Let *node* be null.
2. Replace each string in *nodes* with a new Text node whose data is the string and node document is *document*.
3. If *nodes* contains one node, set *node* to that node.
4. Otherwise, set *node* to a new DocumentFragment whose node document is *document*, and then append each node in *nodes*, if any, to it.
5. Return *node*.

IDL
```
interface mixin ParentNode {
  [SameObject] readonly attribute HTMLCollection children;
  readonly attribute Element? firstElementChild;
  readonly attribute Element? lastElementChild;
  readonly attribute unsigned long childElementCount;

  [CEReactions, Unscopable] void prepend((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void append((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void replaceChildren((Node or DOMString)... nodes);

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

For web developers (non-normative)

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 . **prepend**(*nodes*)

Inserts *nodes* before the first child of *node*, while replacing strings in *nodes* with equivalent Text nodes.

Throws a "HierarchyRequestError" DOMException if the constraints of the node tree are violated.

node . **append**(*nodes*)

Inserts *nodes* after the last child of *node*, while replacing strings in *nodes* with equivalent Text nodes.

Throws a "HierarchyRequestError" DOMException if the constraints of the node tree are violated.

node . **replaceChildren**(*nodes*)

Replace all children of *node* with *nodes*, while replacing strings in *nodes* with equivalent Text nodes.

Throws a "HierarchyRequestError" DOMException if the constraints of the node tree are violated.

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's getter must return an HTMLCollection collection rooted at this matching only element children.

The **firstElementChild** attribute's getter must return the first child that is an element, and null otherwise.

The **lastElementChild** attribute's getter must return the last child that is an element, and null otherwise.

The **childElementCount** attribute's getter must return the number of children of this that are elements.

The **prepend(*nodes*)** method, when invoked, must run these steps:

1. Let *node* be the result of converting nodes into a node given *nodes* and this's node document.
2. Pre-insert *node* into this before this's first child.

The **append(*nodes*)** method, when invoked, must run these steps:

1. Let *node* be the result of converting nodes into a node given *nodes* and this's node document.
2. Append *node* to this.

The **replaceChildren(*nodes*)** method, when invoked, must run these steps:

1. Let *node* be the result of converting nodes into a node given *nodes* and this's node document.
2. Ensure pre-insertion validity of *node* into this before null.
3. Replace all with *node* within this.

The **querySelector(*selectors*)** method, when invoked, must return the first result of running scope-match a selectors string *selectors* against this, if the result is not an empty list, and null otherwise.

The **querySelectorAll(*selectors*)** method, when invoked, must return the static result of running scope-match a selectors string *selectors* against this.

4.2.7. Mixin NonDocumentTypeChildNode §

Note

Web compatibility prevents the previousElementSibling and nextElementSibling attributes from being exposed on doctypes (and therefore on ChildNode).

IDL
```
interface mixin NonDocumentTypeChildNode {
  readonly attribute Element? previousElementSibling;
  readonly attribute Element? nextElementSibling;
};
Element includes NonDocumentTypeChildNode;
CharacterData includes NonDocumentTypeChildNode;
```

For web developers (non-normative)

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's getter must return the first preceding sibling that is an element, and null otherwise.

The `nextElementSibling` attribute's getter must return the first following sibling that is an element, and null otherwise.

4.2.8. Mixin `ChildNode` §

IDL
```
interface mixin ChildNode {
  [CEReactions, Unscopable] void before((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void after((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void replaceWith((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void remove();
};
DocumentType includes ChildNode;
Element includes ChildNode;
CharacterData includes ChildNode;
```

For web developers (non-normative)

***node* . `before(...nodes)`**

Inserts *nodes* just before *node*, while replacing strings in *nodes* with equivalent `Text` nodes.

Throws a "`HierarchyRequestError`" `DOMException` if the constraints of the node tree are violated.

***node* . `after(...nodes)`**

Inserts *nodes* just after *node*, while replacing strings in *nodes* with equivalent `Text` nodes.

Throws a "`HierarchyRequestError`" `DOMException` if the constraints of the node tree are violated.

***node* . `replaceWith(...nodes)`**

Replaces *node* with *nodes*, while replacing strings in *nodes* with equivalent `Text` nodes.

Throws a "`HierarchyRequestError`" `DOMException` if the constraints of the node tree are violated.

***node* . `remove()`**

Removes *node*.

The `before(`*`nodes`*`)` method, when invoked, must run these steps:

1. Let *parent* be this's parent.
2. If *parent* is null, then return.
3. Let *viablePreviousSibling* be this's first preceding sibling not in *nodes*, and null otherwise.
4. Let *node* be the result of converting nodes into a node, given *nodes* and this's node document.
5. If *viablePreviousSibling* is null, set it to *parent*'s first child, and to *viablePreviousSibling*'s next sibling otherwise.
6. Pre-insert *node* into *parent* before *viablePreviousSibling*.

The `after(`*`nodes`*`)` method, when invoked, must run these steps:

1. Let *parent* be this's parent.
2. If *parent* is null, then return.
3. Let *viableNextSibling* be this's first following sibling not in *nodes*, and null otherwise.
4. Let *node* be the result of converting nodes into a node, given *nodes* and this's node document.
5. Pre-insert *node* into *parent* before *viableNextSibling*.

The **replaceWith(*nodes*)** method, when invoked, must run these steps:

1. Let *parent* be this's parent.
2. If *parent* is null, then return.
3. Let *viableNextSibling* be this's first following sibling not in *nodes*, and null otherwise.
4. Let *node* be the result of converting nodes into a node, given *nodes* and this's node document.
5. If this's parent is *parent*, replace this with *node* within *parent*.

   Note

   This could have been inserted into node.

6. Otherwise, pre-insert *node* into *parent* before *viableNextSibling*.

The **remove()** method, when invoked, must run these steps:

1. If this's parent is null, then return.
2. Remove this.

4.2.9. Mixin `Slottable` §

IDL
```
interface mixin Slottable {
  readonly attribute HTMLSlotElement? assignedSlot;
};
Element includes Slottable;
Text includes Slottable;
```

The **assignedSlot** attribute's getter must return the result of find a slot given this and with the *open flag* set.

4.2.10. Old-style collections: `NodeList` and `HTMLCollection` §

A **collection** is an object that represents a list of 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.

4.2.10.1. 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>;
};
```

For web developers (non-normative)

***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.

4.2.10.2. Interface HTMLCollection §

IDL
```
[Exposed=Window, LegacyUnenumerableNamedProperties]
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

`HTMLCollection` *is a historical artifact we cannot rid the web of. While developers are of course welcome to keep using it, new API standard designers ought not to use it (use* `sequence<T>` *in IDL instead).*

For web developers (non-normative)

***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 elements represented by the collection. If there are no such elements, then there are no supported property indices.

The `length` attribute's getter must return the number of nodes represented by the collection.

The `item(*index*)` method, when invoked, 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 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:
    1. If *element* has an ID which is not 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, when invoked, 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 is in the HTML namespace and has a `name` attribute whose value is *key*;

   or null if there is no such element.

4.3. Mutation observers §

Each similar-origin window agent has a **mutation observer microtask queued** (a boolean), which is initially false. [HTML]

Each similar-origin window agent also has **mutation observers** (a set of zero or more `MutationObserver` objects), which is initially empty.

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

1. If the surrounding agent's mutation observer microtask queued is true, then return.
2. Set the surrounding agent's mutation observer microtask queued to true.
3. Queue a microtask to notify mutation observers.

To **notify mutation observers**, run these steps:

1. Set the surrounding agent's mutation observer microtask queued to false.
2. Let *notifySet* be a clone of the surrounding agent's mutation observers.
3. Let *signalSet* be a clone of the surrounding agent's signal slots.
4. Empty the surrounding agent's signal slots.
5. For each *mo* of *notifySet*:
    1. Let *records* be a clone of *mo*'s record queue.
    2. Empty *mo*'s record queue.
    3. For each *node* of *mo*'s node list, remove all transient registered observers whose observer is *mo* from *node*'s registered observer list.
    4. If *records* is not empty, then invoke *mo*'s callback with « *records*, *mo* », and *mo*. If this throws an exception, then report the exception.
6. For each *slot* of *signalSet*, fire an event named `slotchange`, with its `bubbles` attribute set to true, at *slot*.

Each node has a **registered observer list** (a list of zero or more registered observers), which is initially empty.

A **registered observer** consists of an **observer** (a `MutationObserver` object) and **options** (a `MutationObserverInit` dictionary).

A **transient registered observer** is a registered observer that also consists of a **source** (a registered observer).

Note

Transient registered observers are used to track mutations within a given node's descendants after node has been removed so they do not get lost when `subtree` *is set to true on node's parent.*

4.3.1. Interface `MutationObserver` §

IDL
```
[Exposed=Window]
interface MutationObserver {
  constructor(MutationCallback callback);

  void observe(Node target, optional 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 **node list** (a list of nodes), which is initially empty.
- A **record queue** (a queue of zero or more `MutationRecord` objects), which is initially empty.

For web developers (non-normative)

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` 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, when invoked, must run these steps:

1. Let *mo* be a new `MutationObserver` object whose callback is *callback*.
2. Append *mo* to *mo*'s relevant agent's mutation observers.
3. Return *mo*.

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, then set *options*'s `attributes` to true.
2. If *options*'s `characterDataOldValue` is present and *options*'s `characterData` is omitted, then set *options*'s `characterData` to true.
3. If none of *options*'s `childList`, `attributes`, and `characterData` is true, then throw a `TypeError`.
4. If *options*'s `attributeOldValue` is true and *options*'s `attributes` is false, then throw a `TypeError`.
5. If *options*'s `attributeFilter` is present and *options*'s `attributes` is false, then throw a `TypeError`.
6. If *options*'s `characterDataOldValue` is true and *options*'s `characterData` is false, then throw a `TypeError`.
7. For each *registered* of *target*'s registered observer list, if *registered*'s observer is this:
    1. For each *node* of this's node list, remove all transient registered observers whose source is *registered* from *node*'s registered observer list.
    2. Set *registered*'s options to *options*.
8. Otherwise:
    1. Append a new registered observer whose observer is this and options is *options* to *target*'s registered observer list.
    2. Append *target* to this's node list.

The `disconnect()` method, when invoked, must run these steps:

1. For each *node* of this's node list, remove any registered observer from *node*'s registered observer list for which this is the observer.
2. Empty this's record queue.

The `takeRecords()` method, when invoked, must run these steps:

1. Let *records* be a clone of this's record queue.
2. Empty this's record queue.
3. Return *records*.

4.3.2. Queuing a mutation record §

To **queue a mutation record** of *type* for *target* with *name*, *namespace*, *oldValue*, *addedNodes*, *removedNodes*, *previousSibling*, and *nextSibling*, run these steps:

1. Let *interestedObservers* be an empty map.
2. Let *nodes* be the inclusive ancestors of *target*.
3. For each *node* in *nodes*, and then for each *registered* of *node*'s registered observer list:
    1. Let *options* be *registered*'s options.
    2. If none of the following are true
        - *node* is not *target* and *options*'s `subtree` is false
        - *type* is "`attributes`" and *options*'s `attributes` is not true
        - *type* is "`attributes`", *options*'s `attributeFilter` is present, and *options*'s `attributeFilter` does not contain *name* or *namespace* is non-null
        - *type* is "`characterData`" and *options*'s `characterData` is not true
        - *type* is "`childList`" and *options*'s `childList` is false

       then:

        1. Let *mo* be *registered*'s observer.
        2. If *interestedObservers*[*mo*] does not exist, then set *interestedObservers*[*mo*] to null.
        3. If either *type* is "`attributes`" and *options*'s `attributeOldValue` is true, or *type* is "`characterData`" and *options*'s `characterDataOldValue` is true, then set *interestedObservers*[*mo*] to *oldValue*.
4. For each *observer* → *mappedOldValue* of *interestedObservers*:
    1. Let *record* be a new `MutationRecord` object with its `type` set to *type*, `target` set to *target*, `attributeName` set to *name*, `attributeNamespace` set to *namespace*, `oldValue` set to *mappedOldValue*, `addedNodes` set to *addedNodes*, `removedNodes` set to *removedNodes*, `previousSibling` set to *previousSibling*, and `nextSibling` set to *nextSibling*.
    2. Enqueue *record* to *observer*'s record queue.
5. Queue a mutation observer microtask.

To **queue a tree mutation record** for *target* with *addedNodes*, *removedNodes*, *previousSibling*, and *nextSibling*, run these steps:

1. Assert: either *addedNodes* or *removedNodes* is not empty.
2. Queue a mutation record of "`childList`" for *target* with null, null, null, *addedNodes*, *removedNodes*, *previousSibling*, and *nextSibling*.

4.3.3. Interface MutationRecord §

```
IDL
[Exposed=Window]
interface MutationRecord {
  readonly attribute DOMString type;
  [SameObject] 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;
};
```

For web developers (non-normative)

***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`**, **`target`**, **`addedNodes`**, **`removedNodes`**, **`previousSibling`**, **`nextSibling`**, **`attributeName`**, **`attributeNamespace`**, and **`oldValue`** attributes must return the values they were initialized to.

4.3.4. Garbage collection §

Nodes have a strong reference to registered observers in their registered observer list.

Registered observers in a node's registered observer list 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;
  const unsigned short TEXT_NODE = 3;
  const unsigned short CDATA_SECTION_NODE = 4;
  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 USVString baseURI;

  readonly attribute boolean isConnected;
  readonly attribute Document? ownerDocument;
  Node getRootNode(optional GetRootNodeOptions options = {});
  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;

  [CEReactions] attribute DOMString? nodeValue;
  [CEReactions] attribute DOMString? textContent;
  [CEReactions] void normalize();

  [CEReactions, NewObject] Node cloneNode(optional boolean deep = false);
  boolean isEqualNode(Node? otherNode);
  boolean isSameNode(Node? otherNode); // historical alias of ===

  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);

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

dictionary GetRootNodeOptions {
  boolean composed = false;
};
```

Note

Node is an abstract interface and does not exist as node. It is used by all nodes (Document, DocumentType, DocumentFragment, 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.

A node's get the parent algorithm, given an *event*, returns the node's assigned slot, if node is assigned, and node's parent otherwise.

Note

Each node also has a registered observer list.

For web developers (non-normative)

***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` . `CDATA_SECTION_NODE` (4)
node is a `CDATASection` 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 HTML-uppercased qualified name.

`Attr`
Its qualified name.

`Text`
"`#text`".

`CDATASection`
"`#cdata-section`".

`ProcessingInstruction`
: Its target.

`Comment`
: "`#comment`".

`Document`
: "`#document`".

`DocumentType`
: Its name.

`DocumentFragment`
: "`#document-fragment`".

The **nodeType** attribute's getter, when invoked, must return the first matching statement, switching on this:

↪ `Element`
: `ELEMENT_NODE` (1)

↪ `Attr`
: `ATTRIBUTE_NODE` (2);

↪ `Text`
: `TEXT_NODE` (3);

↪ `CDATASection`
: `CDATA_SECTION_NODE` (4);

↪ `ProcessingInstruction`
: `PROCESSING_INSTRUCTION_NODE` (7);

↪ `Comment`
: `COMMENT_NODE` (8);

↪ `Document`
: `DOCUMENT_NODE` (9);

↪ `DocumentType`
: `DOCUMENT_TYPE_NODE` (10);

↪ `DocumentFragment`
: `DOCUMENT_FRAGMENT_NODE` (11).

The **nodeName** attribute's getter, when invoked, must return the first matching statement, switching on this:

↪ `Element`
: Its HTML-uppercased qualified name.

↪ `Attr`
: Its qualified name.

↪ `Text`
: "`#text`".

↪ `CDATASection`
: "`#cdata-section`".

↪ `ProcessingInstruction`
: Its target.

↪ `Comment`
: "`#comment`".

⮑ `Document`

"`#document`".

⮑ `DocumentType`

Its name.

⮑ `DocumentFragment`

"`#document-fragment`".

For web developers (non-normative)

node . `baseURI`

Returns *node*'s node document's document base URL.

The `baseURI` attribute's getter must return node document's document base URL, serialized.

For web developers (non-normative)

node . `isConnected`

Returns true if *node* is connected and false otherwise.

node . `ownerDocument`

Returns the node document. Returns null for documents.

node . `getRootNode()`

Returns *node*'s root.

node . `getRootNode({ composed:true })`

Returns *node*'s shadow-including root.

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 `isConnected` attribute's getter must return true, if this is connected, and false otherwise.

The `ownerDocument` attribute's getter must return null, if this is a document, and this's node document otherwise.

Note

The node document of a document is that document itself. All nodes have a node document at all times.

The `getRootNode(options)` method, when invoked, must return this's shadow-including root if *options*'s `composed` is true, and this's root otherwise.

The `parentNode` attribute's getter must return this's parent.

Note

An `Attr` node has no parent.

The `parentElement` attribute's getter must return this's parent element.

The `hasChildNodes()` method, when invoked, must return true if this has children, and false otherwise.

The `childNodes` attribute's getter must return a `NodeList` rooted at this matching only children.

The `firstChild` attribute's getter must return this's first child.

The `lastChild` attribute's getter must return this's last child.

The `previousSibling` attribute's getter must return this's previous sibling.

Note

An `Attr` node has no siblings.

The `nextSibling` attribute's getter must return this's next sibling.

The `nodeValue` attribute must return the following, depending on this:

`Attr`
: this's value.

`Text`
`ProcessingInstruction`
`Comment`
: this'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 this:

`Attr`
: Set an existing attribute value with this and new value.

`Text`
`ProcessingInstruction`
`Comment`
: Replace data with node this, offset 0, count this's length, and data new value.

Any other node
: Do nothing.

The **`textContent`** attribute's getter must return the following, switching on this:

`DocumentFragment`
`Element`
The descendant text content of this.

`Attr`
this's value.

`Text`
`ProcessingInstruction`
`Comment`
this's data.

Any other node
Null.

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

1. Let *node* be null.
2. If *string* is not the empty string, then set *node* to a new `Text` node whose data is *string* and node document is *parent*'s node document.
3. Replace all with *node* within *parent*.

The `textContent` attribute's setter must, if the given value is null, act as if it was the empty string instead, and then do as described below, switching on this:

`DocumentFragment`
`Element`
String replace all with the given value within this.

`Attr`
Set an existing attribute value with this and new value.

`Text`
`ProcessingInstruction`
`Comment`
Replace data with node this, offset 0, count this's length, and data the given value.

Any other node
Do nothing.

For web developers (non-normative)

***node* . `normalize()`**
Removes empty exclusive `Text` nodes and concatenates the data of remaining contiguous exclusive `Text` nodes into the first of their nodes.

The **`normalize()`** method, when invoked, must run these steps for each descendant exclusive `Text` node *node* of this:

1. Let *length* be *node*'s length.
2. If *length* is zero, then remove *node* and continue with the next exclusive `Text` node, if any.
3. Let *data* be the concatenation of the data of *node*'s contiguous exclusive `Text` nodes (excluding itself), in tree order.

4. Replace data with node *node*, offset *length*, count 0, and data *data*.
5. Let *currentNode* be *node*'s next sibling.
6. While *currentNode* is an exclusive `Text` node:
    1. For each live range whose start node is *currentNode*, add *length* to its start offset and set its start node to *node*.
    2. For each live range whose end node is *currentNode*, add *length* to its end offset and set its end node to *node*.
    3. For each live range whose start node is *currentNode*'s parent and start offset is *currentNode*'s index, set its start node to *node* and its start offset to *length*.
    4. For each live range whose end node is *currentNode*'s parent and end offset is *currentNode*'s index, set its end node to *node* and its end offset to *length*.
    5. Add *currentNode*'s length to *length*.
    6. Set *currentNode* to its next sibling.
7. Remove *node*'s contiguous exclusive `Text` nodes (excluding itself), in tree order.

For web developers (non-normative)

***node* . `cloneNode([deep = false])`**

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

***node* . `isEqualNode(otherNode)`**

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

Specifications may define **cloning steps** for all or some nodes. The algorithm is passed *copy*, *node*, *document*, and an optional *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*, with an optional *document* and *clone children flag*, run these steps:

1. If *document* is not given, let *document* be *node*'s node document.
2. If *node* is an element, then:
    1. Let *copy* be the result of creating an element, given *document*, *node*'s local name, *node*'s namespace, *node*'s namespace prefix, and *node*'s `is` value, with the *synchronous custom elements flag* unset.
    2. For each *attribute* in *node*'s attribute list:
        1. Let *copyAttribute* be a clone of *attribute*.
        2. Append *copyAttribute* to *copy*.
3. Otherwise, let *copy* be a node that implements the same interfaces as *node*, and fulfills these additional requirements, switching on *node*:

    `Document`

    Set *copy*'s encoding, content type, URL, origin, type, and mode, to those of *node*.

    `DocumentType`

    Set *copy*'s name, public ID, and system ID, to those of *node*.

↪ `Attr`

  Set *copy*'s namespace, namespace prefix, local name, and value, to those of *node*.

↪ `Text`
↪ `Comment`

  Set *copy*'s data, to that of *node*.

↪ `ProcessingInstruction`

  Set *copy*'s target and data to those of *node*.

↪ **Any other node**

  —

4. Set *copy*'s node document and *document* to *copy*, if *copy* is a document, and set *copy*'s node document to *document* otherwise.

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, when invoked, must run these steps:

1. If this is a shadow root, then throw a "`NotSupportedError`" `DOMException`.

2. Return a clone of this, 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 attribute list's size.

  ↪ `Attr`

    Its namespace, local name, and value.

  ↪ `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 that equals an attribute 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(*otherNode*)`** method, when invoked, must return true if *otherNode* is non-null and this equals *otherNode*, and false otherwise.

The **`isSameNode(*otherNode*)`** method, when invoked, must return true if *otherNode* is this, and false otherwise.

For web developers (non-normative)

***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, when invoked, must run these steps:

1. If this is *other*, then return zero.

2. Let *node1* be *other* and *node2* be this.

3. Let *attr1* and *attr2* be null.

4. If *node1* is an attribute, then set *attr1* to *node1* and *node1* to *attr1*'s element.

5. If *node2* is an attribute, then:

    1. Set *attr2* to *node2* and *node2* to *attr2*'s element.

    2. If *attr1* and *node1* are non-null, and *node2* is *node1*, then:

        1. For each *attr* in *node2*'s attribute list:

            1. If *attr* equals *attr1*, then return the result of adding `DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC` and `DOCUMENT_POSITION_PRECEDING`.

            2. If *attr* equals *attr2*, then return the result of adding `DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC` and `DOCUMENT_POSITION_FOLLOWING`.

6. If *node1* or *node2* is null, or *node1*'s root is not *node2*'s root, then 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 a cached `Math.random()` value can be used.

7. If *node1* is an ancestor of *node2* and *attr1* is null, or *node1* is *node2* and *attr2* is non-null, then return the result of adding `DOCUMENT_POSITION_CONTAINS` to `DOCUMENT_POSITION_PRECEDING`.

8. If *node1* is a descendant of *node2* and *attr2* is null, or *node1* is *node2* and *attr1* is non-null, then return the result of adding `DOCUMENT_POSITION_CONTAINED_BY` to `DOCUMENT_POSITION_FOLLOWING`.

9. If *node1* is preceding *node2*, then return `DOCUMENT_POSITION_PRECEDING`.

   Note

   Due to the way attributes are handled in this algorithm this results in a node's attributes counting as preceding that node's children, despite attributes not participating in a tree.

10. Return `DOCUMENT_POSITION_FOLLOWING`.

The **`contains(other)`** method, when invoked, must return true if *other* is an inclusive descendant of this, 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 non-null, then 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, then return the result of running locate a namespace prefix on that element using *namespace*.
4. Return null.

To **locate a namespace** for a *node* using *prefix*, switch on *node*:

`Element`

1. If its namespace is non-null and its namespace prefix is *prefix*, then 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`", then return its value if it is not the empty string, and null otherwise.
3. If its parent element is null, then 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, then return null.
2. Return the result of running locate a namespace on its document element using *prefix*.

`DocumentType`
`DocumentFragment`

Return null.

`Attr`

1. If its element is null, then return null.
2. Return the result of running locate a namespace on its element using *prefix*.

Any other node

1. If its parent element is null, then return null.
2. Return the result of running locate a namespace on its parent element using *prefix*.

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

1. If *namespace* is null or the empty string, then return null.
2. Switch on this:

   Element
   : Return the result of locating a namespace prefix for it using *namespace*.

   Document
   : Return the result of locating a namespace prefix for its document element, if its document element is non-null, and null otherwise.

   DocumentType
   DocumentFragment
   : Return null.

   Attr
   : Return the result of locating a namespace prefix for its element, if its element is non-null, and null otherwise.

   Any other node
   : Return the result of locating a namespace prefix for its parent element, if its parent element is non-null, and null otherwise.

The **lookupNamespaceURI(*prefix*)** method, when invoked, must run these steps:

1. If *prefix* is the empty string, then set it to null.
2. Return the result of running locate a namespace for this using *prefix*.

The **isDefaultNamespace(*namespace*)** method, when invoked, must run these steps:

1. If *namespace* is the empty string, then set it to null.
2. Let *defaultNamespace* be the result of running locate a namespace for this using null.
3. Return true if *defaultNamespace* is the same as *namespace*, and false otherwise.

The **insertBefore(*node*, *child*)** method, when invoked, must return the result of pre-inserting *node* into this before *child*.

The **appendChild(*node*)** method, when invoked, must return the result of appending *node* to this.

The **replaceChild(*node*, *child*)** method, when invoked, must return the result of replacing *child* with *node* within this.

The **removeChild(*child*)** method, when invoked, must return the result of pre-removing *child* from this.

The **list of elements with qualified name *qualifiedName*** for a node *root* is the HTMLCollection returned by the following algorithm:

1. If *qualifiedName* is "*" (U+002A), return a HTMLCollection rooted at *root*, whose filter matches only descendant 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 qualified name is *qualifiedName*, in ASCII lowercase.
   - Whose namespace is *not* the HTML namespace and whose qualified name is *qualifiedName*.

3. Otherwise, return a `HTMLCollection` rooted at *root*, whose filter matches descendant elements whose qualified name is *qualifiedName*.

When invoked with the same argument, and as long as *root*'s node document's type has not changed, 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's mode is "`quirks`", and in an identical to 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
```
[Exposed=Window]
interface Document : Node {
  constructor();

  [SameObject] readonly attribute DOMImplementation implementation;
  readonly attribute USVString URL;
  readonly attribute USVString documentURI;
  readonly attribute DOMString compatMode;
  readonly attribute DOMString characterSet;
  readonly attribute DOMString charset; // historical alias of .characterSet
  readonly attribute DOMString inputEncoding; // historical alias of .characterSet
  readonly attribute DOMString contentType;

  readonly attribute DocumentType? doctype;
  readonly attribute Element? documentElement;
  HTMLCollection getElementsByTagName(DOMString qualifiedName);
  HTMLCollection getElementsByTagNameNS(DOMString? namespace, DOMString localName);
  HTMLCollection getElementsByClassName(DOMString classNames);
  [CEReactions, NewObject] Element createElement(DOMString localName, optional (DOMString or
ElementCreationOptions) options = {});
  [CEReactions, NewObject] Element createElementNS(DOMString? namespace, DOMString qualifiedName,
optional (DOMString or ElementCreationOptions) options = {});
  [NewObject] DocumentFragment createDocumentFragment();
  [NewObject] Text createTextNode(DOMString data);
  [NewObject] CDATASection createCDATASection(DOMString data);
  [NewObject] Comment createComment(DOMString data);
  [NewObject] ProcessingInstruction createProcessingInstruction(DOMString target, DOMString data);

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

  [NewObject] Attr createAttribute(DOMString localName);
  [NewObject] Attr createAttributeNS(DOMString? namespace, DOMString qualifiedName);

  [NewObject] Event createEvent(DOMString interface); // historical

  [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 {};

dictionary ElementCreationOptions {
  DOMString is;
};
```

`Document` nodes are simply known as **documents**.

Each document has an associated **encoding** (an encoding), **content type** (a string), **URL** (a URL), **origin** (an origin), **type** ("`xml`" or "`html`"), and **mode** ("`no-quirks`", "`quirks`", or "`limited-quirks`"). [ENCODING] [URL] [HTML]

Unless stated otherwise, a document's encoding is the utf-8 encoding, content type is "`application/xml`", URL is "`about:blank`", origin is an opaque origin, type is "`xml`", and its mode is "`no-quirks`".

A document is said to be an **XML document** if its type is "`xml`", and an **HTML document** otherwise. Whether a document is an HTML document or an XML document affects the behavior of certain APIs.

A document is said to be in **no-quirks mode** if its mode is "`no-quirks`", **quirks mode** if its mode is "`quirks`", and **limited-quirks mode** if its mode is "`limited-quirks`".

Note

The mode is only ever changed from the default for documents created by the HTML parser based on the presence, absence, or value of the DOCTYPE string, and by a new browsing context (initial "`about:blank`*"). [HTML]*

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.)

A document's get the parent algorithm, given an *event*, returns null if *event*'s `type` attribute value is "`load`" or document does not have a browsing context, and the document's relevant global object otherwise.

For web developers (non-normative)

document `= new Document()`

Returns a new document.

document `. implementation`

Returns *document*'s `DOMImplementation` object.

document `. URL`

document `. documentURI`

Returns *document*'s URL.

document `. compatMode`

Returns the string "`BackCompat`" if *document*'s mode is "`quirks`", and "`CSS1Compat`" otherwise.

document `. characterSet`

Returns *document*'s encoding.

document `. contentType`

Returns *document*'s content type.

The **`Document()`** constructor, when invoked, must return a new document whose origin is the origin of current 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's getter must return the `DOMImplementation` object that is associated with the document.

The **`URL`** attribute's getter and **`documentURI`** attribute's getter must return the URL, serialized.

The **`compatMode`** attribute's getter must return "`BackCompat`" if this's mode is "`quirks`", and "`CSS1Compat`" otherwise.

The **`characterSet`** attribute's getter, **`charset`** attribute's getter, and **`inputEncoding`** attribute's getter, must return this's encoding's name.

The **`contentType`** attribute's getter must return the content type.

For web developers (non-normative)

document **.** `doctype`

Returns the doctype or null if there is none.

document **.** `documentElement`

Returns the document element.

collection* = *document **.** `getElementsByTagName(qualifiedName)`

If *qualifiedName* is "`*`" returns a `HTMLCollection` of all descendant elements.

Otherwise, returns a `HTMLCollection` of all descendant elements whose qualified name is *qualifiedName*. (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(classNames)`**
***collection* = *element* . `getElementsByClassName(classNames)`**

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

The **`doctype`** attribute's getter must return the child of the document that is a doctype, and null otherwise.

The **`documentElement`** attribute's getter must return the document element.

The **`getElementsByTagName(`*`qualifiedName`*`)`** method, when invoked, must return the list of elements with qualified name *qualifiedName* for this.

Note

Thus, in an HTML document, `document.getElementsByTagName("FOO")` *will match* `<FOO>` *elements that are not in the HTML namespace, and* `<foo>` *elements that are in the HTML namespace, but not* `<FOO>` *elements that are in the HTML namespace.*

The **`getElementsByTagNameNS(`*`namespace, localName`*`)`** method, when invoked, must return the list of elements with namespace *namespace* and local name *localName* for this.

The **`getElementsByClassName(`*`classNames`*`)`** method, when invoked, must return the list of elements with class names *classNames* for this.

Example

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.

For web developers (non-normative)

***element* = *document* . `createElement(localName [, options])`**

Returns an element with *localName* as local name (if *document* is an HTML document, *localName* gets lowercased). The element's namespace is the HTML namespace when *document* is an HTML document or *document*'s content type is "`application/xhtml+xml`", and null otherwise.

If *localName* does not match the `Name` production an "`InvalidCharacterError`" `DOMException` will be thrown.

When supplied, *options*'s `is` can be used to create a customized built-in element.

***element* = *document* . `createElementNS(namespace, qualifiedName [, options])`**

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`" `DOMException` will be thrown.

If one of the following conditions is true a "`NamespaceError`" `DOMException` 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`".

When supplied, *options*'s `is` can be used to create a customized built-in element.

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

Returns a `DocumentFragment` node.

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

Returns a `Text` node whose data is *data*.

***text* = *document* . `createCDATASection(data)`**

Returns a `CDATASection` 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`" `DOMException` will be thrown. If *data* contains "`?>`" an "`InvalidCharacterError`" `DOMException` 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, options)`** method, when invoked, must run these steps:

1. If *localName* does not match the `Name` production, then throw an "`InvalidCharacterError`" `DOMException`.
2. If this is an HTML document, then set *localName* to *localName* in ASCII lowercase.
3. Let *is* be null.
4. If *options* is a dictionary and *options*'s `is` is present, then set *is* to it.
5. Let *namespace* be the HTML namespace, if this is an HTML document or this's content type is "`application/xhtml+xml`", and null otherwise.
6. Return the result of creating an element given this, *localName*, *namespace*, null, *is*, and with the *synchronous custom elements* flag set.

The **internal `createElementNS` steps**, given *document*, *namespace*, *qualifiedName*, and *options*, are as follows:

1. Let *namespace*, *prefix*, and *localName* be the result of passing *namespace* and *qualifiedName* to validate and extract.
2. Let *is* be null.

3. If *options* is a dictionary and *options*'s `is` is present, then set *is* to it.
4. Return the result of creating an element given *document*, *localName*, *namespace*, *prefix*, *is*, and with the *synchronous custom elements* flag set.

The **`createElementNS(namespace, qualifiedName, options)`** method, when invoked, must return the result of running the internal `createElementNS` steps, given this, *namespace*, *qualifiedName*, and *options*.

Note

`createElement()` and `createElementNS()`'s options *parameter is allowed to be a string for web compatibility.*

The **`createDocumentFragment()`** method, when invoked, must return a new `DocumentFragment` node with its node document set to this.

The **`createTextNode(data)`** method, when invoked, must return a new `Text` node with its data set to *data* and node document set to this.

Note

No check is performed that data *consists of characters that match the `Char` production.*

The **`createCDATASection(data)`** method, when invoked, must run these steps:

1. If this is an HTML document, then throw a "`NotSupportedError`" `DOMException`.
2. If *data* contains the string "`]]>`", then throw an "`InvalidCharacterError`" `DOMException`.
3. Return a new `CDATASection` node with its data set to *data* and node document set to this.

The **`createComment(data)`** method, when invoked, must return a new `Comment` node with its data set to *data* and node document set to this.

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, when invoked, must run these steps:

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

Note

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

For web developers (non-normative)

***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 or a shadow root, throws a "`NotSupportedError`" `DOMException`.

***node* = *document* . `adoptNode(node)`**

Moves *node* from another document and returns it.

If *node* is a document, throws a "`NotSupportedError`" `DOMException` or, if *node* is a shadow root, throws a "`HierarchyRequestError`" `DOMException`.

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

1. If *node* is a document or shadow root, then throw a "`NotSupportedError`" `DOMException`.
2. Return a clone of *node*, with this 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 non-null, then remove *node*.
3. If *document* is not *oldDocument*, then:
    1. For each *inclusiveDescendant* in *node*'s shadow-including inclusive descendants:
        1. Set *inclusiveDescendant*'s node document to *document*.
        2. If *inclusiveDescendant* is an element, then set the node document of each attribute in *inclusiveDescendant*'s attribute list to *document*.
    2. For each *inclusiveDescendant* in *node*'s shadow-including inclusive descendants that is custom, enqueue a custom element callback reaction with *inclusiveDescendant*, callback name "`adoptedCallback`", and an argument list containing *oldDocument* and *document*.
    3. For each *inclusiveDescendant* in *node*'s shadow-including inclusive descendants, in shadow-including tree order, run the adopting steps with *inclusiveDescendant* and *oldDocument*.

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

1. If *node* is a document, then throw a "`NotSupportedError`" `DOMException`.
2. If *node* is a shadow root, then throw a "`HierarchyRequestError`" `DOMException`.
3. If *node* is a `DocumentFragment` node whose host is non-null, then return.
4. Adopt *node* into this.
5. Return *node*.

The `createAttribute(localName)` method, when invoked, must run these steps:

1. If *localName* does not match the `Name` production in XML, then throw an "`InvalidCharacterError`" `DOMException`.
2. If this is an HTML document, then set *localName* to *localName* in ASCII lowercase.
3. Return a new attribute whose local name is *localName* and node document is this.

The `createAttributeNS(namespace, qualifiedName)` method, when invoked, must run these steps:

1. Let *namespace*, *prefix*, and *localName* be the result of passing *namespace* and *qualifiedName* to validate and extract.
2. Return a new attribute whose namespace is *namespace*, namespace prefix is *prefix*, local name is *localName*, and node document is this.

The `createEvent(interface)` method, when invoked, 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, then set *constructor* to the interface in the second column on the same row as the matching string:

String	Interface	Notes
"beforeunloadevent"	BeforeUnloadEvent	[HTML]
"compositionevent"	CompositionEvent	[UIEVENTS]
"customevent"	CustomEvent	
"devicemotionevent"	DeviceMotionEvent	[DEVICE-ORIENTATION]
"deviceorientationevent"	DeviceOrientationEvent	
"dragevent"	DragEvent	[HTML]
"event"	Event	
"events"		
"focusevent"	FocusEvent	[UIEVENTS]
"hashchangeevent"	HashChangeEvent	[HTML]
"htmlevents"	Event	
"keyboardevent"	KeyboardEvent	[UIEVENTS]
"messageevent"	MessageEvent	[HTML]
"mouseevent"	MouseEvent	[UIEVENTS]
"mouseevents"		
"storageevent"	StorageEvent	[HTML]
"svgevents"	Event	
"textevent"	CompositionEvent	[UIEVENTS]
"touchevent"	TouchEvent	[TOUCH-EVENTS]
"uievent"	UIEvent	[UIEVENTS]
"uievents"		

3. If *constructor* is null, then throw a "`NotSupportedError`" `DOMException`.
4. If the interface indicated by *constructor* is not exposed on the relevant global object of this, then throw a "`NotSupportedError`" `DOMException`.

   Note

   Typically user agents disable support for touch events in some configurations, in which case this clause would be triggered for the interface `TouchEvent`.

5. Let *event* be the result of creating an event given *constructor*.
6. Initialize *event*'s `type` attribute to the empty string.
7. Initialize *event*'s `timeStamp` attribute to a `DOMHighResTimeStamp` representing the high resolution time from the time origin to now.
8. Initialize *event*'s `isTrusted` attribute to false.
9. Unset *event*'s initialized flag.
10. Return *event*.

Note

Event constructors ought to be used instead.

The `createRange()` method, when invoked, must return a new live range with (this, 0) as its start an end.

Note

The `Range()` constructor can be used instead.

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

1. Let *iterator* be a new `NodeIterator` object.
2. Set *iterator*'s root and *iterator*'s reference to *root*.
3. Set *iterator*'s pointer before reference to true.
4. Set *iterator*'s whatToShow to *whatToShow*.
5. Set *iterator*'s filter to *filter*.
6. Return *iterator*.

The `createTreeWalker(root, whatToShow, filter)` method, when invoked, must run these steps:

1. Let *walker* be a new `TreeWalker` object.
2. Set *walker*'s root and *walker*'s current to *root*.
3. Set *walker*'s whatToShow to *whatToShow*.
4. Set *walker*'s filter to *filter*.
5. Return *walker*.

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, [LegacyNullToEmptyString] DOMString
qualifiedName, optional DocumentType? doctype = null);
  [NewObject] Document createHTMLDocument(optional DOMString title);

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

For web developers (non-normative)

***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`" `DOMException` is thrown, and if it does not match the `QName` production, a "`NamespaceError`" `DOMException` is thrown.

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

Returns an `XMLDocument`, 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 *namespace* and *qualifiedName*.

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, when invoked, must run these steps:

1. Validate *qualifiedName*.
2. 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 this.

Note

No check is performed that publicId *code points match the* `PubidChar` *production or that* systemId *does not contain both a* '"' *and a* "'".

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

1. Let *document* be a new `XMLDocument`.
2. Let *element* be null.
3. If *qualifiedName* is not the empty string, then set *element* to the result of running the internal `createElementNS` steps, given *document*, *namespace*, *qualifiedName*, and an empty dictionary.
4. If *doctype* is non-null, append *doctype* to *document*.
5. If *element* is non-null, append *element* to *document*.
6. *document*'s origin is this's associated document's origin.
7. *document*'s content type is determined by *namespace*:

   HTML namespace

   `application/xhtml+xml`

   SVG namespace

   `image/svg+xml`

   Any other namespace

   `application/xml`

8. Return *document*.

The **`createHTMLDocument(`*title*`)`** method, when invoked, 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. Append a new doctype, with "`html`" as its name and with its node document set to *doc*, to *doc*.
4. Append the result of creating an element given *doc*, `html`, and the HTML namespace, to *doc*.
5. Append the result of creating an element given *doc*, `head`, and the HTML namespace, to the `html` element created earlier.
6. If *title* is given:
   1. Append the result of creating an element given *doc*, `title`, and the HTML namespace, to the `head` element created earlier.

    2. Append a new `Text` node, with its data set to *title* (which could be the empty string) and its node document set to *doc*, to the `title` element created earlier.

7. Append the result of creating an element given *doc*, `body`, and the HTML namespace, to the `html` element created earlier.

8. *doc*'s origin is this's associated document's origin.

9. Return *doc*.

The **`hasFeature()`** method, when invoked, 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 is no longer to be used, but continues to exist (and simply returns true) so that old pages don't stop working.

4.6. 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 **doctypes**.

Doctypes 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's getter must return this's name.

The **`publicId`** attribute's getter must return this's public ID.

The **`systemId`** attribute's getter must return this's system ID.

4.7. Interface `DocumentFragment` §

IDL
```
[Exposed=Window]
interface DocumentFragment : Node {
  constructor();
};
```

A `DocumentFragment` node has an associated **host** (null or an element in a different node tree). It is null unless otherwise stated.

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 a non-null 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 for shadow roots, and impacts the pre-insert and replace algorithms.

For web developers (non-normative)

tree = new DocumentFragment()

Returns a new DocumentFragment node.

The **DocumentFragment()** constructor, when invoked, must return a new DocumentFragment node whose node document is current global object's associated Document.

4.8. Interface ShadowRoot §

IDL
```
[Exposed=Window]
interface ShadowRoot : DocumentFragment {
  readonly attribute ShadowRootMode mode;
  readonly attribute Element host;
  attribute EventHandler onslotchange;
};

enum ShadowRootMode { "open", "closed" };
```

ShadowRoot nodes are simply known as **shadow roots**.

Shadow roots have an associated **mode** ("open" or "closed").

Shadow roots have an associated **delegates focus**. It is initially set to false.

Shadow roots's associated host is never null.

A shadow root's get the parent algorithm, given an *event*, returns null if *event*'s composed flag is unset and shadow root is the root of *event*'s path's first struct's invocation target, and shadow root's host otherwise.

The **mode** attribute's getter must return this's mode.

The **host** attribute's getter must return this's host.

The **onslotchange** attribute is an event handler IDL attribute for the **onslotchange** event handler, whose event handler event type is slotchange.

In **shadow-including tree order** is shadow-including preorder, depth-first traversal of a node tree. **Shadow-including preorder, depth-first traversal** of a node tree *tree* is preorder, depth-first traversal of *tree*, with for each shadow host encountered in *tree*, shadow-including preorder, depth-first traversal of that element's shadow root's node tree just after it is encountered.

The **shadow-including root** of an object is its root's host's shadow-including root, if the object's root is a shadow root, and its root otherwise.

An object *A* is a **shadow-including descendant** of an object *B*, if *A* is a descendant of *B*, or *A*'s root is a shadow root and *A*'s root's host is a shadow-including inclusive descendant of *B*.

A **shadow-including inclusive descendant** is an object or one of its shadow-including descendants.

An object *A* is a **shadow-including ancestor** of an object *B*, if and only if *B* is a shadow-including descendant of *A*.

A **shadow-including inclusive ancestor** is an object or one of its shadow-including ancestors.

A node *A* is **closed-shadow-hidden** from a node *B* if all of the following conditions are true:

- *A*'s root is a shadow root.
- *A*'s root is not a shadow-including inclusive ancestor of *B*.
- *A*'s root is a shadow root whose mode is "`closed`" or *A*'s root's host is closed-shadow-hidden from *B*.

To **retarget** an object *A* against an object *B*, repeat these steps until they return an object:

1. If one of the following is true
   - *A* is not a node
   - *A*'s root is not a shadow root
   - *B* is a node and *A*'s root is a shadow-including inclusive ancestor of *B*

   then return *A*.
2. Set *A* to *A*'s root's host.

Note

The retargeting algorithm is used by event dispatch as well as other specifications, such as Fullscreen. *[FULLSCREEN]*

4.9. Interface Element §

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

  [CEReactions] attribute DOMString id;
  [CEReactions] attribute DOMString className;
  [SameObject, PutForwards=value] readonly attribute DOMTokenList classList;
  [CEReactions, Unscopable] attribute DOMString slot;

  boolean hasAttributes();
  [SameObject] readonly attribute NamedNodeMap attributes;
  sequence<DOMString> getAttributeNames();
  DOMString? getAttribute(DOMString qualifiedName);
  DOMString? getAttributeNS(DOMString? namespace, DOMString localName);
  [CEReactions] void setAttribute(DOMString qualifiedName, DOMString value);
  [CEReactions] void setAttributeNS(DOMString? namespace, DOMString qualifiedName, DOMString value);
  [CEReactions] void removeAttribute(DOMString qualifiedName);
  [CEReactions] void removeAttributeNS(DOMString? namespace, DOMString localName);
  [CEReactions] boolean toggleAttribute(DOMString qualifiedName, optional boolean force);
  boolean hasAttribute(DOMString qualifiedName);
  boolean hasAttributeNS(DOMString? namespace, DOMString localName);

  Attr? getAttributeNode(DOMString qualifiedName);
  Attr? getAttributeNodeNS(DOMString? namespace, DOMString localName);
  [CEReactions] Attr? setAttributeNode(Attr attr);
  [CEReactions] Attr? setAttributeNodeNS(Attr attr);
  [CEReactions] Attr removeAttributeNode(Attr attr);

  ShadowRoot attachShadow(ShadowRootInit init);
  readonly attribute ShadowRoot? shadowRoot;

  Element? closest(DOMString selectors);
  boolean matches(DOMString selectors);
  boolean webkitMatchesSelector(DOMString selectors); // historical alias of .matches

  HTMLCollection getElementsByTagName(DOMString qualifiedName);
  HTMLCollection getElementsByTagNameNS(DOMString? namespace, DOMString localName);
  HTMLCollection getElementsByClassName(DOMString classNames);

  [CEReactions] Element? insertAdjacentElement(DOMString where, Element element); // historical
  void insertAdjacentText(DOMString where, DOMString data); // historical
};

dictionary ShadowRootInit {
  required ShadowRootMode mode;
  boolean delegatesFocus = false;
};
```

`Element` nodes are simply known as **elements**.

Elements have an associated **namespace**, **namespace prefix**, **local name**, **custom element state**, **custom element definition**, **`is` value**. When an element is created, all of these values are initialized.

An element's custom element state is one of "`undefined`", "`failed`", "`uncustomized`", or "`custom`". An element whose custom element state is "`uncustomized`" or "`custom`" is said to be **defined**. An element whose custom element state is "`custom`" is said to be **custom**.

Note

Whether or not an element is defined is used to determine the behavior of the :defined pseudo-class. Whether or not an element is custom is used to determine the behavior of the mutation algorithms. The "`failed`" state is used to ensure that if a custom element constructor fails to execute correctly the first time, it is not executed again by an upgrade.

Example

The following code illustrates elements in each of these four states:

```
<!DOCTYPE html>
<script>
  window.customElements.define("sw-rey", class extends HTMLElement {})
  window.customElements.define("sw-finn", class extends HTMLElement {}, { extends: "p" })
  window.customElements.define("sw-kylo", class extends HTMLElement {
    constructor() {
      // super() intentionally omitted for this example
    }
  })
</script>

<!-- "undefined" (not defined, not custom) -->
<sw-han></sw-han>
<p is="sw-luke"></p>
<p is="asdf"></p>

<!-- "failed" (not defined, not custom) -->
<sw-kylo></sw-kylo>
```

```
<!-- "uncustomized" (defined, not custom) -->
<p></p>
<asdf></asdf>

<!-- "custom" (defined, custom) -->
<sw-rey></sw-rey>
<p is="sw-finn"></p>
```

Elements also have an associated **shadow root** (null or a shadow root). It is null unless otherwise stated. An element is a **shadow host** if its shadow root is non-null.

An element's **qualified name** is its local name if its namespace prefix is null, and its namespace prefix, followed by "`:`", followed by its local name, otherwise.

An element's **HTML-uppercased qualified name** is the return value of these steps:

1. Let *qualifiedName* be this's qualified name.
2. If this is in the HTML namespace and its node document is an HTML document, then set *qualifiedName* to *qualifiedName* in ASCII uppercase.
3. Return *qualifiedName*.

Note

User agents could optimize qualified name and HTML-uppercased qualified name by storing them in internal slots.

To **create an element**, given a *document*, *localName*, *namespace*, and optional *prefix*, *is*, and *synchronous custom elements flag*, run these steps:

1. If *prefix* was not given, let *prefix* be null.
2. If *is* was not given, let *is* be null.
3. Let *result* be null.
4. Let *definition* be the result of looking up a custom element definition given *document*, *namespace*, *localName*, and *is*.
5. If *definition* is non-null, and *definition*'s name is not equal to its local name (i.e., *definition* represents a customized built-in element), then:
    1. Let *interface* be the element interface for *localName* and the HTML namespace.
    2. Set *result* to a new element that implements *interface*, with no attributes, namespace set to the HTML namespace, namespace prefix set to *prefix*, local name set to *localName*, custom element state set to "`undefined`", custom element definition set to null, `is` value set to *is*, and node document set to *document*.
    3. If the *synchronous custom elements flag* is set, then run this step while catching any exceptions:
        1. Upgrade *element* using *definition*.

       If this step threw an exception, then:
        1. Report the exception.
        2. Set *result*'s custom element state to "`failed`".
    4. Otherwise, enqueue a custom element upgrade reaction given *result* and *definition*.
6. Otherwise, if *definition* is non-null, then:
    1. If the *synchronous custom elements flag* is set, then run these steps while catching any exceptions:

1. Let *C* be *definition*’s constructor.
2. Set *result* to the result of constructing *C*, with no arguments.
3. Assert: *result*’s custom element state and custom element definition are initialized.
4. Assert: *result*’s namespace is the HTML namespace.

   Note

   IDL enforces that result *is an* `HTMLElement` *object, which all use the HTML namespace.*

5. If *result*’s attribute list is not empty, then throw a "`NotSupportedError`" `DOMException`.
6. If *result* has children, then throw a "`NotSupportedError`" `DOMException`.
7. If *result*’s parent is not null, then throw a "`NotSupportedError`" `DOMException`.
8. If *result*’s node document is not *document*, then throw a "`NotSupportedError`" `DOMException`.
9. If *result*’s local name is not equal to *localName*, then throw a "`NotSupportedError`" `DOMException`.
10. Set *result*’s namespace prefix to *prefix*.
11. Set *result*’s `is` value to null.

If any of these steps threw an exception, then:

1. Report the exception.
2. Set *result* to a new element that implements the `HTMLUnknownElement` interface, with no attributes, namespace set to the HTML namespace, namespace prefix set to *prefix*, local name set to *localName*, custom element state set to "`failed`", custom element definition set to null, `is` value set to null, and node document set to *document*.

2. Otherwise:
   1. Set *result* to a new element that implements the `HTMLElement` interface, with no attributes, namespace set to the HTML namespace, namespace prefix set to *prefix*, local name set to *localName*, custom element state set to "`undefined`", custom element definition set to null, `is` value set to null, and node document set to *document*.
   2. Enqueue a custom element upgrade reaction given *result* and *definition*.

7. Otherwise:
   1. Let *interface* be the element interface for *localName* and *namespace*.
   2. Set *result* to a new element that implements *interface*, with no attributes, namespace set to *namespace*, namespace prefix set to *prefix*, local name set to *localName*, custom element state set to "`uncustomized`", custom element definition set to null, `is` value set to *is*, and node document set to *document*.
   3. If *namespace* is the HTML namespace, and either *localName* is a valid custom element name or *is* is non-null, then set *result*’s custom element state to "`undefined`".
8. Return *result*.

Elements also have an **attribute list**, which is a list exposed through a `NamedNodeMap`. Unless explicitly given when an element is created, its attribute list is empty.

An element **has** an attribute *A* if its attribute list contains *A*.

This and other specifications may define **attribute change steps** for elements. The algorithm is passed *element*, *localName*, *oldValue*, *value*, and *namespace*.

To **handle attribute changes** for an attribute *attribute* with *element*, *oldValue*, and *newValue*, run these steps:

1. Queue a mutation record of "`attributes`" for *element* with *attribute*’s local name, *attribute*’s namespace, *oldValue*, « », « », null, and null.
2. If *element* is custom, then enqueue a custom element callback reaction with *element*, callback name "`attributeChangedCallback`", and an argument list containing *attribute*’s local name, *oldValue*, *newValue*, and *attribute*’s namespace.
3. Run the attribute change steps with *element*, *attribute*’s local name, *oldValue*, *newValue*, and *attribute*’s namespace.

To **change** an attribute *attribute* to *value*, run these steps:

1. Handle attribute changes for *attribute* with *attribute*’s element, *attribute*’s value, and *value*.
2. Set *attribute*’s value to *value*.

To **append** an attribute *attribute* to an element *element*, run these steps:

1. Handle attribute changes for *attribute* with *element*, null, and *attribute*’s value.
2. Append *attribute* to *element*’s attribute list.
3. Set *attribute*’s element to *element*.

To **remove** an attribute *attribute*, run these steps:

1. Handle attribute changes for *attribute* with *attribute*’s element, *attribute*’s value, and null.
2. Remove *attribute* from *attribute*’s element’s attribute list.
3. Set *attribute*’s element to null.

To **replace** an attribute *oldAttr* with an attribute *newAttr*, run these steps:

1. Handle attribute changes for *oldAttr* with *oldAttr*’s element, *oldAttr*’s value, and *newAttr*’s value.
2. Replace *oldAttr* by *newAttr* in *oldAttr*’s element’s attribute list.
3. Set *newAttr*’s element to *oldAttr*’s element.
4. Set *oldAttr*’s element to null.

To **get an attribute by name** given a *qualifiedName* and element *element*, run these steps:

1. If *element* is in the HTML namespace and its node document is an HTML document, then set *qualifiedName* to *qualifiedName* in ASCII lowercase.
2. Return the first attribute in *element*’s attribute list whose qualified name is *qualifiedName*, and null otherwise.

To **get an attribute by namespace and local name** given a *namespace*, *localName*, and element *element*, run these steps:

1. If *namespace* is the empty string, set it to null.
2. Return the attribute in *element*’s attribute list whose namespace is *namespace* and local name is *localName*, if any, and null otherwise.

To **get an attribute value** given an element *element*, *localName*, and optionally a *namespace* (null unless stated otherwise), run these steps:

1. Let *attr* be the result of getting an attribute given *namespace*, *localName*, and *element*.
2. If *attr* is null, then return the empty string.

3. Return *attr*’s value.

To **set an attribute** given an *attr* and *element*, run these steps:

1. If *attr*’s element is neither null nor *element*, throw an "`InUseAttributeError`" `DOMException`.
2. Let *oldAttr* be the result of getting an attribute given *attr*’s namespace, *attr*’s local name, and *element*.
3. If *oldAttr* is *attr*, return *attr*.
4. If *oldAttr* is non-null, then replace *oldAttr* with *attr*.
5. Otherwise, append *attr* to *element*.
6. Return *oldAttr*.

To **set an attribute value** for an element *element*, using a *localName* and *value*, and an optional *prefix*, and *namespace*, run these steps:

1. If *prefix* is not given, set it to null.
2. If *namespace* is not given, set it to null.
3. Let *attribute* be the result of getting an attribute given *namespace*, *localName*, and *element*.
4. If *attribute* is null, create an attribute whose namespace is *namespace*, namespace prefix is *prefix*, local name is *localName*, value is *value*, and node document is *element*’s node document, then append this attribute to *element*, and then return.
5. Change *attribute* to *value*.

To **remove an attribute by name** given a *qualifiedName* and element *element*, run these steps:

1. Let *attr* be the result of getting an attribute given *qualifiedName* and *element*.
2. If *attr* is non-null, then remove *attr*.
3. Return *attr*.

To **remove an attribute by namespace and local name** given a *namespace*, *localName*, and element *element*, run these steps:

1. Let *attr* be the result of getting an attribute given *namespace*, *localName*, and *element*.
2. If *attr* is non-null, then remove *attr*.
3. Return *attr*.

An element can have an associated **unique identifier (ID)**

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.

Use these attribute change steps to update an element’s ID:

1. If *localName* is `id`, *namespace* is null, and *value* is null or the empty string, then unset *element*’s ID.
2. Otherwise, if *localName* is `id`, *namespace* is null, then set *element*’s ID to *value*.

Note

While this specification defines requirements for `class`, `id`, and `slot` 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.

For web developers (non-normative)

***namespace* = *element* . `namespaceURI`**

Returns the namespace.

***prefix* = *element* . `prefix`**

Returns the namespace prefix.

***localName* = *element* . `localName`**

Returns the local name.

***qualifiedName* = *element* . `tagName`**

Returns the HTML-uppercased qualified name.

The **`namespaceURI`** attribute's getter must return this's namespace.

The **`prefix`** attribute's getter must return this's namespace prefix.

The **`localName`** attribute's getter must return this's local name.

The **`tagName`** attribute's getter must return this's HTML-uppercased qualified name.

For web developers (non-normative)

`element` `.` `id` `[ =` *`value`* `]`

Returns the value of *element*'s `id` content attribute. Can be set to change it.

`element` `.` `className` `[ =` *`value`* `]`

Returns the value of *element*'s `class` content attribute. Can be set to change it.

`element` `.` `classList`

Allows for manipulation of *element*'s `class` content attribute as a set of whitespace-separated tokens through a `DOMTokenList` object.

`element` `.` `slot` `[ =` *`value`* `]`

Returns the value of *element*'s `slot` content attribute. Can be set to change it.

IDL attributes that are defined to **reflect** a content attribute of a given *name*, must have a getter and setter that follow these steps:

getter

Return the result of running get an attribute value given this and *name*.

setter

Set an attribute value for this using *name* 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's getter must return a `DOMTokenList` object whose associated element is this and whose associated attribute's local name is `class`. The token set of this particular `DOMTokenList` object are also known as the element's **classes**.

The **`slot`** attribute must reflect the "`slot`" content attribute.

Note

`id`, `class`, and `slot` are effectively superglobal attributes as they can appear on any element, regardless of that element's namespace.

For web developers (non-normative)

`element` **`.`** **`hasAttributes()`**

Returns true if *element* has attributes, and false otherwise.

`element` **`.`** **`getAttributeNames()`**

Returns the qualified names of all *element*'s attributes. Can contain duplicates.

`element` **`.`** **`getAttribute(`*`qualifiedName`*`)`**

Returns *element*'s first attribute whose qualified name is *qualifiedName*, and null if there is no such attribute otherwise.

`element` **`.`** **`getAttributeNS(`*`namespace`*`,`** ***`localName`*****`)`**

Returns *element*'s attribute whose namespace is *namespace* and local name is *localName*, and null if there is no such attribute otherwise.

`element` **`.`** **`setAttribute(`*`qualifiedName`*`,`** ***`value`*****`)`**

Sets the value of *element*'s first attribute whose qualified name is *qualifiedName* to *value*.

`element` **`.`** **`setAttributeNS(`*`namespace`*`,`** ***`localName`*****`,`** ***`value`*****`)`**

Sets the value of *element*'s attribute whose namespace is *namespace* and local name is *localName* to *value*.

`element` **`.`** **`removeAttribute(`*`qualifiedName`*`)`**

Removes *element*'s first attribute whose qualified name is *qualifiedName*.

`element` **`.`** **`removeAttributeNS(`*`namespace`*`,`** ***`localName`*****`)`**

Removes *element*'s attribute whose namespace is *namespace* and local name is *localName*.

`element` **`.`** **`toggleAttribute(`*`qualifiedName`*** **`[,`** ***`force`*****`])`**

If *force* is not given, "toggles" *qualifiedName*, removing it if it is present and adding it if it is not present. If *force* is true, adds *qualifiedName*. If *force* is false, removes *qualifiedName*.

Returns true if *qualifiedName* is now present, and false otherwise.

`element` **`.`** **`hasAttribute(`*`qualifiedName`*`)`**

Returns true if *element* has an attribute whose qualified name is *qualifiedName*, and false otherwise.

`element` **`.`** **`hasAttributeNS(`*`namespace`*`,`** ***`localName`*****`)`**

Returns true if *element* has an attribute whose namespace is *namespace* and local name is *localName*.

The **`hasAttributes()`** method, when invoked, must return false if this's attribute list is empty, and true otherwise.

The **`attributes`** attribute's getter must return the associated `NamedNodeMap`.

The **`getAttributeNames()`** method, when invoked, must return the qualified names of the attributes in this's attribute list, in order, and a new list otherwise.

Note

These are not guaranteed to be unique.

The **`getAttribute(`*`qualifiedName`*`)`** method, when invoked, must run these steps:

1. Let *attr* be the result of getting an attribute given *qualifiedName* and this.
2. If *attr* is null, return null.
3. Return *attr*'s value.

The **`getAttributeNS(`*`namespace, localName`*`)`** method, when invoked, must these steps:

1. Let *attr* be the result of getting an attribute given *namespace*, *localName*, and this.
2. If *attr* is null, return null.
3. Return *attr*'s value.

The **`setAttribute(`*`qualifiedName, value`*`)`** method, when invoked, must run these steps:

1. If *qualifiedName* does not match the `Name` production in XML, then throw an "`InvalidCharacterError`" `DOMException`.
2. If this is in the HTML namespace and its node document is an HTML document, then set *qualifiedName* to *qualifiedName* in ASCII lowercase.
3. Let *attribute* be the first attribute in this's attribute list whose qualified name is *qualifiedName*, and null otherwise.
4. If *attribute* is null, create an attribute whose local name is *qualifiedName*, value is *value*, and node document is this's node document, then append this attribute to this, and then return.
5. Change *attribute* to *value*.

The **`setAttributeNS(`*`namespace, qualifiedName, value`*`)`** method, when invoked, must run these steps:

1. Let *namespace*, *prefix*, and *localName* be the result of passing *namespace* and *qualifiedName* to validate and extract.
2. Set an attribute value for this using *localName*, *value*, and also *prefix* and *namespace*.

The **`removeAttribute(`*`qualifiedName`*`)`** method, when invoked, must remove an attribute given *qualifiedName* and this, and then return undefined.

The **`removeAttributeNS(`*`namespace, localName`*`)`** method, when invoked, must remove an attribute given *namespace*, *localName*, and this, and then return undefined.

The **`hasAttribute(`*`qualifiedName`*`)`** method, when invoked, must run these steps:

1. If this is in the HTML namespace and its node document is an HTML document, then set *qualifiedName* to *qualifiedName* in ASCII lowercase.
2. Return true if this has an attribute whose qualified name is *qualifiedName*, and false otherwise.

The **`toggleAttribute(`*`qualifiedName, force`*`)`** method, when invoked, must run these steps:

1. If *qualifiedName* does not match the `Name` production in XML, then throw an "`InvalidCharacterError`" `DOMException`.
2. If this is in the HTML namespace and its node document is an HTML document, then set *qualifiedName* to *qualifiedName* in ASCII lowercase.
3. Let *attribute* be the first attribute in this's attribute list whose qualified name is *qualifiedName*, and null otherwise.
4. If *attribute* is null, then:
    1. If *force* is not given or is true, create an attribute whose local name is *qualifiedName*, value is the empty string, and node document is this's node document, then append this attribute to this, and then return true.
    2. Return false.

5. Otherwise, if *force* is not given or is false, remove an attribute given *qualifiedName* and this, and then return false.

6. Return true.

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

1. If *namespace* is the empty string, set it to null.

2. Return true if this has an attribute whose namespace is *namespace* and local name is *localName*, and false otherwise.

The **`getAttributeNode(qualifiedName)`** method, when invoked, must return the result of getting an attribute given *qualifiedName* and this.

The **`getAttributeNodeNS(namespace, localName)`** method, when invoked, must return the result of getting an attribute given *namespace*, *localName*, and this.

The **`setAttributeNode(attr)`** and **`setAttributeNodeNS(attr)`** methods, when invoked, must return the result of setting an attribute given *attr* and this.

The **`removeAttributeNode(attr)`** method, when invoked, must run these steps:

1. If this's attribute list does not contain *attr*, then throw a "`NotFoundError`" `DOMException`.

2. Remove *attr*.

3. Return *attr*.

For web developers (non-normative)

`var shadow = element . attachShadow(init)`

Creates a shadow root for *element* and returns it.

`var shadow = element . shadowRoot`

Returns *element*'s shadow root, if any, and if shadow root's mode is "`open`", and null otherwise.

The **`attachShadow(init)`** method, when invoked, must run these steps:

1. If this's namespace is not the HTML namespace, then throw a "`NotSupportedError`" `DOMException`.

2. If this's local name is not one of the following:

   - a valid custom element name
   - "`article`", "`aside`", "`blockquote`", "`body`", "`div`", "`footer`", "`h1`", "`h2`", "`h3`", "`h4`", "`h5`", "`h6`", "`header`", "`main`", "`nav`", "`p`", "`section`", or "`span`"

   then throw a "`NotSupportedError`" `DOMException`.

3. If this's local name is a valid custom element name, or this's `is` value is not null, then:

   1. Let *definition* be the result of looking up a custom element definition given this's node document, its namespace, its local name, and its `is` value.

   2. If *definition* is not null and *definition*'s disable shadow is true, then throw a "`NotSupportedError`" `DOMException`.

4. If this is a shadow host, then throw an "`NotSupportedError`" `DOMException`.

5. Let *shadow* be a new shadow root whose node document is this's node document, host is this, and mode is *init*'s `mode`.

6. Set *shadow*'s delegates focus to *init*'s `delegatesFocus`.

7. Set this's shadow root to *shadow*.

8. Return *shadow*.

The **`shadowRoot`** attribute's getter must run these steps:

1. Let *shadow* be this's shadow root.

2. If *shadow* is null or its mode is "`closed`", then return null.

3. Return *shadow*.

For web developers (non-normative)

`element` **`. closest(selectors)`**

Returns the first (starting at *element*) inclusive ancestor that matches *selectors*, and null otherwise.

`element` **`. matches(selectors)`**

Returns true if matching *selectors* against *element*'s root yields *element*, and false otherwise.

The **`closest(`*`selectors`*`)`** method, when invoked, must run these steps:

1. Let *s* be the result of parse a selector from *selectors*. [SELECTORS4]

2. If *s* is failure, throw a "`SyntaxError`" `DOMException`.

3. Let *elements* be this's inclusive ancestors that are elements, in reverse tree order.

4. For each *element* in *elements*, if match a selector against an element, using *s*, *element*, and :scope element this, returns success, return *element*. [SELECTORS4]

5. Return null.

The **`matches(`*`selectors`*`)`** and **`webkitMatchesSelector(`*`selectors`*`)`** methods, when invoked, must run these steps:

1. Let *s* be the result of parse a selector from *selectors*. [SELECTORS4]

2. If *s* is failure, throw a "`SyntaxError`" `DOMException`.

3. Return true if the result of match a selector against an element, using *s*, *element*, and :scope element this, returns success, and false otherwise. [SELECTORS4]

The **`getElementsByTagName(`*`qualifiedName`*`)`** method, when invoked, must return the list of elements with qualified name *qualifiedName* for this.

The **`getElementsByTagNameNS(`*`namespace, localName`*`)`** method, when invoked, must return the list of elements with namespace *namespace* and local name *localName* for this.

The **`getElementsByClassName(`*`classNames`*`)`** method, when invoked, must return the list of elements with class names *classNames* for this.

To **insert adjacent**, given an element *element*, string *where*, and a node *node*, run the steps associated with the first ASCII case-insensitive match for *where*:

"`beforebegin`"
: If *element*'s parent is null, return null.

  Return the result of pre-inserting *node* into *element*'s parent before *element*.

"`afterbegin`"
: Return the result of pre-inserting *node* into *element* before *element*'s first child.

"`beforeend`"
: Return the result of pre-inserting *node* into *element* before null.

"`afterend`"
: If *element*'s parent is null, return null.

  Return the result of pre-inserting *node* into *element*'s parent before *element*'s next sibling.

Otherwise
: Throw a "`SyntaxError`" `DOMException`.

The **`insertAdjacentElement(where, element)`** method, when invoked, must return the result of running insert adjacent, give this, *where*, and *element*.

The **`insertAdjacentText(where, data)`** method, when invoked, must run these steps:

1. Let *text* be a new `Text` node whose data is *data* and node document is this's node document.

2. Run insert adjacent, given this, *where*, and *text*.

Note

This method returns nothing because it existed before we had a chance to design it.

4.9.1. Interface `NamedNodeMap` §

IDL
```
[Exposed=Window,
 LegacyUnenumerableNamedProperties]
interface NamedNodeMap {
  readonly attribute unsigned long length;
  getter Attr? item(unsigned long index);
  getter Attr? getNamedItem(DOMString qualifiedName);
  Attr? getNamedItemNS(DOMString? namespace, DOMString localName);
  [CEReactions] Attr? setNamedItem(Attr attr);
  [CEReactions] Attr? setNamedItemNS(Attr attr);
  [CEReactions] Attr removeNamedItem(DOMString qualifiedName);
  [CEReactions] Attr removeNamedItemNS(DOMString? namespace, DOMString localName);
};
```

A `NamedNodeMap` has an associated **element** (an element).

A `NamedNodeMap` object's **attribute list** is its element's attribute list.

A `NamedNodeMap` object's supported property indices are the numbers in the range zero to its attribute list's size minus one, unless the attribute list is empty, in which case there are no supported property indices.

The **`length`** attribute's getter must return the attribute list's size.

The **`item(index)`** method, when invoked, must run these steps:

1. If *index* is equal to or greater than this's attribute list's size, then return null.
2. Otherwise, return this's attribute list[*index*].

A `NamedNodeMap` object's supported property names are the return value of running these steps:

1. Let *names* be the qualified names of the attributes in this `NamedNodeMap` object's attribute list, with duplicates omitted, in order.
2. If this `NamedNodeMap` object's element is in the HTML namespace and its node document is an HTML document, then for each *name* in *names*:
    1. Let *lowercaseName* be *name*, in ASCII lowercase.
    2. If *lowercaseName* is not equal to *name*, remove *name* from *names*.
3. Return *names*.

The **`getNamedItem(qualifiedName)`** method, when invoked, must return the result of getting an attribute given *qualifiedName* and element.

The **`getNamedItemNS(namespace, localName)`** method, when invoked, must return the result of getting an attribute given *namespace*, *localName*, and element.

The **`setNamedItem(attr)`** and **`setNamedItemNS(attr)`** methods, when invoked, must return the result of setting an attribute given *attr* and element.

The **`removeNamedItem(qualifiedName)`** method, when invoked, must run these steps:

1. Let *attr* be the result of removing an attribute given *qualifiedName* and element.
2. If *attr* is null, then throw a "`NotFoundError`" `DOMException`.
3. Return *attr*.

The **`removeNamedItemNS(namespace, localName)`** method, when invoked, must run these steps:

1. Let *attr* be the result of removing an attribute given *namespace*, *localName*, and element.
2. If *attr* is null, then throw a "`NotFoundError`" `DOMException`.
3. Return *attr*.

4.9.2. Interface `Attr` §

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

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

`Attr` nodes 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), **value** (a string), and **element** (null or an element).

Note

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

An attribute's **qualified name** is its local name if its namespace prefix is null, and its namespace prefix, followed by "`:`", followed by its local name, otherwise.

Note

User agents could have this as an internal slot as an optimization.

When an attribute is created, its local name is given. Unless explicitly given when an attribute is created, its namespace, namespace prefix, and element are set to null, and its value is set to the empty string.

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

The **`namespaceURI`** attribute's getter must return the namespace.

The **`prefix`** attribute's getter must return the namespace prefix.

The **`localName`** attribute's getter must return the local name.

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

The **`value`** attribute's getter must return the value.

To **set an existing attribute value**, given an attribute *attribute* and string *value*, run these steps:

1. If *attribute*'s element is null, then set *attribute*'s value to *value*.
2. Otherwise, change *attribute* to *value*.

The `value` attribute's setter must set an existing attribute value with this and the given value.

The **`ownerElement`** attribute's getter must return this's element.

The **`specified`** attribute's getter must return true.

4.10. Interface `CharacterData` §

IDL
```
[Exposed=Window]
interface CharacterData : Node {
  attribute [LegacyNullToEmptyString] 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`, `ProcessingInstruction`, and `Comment` 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.
2. If *offset* is greater than *length*, then throw an "`IndexSizeError`" `DOMException`.
3. If *offset* plus *count* is greater than *length*, then set *count* to *length* minus *offset*.
4. Queue a mutation record of "`characterData`" for *node* with null, null, *node*'s data, « », « », null, and null.
5. Insert *data* into *node*'s data after *offset* code units.
6. Let *delete offset* be *offset* + *data*'s length.
7. Starting from *delete offset* code units, remove *count* code units from *node*'s data.
8. For each live 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 live 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 live range whose start node is *node* and start offset is greater than *offset* plus *count*, increase its start offset by *data*'s length and decrease it by *count*.
11. For each live range whose end node is *node* and end offset is greater than *offset* plus *count*, increase its end offset by *data*'s length and decrease it by *count*.
12. If *node*'s parent is non-null, then run the children changed steps for *node*'s parent.

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

1. Let *length* be *node*'s length.
2. If *offset* is greater than *length*, then throw an "`IndexSizeError`" `DOMException`.
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 return.
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's getter must return this's data. Its setter must replace data with node this, offset 0, count this's length, and data new value.

The **`length`** attribute's getter must return this's length.

The **`substringData(`*`offset`*`, `*`count`*`)`** method, when invoked, must return the result of running substring data with node this, offset *offset*, and count *count*.

The **`appendData(`*`data`*`)`** method, when invoked, must replace data with node this, offset this's length, count 0, and data *data*.

The **`insertData(`*`offset`*`, `*`data`*`)`** method, when invoked, must replace data with node this, offset *offset*, count 0, and data *data*.

The **`deleteData(`*`offset`*`, `*`count`*`)`** method, when invoked, must replace data with node this, offset *offset*, count *count*, and data the empty string.

The **`replaceData(`*`offset`*`, `*`count`*`, `*`data`*`)`** method, when invoked, must replace data with node this, offset *offset*, count *count*, and data *data*.

4.11. Interface `Text` §

IDL
```
[Exposed=Window]
interface Text : CharacterData {
  constructor(optional DOMString data = "");

  [NewObject] Text splitText(unsigned long offset);
  readonly attribute DOMString wholeText;
};
```

For web developers (non-normative)

`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.

An **exclusive `Text` node** is a `Text` node that is not a `CDATASection` node.

The **contiguous `Text` nodes** of a node *node* are *node*, *node*'s previous sibling `Text` node, if any, and its contiguous `Text` nodes, and *node*'s next sibling `Text` node, if any, and its contiguous `Text` nodes, avoiding any duplicates.

The **contiguous exclusive `Text` nodes** of a node *node* are *node*, *node*'s previous sibling exclusive `Text` node, if any, and its contiguous exclusive `Text` nodes, and *node*'s next sibling exclusive `Text` node, if any, and its contiguous exclusive `Text` nodes, avoiding any duplicates.

The **child text content** of a node *node* is the concatenation of the data of all the `Text` node children of *node*, in tree order.

The **descendant text content** of a node *node* is the concatenation of the data of all the `Text` node descendants of *node*, in tree order.

The `Text(data)` constructor, when invoked, must return a new `Text` node whose data is *data* and node document is current global object's associated `Document`.

To **split** a `Text` node *node* with offset *offset*, run these steps:

1. Let *length* be *node*'s length.
2. If *offset* is greater than *length*, then throw an "`IndexSizeError`" `DOMException`.
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, then:
    1. Insert *new node* into *parent* before *node*'s next sibling.
    2. For each live 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 live 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 live range whose start node is *parent* and start offset is equal to the index of *node* plus 1, increase its start offset by 1.
    5. For each live range whose end node is *parent* and end offset is equal to the index of *node* plus 1, increase its end offset by 1.
8. Replace data with node *node*, offset *offset*, count *count*, and data the empty string.
9. Return *new node*.

The `splitText(offset)` method, when invoked, must split this with offset *offset*.

The `wholeText` attribute's getter must return the concatenation of the data of the contiguous `Text` nodes of this, in tree order.

4.12. Interface `CDATASection` §

IDL
```
[Exposed=Window]
interface CDATASection : Text {
};
```

4.13. 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.

4.14. Interface `Comment` §

IDL
```
[Exposed=Window]
interface Comment : CharacterData {
  constructor(optional DOMString data = "");
};
```

For web developers (non-normative)

`comment` `= new Comment([`*`data`*` = ""])`

Returns a new `Comment` node whose data is *data*.

The **`Comment(`*`data`*`)`** constructor, when invoked, must return a new `Comment` node whose data is *data* and node document is current global object's associated `Document`.

5. Ranges §

5.1. Introduction to "DOM Ranges" §

`StaticRange` and `Range` objects (ranges) represent 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 an 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.

```
└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. Ranges are only useful for nodes.

`Range` objects, unlike `StaticRange` objects, are affected by mutations to the node tree. Therefore they are also known as live ranges. Such mutations will not invalidate them and will try to ensure that it still represents the same piece of content. Necessarily, a live 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 details.

Updating live ranges in response to node tree mutations can be expensive. For every node tree change, all affected `Range` objects need to be updated. Even if the application is uninterested in some live ranges, it still has to pay the cost of keeping them up-to-date when a mutation occurs.

A `StaticRange` object is a lightweight range that does not update when the node tree mutates. It is therefore not subject to the same maintenance cost as live ranges.

5.2. Boundary points §

A **boundary point** is a tuple consisting of a **node** (a node) and an **offset** (a non-negative integer).

Note

A correct boundary point's offset will be between 0 and the boundary point's node's length, inclusive.

The **position** of a boundary point (*nodeA*, *offsetA*) relative to a boundary point (*nodeB*, *offsetB*) is **before**, **equal**, or **after**, as returned by these steps:

1. Assert: *nodeA* and *nodeB* have the same root.
2. If *nodeA* is *nodeB*, then return equal if *offsetA* is *offsetB*, before if *offsetA* is less than *offsetB*, and after if *offsetA* is greater than *offsetB*.
3. If *nodeA* is following *nodeB*, then if the position of (*nodeB*, *offsetB*) relative to (*nodeA*, *offsetA*) is before, return after, and if it is after, return before.
4. If *nodeA* is an ancestor of *nodeB*:
    1. Let *child* be *nodeB*.
    2. While *child* is not a child of *nodeA*, set *child* to its parent.
    3. If *child*'s index is less than *offsetA*, then return after.
5. Return before.

5.3. Interface `AbstractRange` §

IDL
```
[Exposed=Window]
interface AbstractRange {
  readonly attribute Node startContainer;
  readonly attribute unsigned long startOffset;
  readonly attribute Node endContainer;
  readonly attribute unsigned long endOffset;
  readonly attribute boolean collapsed;
};
```

Objects implementing the `AbstractRange` interface are known as **ranges**.

A range has two associated boundary points — a **start** and **end**.

For convenience, a range's **start node** is its start's node, its **start offset** is its start's offset, its **end node** is its end's node, and its **end offset** is its end's offset.

A range is **collapsed** if its start node is its end node and its start offset is its end offset.

For web developers (non-normative)

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* is collapsed, and false otherwise.

The `startContainer` attribute's getter must return this's start node.

The `startOffset` attribute's getter must return this's start offset.

The `endContainer` attribute's getter must return this's end node.

The `endOffset` attribute's getter must return this's end offset.

The `collapsed` attribute's getter must return true if this is collapsed, and false otherwise.

5.4. Interface `StaticRange` §

IDL
```
dictionary StaticRangeInit {
  required Node startContainer;
  required unsigned long startOffset;
  required Node endContainer;
  required unsigned long endOffset;
};

[Exposed=Window]
interface StaticRange : AbstractRange {
  constructor(StaticRangeInit init);
};
```

For web developers (non-normative)

***staticRange* = new StaticRange(init)**

Returns a new range object that does not update when the node tree mutates.

The `StaticRange(`*`init`*`)` constructor, when invoked, must run these steps:

1. If *init*'s `startContainer` or `endContainer` is a `DocumentType` or `Attr` node, then throw an "`InvalidNodeTypeError`" `DOMException`.
2. Let *staticRange* be a new `StaticRange` object.
3. Set *staticRange*'s start to (*init*'s `startContainer`, *init*'s `startOffset`) and end to (*init*'s `endContainer`, *init*'s `endOffset`).
4. Return *staticRange*.

5.5. Interface `Range` §

IDL
```
[Exposed=Window]
interface Range : AbstractRange {
  constructor();

  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);

  [CEReactions] void deleteContents();
  [CEReactions, NewObject] DocumentFragment extractContents();
  [CEReactions, NewObject] DocumentFragment cloneContents();
  [CEReactions] void insertNode(Node node);
  [CEReactions] 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;
};
```

Objects implementing the Range interface are known as **live ranges**.

Note

Algorithms that modify a tree (in particular the insert, remove, replace data, and split algorithms) modify live ranges associated with that tree.

The **root** of a live range is the root of its start node.

A node *node* is **contained** in a live range *range* if *node*'s root is *range*'s root, and (*node*, 0) is after *range*'s start, and (*node*, *node*'s length) is before *range*'s end.

A node is **partially contained** in a live range if it's an inclusive ancestor of the live range's start node but not its end node, or vice versa.

Note

Some facts to better understand these definitions:

- *The content that one would think of as being within the live 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 live 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 start node and end node of a live range are never contained within it.*
- *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.*
- *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.*

For web developers (non-normative)

`range = new Range()`

Returns a new live range.

The `Range()` constructor, when invoked, must return a new live range with (current global object's associated `Document`, 0) as its start and end.

For web developers (non-normative)

***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 `commonAncestorContainer` attribute's getter 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, then throw an "`InvalidNodeTypeError`" `DOMException`.
2. If *offset* is greater than *node*'s length, then throw an "`IndexSizeError`" `DOMException`.
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, when invoked, must set the start of this to boundary point (*node*, *offset*).

The `setEnd(node, offset)` method, when invoked, must set the end of this to boundary point (*node*, *offset*).

The `setStartBefore(node)` method, when invoked, must run these steps:

1. Let *parent* be *node*'s parent.
2. If *parent* is null, then throw an "`InvalidNodeTypeError`" `DOMException`.
3. Set the start of this to boundary point (*parent*, *node*'s index).

The `setStartAfter(node)` method, when invoked, must run these steps:

1. Let *parent* be *node*'s parent.
2. If *parent* is null, then throw an "`InvalidNodeTypeError`" `DOMException`.
3. Set the start of this to boundary point (*parent*, *node*'s index plus 1).

The `setEndBefore(node)`, when invoked, method must run these steps:

1. Let *parent* be *node*'s parent.
2. If *parent* is null, then throw an "`InvalidNodeTypeError`" `DOMException`.
3. Set the end of this to boundary point (*parent*, *node*'s index).

The `setEndAfter(node)` method, when invoked, must run these steps:

1. Let *parent* be *node*'s parent.
2. If *parent* is null, then throw an "`InvalidNodeTypeError`" `DOMException`.
3. Set the end of this to boundary point (*parent*, *node*'s index plus 1).

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 *node* within a range *range*, run these steps:

1. Let *parent* be *node*'s parent.
2. If *parent* is null, then throw an "`InvalidNodeTypeError`" `DOMException`.
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 1).

The `selectNode(node)` method, when invoked, must select *node* within this.

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

1. If *node* is a doctype, throw an "`InvalidNodeTypeError`" `DOMException`.
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, when invoked, must run these steps:

1. If *how* is not one of

   - START_TO_START,
   - START_TO_END,
   - END_TO_END, and
   - END_TO_START,

   then throw a "NotSupportedError" DOMException.

2. If this's root is not the same as *sourceRange*'s root, then throw a "WrongDocumentError" DOMException.

3. If *how* is:

   START_TO_START:

   Let *this point* be this's start. Let *other point* be *sourceRange*'s start.

   START_TO_END:

   Let *this point* be this's end. Let *other point* be *sourceRange*'s start.

   END_TO_END:

   Let *this point* be this's end. Let *other point* be *sourceRange*'s end.

   END_TO_START:

   Let *this point* be this'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, when invoked, must run these steps:

1. If this is collapsed, then return.

2. Let *original start node*, *original start offset*, *original end node*, and *original end offset* be this'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 return.

4. Let *nodes to remove* be a list of all the nodes that are contained in this, in tree order, omitting any node whose parent is also contained in this.

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 this, 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*.

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 live 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* is collapsed, then 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* is *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, then throw a "`HierarchyRequestError`" `DOMException`.

    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 live 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 live 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, when invoked, must return the result of extracting this.

To **clone the contents** of a live 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* is collapsed, then 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* is *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, then throw a "`HierarchyRequestError`" `DOMException`.

    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 live 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 the contents of *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 live 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 the contents of *subrange*.
    5. Append *subfragment* to *clone*.
18. Return *fragment*.

The **`cloneContents()`** method, when invoked, must return the result of cloning the contents of this.

To **insert** a node *node* into a live range *range*, run these steps:

1. If *range*'s start node is a `ProcessingInstruction` or `Comment` node, is a `Text` node whose parent is null, or is *node*, then throw a "`HierarchyRequestError`" `DOMException`.
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, set *referenceNode* to the result of splitting it with offset *range*'s start offset.
8. If *node* is *referenceNode*, set *referenceNode* to its next sibling.
9. If *node*'s parent is non-null, then remove *node*.
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* is collapsed, then set *range*'s end to (*parent*, *newOffset*).

The **`insertNode(`*`node`*`)`** method, when invoked, must insert *node* into this.

The **`surroundContents(`*`newParent`*`)`** method, when invoked, must run these steps:

1. If a non-`Text` node is partially contained in this, then throw an "`InvalidStateError`" `DOMException`.
2. If *newParent* is a `Document`, `DocumentType`, or `DocumentFragment` node, then throw an "`InvalidNodeTypeError`" `DOMException`.

Note

For historical reasons `Text`, `ProcessingInstruction`, and `Comment` nodes are not checked here and end up throwing later on as a side effect.

3. Let *fragment* be the result of extracting this.
4. If *newParent* has children, then replace all with null within *newParent*.
5. Insert *newParent* into this.
6. Append *fragment* to *newParent*.
7. Select *newParent* within this.

The **`cloneRange()`** method, when invoked, must return a new live range with the same start and end as this.

The **`detach()`** method, when invoked, must do nothing. Note *Its functionality (disabling a `Range` object) was removed, but the method itself is preserved for compatibility.*

For web developers (non-normative)

***position* = *range* . `comparePoint(node, 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`*`)`** method, when invoked, must run these steps:

1. If *node*'s root is different from this's root, return false.
2. If *node* is a doctype, then throw an "`InvalidNodeTypeError`" `DOMException`.
3. If *offset* is greater than *node*'s length, then throw an "`IndexSizeError`" `DOMException`.
4. If (*node*, *offset*) is before start or after end, return false.
5. Return true.

The **`comparePoint(`*`node, offset`*`)`** method, when invoked, must run these steps:

1. If *node*'s root is different from this's root, then throw a "`WrongDocumentError`" `DOMException`.
2. If *node* is a doctype, then throw an "`InvalidNodeTypeError`" `DOMException`.
3. If *offset* is greater than *node*'s length, then throw an "`IndexSizeError`" `DOMException`.
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, when invoked, must run these steps:

1. If *node*'s root is different from this'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* plus 1) is after start, return true.
6. Return false.

The **stringification behavior** must run these steps:

1. Let *s* be the empty string.
2. If this’s start node is this’s end node and it is a `Text` node, then return the substring of that `Text` node’s data beginning at this’s start offset and ending at this’s end offset.
3. If this’s start node is a `Text` node, then append the substring of that node’s data from this’s start offset until the end to *s*.
4. Append the concatenation of the data of all `Text` nodes that are contained in this, in tree order, to *s*.
5. If this’s end node is a `Text` node, then append the substring of that node’s data from its start until this’s end offset to *s*.
6. Return *s*.

Note

The `createContextualFragment()`, `getClientRects()`, and `getBoundingClientRect()` methods are defined in other specifications. [DOM-Parsing] [CSSOM-VIEW]

6. Traversal §

NodeIterator and TreeWalker objects can be used to filter and traverse node trees.

Each NodeIterator and TreeWalker object has an associated **active flag** to avoid recursive invocations. It is initially unset.

Each NodeIterator and TreeWalker object also has an associated **root** (a node), a **whatToShow** (a bitmask), and a **filter** (a callback).

To **filter** a node *node* within a NodeIterator or TreeWalker object *traverser*, run these steps:

1. If *traverser*'s active flag is set, then throw an "InvalidStateError" DOMException.
2. Let *n* be *node*'s nodeType attribute value − 1.
3. If the n^{th} bit (where 0 is the least significant bit) of *traverser*'s whatToShow is not set, then return FILTER_SKIP.
4. If *traverser*'s filter is null, then return FILTER_ACCEPT.
5. Set *traverser*'s active flag.
6. Let *result* be the return value of call a user object's operation with *traverser*'s filter, "acceptNode", and « *node* ». If this throws an exception, then unset *traverser*'s active flag and rethrow the exception.
7. Unset *traverser*'s active flag.
8. 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 on Document objects.

Each NodeIterator object has an associated **iterator collection**, which is a collection rooted at the NodeIterator object's root, whose filter matches any node.

Each NodeIterator object also has an associated **reference** (a node) and **pointer before reference** (a boolean).

Note

As mentioned earlier, NodeIterator objects have an associated active flag, root, whatToShow, and filter as well.

The **NodeIterator pre-removing steps** given a *nodeIterator* and *toBeRemovedNode*, are as follows:

1. If *toBeRemovedNode* is not an inclusive ancestor of *nodeIterator*'s reference, or *toBeRemovedNode* is *nodeIterator*'s root, then return.

2. If *nodeIterator*'s pointer before reference is true, then:

    1. Let *next* be *toBeRemovedNode*'s first following node that is an inclusive descendant of *nodeIterator*'s root and is not an inclusive descendant of *toBeRemovedNode*, and null if there is no such node.

    2. If *next* is non-null, then set *nodeIterator*'s reference to *next* and return.

    3. Otherwise, set *nodeIterator*'s pointer before reference to false.

    Note

    Steps are not terminated here.

3. Set *nodeIterator*'s reference to *toBeRemovedNode*'s parent, if *toBeRemovedNode*'s previous sibling is null, and to the inclusive descendant of *toBeRemovedNode*'s previous sibling that appears last in tree order otherwise.

The **`root`** attribute's getter, when invoked, must return this's root.

The **`referenceNode`** attribute's getter, when invoked, must return this's reference.

The **`pointerBeforeReferenceNode`** attribute's getter, when invoked, must return this's pointer before reference.

The **`whatToShow`** attribute's getter, when invoked, must return this's whatToShow.

The **`filter`** attribute's getter, when invoked, must return this's filter.

To **traverse**, given a `NodeIterator` object *iterator* and a direction *direction*, run these steps:

1. Let *node* be *iterator*'s reference.

2. Let *beforeNode* be *iterator*'s pointer before reference.

3. While true:

    1. Branch on *direction*:

        next

        If *beforeNode* is false, then set *node* to the first node following *node* in *iterator*'s iterator collection. If there is no such node, then return null.

        If *beforeNode* is true, then set it to false.

        previous

        If *beforeNode* is true, then set *node* to the first node preceding *node* in *iterator*'s iterator collection. If there is no such node, then return null.

        If *beforeNode* is false, then set it to true.

    2. Let *result* be the result of filtering *node* within *iterator*.

    3. If *result* is `FILTER_ACCEPT`, then break.

4. Set *iterator*'s reference to *node*.

5. Set *iterator*'s pointer before reference to *beforeNode*.

6. Return *node*.

The **`nextNode()`** method, when invoked, must return the result of traversing with this and next.

The **`previousNode()`** method, when invoked, must return the result of traversing with this and previous.

The **`detach()`** method, when invoked, 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 on `Document` objects.

Each `TreeWalker` object has an associated **current** (a node).

Note

As mentioned earlier `TreeWalker` objects have an associated root, whatToShow, and filter as well.

The **`root`** attribute's getter, when invoked, must return this's root.

The **`whatToShow`** attribute's getter, when invoked, must return this's whatToShow.

The **`filter`** attribute's getter, when invoked, must return this's filter.

The **`currentNode`** attribute's getter, when invoked, must return this's current.

The `currentNode` attribute's setter, when invoked, must set this's current to the given value.

The **`parentNode()`** method, when invoked, must run these steps:

```
1. Let node be this's current.
2. While node is non-null and is not this's root:
    1. Set node to node's parent.
    2. If node is non-null and filtering node within this returns FILTER_ACCEPT, then set this's current to node and return node.
```

3. Return null.

To **traverse children**, given a *walker* and *type*, run these steps:

1. Let *node* be *walker*’s current.
2. Set *node* to *node*’s first child if *type* is first, and *node*’s last child if *type* is last.
3. While *node* is non-null:
    1. Let *result* be the result of filtering *node* within *walker*.
    2. If *result* is `FILTER_ACCEPT`, then set *walker*’s current to *node* and return *node*.
    3. If *result* is `FILTER_SKIP`, then:
        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 non-null, then set *node* to *child* and continue.
    4. While *node* is non-null:
        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 non-null, then set *node* to *sibling* and break.
        3. Let *parent* be *node*’s parent.
        4. If *parent* is null, *walker*’s root, or *walker*’s current, then return null.
        5. Set *node* to *parent*.
4. Return null.

The **`firstChild()`** method, when invoked, must traverse children with this and first.

The **`lastChild()`** method, when invoked, must traverse children with this and last.

To **traverse siblings**, given a *walker* and *type*, run these steps:

1. Let *node* be *walker*’s current.
2. If *node* is root, then return null.
3. While true:
    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 non-null:
        1. Set *node* to *sibling*.
        2. Let *result* be the result of filtering *node* within *walker*.
        3. If *result* is `FILTER_ACCEPT`, then set *walker*’s current 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 *node*’s parent.
    4. If *node* is null or *walker*’s root, then return null.
    5. If the return value of filtering *node* within *walker* is `FILTER_ACCEPT`, then return null.

The **`nextSibling()`** method, when invoked, must traverse siblings with this and next.

The **`previousSibling()`** method, when invoked, must traverse siblings with this and previous.

The **`previousNode()`** method, when invoked, must run these steps:

1. Let *node* be this's current.
2. While *node* is not this's root:
    1. Let *sibling* be *node*'s previous sibling.
    2. While *sibling* is non-null:
        1. Set *node* to *sibling*.
        2. Let *result* be the result of filtering *node* within this.
        3. While *result* is not `FILTER_REJECT` and *node* has a child:
            1. Set *node* to *node*'s last child.
            2. Set *result* to the result of filtering *node* within this.
        4. If *result* is `FILTER_ACCEPT`, then set this's current to *node* and return *node*.
        5. Set *sibling* to *node*'s previous sibling.
    3. If *node* is this's root or *node*'s parent is null, then return null.
    4. Set *node* to *node*'s parent.
    5. If the return value of filtering *node* within this is `FILTER_ACCEPT`, then set this's current to *node* and return *node*.
3. Return null.

The **`nextNode()`** method, when invoked, must run these steps:

1. Let *node* be this's current.
2. Let *result* be `FILTER_ACCEPT`.
3. While true:
    1. While *result* is not `FILTER_REJECT` and *node* has a child:
        1. Set *node* to its first child.
        2. Set *result* to the result of filtering *node* within this.
        3. If *result* is `FILTER_ACCEPT`, then set this's current to *node* and return *node*.
    2. Let *sibling* be null.
    3. Let *temporary* be *node*.
    4. While *temporary* is non-null:
        1. If *temporary* is this's root, then return null.
        2. Set *sibling* to *temporary*'s next sibling.
        3. If *sibling* is non-null, then set *node* to *sibling* and break.
        4. Set *temporary* to *temporary*'s parent.
    5. Set *result* to the result of filtering *node* within this.

6. If *result* is FILTER_ACCEPT, then set this's current to *node* and return *node*.

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;
  const unsigned long SHOW_TEXT = 0x4;
  const unsigned long SHOW_CDATA_SECTION = 0x8;
  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);
};
```

Note

NodeFilter objects can be used as filter for NodeIterator and TreeWalker objects and also provide constants for their whatToShow bitmask. A NodeFilter object is typically implemented as a JavaScript function.

These constants can be used as filter return value:

- **FILTER_ACCEPT** (1);
- **FILTER_REJECT** (2);
- **FILTER_SKIP** (3).

These constants can be used for whatToShow:

- **SHOW_ALL** (4294967295, FFFFFFFF in hexadecimal);
- **SHOW_ELEMENT** (1);
- **SHOW_ATTRIBUTE** (2);
- **SHOW_TEXT** (4);
- **SHOW_CDATA_SECTION** (8);
- **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 name DOMTokenList is an unfortunate legacy mishap.

7.1. Interface DOMTokenList §

```
IDL
[Exposed=Window]
interface DOMTokenList {
  readonly attribute unsigned long length;
  getter DOMString? item(unsigned long index);
  boolean contains(DOMString token);
  [CEReactions] void add(DOMString... tokens);
  [CEReactions] void remove(DOMString... tokens);
  [CEReactions] boolean toggle(DOMString token, optional boolean force);
  [CEReactions] boolean replace(DOMString token, DOMString newToken);
  boolean supports(DOMString token);
  [CEReactions] stringifier attribute DOMString value;
  iterable<DOMString>;
};
```

A DOMTokenList object has an associated **token set** (a set), which is initially empty.

A DOMTokenList object also has an associated element and an attribute's local name.

Specifications may define **supported tokens** for a DOMTokenList's associated attribute's local name.

A DOMTokenList object's **validation steps** for a given *token* are:

1. If the associated attribute's local name does not define supported tokens, throw a TypeError.
2. Let *lowercase token* be a copy of *token*, in ASCII lowercase.
3. If *lowercase token* is present in supported tokens, return true.
4. Return false.

A DOMTokenList object's **update steps** are:

1. If the associated element does not have an associated attribute and token set is empty, then return.
2. Set an attribute value for the associated element using associated attribute's local name and the result of running the ordered set serializer for token set.

A DOMTokenList object's **serialize steps** are to return the result of running get an attribute value given the associated element and the associated attribute's local name.

A DOMTokenList object has these attribute change steps for its associated element:

1. If *localName* is associated attribute's local name, *namespace* is null, and *value* is null, then empty token set.
2. Otherwise, if *localName* is associated attribute's local name, *namespace* is null, then set token set to *value*, parsed.

When a DOMTokenList object is created, then:

1. Let *element* be associated element.
2. Let *localName* be associated attribute's local name.
3. Let *value* be the result of getting an attribute value given *element* and *localName*.
4. Run the attribute change steps for *element*, *localName*, *value*, *value*, and null.

For web developers (non-normative)

***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.

***tokenlist* . add(*tokens*…)**

Adds all arguments passed, except those already present.

Throws a "SyntaxError" DOMException if one of the arguments is the empty string.

Throws an "InvalidCharacterError" DOMException if one of the arguments contains any ASCII whitespace.

***tokenlist* . remove(*tokens*…)**

Removes arguments passed, if they are present.

Throws a "SyntaxError" DOMException if one of the arguments is the empty string.

Throws an "InvalidCharacterError" DOMException 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 present. 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" DOMException if *token* is empty.

Throws an "InvalidCharacterError" DOMException if *token* contains any spaces.

***tokenlist* . replace(*token*, *newToken*)**

Replaces *token* with *newToken*.

Returns true if *token* was replaced with *newToken*, and false otherwise.

Throws a "SyntaxError" DOMException if one of the arguments is the empty string.

Throws an "InvalidCharacterError" DOMException if one of the arguments contains any ASCII whitespace.

***tokenlist* . supports(*token*)**

Returns true if *token* is in the associated attribute's supported tokens. Returns false otherwise.

Throws a `TypeError` if the associated attribute has no supported tokens defined.

***tokenlist* . value**

Returns the associated set as string.

Can be set, to change the associated attribute.

The **length** attribute' getter must return this's token set's size.

The object's supported property indices are the numbers in the range zero to object's token set's size minus one, unless token set is empty, in which case there are no supported property indices.

The **`item(index)`** method, when invoked, must run these steps:

1. If *index* is equal to or greater than this's token set's size, then return null.
2. Return this's token set[*index*].

The **`contains(token)`** method, when invoked, must return true if this's token set[*token*] exists, and false otherwise.

The **`add(tokens…)`** method, when invoked, must run these steps:

1. For each *token* in *tokens*:
    1. If *token* is the empty string, then throw a "`SyntaxError`" `DOMException`.
    2. If *token* contains any ASCII whitespace, then throw an "`InvalidCharacterError`" `DOMException`.
2. For each *token* in *tokens*, append *token* to this's token set.
3. Run the update steps.

The **`remove(tokens…)`** method, when invoked, must run these steps:

1. For each *token* in *tokens*:
    1. If *token* is the empty string, then throw a "`SyntaxError`" `DOMException`.
    2. If *token* contains any ASCII whitespace, then throw an "`InvalidCharacterError`" `DOMException`.
2. For each *token* in *tokens*, remove *token* from this's token set.
3. Run the update steps.

The **`toggle(token, force)`** method, when invoked, must run these steps:

1. If *token* is the empty string, then throw a "`SyntaxError`" `DOMException`.
2. If *token* contains any ASCII whitespace, then throw an "`InvalidCharacterError`" `DOMException`.
3. If this's token set[*token*] exists, then:
    1. If *force* is either not given or is false, then remove *token* from this's token set, run the update steps and return false.
    2. Return true.
4. Otherwise, if *force* not given or is true, append *token* to this's token set, run the update steps, and return true.
5. Return false.

Note

The update steps are not always run for `toggle()` for web compatibility.

The **`replace(token, newToken)`** method, when invoked, must run these steps:

1. If either *token* or *newToken* is the empty string, then throw a "`SyntaxError`" `DOMException`.
2. If either *token* or *newToken* contains any ASCII whitespace, then throw an "`InvalidCharacterError`" `DOMException`.
3. If this's token set does not contain *token*, then return false.
4. Replace *token* in this's token set with *newToken*.
5. Run the update steps.
6. Return true.

Note

The update steps are not always run for `replace()` for web compatibility.

The **`supports(token)`** method, when invoked, must run these steps:

1. Let *result* be the return value of validation steps called with *token*.
2. Return *result*.

The **`value`** attribute must return the result of running this's serialize steps.

Setting the `value` attribute must set an attribute value for the associated element using associated attribute's local name and the given value.

8. XPath §

DOM Level 3 XPath defined an API for evaluating *XPath 1.0* expressions. These APIs are widely implemented, but have not been maintained. The interface definitions are maintained here so that they can be updated when *Web IDL* changes. Complete definitions of these APIs remain necessary and such work is tracked and can be contributed to in whatwg/dom#67. [DOM-Level-3-XPath] [XPath] [WEBIDL]

8.1. Interface `XPathResult` §

IDL
```
[Exposed=Window]
interface XPathResult {
  const unsigned short ANY_TYPE = 0;
  const unsigned short NUMBER_TYPE = 1;
  const unsigned short STRING_TYPE = 2;
  const unsigned short BOOLEAN_TYPE = 3;
  const unsigned short UNORDERED_NODE_ITERATOR_TYPE = 4;
  const unsigned short ORDERED_NODE_ITERATOR_TYPE = 5;
  const unsigned short UNORDERED_NODE_SNAPSHOT_TYPE = 6;
  const unsigned short ORDERED_NODE_SNAPSHOT_TYPE = 7;
  const unsigned short ANY_UNORDERED_NODE_TYPE = 8;
  const unsigned short FIRST_ORDERED_NODE_TYPE = 9;

  readonly attribute unsigned short resultType;
  readonly attribute unrestricted double numberValue;
  readonly attribute DOMString stringValue;
  readonly attribute boolean booleanValue;
  readonly attribute Node? singleNodeValue;
  readonly attribute boolean invalidIteratorState;
  readonly attribute unsigned long snapshotLength;

  Node? iterateNext();
  Node? snapshotItem(unsigned long index);
};
```

8.2. Interface `XPathExpression` §

IDL
```
[Exposed=Window]
interface XPathExpression {
  // XPathResult.ANY_TYPE = 0
  XPathResult evaluate(Node contextNode, optional unsigned short type = 0, optional XPathResult?
result = null);
};
```

8.3. Mixin XPathEvaluatorBase §

IDL
```
callback interface XPathNSResolver {
  DOMString? lookupNamespaceURI(DOMString? prefix);
};

interface mixin XPathEvaluatorBase {
  [NewObject] XPathExpression createExpression(DOMString expression, optional XPathNSResolver?
resolver = null);
  XPathNSResolver createNSResolver(Node nodeResolver);
  // XPathResult.ANY_TYPE = 0
  XPathResult evaluate(DOMString expression, Node contextNode, optional XPathNSResolver? resolver =
null, optional unsigned short type = 0, optional XPathResult? result = null);
};
Document includes XPathEvaluatorBase;
```

8.4. Interface XPathEvaluator §

IDL
```
[Exposed=Window]
interface XPathEvaluator {
  constructor();
};

XPathEvaluator includes XPathEvaluatorBase;
```

Note

For historical reasons you can both construct `XPathEvaluator` *and access the same methods on* `Document`.

9. Historical §

As explained in goals this standard is a significant revision of various obsolete DOM specifications. This section enumerates the incompatible changes.

9.1. 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 *DOM Level 3 Events*. The other chapters are defined by the *UI Events* specification. [UIEVENTS]

- 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.
- `Event`'s `timeStamp` attribute is a `DOMHighResTimeStamp` rather than a `DOMTimeStamp`.

9.2. DOM Core §

These are the changes made to the features described in *DOM Level 3 Core*.

`DOMString`, `DOMException`, and `DOMTimeStamp` are now defined in Web IDL.

`DOMStringList` is now defined in HTML.

`hasAttributes()` and `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 web platform. Implementations conforming to this specification will not support them.

Interfaces:

- `DOMConfiguration`
- `DOMError`
- `DOMErrorHandler`
- `DOMImplementationList`
- `DOMImplementationSource`
- `DOMLocator`
- `DOMObject`
- `DOMUserData`
- `Entity`
- `EntityReference`
- `NameList`
- `Notation`
- `TypeInfo`
- `UserDataHandler`

Interface members:

`Node`

- `isSupported`
- `getFeature()`
- `getUserData()`
- `setUserData()`

`Document`

- `createEntityReference()`
- `xmlEncoding`

- `xmlStandalone`
- `xmlVersion`
- `strictErrorChecking`
- `domConfig`
- `normalizeDocument()`
- `renameNode()`

`DOMImplementation`

- `getFeature()`

`Attr`

- `schemaTypeInfo`
- `isId`

`Element`

- `schemaTypeInfo`
- `setIdAttribute()`
- `setIdAttributeNS()`
- `setIdAttributeNode()`

`DocumentType`

- `entities`
- `notations`
- `internalSubset`

`Text`

- `isElementContentWhitespace`
- `replaceWholeText()`

9.3. 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.
- `detach()` is now a no-op.

9.4. 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.
- `nextNode()` and `previousNode()` now throw when invoked from a `NodeFilter` to align with user agents.
- `detach()` is now a no-op.

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, Ahmid *snuggs*, Alex Komoroske, Alex Russell, Alexey Shvayka, Andreu Botella, Anthony Ramine, Arkadiusz Michalski, Arnaud Le Hors, Arun Ranganathan, Björn Höhrmann, Boris Zbarsky, Brandon Payton, Brandon Slade, Brandon Wallace, Brian Kardell, Cameron McCormack, Chris Dumez, Chris Paris, Chris Rebert, Cyrille Tuzi, Daniel Glazman, Darin Fisher, David Bruant, David Flanagan, David Håsäther, David Hyatt, Deepak Sherveghar, Dethe Elza, Dimitri Glazkov, Domenic Denicola, Dominic Cooney, Dominique Hazaël-Massieux, Don Jordan, Doug Schepers, Edgar Chen, Elisée Maurer, Elliott Sprehn, Emilio Cobos Álvarez, Eric Bidelman, Erik Arvidsson, Gary Kacmarcik, Gavin Nicol, Giorgio Liscio, Glen Huang, Glenn Adams, Glenn Maynard, Hajime Morrita, Harald Alvestrand, Hayato Ito, Henri Sivonen, Hongchan Choi, Hunan Rostomyan, Ian Hickson, Igor Bukanov, Jacob Rossi, Jake Archibald, Jake Verbaten, James Graham, James Greene, James Robinson, Jeffrey Yasskin, Jens Lindström, Jesse McCarthy, Jinho Bang, João Eiras, Joe Kesselman, John Atkins, John Dai, Jonas Sicking, Jonathan Kingston, Jonathan Robie, Joris van der Wel, Joshua Bell, J. S. Choi, Jungkee Song, Justin Summerlin, Kagami Sascha Rosylight, 呂康豪 (Kang-Hao Lu), 田村健人 (Kent TAMURA), Kevin Sweeney, Kirill Topolyan, Koji Ishii, Lachlan Hunt, Lauren Wood, Luke Zielinski, Magne Andersson, Majid Valipour, Malte Ubl, Manish Goregaokar, Manish Tripathi, Marcos Caceres, Mark Miller, Martijn van der Ven, Mats Palmgren, Mounir Lamouri, Michael Stramel, Michael™ Smith, Mike Champion, Mike Taylor, Mike West, Ojan Vafai, Oliver Nightingale, Olli Pettay, Ondřej Žára, Peter Sharpe, Philip Jägenstedt, Philippe Le Hégaret, Piers Wombwell, Pierre-Marie Dartus, prosody—Gab Vereable Context(, Rafael Weinstein, Rakina Zata Amni, Richard Bradshaw, Rick Byers, Rick Waldron, Robbert Broersma, Robin Berjon, Roland Steiner, Rune F. Halvorsen, Russell Bicknell, Ruud Steltenpool, Ryosuke Niwa, Sam Dutton, Sam Sneddon, Samuel Giles, Sanket Joshi, Sebastian Mayr, Seo Sanghyeon, Sergey G. Grekhov, Shiki Okasaka, Shinya Kawanaka, Simon Pieters, Stef Busking, Steve Byrne, Stig Halvorsen, Tab Atkins, Takashi Sakamoto, Takayoshi Kochi, Theresa O'Connor, Theodore Dubois, *timeless*, Timo Tijhof, Tobie Langel, Tom Pixley, Travis Leithead, Trevor Rowbotham, *triple-underscore*, Veli Şenol, Vidur Apparao, Warren He, Xidorn Quan, Yash Handa, Yehuda Katz, Yoav Weiss, Yoichi Osato, Yoshinori Sano, and Zack Weinberg for being awesome!

This standard is written by Anne van Kesteren (Mozilla, annevk@annevk.nl) with substantial contributions from Aryeh Gregor (ayg@aryeh.name) and Ms2ger (ms2ger@gmail.com).

Index §

Terms defined by this specification §

- XPathEvaluator(), in §8.4
- XPathEvaluatorBase, in §8.3
- XPathExpression, in §8.2
- XPathNSResolver, in §8.3
- XPathResult, in §8.1

Terms defined by reference §

- [CONSOLE] defines the following terms:
    - report a warning to the console
- [CSSOM-VIEW] defines the following terms:
    - getBoundingClientRect()
    - getClientRects()
- [ECMASCRIPT] defines the following terms:
    - realm
    - surrounding agent
- [ENCODING] defines the following terms:
    - encoding
    - name
    - utf-8
- [HR-TIME] defines the following terms:
    - DOMHighResTimeStamp
    - clock resolution
    - time origin
- [HTML] defines the following terms:
    - BeforeUnloadEvent
    - CEReactions
    - DOMStringList
    - DragEvent
    - EventHandler
    - HTMLElement
    - HTMLHtmlElement
    - HTMLSlotElement
    - HTMLUnknownElement
    - HashChangeEvent
    - MessageEvent
    - StorageEvent
    - Window
    - area
    - associated document
    - body
    - browsing context (for Document)
    - click()
    - constructor
    - current global object
    - custom element constructor
    - customized built-in element
    - disable shadow
    - document base url
    - enqueue a custom element callback reaction
    - enqueue a custom element upgrade reaction
    - event handler
    - event handler event type
    - event handler idl attribute
    - global object

    - head
    - html
    - html parser
    - in parallel
    - input
    - local name
    - look up a custom element definition
    - microtask
    - name
    - opaque origin
    - origin
    - queue a microtask
    - relevant agent
    - relevant global object
    - relevant realm
    - report the exception
    - script
    - similar-origin window agent
    - slot
    - slotchange
    - template
    - title
    - try to upgrade an element
    - upgrade an element
    - valid custom element name
- [INFRA] defines the following terms:
    - append (for set)
    - ascii case-insensitive
    - ascii lowercase
    - ascii uppercase
    - ascii whitespace
    - break
    - clone
    - code unit
    - concatenation
    - contain
    - continue
    - empty
    - enqueue
    - exist (for map)
    - for each (for map)
    - html namespace
    - identical to
    - insert
    - is empty
    - is not empty
    - length
    - list
    - map
    - ordered set
    - prepend
    - queue
    - remove
    - replace (for set)
    - set (for map)
    - size
    - split on ascii whitespace
    - struct

- svg namespace
  - tuple
  - xml namespace
  - xmlns namespace
- [SELECTORS4] defines the following terms:
  - :defined
  - :scope element
  - match a selector against a tree
  - match a selector against an element
  - parse a selector
  - scoping root
- [SERVICE-WORKERS] defines the following terms:
  - ServiceWorkerGlobalScope
  - has ever been evaluated flag
  - script resource
  - service worker
  - service worker events
  - set of event types to handle
- [UIEVENTS] defines the following terms:
  - CompositionEvent
  - FocusEvent
  - KeyboardEvent
  - MouseEvent
  - UIEvent
  - detail
- [URL] defines the following terms:
  - url
  - url serializer
- [WEBIDL] defines the following terms:
  - AbortError
  - DOMException
  - DOMString
  - DOMTimeStamp
  - Exposed
  - HierarchyRequestError
  - InUseAttributeError
  - IndexSizeError
  - InvalidCharacterError
  - InvalidNodeTypeError
  - InvalidStateError
  - LegacyNullToEmptyString
  - LegacyUnenumerableNamedProperties
  - LegacyUnforgeable
  - NamespaceError
  - NewObject
  - NotFoundError
  - NotSupportedError
  - PutForwards
  - Replaceable
  - SameObject
  - SyntaxError
  - USVString
  - Unscopable
  - WrongDocumentError
  - a new promise
  - associated realm
  - boolean
  - call a user object's operation

- construct
- converted to an idl value
- dictionary
- identifier
- invoke
- reject
- resolve
- short
- supported property indices
- supported property names
- this
- throw
- unrestricted double
- unsigned long
- unsigned short

References §

Normative References §

[CONSOLE]

Dominic Farolino; Robert Kowalski; Terin Stock. Console Standard. Living Standard. URL: https://console.spec.whatwg.org/

[DEVICE-ORIENTATION]

Rich Tibbett; et al. DeviceOrientation Event Specification. URL: https://w3c.github.io/deviceorientation/

[ECMASCRIPT]

ECMAScript Language Specification. URL: https://tc39.es/ecma262/

[ENCODING]

Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[HR-TIME]

Ilya Grigorik. High Resolution Time Level 2. URL: https://w3c.github.io/hr-time/

[HTML]

Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[SELECTORS4]

Elika Etemad; Tab Atkins Jr.. Selectors Level 4. URL: https://drafts.csswg.org/selectors/

[SERVICE-WORKERS]

Alex Russell; et al. Service Workers 1. URL: https://w3c.github.io/ServiceWorker/

[TOUCH-EVENTS]

Doug Schepers; et al. Touch Events. URL: https://w3c.github.io/touch-events/

[UIEVENTS]

Gary Kacmarcik; Travis Leithead; Doug Schepers. UI Events. URL: https://w3c.github.io/uievents/

[URL]

Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[WEBIDL]

Boris Zbarsky. Web IDL. URL: https://heycam.github.io/webidl/

[XML]

Tim Bray; et al. Extensible Markup Language (XML) 1.0 (Fifth Edition). 26 November 2008. REC. URL: https://www.w3.org/TR/xml/

[XML-NAMES]

Tim Bray; et al. Namespaces in XML 1.0 (Third Edition). 8 December 2009. REC. URL: https://www.w3.org/TR/xml-names/

Informative References §

[CSSOM-VIEW]

Simon Pieters. CSSOM View Module. URL: https://drafts.csswg.org/cssom-view/

[DOM-Level-2-Traversal-Range]

Joseph Kesselman; et al. Document Object Model (DOM) Level 2 Traversal and Range Specification. 13 November 2000. REC. URL: https://www.w3.org/TR/DOM-Level-2-Traversal-Range/

[DOM-Level-3-Core]

Arnaud Le Hors; et al. Document Object Model (DOM) Level 3 Core Specification. 7 April 2004. REC. URL: https://www.w3.org/TR/DOM-Level-3-Core/

[DOM-Level-3-XPath]

Ray Whitmer. Document Object Model (DOM) Level 3 XPath Specification. 26 February 2004. NOTE. URL: https://www.w3.org/TR/DOM-Level-3-XPath/

[DOM-Parsing]

Travis Leithead. DOM Parsing and Serialization. URL: https://w3c.github.io/DOM-Parsing/

[ELEMENTTRAVERSAL]

Doug Schepers; Robin Berjon. Element Traversal Specification. 22 December 2008. REC. URL: https://www.w3.org/TR/ElementTraversal/

[FULLSCREEN]

Philip Jägenstedt. Fullscreen API Standard. Living Standard. URL: https://fullscreen.spec.whatwg.org/

[INDEXEDDB]

Nikunj Mehta; et al. Indexed Database API. URL: https://w3c.github.io/IndexedDB/

[SELECTORS-API2]

Lachlan Hunt. Selectors API Level 2. URL: https://dev.w3.org/2006/webapi/selectors-api2/

[UIEVENTS-20031107]

Philippe Le Hégaret; Tim Pixley. Document Object Model (DOM) Level 3 Events Specification. URL: https://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107

[XPath]

James Clark; Steven DeRose. XML Path Language (XPath) Version 1.0. 16 November 1999. REC. URL: https://www.w3.org/TR/xpath-10/

IDL Index §

```
IDL
[Exposed=(Window,Worker,AudioWorklet)]
interface Event {
  constructor(DOMString type, optional EventInit eventInitDict = {});

  readonly attribute DOMString type;
  readonly attribute EventTarget? target;
  readonly attribute EventTarget? srcElement; // historical
  readonly attribute EventTarget? currentTarget;
  sequence<EventTarget> composedPath();

  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();
           attribute boolean cancelBubble; // historical alias of .stopPropagation
  void stopImmediatePropagation();

  readonly attribute boolean bubbles;
  readonly attribute boolean cancelable;
           attribute boolean returnValue;  // historical
  void preventDefault();
  readonly attribute boolean defaultPrevented;
  readonly attribute boolean composed;

  [LegacyUnforgeable] readonly attribute boolean isTrusted;
  readonly attribute DOMHighResTimeStamp timeStamp;

  void initEvent(DOMString type, optional boolean bubbles = false, optional boolean cancelable =
false); // historical
};

dictionary EventInit {
  boolean bubbles = false;
  boolean cancelable = false;
  boolean composed = false;
};

partial interface Window {
  [Replaceable] readonly attribute any event; // historical
};

[Exposed=(Window,Worker)]
interface CustomEvent : Event {
  constructor(DOMString type, optional CustomEventInit eventInitDict = {});

  readonly attribute any detail;

  void initCustomEvent(DOMString type, optional boolean bubbles = false, optional boolean cancelable =
false, optional any detail = null); // historical
};

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

```
};

[Exposed=(Window,Worker,AudioWorklet)]
interface EventTarget {
  constructor();

  void addEventListener(DOMString type, EventListener? callback, optional (AddEventListenerOptions or
boolean) options = {});
  void removeEventListener(DOMString type, EventListener? callback, optional (EventListenerOptions or
boolean) options = {});
  boolean dispatchEvent(Event event);
};

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

dictionary EventListenerOptions {
  boolean capture = false;
};

dictionary AddEventListenerOptions : EventListenerOptions {
  boolean passive = false;
  boolean once = false;
};

[Exposed=(Window,Worker)]
interface AbortController {
  constructor();

  [SameObject] readonly attribute AbortSignal signal;

  void abort();
};

[Exposed=(Window,Worker)]
interface AbortSignal : EventTarget {
  readonly attribute boolean aborted;

  attribute EventHandler onabort;
};
interface mixin NonElementParentNode {
  Element? getElementById(DOMString elementId);
};
Document includes NonElementParentNode;
DocumentFragment includes NonElementParentNode;

interface mixin DocumentOrShadowRoot {
};
Document includes DocumentOrShadowRoot;
ShadowRoot includes DocumentOrShadowRoot;

interface mixin ParentNode {
  [SameObject] readonly attribute HTMLCollection children;
  readonly attribute Element? firstElementChild;
  readonly attribute Element? lastElementChild;
  readonly attribute unsigned long childElementCount;
  [CEReactions, Unscopable] void prepend((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void append((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void replaceChildren((Node or DOMString)... nodes);

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

interface mixin NonDocumentTypeChildNode {
  readonly attribute Element? previousElementSibling;
  readonly attribute Element? nextElementSibling;
};
Element includes NonDocumentTypeChildNode;
CharacterData includes NonDocumentTypeChildNode;

interface mixin ChildNode {
  [CEReactions, Unscopable] void before((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void after((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void replaceWith((Node or DOMString)... nodes);
  [CEReactions, Unscopable] void remove();
};
DocumentType includes ChildNode;
Element includes ChildNode;
CharacterData includes ChildNode;

interface mixin Slottable {
  readonly attribute HTMLSlotElement? assignedSlot;
};
Element includes Slottable;
Text includes Slottable;

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

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

[Exposed=Window]
interface MutationObserver {
  constructor(MutationCallback callback);

  void observe(Node target, optional 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;
};

[Exposed=Window]
interface MutationRecord {
  readonly attribute DOMString type;
  [SameObject] 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;
};

[Exposed=Window]
interface Node : EventTarget {
  const unsigned short ELEMENT_NODE = 1;
  const unsigned short ATTRIBUTE_NODE = 2;
  const unsigned short TEXT_NODE = 3;
  const unsigned short CDATA_SECTION_NODE = 4;
  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 USVString baseURI;

  readonly attribute boolean isConnected;
  readonly attribute Document? ownerDocument;
  Node getRootNode(optional GetRootNodeOptions options = {});
  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;

  [CEReactions] attribute DOMString? nodeValue;
  [CEReactions] attribute DOMString? textContent;
  [CEReactions] void normalize();
  [CEReactions, NewObject] Node cloneNode(optional boolean deep = false);
  boolean isEqualNode(Node? otherNode);
  boolean isSameNode(Node? otherNode); // historical alias of ===

  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);

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

dictionary GetRootNodeOptions {
  boolean composed = false;
};

[Exposed=Window]
interface Document : Node {
  constructor();

  [SameObject] readonly attribute DOMImplementation implementation;
  readonly attribute USVString URL;
  readonly attribute USVString documentURI;
  readonly attribute DOMString compatMode;
  readonly attribute DOMString characterSet;
  readonly attribute DOMString charset; // historical alias of .characterSet
  readonly attribute DOMString inputEncoding; // historical alias of .characterSet
  readonly attribute DOMString contentType;

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

  [CEReactions, NewObject] Element createElement(DOMString localName, optional (DOMString or
ElementCreationOptions) options = {});
  [CEReactions, NewObject] Element createElementNS(DOMString? namespace, DOMString qualifiedName,
optional (DOMString or ElementCreationOptions) options = {});
  [NewObject] DocumentFragment createDocumentFragment();
  [NewObject] Text createTextNode(DOMString data);
  [NewObject] CDATASection createCDATASection(DOMString data);
  [NewObject] Comment createComment(DOMString data);
  [NewObject] ProcessingInstruction createProcessingInstruction(DOMString target, DOMString data);

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

  [NewObject] Attr createAttribute(DOMString localName);
  [NewObject] Attr createAttributeNS(DOMString? namespace, DOMString qualifiedName);

  [NewObject] Event createEvent(DOMString interface); // historical

  [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 {};

dictionary ElementCreationOptions {
  DOMString is;
};

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

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

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

[Exposed=Window]
interface DocumentFragment : Node {
  constructor();
};

[Exposed=Window]
interface ShadowRoot : DocumentFragment {
  readonly attribute ShadowRootMode mode;
  readonly attribute Element host;
  attribute EventHandler onslotchange;
};

enum ShadowRootMode { "open", "closed" };

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

  [CEReactions] attribute DOMString id;
  [CEReactions] attribute DOMString className;
  [SameObject, PutForwards=value] readonly attribute DOMTokenList classList;
  [CEReactions, Unscopable] attribute DOMString slot;

  boolean hasAttributes();
  [SameObject] readonly attribute NamedNodeMap attributes;
  sequence<DOMString> getAttributeNames();
  DOMString? getAttribute(DOMString qualifiedName);
  DOMString? getAttributeNS(DOMString? namespace, DOMString localName);
  [CEReactions] void setAttribute(DOMString qualifiedName, DOMString value);
  [CEReactions] void setAttributeNS(DOMString? namespace, DOMString qualifiedName, DOMString value);
  [CEReactions] void removeAttribute(DOMString qualifiedName);
  [CEReactions] void removeAttributeNS(DOMString? namespace, DOMString localName);
  [CEReactions] boolean toggleAttribute(DOMString qualifiedName, optional boolean force);
  boolean hasAttribute(DOMString qualifiedName);
  boolean hasAttributeNS(DOMString? namespace, DOMString localName);

  Attr? getAttributeNode(DOMString qualifiedName);
  Attr? getAttributeNodeNS(DOMString? namespace, DOMString localName);
  [CEReactions] Attr? setAttributeNode(Attr attr);
  [CEReactions] Attr? setAttributeNodeNS(Attr attr);
  [CEReactions] Attr removeAttributeNode(Attr attr);

  ShadowRoot attachShadow(ShadowRootInit init);
  readonly attribute ShadowRoot? shadowRoot;

  Element? closest(DOMString selectors);
  boolean matches(DOMString selectors);
  boolean webkitMatchesSelector(DOMString selectors); // historical alias of .matches

  HTMLCollection getElementsByTagName(DOMString qualifiedName);
  HTMLCollection getElementsByTagNameNS(DOMString? namespace, DOMString localName);
  HTMLCollection getElementsByClassName(DOMString classNames);

  [CEReactions] Element? insertAdjacentElement(DOMString where, Element element); // historical
  void insertAdjacentText(DOMString where, DOMString data); // historical
};

dictionary ShadowRootInit {
  required ShadowRootMode mode;
  boolean delegatesFocus = false;
};

[Exposed=Window,
 LegacyUnenumerableNamedProperties]
interface NamedNodeMap {
  readonly attribute unsigned long length;
  getter Attr? item(unsigned long index);
  getter Attr? getNamedItem(DOMString qualifiedName);
  Attr? getNamedItemNS(DOMString? namespace, DOMString localName);
  [CEReactions] Attr? setNamedItem(Attr attr);
  [CEReactions] Attr? setNamedItemNS(Attr attr);
  [CEReactions] Attr removeNamedItem(DOMString qualifiedName);
  [CEReactions] Attr removeNamedItemNS(DOMString? namespace, DOMString localName);
};

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

  readonly attribute Element? ownerElement;

  readonly attribute boolean specified; // useless; always returns true
};
[Exposed=Window]
interface CharacterData : Node {
  attribute [LegacyNullToEmptyString] 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);
};

[Exposed=Window]
interface Text : CharacterData {
  constructor(optional DOMString data = "");

  [NewObject] Text splitText(unsigned long offset);
  readonly attribute DOMString wholeText;
};

[Exposed=Window]
interface CDATASection : Text {
};
[Exposed=Window]
interface ProcessingInstruction : CharacterData {
  readonly attribute DOMString target;
};
[Exposed=Window]
interface Comment : CharacterData {
  constructor(optional DOMString data = "");
};

[Exposed=Window]
interface AbstractRange {
  readonly attribute Node startContainer;
  readonly attribute unsigned long startOffset;
  readonly attribute Node endContainer;
  readonly attribute unsigned long endOffset;
  readonly attribute boolean collapsed;
};

dictionary StaticRangeInit {
  required Node startContainer;
  required unsigned long startOffset;
  required Node endContainer;
  required unsigned long endOffset;
};

[Exposed=Window]
interface StaticRange : AbstractRange {
  constructor(StaticRangeInit init);
};

[Exposed=Window]
interface Range : AbstractRange {
  constructor();

  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);

  [CEReactions] void deleteContents();
  [CEReactions, NewObject] DocumentFragment extractContents();
  [CEReactions, NewObject] DocumentFragment cloneContents();
  [CEReactions] void insertNode(Node node);
  [CEReactions] 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;
};

[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();
};

[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();
};
[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;
  const unsigned long SHOW_TEXT = 0x4;
  const unsigned long SHOW_CDATA_SECTION = 0x8;
  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);
};

[Exposed=Window]
interface DOMTokenList {
  readonly attribute unsigned long length;
  getter DOMString? item(unsigned long index);
  boolean contains(DOMString token);
  [CEReactions] void add(DOMString... tokens);
  [CEReactions] void remove(DOMString... tokens);
  [CEReactions] boolean toggle(DOMString token, optional boolean force);
  [CEReactions] boolean replace(DOMString token, DOMString newToken);
  boolean supports(DOMString token);
  [CEReactions] stringifier attribute DOMString value;
  iterable<DOMString>;
};
```

```
[Exposed=Window]
interface XPathResult {
  const unsigned short ANY_TYPE = 0;
  const unsigned short NUMBER_TYPE = 1;
  const unsigned short STRING_TYPE = 2;
  const unsigned short BOOLEAN_TYPE = 3;
  const unsigned short UNORDERED_NODE_ITERATOR_TYPE = 4;
  const unsigned short ORDERED_NODE_ITERATOR_TYPE = 5;
  const unsigned short UNORDERED_NODE_SNAPSHOT_TYPE = 6;
  const unsigned short ORDERED_NODE_SNAPSHOT_TYPE = 7;
  const unsigned short ANY_UNORDERED_NODE_TYPE = 8;
  const unsigned short FIRST_ORDERED_NODE_TYPE = 9;

  readonly attribute unsigned short resultType;
  readonly attribute unrestricted double numberValue;
  readonly attribute DOMString stringValue;
  readonly attribute boolean booleanValue;
  readonly attribute Node? singleNodeValue;
  readonly attribute boolean invalidIteratorState;
  readonly attribute unsigned long snapshotLength;

  Node? iterateNext();
  Node? snapshotItem(unsigned long index);
};

[Exposed=Window]
interface XPathExpression {
  // XPathResult.ANY_TYPE = 0
  XPathResult evaluate(Node contextNode, optional unsigned short type = 0, optional XPathResult?
result = null);
};

callback interface XPathNSResolver {
  DOMString? lookupNamespaceURI(DOMString? prefix);
};

interface mixin XPathEvaluatorBase {
  [NewObject] XPathExpression createExpression(DOMString expression, optional XPathNSResolver?
resolver = null);
  XPathNSResolver createNSResolver(Node nodeResolver);
  // XPathResult.ANY_TYPE = 0
  XPathResult evaluate(DOMString expression, Node contextNode, optional XPathNSResolver? resolver =
null, optional unsigned short type = 0, optional XPathResult? result = null);
};
Document includes XPathEvaluatorBase;

[Exposed=Window]
interface XPathEvaluator {
  constructor();
};

XPathEvaluator includes XPathEvaluatorBase;
```