

UI Events

W3C Working Draft, 04 August 2016



This version:

<https://www.w3.org/TR/2016/WD-UIEvents-20160804/>

Latest version:

<https://www.w3.org/TR/UIEvents/>

Editor's Draft:

<https://github.com/w3c/UIEvents/>

Previous Versions:

<https://www.w3.org/TR/2015/WD-UIEvents-20151215/>

<https://www.w3.org/TR/2015/WD-UIEvents-20150428/>

<https://www.w3.org/TR/2015/WD-UIEvents-20150319/>

Issue Tracking:

[GitHub](#)

Editors:

[Gary Kacmarcik](#) (Google)

[Travis Leithead](#) (Microsoft)

Copyright © 2016 W3C® (MIT, ERCIM, Keio, Beihang). W3C [liability](#), [trademark](#) and [document use](#) rules apply.

Abstract

This specification defines UI Events which extend the DOM Event objects defined in [\[DOM\]](#). UI Events are those typically implemented by visual user agents for handling user interaction such as mouse and keyboard input.

Status of This Document

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

The management and production of this document follows a relatively complex setup. Details are provided as part of the introduction.

This document was published by the [Web Platform Working Group](#) as a Working Draft. Feedback and comments on this specification are welcome. Please use [Github issues](#). Historical discussions can be found in the [public-webapps@w3.org archives](#).

Publication as a Working Draft does not imply endorsement by the [W3C](#) Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

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

This document is governed by the [1 September 2015 W3C Process Document](#).

Table of Contents

1	Introduction
1.1	Overview
1.2	Conformance
1.2.1	Web browsers and other dynamic or interactive user agents
1.2.2	Authoring tools
1.2.3	Content authors and content
1.2.4	Specifications and host languages
2	Stylistic Conventions
3	DOM Event Architecture
3.1	Event dispatch and DOM event flow
3.2	Default actions and cancelable events
3.3	Synchronous and asynchronous events
3.4	Trusted events
3.5	Activation triggers and behavior
3.6	Constructing Mouse and Keyboard Events
4	Event Types
4.1	User Interface Events
4.1.1	Interface UIEvent
4.1.2	UI Event Types
4.1.2.1	load
4.1.2.2	unload
4.1.2.3	abort
4.1.2.4	error
4.1.2.5	select
4.2	Focus Events
4.2.1	Interface FocusEvent
4.2.2	Focus Event Order

4.2.3	Document Focus and Focus Context
4.2.4	Focus Event Types
4.2.4.1	blur
4.2.4.2	focus
4.2.4.3	focusin
4.2.4.4	focusout
4.3	Mouse Events
4.3.1	Interface MouseEvent
4.3.2	Event Modifier Initializers
4.3.3	Mouse Event Order
4.3.4	Mouse Event Types
4.3.4.1	click
4.3.4.2	dblclick
4.3.4.3	mousedown
4.3.4.4	mouseenter
4.3.4.5	mouseleave
4.3.4.6	mousemove
4.3.4.7	mouseout
4.3.4.8	mouseover
4.3.4.9	mouseup
4.4	Wheel Events
4.4.1	Interface WheelEvent
4.4.2	Wheel Event Types
4.4.2.1	wheel
4.5	Input Events
4.5.1	Interface InputEvent
4.5.2	Input Event Order
4.5.3	Input Event Types
4.5.3.1	beforeinput
4.5.3.2	input
4.6	Keyboard Events
4.6.1	Interface KeyboardEvent
4.6.2	Keyboard Event Key Location
4.6.3	Keyboard Event Order
4.6.4	Keyboard Event Types
4.6.4.1	keydown
4.6.4.2	keyup
4.7	Composition Events
4.7.1	Interface CompositionEvent
4.7.2	Composition Event Order
4.7.3	Handwriting Recognition Systems
4.7.4	Canceling Composition Events
4.7.5	Key Events During Composition

- 4.7.6 Input Events During Composition
- 4.7.7 Composition Event Types
 - 4.7.7.1 compositionstart
 - 4.7.7.2 compositionupdate
 - 4.7.7.3 compositionend

5 Keyboard events and key values

- 5.1 Keyboard Input
 - 5.1.1 Key Legends
- 5.2 Key codes
 - 5.2.1 Motivation for the `code` Attribute
 - 5.2.2 The Relationship Between `key` and `code`
 - 5.2.3 `code` Examples
 - 5.2.4 `code` and Virtual Keyboards
- 5.3 Keyboard Event `key` Values
 - 5.3.1 Key Values and Unicode
 - 5.3.2 Modifier keys
 - 5.3.3 Dead keys
 - 5.3.4 Input Method Editors
 - 5.3.4.1 Input Method Editor mode keys
 - 5.3.5 Default actions and cancelable keyboard events
 - 5.3.6 Guidelines for selecting key values

6 Legacy Event Initializers

- 6.1 Legacy Event Initializer Interfaces
 - 6.1.1 Initializers for interface `UIEvent`
 - 6.1.2 Initializers for interface `MouseEvent`
 - 6.1.3 Initializers for interface `WheelEvent`
 - 6.1.4 Initializers for interface `KeyboardEvent`
 - 6.1.5 Initializers for interface `CompositionEvent`

7 Legacy Key Attributes

- 7.1 Legacy `KeyboardEvent` supplemental interface
 - 7.1.1 Interface `KeyboardEvent` (supplemental)
 - 7.1.2 Interface `KeyboardEventInit` (supplemental)
- 7.2 Legacy key models
 - 7.2.1 How to determine `keyCode` for `keydown` and `keyup` events
 - 7.2.2 How to determine `keyCode` for `keypress` events
 - 7.2.3 Fixed virtual key codes
 - 7.2.4 Optionally fixed virtual key codes

8 Legacy Event Types

- 8.1 Legacy `UIEvent` events
 - 8.1.1 Legacy `UIEvent` event types

8.1.1.1	DOMActivate
8.1.2	Activation event order
8.2	Legacy FocusEvent events
8.2.1	Legacy FocusEvent event types
8.2.1.1	DOMFocusIn
8.2.1.2	DOMFocusOut
8.2.2	Legacy FocusEvent event order
8.3	Legacy KeyboardEvent events
8.3.1	Legacy KeyboardEvent event types
8.3.1.1	keypress
8.3.2	keypress event order
8.4	Legacy MutationEvent events
8.4.1	Interface MutationEvent
8.4.2	Legacy MutationEvent event types
8.4.2.1	DOMAttrModified
8.4.2.2	DOMCharacterDataModified
8.4.2.3	DOMNodeInserted
8.4.2.4	DOMNodeInsertedIntoDocument
8.4.2.5	DOMNodeRemoved
8.4.2.6	DOMNodeRemovedFromDocument
8.4.2.7	DOMSubtreeModified
9	Extending Events
9.1	Introduction
9.2	Custom Events
9.3	Implementation-Specific Extensions
9.3.1	Known Implementation-Specific Prefixes
10	Security Considerations
11	Changes
11.1	Changes between DOM Level 2 Events and UI Events
11.1.1	Changes to DOM Level 2 event flow
11.1.2	Changes to DOM Level 2 event types
11.1.3	Changes to DOM Level 2 Events interfaces
11.1.4	New Interfaces
11.2	Changes between different drafts of UI Events
12	Acknowledgements
13	Glossary
	Index
	Terms defined by this specification

Terms defined by reference

References

Normative References

Informative References

IDL Index

1. Introduction

1.1. Overview

UI Events is designed with two main goals. The first goal is the design of an [event](#) system which allows registration of event listeners and describes event flow through a tree structure. Additionally, the specification will provide standard modules of events for user interface control and document mutation notifications, including defined contextual information for each of these event modules.

The second goal of UI Events is to provide a common subset of the current event systems used in existing browsers. This is intended to foster interoperability of existing scripts and content. It is not expected that this goal will be met with full backwards compatibility. However, the specification attempts to achieve this when possible.

1.2. Conformance

This section is normative.

Within this specification, the key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” are to be interpreted as described in [\[RFC2119\]](#).

This specification is to be understood in the context of the DOM Level 3 Core specification [\[DOM-Level-3-Core\]](#) and the general considerations for DOM implementations apply. For example, handling of [namespace URIs](#) is discussed in [XML Namespaces](#). For additional information about [conformance](#), please see the DOM Level 3 Core specification [\[DOM-Level-3-Core\]](#). A [user agent](#) is not required to conform to the entirety of another specification in order to conform to this specification, but it **MUST** conform to the specific parts of any other specification which are called out in this specification (e.g., a conforming UI Events [user agent](#) **MUST** support the `DOMString` data type as defined in [\[\[Web IDL\]\]](#), but need not support every method or data type defined in [\[\[Web IDL\]\]](#) in order to conform to UI Events).

This specification defines several classes of conformance for different [user agents](#), specifications, and content authors:

1.2.1. Web browsers and other dynamic or interactive [user agents](#)

A dynamic or interactive [user agent](#), referred to here as a “browser” (be it a Web browser, AT (Accessibility Technology) application, or other similar program), conforms to UI Events if it supports:

- the Core module defined in [\[DOM-Level-3-Core\]](#)
- the [§3.1 Event dispatch and DOM event flow](#) mechanism
- all the interfaces and events with their associated methods, attributes, and semantics defined in this specification with the exception of those marked as [deprecated](#) (a conforming user agent MAY implement the deprecated interfaces, events, or APIs for backwards compatibility, but is not required to do so in order to be conforming)
- the complete set of key and code values defined in [\[UIEvents-Key\]](#) and [\[UIEvents-Code\]](#) (subject to platform availability), and
- all other normative requirements defined in this specification.

A conforming browser MUST [dispatch](#) events appropriate to the given [EventTarget](#) when the conditions defined for that [event type](#) have been met.

A browser conforms specifically to UI Events if it implements the interfaces and related [event types](#) specified in [§4 Event Types](#).

A conforming browser MUST support scripting, declarative interactivity, or some other means of detecting and dispatching events in the manner described by this specification, and MUST support the APIs specified for that [event type](#).

In addition to meeting all other conformance criteria, a conforming browser MAY implement features of this specification marked as [deprecated](#), for backwards compatibility with existing content, but such implementation is discouraged.

A conforming browser MAY also support features not found in this specification, but which use the [§3.1 Event dispatch and DOM event flow](#) mechanism, interfaces, events, or other features defined in this specification, and MAY implement additional interfaces and [event types](#) appropriate to that implementation. Such features can be later standardized in future specifications.

A browser which does not conform to all required portions of this specification MUST NOT claim conformance to UI Events. Such an implementation which does conform to portions of this specification MAY claim conformance to those specific portions.

A conforming browser MUST also be a *conforming implementation* of the IDL fragments in this specification, as described in the Web IDL specification [\[WebIDL\]](#).

1.2.2. Authoring tools

A content authoring tool conforms to UI Events if it produces content which uses the [event types](#) and [§3.1 Event dispatch and DOM event flow](#) model, consistent in a manner as defined in this specification.

A content authoring tool MUST NOT claim conformance to UI Events for content it produces which uses features of this specification marked as [deprecated](#) in this specification.

A conforming content authoring tool SHOULD provide to the content author a means to use all [event types](#) and interfaces appropriate to all [host languages](#) in the content document being produced.

1.2.3. Content authors and content

A content author creates conforming UI Events content if that content uses the [event types](#) and [§3.1 Event dispatch and DOM event flow](#) model, consistent in a manner as defined in this specification.

A content author SHOULD NOT use features of this specification marked as [deprecated](#), but SHOULD rely instead upon replacement mechanisms defined in this specification and elsewhere.

Conforming content MUST use the semantics of the interfaces and [event types](#) as described in this specification.

NOTE:

Content authors are advised to follow best practices as described in [accessibility](#) and [internationalization](#) guideline specifications.

1.2.4. Specifications and host languages




A specification or [host language](#) conforms to UI Events if it references and uses the [§3.1 Event dispatch and DOM event flow](#) mechanism, interfaces, events, or other features defined in [\[DOM\]](#), and does not extend these features in incompatible ways.

A specification or [host language](#) conforms specifically to UI Events if it references and uses the interfaces and related [event types](#) specified in [§4 Event Types](#). A conforming specification MAY define additional interfaces and [event types](#) appropriate to that specification, or MAY extend the UI Events interfaces and [event types](#) in a manner that does not contradict or conflict with the definitions of those interfaces and [event types](#) in this specification.

Specifications or [host languages](#) which reference UI Events SHOULD NOT use or recommend features of this specification marked as [deprecated](#), but SHOULD use or recommend the indicated replacement for that the feature (if available).

2. Stylistic Conventions

This specification follows the [Proposed W3C Specification Conventions](#), with the following supplemental additions:

- The [key cap](#) printed on a key is shown as ,  or . This is used to refer to a key from the user's perspective without regard for the [key](#) and [code](#) values in the generated [KeyboardEvent](#).
- Glyphs representing character are shown as: "𐤃".

- Unicode character encodings are shown as: `\u003d`.
- Names of key values generated by a key press (i.e., the value of `KeyboardEvent.key`) are shown as: `"ArrowDown"`, `"="`, `"q"` or `"Q"`.
- Names of key codes associated with the physical keys (i.e., the value of `KeyboardEvent.code`) are shown as: `"ArrowDown"`, `"Equal"` or `"KeyQ"`.

In addition, certain terms are used in this specification with particular meanings. The term “implementation” applies to a browser, content authoring tool, or other [user agent](#) that implements this specification, while a content author is a person who writes script or code that takes advantage of the interfaces, methods, attributes, events, and other features described in this specification in order to make Web applications, and a user is the person who uses those Web applications in an implementation.

And finally:

NOTE:

This is a note.

This is an open issue.

⚠Warning! This is a warning.

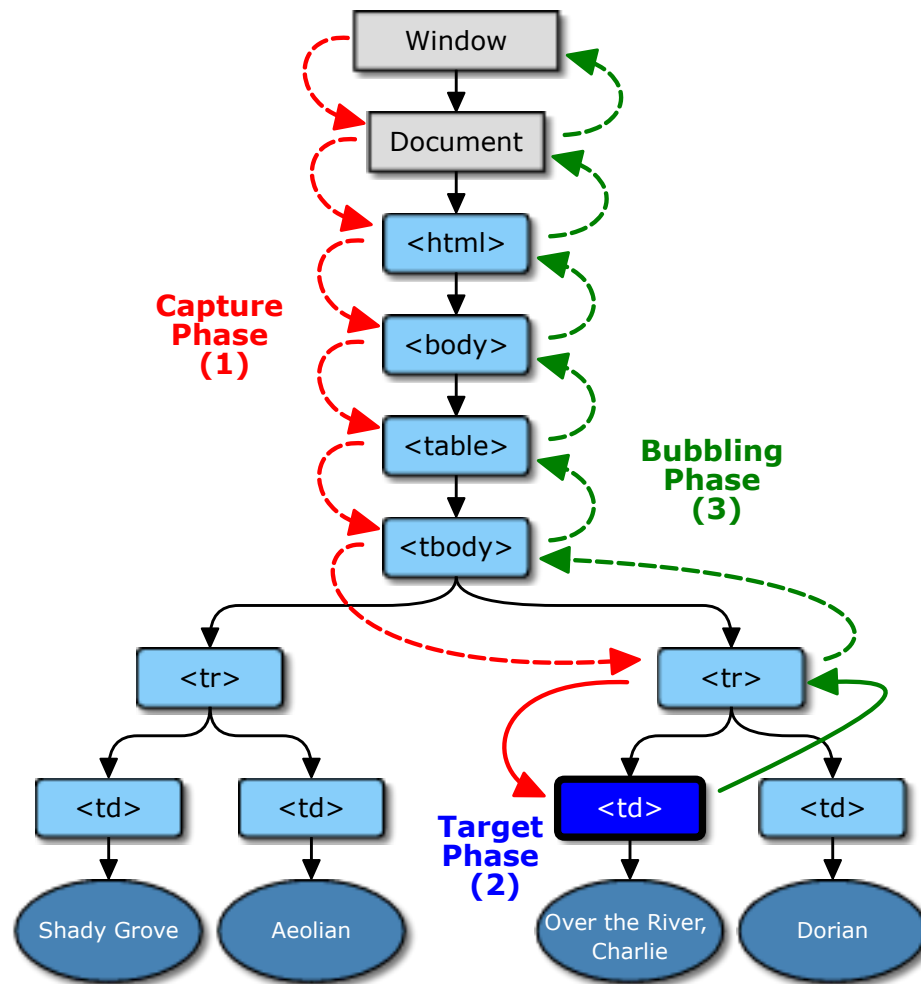
```
interface Example {
    // This is an IDL definition.
};
```

3. DOM Event Architecture

This section is non-normative. Refer to [\[DOM\]](#) for a normative description of the DOM event architecture

3.1. Event dispatch and DOM event flow

This section gives a brief overview of the event [dispatch](#) mechanism and describes how events propagate through the DOM tree. Applications can dispatch event objects using the `dispatchEvent()` method, and the event object will propagate through the DOM tree as determined by the DOM event flow.



Graphical representation of an event dispatched in a DOM tree using the DOM event flow

Event objects are dispatched to an [event target](#). But before dispatch can begin, the event object's [propagation path](#) must first be determined.

The [propagation path](#) is an ordered list of [current event targets](#) through which the event passes. This propagation path reflects the hierarchical tree structure of the document. The last item in the list is the [event target](#), and the preceding items in the list are referred to as the *target's ancestors*, with the immediately preceding item as the *target's parent*.

Once the [propagation path](#) has been determined, the event object passes through one or more [event phases](#). There are three event phases: [capture phase](#), [target phase](#) and [bubble phase](#). Event objects complete these phases as described below. A phase will be skipped if it is not supported, or if the event object's propagation has been stopped. For example, if the [bubbles](#) attribute is set to false, the bubble phase will be skipped, and if [stopPropagation\(\)](#) has been called prior to the dispatch, all phases will be skipped.

- The **capture phase**: The event object propagates through the target's ancestors from the [Window](#) to the target's parent. This phase is also known as the *capturing phase*.
- The **target phase**: The event object arrives at the event object's [event target](#). This phase is also known as the *at-target phase*. If the [event type](#) indicates that the event doesn't bubble, then the event object will halt after

completion of this phase.

- The **bubble phase**: The event object propagates through the target's ancestors in reverse order, starting with the target's parent and ending with the [Window](#). This phase is also known as the *bubbling phase*.

3.2. Default actions and cancelable events

Events are typically dispatched by the implementation as a result of a user action, in response to the completion of a task, or to signal progress during asynchronous activity (such as a network request). Some events can be used to control the behavior that the implementation may take next (or undo an action that the implementation already took). Events in this category are said to be *cancelable* and the behavior they cancel is called their [default action](#). Cancelable event objects can be associated with one or more 'default actions'. To cancel an event, call the [preventDefault\(\)](#) method.

EXAMPLE 1:

A [mousedown](#) event is dispatched immediately after the user presses down a button on a pointing device (typically a mouse). One possible [default action](#) taken by the implementation is to set up a state machine that allows the user to drag images or select text. The [default action](#) depends on what happens next — for example, if the user's pointing device is over text, a text selection might begin. If the user's pointing device is over an image, then an image-drag action could begin. Preventing the [default action](#) of a [mousedown](#) event prevents these actions from occurring.

[Default actions](#) are usually performed after the event dispatch has been completed, but in exceptional cases they may also be performed immediately before the event is dispatched.

EXAMPLE 2:

The default action associated with the [click](#) event on `<input type="checkbox">` elements toggles the checked IDL attribute value of that element. If the [click](#) event's default action is cancelled, then the value is restored to its former state.

When an event is canceled, then the conditional [default actions](#) associated with the event is skipped (or as mentioned above, if the [default actions](#) are carried out before the dispatch, their effect is undone). Whether an event object is cancelable is indicated by the [cancelable](#) attribute. Calling [preventDefault\(\)](#) stops all related [default actions](#) of an event object. The [defaultPrevented](#) attribute indicates whether an event has already been canceled (e.g., by a prior event listener). If the [DOM application](#) itself initiated the dispatch, then the return value of the [dispatchEvent\(\)](#) method indicates whether the event object was cancelled.

NOTE:

Many implementations additionally interpret an event listener's return value, such as the value `false`, to mean that the [default action](#) of cancelable events will be cancelled (though `window.onerror` handlers are cancelled by returning `true`).

3.3. Synchronous and asynchronous events

Events may be dispatched either synchronously or asynchronously.

Events which are synchronous (“*sync events*”) are treated as if they are in a virtual queue in a first-in-first-out model, ordered by sequence of temporal occurrence with respect to other events, to changes in the DOM, and to user interaction. Each event in this virtual queue is delayed until the previous event has completed its propagation behavior, or been canceled. Some sync events are driven by a specific device or process, such as mouse button events. These events are governed by the [event order](#) algorithms defined for that set of events, and user agents will dispatch these events in the defined order.

Events which are asynchronous (“*async events*”) may be dispatched as the results of the action are completed, with no relation to other events, to other changes in the DOM, nor to user interaction.

EXAMPLE 3:

During loading of a document, an inline script element is parsed and executed. The [load](#) event is queued to be fired asynchronously at the script element. However, because it is an async event, its order with relation to other synchronous events fired during document load (such as the `DOMContentLoaded` event from [\[HTML5\]](#)) is not guaranteed.

3.4. Trusted events

Events that are generated by the [user agent](#), either as a result of user interaction, or as a direct result of changes to the DOM, are trusted by the [user agent](#) with privileges that are not afforded to events generated by script through the [createEvent\(\)](#) method, modified using the [initEvent\(\)](#) method, or dispatched via the [dispatchEvent\(\)](#) method. The [isTrusted](#) attribute of trusted events has a value of `true`, while untrusted events have a [isTrusted](#) attribute value of `false`.

Most untrusted events will not trigger [default actions](#), with the exception of the [click](#) event. This event always triggers the [default action](#), even if the [isTrusted](#) attribute is `false` (this behavior is retained for backward-compatibility). All other untrusted events behave as if the [preventDefault\(\)](#) method had been called on that event.

3.5. Activation triggers and behavior

Certain [event targets](#) (such as a link or button element) may have associated [activation behavior](#) (such as following a link) that implementations perform in response to an [activation trigger](#) (such as clicking a link).

EXAMPLE 4:

Both HTML and SVG have an `<a>` element which indicates a link. Relevant [activation triggers](#) for an `<a>` element are a [click](#) event on the text or image content of the `<a>` element, or a [keydown](#) event with a [key](#) attribute value of `"Enter"` key when the `<a>` element has focus. The activation behavior for an `<a>` element is normally to change the content of the window to the content of the new document, in the case of external links, or to reposition the current document relative to the new anchor, in the case of internal links.

An [activation trigger](#) is a user action or an event which indicates to the implementation that an activation behavior should be initiated. User-initiated [activation triggers](#) include clicking a mouse button on an activatable element, pressing the `[Enter]` key when an activatable element has focus, or pressing a key that is somehow linked to an activatable element (a “hotkey” or “access key”) even when that element does not have focus. Event-based [activation triggers](#) may include timer-based events that activate an element at a certain clock time or after a certain time period has elapsed, progress events after a certain action has been completed, or many other condition-based or state-based events.

3.6. Constructing Mouse and Keyboard Events

Generally, when a constructor of an [Event](#) interface, or of an interface inherited from the [Event](#) interface, is invoked, the steps described in [\[DOM\]](#) should be followed. However the [KeyboardEvent](#) and [MouseEvent](#) interfaces provide additional dictionary members for initializing the internal state of the [Event](#) object’s key modifiers: specifically, the internal state queried for using the [getModifierState\(\)](#) and [getModifierState\(\)](#) methods. This section supplements the DOM4 steps for initializing a new [Event](#) object with these optional modifier states.

For the purposes of constructing a [KeyboardEvent](#), [MouseEvent](#), or object derived from these objects using the algorithm below, all [KeyboardEvent](#), [MouseEvent](#), and derived objects have *internal key modifier state* which can be set and retrieved using the [key modifier names](#) described in the [Modifier Keys table](#) in [\[UIEvents-Key\]](#).

The following steps supplement the algorithm defined for constructing events in DOM4:

- If the [Event](#) being constructed is a [KeyboardEvent](#) or [MouseEvent](#) object or an object that derives from either of these, and a [EventModifierInit](#) argument was provided to the constructor, then run the following sub-steps:
 - For each [EventModifierInit](#) argument, if the dictionary member begins with the string “modifier”, then let the *key modifier name* be the dictionary member’s name excluding the prefix “modifier”, and set the [Event](#) object’s [internal key modifier state](#) that matches the [key modifier name](#) to the corresponding value.

4. Event Types

The DOM Event Model allows a DOM implementation to support multiple modules of events. The model has been designed to allow addition of new event modules in the future. This document does not attempt to define all possible events. For purposes of interoperability, the DOM defines a module of user interface events including lower level device dependent events and a module of document mutation events.

4.1. User Interface Events

The User Interface event module contains basic event types associated with user interfaces and document manipulation.

4.1.1. Interface UIEvent

Introduced in DOM Level 2

The [UIEvent](#) interface provides specific contextual information associated with User Interface events.

To create an instance of the [UIEvent](#) interface, use the UIEvent constructor, passing an optional [UIEventInit](#) dictionary.

```
[Constructor(DOMString type, optional UIEventInit eventInitDict)]  
interface UIEvent : Event {  
  readonly attribute Window? view;  
  readonly attribute long detail;  
};
```

UIEvent . view

The view attribute identifies the Window from which the event was generated.

The [un-initialized value](#) of this attribute MUST be null.

UIEvent . detail

Specifies some detail information about the [Event](#), depending on the type of event.

The [un-initialized value](#) of this attribute MUST be 0.

```
dictionary UIEventInit : EventInit {  
  Window? view = null;  
  long detail = 0;  
};
```

UIEventInit . view

Should be initialized to the Window object of the global environment in which this event will be dispatched. If this event will be dispatched to an element, the view property should be set to the Window object containing the element's ownerDocument.

UIEventInit . detail

This value is initialized to a number that is application-specific.

4.1.2. UI Event Types

The User Interface event types are listed below. Some of these events use the [UIEvent](#) interface if generated from a user interface, but the [Event](#) interface otherwise, as detailed in each event.

4.1.2.1. *load*

Type	load
Interface	UIEvent if generated from a user interface, Event otherwise.
Sync / Async	Async
Bubbles	No
Trusted Targets	Window , Document, Element
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : common object whose contained resources have loaded• UIEvent.view : Window• UIEvent.detail : 0

A [user agent](#) MUST dispatch this event when the DOM implementation finishes loading the resource (such as the document) and any dependent resources (such as images, style sheets, or scripts). Dependent resources that fail to load MUST NOT prevent this event from firing if the resource that loaded them is still accessible via the DOM. If this event type is dispatched, implementations are REQUIRED to dispatch this event at least on the Document node.

NOTE:

For legacy reasons, [load](#) events for resources inside the document (e.g., images) do not include the [Window](#) in the propagation path in HTML implementations. See [\[HTML5\]](#) for more information.

4.1.2.2. *unload*

Type	unload
Interface	UIEvent if generated from a user interface, Event otherwise.
Sync / Async	Sync
Bubbles	No
Trusted Targets	Window , Document, Element
Cancelable	No
Default action	None
Context	

(trusted events)	<ul style="list-style-type: none"> • Event.target : common object whose contained resources have been removed • UIEvent.view : Window • UIEvent.detail : 0
------------------	---

A [user agent](#) MUST dispatch this event when the DOM Implementation removes from the environment the resource (such as the document) or any dependent resources (such as images, style sheets, scripts). The document MUST be unloaded after the dispatch of this event type. If this event type is dispatched, implementations are REQUIRED to dispatch this event at least on the Document node.

4.1.2.3. *abort*

Type	abort
Interface	UIEvent if generated from a user interface, Event otherwise.
Sync / Async	Sync
Bubbles	No
Trusted Targets	Window , Element
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : element whose resources have been stopped from loading without error • UIEvent.view : Window • UIEvent.detail : 0

A [user agent](#) MUST dispatch this event when the loading of a resource has been aborted, such as by a user canceling the load while it is still in progress.

4.1.2.4. *error*

Type	error
Interface	UIEvent if generated from a user interface, Event otherwise.
Sync / Async	Async
Bubbles	No
Trusted Targets	Window , Element
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : element whose resources have been stopped from loading due to an error • UIEvent.view : Window • UIEvent.detail : 0

A [user agent](#) MUST dispatch this event when a resource failed to load, or has been loaded but cannot be interpreted according to its semantics, such as an invalid image, a script execution error, or non-well-formed XML.

4.1.2.5. *select*

Type	select
Interface	UIEvent if generated from a user interface, Event otherwise.
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : element whose text content has been selected• UIEvent.view : Window• UIEvent.detail : 0

A [user agent](#) MUST dispatch this event when a user selects some text. This event is dispatched after the selection has occurred.

This specification does not provide contextual information to access the selected text. Where applicable, a [host language](#) SHOULD define rules for how a user MAY select content (with consideration for international language conventions), at what point the [select](#) event is dispatched, and how a content author MAY access the user-selected content.

NOTE:

In order to access to user-selected content, content authors will use native capabilities of the [host languages](#), such as the `Document.getSelection()` method of the HTML Editing APIs [\[Editing\]](#).

NOTE:

The [select](#) event might not be available for all elements in all languages. For example, in [\[HTML5\]](#), [select](#) events can be dispatched only on form [input](#) and [textarea](#) elements. Implementations can dispatch [select](#) events in any context deemed appropriate, including text selections outside of form controls, or image or markup selections such as in SVG.

4.2. Focus Events

NOTE:

This interface and its associated event types and [§4.2.2 Focus Event Order](#) were designed in accordance to the concepts and guidelines defined in [User Agent Accessibility Guidelines 2.0 \[UAAG20\]](#), with particular attention on the [focus mechanism](#) and the terms defined in the [glossary entry for focus](#).

4.2.1. Interface FocusEvent

Introduced in this specification

The [FocusEvent](#) interface provides specific contextual information associated with Focus events.

To create an instance of the [FocusEvent](#) interface, use the `FocusEvent` constructor, passing an optional [FocusEventInit](#) dictionary.

```
[Constructor(DOMString type, optional FocusEventInit eventInitDict)]  
interface FocusEvent : UIEvent {  
  readonly attribute EventTarget? relatedTarget;  
};
```

FocusEvent . *relatedTarget*

Used to identify a secondary [EventTarget](#) related to a Focus event, depending on the type of event.

For security reasons with nested browsing contexts, when tabbing into or out of a nested context, the relevant [EventTarget](#) SHOULD be null.

The [un-initialized value](#) of this attribute MUST be null.

```
dictionary FocusEventInit : UIEventInit {  
  EventTarget? relatedTarget = null;  
};
```

FocusEventInit . *relatedTarget*

The [relatedTarget](#) should be initialized to the element losing focus (in the case of a [focus](#) or [focusin](#) event) or the element gaining focus (in the case of a [blur](#) or [focusout](#) event).

4.2.2. Focus Event Order

The focus events defined in this specification occur in a set order relative to one another. The following is the typical sequence of events when a focus is shifted between elements (this order assumes that no element is initially focused):

	Event Type	Notes
--	------------	-------

User shifts focus

- | | | |
|---|-------------------------|---|
| 1 | focusin | Sent before first target element receives focus |
| 2 | focus | Sent after first target element receives focus |

	<i>User shifts focus</i>
3	<u>focusout</u> Sent before first target element loses focus
4	<u>focusin</u> Sent before second target element receives focus
5	<u>blur</u> Sent after first target element loses focus
6	<u>focus</u> Sent after second target element receives focus

NOTE:

This specification does not define the behavior of focus events when interacting with methods such as `focus()` or `blur()`. See the relevant specifications where those methods are defined for such behavior.

4.2.3. Document Focus and Focus Context

This event module includes event types for notification of changes in document [focus](#). There are three distinct focus contexts that are relevant to this discussion:

- The *operating system focus context* which MAY be on one of many different applications currently running on the computer. One of these applications with focus can be a browser.
- When the browser has focus, the user can switch (such as with the tab key) the *application focus context* among the different browser user interface fields (e.g., the Web site location bar, a search field, etc.). One of these user interface fields can be the document being shown in a tab.
- When the document itself has focus, the *document focus context* can be set to any of the focusable elements in the document.

The event types defined in this specification deal exclusively with document focus, and the [event target](#) identified in the event details MUST only be part of the document or documents in the window, never a part of the browser or operating system, even when switching from one focus context to another.

Normally, a document always has a focused element (even if it is the document element itself) and a persistent [focus ring](#). When switching between focus contexts, the document's currently focused element and focus ring normally remain in their current state. For example, if a document has three focusable elements, with the second element focused, when a user changes operating system focus to another application and then back to the browser, the second element will still be focused within the document, and tabbing will change the focus to the third element. A [host language](#) MAY define specific elements which might receive focus, the conditions under which an element MAY receive focus, the means by which focus MAY be changed, and the order in which the focus changes. For example, in some cases an element might be given focus by moving a pointer over it, while other circumstances might require a mouse click. Some elements might not be focusable at all, and some might be focusable only by special means (clicking on the element), but not by tabbing to it. Documents MAY contain multiple focus rings. Other specifications MAY define a more complex focus model than is described in this specification, including allowing multiple elements to have the current focus.

4.2.4. Focus Event Types

The Focus event types are listed below.

4.2.4.1. *blur*

Type	blur
Interface	FocusEvent
Sync / Async	Sync
Bubbles	No
Trusted Targets	Window , Element
Cancelable	No
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : event target losing focus• UIEvent.view : Window• UIEvent.detail : 0• FocusEvent.relatedTarget : event target receiving focus.

A [user agent](#) MUST dispatch this event when an [event target](#) loses focus. The focus MUST be taken from the element before the dispatch of this event type. This event type is similar to [focusout](#), but is dispatched after focus is shifted, and does not bubble.

4.2.4.2. *focus*

Type	focus
Interface	FocusEvent
Sync / Async	Sync
Bubbles	No
Trusted Targets	Window , Element
Cancelable	No
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : event target receiving focus• UIEvent.view : Window• UIEvent.detail : 0• FocusEvent.relatedTarget : event target losing focus (if any).

A [user agent](#) MUST dispatch this event when an [event target](#) receives focus. The focus MUST be given to the element before the dispatch of this event type. This event type is similar to [focusin](#), but is dispatched after focus is shifted, and does not bubble.

4.2.4.3. *focusin*

Type	focusin
Interface	FocusEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Window , Element
Cancelable	No
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : event target receiving focus• UIEvent.view : Window• UIEvent.detail : 0• FocusEvent.relatedTarget : event target losing focus (if any).

A [user agent](#) MUST dispatch this event when an [event target](#) is about to receive focus. This event type MUST be dispatched before the element is given focus. The [event target](#) MUST be the element which is about to receive focus. This event type is similar to [focus](#), but is dispatched before focus is shifted, and does bubble.

NOTE:

When using this event type, the content author can use the event's [relatedTarget](#) attribute (or a host-language-specific method or means) to get the currently focused element before the focus shifts to the next focus [event target](#), thus having access to both the element losing focus and the element gaining focus without the use of the [blur](#) or [focusout](#) event types.

4.2.4.4. *focusout*

Type	focusout
Interface	FocusEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Window , Element
Cancelable	No
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : event target losing focus• UIEvent.view : Window• UIEvent.detail : 0• FocusEvent.relatedTarget : event target receiving focus.

A [user agent](#) MUST dispatch this event when an [event target](#) is about to lose focus. This event type MUST be dispatched before the element loses focus. The [event target](#) MUST be the element which is about to lose focus. This event type is similar to [blur](#), but is dispatched before focus is shifted, and does bubble.

4.3. Mouse Events

The mouse event module originates from the [\[HTML40\]](#) `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, and `onmouseout` attributes. This event module is specifically designed for use with pointing input devices, such as a mouse or a trackball.

4.3.1. Interface `MouseEvent`

Introduced in DOM Level 2, modified in this specification

The [MouseEvent](#) interface provides specific contextual information associated with Mouse events.

In the case of nested elements, mouse events are always targeted at the most deeply nested element.

NOTE:

Ancestors of the targeted element can use event bubbling to obtain notifications of mouse events which occur within their descendent elements.

To create an instance of the [MouseEvent](#) interface, use the [MouseEvent](#) constructor, passing an optional [MouseEventInit](#) dictionary.

NOTE:

When initializing [MouseEvent](#) objects using `initMouseEvent`, implementations can use the client coordinates [clientX](#) and [clientY](#) for calculation of other coordinates (such as target coordinates exposed by [DOM Level 0](#) implementations or other proprietary attributes, e.g., `pageX`).

```
[Constructor(DOMString type, optional MouseEventInit eventInitDict)]
interface MouseEvent : UIEvent {
  readonly attribute long screenX;
  readonly attribute long screenY;
  readonly attribute long clientX;
  readonly attribute long clientY;

  readonly attribute boolean ctrlKey;
  readonly attribute boolean shiftKey;
  readonly attribute boolean altKey;
  readonly attribute boolean metaKey;

  readonly attribute short button;
  readonly attribute unsigned short buttons;

  readonly attribute EventTarget? relatedTarget;

  boolean getModifierState(DOMString keyArg);
};
```

MouseEvent . screenX

The horizontal coordinate at which the event occurred relative to the origin of the screen coordinate system.
The un-initialized value of this attribute MUST be 0.

MouseEvent . screenY

The vertical coordinate at which the event occurred relative to the origin of the screen coordinate system.
The un-initialized value of this attribute MUST be 0.

MouseEvent . clientX

The horizontal coordinate at which the event occurred relative to the viewport associated with the event.
The un-initialized value of this attribute MUST be 0.

MouseEvent . clientY

The vertical coordinate at which the event occurred relative to the viewport associated with the event.
The un-initialized value of this attribute MUST be 0.

MouseEvent . ctrlKey

Refer to the ctrlKey attribute.

The un-initialized value of this attribute MUST be false.

MouseEvent . shiftKey

Refer to the shiftKey attribute.

The un-initialized value of this attribute MUST be false.

MouseEvent . altKey

Refer to the altKey attribute.

The un-initialized value of this attribute MUST be false.

MouseEvent . metaKey

Refer to the [metaKey](#) attribute.

The [un-initialized value](#) of this attribute MUST be false.

MouseEvent . button

During mouse events caused by the depression or release of a mouse button, `button` MUST be used to indicate which pointer device button changed state.

The value of the [button](#) attribute MUST be as follows:

- 0 MUST indicate the primary button of the device (in general, the left button or the only button on single-button devices, used to activate a user interface control or select text) or the un-initialized value.
- 1 MUST indicate the auxiliary button (in general, the middle button, often combined with a mouse wheel).
- 2 MUST indicate the secondary button (in general, the right button, often used to display a context menu).

Some pointing devices provide or simulate more button states, and values higher than 2 or lower than 0 MAY be used to represent such buttons.

NOTE:

The value of [button](#) is not updated for events not caused by the depression/release of a mouse button. In these scenarios, take care not to interpret the value 0 as the left button, but rather as the [un-initialized value](#).

NOTE:

Some [default actions](#) related to events such as [mousedown](#) and [mouseup](#) depend on the specific mouse button in use.

The [un-initialized value](#) of this attribute MUST be 0.

MouseEvent . buttons

During any mouse events, `buttons` MUST be used to indicate which combination of mouse buttons are currently being pressed, expressed as a bitmask.

NOTE:

Though similarly named, the values for the [buttons](#) attribute and the [button](#) attribute are very different. The value of [button](#) is assumed to be valid during [mousedown](#) / [mouseup](#) event handlers, whereas the [buttons](#) attribute reflects the state of the mouse's buttons for any trusted [MouseEvent](#) object (while it is being dispatched), because it can represent the "no button currently active" state (0).

The value of the [buttons](#) attribute MUST be as follows:

- 0 MUST indicate no button is currently active.

- 1 MUST indicate the primary button of the device (in general, the left button or the only button on single-button devices, used to activate a user interface control or select text).
- 2 MUST indicate the secondary button (in general, the right button, often used to display a context menu), if present.
- 4 MUST indicate the auxiliary button (in general, the middle button, often combined with a mouse wheel).

Some pointing devices provide or simulate more buttons. To represent such buttons, the value MUST be doubled for each successive button (in the binary series 8, 16, 32, ...).

NOTE:

Because the sum of any set of button values is a unique number, a content author can use a bitwise operation to determine how many buttons are currently being pressed and which buttons they are, for an arbitrary number of mouse buttons on a device. For example, the value 3 indicates that the left and right button are currently both pressed, while the value 5 indicates that the left and middle button are currently both pressed.

NOTE:

Some [default actions](#) related to events such as [mousedown](#) and [mouseup](#) depend on the specific mouse button in use.

The [un-initialized value](#) of this attribute MUST be 0.

MouseEvent . relatedTarget

Used to identify a secondary [EventTarget](#) related to a UI event, depending on the type of event.

The [un-initialized value](#) of this attribute MUST be null.

MouseEvent . getModifierState(keyArg)

Introduced in this specification

Queries the state of a modifier using a key value. See [§5.3.2 Modifier keys](#) for a list of valid (case-sensitive) arguments to this method.

Returns `true` if it is a modifier key and the modifier is activated, `false` otherwise.

DOMString keyArg

Refer to the [KeyboardEvent.getModifierState\(\)](#) method for a description of this parameter.

```
dictionary MouseEventInit : EventModifierInit {  
    long screenX = 0;  
    long screenY = 0;  
    long clientX = 0;  
    long clientY = 0;  
  
    short button = 0;  
    unsigned short buttons = 0;  
    EventTarget? relatedTarget = null;  
};
```

MouseEventInit . screenX

See `screenY` (substituting "horizontal" for "vertical").

MouseEventInit . screenY

Initializes the `screenY` attribute of the `MouseEvent` object to the desired vertical relative position of the mouse pointer on the user's screen.

Initializing the event object to the given mouse position must not move the user's mouse pointer to the initialized position.

MouseEventInit . clientX

See `clientY` (substituting "horizontal" for "vertical").

MouseEventInit . clientY

Initializes the `clientY` attribute of the `MouseEvent` object to the desired vertical position of the mouse pointer relative to the client window of the user's browser.

Initializing the event object to the given mouse position must not move the user's mouse pointer to the initialized position.

MouseEventInit . button

Initializes the `button` attribute of the `MouseEvent` object to a number representing the desired state of the button(s) of the mouse.

NOTE:

The value 0 is used to represent the primary mouse button, 1 is used to represent the auxiliary/middle mouse button, and 2 to represent the right mouse button. Numbers greater than 2 are also possible, but are not specified in this document.

MouseEventInit . buttons

Initializes the `buttons` attribute of the `MouseEvent` object to a number representing one *or more* of the button(s) of the mouse that are to be considered active.

NOTE:

The `buttons` attribute is a bit-field. If a mask value of 1 is true when applied to the value of the bit field, then the primary mouse button is down. If a mask value of 2 is true when applied to the value of the bit field, then the right mouse button is down. If a mask value of 4 is true when applied to the value of the bit field, then the auxiliary/middle button is down.

EXAMPLE 5:

In JavaScript, to initialize the `buttons` attribute as if the right (2) and middle button (4) were being pressed simultaneously, the `buttons` value can be assigned as either:

```
{ buttons: 2 | 4 }
```

or:

```
{ buttons: 6 }
```

MouseEventInit relatedTarget

The `relatedTarget` should be initialized to the element whose bounds the mouse pointer just left (in the case of a *mouseover* or *mouseenter* event) or the element whose bounds the mouse pointer is entering (in the case of a *mouseout* or *mouseleave* or *focusout* event). For other events, this value need not be assigned (and will default to null).

Implementations MUST maintain the *current click count* when generating mouse events. This MUST be a non-negative integer indicating the number of consecutive clicks of a pointing device button within a specific time. The delay after which the count resets is specific to the environment configuration.

4.3.2. Event Modifier Initializers

The [MouseEvent](#) and [KeyboardEvent](#) interfaces share a set of keyboard modifier attributes and support a mechanism for retrieving additional modifier states. The following dictionary enables authors to initialize keyboard modifier attributes of the [MouseEvent](#) and [KeyboardEvent](#) interfaces, as well as the additional modifier states queried via [getModifierState\(\)](#). The steps for constructing events using this dictionary are defined in the [event constructors](#) section.

```
dictionary EventModifierInit : UIEventInit {
    boolean ctrlKey = false;
    boolean shiftKey = false;
    boolean altKey = false;
    boolean metaKey = false;

    boolean modifierAltGraph = false;
    boolean modifierCapsLock = false;
    boolean modifierFn = false;
    boolean modifierFnLock = false;
    boolean modifierHyper = false;
    boolean modifierNumLock = false;
    boolean modifierScrollLock = false;
    boolean modifierSuper = false;
    boolean modifierSymbol = false;
    boolean modifierSymbolLock = false;
};
```

EventModifierInit . ctrlKey

Initializes the **altKey** attribute of the [MouseEvent](#) or [KeyboardEvent](#) objects to **true** if the **Control** key modifier is to be considered active, **false** otherwise.

When **true**, implementations must also initialize the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter **Control** must return **true**.

EventModifierInit . shiftKey

Initializes the **shiftKey** attribute of the [MouseEvent](#) or [KeyboardEvent](#) objects to **true** if the **Shift** key modifier is to be considered active, **false** otherwise.

When **true**, implementations must also initialize the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter **Shift** must return **true**.

EventModifierInit . altKey

Initializes the **altKey** attribute of the [MouseEvent](#) or [KeyboardEvent](#) objects to **true** if the **Alt** (alternative) (or **Option**) key modifier is to be considered active, **false** otherwise.

When **true**, implementations must also initialize the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter **Alt** must return **true**.

EventModifierInit . metaKey

Initializes the **metaKey** attribute of the [MouseEvent](#) or [KeyboardEvent](#) objects to **true** if the **Meta** key modifier is to be considered active, **false** otherwise.

When **true**, implementations must also initialize the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with either the parameter **Meta** must

return true.

EventModifierInit . modifierAltGraph

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `AltGraph` must return true.

EventModifierInit . modifierCapsLock

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `CapsLock` must return true.

EventModifierInit . modifierFn

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `Fn` must return true.

EventModifierInit . modifierFnLock

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `FnLock` must return true.

EventModifierInit . modifierHyper

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `Hyper` must return true.

EventModifierInit . modifierNumLock

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `NumLock` must return true.

EventModifierInit . modifierScrollLock

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `ScrollLock` must return true.

EventModifierInit . modifierSuper

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `Super` must return true.

EventModifierInit . modifierSymbol

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `Symbol` must return true.

EventModifierInit . modifierSymbolLock

Initializes the event object's key modifier state such that calls to the [getModifierState\(\)](#) or [getModifierState\(\)](#) when provided with the parameter `SymbolLock` must return true.

4.3.3. Mouse Event Order

Certain mouse events defined in this specification MUST occur in a set order relative to one another. The following shows the event sequence that MUST occur when a pointing device's cursor is moved over an element:

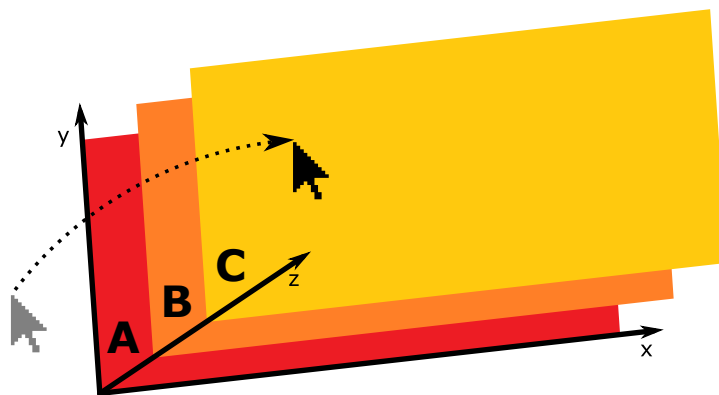
	Event Type	Element	Notes
1	mousemove		<i>Pointing device is moved into element A...</i>
2	mouseover	A	
3	mouseenter	A	

4	mousemove	A	Multiple mousemove events <i>Pointing device is moved out of element A...</i>
5	mouseout	A	
6	mouseleave	A	

When a pointing device is moved into an element *A*, and then into a nested element *B* and then back out again, the following sequence of events MUST occur:

	Event Type	Element	Notes
1	mousemove		<i>Pointing device is moved into element A...</i>
2	mouseover	A	
3	mouseenter	A	
4	mousemove	A	Multiple mousemove events <i>Pointing device is moved into nested element B...</i>
5	mouseout	A	
6	mouseover	B	
7	mouseenter	B	
8	mousemove	B	Multiple mousemove events <i>Pointing device is moved from element B into A...</i>
9	mouseout	B	
10	mouseleave	B	
11	mouseover	A	
12	mousemove	A	Multiple mousemove events <i>Pointing device is moved out of element A...</i>
13	mouseout	A	
14	mouseleave	A	

Sometimes elements can be visually overlapped using CSS. In the following example, three elements labeled A, B, and C all have the same dimensions and absolute position on a web page. Element C is a child of B, and B is a child of A in the DOM:



Graphical representation of three stacked elements all on top of each other, with the pointing device moving over the stack.

When the pointing device is moved from outside the element stack to the element labeled C and then moved out again, the following series of events MUST occur:

	Event Type	Element	Notes
1	<u>mousemove</u>		<i>Pointing device is moved into element C, the topmost element in the stack</i>
2	<u>mouseover</u>	C	
3	<u>mouseenter</u>	A	
4	<u>mouseenter</u>	B	
5	<u>mouseenter</u>	C	
6	<u>mousemove</u>	C	Multiple <u>mousemove</u> events <i>Pointing device is moved out of element C...</i>
7	<u>mouseout</u>	C	
8	<u>mouseleave</u>	C	
9	<u>mouseleave</u>	B	
10	<u>mouseleave</u>	A	

NOTE:

The [mouseover](#)/[mouseout](#) events are only fired once, while [mouseenter](#)/[mouseleave](#) events are fired three times (once to each element).

The following is the typical sequence of events when a button associated with a pointing device (e.g., a mouse button or trackpad) is pressed and released over an element:

	Event Type	Notes
1	<u>mousedown</u>	
2	<u>mousemove</u>	OPTIONAL, multiple events, some limits
3	<u>mouseup</u>	
4	<u>click</u>	
5	<u>mousemove</u>	OPTIONAL, multiple events, some limits
6	<u>mousedown</u>	
7	<u>mousemove</u>	OPTIONAL, multiple events, some limits
8	<u>mouseup</u>	
9	<u>click</u>	
10	<u>dblclick</u>	

NOTE:

The lag time, degree, distance, and number of [mousemove](#) events allowed between the [mousedown](#) and [mouseup](#) events while still firing a [click](#) or [dblclick](#) event will be implementation-, device-, and platform-specific. This tolerance can aid users that have physical disabilities like unsteady hands when these users interact with a pointing device.

Each implementation will determine the appropriate [hysteresis](#) tolerance, but in general SHOULD fire [click](#) and [dblclick](#) events when the event target of the associated [mousedown](#) and [mouseup](#) events is the same element with no [mouseout](#) or [mouseleave](#) events intervening, and SHOULD fire [click](#) and [dblclick](#) events on the nearest common inclusive ancestor when the associated [mousedown](#) and [mouseup](#) event targets are different.

EXAMPLE 6:

If a [mousedown](#) event was targeted at an HTML document's [body element](#), and the corresponding [mouseup](#) event was targeted at the [root element](#), then the [click](#) event will be dispatched to the [root element](#), since it is the nearest common inclusive ancestor.

If the [event target](#) (e.g. the target element) is removed from the DOM during the mouse events sequence, the remaining events of the sequence MUST NOT be fired on that element.

EXAMPLE 7:

If the target element is removed from the DOM as the result of a [mousedown](#) event, no events for that element will be dispatched for [mouseup](#), [click](#), or [dblclick](#), nor any default activation events. However, the [mouseup](#) event will still be dispatched on the element that is exposed to the mouse after the removal of the initial target element. Similarly, if the target element is removed from the DOM during the dispatch of a [mouseup](#) event, the [click](#) and subsequent events will not be dispatched.

4.3.4. Mouse Event Types

The Mouse event types are listed below. In the case of nested elements, mouse event types are always targeted at the most deeply nested element. Ancestors of the targeted element MAY use bubbling to obtain notification of mouse events which occur within its descendent elements.

4.3.4.1. *click*

Type	click
Interface	MouseEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	Varies
Context (trusted events)	<ul style="list-style-type: none">• Event.target : topmost event target• UIEvent.view : Window• UIEvent.detail : indicates the current click count; the attribute value MUST be 1 when the user begins this action and increments by 1 for each click.• MouseEvent.screenX : value based on the pointer position on the screen• MouseEvent.screenY : value based on the pointer position on the screen• MouseEvent.clientX : value based on the pointer position within the viewport• MouseEvent.clientY : value based on the pointer position within the viewport• MouseEvent.altKey : true if Alt modifier was active, otherwise false• MouseEvent.ctrlKey : true if Control modifier was active, otherwise false

- [MouseEvent.shiftKey](#) : true if **Shift** modifier was active, otherwise false
- [MouseEvent.metaKey](#) : true if **Meta** modifier was active, otherwise false
- [MouseEvent.button](#) : value based on current button pressed
- [MouseEvent.buttons](#) : value based on all buttons current depressed, 0 if no buttons pressed
- [MouseEvent.relatedTarget](#) : null

The [click](#) event type MUST be dispatched on the [topmost event target](#) indicated by the pointer, when the user presses down and releases the primary pointer button, or otherwise activates the pointer in a manner that simulates such an action. The actuation method of the mouse button depends upon the pointer device and the environment configuration, e.g., it MAY depend on the screen location or the delay between the press and release of the pointing device button.

The [click](#) event should only be fired for the primary pointer button (i.e., when [button](#) value is 0, [buttons](#) value is 1). Secondary buttons (like the middle or right button on a standard mouse) MUST NOT fire [click](#) events. The [click](#) event MAY be preceded by the [mousedown](#) and [mouseup](#) events on the same element, disregarding changes between other node types (e.g., text nodes). Depending upon the environment configuration, the [click](#) event MAY be dispatched if one or more of the event types [mouseover](#), [mousemove](#), and [mouseout](#) occur between the press and release of the pointing device button. The [click](#) event MAY also be followed by the [dblclick](#) event.

EXAMPLE 8:

If a user mouses down on a text node child of a `<p>` element which has been styled with a large line-height, shifts the mouse slightly such that it is no longer over an area containing text but is still within the containing block of that `<p>` element (i.e., the pointer is between lines of the same text block, but not over the text node per se), then subsequently mouses up, this will likely still trigger a [click](#) event (if it falls within the normal temporal [hysteresis](#) for a [click](#)), since the user has stayed within the scope of the same element. Note that user-agent-generated mouse events are not dispatched on text nodes.

In addition to being associated with pointer devices, the [click](#) event type MUST be dispatched as part of an element activation, as described in [§3.5 Activation triggers and behavior](#).

NOTE:

For maximum accessibility, content authors are encouraged to use the [click](#) event type when defining activation behavior for custom controls, rather than other pointing-device event types such as [mousedown](#) or [mouseup](#), which are more device-specific. Though the [click](#) event type has its origins in pointer devices (e.g., a mouse), subsequent implementation enhancements have extended it beyond that association, and it can be considered a device-independent event type for element activation.

The [default action](#) of the [click](#) event type varies based on the [event target](#) of the event and the value of the [button](#) or [buttons](#) attributes. Typical [default actions](#) of the [click](#) event type are as follows:

- If the [event target](#) has associated activation behavior, the [default action](#) MUST be to execute that activation behavior (see [§3.5 Activation triggers and behavior](#)).
- If the [event target](#) is focusable, the [default action](#) MUST be to give that element document focus.

4.3.4.2. *dblclick*

Type	dblclick
Interface	MouseEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : topmost event target• UIEvent.view : Window• UIEvent.detail : indicates the current click count• MouseEvent.screenX : value based on the pointer position on the screen• MouseEvent.screenY : value based on the pointer position on the screen• MouseEvent.clientX : value based on the pointer position within the viewport• MouseEvent.clientY : value based on the pointer position within the viewport• MouseEvent.altKey : true if Alt modifier was active, otherwise false• MouseEvent.ctrlKey : true if Control modifier was active, otherwise false• MouseEvent.shiftKey : true if Shift modifier was active, otherwise false• MouseEvent.metaKey : true if Meta modifier was active, otherwise false• MouseEvent.button : value based on current button pressed• MouseEvent.buttons : value based on all buttons current depressed, 0 if no buttons pressed• MouseEvent.relatedTarget : null

A [user agent](#) MUST dispatch this event when the primary button of a pointing device is clicked twice over an element. The definition of a double click depends on the environment configuration, except that the event target MUST be the same between [mousedown](#), [mouseup](#), and [dblclick](#). This event type MUST be dispatched after the event type [click](#) if a click and double click occur simultaneously, and after the event type [mouseup](#) otherwise. As with the [click](#) event, the [dblclick](#) event should only be fired for the primary pointer button. Secondary buttons MUST NOT fire [dblclick](#) events.

NOTE:

Canceling the [click](#) event does not affect the firing of a [dblclick](#) event.

As with the [click](#) event type, the [default action](#) of the [dblclick](#) event type varies based on the [event target](#) of the event and the value of the [button](#) or [buttons](#) attributes. Normally, the typical [default actions](#) of the [dblclick](#) event type match those of the [click](#) event type, with the following additional behavior:

- If the [event target](#) is selectable, the [default action](#) MUST be to select part or all of the selectable content. Subsequent clicks MAY select additional selectable portions of that content.

4.3.4.3. *mousedown*

Type	mousedown
Interface	MouseEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	Varies: Start a drag/drop operation; start a text selection; start a scroll/pan interaction (in combination with the middle mouse button, if supported)
Context (trusted events)	<ul style="list-style-type: none">• Event.target : topmost event target• UIEvent.view : Window• UIEvent.detail : indicates the current click count incremented by one. For example, if no click happened before the mousedown, detail will contain the value 1• MouseEvent.screenX : value based on the pointer position on the screen• MouseEvent.screenY : value based on the pointer position on the screen• MouseEvent.clientX : value based on the pointer position within the viewport• MouseEvent.clientY : value based on the pointer position within the viewport• MouseEvent.altKey : true if Alt modifier was active, otherwise false• MouseEvent.ctrlKey : true if Control modifier was active, otherwise false• MouseEvent.shiftKey : true if Shift modifier was active, otherwise false• MouseEvent.metaKey : true if Meta modifier was active, otherwise false• MouseEvent.button : value based on current button pressed• MouseEvent.buttons : value based on all buttons current depressed, 0 if no buttons pressed• MouseEvent.relatedTarget : null

A [user agent](#) MUST dispatch this event when a pointing device button is pressed over an element.

NOTE:

Many implementations use the [mousedown](#) event to begin a variety of contextually dependent [default actions](#). These default actions can be prevented if this event is canceled. Some of these default actions could include: beginning a drag/drop interaction with an image or link, starting text selection, etc. Additionally, some implementations provide a mouse-driven panning feature that is activated when the middle mouse button is pressed at the time the [mousedown](#) event is dispatched.

4.3.4.4. *mouseenter*

Type	mouseenter
Interface	MouseEvent
Sync / Async	Sync

Bubbles	No
Trusted Targets	Element
Cancelable	No
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : topmost event target • UIEvent.view : Window • UIEvent.detail : 0 • MouseEvent.screenX : value based on the pointer position on the screen • MouseEvent.screenY : value based on the pointer position on the screen • MouseEvent.clientX : value based on the pointer position within the viewport • MouseEvent.clientY : value based on the pointer position within the viewport • MouseEvent.altKey : true if Alt modifier was active, otherwise false • MouseEvent.ctrlKey : true if Control modifier was active, otherwise false • MouseEvent.shiftKey : true if Shift modifier was active, otherwise false • MouseEvent.metaKey : true if Meta modifier was active, otherwise false • MouseEvent.button : 0 • MouseEvent.buttons : value based on all buttons current depressed, 0 if no buttons pressed • MouseEvent.relatedTarget : indicates the event target a pointing device is exiting, if any.

A [user agent](#) MUST dispatch this event when a pointing device is moved onto the boundaries of an element or one of its descendent elements. This event type is similar to [mouseover](#), but differs in that it does not bubble, and MUST NOT be dispatched when the pointer device moves from an element onto the boundaries of one of its descendent elements.

NOTE:

There are similarities between this event type and the CSS [:hover pseudo-class](#) [CSS2]. See also the [mouseleave](#) event type.

4.3.4.5. *mouseleave*

Type	mouseleave
Interface	MouseEvent
Sync / Async	Sync
Bubbles	No
Trusted Targets	Element
Cancelable	No
Composed	Yes
Default action	None
Context	

(trusted events)	<ul style="list-style-type: none"> • Event.target : topmost event target • UIEvent.view : Window • UIEvent.detail : 0 • MouseEvent.screenX : value based on the pointer position on the screen • MouseEvent.screenY : value based on the pointer position on the screen • MouseEvent.clientX : value based on the pointer position within the viewport • MouseEvent.clientY : value based on the pointer position within the viewport • MouseEvent.altKey : true if Alt modifier was active, otherwise false • MouseEvent.ctrlKey : true if Control modifier was active, otherwise false • MouseEvent.shiftKey : true if Shift modifier was active, otherwise false • MouseEvent.metaKey : true if Meta modifier was active, otherwise false • MouseEvent.button : 0 • MouseEvent.buttons : value based on all buttons current depressed, 0 if no buttons pressed • MouseEvent.relatedTarget : indicates the event target a pointing device is entering, if any.
------------------	--

A [user agent](#) MUST dispatch this event when a pointing device is moved off of the boundaries of an element and all of its descendent elements. This event type is similar to [mouseout](#), but differs in that does not bubble, and that it MUST NOT be dispatched until the pointing device has left the boundaries of the element and the boundaries of all of its children.

NOTE:

There are similarities between this event type and the CSS [:hover pseudo-class](#) [\[CSS2\]](#). See also the [mouseenter](#) event type.

4.3.4.6. *mousemove*

Type	mousemove
Interface	MouseEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : topmost event target • UIEvent.view : Window • UIEvent.detail : 0 • MouseEvent.screenX : value based on the pointer position on the screen • MouseEvent.screenY : value based on the pointer position on the screen • MouseEvent.clientX : value based on the pointer position within the viewport • MouseEvent.clientY : value based on the pointer position within the viewport

- [MouseEvent.altKey](#) : true if **Alt** modifier was active, otherwise false
- [MouseEvent.ctrlKey](#) : true if **Control** modifier was active, otherwise false
- [MouseEvent.shiftKey](#) : true if **Shift** modifier was active, otherwise false
- [MouseEvent.metaKey](#) : true if **Meta** modifier was active, otherwise false
- [MouseEvent.button](#) : 0
- [MouseEvent.buttons](#) : value based on all buttons current depressed, 0 if no buttons pressed
- [MouseEvent.relatedTarget](#) : null

A [user agent](#) MUST dispatch this event when a pointing device is moved while it is over an element. The frequency rate of events while the pointing device is moved is implementation-, device-, and platform-specific, but multiple consecutive [mousemove](#) events SHOULD be fired for sustained pointer-device movement, rather than a single event for each instance of mouse movement. Implementations are encouraged to determine the optimal frequency rate to balance responsiveness with performance.

NOTE:

In some implementation environments, such as a browser, [mousemove](#) events can continue to fire if the user began a drag operation (e.g., a mouse button is pressed) and the pointing device has left the boundary of the user agent.

NOTE:

This event was formerly specified to be non-cancelable in DOM Level 2 Events [\[DOM-Level-2-Events\]](#), but was changed to reflect existing interoperability between user agents.

4.3.4.7. *mouseout*

Type	mouseout
Interface	MouseEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : topmost event target • UIEvent.view : Window • UIEvent.detail : 0 • MouseEvent.screenX : value based on the pointer position on the screen • MouseEvent.screenY : value based on the pointer position on the screen • MouseEvent.clientX : value based on the pointer position within the viewport • MouseEvent.clientY : value based on the pointer position within the viewport • MouseEvent.altKey : true if Alt modifier was active, otherwise false

- [MouseEvent.ctrlKey](#) : true if **Control** modifier was active, otherwise false
- [MouseEvent.shiftKey](#) : true if **Shift** modifier was active, otherwise false
- [MouseEvent.metaKey](#) : true if **Meta** modifier was active, otherwise false
- [MouseEvent.button](#) : 0
- [MouseEvent.buttons](#) : value based on all buttons current depressed, 0 if no buttons pressed
- [MouseEvent.relatedTarget](#) : indicates the [event target](#) a pointing device is entering, if any.

A [user agent](#) MUST dispatch this event when a pointing device is moved off of the boundaries of an element. This event type is similar to [mouseleave](#), but differs in that does bubble, and that it MUST be dispatched when the pointer device moves from an element onto the boundaries of one of its descendent elements.

NOTE:

See also the [mouseover](#) event type.

4.3.4.8. *mouseover*

Type	mouseover
Interface	MouseEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : topmost event target • UIEvent.view : Window • UIEvent.detail : 0 • MouseEvent.screenX : value based on the pointer position on the screen • MouseEvent.screenY : value based on the pointer position on the screen • MouseEvent.clientX : value based on the pointer position within the viewport • MouseEvent.clientY : value based on the pointer position within the viewport • MouseEvent.altKey : true if Alt modifier was active, otherwise false • MouseEvent.ctrlKey : true if Control modifier was active, otherwise false • MouseEvent.shiftKey : true if Shift modifier was active, otherwise false • MouseEvent.metaKey : true if Meta modifier was active, otherwise false • MouseEvent.button : 0 • MouseEvent.buttons : value based on all buttons current depressed, 0 if no buttons pressed • MouseEvent.relatedTarget : indicates the event target a pointing device is exiting, if any.

A [user agent](#) MUST dispatch this event when a pointing device is moved onto the boundaries of an element. This event type is similar to [mouseenter](#), but differs in that it bubbles, and that it MUST be dispatched when the pointer device moves onto the boundaries of an element whose ancestor element is the [event target](#) for the same event listener instance.

NOTE:

See also the [mouseout](#) event type.

4.3.4.9. *mouseup*

Type	mouseup
Interface	MouseEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	Invoke a context menu (in combination with the right mouse button, if supported)
Context (trusted events)	<ul style="list-style-type: none">• Event.target : topmost event target• UIEvent.view : Window• UIEvent.detail : indicates the current click count incremented by one.• MouseEvent.screenX : value based on the pointer position on the screen• MouseEvent.screenY : value based on the pointer position on the screen• MouseEvent.clientX : value based on the pointer position within the viewport• MouseEvent.clientY : value based on the pointer position within the viewport• MouseEvent.altKey : true if Alt modifier was active, otherwise false• MouseEvent.ctrlKey : true if Control modifier was active, otherwise false• MouseEvent.shiftKey : true if Shift modifier was active, otherwise false• MouseEvent.metaKey : true if Meta modifier was active, otherwise false• MouseEvent.button : value based on current button pressed• MouseEvent.buttons : value based on all buttons current depressed, 0 if no buttons pressed• MouseEvent.relatedTarget : null

A [user agent](#) MUST dispatch this event when a pointing device button is released over an element.

NOTE:

Many implementations will invoke a context menu as the default action of this event if the right mouse button is being released.

NOTE:

In some implementation environments, such as a browser, a [mouseup](#) event can be dispatched even if the pointing device has left the boundary of the user agent, e.g., if the user began a drag operation with a mouse button pressed.

4.4. Wheel Events

Wheels are devices that can be rotated in one or more spatial dimensions, and which can be associated with a pointer device. The coordinate system depends on the environment configuration.

EXAMPLE 9:

The user's environment might be configured to associate vertical scrolling with rotation along the y-axis, horizontal scrolling with rotation along the x-axis, and zooming with rotation along the z-axis.

The `deltaX`, `deltaY`, and `deltaZ` attributes of [WheelEvent](#) objects indicate a measurement along their respective axes in units of pixels, lines, or pages. The reported measurements are provided after an environment-specific algorithm translates the actual rotation/movement of the wheel device into the appropriate values and units.

NOTE:

A user's environment settings can be customized to interpret actual rotation/movement of a wheel device in different ways. One movement of a common "dented" mouse wheel can produce a measurement of 162 pixels (162 is just an example value, actual values can depend on the current screen dimensions of the user-agent). But a user can change their default environment settings to speed-up their mouse wheel, increasing this number. Furthermore, some mouse wheel software can support acceleration (the faster the wheel is rotated/moved, the greater the [delta](#) of each measurement) or even sub-pixel [rotation](#) measurements. Because of this, authors can not assume a given [rotation](#) amount in one user agent will produce the same [delta](#) value in all user agents.

The sign (positive or negative) of the values of the `deltaX`, `deltaY`, and `deltaZ` attributes MUST be consistent between multiple dispatches of the [wheel](#) event while the motion of the actual wheel device is rotating/moving in the same direction. If a user agent scrolls as the default action of the [wheel](#) event then the sign of the [delta](#) SHOULD be given by a right-hand coordinate system where positive X, Y, and Z axes are directed towards the right-most edge, bottom-most edge, and farthest depth (away from the user) of the document, respectively.

NOTE:

Individual user agents can (depending on their environment and hardware configuration) interpret the same physical user interaction on the wheel differently. For example, a vertical swipe on the edge of a trackpad from top to bottom can be interpreted as a wheel action intended to either scroll the page down or to pan the page up (i.e., resulting in either a positive or negative `deltaY` value respectively).

4.4.1. Interface `WheelEvent`

Introduced in this specification

The [WheelEvent](#) interface provides specific contextual information associated with [wheel](#) events. To create an instance of the [WheelEvent](#) interface, use the [WheelEvent](#) constructor, passing an optional [WheelEventInit](#) dictionary.

```
[Constructor(DOMString type, optional WheelEventInit eventInitDict)]  
interface WheelEvent : MouseEvent {  
  // DeltaModeCode  
  const unsigned long DOM_DELTA_PIXEL = 0x00;  
  const unsigned long DOM_DELTA_LINE = 0x01;  
  const unsigned long DOM_DELTA_PAGE = 0x02;  
  
  readonly attribute double deltaX;  
  readonly attribute double deltaY;  
  readonly attribute double deltaZ;  
  readonly attribute unsigned long deltaMode;  
};
```

WheelEvent . DOM_DELTA_PIXEL

The units of measurement for the [delta](#) MUST be pixels. This is the most typical case in most operating system and implementation configurations.

WheelEvent . DOM_DELTA_LINE

The units of measurement for the [delta](#) MUST be individual lines of text. This is the case for many form controls.

WheelEvent . DOM_DELTA_PAGE

The units of measurement for the [delta](#) MUST be pages, either defined as a single screen or as a demarcated page.

WheelEvent . deltaX

In user agents where the default action of the [wheel](#) event is to scroll, the value MUST be the measurement along the x-axis (in pixels, lines, or pages) to be scrolled in the case where the event is not cancelled. Otherwise, this is an implementation-specific measurement (in pixels, lines, or pages) of the movement of a wheel device around the x-axis.

The [un-initialized value](#) of this attribute MUST be 0.0.

WheelEvent . deltaY

In user agents where the default action of the [wheel](#) event is to scroll, the value MUST be the measurement along the y-axis (in pixels, lines, or pages) to be scrolled in the case where the event is not cancelled. Otherwise, this is an implementation-specific measurement (in pixels, lines, or pages) of the movement of a wheel device around the y-axis.

The [un-initialized value](#) of this attribute MUST be 0.0.

WheelEvent . deltaZ

In user agents where the default action of the [wheel](#) event is to scroll, the value MUST be the measurement along the z-axis (in pixels, lines, or pages) to be scrolled in the case where the event is not cancelled.

Otherwise, this is an implementation-specific measurement (in pixels, lines, or pages) of the movement of a wheel device around the z-axis.

The [un-initialized value](#) of this attribute MUST be 0.0.

WheelEvent . deltaMode

The `deltaMode` attribute contains an indication of the units of measurement for the [delta](#) values. The default value is [DOM_DELTA_PIXEL](#) (pixels).

This attribute MUST be set to one of the `DOM_DELTA` constants to indicate the units of measurement for the [delta](#) values. The precise measurement is specific to device, operating system, and application configurations.

The [un-initialized value](#) of this attribute MUST be 0.

```
dictionary WheelEventInit : MouseEventInit {  
  double deltaX = 0.0;  
  double deltaY = 0.0;  
  double deltaZ = 0.0;  
  unsigned long deltaMode = 0;  
};
```

WheelEventInit . deltaX

See `deltaZ` attribute.

WheelEventInit . deltaY

See `deltaZ` attribute.

WheelEventInit . deltaZ

Initializes the `deltaZ` attribute of the `WheelEvent` object. Relative positive values for this attribute (as well as the `deltaX` and `deltaY` attributes) are given by a right-hand coordinate system where the X, Y, and Z axes are directed towards the right-most edge, bottom-most edge, and farthest depth (away from the user) of the document, respectively. Negative relative values are in the respective opposite directions.

WheelEventInit . deltaMode

Initializes the `deltaMode` attribute on the `WheelEvent` object to the enumerated values 0, 1, or 2, which represent the amount of pixels scrolled ([DOM_DELTA_PIXEL](#)), lines scrolled ([DOM_DELTA_LINE](#)), or pages scrolled ([DOM_DELTA_PAGE](#)) if the [rotation](#) of the wheel would have resulted in scrolling.

4.4.2. Wheel Event Types

4.4.2.1. *wheel*

Type	wheel
Interface	WheelEvent
Sync / Async	Async
Bubbles	Yes
Trusted Targets	Element

Cancelable	Yes
Composed	Yes
Default action	Scroll (or zoom) the document
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : topmost event target • UIEvent.view : Window • UIEvent.detail : 0 • MouseEvent.screenX : if the wheel is associated with a pointing device, the value based on the pointer position on the screen, otherwise 0 • MouseEvent.screenY : if the wheel is associated with a pointing device, the value based on the pointer position on the screen, otherwise 0 • MouseEvent.clientX : if the wheel is associated with a pointing device, the value based on the pointer position within the viewport, otherwise 0 • MouseEvent.clientY : if the wheel is associated with a pointing device, the value based on the pointer position within the viewport, otherwise 0 • MouseEvent.altKey : true if Alt modifier was active, otherwise false • MouseEvent.ctrlKey : true if Control modifier was active, otherwise false • MouseEvent.shiftKey : true if Shift modifier was active, otherwise false • MouseEvent.metaKey : true if Meta modifier was active, otherwise false • MouseEvent.button : if wheel is associated with a pointing device, value based on current button pressed, otherwise 0 • MouseEvent.buttons : if wheel is associated with a pointing device, value based on all buttons current depressed, 0 if no buttons pressed • MouseEvent.relatedTarget : indicates the event target the pointing device is pointing at, if any. • WheelEvent.deltaX : expected amount that the page will scroll along the x-axis according to the deltaMode units; or an implementation-specific value of movement of a wheel around the x-axis • WheelEvent.deltaY : expected amount that the page will scroll along the y-axis according to the deltaMode units; or an implementation-specific value of movement of a wheel around the y-axis • WheelEvent.deltaZ : expected amount that the page will scroll along the z-axis according to the deltaMode units; or an implementation-specific value of movement of a wheel around the z-axis • WheelEvent.deltaMode : unit indicator (pixels, lines, or pages) for the deltaX, deltaY, and deltaZ attributes

A [user agent](#) MUST dispatch this event when a mouse wheel has been rotated around any axis, or when an equivalent input device (such as a mouse-ball, certain tablets or touchpads, etc.) has emulated such an action. Depending on the platform and input device, diagonal wheel [deltas](#) MAY be delivered either as a single [wheel](#) event with multiple non-zero axes or as separate [wheel](#) events for each non-zero axis. The typical [default action](#) of the [wheel](#) event type is to scroll (or in some cases, zoom) the document by the indicated amount. If this event is canceled, the implementation MUST NOT scroll or zoom the document (or perform whatever other implementation-specific default action is associated with this event type).

NOTE:

In some [user agents](#), or with some input devices, the speed that the wheel has been turned can affect the [delta](#) values, with a faster speed producing a higher [delta](#) value.

4.5. Input Events

Input events are sent as notifications whenever the DOM is being updated.

4.5.1. Interface `InputEvent`

Introduced in DOM Level 3

```
[Constructor(DOMString type, optional InputEventInit eventInitDict)]  
interface InputEvent : UIEvent {  
  readonly attribute DOMString data;  
  readonly attribute boolean isComposing;  
};
```

InputEvent . data

data holds the value of the characters generated by an input method. This MAY be a single Unicode character or a non-empty sequence of Unicode characters [\[Unicode\]](#). Characters SHOULD be normalized as defined by the Unicode normalization form *NFC*, defined in [\[UAX15\]](#). This attribute MAY contain the [empty string](#).

The [un-initialized value](#) of this attribute MUST be "" (the empty string).

InputEvent . isComposing

true if the input event occurs as part of a composition session, i.e., after a [compositionstart](#) event and before the corresponding [compositionend](#) event. The [un-initialized value](#) of this attribute MUST be false.

```
dictionary InputEventInit : UIEventInit {  
  DOMString data = "";  
  boolean isComposing = false;  
};
```

InputEventInit . data

Initializes the *data* attribute of the `InputEvent` object.

InputEventInit . isComposing

Initializes the *isComposing* attribute of the `InputEvent` object.

4.5.2. Input Event Order

The input events defined in this specification MUST occur in a set order relative to one another.

Event Type	Notes
1 beforeinput	<i>DOM element is updated</i>
2 input	

4.5.3. Input Event Types

4.5.3.1. *beforeinput*

Type	beforeinput
Interface	InputEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element (specifically: control types such as <code>HTMLInputElement</code> , etc.) or any <code>Element</code> with <code>contenteditable</code> attribute enabled.
Cancelable	Yes
Composed	Yes
Default action	Update the DOM element
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : event target that is about to be updated • UIEvent.view : Window • UIEvent.detail : 0 • InputEvent.data : the string containing the data that will be added to the element, which MAY be the empty string if the content has been deleted • InputEvent.isComposing : <code>true</code> if this event is dispatched during a dead key sequence or while an input method editor is active (such that composition events are being dispatched); <code>false</code> otherwise.

A [user agent](#) MUST dispatch this event when the DOM is about to be updated.

4.5.3.2. *input*

Type	input
Interface	InputEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element (specifically: control types such as <code>HTMLInputElement</code> , etc.) or any <code>Element</code> with <code>contenteditable</code> attribute enabled.
Cancelable	No
Composed	Yes
Default action	None
Context	

(trusted events)

- [Event.target](#) : [event target](#) that was just updated
- [UIEvent.view](#) : [Window](#)
- [UIEvent.detail](#) : 0
- [InputEvent.data](#) : the string containing the data that was added to the element, which MAY be the [empty string](#) if the content has been deleted
- [InputEvent.isComposing](#) : true if this event is dispatched during a [dead key](#) sequence or while an [input method editor](#) is active (such that [composition events](#) are being dispatched); false otherwise.

A [user agent](#) MUST dispatch this event immediately after the DOM has been updated.

4.6. Keyboard Events

Keyboard events are device dependent, i.e., they rely on the capabilities of the input devices and how they are mapped in the operating systems. Refer to [Keyboard events and key values](#) for more details, including examples on how Keyboard Events are used in combination with Composition Events. Depending on the character generation device, keyboard events might not be generated.

NOTE:

Keyboard events are only one modality of providing textual input. For editing scenarios, consider also using the [InputEvent](#) as an alternate to (or in addition to) keyboard events.

4.6.1. Interface KeyboardEvent

Introduced in this specification

The [KeyboardEvent](#) interface provides specific contextual information associated with keyboard devices. Each keyboard event references a key using a value. Keyboard events are commonly directed at the element that has the focus.

The [KeyboardEvent](#) interface provides convenient attributes for some common modifiers keys: [ctrlKey](#), [shiftKey](#), [altKey](#), [metaKey](#). These attributes are equivalent to using the method [getModifierState\(\)](#) with `[Control]`, `[Shift]`, `[Alt]`, or `[Meta]` respectively. To create an instance of the [KeyboardEvent](#) interface, use the [KeyboardEvent](#) constructor, passing an optional [KeyboardEventInit](#) dictionary.

```
[Constructor(DOMString type, optional KeyboardEventInit eventInitDict)]
interface KeyboardEvent : UIEvent {
    // KeyLocationCode
    const unsigned long DOM_KEY_LOCATION_STANDARD = 0x00;
    const unsigned long DOM_KEY_LOCATION_LEFT = 0x01;
    const unsigned long DOM_KEY_LOCATION_RIGHT = 0x02;
    const unsigned long DOM_KEY_LOCATION_NUMPAD = 0x03;

    readonly attribute DOMString key;
    readonly attribute DOMString code;
    readonly attribute unsigned long location;

    readonly attribute boolean ctrlKey;
    readonly attribute boolean shiftKey;
    readonly attribute boolean altKey;
    readonly attribute boolean metaKey;

    readonly attribute boolean repeat;
    readonly attribute boolean isComposing;

    boolean getModifierState(DOMString keyArg);
};
```

KeyboardEvent . DOM_KEY_LOCATION_STANDARD

The key activation **MUST NOT** be distinguished as the left or right version of the key, and (other than the **NumLock** key) did not originate from the numeric keypad (or did not originate with a virtual key corresponding to the numeric keypad).

EXAMPLE 10:

The **Q** key on a PC 101 Key US keyboard.

The **NumLock** key on a PC 101 Key US keyboard.

The **1** key on a PC 101 Key US keyboard located in the main section of the keyboard.

KeyboardEvent . DOM_KEY_LOCATION_LEFT

The key activated originated from the left key location (when there is more than one possible location for this key).

EXAMPLE 11:

The left **Control** key on a PC 101 Key US keyboard.

KeyboardEvent . DOM_KEY_LOCATION_RIGHT

The key activation originated from the right key location (when there is more than one possible location for this key).

EXAMPLE 12:

The right **Shift** key on a PC 101 Key US keyboard.

KeyboardEvent . DOM_KEY_LOCATION_NUMPAD

The key activation originated on the numeric keypad or with a virtual key corresponding to the numeric keypad (when there is more than one possible location for this key). Note that the `NumLock` key should always be encoded with a [location](#) of `DOM_KEY_LOCATION_STANDARD`.

EXAMPLE 13:

The `1` key on a PC 101 Key US keyboard located on the numeric pad.

KeyboardEvent . key

key holds the key value of the key pressed. If the value has a printed representation, it MUST be a non-empty Unicode character string, conforming to the [algorithm for determining the key value](#) defined in this specification. If the value is a control key which has no printed representation, it MUST be one of the key values defined in the [key values set](#), as determined by the [algorithm for determining the key value](#). Implementations that are unable to identify a key MUST use the key value `Unidentified`.

NOTE:

The key attribute is not related to the legacy `keyCode` attribute and does not have the same set of values.

The [un-initialized value](#) of this attribute MUST be "" (the empty string).

KeyboardEvent . code

code holds a string that identifies the physical key being pressed. The value is not affected by the current keyboard layout or modifier state, so a particular key will always return the same value.

The [un-initialized value](#) of this attribute MUST be "" (the empty string).

KeyboardEvent . location

The [location](#) attribute contains an indication of the logical location of the key on the device.

This attribute MUST be set to one of the `DOM_KEY_LOCATION` constants to indicate the location of a key on the device.

If a [user agent](#) allows keys to be remapped, then the [location](#) value for a remapped key MUST be set to a value which is appropriate for the new key. For example, if the `"ControlLeft"` key is mapped to the `"KeyQ"` key, then the [location](#) attribute MUST be set to `DOM_KEY_LOCATION_STANDARD`. Conversely, if the `"KeyQ"` key is remapped to one of the `Control` keys, then the [location](#) attribute MUST be set to either `DOM_KEY_LOCATION_LEFT` or `DOM_KEY_LOCATION_RIGHT`.

The [un-initialized value](#) of this attribute MUST be 0.

KeyboardEvent . ctrlKey

true if the `Control` (control) key modifier was active.

The [un-initialized value](#) of this attribute MUST be false.

KeyboardEvent . shiftKey

true if the shift (`Shift`) key modifier was active.

The [un-initialized value](#) of this attribute MUST be `false`.

KeyboardEvent . altKey

true if the `Alt` (alternative) (or "`Option`") key modifier was active. The [un-initialized value](#) of this attribute MUST be `false`.

KeyboardEvent . metaKey

true if the meta (`Meta`) key modifier was active.

NOTE:

The "`Command`" ("`⌘`") key modifier on Macintosh systems is represented using this key modifier.

The [un-initialized value](#) of this attribute MUST be `false`.

KeyboardEvent . repeat

true if the key has been pressed in a sustained manner. Holding down a key MUST result in the repeating the events `keydown`, `beforeinput`, `input` in this order, at a rate determined by the system configuration. For mobile devices which have *long-key-press* behavior, the first key event with a `repeat` attribute value of `true` MUST serve as an indication of a *long-key-press*. The length of time that the key MUST be pressed in order to begin repeating is configuration-dependent.

The [un-initialized value](#) of this attribute MUST be `false`.

KeyboardEvent . isComposing

true if the key event occurs as part of a composition session, i.e., after a `compositionstart` event and before the corresponding `compositionend` event. The [un-initialized value](#) of this attribute MUST be `false`.

KeyboardEvent . getModifierState()

Queries the state of a modifier using a key value. See [Modifier keys](#) for a list of valid (case-sensitive) arguments to this method.

DOMString keyArg

A modifier key value. Valid modifier keys are defined in the [Modifier Keys](#) table in [\[UIEvents-Key\]](#).

Returns `true` if it is a modifier key and the modifier is activated, `false` otherwise.

NOTE:

If an application wishes to distinguish between right and left modifiers, this information could be deduced using keyboard events and [location](#).

```
dictionary KeyboardEventInit : EventModifierInit {  
  DOMString key = "";  
  DOMString code = "";  
  unsigned long location = 0;  
  boolean repeat = false;  
  boolean isComposing = false;  
};
```

KeyboardEventInit . key

Initializes the `key` attribute of the `KeyboardEvent` object to the unicode character string representing the meaning of a key after taking into account all keyboard modifications (such as shift-state). This value is the final effective value of the key. If the key is not a printable character, then it should be one of the key values defined in [\[UIEvents-Key\]](#).

KeyboardEventInit . code

Initializes the `code` attribute of the `KeyboardEvent` object to the unicode character string representing the key that was pressed, ignoring any keyboard modifications such as keyboard layout. This value should be one of the code values defined in [\[UIEvents-Code\]](#).

KeyboardEventInit . location

Initializes the [location](#) attribute of the `KeyboardEvent` object to one of the following location numerical constants:

- [DOM_KEY_LOCATION_STANDARD](#) (numerical value 0)
- [DOM_KEY_LOCATION_LEFT](#) (numerical value 1)
- [DOM_KEY_LOCATION_RIGHT](#) (numerical value 2)
- [DOM_KEY_LOCATION_NUMPAD](#) (numerical value 3)

KeyboardEventInit . repeat

Initializes the `repeat` attribute of the `KeyboardEvent` object. This attribute should be set to `true` if the current `KeyboardEvent` is considered part of a repeating sequence of similar events caused by the long depression of any single key, `false` otherwise.

KeyboardEventInit . isComposing

Initializes the `isComposing` attribute of the `KeyboardEvent` object. This attribute should be set to `true` if the event being constructed occurs as part of a composition sequence, `false` otherwise.

Legacy keyboard event implementations include three additional attributes, `keyCode`, `charCode`, and `which`. The `keyCode` attribute indicates a numeric value associated with a particular key on a computer keyboard, while the `charCode` attribute indicates the [ASCII](#) value of the character associated with that key (which might be the same as the `keyCode` value) and is applicable only to keys that produce a [character value](#).

In practice, `keyCode` and `charCode` are inconsistent across platforms and even the same implementation on different operating systems or using different localizations. This specification does not define values for either `keyCode` or `charCode`, or behavior for `charCode`. In conforming UI Events implementations, content authors can instead use [key](#) and [code](#).

For more information, see the informative appendix on [Legacy key attributes](#).

NOTE:

For compatibility with existing content, virtual keyboards, such as software keyboards on screen-based input devices, are expected to produce the normal range of keyboard events, even though they do not possess physical keys.

NOTE:

In some implementations or system configurations, some key events, or their values, might be suppressed by the [IME](#) in use.

4.6.2. Keyboard Event Key Location

The [location](#) attribute can be used to disambiguate between [key](#) values that can be generated by different physical keys on the keyboard, for example, the left and right [Shift](#) key or the physical arrow keys vs. the numpad arrow keys (when [NumLock](#) is off). The following table defines the valid [location](#) values for the special keys that have more than one location on the keyboard:

KeyboardEvent.key	Valid location values
"Shift" , "Control" , "Alt" , "Meta"	DOM_KEY_LOCATION_LEFT , DOM_KEY_LOCATION_RIGHT
"ArrowDown" , "ArrowLeft" , "ArrowRight" , "ArrowUp"	DOM_KEY_LOCATION_STANDARD , DOM_KEY_LOCATION_NUMPAD
"End" , "Home" , "PageDown" , "PageUp"	DOM_KEY_LOCATION_STANDARD , DOM_KEY_LOCATION_NUMPAD
"0" , "1" , "2" , "3" , "4" , "5" , "6" , "7" , "8" , "9" , "." , "Enter" , "+" , "-" , "*" , "/"	DOM_KEY_LOCATION_STANDARD , DOM_KEY_LOCATION_NUMPAD

For all other keys not listed in this table, the [location](#) attribute MUST always be set to [DOM_KEY_LOCATION_STANDARD](#).

4.6.3. Keyboard Event Order

The keyboard events defined in this specification occur in a set order relative to one another, for any given key:

Event Type	Notes
1 keydown	
2 beforeinput	<i>(only for keys which produce a character value)</i> <i>Any default actions related to this key, such as inserting a character in to the DOM.</i>
3 input	<i>(only for keys which have updated the DOM)</i> <i>Any events as a result of the key being held for a sustained period (see below).</i>
4 keyup	

If the key is depressed for a sustained period, the following events MAY repeat at an environment-dependent rate:

Event Type	Notes
----------------------------	-----------------------

- 1 [keydown](#) (with [repeat](#) attribute set to `true`)
- 2 [beforeinput](#) (only for keys which produce a [character value](#))
Any [default actions](#) related to this key, such as inserting a character in to the DOM.
- 3 [input](#) (only for keys which have updated the DOM)

NOTE:

Typically, any [default actions](#) associated with any particular key are completed before the [keyup](#) event is dispatched. This might delay the [keyup](#) event slightly (though this is not likely to be a perceptible delay).

The [event target](#) of a key event is the currently focused element which is processing the keyboard activity. This is often an HTML input element or a textual element which is editable, but MAY be an element defined by the [host language](#) to accept keyboard input for non-text purposes, such as the activation of an accelerator key or trigger of some other behavior. If no suitable element is in focus, the event target will be the HTML [body element](#) if available, otherwise the [root element](#).

NOTE:

The [event target](#) might change between different key events. For example, a [keydown](#) event for the `Tab` key will likely have a different [event target](#) than the [keyup](#) event on the same keystroke.

4.6.4. Keyboard Event Types

4.6.4.1. *keydown*

Type	keydown
Interface	KeyboardEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	Varies: beforeinput and input events; launch text composition system ; blur and focus events; keypress event (if supported); activation behavior ; other event
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : focused element processing the key event or if no element focused, then the body element if available, otherwise the root element • UIEvent.view : Window • UIEvent.detail : 0 • KeyboardEvent.key : the key value of the key pressed. • KeyboardEvent.code : the code value associated with the key's physical placement on the keyboard. • KeyboardEvent.location : the location of the key on the device. • KeyboardEvent.altKey : true if <code>Alt</code> modifier was active, otherwise false • KeyboardEvent.shiftKey : true if <code>Shift</code> modifier was active, otherwise false • KeyboardEvent.ctrlKey : true if <code>Control</code> modifier was active, otherwise false

- [KeyboardEvent.metaKey](#) : true if **Meta** modifier was active, otherwise false
- [KeyboardEvent.repeat](#) : true if a key has been depressed long enough to trigger key repetition, otherwise false
- [KeyboardEvent.isComposing](#) : true if the key event occurs as part of a composition session, otherwise false

A [user agent](#) MUST dispatch this event when a key is pressed down. The [keydown](#) event type is device dependent and relies on the capabilities of the input devices and how they are mapped in the operating system. This event type MUST be generated after the [key mapping](#). This event type MUST be dispatched before the [beforeinput](#), [input](#), and [keyup](#) events associated with the same key.

The default action of the [keydown](#) event depends upon the key:

- If the key is associated with a character, the default action MUST be to dispatch a [beforeinput](#) event followed by an [input](#) event. In the case where the key which is associated with multiple characters (such as with a macro or certain sequences of dead keys), the default action MUST be to dispatch one set of [beforeinput](#) / [input](#) events for each character
- If the key is associated with a [text composition system](#), the default action MUST be to launch that system
- If the key is the **Tab** key, the default action MUST be to shift the document focus from the currently focused element (if any) to the new focused element, as described in [Focus Event Types](#)
- If the key is the **Enter** or **↵** key and the current focus is on a state-changing element, the default action MUST be to dispatch a [click](#) event, and a [DOMActivate](#) event if that event type is supported by the [user agent](#) (refer to [§3.5 Activation triggers and behavior](#) for more details)

If this event is canceled, the associated event types MUST NOT be dispatched, and the associated actions MUST NOT be performed.

NOTE:

The [keydown](#) and [keyup](#) events are traditionally associated with detecting any key, not just those which produce a [character value](#).

4.6.4.2. *keyup*

Type	keyup
Interface	KeyboardEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	None

Context (trusted events)	<ul style="list-style-type: none"> • Event.target : focused element processing the key event or if no element focused, then the body element if available, otherwise the root element • UIEvent.view : Window • UIEvent.detail : 0 • KeyboardEvent.key : the key value of the key pressed. • KeyboardEvent.code : the code value associated with the key's physical placement on the keyboard. • KeyboardEvent.location : the location of the key on the device. • KeyboardEvent.altKey : true if Alt modifier was active, otherwise false • KeyboardEvent.shiftKey : true if Shift modifier was active, otherwise false • KeyboardEvent.ctrlKey : true if Control modifier was active, otherwise false • KeyboardEvent.metaKey : true if Meta modifier was active, otherwise false • KeyboardEvent.repeat : false • KeyboardEvent.isComposing : true if the key event occurs as part of a composition session, otherwise false
-----------------------------	---

A [user agent](#) MUST dispatch this event when a key is released. The [keyup](#) event type is device dependent and relies on the capabilities of the input devices and how they are mapped in the operating system. This event type MUST be generated after the [key mapping](#). This event type MUST be dispatched after the [keydown](#), [beforeinput](#), and [input](#) events associated with the same key.

NOTE:

The [keydown](#) and [keyup](#) events are traditionally associated with detecting any key, not just those which produce a [character value](#).

4.7. Composition Events

Composition Events provide a means for inputting text in a supplementary or alternate manner than by Keyboard Events, in order to allow the use of characters that might not be commonly available on keyboard. For example, Composition Events might be used to add accents to characters despite their absence from standard US keyboards, to build up logograms of many Asian languages from their base components or categories, to select word choices from a combination of key presses on a mobile device keyboard, or to convert voice commands into text using a speech recognition processor. Refer to [§5 Keyboard events and key values](#) for examples on how Composition Events are used in combination with keyboard events.

Conceptually, a composition session consists of one [compositionstart](#) event, one or more [compositionupdate](#) events, and one [compositionend](#) event, with the value of the [data](#) attribute persisting between each “stage” of this event chain during each session.

NOTE:

Note: While a composition session is active, keyboard events can be dispatched to the DOM if the keyboard is the input device used with the composition session. See the [compositionstart](#) event details and [IME section](#) for relevant event ordering.

Not all [IME](#) systems or devices expose the necessary data to the DOM, so the active composition string (the “Reading Window” or “candidate selection menu option”) might not be available through this interface, in which case the selection MAY be represented by the [empty string](#).

4.7.1. Interface CompositionEvent

Introduced in this specification

The [CompositionEvent](#) interface provides specific contextual information associated with Composition Events.

To create an instance of the [CompositionEvent](#) interface, use the [CompositionEvent](#) constructor, passing an optional [CompositionEventInit](#) dictionary.

```
[Constructor(DOMString type, optional CompositionEventInit eventInitDict)]  
interface CompositionEvent : UIEvent {  
  readonly attribute DOMString data;  
};
```

CompositionEvent . data

data holds the value of the characters generated by an input method. This MAY be a single Unicode character or a non-empty sequence of Unicode characters [\[Unicode\]](#). Characters SHOULD be normalized as defined by the Unicode normalization form *NFC*, defined in [\[UAX15\]](#). This attribute MAY be the [empty string](#).

The [un-initialized value](#) of this attribute MUST be "" (the empty string).

```
dictionary CompositionEventInit : UIEventInit {  
  DOMString data = "";  
};
```

CompositionEvent . data

Initializes the *data* attribute of the CompositionEvent object to the characters generated by the IME composition.

4.7.2. Composition Event Order

The Composition Events defined in this specification MUST occur in the following set order relative to one another:

Event Type	Notes
1 compositionstart	
2 compositionupdate	Multiple events
3 compositionend	

4.7.3. Handwriting Recognition Systems

The following example describes a possible sequence of events when composing a text passage “text” with a handwriting recognition system, such as on a pen tablet, as modeled using Composition Events.

Event Type	<u>CompositionEvent</u> <u>data</u>	Notes
1 <u>compositionstart</u>	" "	
		<i>User writes word on tablet surface</i>
2 <u>compositionupdate</u>	"test"	
		<i>User rejects first word-match suggestion, selects different match</i>
3 <u>compositionupdate</u>	"text"	
4 <u>compositionend</u>	"text"	

4.7.4. Canceling Composition Events

If a keydown event is canceled then any Composition Events that would have fired as a result of that keydown SHOULD not be dispatched:

Event Type	Notes
1 <u>keydown</u>	The <u>default action</u> is prevented, e.g., by invoking <u>preventDefault()</u> . <i>No Composition Events are dispatched</i>
2 <u>keyup</u>	

If the initial compositionstart event is canceled then the text composition session SHOULD be terminated. Regardless of whether or not the composition session is terminated, the compositionend event MUST be sent.

Event Type	Notes
1 <u>keydown</u>	
2 <u>compositionstart</u>	The <u>default action</u> is prevented, e.g., by invoking <u>preventDefault()</u> . <i>No Composition Events are dispatched</i>
3 <u>compositionend</u>	
4 <u>keyup</u>	

4.7.5. Key Events During Composition

During the composition session, keydown and keyup events MUST still be sent, and these events MUST have the isComposing attribute set to `true`.

Event Type	<u>KeyboardEvent</u> <u>isComposing</u>	Notes
1 <u>keydown</u>	false	This is the key event that initiates the composition.
2 <u>compositionstart</u>		
3 <u>compositionupdate</u>		
4 <u>keyup</u>	true	
...		Any key events sent during the composition session MUST have <code>isComposing</code> set to <code>true</code> .
5 <u>keydown</u>	true	This is the key event that exits the composition.

6 [compositionend](#)
7 [keyup](#) false

4.7.6. Input Events During Composition

During the composition session, the [compositionupdate](#) MUST be dispatched after the [beforeinput](#) is sent, but before the [input](#) event is sent.

Event Type	Notes
1 beforeinput	
2 compositionupdate	<i>Any DOM updates occur at this point.</i>
3 input	

NOTE:

Most IMEs do not support canceling updates during a composition session.

The [beforeinput](#) and [input](#) events are sent along with the [compositionupdate](#) event whenever the DOM is updated as part of the composition. Since there are no DOM updates associated with the [compositionend](#) event, [beforeinput](#) and [input](#) events should not be sent at that time.

Event Type	Notes
1 beforeinput	<i>Canceling this will prevent the DOM update and the input event.</i>
2 compositionupdate	<i>Any DOM updates occur at this point.</i>
3 input	<i>Sent only if the DOM was updated.</i>
4 compositionend	

4.7.7. Composition Event Types

4.7.7.1. [compositionstart](#)

Type	compositionstart
Interface	CompositionEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Composed	Yes
Default action	Start a new composition session when a text composition system is enabled
Context (trusted events)	<ul style="list-style-type: none">• Event.target : focused element processing the composition• UIEvent.view : Window• UIEvent.detail : 0

- [CompositionEvent.data](#) : the original string being edited, otherwise the [empty string](#)

A [user agent](#) MUST dispatch this event when a [text composition system](#) is enabled and a new composition session is about to begin (or has begun, depending on the [text composition system](#)) in preparation for composing a passage of text. This event type is device-dependent, and MAY rely upon the capabilities of the text conversion system and how it is mapped into the operating system. When a keyboard is used to feed an input method editor, this event type is generated after a [keydown](#) event, but speech or handwriting recognition systems MAY send this event type without keyboard events. Some implementations MAY populate the [data](#) attribute of the [compositionstart](#) event with the text currently selected in the document (for editing and replacement). Otherwise, the value of the [data](#) attribute MUST be the [empty string](#).

This event MUST be dispatched immediately before a [text composition system](#) begins a new composition session, and before the DOM is modified due to the composition process. The default action of this event is for the [text composition system](#) to start a new composition session. If this event is canceled, the [text composition system](#) SHOULD discard the current composition session.

NOTE:

Canceling the [compositionstart](#) event type is distinct from canceling the [text composition system](#) itself (e.g., by hitting a cancel button or closing an [IME](#) window).

NOTE:

Some IMEs do not support cancelling an in-progress composition session (e.g., such as GTK which doesn't presently have such an API). In these cases, calling [preventDefault\(\)](#) will not stop this event's default action.

4.7.7.2. *compositionupdate*

Type	compositionupdate
Interface	CompositionEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	No
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : focused element processing the composition, null if not accessible • UIEvent.view : Window • UIEvent.detail : 0 • CompositionEvent.data : the string comprising the current results of the composition session, which MAY be the empty string if the content has been deleted

A [user agent](#) SHOULD dispatch this event during a composition session when a [text composition system](#) updates its active text passage with a new character, which is reflected in the string in [data](#).

In [text composition systems](#) which keep the ongoing composition in sync with the input control, the [compositionupdate](#) event MUST be dispatched before the control is updated.

Some [text composition systems](#) might not expose this information to the DOM, in which case this event will not fire during the composition process.

If the composition session is canceled, this event will be fired immediately before the [compositionend](#) event, and the [data](#) attribute will be set to the [empty string](#).

4.7.7.3. *compositionend*

Type	compositionend
Interface	CompositionEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	No
Composed	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : focused element processing the composition• UIEvent.view : Window• UIEvent.detail : 0• CompositionEvent.data : the string comprising the final result of the composition session, which MAY be the empty string if the content has been deleted or if the composition process has been canceled

A [user agent](#) MUST dispatch this event when a [text composition system](#) completes or cancels the current composition session, and the [compositionend](#) event MUST be dispatched after the control is updated.

This event is dispatched immediately after the [text composition system](#) completes the composition session (e.g., the [IME](#) is closed, minimized, switched out of focus, or otherwise dismissed, and the focus switched back to the [user agent](#)).

5. Keyboard events and key values

This section contains necessary information regarding keyboard events:

- Explanation of keyboard layout, mapping, and key values.
- Relations between keys, such as [dead keys](#) or modifiers keys.

- Relations between keyboard events and their default actions.
- The set of key values, and guidelines on how to extend this set.

NOTE:

This section uses Serbian and Kanji characters which could be misrepresented or unavailable in the PDF version or printed version of this specification.

5.1. Keyboard Input

This section is non-normative

The relationship of each key to the complete keyboard has three separate aspects, each of which vary among different models and configurations of keyboards, particularly for locale-specific reasons:

- **Mechanical layout:** the dimensions, size, and placement of the physical keys on the keyboard
- **Visual markings:** the labels (or *legends*) that mark each key
- **Functional mapping:** the abstract key-value association of each key.

This specification only defines the functional mapping, in terms of [key](#) values and [code](#) values, but briefly describes [key legends](#) for background.

5.1.1. Key Legends

This section is informative

The key legend is the visual marking that is printed or embossed on the *key cap* (the rectangular "cap" that covers the mechanical switch for the key). These markings normally consist of one or more characters that a keystroke on that key will produce (such as "G", "8", or "Ш"), or names or symbols which indicate that key's function (such as an upward-pointing arrow "↑" indicating **Shift**, or the string "Enter"). Keys are often referred to by this marking (e.g., "Press the "Shift" and "G" keys."). Note, however, that the visual appearance of the key has no bearing on its digital representation, and in many configurations may be completely inaccurate. Even the control and function keys, such as **Enter**, MAY be mapped to different functionality, or even mapped as character keys.

For historical reasons, the character keys are typically marked with the capital-letter equivalents of the character value they produce, e.g., the **G** key (the key marked with the glyph "G"), will produce the character value "g" when pressed without an active modifier key (e.g., **Shift**) or modifier state (e.g., **CapsLock**).

NOTE:

Many keyboards contain keys that do not normally produce any characters, even though the symbol might have a Unicode equivalent. For example, the `[Shift]` key might bear the symbol "↑", which has the [Unicode code point \u21E7](#), but pressing the `[Shift]` key will not produce this character value, and there is no [Unicode code point](#) for `[Shift]`.

5.2. Key codes

A key [code](#) is an attribute of a keyboard event that can be used to identify the physical key associated with the keyboard event. It is similar to USB Usage IDs in that it provides a low-level value (similar to a scancode) that is vendor-neutral.

The primary purpose of the [code](#) attribute is to provide a consistent and coherent way to identify keys based on their physical location. In addition, it also provides a stable name (unaffected by the current keyboard state) that uniquely identifies each key on the keyboard.

The list of valid [code](#) values is defined in the [UIEvents-Code](#).

5.2.1. Motivation for the [code](#) Attribute

The standard PC keyboard has a set of keys (which we refer to as *writing system keys*) that generate different [key](#) values based on the current keyboard layout selected by the user. This situation makes it difficult to write code that detects keys based on their physical location since the code would need to know which layout is in effect in order to know which [key](#) values to check for. A real-world example of this is a game that wants to use the "W", "A", "S" and "D" keys to control player movement. The [code](#) attribute solves this problem by providing a stable value to check that is *not affected by the current keyboard layout*.

In addition, the values in the [key](#) attribute depend as well on the current keyboard state. Because of this, the order in which keys are pressed and released in relation to modifier keys can affect the values stored in the [key](#) attribute. The [code](#) attribute solves this problem by providing a stable value that is *not affected by the current keyboard state*.

5.2.2. The Relationship Between [key](#) and [code](#)

[key](#)

The [key](#) attribute is intended for users who are interested in the meaning of the key being pressed, taking into account the current keyboard layout (and IME; [dead keys](#) are given a unique [key](#) value). Example use case: Detecting modified keys or bare modifier keys (e.g., to perform an action in response to a keyboard shortcut).

[code](#)

The [code](#) attribute is intended for users who are interested in the key that was pressed by the user, without any layout modifications applied. Example use case: Detecting WASD keys (e.g., for movement controls in a game) or trapping all keys (e.g., in a remote desktop client to send all keys to the remote host).

5.2.3. code Examples

EXAMPLE 14:

Handling the Left and Right Alt Keys

Keyboard Layout	KeyboardEvent	KeyboardEvent	Notes
	key	code	
US	"Alt"	"AltLeft"	DOM_KEY_LOCATION_LEFT
French	"Alt"	"AltLeft"	DOM_KEY_LOCATION_LEFT
US	"Alt"	"AltRight"	DOM_KEY_LOCATION_RIGHT
French	"AltGr"	"AltRight"	DOM_KEY_LOCATION_RIGHT

In this example, checking the key attribute permits matching `Alt` without worrying about which Alt key (left or right) was pressed. Checking the code attribute permits matching the right Alt key (`"AltRight"`) without worrying about which layout is currently in effect.

Note that, in the French example, the `Alt` and `AltGr` keys retain their left and right location, even though there is only one of each key.

EXAMPLE 15:

Handling the Single Quote Key

Keyboard Layout	KeyboardEvent	KeyboardEvent	Notes
	key	code	
US	"'"	"Quote"	
Japanese	"`"	"Quote"	
US Intl	"Dead"	"Quote"	

This example shows how dead key values are encoded in the attributes. The key values vary based on the current locale, whereas the code attribute returns a consistent value.

EXAMPLE 16:

Handling the "2" Key (with and without Shift pressed) on various keyboard layouts.

Keyboard Layout	KeyboardEvent	KeyboardEvent	Notes
	key	code	
US	"2"	"Digit2"	
US	"@"	"Digit2"	shiftKey
UK	"2"	"Digit2"	
UK	"'"	"Digit2"	shiftKey
French	"é"	"Digit2"	
French	"2"	"Digit2"	shiftKey

Regardless of the current locale or the modifier key state, pressing the key labelled "2" on a US keyboard always results in `"Digit2"` in the code attribute.

EXAMPLE 17:

Sequence of Keyboard Events : **Shift** and **2**

Compare the attribute values in the following two key event sequences. They both produce the "@" character on a US keyboard, but differ in the order in which the keys are released. In the first sequence, the order is: **Shift** (down), **2** (down), **2** (up), **Shift** (up).

Event Type	KeyboardEvent key	KeyboardEvent code	Notes
1 <u>keydown</u>	"Shift"	"ShiftLeft"	DOM_KEY_LOCATION_LEFT
2 <u>keydown</u>	"@"	"Digit2"	shiftKey
3 <u>keypress</u>	"@"	" "	(if supported)
4 <u>keyup</u>	"@"	"Digit2"	shiftKey
5 <u>keyup</u>	"Shift"	"ShiftLeft"	DOM_KEY_LOCATION_LEFT

In the second sequence, the Shift is released before the 2, resulting in the following event order: **Shift** (down), **2** (down), **Shift** (up), **2** (up).

Event Type	KeyboardEvent key	KeyboardEvent code	Notes
1 <u>keydown</u>	"Shift"	"ShiftLeft"	DOM_KEY_LOCATION_LEFT
2 <u>keydown</u>	"@"	"Digit2"	shiftKey
3 <u>keypress</u>	"@"	" "	(if supported)
4 <u>keyup</u>	"Shift"	"ShiftLeft"	DOM_KEY_LOCATION_LEFT
5 <u>keyup</u>	"2"	"Digit2"	

Note that the values contained in the key attribute does not match between the keydown and keyup events for the "2" key. The code attribute provides a consistent value that is not affected by the current modifier state.

5.2.4. code and Virtual Keyboards

The usefulness of the code attribute is less obvious for virtual keyboards (and also for remote controls and chording keyboards). In general, if a virtual (or remote control) keyboard is mimicking the layout and functionality of a standard keyboard, then it MUST also set the code attribute as appropriate. For keyboards which are not mimicking the layout of a standard keyboard, then the code attribute MAY be set to the closest match on a standard keyboard or it MAY be left undefined.

For virtual keyboards with keys that produce different values based on some modifier state, the code value should be the key value generated when the button is pressed while the device is in its factory-reset state.

5.3. Keyboard Event key Values

A key value is a DOMString that can be used to indicate any given key on a keyboard, regardless of position or state, by the value it produces. These key values MAY be used as return values for keyboard events generated by the implementation, or as input values by the content author to specify desired input (such as for keyboard shortcuts).

The list of valid key values is defined in [\[UIEvents-Key\]](#).

Key values can be used to detect the value of a key which has been pressed, using the [key](#) attribute. Content authors can retrieve the [character value](#) of upper- or lower-case letters, number, symbols, or other character-producing keys, and also the [key value](#) of control keys, modifier keys, function keys, or other keys that do not generate characters. These values can be used for monitoring particular input strings, for detecting and acting on modifier key input in combination with other inputs (such as a mouse), for creating virtual keyboards, or for any number of other purposes.

Key values can also be used by content authors in string comparisons, as values for markup attributes (such as the HTML `accesskey`) in conforming [host languages](#), or for other related purposes. A conforming [host language](#) SHOULD allow content authors to use either of the two equivalent string values for a key value: the [character value](#), or the [key value](#).

NOTE:

While implementations will use the most relevant value for a key independently of the platform or keyboard layout mappings, content authors can not make assumptions on the ability of keyboard devices to generate them. When using keyboard events and key values for shortcut-key combinations, content authors can “consider using numbers and function keys (`[F4]`, `[F5]`, and so on) instead of letters” ([\[DWW95\]](#)) given that most keyboard layouts will provide keys for those.

A key value does not indicate a specific key on the physical keyboard, nor does it reflect the character printed on the key. A key value indicates the current value of the event with consideration to the current state of all active keys and key input modes (including shift modes), as reflected in the operating-system mapping of the keyboard and reported to the implementation. In other words, the key value for the key marked `"0"` on a [QWERTY](#) keyboard has the key value `"o"` in an unshifted state and `"O"` in a shifted state. Because a user can map their keyboard to an arbitrary custom configuration, the content author is encouraged not to assume that a relationship exists between the shifted and unshifted states of a key and the majuscule form (uppercase or capital letters) and minuscule form (lowercase or small letters) of a character representation, but is encouraged instead to use the value of the [key](#) attribute. For example, the Standard "102" Keyboard layout depicted in [\[UIEvents-Code\]](#) illustrates one possible set of [key mappings](#) on one possible keyboard layout. Many others exist, both standard and idiosyncratic.

NOTE:

To simplify [dead key](#) support, when the operating-system mapping of the keyboard is handling a [dead key](#) state, the current state of the dead key sequence is not reported via the [key](#) attribute. Rather, a key value of `"Dead"` is reported. Instead, implementations generate [composition events](#) which contain the intermediate state of the dead key sequence reported via the [data](#) attribute. As in the previous example, the key value for the key marked `[0]` on a [QWERTY](#) keyboard has a [data](#) value of `'ö'` in an unshifted state during a dead-key operation to add an umlaut diacritic, and `'Ö'` in a shifted state during a dead-key operation to add an umlaut diacritic.

It is also important to note that there is not a one-to-one relationship between key event states and key values. A particular key value might be associated with multiple keys. For example, many standard keyboards contain more than one key with the `Shift` key value (normally distinguished by the `location` values `DOM_KEY_LOCATION_LEFT` and `DOM_KEY_LOCATION_RIGHT`) or `8` key value (normally distinguished by the `location` values `DOM_KEY_LOCATION_STANDARD` and `DOM_KEY_LOCATION_NUMPAD`), and user-configured custom keyboard layouts MAY duplicate any key value in multiple key-state scenarios (note that `location` is intended for standard keyboard layouts, and cannot always indicate a meaningful distinction).

Finally, the meaning of any given character representation is context-dependent and complex. For example, in some contexts, the asterisk (star) glyph ("`*`") represents a footnote or emphasis (when bracketing a passage of text). However, in some documents or executable programs it is equivalent to the mathematical multiplication operation, while in other documents or executable programs, that function is reserved for the multiplication symbol ("`×`", Unicode value `\u00D7`) or the Latin small letter `x` (due to the lack of a multiplication key on many keyboards and the superficial resemblance of the glyphs "`×`" and "`x`"). Thus, the semantic meaning or function of character representations is outside the scope of this specification.

5.3.1. Key Values and Unicode

The `character values` described in this specification are Unicode `[Unicode]` codepoints, and as such, have certain advantages.

The most obvious advantage is that it allows the content author to use the full range of internationalized language functionality available in the implementation, regardless of the limitations of the text input devices on the system. This opens up possibilities for virtual keyboards and Web-application-based `input method editors`.

Another benefit is that it allows the content author to utilize the Unicode general category properties programmatically.

With legacy keyboard event attributes such as `keyCode` and `charCode`, content authors are forced to filter key input using cryptic, platform- and implementation-specific numeric codes, with poor internationalization, such as the following pseudocode:

EXAMPLE 18:

```
-  
if ( ( event.charCode == 45 || event.charCode == 36 ) ||  
    ( event.charCode >= 48 && event.charCode <= 57 ) ||  
    ( event.charCode >= 96 && event.charCode <= 105 ) ) {  
    // minus sign, dollar sign, and numeric characters from keyboard and numpad  
    // ...  
}  
else if ( ( event.charCode >= 65 && event.charCode <= 90 ) ||  
    ( event.charCode >= 97 && event.charCode <= 122 ) ) {  
    // alphabetic characters from Latin character set, A-Z, a-z  
    // ...  
}  
else {  
    // ...  
}
```

With key values and regular expressions, however, content authors can support selective and intuitive ranges for key-based input, in a cross-platform manner with advanced internationalization support, such as the following pseudocode:

EXAMPLE 19:

```
-  
if ( event.key == "-" || event.key.match("\p{Sc}") || event.key.match("\p{N}") ) {  
    // minus sign, any currency symbol, and numeric characters (regardless of key loc  
    // ...  
}  
else if ( event.key.match("\p{L}") ) {  
    // alphabetic characters from any language, upper and lower case  
    // ...  
}  
else {  
    // ...  
}
```

In addition, because Unicode [categorizes](#) each assigned [code point](#) into a group of code points used by a particular human writing system, even more advanced capabilities are possible.

EXAMPLE 20:

A content author can match characters from a particular human script (e.g., Tibetan) using a regular expression such as `\p{Tibetan}`, to filter out other characters, or discover if a [code point](#) is in a certain code block (range of code points), using a regular expression like `\p{InCyrillic}`.

To facilitate this, implementations SHOULD support Unicode range detection using regular expressions, in a manner such as the Perl Compatible Regular Expressions (PCRE) [\[PCRE\]](#).

5.3.2. Modifier keys

Keyboard input uses modifier keys to change the normal behavior of a key. Like other keys, modifier keys generate [keydown](#) and [keyup](#) events, as shown in the example below. Some modifiers are activated while the key is being pressed down or maintained pressed such as [Alt](#), [Control](#), [Shift](#), [AltGraph](#), or [Meta](#). Others modifiers are activated depending on their state such as [CapsLock](#), [NumLock](#), or [ScrollLock](#). Change in the state happens when the modifier key is being pressed down. The [KeyboardEvent](#) interface provides convenient attributes for some common modifiers keys: [ctrlKey](#), [shiftKey](#), [altKey](#), [metaKey](#). Some operating systems simulate the [AltGraph](#) modifier key with the combination of the [Alt](#) and [Control](#) modifier keys. Implementations are encouraged to use the [AltGraph](#) modifier key.

EXAMPLE 21:

This example describes a possible sequence of events associated with the generation of the Unicode character Q (Latin Capital Letter Q, [Unicode code point \u0051](#)) on a US keyboard using a US mapping:

Event Type	KeyboardEvent key	Modifiers	Notes
1 keydown	"Shift"	shiftKey	
2 keydown	"Q"	shiftKey	Latin Capital Letter Q
3 beforeinput			
4 input			
5 keyup	"Q"	shiftKey	
6 keyup	"Shift"		

EXAMPLE 22:

This example describes an alternate sequence of keys to the example above, where the [Shift](#) key is released before the [Q](#) key. The key value for the [Q](#) key will revert to its unshifted value for the [keyup](#) event:

Event Type	KeyboardEvent key	Modifiers	Notes
1 keydown	"Shift"	shiftKey	
2 keydown	"Q"	shiftKey	Latin Capital Letter Q
3 beforeinput			
4 input			
5 keyup	"Shift"		
6 keyup	"q"		Latin Small Letter Q

EXAMPLE 23:

The following example describes a possible sequence of keys that does not generate a Unicode character (using the same configuration as the previous example):

Event Type	KeyboardEvent key	Modifiers	Notes
1 keydown	"Control"	ctrlKey	
2 keydown	"v"	ctrlKey	Latin Small Letter V <i>No beforeinput or input events are generated.</i>
3 keyup	"v"	ctrlKey	Latin Small Letter V
4 keyup	"Control"		

EXAMPLE 24:

The following example shows the sequence of events when both Shift and Control are pressed:

Event Type	KeyboardEvent key	Modifiers	Notes
1 keydown	"Control"	ctrlKey	
2 keydown	"Shift"	ctrlKey, shiftKey	
3 keydown	"V"	ctrlKey, shiftKey	Latin Capital Letter V <i>No beforeinput or input events are generated.</i>
4 keyup	"V"	ctrlKey, shiftKey	Latin Capital Letter V
5 keyup	"Shift"	ctrlKey	
6 keyup	"Control"		

EXAMPLE 25:

For non-US keyboard layouts, the sequence of events is the same, but the value of the key is based on the current keyboard layout. This example shows a sequence of events when an Arabic keyboard layout is used:

Event Type	KeyboardEvent key	Modifiers	Notes
1 keydown	"Control"	ctrlKey	
2 keydown	"ر"	ctrlKey	Arabic Letter Reh <i>No beforeinput or input events are generated.</i>
3 keyup	"ر"	ctrlKey	Arabic Letter Reh
4 keyup	"Control"		

NOTE:

The value in the [keydown](#) and [keyup](#) events varies based on the current keyboard layout in effect when the key is pressed. This means that the ⌫ key on a US layout and the ⌫ key on an Arabic layout will generate different events even though they are the same physical key. To identify these events as coming from the same physical key, you will need to make use of the [code](#) attribute.

In some cases, [modifier keys](#) change the [key](#) value for a key event. For example, on some MacOS keyboards, the key labeled "delete" functions the same as the Backspace key on the Windows OS when unmodified, but when

modified by the **[Fn]** key, acts as the **[Delete]** key, and the value of key will match the most appropriate function of the key in its current modified state.

5.3.3. Dead keys

Some keyboard input uses dead keys for the input of composed character sequences. Unlike the handwriting sequence, in which users enter the base character first, keyboard input requires to enter a special state when a dead key is pressed and emit the character(s) only when one of a limited number of “legal” base character is entered.

NOTE:

The MacOS and Linux operating systems use input methods to process dead keys.

The dead keys (across all keyboard layouts and mappings) are represented by the key value **[Dead]**. In response to any dead key press, composition events must be dispatched by the user agent and the compositionupdate event's data value must be the character value of the current state of the dead key combining sequence.

While Unicode combining characters always follow the handwriting sequence, with the combining character trailing the corresponding letter, typical dead key input MAY reverse the sequence, with the combining character before the corresponding letter. For example, the word *naïve*, using the combining diacritic *¨*, would be represented sequentially in Unicode as *nai¨ve*, but MAY be typed *na¨ive*. The sequence of keystrokes \u0302 (Combining Circumflex Accent key) and \u0065 (key marked with the Latin Small Letter E) will likely produce (on a French keyboard using a french mapping and without any modifier activated) the Unicode character **"ê"** (Latin Small Letter E With Circumflex), as preferred by the Unicode Normalization Form *NFC*.

EXAMPLE 26:

Event Type	KeyboardEvent <u>key</u>	KeyboardEvent <u>isComposing</u>	CompositionEvent <u>data</u>	Notes
1 <u>keydown</u>	"Dead"	false		Combining Circumflex Accent (Dead Key)
2 <u>compositionstart</u>			" "	
3 <u>compositionupdate</u>			\u0302	
4 <u>keyup</u>	"Dead"	true		
5 <u>keydown</u>	"ê"	true		
6 <u>compositionupdate</u>			"ê"	
7 <u>compositionend</u>			"ê"	
8 <u>keyup</u>	"e"	false		Latin Small Letter E

NOTE:

In the second keydown event (step 5), the key value (assuming the event is not suppressed) will not be **"e"** (Latin Small Letter E key) under normal circumstances because the value delivered to the user agent will already be modified by the dead key operation.

This process might be aborted when a user types an unsupported base character (that is, a base character for which the which the active diacritical mark is not available) after pressing a [dead key](#):

EXAMPLE 27:

Event Type	KeyboardEvent key	KeyboardEvent isComposing	CompositionEvent data	Notes
1 keydown	"Dead"	false		Combining Circumflex Accent (Dead Key)
2 compositionstart			" "	
3 compositionupdate			\u0302	
4 keyup	"Dead"	true		Latin Small Letter Q
5 keydown	"q"	true		
6 compositionupdate			" "	
7 compositionend			" "	
8 keyup	"q"	false		

5.3.4. Input Method Editors

This specification includes a model for [input method editors](#) (IMEs), through the [CompositionEvent](#) interface and events. However, Composition Events and Keyboard Events do not necessarily map as a one-to-one relationship. As an example, receiving a [keydown](#) for the [Accept](#) key value does not necessarily imply that the text currently selected in the [IME](#) is being accepted, but indicates only that a keystroke happened, disconnected from the [IME](#) Accept functionality (which would normally result in a [compositionend](#) event in most [IME](#) systems). Keyboard events cannot be used to determine the current state of the input method editor, which can be obtained through the [data](#) attribute of the [CompositionEvent](#) interface. Additionally, [IME](#) systems and devices vary in their functionality, and in which keys are used for activating that functionality, such that the [Convert](#) and [Accept](#) keys MAY be represented by other available keys. Keyboard events correspond to the events generated by the input device after the keyboard layout mapping.

NOTE:

In some implementations or system configurations, some key events, or their values, might be suppressed by the [IME](#) in use.

The following example describes a possible sequence of keys to generate the Unicode character "市" (Kanji character, part of CJK Unified Ideographs) using Japanese input methods. This example assumes that the input method editor is activated and in the Japanese-Romaji input mode. The keys [Convert](#) and [Accept](#) MAY be replaced by others depending on the input device in use and the configuration of the IME, e.g., it can be respectively [\u0020](#) (Space key) and [Enter](#).

NOTE:

"詩" ("poem") and "市" ("city") are homophones, both pronounced し ("shi"/"si"), so the user needs to use the [Convert](#) key to select the proper option.

EXAMPLE 28:

	Event Type	KeyboardEvent <u>key</u>	KeyboardEvent <u>isComposing</u>	CompositionEvent <u>data</u>	Notes
1	<u>keydown</u>	"s"	false		Latin Small Letter S
2	<u>compositionstart</u>			" "	
3	<u>beforeinput</u>				
4	<u>compositionupdate</u>			"s"	DOM is updated
5	<u>input</u>				
6	<u>keyup</u>	"s"	true		
7	<u>keydown</u>	"i"	true		Latin Small Letter I
8	<u>beforeinput</u>				
9	<u>compositionupdate</u>			"し"	<i>shi</i> DOM is updated
10	<u>input</u>				
11	<u>keyup</u>	"i"	true		
12	<u>keydown</u>	"Convert"	true		Convert
13	<u>beforeinput</u>				
14	<u>compositionupdate</u>			"詩"	"poem" DOM is updated
15	<u>input</u>				
16	<u>keyup</u>	"Convert"	true		
17	<u>keydown</u>	"Convert"	true		Convert
18	<u>beforeinput</u>				
19	<u>compositionupdate</u>			"市"	"city" DOM is updated
20	<u>input</u>				
21	<u>keyup</u>	"Convert"	true		
22	<u>keydown</u>	"Accept"	true		Accept
23	<u>compositionend</u>			"市"	
24	<u>keyup</u>	"Accept"	false		

IME composition can also be canceled as in the following example, with conditions identical to the previous example. The key Cancel might also be replaced by others depending on the input device in use and the configuration of the IME, e.g., it could be `\u001B` (Escape key).

EXAMPLE 29:

	Event Type	KeyboardEvent	KeyboardEvent	CompositionEvent	Notes
		key	isComposing	data	
1	<u>keydown</u>	"s"	false		Latin Small Letter S
2	<u>compositionstart</u>			" "	
3	<u>compositionupdate</u>			"s"	
4	<u>keyup</u>	"s"	true		Latin Small Letter I
5	<u>keydown</u>	"i"	true		
6	<u>compositionupdate</u>			"し"	
7	<u>keyup</u>	"i"	true		Convert
8	<u>keydown</u>	"Convert"	true		
9	<u>compositionupdate</u>			"詩"	
10	<u>keyup</u>	"Convert"	true		Convert
11	<u>keydown</u>	"Convert"	true		
12	<u>compositionupdate</u>			"市"	
13	<u>keyup</u>	"Convert"	true		Cancel
14	<u>keydown</u>	"Cancel"	true		
15	<u>compositionupdate</u>			" "	
16	<u>compositionend</u>			" "	
17	<u>keyup</u>	"Cancel"	false		

NOTE:

Some input method editors (such as on the MacOS operating system) might set an empty string to the composition data attribute before canceling a composition.

5.3.4.1. Input Method Editor mode keys

Some keys on certain devices are intended to activate input method editor functionality, or to change the mode of an active input method editor. Custom keys for this purpose can be defined for different devices or language modes. The keys defined in this specification for this purpose are: "Alphanumeric", "CodeInput", "FinalMode", "HangulMode", "HanjaMode", "Hiragana", "JunjaMode", "KanaMode", "KanjiMode", "Katakana", and "RomanCharacters". When one of these keys is pressed, and no IME is currently active, the appropriate IME is expected to be activated in the mode indicated by the key (if available). If an IME is already active when the key is pressed, the active IME might change to the indicated mode, or a different IME might be launched, or the might MAY be ignored, on a device- and application-specific basis.

This specification also defines other keys which are intended for operation specifically with input method editors: "Accept", "AllCandidates", "Cancel", "Convert", "Compose", "FullWidth", "HalfWidth", "NextCandidate", "Nonconvert", and "PreviousCandidate". The functions of these keys are not defined in this specification — refer to other resources for details on input method editor functionality.

NOTE:

Keys with [input method editor](#) functions are not restricted to that purpose, and can have other device- or implementation-specific purposes.

5.3.5. Default actions and cancelable keyboard events

Canceling the [default action](#) of a [keydown](#) event MUST NOT affect its respective [keyup](#) event, but it MUST prevent the respective [beforeinput](#) and [input](#) (and [keypress](#) if supported) events from being generated. The following example describes a possible sequence of keys to generate the Unicode character Q (Latin Capital Letter Q) on a US keyboard using a US mapping:

EXAMPLE 30:

Event Type	KeyboardEvent <u>key</u>	InputEvent <u>data</u>	Modifiers	Notes
1 keydown	"Shift"		shiftKey	
2 keydown	"Q"		shiftKey	The default action is prevented, e.g., by invoking preventDefault() . <i>No beforeinput or input (or keypress, if supported) events are generated</i>
3 keyup	"Q"		shiftKey	
4 keyup	"Shift"			

If the key is a modifier key, the keystroke MUST still be taken into account for the modifiers states. The following example describes a possible sequence of keys to generate the Unicode character Q (Latin Capital Letter Q) on a US keyboard using a US mapping:

EXAMPLE 31:

Event Type	KeyboardEvent <u>key</u>	InputEvent <u>data</u>	Modifiers	Notes
1 keydown	"Shift"		shiftKey	The default action is prevented, e.g., by invoking preventDefault() .
2 keydown	"Q"		shiftKey	
3 beforeinput		"Q"		
4 input				
5 keyup	"Q"		shiftKey	
6 keyup	"Shift"			

If the key is part of a sequence of several keystrokes, whether it is a [dead key](#) or it is contributing to an Input Method Editor sequence, the keystroke MUST be ignored (not taken into account) only if the [default action](#) is canceled on the [keydown](#) event. Canceling a [dead key](#) on a [keyup](#) event has no effect on [beforeinput](#) or [input](#) events. The

following example uses the dead key `"Dead"` (`\u0302` Combining Circumflex Accent key) and `"e"` (`\u0065`, Latin Small Letter E key) on a French keyboard using a French mapping and without any modifier activated:

EXAMPLE 32:

	Event Type	KeyboardEvent	InputEvent	Notes
		key	data	
1	<code>keydown</code>	<code>"Dead"</code>		The <u>default action</u> is prevented, e.g., by invoking <code>preventDefault()</code> .
2	<code>keyup</code>	<code>"Dead"</code>		
3	<code>keydown</code>	<code>"e"</code>		
4	<code>beforeinput</code>		<code>"e"</code>	
5	<code>input</code>			
6	<code>keyup</code>	<code>"e"</code>		

5.3.6. Guidelines for selecting key values

This section is normative.

To determine the appropriate key value for a key, use the following algorithm:

- If there exists an appropriate character in the [key values set](#), then the [key](#) attribute MUST be a string consisting of the *key value* of that character.
- If the key generates a printable character, and there exists an appropriate [Unicode code point](#), then the [key](#) attribute MUST be a string consisting of the *char value* of that character.
- If more than one key is being pressed and the key combination includes one or more modifier keys that result in the key no longer producing a printable character (e.g., `Control` + `a`), then the key value should be the printable key value that would have been produced if the key had been typed with the default keyboard layout with no modifier keys except for `Shift` and `AltGr` applied.
- Otherwise, the special value `"Unidentified"` should be used.

EXAMPLE 33:

- On a US keyboard with a right-handed single-hand Dvorak [key mapping](#), the key labeled **Q** maps to the key values **"5"** (unmodified) and **"%"** (shifted). The primary function of this key is to generate the character **"5"** (`\u0035`). Since this character is a character (in Unicode general category `Nd`), the [key](#) attribute value for the unmodified key will be **"5"**.
- On the same US Dvorak keyboard layout as the previous example, pressing **Control** and the key labeled **Q** will produce a key value of **"q"**.
- On a French PC keyboard with a standard French mapping, the primary function of the **^** key is to act as a [dead key](#) for the combining circumflex diacritical mark. The value for this key is **"Dead"**.
- Also on a French PC keyboard with a standard French mapping, the **é** key (which corresponds to the **2** key on a US keyboard) generates a key value of **é** (`\u00e9`).
- On a Korean PC keyboard with a standard Korean mapping, the primary function of the **Ha/En** key is to switch between Hangul and English input. The predefined key value list has an appropriate entry for this key, **"HangulMode"**, so this will be the key value.
- On some mobile devices, there are special keys to launch specific applications. For a standard application like Calendar, there is a predefined key value of **"LaunchCalendar"**.

While every attempt has been made to make this list of key values as complete as possible, new key values will periodically need to be defined as new input devices are introduced. Rather than allowing user agents to define their own key values (which may not work across multiple user agents), bugs should be filed so that this specification can be updated.

6. Legacy Event Initializers

This section is normative. The following features are obsolete and should only be implemented by [user agents](#) that require compatibility with legacy software.

Early versions of this specification included an initialization method on the interface (for example `initMouseEvent`) that required a long list of parameters that, in most cases, did not fully initialize all attributes of the event object. Because of this, event interfaces which were derived from the basic [Event](#) interface required that the initializer of *each* of the derived interfaces be called explicitly in order to fully initialize an event.

EXAMPLE 34:

Initializing all the attributes of a `MutationEvent` requires calls to two initializer methods: `initEvent` and `initMutationEvent`.

Due in part to the length of time in the development of this standard, some implementations MAY have taken a dependency on these (now deprecated) initializer methods. For completeness, these legacy event initializers are described in this Appendix.

6.1. Legacy Event Initializer Interfaces

This section is informative

This section documents legacy initializer methods that were introduced in earlier versions of this specification.

6.1.1. Initializers for interface UIEvent

```
partial interface UIEvent {  
  // Deprecated in this specification  
  void initUIEvent();  
};
```

UIEvent . initUIEvent()

Initializes attributes of an [UIEvent](#) object. This method has the same behavior as [initEvent\(\)](#).

⚠Warning! The `initUIEvent` method is deprecated, but supported for backwards-compatibility with widely-deployed implementations.

DOMString typeArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean bubblesArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean cancelableArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

Window? viewArg

Specifies [view](#). This value MAY be null.

long detailArg

Specifies [detail](#).

6.1.2. Initializers for interface MouseEvent

```
partial interface MouseEvent {  
  // Deprecated in this specification  
  void initMouseEvent();  
};
```

MouseEvent . initMouseEvent()

Initializes attributes of a [MouseEvent](#) object. This method has the same behavior as `UIEvent.initUIEvent()`.

⚠Warning! The `initMouseEvent` method is deprecated, but supported for backwards-compatibility with widely-deployed implementations.

DOMString typeArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean bubblesArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean cancelableArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

Window? viewArg

Specifies [view](#). This value MAY be null.

long detailArg

Specifies [detail](#).

long screenXArg

Specifies [screenX](#).

long screenYArg

Specifies [screenY](#).

long clientXArg

Specifies [clientX](#).

long clientYArg

Specifies [clientY](#).

boolean ctrlKeyArg

Specifies [ctrlKey](#).

boolean altKeyArg

Specifies [altKey](#).

boolean shiftKeyArg

Specifies [shiftKey](#).

boolean metaKeyArg

Specifies [metaKey](#).

short buttonArg

Specifies [button](#).

EventTarget? relatedTargetArg

Specifies [relatedTarget](#). This value MAY be null.

6.1.3. Initializers for interface WheelEvent

```
partial interface WheelEvent {  
    // Originally introduced (and deprecated) in this specification  
    void initWheelEvent();  
};
```

void initWheelEvent()

Initializes attributes of a `WheelEvent` object. This method has the same behavior as `MouseEvent.initMouseEvent()`.

⚠Warning! The `initWheelEvent` method is deprecated.

DOMString typeArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean bubblesArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean cancelableArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

Window? viewArg

Specifies [view](#). This value MAY be null.

long detailArg

Specifies [detail](#).

long screenXArg

Specifies [screenX](#).

long screenYArg

Specifies [screenY](#).

long clientXArg

Specifies [clientX](#).

long clientYArg

Specifies [clientY](#).

short buttonArg

Specifies [button](#).

EventTarget? relatedTargetArg

Specifies [relatedTarget](#). This value MAY be null.

DOMString modifiersListArg

A [white space](#) separated list of modifier key values to be activated on this object. As an example, "Control Shift" marks the control and shift modifiers as activated (the [ctrlKey](#) and [shiftKey](#) inherited attributes will be true on the initialized [WheelEvent](#) object).

double deltaXArg

Specifies [deltaX](#).

double deltaYArg

Specifies [deltaY](#).

double deltaZArg

Specifies [deltaZ](#).

unsigned long deltaMode

Specifies [deltaMode](#).

6.1.4. Initializers for interface **KeyboardEvent**

NOTE:

The argument list to this legacy **KeyboardEvent** initializer does not include the **detailArg** (present in other initializers) and adds the **locale** argument (see [§11.2 Changes between different drafts of UI Events](#)); it is necessary to preserve this inconsistency for compatibility with existing implementations.

```
partial interface KeyboardEvent {  
  // Originally introduced (and deprecated) in this specification  
  void initKeyboardEvent();  
};
```

void initKeyboardEvent()

Initializes attributes of a [KeyboardEvent](#) object. This method has the same behavior as `UIEvent.initUIEvent()`. The value of [detail](#) remains undefined.

⚠Warning! The `initKeyboardEvent` method is deprecated.

DOMString typeArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean bubblesArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean cancelableArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

Window? viewArg

Specifies [view](#). This value MAY be null.

DOMString keyArg

Specifies [key](#).

unsigned long locationArg

Specifies [location](#).

DOMString modifiersListArg

A [white space](#) separated list of modifier key values to be activated on this object. As an example, "Control Alt" marks the `Control` and `Alt` modifiers as activated.

boolean repeat

Specifies whether the key event is repeating. See [repeat](#).

DOMString locale

Specifies the `locale` attribute of the `KeyboardEvent`.

6.1.5. Initializers for interface `CompositionEvent`

NOTE:

The argument list to this legacy `CompositionEvent` initializer does not include the `detailArg` (present in other initializers) and adds the `locale` argument (see [§11.2 Changes between different drafts of UI Events](#)); it is necessary to preserve this inconsistency for compatibility with existing implementations.

```
partial interface CompositionEvent {  
  // Originally introduced (and deprecated) in this specification  
  void initKeyboardEvent();  
};
```

`void initCompositionEvent()`

Initializes attributes of a `CompositionEvent` object. This method has the same behavior as `UIEvent.initUIEvent()`. The value of [detail](#) remains undefined.

⚠Warning! The `initCompositionEvent` method is deprecated.

`DOMString typeArg`

Refer to the [initEvent\(\)](#) method for a description of this parameter.

`boolean bubblesArg`

Refer to the [initEvent\(\)](#) method for a description of this parameter.

`boolean cancelableArg`

Refer to the [initEvent\(\)](#) method for a description of this parameter.

`Window? viewArg`

Specifies [view](#). This value MAY be null.

`DOMString dataArg`

Specifies [data](#).

`DOMString locale`

Specifies the `locale` attribute of the `CompositionEvent`.

7. Legacy Key Attributes

This section is non-normative. The following attributes are obsolete and should only be implemented by [user agents](#) that require compatibility with legacy software that requires these keyboard events.

These features were never formally specified and the current browser implementations vary in significant ways. The large amount of legacy content, including script libraries, that relies upon detecting the [user agent](#) and acting accordingly means that any attempt to formalize these legacy attributes and events would risk breaking as much

content as it would fix or enable. Additionally, these attributes are not suitable for international usage, nor do they address accessibility concerns.

Therefore, this specification does not normatively define the events and attributes commonly employed for handling keyboard input, though they MAY be present in [user agents](#) for compatibility with legacy content. Authors SHOULD use the [key](#) attribute instead of the [charCode](#) and [keyCode](#) attributes.

However, for the purpose of documenting the current state of these features and their relation to normative events and attributes, this section provides an informative description. For implementations which do support these attributes and events, it is suggested that the definitions provided in this section be used.

7.1. Legacy [KeyboardEvent](#) supplemental interface

This section is non-informative

Browser support for keyboards has traditionally relied on three ad-hoc attributes, [keyCode](#), [charCode](#), and [which](#).

All three of these attributes return a numerical code that represents some aspect of the key pressed: [keyCode](#) is an index of the key itself. [charCode](#) is the ASCII value of the character keys. [which](#) is the character value where available and otherwise the key index. The values for these attributes, and the availability of the attribute, is inconsistent across platforms, keyboard languages and layouts, [user agents](#), versions, and even event types.

7.1.1. Interface [KeyboardEvent](#) (supplemental)

Introduced in this specification

The partial [KeyboardEvent](#) interface is an informative extension of the [KeyboardEvent](#) interface, which adds the [charCode](#), [keyCode](#), and [which](#) attributes.

The partial [KeyboardEvent](#) interface can be obtained by using the [createEvent\(\)](#) method call in implementations that support this extension.

```
partial interface KeyboardEvent {  
  // The following support legacy user agents  
  readonly attribute unsigned long charCode;  
  readonly attribute unsigned long keyCode;  
  readonly attribute unsigned long which;  
};
```

KeyboardEvent . charCode

[charCode](#) holds a character value, for [keypress](#) events which generate character input. The value is the Unicode reference number (code point) of that character (e.g. `event.charCode = event.key.charCodeAt(0)` for printable characters). For [keydown](#) or [keyup](#) events, the value of [charCode](#) is 0.

KeyboardEvent . keyCode

[keyCode](#) holds a system- and implementation-dependent numerical code signifying the unmodified identifier associated with the key pressed. Unlike the [key](#) attribute, the set of possible values are not normatively defined in this specification. Typically, these value of the [keyCode](#) SHOULD represent the decimal codepoint in ASCII [\[RFC20\]\[US-ASCII\]](#) or Windows 1252 [\[WIN1252\]](#), but MAY be drawn from a different appropriate character set. Implementations that are unable to identify a key use the key value [\[0\]](#). See [§7.2 Legacy key models](#) for more details on how to determine the values for [keyCode](#).

KeyboardEvent . which

[which](#) holds a system- and implementation-dependent numerical code signifying the unmodified identifier associated with the key pressed. In most cases, the value is identical to [keyCode](#).

7.1.2. Interface KeyboardEventInit (supplemental)

Browsers that include support for [keyCode](#), [charCode](#), and [which](#) in [KeyboardEvent](#) should also add the following members to the [KeyboardEventInit](#) dictionary.

The partial [KeyboardEventInit](#) dictionary is an informative extension of the [KeyboardEventInit](#) dictionary, which adds [charCode](#), [keyCode](#), and [which](#) members to initialize the corresponding [KeyboardEvent](#) attributes.

```
partial dictionary KeyboardEventInit {  
  // The following support legacy user agents  
  unsigned long charCode = 0;  
  unsigned long keyCode = 0;  
  unsigned long which = 0;  
};
```

KeyboardEventInit . charCode

Initializes the [charCode](#) attribute of the [KeyboardEvent](#) to the Unicode code point for the event's character.

KeyboardEventInit . keyCode

Initializes the [keyCode](#) attribute of the [KeyboardEvent](#) to the system- and implementation-dependent numerical code signifying the unmodified identifier associated with the key pressed.

KeyboardEventInit . which

Initializes the [which](#) attribute of the [KeyboardEvent](#) to the implementation-dependent numerical code signifying the unmodified identifier associated with the key pressed. In most cases, the value is identical to [keyCode](#).

7.2. Legacy key models

This section is non-normative

Implementations differ on which values are exposed on these attributes for different event types. An implementation MAY choose to expose both virtual key codes and character codes in the [keyCode](#) property (*conflated model*), or

report separate [keyCode](#) and [charCode](#) properties (*split model*).

7.2.1. How to determine [keyCode](#) for [keydown](#) and [keyup](#) events

The [keyCode](#) for [keydown](#) or [keyup](#) events is calculated as follows:

- Read the virtual key code from the operating system's event information, if such information is available.
- If an Input Method Editor is processing key input and the event is [keydown](#), return 229.
- If input key when pressed without modifiers would insert a numerical character (0-9), return the ASCII code of that numerical character.
- If input key when pressed without modifiers would insert a lower case character in the a-z alphabetical range, return the ASCII code of the upper case equivalent.
- If the implementation supports a key code conversion table for the operating system and platform, look up the value. If the conversion table specifies an alternate virtual key value for the given input, return the specified value.
- If the key's function, as determined in an implementation-specific way, corresponds to one of the keys in the [§7.2.3 Fixed virtual key codes](#) table, return the corresponding key code.
- Return the virtual key code from the operating system.
- If no key code was found, return 0.

7.2.2. How to determine [keyCode](#) for [keypress](#) events

The [keyCode](#) for [keypress](#) events is calculated as follows:

- If the implementation supports a *conflated model*, set [keyCode](#) to the Unicode code point of the character being entered.
- If the implementation supports a *split model*, set [keyCode](#) to 0.

7.2.3. Fixed virtual key codes

The virtual key codes for the following keys do not usually change with keyboard layouts on desktop systems:

Key	Virtual Key Code	Notes
Backspace	8	
Tab	9	
Enter	13	
Shift	16	

Control	17	
Alt	18	
CapsLock	20	
Escape	27	Esc
Space	32	
PageUp	33	
PageDown	34	
End	35	
Home	36	
ArrowLeft	37	
ArrowUp	38	
ArrowRight	39	
ArrowDown	40	
Delete	46	Del

7.2.4. Optionally fixed virtual key codes

The following punctuation characters MAY change virtual codes between keyboard layouts, but reporting these values will likely be more compatible with legacy content expecting US-English keyboard layout:

Key	Character	Virtual Key Code
Semicolon	" ; "	186
Colon	" : "	186
Equals sign	" = "	187
Plus	" + "	187
Comma	" , "	188
Less than sign	" < "	188
Minus	" _ "	189
Underscore	" _ "	189
Period	" . "	190
Greater than sign	" > "	190
Forward slash	" / "	191
Question mark	" ? "	191
Backtick	" ` "	192
Tilde	" ~ "	192
Opening square bracket	" ["	219
Opening curly brace	" { "	219
Backslash	" \ "	220
Pipe	" "	220
Closing square bracket	"] "	221
Closing curly brace	" } "	221
Single quote	" ' "	222
Double quote	" " "	222

8. Legacy Event Types

This section is normative. The following event types are obsolete and should only be implemented by user agents that require compatibility with legacy software.

The purpose of this section is to document the current state of these features and their relation to normative events. For implementations which do support these events, it is suggested that the definitions provided in this section be used.

The following table provides an informative summary of the event types which are deprecated in this specification. They are included here for reference and completeness.

Event Type	Sync / Async	Bubbling Phase	Trusted event target types	DOM Interface	Cancelable	Default Action
<u>DOMActivate</u>	Sync	Yes	Element	<u>UIEvent</u>	Yes	None
<u>DOMAttrModified</u>	Sync	Yes	Element	<u>MutationEvent</u>	No	None
<u>DOMCharacterDataModified</u>	Sync	Yes	Text, Comment, ProcessingInstruction	<u>MutationEvent</u>	No	None
<u>DOMFocusIn</u>	Sync	Yes	<u>Window</u> , Element	<u>FocusEvent</u>	No	None
<u>DOMFocusOut</u>	Sync	Yes	<u>Window</u> , Element	<u>FocusEvent</u>	No	None
<u>DOMNodeInserted</u>	Sync	Yes	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction	<u>MutationEvent</u>	No	None
<u>DOMNodeInsertedIntoDocument</u>	Sync	No	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction	<u>MutationEvent</u>	No	None
<u>DOMNodeRemoved</u>	Sync	Yes	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction	<u>MutationEvent</u>	No	None
<u>DOMNodeRemovedFromDocument</u>	Sync	No	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction	<u>MutationEvent</u>	No	None
<u>DOMSubtreeModified</u>	Sync	Yes	<u>Window</u> , Document, DocumentFragment, Element, Attr	<u>MutationEvent</u>	No	None
<u>keypress</u>	Sync	Yes	Element	<u>KeyboardEvent</u>	Yes	Varies: launch <u>text composition system</u> ; <u>blur</u> and <u>focus</u> events; <u>DOMActivate</u> event; other event

8.1. Legacy [UIEvent](#) events

8.1.1. Legacy [UIEvent](#) event types

8.1.1.1. *DOMActivate*

Type	DOMActivate
Interface	UIEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	Yes
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : element being activated • UIEvent.view : Window • UIEvent.detail : 0

A [user agent](#) MUST dispatch this event when a button, link, or other state-changing element is activated. Refer to [§3.5 Activation triggers and behavior](#) for more details.

⚠Warning! The [DOMActivate event type](#) is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type in favor of the related [event type click](#). Other specifications MAY define and maintain their own [DOMActivate event type](#) for backwards compatibility.

NOTE:

While [DOMActivate](#) and [click](#) are not completely equivalent, implemented behavior for the [click event type](#) has developed to encompass the most critical accessibility aspects for which the [DOMActivate event type](#) was designed, and is more widely implemented. Content authors are encouraged to use the [click event type](#) rather than the related [mousedown](#) or [mouseup event type](#) to ensure maximum accessibility.

Implementations which support the [DOMActivate event type](#) SHOULD also dispatch a [DOMActivate](#) event as a [default action](#) of a [click](#) event which is associated with an [activation trigger](#). However, such implementations SHOULD only initiate the associated [activation behavior](#) once for any given occurrence of an [activation trigger](#).

EXAMPLE 35:

The [DOMActivate event type](#) is REQUIRED to be supported for XForms [\[XFORMS\]](#), which is intended for implementation within a [host language](#). In a scenario where a plugin or script-based implementation of XForms is intended for installation in a native implementation of this specification which does not support the [DOMActivate event type](#), the XForms [user agent](#) has to synthesize and dispatch its own [DOMActivate](#) events based on the appropriate [activation triggers](#).

Thus, when a [click](#) event is dispatched by a [user agent](#) conforming to UI Events, the XForms [user agent](#) has to determine whether to synthesize a [DOMActivate](#) event with the same relevant properties as a [default action](#) of that [click](#) event. Appropriate cues might be whether the [click](#) event is [trusted](#), or whether its [event target](#) has a [DOMActivate](#) event listener registered.

NOTE:

Don't rely upon the interoperable support of [DOMActivate](#) in many [user agents](#). Instead, the [click event type](#) should be used since it will provide more accessible behavior due to broader implementation support.

⚠Warning! The [DOMActivate event type](#) is deprecated in this specification.

8.1.2. Activation event order

If the [DOMActivate](#) event is supported by the [user agent](#), then the events MUST be dispatched in a set order relative to each other: (with only pertinent events listed):

Event Type	Notes
1 click	
2 DOMActivate	default action , if supported by the user agent ; synthesized; <code>isTrusted="true"</code>
3	All other default actions , including the activation behavior

If the focused element is activated by a key event, then the following shows the typical sequence of events (with only pertinent events listed):

Event Type	Notes
1 keydown	MUST be a key which can activate the element, such as the <code>[Enter]</code> or <code>[spacebar]</code> key, or the element is not activated
2 click	default action ; synthesized; <code>isTrusted="true"</code>
3 DOMActivate	default action , if supported by the user agent ; synthesized; <code>isTrusted="true"</code>
4	All other default actions , including the activation behavior

8.2. Legacy [FocusEvent](#) events

8.2.1. Legacy [FocusEvent](#) event types

8.2.1.1. *DOMFocusIn*

Type	DOMFocusIn
Interface	FocusEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Window , Element
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : event target receiving focus• UIEvent.view : Window• UIEvent.detail : 0• FocusEvent.relatedTarget : null

A [user agent](#) MUST dispatch this event when an [event target](#) receives focus. The focus MUST be given to the element before the dispatch of this event type. This event type MUST be dispatched after the event type [focus](#).

⚠Warning! The [DOMFocusIn](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type in favor of the related event types [focus](#) and [focusin](#).

8.2.1.2. *DOMFocusOut*

Type	DOMFocusOut
Interface	FocusEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Window , Element
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : event target losing focus• UIEvent.view : Window• UIEvent.detail : 0• FocusEvent.relatedTarget : null

A [user agent](#) MUST dispatch this event when an [event target](#) loses focus. The focus MUST be taken from the element before the dispatch of this event type. This event type MUST be dispatched after the event type [blur](#).

⚠Warning! The [DOMFocusOut](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type in favor of the related event types [blur](#) and [focusout](#).

8.2.2. Legacy FocusEvent event order

The following is the typical sequence of events when a focus is shifted between elements, including the deprecated [DOMFocusIn](#) and [DOMFocusOut](#) events. The order shown assumes that no element is initially focused.

Event Type	Notes
	<i>User shifts focus</i>
1 focusin	Sent before first target element receives focus
2 focus	Sent after first target element receives focus
3 DOMFocusIn	If supported
	<i>User shifts focus</i>
4 focusout	Sent before first target element loses focus
5 focusin	Sent before second target element receives focus
6 blur	Sent after first target element loses focus
7 DOMFocusOut	If supported
8 focus	Sent after second target element receives focus
9 DOMFocusIn	If supported

8.3. Legacy [KeyboardEvent](#) events

The [keypress](#) event is the traditional method for capturing key events and processing them before the DOM is updated with the effects of the key press. Code that makes use of the [keypress](#) event typically relies on the legacy [charCode](#), [keyCode](#), and [which](#) attributes.

Note that the [keypress](#) event is specific to key events, and has been replaced by the more general event sequence of [beforeinput](#) and [input](#) events. These new [input](#) events are not specific to keyboard actions and can be used to capture user input regardless of the original source.

8.3.1. Legacy [KeyboardEvent](#) event types

8.3.1.1. [keypress](#)

Type	keypress
Interface	KeyboardEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element

Cancelable	Yes
Default action	Varies: launch text composition system ; blur and focus events; DOMActivate event; other event
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : focused element processing the key event or if no element focused, then the body element if available, otherwise the root element • UIEvent.view : Window • UIEvent.detail : 0 • KeyboardEvent.charCode : legacy character value for this event • KeyboardEvent.keyCode : legacy numerical code for this key • KeyboardEvent.which : legacy numerical code for this key • KeyboardEvent.key : the key value of the key pressed. • KeyboardEvent.code : the code value associated with the key's physical placement on the keyboard. • KeyboardEvent.location : the location of the key on the device. • KeyboardEvent.altKey : true if Alt modifier was active, otherwise false • KeyboardEvent.shiftKey : true if Shift modifier was active, otherwise false • KeyboardEvent.ctrlKey : true if Control modifier was active, otherwise false • KeyboardEvent.metaKey : true if Meta modifier was active, otherwise false • KeyboardEvent.repeat : false • KeyboardEvent.isComposing : true if the key event occurs as part of a composition session, otherwise false

If supported by a [user agent](#), this event MUST be dispatched when a key is pressed down, if and only if that key normally produces a [character value](#). The [keypress](#) event type is device dependent and relies on the capabilities of the input devices and how they are mapped in the operating system.

This event type MUST be generated after the [key mapping](#). It MUST NOT be fired when using an [input method editor](#).

If this event is canceled, it should prevent the [input](#) event from firing, in addition to canceling the [default action](#).

Authors SHOULD use the [beforeinput](#) event instead of the [keypress](#) event.

NOTE:

The [keypress](#) event is traditionally associated with detecting a [character value](#) rather than a physical key, and might not be available on all keys in some configurations.

⚠Warning! The [keypress](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type. When in editing contexts, authors can subscribe to the [beforeinput](#) event instead.

8.3.2. [keypress](#) event order

The [keypress](#) event type MUST be dispatched after the [keydown](#) event and before the [keyup](#) event associated with the same key.

The [keypress](#) event type MUST be dispatched after the [beforeinput](#) event and before the [input](#) event associated with the same key.

The sequence of key events for user-agents that support the [keypress](#) event is demonstrated in the following example:

EXAMPLE 36:

Event Type	KeyboardEvent	InputEvent	Notes
	key	data	
1 keydown	"a"		
2 beforeinput		"a"	
3 keypress	"a"		
			<i>Any default actions related to this key, such as inserting a character in to the DOM.</i>
4 input			
5 keyup	"a"		

8.4. Legacy [MutationEvent](#) events

The mutation and mutation name event modules are designed to allow notification of any changes to the structure of a document, including attribute, text, or name modifications.

NOTE:

None of the event types associated with the [MutationEvent](#) interface are designated as cancelable. This stems from the fact that it is very difficult to make use of existing DOM interfaces which cause document modifications if any change to the document might or might not take place due to cancelation of the resulting event. Although this is still a desired capability, it was decided that it would be better left until the addition of transactions into the DOM.

Many single modifications of the tree can cause multiple mutation events to be dispatched. Rather than attempt to specify the ordering of mutation events due to every possible modification of the tree, the ordering of these events is left to the implementation.

The [MutationEvent](#) interface was introduced in DOM Level 2 Events, but has not yet been completely and interoperably implemented across [user agents](#). In addition, there have been critiques that the interface, as designed, introduces a performance and implementation challenge.

DOM4 [\[DOM\]](#) provides a new mechanism using a `MutationObserver` interface which addresses the use cases that mutation events solve, but in a more performant manner. Thus, this specification describes

mutation events for reference and completeness of legacy behavior, but [deprecates](#) the use of the [MutationEvent](#) interface.

8.4.1. Interface MutationEvent

Introduced in DOM Level 2, deprecated in this specification

The `MutationEvent` interface provides specific contextual information associated with Mutation events.

To create an instance of the `MutationEvent` interface, use the [createEvent\(\)](#) method call.

```
interface MutationEvent : Event {
  // attrChangeType
  const unsigned short MODIFICATION = 1;
  const unsigned short ADDITION = 2;
  const unsigned short REMOVAL = 3;

  readonly attribute Node? relatedNode;
  readonly attribute DOMString prevValue;
  readonly attribute DOMString newValue;
  readonly attribute DOMString attrName;
  readonly attribute unsigned short attrChange;

  void initMutationEvent();
};
```

MutationEvent . MODIFICATION

The `Attr` was modified in place.

MutationEvent . ADDITION

The `Attr` was just added.

MutationEvent . REMOVAL

The `Attr` was just removed.

MutationEvent . relatedNode

`relatedNode` MUST be used to identify a secondary node related to a mutation event. For example, if a mutation event is dispatched to a node indicating that its parent has changed, the `relatedNode` will be the changed parent. If an event is instead dispatched to a subtree indicating a node was changed within it, the `relatedNode` MUST be the changed node. In the case of the [DOMAttrModified](#) event, it indicates the `Attr` node which was modified, added, or removed.

The [un-initialized value](#) of this attribute MUST be `null`.

MutationEvent . prevValue

`prevValue` indicates the previous value of the `Attr` node in [DOMAttrModified](#) events, and of the `CharacterData` node in [DOMCharacterDataModified](#) events. The [un-initialized value](#) of this attribute MUST be `" "` (the empty string).

MutationEvent . newValue

newValue indicates the new value of the Attr node in [DOMAttrModified](#) events, and of the CharacterData node in [DOMCharacterDataModified](#) events. The [un-initialized value](#) of this attribute MUST be "" (the empty string).

MutationEvent . attrName

attrName indicates the name of the changed Attr node in a [DOMAttrModified](#) event. The [un-initialized value](#) of this attribute MUST be "" (the empty string).

MutationEvent . attrChange

attrChange indicates the type of change which triggered the [DOMAttrModified](#) event. The values can be MODIFICATION, ADDITION, or REMOVAL.

The [un-initialized value](#) of this attribute MUST be 0.

NOTE:

There is no defined constant for the attrChange value of 0 (the un-initialized value).

MutationEvent . initMutationEvent()

Initializes attributes of a MutationEvent object. This method has the same behavior as [initEvent\(\)](#).

DOMString typeArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean bubblesArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

boolean cancelableArg

Refer to the [initEvent\(\)](#) method for a description of this parameter.

Node? relatedNodeArg

Specifies [MutationEvent.relatedNode](#).

DOMString prevValueArg

Specifies [MutationEvent.prevValue](#). This value MAY be the [empty string](#).

DOMString newValueArg

Specifies [MutationEvent.newValue](#). This value MAY be the [empty string](#).

DOMString attrNameArg

Specifies [MutationEvent.attrName](#). This value MAY be the [empty string](#).

unsigned short attrChangeArg

Specifies [MutationEvent.attrChange](#). This value MAY be 0.

8.4.2. Legacy [MutationEvent](#) event types

The mutation event types are listed below.

8.4.2.1. [DOMAttrModified](#)

Type	DOMAttrModified
Interface	MutationEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : element whose attribute is being modified • MutationEvent.attrName : the name of the changed Attr node • MutationEvent.attrChange : the numerical code corresponding to the most applicable attrChangeType • MutationEvent.relatedNode : the Attr node that has been modified, added, or removed. • MutationEvent.newValue : new value of the attribute, if the Attr node has been added or modified • MutationEvent.prevValue : previous value of the attribute, if the Attr node has been removed or modified

A [user agent](#) MUST dispatch this event after an `Attr.value` has been modified and after an `Attr` node has been added to or removed from an `Element`. The [event target](#) of this event MUST be the `Element` node where the change occurred. It is implementation dependent whether this event type occurs when the children of the `Attr` node are changed in ways that do not affect the value of `Attr.value`.

⚠Warning! The [DOMAttrModified](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type.

8.4.2.2. *DOMCharacterDataModified*

Type	DOMCharacterDataModified
Interface	MutationEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Text, Comment, ProcessingInstruction
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : object whose content is being modified • MutationEvent.attrName : the empty string • MutationEvent.attrChange : 0 • MutationEvent.relatedNode : parent node of the object whose content is being modified • MutationEvent.newValue : new value of the object

- [MutationEvent](#).prevValue : previous value of the object

A [user agent](#) MUST dispatch this event after `CharacterData.data` or `ProcessingInstruction.data` have been modified, but the node itself has not been inserted or deleted. The [event target](#) of this event MUST be the `CharacterData` node or the `ProcessingInstruction` node.

⚠Warning! The [DOMCharacterDataModified](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type.

8.4.2.3. *DOMNodeInserted*

Type	DOMNodeInserted
Interface	MutationEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : element which is being inserted • MutationEvent.attrName : the empty string • MutationEvent.attrChange : 0 • MutationEvent.relatedNode : parent node of the node that has been inserted, or the <code>ownerElement</code> in the case of <code>Attr</code> nodes • MutationEvent.newValue : the empty string • MutationEvent.prevValue : the empty string

A [user agent](#) MUST dispatch this event type when a node other than an `Attr` node has been added as a child of another node. A [user agent](#) MAY dispatch this event when an `Attr` node has been added to an `Element` node (see [note](#) below). This event MUST be dispatched after the insertion has taken place. The [event target](#) of this event MUST be the node being inserted.

NOTE:

For detecting attribute insertion, use the [DOMAttrModified](#) event type instead.

⚠Warning! The [DOMNodeInserted](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type.

8.4.2.4. *DOMNodeInsertedIntoDocument*

Type	DOMNodeInsertedIntoDocument
Interface	MutationEvent
Sync / Async	Sync
Bubbles	No
Trusted Targets	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : element which is being inserted • MutationEvent.attrName : the empty string • MutationEvent.attrChange : 0 • MutationEvent.relatedNode : parent node of the node that has been inserted, or the ownerElement in the case of Attr nodes • MutationEvent.newValue : the empty string • MutationEvent.prevValue : the empty string

A [user agent](#) MUST dispatch this event when a node has been inserted into a document, either through direct insertion of the node or insertion of a subtree in which it is contained. A [user agent](#) MAY treat an Attr node as part of an Element's subtree. This event MUST be dispatched after the insertion has taken place. The [event target](#) of this event MUST be the node being inserted. If the node is being directly inserted, the event type [DOMNodeInserted](#) MUST occur before this event type.

⚠Warning! The [DOMNodeInsertedIntoDocument](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type.

8.4.2.5. *DOMNodeRemoved*

Type	DOMNodeRemoved
Interface	MutationEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none"> • Event.target : element which is being removed • MutationEvent.attrName : the empty string • MutationEvent.attrChange : 0 • MutationEvent.relatedNode : the parent node of the node being removed, or the ownerElement in the case of Attr nodes • MutationEvent.newValue : the empty string • MutationEvent.prevValue : the empty string

A [user agent](#) MUST dispatch this event when a node other than an `Attr` node is being removed from its parent node. A [user agent](#) MAY dispatch this event when an `Attr` node is being removed from its `ownerElement` (see [note](#) below). This event MUST be dispatched before the removal takes place. The [event target](#) of this event MUST be the node being removed.

NOTE:

For reliably detecting attribute removal, use the [DOMAttrModified](#) event type instead.

⚠Warning! The [DOMNodeRemoved](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type.

8.4.2.6. *DOMNodeRemovedFromDocument*

Type	DOMNodeRemovedFromDocument
Interface	MutationEvent
Sync / Async	Sync
Bubbles	No
Trusted Targets	Element, Attr, Text, Comment, DocumentType, ProcessingInstruction
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : element which is being removed• MutationEvent.attrName : the empty string• MutationEvent.attrChange : 0• MutationEvent.relatedNode : the parent node of the node being removed, or the <code>ownerElement</code> in the case of <code>Attr</code> nodes• MutationEvent.newValue : the empty string• MutationEvent.prevValue : the empty string

A [user agent](#) MUST dispatch this event when a node is being removed from a document, either through direct removal of the node or removal of a subtree in which it is contained. A [user agent](#) MAY treat an `Attr` node as part of an `Element`'s subtree. This event MUST be dispatched before the removal takes place. The [event target](#) of this event type MUST be the node being removed. If the node is being directly removed, the event type [DOMNodeRemoved](#) MUST occur before this event type.

NOTE:

For reliably detecting attribute removal, use the [DOMAttrModified](#) event type instead.

⚠Warning! The [DOMNodeRemovedFromDocument](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type.

8.4.2.7. *DOMSubtreeModified*

Type	DOMSubtreeModified
Interface	MutationEvent
Sync / Async	Sync
Bubbles	Yes
Trusted Targets	Window , Document, DocumentFragment, Element, Attr
Cancelable	No
Default action	None
Context (trusted events)	<ul style="list-style-type: none">• Event.target : parent node of subtree being modified• MutationEvent.attrName : the empty string• MutationEvent.attrChange : 0• MutationEvent.relatedNode : null• MutationEvent.newValue : the empty string• MutationEvent.prevValue : the empty string

This is a general event for notification of all changes to the document. It can be used instead of the more specific mutation and mutation name events. It MAY be dispatched after a single modification to the document or, at the implementation's discretion, after multiple changes have occurred. The latter case SHOULD generally be used to accommodate multiple changes which occur either simultaneously or in rapid succession. The target of this event MUST be the lowest common parent of the changes which have taken place. This event MUST be dispatched after any other events caused by the mutation(s) have occurred.

⚠Warning! The [DOMSubtreeModified](#) event type is defined in this specification for reference and completeness, but this specification [deprecates](#) the use of this event type.

9. Extending Events

This section is non-normative

9.1. Introduction

This specification defines several interfaces and many events, however, this is not an exhaustive set of events for all purposes. To allow content authors and implementers to add desired functionality, this specification provides two

mechanisms for extend this set of interfaces and events without creating conflicts: [custom events](#) and [implementation-specific extensions](#).

9.2. Custom Events

A script author MAY wish to define an application in terms of functional components, with event types that are meaningful to the application architecture. The content author can use the [CustomEvent](#) interface to create their own events appropriate to the level of abstraction they are using.

EXAMPLE 37:

A content author might have created an application which features a dynamically generated bar chart. This bar chart is meant to be updated every 5 minutes, or when a feed shows new information, or when the user refreshes it manually by clicking a button. There are several handlers that have to be called when the chart needs to be updated: the application has to fetch the most recent data, show an icon to the user that the event is being updated, and rebuild the chart. To manage this, the content author can choose to create a custom “updateChart” event, which is fired whenever one of the trigger conditions is met:

```
var chartData = ...;
var evt = document.createEvent("CustomEvent");
evt.initCustomEvent( "updateChart", true, false, { data: chartData } );
document.documentElement.dispatchEvent(evt);
```

9.3. Implementation-Specific Extensions

While a new event is being designed and prototyped, or when an event is intended for implementation-specific functionality, it is desirable to distinguish it from standardized events. Implementors SHOULD prefix event types specific to their implementations with a short string to distinguish it from the same event in other implementations and from standardized events. This is similar to the [vendor-specific keyword prefixes](#) in CSS, though without the dashes (" - ") used in CSS, since that can cause problems when used as an attribute name in Javascript.

EXAMPLE 38:

A particular browser vendor, “FooCorp”, might wish to introduce a new event, `jump`. This vendor implements `fooJump` in their browser, using their vendor-specific prefix: “foo”. Early adopters start experimenting with the event, using `someElement.addEventListener("fooJump", doJump, false)`, and provide feedback to FooCorp, who change the behavior of `fooJump` accordingly.

After some time, another vendor, “BarOrg”, decides they also want the functionality, but implement it slightly differently, so they use their own vendor-specific prefix, “bar” in their event type name: `barJump`. Content authors experimenting with this version of the `jump` event type register events with BarOrg’s event type name. Content authors who wish to write code that accounts for both browsers can either register each event type separately with specific handlers, or use the same handler and switch on the name of the event type. Thus, early experiments in different codebases do not conflict, and the early adopter is able to write easily-maintained code for multiple implementations.

Eventually, as the feature matures, the behavior of both browsers stabilizes and might converge due to content author and user feedback or through formal standardization. As this stabilization occurs, and risk of conflicts decrease, content authors can remove the forked code, and use the `jump` event type name (even before it is formally standardized) using the same event handler and the more generic registration method `someElement.addEventListener("jump", doJump, false)`.

9.3.1. Known Implementation-Specific Prefixes

At the time of writing, the following event-type name prefixes are known to exist:

Prefix	Web Engine	Organization
moz, Moz	Gecko	Mozilla
ms, MS	Trident	Microsoft
o, O	Presto	Opera Software
webkit	WebKit	Apple, Google, others

10. Security Considerations

This appendix discusses security considerations for UI Events implementations. The discussion is limited to security issues that arise directly from implementation of the event model, APIs and events defined in this specification. Implementations typically support other features like scripting languages, other APIs and additional events not defined in this document. These features constitute an unknown factor and are out of scope of this document. Implementers SHOULD consult the specifications of such features for their respective security considerations.

Many of the event types defined in this specification are dispatched in response to user actions. This allows malicious event listeners to gain access to information users would typically consider confidential, e.g., typos they might have made when filling out a form, if they reconsider their answer to a multiple choice question shortly before submitting a form, their typing rate or primary input mechanism. In the worst case, malicious event listeners could

capture all user interactions and submit them to a third party through means (not defined in this specification) that are generally available in DOM implementations, such as the XMLHttpRequest interface.

In DOM implementations that support facilities to load external data, events like the [error](#) event can provide access to sensitive information about the environment of the computer system or network. An example would be a malicious HTML document that attempts to embed a resource on the local network or the localhost on different ports. An embedded [DOM application](#) could then listen for [error](#) and [load](#) events to determine which other computers in a network are accessible from the local system or which ports are open on the system to prepare further attacks.

An implementation of UI Events alone is generally insufficient to perform attacks of this kind and the security considerations of the facilities that possibly support such attacks apply. For conformance with this specification, DOM implementations MAY take reasonable steps to ensure that [DOM applications](#) do not get access to confidential or sensitive information. For example, they might choose not to dispatch [load](#) events to nodes that attempt to embed resources on the local network.

11. Changes

11.1. Changes between DOM Level 2 Events and UI Events

Numerous clarifications to the interfaces and event types have been made. The `HTMLEvents` module is no longer defined in this document. The [focus](#) and [blur](#) events have been added to the [UIEvent](#) module, and the [dblclick](#) event has been added to the [MouseEvent](#) module. This new specification provides a better separation between the DOM event flow, the event types, and the DOM interfaces.

11.1.1. Changes to DOM Level 2 event flow

This new specification introduced the following new concepts in the event flow:

- Event listeners are now ordered. In DOM Level 2, the event ordering was unspecified.
- The event flow now includes the [Window](#), to reflect existing implementations.

11.1.2. Changes to DOM Level 2 event types

Many clarifications have been made on the event types. The conformance is now explicitly defined against the event types, and not only in terms of interfaces used by the event types.

"MutationEvents" have been deprecated. Support for namespaced events, present in early drafts of this specification, has also been removed.

For user agents which support the [DOMNodeInserted](#) and [DOMNodeRemoved](#) event types, this specification no longer requires that the event type be fired for `Attr` nodes.

The `resize` event type no longer bubbles and the [mousemove](#) event is now cancelable, reflecting existing implementations.

11.1.3. Changes to DOM Level 2 Events interfaces

Interface [Event](#)

- The [Event](#) interface has one new attribute, [defaultPrevented](#), and one new method, [stopImmediatePropagation\(\)](#).
- [timeStamp](#) is now a `Number` in the ECMAScript binding. A proposed correction to make the same change in [\[DOM-Level-3-Core\]](#) is forthcoming.
- This specification considers the [type](#) attribute to be case-sensitive, while DOM Level 2 Events considers [type](#) to be case-insensitive.

Interface [EventTarget](#)

- The method [dispatchEvent\(\)](#) was modified.

Interface [MouseEvent](#)

- The [MouseEvent](#) interface has one new method [getModifierState\(\)](#).

Exception `EventException`

- The exception `EventException` is removed in this specification. The prior `DISPATCH_REQUEST_ERR` code is re-mapped to a `DOMException` of type `InvalidStateError`.

11.1.4. New Interfaces

The interfaces [CustomEvent](#), [FocusEvent](#), [KeyboardEvent](#), [CompositionEvent](#), and [WheelEvent](#) were added to the Events module.

11.2. Changes between different drafts of UI Events

The DOM Level 3 Events document was originally developed between 2000 and 2003, and published as a W3C Note, pending further feedback and interest from implementers. In 2006, it was picked up for revision and progress on the Recommendation Track, and was then revised to reflect the current state of implementation and the needs of script authors.

Despite its status only as a W3C Note, rather than an official Recommendation, DOM 3 Events saw some implementation, and was also referenced by other specifications, so care is being taken to cause minimal disruption, while still adapting the specification to the current environment.

The current specification has been reordered significantly from the earlier W3C Note form, and also from the structure of DOM2 Events, in order to clarify the material. New diagrams have been put in place to represent hierarchies and events flows more clearly. Here are some of the more important changes between drafts:

- The “key identifier” feature has been renamed “key value” to disambiguate them from unique identifiers for keys.
- The [KeyboardEvent](#) interface was briefly (from 2003-2010) defined to have `keyIdentifier` and `keyLocation` attributes, but these were removed in favor of the current [key](#) and [location](#) attributes. These attributes were not widely implemented.
- The [KeyboardEvent](#) and [CompositionEvent](#) interfaces defined a `locale` attribute. This attribute was underspecified and moved into a technical note until it can be specified adequately. Refer to this [old version of UI Events](#) (before the DOM3Events spec was renamed "UI Events") for details.
- The [KeyboardEvent](#) also had a `char` attribute that was only used by the [keypress](#) event. Since the `keypress` event has been deprecated, this attribute was no longer useful and was removed.
- The `change`, `submit`, and `reset` events were removed, since they were specific to HTML forms, and are specified in [\[HTML5\]](#).
- The `textInput` event, originally proposed as a replacement for [keypress](#), was removed in favor of the current [beforeinput](#) and [input](#) events.
- Namespaced events have been removed.
- Updated to use [\[WebIDL\]](#).
- `EventException` has been removed.

12. Acknowledgements

Many people contributed to the DOM specifications (Level 1, 2 or 3), including participants of the DOM Working Group, the DOM Interest Group, the WebAPI Working Group, and the WebApps Working Group. We especially thank the following:

Andrew Watson (Object Management Group), Andy Heninger (IBM), Angel Diaz (IBM), Anne van Kesteren (Opera Software), Arnaud Le Hors (W3C and IBM), Arun Ranganathan (AOL), Ashok Malhotra (IBM and Microsoft), Ben Chang (Oracle), Bill Shea (Merrill Lynch), Bill Smith (Sun), Björn Höhrmann, Bob Sutor (IBM), Charles McCathie-Neville (Opera Software, *Co-Chair*), Chris Lovett (Microsoft), Chris Wilson (Microsoft), Christophe Jolif (ILOG), David Brownell (Sun), David Ezell (Hewlett-Packard Company), David Singer (IBM), Dean Jackson (W3C, *W3C Team Contact*), Dimitris Dimitriadis (Improve AB and invited expert), Don Park (invited), Doug Schepers (Vectoreal), Elena Litani (IBM), Eric Vasilik (Microsoft), Gavin Nicol (INSO), Gorm Haug Eriksen (Opera Software), Ian Davis (Talis Information Limited), Ian Hickson (Google), Ian Jacobs (W3C), James Clark (invited), James Davidson (Sun), Jared Sorensen (Novell), Jeroen van Rotterdam (X-Hive Corporation), Joe Kesselman (IBM), Joe Lapp (webMethods), Joe Marini (Macromedia), John Robinson (AOL), Johnny Stenback

(Netscape/AOL), Jon Ferraiolo (Adobe), Jonas Sicking (Mozilla Foundation), Jonathan Marsh (Microsoft), Jonathan Robie (Texcel Research and Software AG), Kim Adamson-Sharpe (SoftQuad Software Inc.), Lauren Wood (SoftQuad Software Inc., *former Chair*), Laurence Cable (Sun), Luca Mascaro (HTML Writers Guild), Maciej Stachowiak (Apple Computer), Marc Hadley (Sun Microsystems), Mark Davis (IBM), Mark Scardina (Oracle), Martin Dürst (W3C), Mary Brady (NIST), Michael Shenfield (Research In Motion), Mick Goulish (Software AG), Mike Champion (Arbortext and Software AG), Miles Sabin (Cromwell Media), Patti Lutsky (Arbortext), Paul Grosso (Arbortext), Peter Sharpe (SoftQuad Software Inc.), Phil Karlton (Netscape), Philippe Le Hégarret (W3C, *W3C Team Contact and former Chair*), Ramesh Lekshmyanarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home, and Netscape/AOL, *Chair*), Rezaur Rahman (Intel), Rich Rollman (Microsoft), Rick Gessner (Netscape), Rick Jelliffe (invited), Rob Relyea (Microsoft), Robin Berjon (Expway, *Co-Chair*), Scott Hayman (Research In Motion), Scott Isaacs (Microsoft), Sharon Adler (INSO), Stéphane Sire (IntuiLab), Steve Byrne (JavaSoft), Tim Bray (invited), Tim Yu (Oracle), Tom Pixley (Netscape/AOL), T.V. Raman (Google). Vidur Apparao (Netscape) and Vinod Anupam (Lucent).

Former editors: Tom Pixley (Netscape Communications Corporation) until July 2002; Philippe Le Hégarret (W3C) until November 2003; Björn Höhrmann (Invited Expert) until January 2008; Doug Schepers (W3C) from March 2008 to May 2011; and Jacob Rossi (Microsoft) from March 2011 to October 2011.

Contributors: In the WebApps Working Group, the following people made substantial material contributions in the process of refining and revising this specification: Bob Lund (Cable Laboratories), Cameron McCormack (Invited Expert / Mozilla), Daniel Danilatos (Google), Gary Kacmarcik (Google), Glenn Adams (Samsung), Hallvord R. M. Steen (Opera), Hironori Bono (Google), Mark Vickers (Comcast), Masayuki Nakano (Mozilla), Olli Pettay (Mozilla), Takayoshi Kochi (Google) and Travis Leithead (Microsoft).

Glossary contributors: Arnaud Le Hors (W3C) and Robert S. Sutor (IBM Research).

Test suite contributors: Carmelo Montanez (NIST), Fred Drake, Mary Brady (NIST), Neil Delima (IBM), Rick Rivello (NIST), Robert Clary (Netscape), with a special mention to Curt Arnold.

Thanks to all those who have helped to improve this specification by sending suggestions and corrections (please, keep bugging us with your issues!), or writing informative books or Web sites: Al Gilman, Alex Russell, Alexander J. Vincent, Alexey Proskuryakov, Arkadiusz Michalski, Brad Pettit, Cameron McCormack, Chris Rebert, Curt Arnold, David Flanagan, Dylan Schiemann, Erik Arvidsson, Garrett Smith, Giuseppe Pascale, James Su, Jan Goyvaerts (regular-expressions.info), Jorge Chamorro, Kazuyuki Ashimura, Ken Rehor, Magnus Kristiansen, Martijn Wargers, Martin Dürst, Michael B. Allen, Mike Taylor, Misha Wolf, Ojan Vafai, Oliver Hunt, Paul Irish, Peter-Paul Koch, Richard Ishida, Sean Hogan, Sergey Ilinsky, Sigurd Lerstad, Steven Pemberton, Tony Chang, William Edney and Øistein E. Andersen.

13. Glossary

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

activation behavior

The action taken when an [event](#), typically initiated by users through an input device, causes an element to fulfill a defined task. The task MAY be defined for that element by the [host language](#), or by author-defined variables, or both. The default task for any given element MAY be a generic action, or MAY be unique to that element. For example, the activation behavior of an HTML or SVG `<a>` element is to cause the [user agent](#) to traverse the link specified in the `href` attribute, with the further optional parameter of specifying the browsing context for the traversal (such as the current window or tab, a named window, or a new window). The activation behavior of an HTML `<input>` element with the `type` attribute value `submit` is to send the values of the form elements to an author-defined IRI by the author-defined HTTP method. See [§3.5 Activation triggers and behavior](#) for more details.

activation trigger

An event which is defined to initiate an [activation behavior](#). Refer to [§3.5 Activation triggers and behavior](#) for more details.

author

In the context of this specification, an *author*, *content author*, or *script author* is a person who writes script or other executable content that uses the interfaces, events, and event flow defined in this specification. See [§1.2.3 Content authors and content](#) conformance category for more details.

body element

In HTML or XHTML documents, the body element represents the contents of the document. In a well-formed HTML document, the body element is a first descendant of the [root element](#).

bubbling phase

The process by which an [event](#) can be handled by one of the target's ancestors *after* being handled by the [event target](#). See the description of the [bubble phase](#) in the context of event flow for more details.

capture phase

The process by which an [event](#) can be handled by one of the target's ancestors *before* being handled by the [event target](#). See the description of the [capture phase](#) in the context of event flow for more details.

candidate event handlers

candidate event listeners

The list of all [event listeners](#) that have been registered on the target object in their order of registration. When an event is about to be dispatched to a target object, the list of current listeners is captured before any event listeners are dispatched on the target. Listeners which are newly added after the event dispatch begins will not affect the order of listeners in the captured list, although listeners which are removed after dispatch begins will be removed.

NOTE:

Initially capturing the candidate event handlers and not allowing new listeners to be added prevents infinite loops of event listener dispatch on a given target object.

character value

In the context of key values, a character value is a string representing one or more Unicode characters, such as a letter or symbol, or a set of letters. In this specification, character values are denoted as a unicode string (e.g., `\u0020`) or a glyph representation of the same code point (e.g., " "), and are color coded to help distinguish these two representations.

NOTE:

In source code, some key values, such as non-graphic characters, can be represented using the character escape syntax of the programming language in use.

current event target

In an event flow, the current event target is the object associated with the [event handler](#) that is currently being dispatched. This object MAY be the [event target](#) itself or one of its ancestors. The current event target changes as the [event](#) propagates from object to object through the various [phases](#) of the event flow. The current event target is the value of the [currentTarget](#) attribute.

dead key

A dead key is a key or combination of keys which produces no character by itself, but which in combination or sequence with another key produces a modified character, such as a character with diacritical marks (e.g., "ö", "é", "â").

default action

A [default action](#) is an OPTIONAL supplementary behavior that an implementation MUST perform in combination with the dispatch of the event object. Each event type definition, and each specification, defines the [default action](#) for that event type, if it has one. An instance of an event MAY have more than one [default action](#) under some circumstances, such as when associated with an [activation trigger](#). A [default action](#) MAY be cancelled through the invocation of the [preventDefault\(\)](#) method. For more details, see [§3.2 Default actions and cancelable events](#).

delta

The estimated scroll amount (in pixels, lines, or pages) that the user agent will scroll or zoom the page in response to the physical movement of an input device that supports the [WheelEvent](#) interface (such as a mouse wheel or touch pad). The value of a [delta](#) (e.g., the [deltaX](#), [deltaY](#), or [deltaZ](#) attributes) is to be interpreted in the context of the current [deltaMode](#) property. The relationship between the physical movement of a wheel (or other device) and whether the [delta](#) is positive or negative is environment and device dependent. However, if a user agent scrolls as the [default action](#) then the sign of the [delta](#) is given by a right-hand coordinate system where positive X,Y, and Z axes are directed towards the right-most edge, bottom-most edge, and farthest depth (away from the user) of the document, respectively.

deprecated

Features marked as deprecated are included in the specification as reference to older implementations or specifications, but are OPTIONAL and discouraged. Only features which have existing or in-progress

replacements MUST be deprecated in this specification. Implementations which do not already include support for the feature MAY implement deprecated features for reasons of backwards compatibility with existing content, but content authors creating content SHOULD NOT use deprecated features, unless there is no other way to solve a use case. Other specifications which reference this specification SHOULD NOT use deprecated features, but SHOULD point instead to the replacements of which the feature is deprecated in favor. Features marked as deprecated in this specification are expected to be dropped from future specifications.

dispatch

To create an event with attributes and methods appropriate to its type and context, and propagate it through the DOM tree in the specified manner. Interchangeable with the term “[fire](#)”, e.g., “fire a [click](#) event” or “dispatch a [load](#) event”.

document

An object instantiating the [Document](#) interface [[DOM-Level-3-Core](#)], representing the entire HTML or XML text document. Conceptually, it is the root of the document tree, and provides the primary access to the document’s data.

DOM application

A DOM application is script or code, written by a content author or automatically generated, which takes advantage of the interfaces, methods, attributes, events, and other features described in this specification in order to make dynamic or interactive content, such as Web applications, exposed to users in a [user agent](#).

DOM Level 0

The term “DOM Level 0” refers to a mix of HTML document functionalities, often not formally specified but traditionally supported as de facto standards, implemented originally by Netscape Navigator version 3.0 or Microsoft Internet Explorer version 3.0. In many cases, attributes or methods have been included for reasons of backward compatibility with “DOM Level 0”.

empty string

The empty string is a value of type `DOMString` of length 0, i.e., a string which contains no characters (neither printing nor control characters).

event

An event is the representation of some occurrence (such as a mouse click on the presentation of an element, the removal of child node from an element, or any number of other possibilities) which is associated with its [event target](#). Each event is an instantiation of one specific [event type](#).

event focus

Event focus is a special state of receptivity and concentration on a particular element or other [event target](#) within a document. Each element has different behavior when focused, depending on its functionality, such as priming the element for activation (as for a button or hyperlink) or toggling state (as for a checkbox), receiving text input (as for a text form field), or copying selected text. For more details, see [§4.2.3 Document Focus and Focus Context](#).

event focus ring

An event focus ring is an ordered set of [event focus](#) targets within a document. A [host language](#) MAY define one or more ways to determine the order of targets, such as document order, a numerical index defined per focus target, explicit pointers between focus targets, or a hybrid of different models. Each document MAY contain multiple focus rings, or conditional focus rings. Typically, for document-order or indexed focus rings, focus “wraps around” from the last focus target to the first.

event handler

event listener

An object that implements the [EventListener](#) interface and provides an [handleEvent\(\)](#) callback method. Event handlers are language-specific. Event handlers are invoked in the context of a particular object (the [current event target](#)) and are provided with the event object itself.

NOTE:

In JavaScript, user-defined functions are considered to implement the [EventListener](#) interface. Thus the event object will be provided as the first parameter to the user-defined function when it is invoked. Additionally, JavaScript objects can also implement the [EventListener](#) interface when they define a [handleEvent](#) method.

event order

The sequence in which events from the same event source or process occur, using the same or related event interfaces. For example, in an environment with a mouse, a track pad, and a keyboard, each of those input devices would constitute a separate event source, and each would follow its own event order. A [mousedown](#) event from the trackpad followed by a [mouseup](#) event from the mouse would not result in a [click](#) event.

NOTE:

There can be interactions between different event orders. For example, a [click](#) event might be modified by a concurrent [keydown](#) event (e.g., via [Shift+click](#)). However, the event orders of these different event sources would be distinct.

The event order of some interfaces are device-independent. For example, a user might change focus using the [Tab](#) key, or by clicking the new focused element with the mouse. The event order in such cases depends on the state of the process, not on the state of the device that initiates the state change.

event phase

See [phase](#).

event target

The object to which an [event](#) is targeted using the [§3.1 Event dispatch and DOM event flow](#). The event target is the value of the [target](#) attribute.

event type

An *event type* is an [event](#) object with a particular name and which defines particular trigger conditions, properties, and other characteristics which distinguish it from other event types. For example, the [click](#) event type has different characteristics than the [mouseover](#) or [load](#) event types. The event type is exposed as the [type](#) attribute on the event object. See [§4 Event Types](#) for more details. Also loosely referred to as "*event*", such as the [click event](#).

fire

A synonym for [dispatch](#).

host language

Any language which integrates the features of another language or API specification, while normatively referencing the origin specification rather than redefining those features, and extending those features only in ways defined by the origin specification. An origin specification typically is only intended to be implemented in the context of one or more host languages, not as a standalone language. For example, XHTML, HTML, and SVG are host languages for UI Events, and they integrate and extend the objects and models defined in this specification.

hysteresis

A feature of human interface design to accept input values within a certain range of location or time, in order to improve the user experience. For example, allowing for small deviation in the time it takes for a user to double-click a mouse button is temporal hysteresis, and not immediately closing a nested menu if the user mouses out from the parent window when transitioning to the child menu is locative hysteresis.

IME

input method editor

An *input method editor* (IME), also known as a *front end processor*, is an application that performs the conversion between keystrokes and ideographs or other characters, usually by user-guided dictionary lookup, often used in East Asian languages (e.g., Chinese, Japanese, Korean). An [IME](#) MAY also be used for dictionary-based word completion, such as on mobile devices. See [§5.3.4 Input Method Editors](#) for treatment of IMEs in this specification. See also [text composition system](#).

key mapping

Key mapping is the process of assigning a key value to a particular key, and is the result of a combination of several factors, including the operating system and the keyboard layout (e.g., [QWERTY](#), Dvorak, Spanish, InScript, Chinese, etc.), and after taking into account all [modifier key](#) ([Shift](#), [Alt](#), et al.) and [dead key](#) states.

key value

A key value is a [character value](#) or multi-character string (such as ["Enter"](#), ["Tab"](#), or ["MediaTrackNext"](#)) associated with a key in a particular state. Every key has a key value, whether or not it has a [character value](#). This includes control keys, function keys, [modifier keys](#), [dead keys](#), and any other key. The key value of any given key at any given time depends upon the [key mapping](#).

local name

See local name in [\[XML-Names11\]](#).

modifier key

A modifier key changes the normal behavior of a key, such as to produce a character of a different case (as with the [Shift](#) key), or to alter what functionality the key triggers (as with the [Fn](#) or [Alt](#) keys). Refer to [§5.3.2 Modifier keys](#) for a list of modifier keys.

namespace URI

A *namespace URI* is a URI that identifies an XML namespace. This is called the namespace name in [\[XML-Names11\]](#). See also sections 1.3.2 [DOM URIs](#) and 1.3.3 [XML Namespaces](#) regarding URIs and namespace URIs handling and comparison in the DOM APIs.

phase

In the context of [events](#), a phase is set of logical traversals from node to node along the DOM tree, from the [Window](#) to the [Document](#) object, [root element](#), and down to the [event target \(capture phase\)](#), at the [event target](#) itself ([target phase](#)), and back up the same chain ([bubbling phase](#)).

propagation path

The ordered set of [current event targets](#) through which an [event](#) object will pass sequentially on the way to and back from the [event target](#). As the event propagates, each [current event target](#) in the propagation path is in turn set as the [currentTarget](#). The propagation path is initially composed of one or more [event phases](#) as defined by the [event type](#), but MAY be interrupted. Also known as an *event target chain*.

QWERTY

QWERTY (pronounced “kw3rti”) is a common keyboard layout, so named because the first five character keys on the top row of letter keys are Q, W, E, R, T, and Y. There are many other popular keyboard layouts (including the Dvorak and Colemak layouts), most designed for localization or ergonomics.

root element

The first element node of a document, of which all other elements are children. The document element.

rotation

An indication of incremental change on an input device using the [WheelEvent](#) interface. On some devices this MAY be a literal rotation of a wheel, while on others, it MAY be movement along a flat surface, or pressure on a particular button.

target phase

The process by which an [event](#) can be handled by the [event target](#). See the description of the [target phase](#) in the context of event flow for more details.

text composition system

A software component that interprets some form of alternate input (such as a [input method editor](#), a speech processor, or a handwriting recognition system) and converts it to text.

topmost event target

The [topmost event target](#) MUST be the element highest in the rendering order which is capable of being an [event target](#). In graphical user interfaces this is the element under the user's pointing device. A user interface's "hit testing" facility is used to determine the target. For specific details regarding hit testing and stacking order, refer to the [host language](#).

tree

A data structure that represents a document as a hierarchical set of nodes with child-parent-sibling relationships, i.e., each node having one or more possible ancestors (nodes higher in the hierarchy in a direct lineage), one or more possible descendants (nodes lower in the hierarchy in a direct lineage), and one or more possible peers (nodes of the same level in the hierarchy, with the same immediate ancestor).

Unicode character categories

A subset of the General Category values that are defined for each Unicode code point. This subset contains all the Letter (Ll, Lm, Lo, Lt, Lu), Number (Nd, Nl, No), Punctuation (Pc, Pd, Pe, Pf, Pi, Po, Ps) and Symbol (Sc, Sk, Sm, So) category values.

Unicode code point

A Unicode code point is a unique hexadecimal number signifying a character by its index in the Unicode codespace (or library of characters). In the context of key values, a Unicode code point is expressed as a string in the format \u followed by a hexadecimal character index in the range 0000 to 10FFFF, using at least four digits. See also [character value](#).

un-initialized value

The value of any event attribute (such as [bubbles](#) or [currentTarget](#)) before the event has been initialized with [initEvent\(\)](#). The un-initialized values of an event apply immediately after a new event has been created using the method [createEvent\(\)](#).

user agent

A program, such as a browser or content authoring tool, normally running on a client machine, which acts on a user's behalf in retrieving, interpreting, executing, presenting, or creating content. Users MAY act on the content using different user agents at different times, for different purposes. See the [§1.2.1 Web browsers and other dynamic or interactive user agents](#) and [§1.2.2 Authoring tools](#) for details on the requirements for a *conforming* user agent.

Window

The Window is the object referred to by the current document's browsing context's Window Proxy object as defined in [HTML5 \[HTML5\]](#).

Index

Terms defined by this specification

[abort](#), in §4.1.2.3

[activation behavior](#), in §13

[activation trigger](#), in §13

[altKey](#)

[attribute for MouseEvent](#), in §4.3.1

[dict-member for EventModifierInit](#), in §4.3.2

[attribute for KeyboardEvent](#), in §4.6.1

[author](#), in §13

[beforeinput](#), in §4.5.3.1

[blur](#), in §4.2.4.1

[body element](#), in §13

[bubble phase](#), in §13

[bubbling phase](#), in §13

[button](#)

[attribute for MouseEvent](#), in §4.3.1

[dict-member for MouseEventInit](#), in §4.3.1

[buttons](#)

[attribute for MouseEvent](#), in §4.3.1

[dict-member for MouseEventInit](#), in §4.3.1

[candidate event handlers](#), in §13

[candidate event listeners](#), in §13

[capture phase](#), in §13

[character value](#), in §13

[charCode](#), in §7.1.1

[click](#), in §4.3.4.1

[code](#)

[attribute for KeyboardEvent](#), in §4.6.1

[dict-member for KeyboardEventInit](#), in §4.6.1

[compositionend](#), in §4.7.7.3

[CompositionEvent](#), in §4.7.1

[CompositionEventInit](#), in §4.7.1

[CompositionEvent\(type, eventInitDict\)](#), in §4.7.1

[compositionstart](#), in §4.7.7.1

[compositionupdate](#), in §4.7.7.2

[ctrlKey](#)

[attribute for MouseEvent](#), in §4.3.1

[dict-member for EventModifierInit](#), in §4.3.2

[attribute for KeyboardEvent](#), in §4.6.1

[current event target](#), in §13

[data](#)

[attribute for InputEvent](#), in §4.5.1

[dict-member for InputEventInit](#), in §4.5.1

[attribute for CompositionEvent](#), in §4.7.1

[dict-member for CompositionEventInit](#), in §4.7.1

[dblclick](#), in §4.3.4.2

[dead key](#), in §13

[default action](#), in §13

[delta](#), in §13

[deltaMode](#)

[attribute for WheelEvent](#), in §4.4.1

[dict-member for WheelEventInit](#), in §4.4.1

[deltaX](#)

[attribute for WheelEvent](#), in §4.4.1

[dict-member for WheelEventInit](#), in §4.4.1

[deltaY](#)

[attribute for WheelEvent](#), in §4.4.1

[dict-member for WheelEventInit](#), in §4.4.1

[deltaZ](#)

[attribute for WheelEvent](#), in §4.4.1

[dict-member for WheelEventInit](#), in §4.4.1

[deprecated](#), in §13

[deprecates](#), in §13

[detail](#)

[attribute for UIEvent](#), in §4.1.1

[dict-member for UIEventInit](#), in §4.1.1

[dispatch](#), in §13

[document](#), in §13

[DOMActivate](#), in §8.1.1.1

[DOM application](#), in §13

[DOMAttrModified](#), in §8.4.2.1

[DOMCharacterDataModified](#), in §8.4.2.2

[DOM_DELTA_LINE](#), in §4.4.1

[DOM_DELTA_PAGE](#), in §4.4.1

[DOM_DELTA_PIXEL](#), in §4.4.1

[DOMFocusIn](#), in §8.2.1.1

[DOMFocusOut](#), in §8.2.1.2

[DOM_KEY_LOCATION_LEFT](#), in §4.6.1

[DOM_KEY_LOCATION_NUMPAD](#), in §4.6.1

[DOM_KEY_LOCATION_RIGHT](#), in §4.6.1

[DOM_KEY_LOCATION_STANDARD](#), in §4.6.1

[DOM Level 0](#), in §13

[DOMNodeInserted](#), in §8.4.2.3

[DOMNodeInsertedIntoDocument](#), in §8.4.2.4

[DOMNodeRemoved](#), in §8.4.2.5

[DOMNodeRemovedFromDocument](#), in §8.4.2.6

[DOMSubtreeModified](#), in §8.4.2.7

[empty string](#), in §13

[error](#), in §4.1.2.4

[event](#), in §13

[event focus](#), in §13

[event handler](#), in §13

[event listener](#), in §13

[EventModifierInit](#), in §4.3.2

[event order](#), in §13

[event phase](#), in §13

[event target](#), in §13

[event type](#), in §13

[fire](#), in §13

[focus](#), in §4.2.4.2

[FocusEvent](#), in §4.2.1

[FocusEventInit](#), in §4.2.1

[FocusEvent\(type, eventInitDict\)](#), in §4.2.1

[focusin](#), in §4.2.4.3

[focusout](#), in §4.2.4.4

[focus ring](#), in §13

[getModifierState\(keyArg\)](#)
[method for MouseEvent](#), in §4.3.1
[method for KeyboardEvent](#), in §4.6.1

[host language](#), in §13

[hysteresis](#), in §13

[IME](#), in §13

[input](#), in §4.5.3.2

[InputEvent](#), in §4.5.1

[InputEventInit](#), in §4.5.1

[InputEvent\(type, eventInitDict\)](#), in §4.5.1

[input method editor](#), in §13

[internal key modifier state](#), in §3.6

[isComposing](#)
[attribute for InputEvent](#), in §4.5.1
[dict-member for InputEventInit](#), in §4.5.1
[attribute for KeyboardEvent](#), in §4.6.1
[dict-member for KeyboardEventInit](#), in §4.6.1

[key](#)
[attribute for KeyboardEvent](#), in §4.6.1
[dict-member for KeyboardEventInit](#), in §4.6.1

[KeyboardEvent](#), in §4.6.1

[KeyboardEventInit](#), in §4.6.1

[KeyboardEvent\(type, eventInitDict\)](#), in §4.6.1

[keyCode](#), in §7.1.1

[keydown](#), in §4.6.4.1

[key mapping](#), in §13

[key modifier name](#), in §3.6

[keypress](#), in §8.3.1.1

[keyup](#), in §4.6.4.2

[key value](#), in §13

[load](#), in §4.1.2.1

[local name](#), in §13

location

[attribute for KeyboardEvent](#), in §4.6.1

[dict-member for KeyboardEventInit](#), in §4.6.1

metaKey

[attribute for MouseEvent](#), in §4.3.1

[dict-member for EventModifierInit](#), in §4.3.2

[attribute for KeyboardEvent](#), in §4.6.1

[modifierAltGraph](#), in §4.3.2

[modifierCapsLock](#), in §4.3.2

[modifierFn](#), in §4.3.2

[modifierFnLock](#), in §4.3.2

[modifierHyper](#), in §4.3.2

[modifier key](#), in §13

[modifierNumLock](#), in §4.3.2

[modifierScrollLock](#), in §4.3.2

[modifierSuper](#), in §4.3.2

[modifierSymbol](#), in §4.3.2

[modifierSymbolLock](#), in §4.3.2

[mousedown](#), in §4.3.4.3

[mouseenter](#), in §4.3.4.4

[MouseEvent](#), in §4.3.1

[MouseEvent\(type, eventInitDict\)](#), in §4.3.1

[mouseleave](#), in §4.3.4.5

[mousemove](#), in §4.3.4.6

[mouseout](#), in §4.3.4.7

[mouseover](#), in §4.3.4.8

[mouseup](#), in §4.3.4.9

[namespace URIs](#), in §13

[phase](#), in §13

[propagation path](#), in §13

[QWERTY](#), in §13

relatedTarget

[attribute for FocusEvent](#), in §4.2.1

[dict-member for FocusEventInit](#), in §4.2.1

[attribute for MouseEvent](#), in §4.3.1

[dict-member for MouseEventInit](#), in §4.3.1

repeat

[attribute for KeyboardEvent](#), in §4.6.1

[dict-member for KeyboardEventInit](#), in §4.6.1

[root element](#), in §13

[rotation](#), in §13

[select](#), in §4.1.2.5

shiftKey

[attribute for MouseEvent](#), in §4.3.1

[dict-member for EventModifierInit](#), in §4.3.2

[attribute for KeyboardEvent](#), in §4.6.1

[target phase](#), in §13

[text composition system](#), in §13

[topmost event target](#), in §13

[tree](#), in §13

[UIEvent](#), in §4.1.1

[UIEventInit](#), in §4.1.1

[UIEvent\(type, eventInitDict\)](#), in §4.1.1

[Unicode character categories](#), in §13

[Unicode code point](#), in §13

[un-initialized value](#), in §13

[unload](#), in §4.1.2.2

[user agent](#), in §13

view

[attribute for UIEvent](#), in §4.1.1

[dict-member for UIEventInit](#), in §4.1.1

[wheel](#), in §4.4.2.1

[WheelEvent](#), in §4.4.1

[WheelEventInit](#), in §4.4.1

[WheelEvent\(type, eventInitDict\)](#), in §4.4.1

[which](#), in §7.1.1

Window, in §13

Terms defined by reference

[\[css-syntax-3\]](#) defines the following terms:

[code point](#)

[\[cssom-view\]](#) defines the following terms:

[MouseEventInit](#)

[clientX](#)

[clientY](#)

[screenX](#)

[screenY](#)

[\[dom\]](#) defines the following terms:

[CustomEvent](#)

[Document](#)

[Event](#)

[EventInit](#)

[EventListener](#)

[EventTarget](#)

[MutationEvent](#)

[bubbles](#)

[cancelable](#)

[createEvent\(interface\)](#)

[currentTarget](#)

[defaultPrevented](#)

[dispatchEvent\(event\)](#)

[handleEvent](#)

[handleEvent\(event\)](#)

[initEvent\(type, bubbles, cancelable\)](#)

[isTrusted](#)

[preventDefault\(\)](#)

[stopImmediatePropagation\(\)](#)

[stopPropagation\(\)](#)

[target](#)

[timeStamp](#)

[type](#)

[\[HTML\]](#) defines the following terms:

[Window](#)

[input](#)

[textarea](#)

[\[SVG2\]](#) defines the following terms:

[x](#)

References

Normative References

[DOM-Level-2-Events]

Tom Pixley. [Document Object Model \(DOM\) Level 2 Events Specification](#). 13 November 2000. REC. URL: <http://www.w3.org/TR/DOM-Level-2-Events/>

[DOM-Level-3-Core]

Arnaud Le Hors; et al. [Document Object Model \(DOM\) Level 3 Core Specification](#). 7 April 2004. REC. URL: <http://www.w3.org/TR/DOM-Level-3-Core/>

[HTML]

Ian Hickson. [HTML Standard](#). Living Standard. URL: <https://html.spec.whatwg.org/multipage/>

[SVG2]

Nikos Andronikos; et al. [Scalable Vector Graphics \(SVG\) 2](#). 15 September 2015. WD. URL: <https://svgwg.org/svg2-draft/>

[CSS-SYNTAX-3]

Tab Atkins Jr.; Simon Sapin. [CSS Syntax Module Level 3](#). 20 February 2014. CR. URL: <http://dev.w3.org/csswg/css-syntax/>

[CSSOM-VIEW]

Simon Pieters; Glenn Adams. [CSSOM View Module](#). 17 December 2013. WD. URL: <http://dev.w3.org/csswg/cssom-view/>

[DOM]

Anne van Kesteren; et al. [W3C DOM4](#). 19 November 2015. REC. URL: <http://www.w3.org/TR/dom/>

[HTML5]

Ian Hickson; et al. [HTML5](#). 28 October 2014. REC. URL: <http://www.w3.org/TR/html5/>

[RFC2119]

S. Bradner. [Key words for use in RFCs to Indicate Requirement Levels](#). March 1997. Best Current Practice. URL: <https://tools.ietf.org/html/rfc2119>

[UIEVENTS-CODE]

Gary Kacmarcik; Travis Leithead. [UI Events KeyboardEvent code Values](#). 15 December 2015. WD. URL: <http://www.w3.org/TR/uievents-code/>

[UIEVENTS-KEY]

Gary Kacmarcik; Travis Leithead. [UI Events KeyboardEvent key Values](#). 15 December 2015. WD. URL: <http://www.w3.org/TR/uievents-key/>

Informative References

[CSS2]

Bert Bos; et al. [Cascading Style Sheets Level 2 Revision 1 \(CSS 2.1\) Specification](#). 7 June 2011. REC. URL: <http://www.w3.org/TR/CSS2>

[DWW95]

N. Kano. Developing International Software for Windows 95 and Windows NT: A Handbook for International Software Design. 1995.

[EDITING]

[HTML Editing APIs](#), A. Gregor. W3C Editing APIs CG.

[PCRE]

Perl Compatible Regular Expressions library. URL: <http://www.pcre.org/>

[UAAG20]

James Allan; et al. User Agent Accessibility Guidelines (UAAG) 2.0. 15 December 2015. NOTE. URL: <http://www.w3.org/TR/UAAG20/>

[UAX15]

Mark Davis; Ken Whistler. Unicode Normalization Forms. 31 August 2012. Unicode Standard Annex #15. URL: <http://www.unicode.org/reports/tr15>

[UNICODE]

The Unicode Standard. URL: <http://www.unicode.org/versions/latest/>

[US-ASCII]

Coded Character Set - 7-Bit American Standard Code for Information Interchange. 1986.

[WIN1252]

Windows 1252 a Coded Character Set - 8-Bit. URL: <http://www.microsoft.com/globaldev/reference/sbcs/1252.htm>

[WebIDL]

Cameron McCormack; Boris Zbarsky. WebIDL Level 1. 4 August 2015. WD. URL: <http://www.w3.org/TR/WebIDL-1/>

[HTML40]

Dave Raggett; Arnaud Le Hors; Ian Jacobs. HTML 4.0 Recommendation. 24 April 1998. REC. URL: <http://www.w3.org/TR/html40>

[RFC20]

V.G. Cerf. ASCII format for network interchange. October 1969. Internet Standard. URL: <https://tools.ietf.org/html/rfc20>

[XFORMS]

John Boyer. XForms 1.0 (Third Edition). 29 October 2007. REC. URL: <http://www.w3.org/TR/xforms/>

[XML-NAMES11]

Tim Bray; et al. Namespaces in XML 1.1 (Second Edition). 16 August 2006. REC. URL: <http://www.w3.org/TR/xml-names11/>

IDL Index

```

[Constructor(DOMString type, optional UIEventInit eventInitDict)]
interface UIEvent : Event {
    readonly attribute Window? view;
    readonly attribute long detail;
};

dictionary UIEventInit : EventInit {
    Window? view = null;
    long detail = 0;
};

[Constructor(DOMString type, optional FocusEventInit eventInitDict)]
interface FocusEvent : UIEvent {
    readonly attribute EventTarget? relatedTarget;
};

dictionary FocusEventInit : UIEventInit {
    EventTarget? relatedTarget = null;
};

[Constructor(DOMString type, optional MouseEventInit eventInitDict)]
interface MouseEvent : UIEvent {
    readonly attribute long screenX;
    readonly attribute long screenY;
    readonly attribute long clientX;
    readonly attribute long clientY;

    readonly attribute boolean ctrlKey;
    readonly attribute boolean shiftKey;
    readonly attribute boolean altKey;
    readonly attribute boolean metaKey;

    readonly attribute short button;
    readonly attribute unsigned short buttons;

    readonly attribute EventTarget? relatedTarget;

    boolean getModifierState(DOMString keyArg);
};

dictionary MouseEventInit : EventModifierInit {
    long screenX = 0;
    long screenY = 0;
    long clientX = 0;
    long clientY = 0;

    short button = 0;
    unsigned short buttons = 0;
    EventTarget? relatedTarget = null;
};

```

```

};

dictionary EventModifierInit : UIEventInit {
    boolean ctrlKey = false;
    boolean shiftKey = false;
    boolean altKey = false;
    boolean metaKey = false;

    boolean modifierAltGraph = false;
    boolean modifierCapsLock = false;
    boolean modifierFn = false;
    boolean modifierFnLock = false;
    boolean modifierHyper = false;
    boolean modifierNumLock = false;
    boolean modifierScrollLock = false;
    boolean modifierSuper = false;
    boolean modifierSymbol = false;
    boolean modifierSymbolLock = false;
};

[Constructor(DOMString type, optional WheelEventInit eventInitDict)]
interface WheelEvent : MouseEvent {
    // DeltaModeCode
    const unsigned long DOM_DELTA_PIXEL = 0x00;
    const unsigned long DOM_DELTA_LINE = 0x01;
    const unsigned long DOM_DELTA_PAGE = 0x02;

    readonly attribute double deltaX;
    readonly attribute double deltaY;
    readonly attribute double deltaZ;
    readonly attribute unsigned long deltaMode;
};

dictionary WheelEventInit : MouseEventInit {
    double deltaX = 0.0;
    double deltaY = 0.0;
    double deltaZ = 0.0;
    unsigned long deltaMode = 0;
};

[Constructor(DOMString type, optional InputEventInit eventInitDict)]
interface InputEvent : UIEvent {
    readonly attribute DOMString data;
    readonly attribute boolean isComposing;
};

dictionary InputEventInit : UIEventInit {
    DOMString data = "";
    boolean isComposing = false;
};

```



```

[Constructor(DOMString type, optional KeyboardEventInit eventInitDict)]
interface KeyboardEvent : UIEvent {
    // KeyLocationCode
    const unsigned long DOM_KEY_LOCATION_STANDARD = 0x00;
    const unsigned long DOM_KEY_LOCATION_LEFT = 0x01;
    const unsigned long DOM_KEY_LOCATION_RIGHT = 0x02;
    const unsigned long DOM_KEY_LOCATION_NUMPAD = 0x03;

    readonly attribute DOMString key;
    readonly attribute DOMString code;
    readonly attribute unsigned long location;

    readonly attribute boolean ctrlKey;
    readonly attribute boolean shiftKey;
    readonly attribute boolean altKey;
    readonly attribute boolean metaKey;

    readonly attribute boolean repeat;
    readonly attribute boolean isComposing;

    boolean getModifierState(DOMString keyArg);
};

dictionary KeyboardEventInit : EventModifierInit {
    DOMString key = "";
    DOMString code = "";
    unsigned long location = 0;
    boolean repeat = false;
    boolean isComposing = false;
};

[Constructor(DOMString type, optional CompositionEventInit eventInitDict)]
interface CompositionEvent : UIEvent {
    readonly attribute DOMString data;
};

dictionary CompositionEventInit : UIEventInit {
    DOMString data = "";
};

partial interface KeyboardEvent {
    // The following support legacy user agents
    readonly attribute unsigned long charCode;
    readonly attribute unsigned long keyCode;
    readonly attribute unsigned long which;
};

```

