

Client-Side JavaScript Reference

This part of the book is a reference to client-side JavaScript. It includes entries for important client-side JavaScript object such as `Window`, `Document`, `Element`, `Event`, `XMLHttpRequest`, `Storage`, `Canvas`, and `File`. There is also an entry for the `jQuery` library. The entries are arranged alphabetically by object name and each entry includes a complete list of the constants, properties, methods and event handlers supported by that object.

Previous editions of this book included a separate reference entry for each method, but in this edition, the reference material is made more compact (without omitting details) by including the method descriptions directly in the parent entry.

ApplicationCache	DOMException	HTMLOptionsCollection	Script
ArrayBuffer	DOMImplementation	IFrame	Select
ArrayBufferView	DOMSettableTokenList	Image	Storage
Attr	DOMTokenList	ImageData	StorageEvent
Audio	Element	Input	Style
BeforeUnloadEvent	ErrorEvent	jQuery	Table
Blob	Event	Label	TableCell
BlobBuilder	EventSource	Link	TableRow
Button	EventTarget	Location	TableSection
Canvas	FieldSet	MediaElement	Text
CanvasGradient	File	MediaError	TextArea
CanvasPattern	FileError	MessageChannel	TextMetrics
CanvasRenderingContext2D	FileReader	MessageEvent	TimeRanges
ClientRect	FileReaderSync	MessagePort	TypedArray
CloseEvent	Form	Meter	URL
Comment	FormControl	Navigator	Video
Console	FormData	Node	WebSocket
ConsoleCommandLine	FormValidity	NodeList	Window
CSSRule	Geocoordinates	Option	Worker
CSSStyleDeclaration	Geolocation	Output	WorkerGlobalScope
CSSStyleSheet	GeolocationError	PageTransitionEvent	WorkerLocation
DataTransfer	Geoposition	PopStateEvent	WorkerNavigator
DataView	HashChangeEvent	ProcessingInstruction	XMLHttpRequest
Document	History	Progress	XMLHttpRequestUpload
DocumentFragment	HTMLCollection	ProgressEvent	
DocumentType	HTMLFormControlsCollection	Screen	

Client-Side JavaScript Reference

ApplicationCache

application cache management API

EventTarget

The `ApplicationCache` object is the value of the `applicationCache` property of the `Window` object. It defines an API for managing updates to cached applications. For simple cached applications, there is no need to use this API: it is sufficient to create (and update, as needed) an appropriate cache manifest, as described in §20.4. More complex cached applications that want to more actively manage updates can use the properties, methods, and event handlers described here. See §20.4.2 for more details.

Constants

The following constants are the possible values of the `status` property.

unsigned short `UNCACHED` = 0

This application does not have a manifest attribute: it is not cached.

unsigned short `IDLE` = 1

The manifest has been checked and this application is cached and up-to-date.

unsigned short `CHECKING` = 2

The browser is currently checking the manifest file.

unsigned short `DOWNLOADING` = 3

The browser is downloading and caching files listed in the manifest.

unsigned short `UPDATEREADY` = 4

A new version of the application has been downloaded and cached.

unsigned short `OBSOLETE` = 5

The manifest no longer exists and the cache will be deleted.

Properties

readonly unsigned short `status`

This property describes the cache status of the current document. Its value will be one of the constants listed above.

Methods

`void swapCache()`

When the `status` property is `UPDATEREADY`, the browser is maintaining two cached versions of the application: files are being served from the old version of the cache, and the new version is freshly downloaded and ready for use when the application is next reloaded. You can call `swapCache()` to tell the browser to immediately discard the old cache and begin serving files from the new cache. Note, however, that this can lead to version skew issues, and a safer way to flush the old cache and begin using the new one is to reload the application with `Location.reload()`.

`void update()`

Normally, the browser checks for a new version of the manifest file for a cached application each time the application is loaded. Long-lived web apps can use this method to check for updates more frequently.

Event Handlers

The browser fires a sequence of events on the `ApplicationCache` during the manifest check and cache update process. You can use the following event handler properties of the `ApplicationCache` object to register event handlers, or you can use the `EventTarget` methods implemented by the `ApplicationCache` object. Handlers for most of these events are passed a simple `Event` object. Handlers for progress events, however, are passed a `ProgressEvent` object, which can be used to track how many bytes have been downloaded.

`oncached`

Triggered when an application has cached for the first time. This will be the last event in the sequence of events.

`onchecking`

Triggered when the browser begins checking the manifest file for updates. This is the first event in any sequence of application cache events.

`ondownloading`

Triggered when the browser begins downloading the resources listed in a manifest file, either the first time the application is cached or when there is an update. This event will generally be followed by one or more progress events.

`onerror`

Triggered when an error occurs during the cache update process. This can occur if the browser is offline, for example, or if an uncached application references a nonexistent manifest file.

`onnoupdate`

Triggered when the browser determines that the manifest has not changed and the cached application is current. This is the last event in the sequence.

`onobsolete`

Triggered when the manifest file for a cached application no longer exists. This causes the cache to be deleted. This is the last event in the sequence.

onprogress

Triggered periodically while the application files are being downloaded and cached. The event object associated with this event is a `ProgressEvent`.

onupdate

Triggered when a new version of the application has been downloaded and cached (and is ready for use the next time the application is loaded). This is the last event in the sequence.

ArrayBuffer

a fixed-length sequence of bytes

An `ArrayBuffer` represents a fixed-length sequence of bytes in memory, but it defines no way to get or set those bytes. `ArrayBufferViews` like the `TypedArray` classes provide a way to access and interpret the bytes.

Constructor

`new ArrayBuffer(unsigned long length)`

Creates a new `ArrayBuffer` with the specified number of bytes. All bytes in the new `ArrayBuffer` are initialized to 0.

Properties

readonly unsigned long **byteLength**
the length, in bytes, of the `ArrayBuffer`.

ArrayBufferView

common properties for types based on `ArrayBuffers`

`ArrayBufferView` serves as a superclass for types that provide access to the bytes of an `ArrayBuffer`. You can't create an `ArrayBufferView` directly: it exists to define the common properties for subtypes like `TypedArray` and `DataView`.

Properties

readonly `ArrayBuffer` **buffer**

The underlying `ArrayBuffer` that this object is a view of.

readonly unsigned long **byteLength**

The length, in bytes, of the portion of `buffer` that is accessible through this view.

readonly unsigned long **byteOffset**

The starting position, in bytes, of the portion of the buffer that is accessible through this view.

Attr

an element attribute

An `Attr` object represents an attribute of an `Element` node. You can obtain an `Attr` object through the `attributes` property of the `Node` interface or by calling the `getAttributeNode()` or `getAttributeNodeNS()` methods of the `Element` interface.

Since attribute values can be completely represented by strings, it is not usually necessary to use the `Attr` interface at all. In most cases, the easiest way to work with attributes is with the `Element.getAttribute()` and `Element.setAttribute()` methods. These methods use strings for attribute values and avoid the use of `Attr` objects altogether.

Properties

readonly string `localName`

The name of the attribute, not including any namespace prefix.

readonly string `name`

The name of the attribute, including the namespace prefix, if there is one.

readonly string `namespaceURI`

The URI that identifies the attribute's namespace, or `null` if it doesn't have one.

readonly string `prefix`

The namespace prefix of the attribute, or `null` if it doesn't have one.

string `value`

The value of the attribute.

Audio

an HTML `<audio>` element

`Node`, `Element`, `MediaElement`

An `Audio` object represents an HTML `<audio>` element. Except for its constructor, an `Audio` object has no properties, methods or event handlers other than those inherited from `MediaElement`.

Constructor

`new Audio([string src])`

This constructor creates a new `<audio>` element with a `preload` attribute set to "auto". If the `src` argument is specified, it is used as the value of the `src` attribute.

BeforeUnloadEvent

Event object for unload events

`Event`

The unload event is triggered on a `Window` object just before the browser navigates to a new document, and gives a web application the opportunity to ask the user if he is really sure he wants to leave the page. The object passed to unload event handler functions is a

BeforeUnloadEvent object. If you want to require the user to confirm that he really wants to leave the page, you do not need to, and should not, call the `Window.confirm()` method. Instead, return a string from the event handler or set the `returnValue` of this event object to a string. The string you return or set will be presented to the user in the confirmation dialog the user sees.

See also [Event](#) and [Window](#).

Properties

string `returnValue`

A message to be displayed to the user in a confirmation dialog before navigating away from the page. Leave this property unset if you do not want to display a confirmation dialog.

Blob

an opaque chunk of data, such as file contents

A Blob is an opaque type used to exchange data between APIs. Blobs may be very large and may represent binary data, but neither is required. Blobs are often stored in files, but this is an implementation detail. Blobs expose only their size and, optionally, a MIME type, and they define a single method for treating a region of a Blob as a Blob.

A number of APIs use Blobs: see [FileReader](#) for a way to read the content of a Blob and [BlobBuilder](#) for a way to create new Blob objects. See [XMLHttpRequest](#) for ways to download and upload Blobs. See §22.6 for discussion of Blobs and the APIs that use them.

Properties

readonly unsigned long `size`

The length, in bytes, of the Blob.

readonly string `type`

The MIME type of the Blob, if it has one, or the empty string otherwise.

Methods

Blob `slice`(unsigned long *start*, unsigned long *length*, [string *contentType*])

Return a new Blob that represents the *length* bytes of this Blob starting at offset *start*. If *contentType* is specified, it will be used as the `type` property of the returned Blob

BlobBuilder

create new Blobs

A BlobBuilder object is used to create new Blob objects out of strings of text and bytes from [ArrayBuffer](#) objects and other Blobs. To build a Blob, create a BlobBuilder, call `append()` one or more times, and then call `getBlob()`.

Constructor

```
new BlobBuilder()
```

Create a new `BlobBuilder` by calling the `BlobBuilder()` constructor with no arguments.

Methods

```
void append(string text, [string endings])
```

Appends the specified *text*, encoded using UTF-8, to the Blob that is being built.

```
void append(Blob data)
```

Append the content of the Blob *data* to the Blob that is being built.

```
void append(ArrayBuffer data)
```

Append the bytes of the `ArrayBuffer` *data* to the Blob that is being built.

```
Blob getBlob([string contentType])
```

Return a Blob that represents all the data that has been appended to this `BlobBuilder` since it was created. Each call to this method returns a new Blob. If *contentType* is specified, it will be the value of the `type` property of the returned Blob. If unspecified, the returned Blob will have the empty string as its `type`.

Button

an HTML `<button>`

Node, Element, `FormControl`

A `Button` object represents an HTML `<button>` element. Most of the properties and methods of Buttons are described in `FormControl` and `Element`. When a `Button` has a `type` property (see `FormControl`) “submit”, however, the other properties listed here specify form submission parameters that override similar properties on the `Button`’s `form` (see `FormControl`).

Properties

The following properties are meaningful only when the `<button>` has a `type` of “submit”.

string `formAction`

This property mirrors the `formaction` HTML attribute. For submit buttons, it overrides the `action` property of the form.

string `formEnctype`

This property mirrors the `formenctype` HTML attribute. For submit buttons, it overrides the `enctype` property of the form and has the same legal values as that property.

string `formMethod`

This property mirrors the `formmethod` HTML attribute. For submit buttons, it overrides the `method` property of the form.

string `formNoValidate`

This property mirrors the `formnovalidate` HTML attribute. For submit buttons, it overrides the `noValidate` property of the form.

string formTarget

This property mirrors the `formtarget` HTML attribute. For submit buttons, it overrides the `target` property of the form.

Canvas

an HTML element for scripted drawing

Node, Element

The Canvas object represents an HTML canvas element. It has no behavior of its own, but it defines an API that supports scripted client-side drawing operations. You can specify the `width` and `height` directly on this object, and you can extract an image from the canvas with `toDataURL()`, but the actual drawing API is implemented by a separate “context” object returned by the `getContext()` method. See `CanvasRenderingContext2D`.

Properties

unsigned long **height**

unsigned long **width**

These properties mirror the `width` and `height` attributes of the `<canvas>` tag and specify the dimensions of the canvas coordinate space. The defaults are 300 for `width` and 150 for `height`.

If the size of the canvas element is not otherwise specified in a stylesheet or with the `inline style` attribute, these `width` and `height` properties also specify the on-screen dimensions of the canvas element.

Setting either of these properties (even setting it to its current value) clears the canvas to transparent black and resets all of its graphics attributes to their default values.

Methods

object **getContext**(string *contextId*, [any *args...*])

This method returns an object with which you can draw into the Canvas element. When you pass the string “2d”, it will return a `CanvasRenderingContext2D` object for 2D drawing. No additional *args* are required in this case.

There is only one `CanvasRenderingContext2D` object per canvas element, so repeated calls to `getContext(“2d”)` return the same object.

HTML5 standardizes the “2d” argument to this method and defines no other valid arguments. A separate standard, WebGL, is under development for 3D graphics. In browsers that support it, you can pass the string “webgl” to this method to obtain an object that allows 3D rendering. Note, however, that the `CanvasRenderingContext2D` object is the only drawing context documented in this book.

string **toDataURL**([string *type*], [any *args...*])

`toDataURL()` returns the contents of the canvas bitmap as a `data://` URL that can easily be used with an `<img>` tag or transmitted across the network. For example:

```
// Copy the content of a canvas into an <img> and append to the document
var canvas = document.getElementById("my_canvas");
```

```
var image = document.createElement("img");
image.src = canvas.toDataURL();
document.body.appendChild(image);
```

The *type* argument specifies the MIME type of the image format to use. If this argument is omitted, the default value is "image/png". The PNG image format is the only one that implementations are required to support. For image types other than PNG, additional arguments may be passed to specify encoding details. If *type* is "image/jpeg", for example, the second argument should be a number between 0 and 1 specifying the image quality level. No other parameter arguments are standardized at the time of this writing.

To prevent cross-origin information leaks, `toDataURL()` does not work on `<canvas>` tags that are not "origin-clean." A canvas is not origin-clean if it has ever had an image drawn in it (directly by `drawImage()` or indirectly through a `CanvasPattern`) that has a different origin than the document that contains the canvas. Also, a canvas is not origin-clean if it has ever had text drawn to it using a web font from a different origin.

CanvasGradient

a color gradient for use in a canvas

A `CanvasGradient` object represents a color gradient that can be assigned to both the `strokeStyle` and `fillStyle` properties of a `CanvasRenderingContext2D` object. The `createLinearGradient()` and `createRadialGradient()` methods of `CanvasRenderingContext2D` both return `CanvasGradient` objects.

Once you have created a `CanvasGradient` object, use `addColorStop()` to specify what colors should appear at what positions within the gradient. Between the positions you specify, colors are interpolated to create a smooth gradient or fade. If you specify no color stops, the gradient will be uniform transparent black.

Methods

void addColorStop(double offset, string color)

`addColorStop()` specifies fixed colors within a gradient. *color* is a CSS color string. *offset* is a floating-point value in the range 0.0 to 1.0 that represents a fraction between the start and end points of the gradient. An offset of 0 corresponds to the start point, and an offset of 1 corresponds to the end point.

If you specify two or more color stops, the gradient will smoothly interpolate colors between the stops. Before the first stop, the gradient will display the color of the first stop. After the last stop, the gradient will display the color of the last stop. If you specify only a single stop, the gradient will be one solid color. If you specify no color stops, the gradient will be uniform transparent black.

CanvasPattern

an image-based pattern for use in a Canvas

A `CanvasPattern` object is an opaque object returned by the `createPattern()` method of a `CanvasRenderingContext2D` object. A `CanvasPattern` object can be used as the value of the `strokeStyle` and `fillStyle` properties of a `CanvasRenderingContext2D` object.

CanvasRenderingContext2D

the object used for drawing on a canvas

The `CanvasRenderingContext2D` object provides properties and methods for drawing two-dimensional graphics. The following sections provide an overview. See §21.4, `Canvas`, `CanvasGradient`, `CanvasPattern`, `ImageData`, and `TextMetrics` for further details.

Creating and rendering paths

A powerful feature of the canvas is its ability to build shapes up from basic drawing operations, then either draw their outlines (*stroke* them) or paint their contents (*fill* them). The operations accumulated are collectively referred to as the *current path*. A canvas maintains a single current path.

In order to build a connected shape out of multiple segments, a joining point is needed between drawing operations. For this purpose, the canvas maintains a *current position*. The canvas drawing operations implicitly use this as their start point and update it to what is typically their end point. You can think of this like drawing with a pen on paper: when finishing a particular line or curve, the current position is where the pen rested after completing the operation.

You can create a sequence of disconnected shapes in the current path that will be rendered together with the same drawing parameters. To separate shapes, use the `moveTo()` method; this moves the current position to a new location without adding a connecting line. When you do this, you create a new *subpath*, which is the canvas term used for a collection of operations that are connected.

The available path operations are `lineTo()` for drawing straight lines, `rect()` for drawing rectangles, `arc()` and `arcTo()` for drawing partial circles, and `bezierCurveTo()` and `quadraticCurveTo()` for drawing curves.

Once the path is complete, you can draw its outline with `stroke()`, paint its contents with `fill()`, or do both.

In addition to stroking and filling, you can also use the current path to specify the *clipping region* the canvas uses when rendering. Pixels inside this region are displayed; those outside are not. The clipping region is cumulative; calling `clip()` intersects the current path with the current clipping region to yield a new region.

If the segments in any of the subpaths do not form a closed shape, `fill()` and `clip()` operations implicitly close them for you by adding a virtual (not visible with a stroke) line segment

from the start to the end of the subpath. Optionally, you can call `closePath()` to explicitly add this line segment.

To test whether a point is inside (or on the boundary of) the current path, use `isPointInPath()`. When a path intersects itself or consists of multiple overlapping subpaths, the definition of “inside” is based on the nonzero winding rule. If you draw a circle inside another circle and both circles are drawn in the same direction, everything inside the larger circle is considered inside the path. If, on the other hand, one circle is drawn clockwise and the other counterclockwise, you have defined a donut shape and the interior of the smaller circle is outside of the path. This same definition of insideness is used by the `fill()` and `clip()` methods.

Colors, gradients, and patterns

When filling or stroking paths, you can specify how the lines or filled area are rendered using the `fillStyle` and `strokeStyle` properties. Both accept CSS-style color strings, as well as `CanvasGradient` and `CanvasPattern` objects that describe gradients and patterns. To create a gradient, use the `createLinearGradient()` or `createRadialGradient()` methods. To create a pattern, use `createPattern()`.

To specify an opaque color using CSS notation, use a string of the form “#RRGGBB”, where RR, GG, and BB are hexadecimal digits that specify the red, green, and blue components of the color as values between 00 and FF. For example, bright red is “#FF0000”. To specify a partially transparent color, use a string of the form “rgba(R,G,B,A)”. In this form, R, G, and B specify the red, green, and blue components of the color as decimal integers between 0 and 255, and A specifies the alpha (opacity) component as a floating-point value between 0.0 (fully transparent) and 1.0 (fully opaque). For example, half-transparent bright red is “rgba(255,0,0,0.5)”.

Line width, line caps, and line joins

Canvas defines several properties that specify how lines are stroked. You can specify the width of the line with the `lineWidth` property, how the end points of lines are drawn with the `lineCap` property, and how lines are joined using the `lineJoin` property.

Drawing rectangles

You can outline and fill rectangles with `strokeRect()` and `fillRect()`. In addition, you can clear the area defined by a rectangle with `clearRect()`.

Drawing images

In the Canvas API, images are specified using `Image` objects that represent HTML `<img>` elements or offscreen images created with the `Image()` constructor. (See the `Image` reference page for details.) A `<canvas>` element or `<video>` element can also be used as an image source.

You can draw an image into a canvas with the `drawImage()` method, which, in its most general form, allows an arbitrary rectangular region of the source image to be scaled and rendered into the canvas.

Drawing Text

The `fillText()` method draws text and the `strokeText()` method draws outlined text. The `font` property specifies the font to use; the value of this property should be a CSS font specification string. The `textAlign` property specifies whether text is left-justified, centered, or right-justified on the X coordinate you pass, and the `textBaseline` property specifies where the text is drawn in relation to the Y coordinate you pass.

Coordinate space and transformations

By default, the coordinate space for a canvas has its origin at (0,0) in the upper left corner of the canvas, with *x* values increasing to the right and *y* values increasing down. The `width` and `height` attributes of the `<canvas>` tag specify the maximum X and Y coordinates, and a single unit in this coordinate space normally translates to a single on-screen pixel.

You can define your own coordinate space and the coordinates you pass to the canvas drawing methods will automatically be transformed. This is done with the `translate()`, `scale()`, and `rotate()` methods, which affect the *transformation matrix* of the canvas. Because the coordinate space can be transformed like this, the coordinates you pass to methods such as `lineTo()` cannot be measured in pixels and the Canvas API uses floating-point numbers instead of integers.

Shadows

`CanvasRenderingContext2D` can automatically add a drop shadow to anything you draw. The color of the shadow is specified with `shadowColor`, and its offset is changed using `shadowOffsetX` and `shadowOffsetY`. In addition, the amount of feathering applied to the shadow's edge can be set with `shadowBlur`.

Compositing

Usually, when you draw on a canvas, the newly drawn graphics appear on top of the previous content of the canvas, partially or fully obscuring the old content, depending on the opacity of the new graphics. The process of combining new pixels with old pixels is called "compositing" and you can alter the way the canvas composites pixels by specifying different values for the `globalCompositeOperation` property. For example, you can set this property so that newly drawn graphics appear underneath the existing content.

The following table lists the allowed property values and their meanings. The word *source* in the table refers to the pixels being drawn onto the canvas, and the word *destination* refers to the existing pixels on the canvas. The word *result* refers to the pixels that result from the combination of the source and destination. In the formulas, the letter *S* is the source pixel, *D* is the destination pixel, *R* is the result pixel, α_s is the alpha channel (the opacity) of the source pixel, and α_d is the alpha channel of the destination:

Value	Formula	Meaning
"copy"	$R = S$	Draws the source pixel, ignoring the destination pixel.
"destination-atop"	$R = (1 - \alpha_d)S + \alpha_s D$	Draw the source pixel underneath the destination. If the source is transparent, the result is also transparent.

Value	Formula	Meaning
"destination-in"	$R = \alpha_S D$	Multiply the destination pixel by the opacity of the source pixel, but ignore the color of the source.
"destination-out"	$R = (1 - \alpha_S) D$	The destination pixel is made transparent when the source is opaque and is left unchanged when the source is transparent. The color of the source pixel is ignored.
"destination-over"	$R = (1 - \alpha_d) S + D$	The source pixel appears behind the destination pixel, showing through based on the transparency of the destination.
"lighter"	$R = S + D$	The color components of the two pixels are simply added together and clipped if the sum exceeds the maximum value.
"source-atop"	$R = \alpha_d S + (1 - \alpha_S) D$	Draw the source pixel on top of the destination but multiply it by the opacity of the destination. Don't draw anything over a transparent destination.
"source-in"	$R = \alpha_d S$	Draw the source pixel, but multiply it by the opacity of the destination. The color of the destination is ignored. If the destination is transparent, the result is transparent, too.
"source-out"	$R = (1 - \alpha_d) S$	The result is the source pixel where the destination is transparent and transparent pixels where the destination is opaque. The color of the destination is ignored.
"source-over"	$R = S + (1 - \alpha_S) D$	The source pixel is drawn on top of the destination pixel. If the source is translucent, the destination pixel contributes to the result. This is the default value of the global Composite Operation property.
"xor"	$R = (1 - \alpha_d) S + (1 - \alpha_S) D$	If the source is transparent, the result is the destination. If the destination is transparent, the result is the source. If source and destination are both transparent or both opaque, the result is transparent.

Saving graphics state

The `save()` and `restore()` methods allow you to save and restore the state of a `CanvasRenderingContext2D` object. `save()` pushes the current state onto a stack, and `restore()` pops the most recently saved state off the top of the stack and sets the current drawing state based on those stored values.

All properties of the `CanvasRenderingContext2D` object (except for the `canvas` property, which is a constant) are part of the saved state. The transformation matrix and clipping region are also part of the state, but the current path and current point are not.

Manipulating Pixels

The `getImageData()` method allows you to query the raw pixels of a canvas, and `putImageData()` allows you to set individual pixels. These can be useful if you want to implement image processing operations in JavaScript.

Properties

readonly `Canvas canvas`

The Canvas element upon which this context will draw.

any `fillStyle`

The current color, pattern, or gradient used for filling paths. This property can be set to a CSS color string or to a `CanvasGradient` or `CanvasPattern` object. The default fill style is solid black.

string `font`

The font to be used by text-drawing methods, specified as a string, using the same syntax as the CSS `font` attribute. The default is "10px sans-serif". If the font string uses font size units like "em" or "ex" or uses relative keywords like "larger", "smaller", "bolder", or "lighter", these are interpreted relative to the computed style of the CSS font of the `<canvas>` element.

double `globalAlpha`

Specifies additional transparency to be added to everything drawn on the canvas. The alpha value of all pixels drawn on the canvas is multiplied by the value of this property. The value must be a number between 0.0 (makes everything completely transparent) and 1.0 (the default: adds no additional transparency).

string `globalCompositeOperation`

This property specifies how source pixels being rendered onto the canvas are combined (or "composited") with the destination pixels that already exist in the canvas. This property is typically only useful when you are working with partially transparent colors or have set the `globalAlpha` property. The default value is "source-over". Other commonly used values are "destination-over" and "copy". See the table of legal values above. Note that at the time of this writing, browsers have differing implementations of certain compositing modes: some composite locally and some composite globally. See §21.4.13 for details.

string `lineCap`

The `lineCap` property specifies how lines should be terminated. It matters only when drawing wide lines. Legal values for this property are listed in the following table. The default value is "butt".

Value	Meaning
"butt"	This default value specifies that the line should have no cap. The end of the line is straight and is perpendicular to the direction of the line. The line is not extended beyond its endpoint.
"round"	This value specifies that lines should be capped with a semicircle whose diameter is equal to the width of the line and which extends beyond the end of the line by one half the width of the line.
"square"	This value specifies that lines should be capped with a rectangle. This value is like "butt", but the line is extended by half of its width.

string lineJoin

When a path includes vertices where line segments and/or curves meet, the `lineJoin` property specifies how those vertices are drawn. The effect of this property is apparent only when drawing with wide lines.

The default value of the property is “miter”, which specifies that the outside edges of the two line segments are extended until they intersect. When two lines meet at an acute angle, mitered joins can become quite long. The `miterLimit` property places an upper bound on the length of a miter. If a miter would exceed this limit, it is converted to a bevel.

The value “round” specifies that the outside edges of the vertex should be joined with a filled arc whose diameter is equal to the width of the line. The value “bevel” specifies that the outside edges of the vertex should be joined with a filled triangle.

double lineWidth

Specifies the line width for stroking (line drawing) operations. The default is 1. Lines are centered over the path, with half of the line width on each side.

double miterLimit

When lines are drawn with the `lineJoin` property set to “miter” and two lines meet at an acute angle, the resulting miter can be quite long. When miters are too long, they become visually jarring. This `miterLimit` property places an upper bound on the length of the miter. This property expresses a ratio of the miter length to half the line width. The default value is 10, which means that a miter should never be longer than 5 times the line width. If a miter formed by two lines would be longer than the maximum allowed by `miterLimit`, those two lines will be joined with a bevel instead of a miter.

double shadowBlur

Specifies how much blur shadows should have. The default is 0, which produces crisp-edged shadows. Larger values produce larger blurs, but note that the units are not measured in pixels and are not affected by the current transform.

string shadowColor

Specifies the color of shadows as a CSS color string. The default is transparent black.

double shadowOffsetX**double shadowOffsetY**

Specify the horizontal and vertical offset of the shadows. Larger values make the shadowed object appear to float higher above the background. The default is 0. These values are in coordinate space units and they are independent of the current transform.

any strokeStyle

Specifies the color, pattern, or gradient used for stroking (drawing) paths. This property can be a CSS color string or a `CanvasGradient` or a `CanvasPattern` object.

string textAlign

Specifies the horizontal alignment of text and the meaning of the X coordinate passed to `fillText()` and `strokeText()`. Legal values are “left”, “center”, “right”, “start”, and “end”. The meaning of “start” and “end” depend on the `dir` (text direction) attribute of the `<canvas>` tag. The default is “start”.

string textBaseline

Specifies the vertical alignment of text and the meaning of the Y coordinate passed to `fillText()` and `strokeText()`. Legal values are “top”, “middle”, “bottom”, “alphabetic”, “hanging”, and “ideographic”. The default is “alphabetic”.

Methods

void arc(double x, y, radius, startAngle, endAngle, [boolean anticlockwise])

This method adds an arc to the current subpath of a canvas, using a center point and radius. The first three arguments to this method specify the center and radius of a circle. The next two arguments are angles that specify the start and end points of an arc along the circle. These angles are measured in radians. The three o'clock position along the positive X axis is an angle of 0, and angles increase in the clockwise direction. The final argument specifies whether the arc is traversed counterclockwise (true) or clockwise (false or omitted) along the circle's circumference.

Invoking this method adds a straight line between the current point and the start point of the arc and then adds the arc itself to the current path.

void arcTo(double x1, y1, x2, y2, radius)

This method adds a straight line and an arc to the current subpath and describes that arc in a way that makes it particularly useful for adding rounded corners to polygons. The arguments `x1` and `y1` specify a point P1, and the arguments `x2` and `y2` specify a point P2. The arc that is added to the path is a portion of a circle with the specified *radius*. The arc has one point tangent to the line from the current position to P1 and one point that is tangent to the line from P1 to P2. The arc begins and ends at these two tangent points and is drawn in the direction that connects those two points with the shortest arc. Before adding the arc to the path, this method adds a straight line from the current point to the start point of the arc. After calling this method, the current point is at the end point of the arc, which lies on the line between P1 and P2.

Given a context object `c`, you can draw a 100x100 square with rounded corners (of varying radii) with code like this:

```
c.beginPath();
c.moveTo(150, 100);           // Start in the middle of the top edge
c.arcTo(200,100,200,200,40);  // Draw top edge and rounded upper right corner
c.arcTo(200,200,100,200,30);  // Draw right edge and (less) rounded lower right
c.arcTo(100,200,100,100,20);  // Draw bottom and rounded lower left corner
c.arcTo(100,100,200,100,10);  // Draw left and rounded upper left corner
c.closePath();                // Back to the starting point.
c.stroke();                    // Draw the path
```

void beginPath()

`beginPath()` discards any currently defined path and begins a new one. There is no current point after a call to `beginPath()`.

When the context for a canvas is first created, `beginPath()` is implicitly called.

void bezierCurveTo(double cp1x, cp1y, cp2x, cp2y, x, y)

bezierCurveTo() adds a cubic Bezier curve to the current subpath of a canvas. The start point of the curve is the current point of the canvas, and the end point is (x,y). The two Bezier control points (cpX1, cpY1) and (cpX2, cpY2) define the shape of the curve. When this method returns, the current position is (x,y).

void clearRect(double x, y, width, height)

clearRect() fills the specified rectangle with transparent black. Unlike **rect()**, it does not affect the current point or the current path.

void clip()

This method computes the intersection of the inside of the current path with the current clipping region and uses that (smaller) region as the new clipping region. Note that there is no way to enlarge the clipping region. If you want a temporary clipping region, you should first call **save()** so that you can later **restore()** the original clipping region. The default clipping region for a canvas is the canvas rectangle itself.

Like the **fill()** method, **clip()** treats all subpaths as closed and uses the nonzero winding rule for distinguishing the inside of the path from the outside of the path.

void closePath()

If the current subpath of the canvas is open, **closePath()** closes it by adding a line connecting the current point to the first point of the subpath. It then begins a new subpath (as if by calling **moveTo()**) at that same point.

fill() and **clip()** treat all subpaths as if they had been closed, so you only need to call **closePath()** explicitly if you want to **stroke()** a closed path.

ImageData createImageData(ImageData imagedata)

Returns a new **ImageData** object with the same dimensions as *data*.

ImageData createImageData(double w, double h)

Returns a new **ImageData** object with the specified width and height. All pixels within this new **ImageData** object are initialized to transparent black (all color components and alpha are 0).

The *w* and *h* arguments specify image dimensions in CSS pixels. Implementations are allowed to map single CSS pixels to more than one underlying device pixel. The **width** and **height** properties of the returned **ImageData** object specify the image dimensions in device pixels, and these values may not match the *w* and *h* arguments.

CanvasGradient createLinearGradient(double x0, y0, x1, y1)

This method creates and returns a new **CanvasGradient** object that linearly interpolates colors between the start point (x0,y0) and the end point (x1,y1). Note that this method does not specify any colors for the gradient. Use the **addColorStop()** method of the returned object to do that. To stroke lines or fill areas using a gradient, assign a **CanvasGradient** object to the **strokeStyle** or **fillStyle** properties.

CanvasPattern **createPattern**(*Element image*, *string repetition*)

This method creates and returns a **CanvasPattern** object that represents the pattern defined by a tiled image. The *image* argument must be an `<img>`, `<canvas>`, or `<video>` element containing the image to be used as the pattern. The *repetition* argument specifies how the image is tiled. The possible values are:

Value	Meaning
"repeat"	Tile the image in both directions. This is the default.
"repeat-x"	Tile the image in the X dimension only.
"repeat-y"	Tile the image in the Y dimension only.
"no-repeat"	Do not tile the image; use it a single time only.

To use a pattern for stroking lines or filling areas, use a **CanvasPattern** object as the value of the `strokeStyle` or `fillStyle` properties.

CanvasGradient **createRadialGradient**(*double x0, y0, r0, x1, y1, r1*)

This method creates and returns a new **CanvasGradient** object that radially interpolates colors between the circumferences of the two specified circles. Note that this method does not specify any colors for the gradient. Use the `addColorStop()` method of the returned object to do that. To stroke lines or fill areas using a gradient, assign a **CanvasGradient** object to the `strokeStyle` or `fillStyle` properties.

Radial gradients are rendered by using the color at offset 0 for the circumference of the first circle, the color at offset 1 for the second circle, and interpolated color values at circles between the two.

void drawImage(*Element image*, *double dx, dy, [dw, dh]*)

Copy the specified *image* (which must be an `<img>`, `<canvas>`, or `<video>` element) into the canvas with its upper left corner at (*dx*, *dy*). If *dw* and *dh* are specified, the image is scaled so that it is *dw* pixels wide and *dh* pixels high.

void drawImage(*Element image*, *double sx, sy, sw, sh, dx, dy, dw, dh*)

This version of the `drawImage()` method copies a source rectangle of the specified *image* into a destination rectangle of the canvas. *image* must be an `<img>`, `<canvas>`, or `<video>` element. (*sx*, *sy*) specifies the upper left corner of the source rectangle within that image and *sw* and *sh* specify the width and height of the source rectangle. Note that these arguments are in CSS pixels and are not subject to transformation. The remaining arguments specify the destination rectangle into which the image should be copied: see the five-argument version of `drawImage()` for details. Note that these destination rectangle arguments are transformed by the current transformation matrix.

void fill()

`fill()` fills the current path with the color, gradient, or pattern specified by the `fillStyle` property. Any subpaths that are not closed are filled as if the `closePath()` method had been called on them. (Note, however, that this does not actually cause those subpaths to become closed.)

Filling a path does not clear the path. You can call `stroke()` after calling `fill()` without redefining the path.

When the path intersects itself or when subpaths overlap, `fill()` canvas uses the nonzero winding rule to determine which points are inside the path and which are outside. This means, for example, that if your path defines a square inside of a circle and the square's subpath winds in the opposite direction of the circle's path, the interior of the square will be outside of the path and will not be filled.

void fillRect(double x, y, width, height)

`fillRect()` fills the specified rectangle with the color, gradient, or pattern specified by the `fillStyle` property.

Unlike the `rect()` method, `fillRect()` has no effect on the current point or the current path.

void fillText(string text, double x, y, [double maxWidth])

`fillText()` draws `text` using the current font and `fillStyle` properties. The `x` and `y` arguments specify where on the canvas the text should be drawn, but the interpretation of these arguments depends on the `textAlign` and `textBaseline` properties, respectively.

If `textAlign` is `left` or is `start` (the default) for a canvas that uses left-to-right text (also the default) or `end` for a canvas that uses right-to-left text, the text is drawn to the right of the specified X coordinate. If `textAlign` is `center`, the text is horizontally centered around the specified X coordinate. Otherwise (if `textAlign` is `"right"`, is `"end"` for left-to-right text, or is `"start"` for right-to-left text), the text is drawn to the left of the specified X coordinate.

If `textBaseline` is `"alphabetic"` (the default), `"bottom"`, or `"ideographic"`, most of the glyphs will appear above the specified Y coordinate. If `textBaseline` is `"center"`, the text will be approximately vertically centered on the specified Y coordinate. And if `textBaseline` is `"top"` or `"hanging"`, most of the glyphs will appear below the specified Y coordinate.

The optional `maxWidth` argument specifies a maximum width for the text. If the `text` would be wider than `maxWidth`, the text will be drawn using a smaller or more condensed version of the font instead.

ImageData getImageData(double sx, sy, sw, sh)

The arguments to this method are untransformed coordinates that specify a rectangular region of the canvas. The method copies the pixel data from that region of the canvas into a new `ImageData` object and returns that object. See `ImageData` for an explanation of how to access the red, green, blue, and alpha components of the individual pixels.

The RGB color components of the returned pixels are not premultiplied by the alpha value. If any portions of the requested rectangle lie outside the bounds of the canvas, the associated pixels in the `ImageData` are set to transparent black (all zeros). If the implementation uses more than one device pixel per CSS pixel, the `width` and `height` properties of the returned `ImageData` object will be different from the `sw` and `sh` arguments.

Like `Canvas.toDataURL()`, this method is subject to a security check to prevent cross-origin information leakage. `getImageData()` only returns an `ImageData` object if the underlying canvas is `"origin-clean"`; otherwise, it raises an exception. A canvas is not `origin-clean` if it has ever had an image drawn in it (directly by `drawImage()` or indirectly through a `CanvasPattern`)

that has a different origin than the document that contains the canvas. Also, a canvas is not origin-clean if it has ever had text drawn to it using a web font from a different origin.

boolean isPointInPath(double x, y)

`isPointInPath()` returns `true` if the specified point falls within or on the edge of the current path and returns `false` otherwise. The specified point is not transformed by the current transformation matrix. `x` should be a value between 0 and `canvas.width` and `y` should be a value between 0 and `canvas.height`.

The reason that `isPointInPath()` tests untransformed points is that it is designed for “hit-testing”: determining whether a user’s mouse click (for example) is on top of the portion of the canvas described by the path. In order to do hit-testing, mouse coordinates must first be translated so that they are relative to the canvas rather than the window. If the canvas’s size on the screen is different than the size declared by its `width` and `height` attributes (if `style.width` and `style.height` have been set, for example), the mouse coordinates also have to be scaled to match the canvas coordinates. The following function is designed for use as an onclick handler of a `<canvas>` and performs the necessary transformation to convert mouse coordinates to canvas coordinates:

```
// An onclick handler for a canvas tag. Assumes a path is currently defined.
function hittest(event) {
    var canvas = this;           // Called in the context of the canvas
    var c = canvas.getContext("2d"); // Get drawing context of the canvas

    // Get the canvas size and position
    var bb = canvas.getBoundingClientRect();

    // Convert mouse event coordinates to canvas coordinates
    var x = (event.clientX-bb.left)*(canvas.width/bb.width);
    var y = (event.clientY-bb.top)*(canvas.height/bb.height);

    // Fill the path if the user clicked on it
    if (c.isPointInPath(x,y)) c.fill();
}
```

void lineTo(double x, double y)

`lineTo()` adds a straight line to the current subpath. The line begins at the current point and ends at (x,y). When this method returns, the current position is (x,y).

TextMetrics measureText(string text)

`measureText()` measures the width that the specified `text` would occupy if drawn with the current font and returns a `TextMetrics` object containing the results of the measurement. At the time of this writing, the returned object has only a single `width` property, and the text height and bounding box are not measured.

void moveTo(double x, double y)

`moveTo()` sets the current position to (x,y) and begins a new subpath with this as its first point. If there was a previous subpath and it consisted of just one point, that empty subpath is removed from the path.

```
void putImageData(ImageData imagedata, double dx, dy, [sx, sy, sw, sh])
```

`putImageData()` copies a rectangular block of pixels from an `ImageData` object onto the canvas. This is a low-level pixel copy operation: the `globalCompositeOperation` and `globalAlpha` attribute are ignored, as are the clipping region, transformation matrix, and shadow-drawing attributes.

The `dx` and `dy` arguments specify the *destination* point in the canvas. Pixels from `data` will be copied to the canvas starting at that point. These arguments are not transformed by the current transformation matrix.

The last four arguments specify a source rectangle within the `ImageData`. If specified, only the pixels within that rectangle will be copied to the canvas. If these arguments are omitted, all pixels in the `ImageData` will be copied. If these arguments specify a rectangle that exceeds the bounds of the `ImageData`, the rectangle will be clipped to those bounds. Negative values for `sx` and `sy` are allowed.

One use for `ImageData` objects is as a “backing store” for a canvas—saving a copy of the canvas pixels in an `ImageData` (using `getImageData()`) allows you to draw temporarily on the canvas and then restore it to its original state with `putImageData()`.

```
void quadraticCurveTo(double cpx, cpy, x, y)
```

This method adds a quadratic Bezier curve segment to the current subpath. The curve starts at the current point and ends at (x,y) . The control point (cpX, cpY) specifies the shape of the curve between these two points. (The mathematics of Bezier curves is beyond the scope of this book, however.) When this method returns, the current position is (x,y) . Also see the `bezierCurveTo()` method.

```
void rect(double x, y, w, h)
```

This method adds a rectangle to the path. This rectangle is in a subpath of its own and is not connected to any other subpaths in the path. When this method returns, the current position is (x,y) . A call to this method is equivalent to the following sequence of calls:

```
c.moveTo(x,y);
c.lineTo(x+w, y);
c.lineTo(x+w, y+h);
c.lineTo(x, y+h);
c.closePath();
```

```
void restore()
```

This method pops the stack of saved graphics states and restores the values of the `CanvasRenderingContext2D` properties, the clipping path, and the transformation matrix. See the `save()` method for further information.

```
void rotate(double angle)
```

This method alters the current transformation matrix so that any subsequent drawing appears rotated within the canvas by the specified angle. It does not rotate the `<canvas>` element itself. Note that the angle is specified in radians. To convert degrees to radians, multiply by `Math.PI` and divide by 180.

void save()

save() pushes a copy of the current graphics state onto a stack of saved graphics states. This allows you to temporarily change the graphics state, and then restore the previous values with a call to **restore()**.

The graphics state of a canvas includes all the properties of the **CanvasRenderingContext2D** object (except for the read-only canvas property). It also includes the transformation matrix that is the result of calls to **rotate()**, **scale()**, and **translate()**. Additionally, it includes the clipping path, which is specified with the **clip()** method. Note, however, that the current path and current position are not part of the graphics state and are not saved by this method.

void scale(double sx, double sy)

scale() adds a scale transformation to the current transformation matrix of the canvas. Scaling is done with independent horizontal and vertical scaling factors. For example, passing the values 2.0 and 0.5 causes subsequently drawn paths to be twice as wide and half as high as they would otherwise have been. Specifying a negative value for **sx** causes X coordinates to be flipped across the Y axis, and a negative value of **sy** causes Y coordinates to be flipped across the X axis.

void setTransform(double a, b, c, d, e, f)

This method allows you to set the current transformation matrix directly rather than through a series of calls to **translate()**, **scale()**, and **rotate()**. After calling this method, the new transformation is:

$$\begin{array}{rcl} x' & a & c & e & x & = & ax + cy + e \\ y' & b & d & f & y & = & bx + dy + f \\ 1 & 0 & 0 & 1 & 1 & & \end{array}$$

void stroke()

The **stroke()** method draws the outline of the current path. The path defines the geometry of the line that is produced, but the visual appearance of that line depends on the **strokeStyle**, **lineWidth**, **lineCap**, **lineJoin**, and **miterLimit** properties.

The term *stroke* refers to a pen or brush stroke. It means “draw the outline of.” Contrast this **stroke()** method with **fill()**, which fills the interior of a path rather than stroking the outline of the path.

void strokeRect(double x, y, w, h)

This method draws the outline (but does not fill the interior) of a rectangle with the specified position and size. Line color and line width are specified by the **strokeStyle** and **lineWidth** properties. The appearance of the rectangle corners is specified by the **lineJoin** property.

Unlike the **rect()** method, **strokeRect()** has no effect on the current path or the current point.

void strokeText(string text, double x, y, [maxWidth])

strokeText() works just like **fillText()**, except that instead of filling the individual character glyphs with **fillStyle**, it strokes the outline of each glyph using **strokeStyle**. **strokeText()** produces interesting graphical effects when used at large font sizes, but **fillText()** is more commonly used for actually drawing text.

void transform(double a, b, c, d, e, f)

The arguments to this method specify the six nontrivial elements of a 3x3 affine transformation matrix T:

```
a c e
b d f
0 0 1
```

transform() sets the current transformation matrix to the product of the transformation matrix and the T:

$$CTM' = CTM \times T$$

Translations, scales, and rotations can be implemented in terms of this general-purpose **transform()** method. For a translation, call **transform(1,0,0,1,dx,dy)**. For a scale, call **transform(sx, 0, 0, sy, 0, 0)**. For a clockwise rotation around the origin by an angle x, use:

```
transform(cos(x),sin(x),-sin(x), cos(x), 0, 0)
```

For a shear by a factor of k parallel to the X axis, call **transform(1,0,k,1,0,0)**. For a shear parallel to the Y axis, call **transform(1,k,0,1,0,0)**.

void translate(double x, double y)

translate() adds horizontal and vertical offsets to the transformation matrix of the canvas. The arguments *dx* and *dy* are added to all points in any subsequently defined paths.

ClientRect

an element bounding box

A **ClientRect** object describes a rectangle, using Window or viewport coordinates. The **get BoundingClientRect()** method of **Element** returns objects of this kind to describe the on-screen bounding box of an element. **ClientRect** objects are x static: they do not change when the element they describe changes.

Properties

readonly float **bottom**

The Y position, in viewport coordinates, of the bottom edge of the rectangle.

readonly float **height**

The height, in pixels, of the rectangle. In IE8 and before, this property is not defined; use **bottom-top** instead.

readonly float **left**

The X position, in viewport coordinates, of the left edge of the rectangle.

readonly float **right**

The X position, in viewport coordinates, of the right edge of the rectangle.

readonly float **top**

The Y position, in viewport coordinates, of the top edge of the rectangle.

readonly float **width**

The width, in pixels, of the rectangle. In IE8 and before, this property is not defined; use `right-left` instead.

CloseEvent

specifies whether a WebSocket closed cleanly

Event

When a WebSocket connection closes, a nonbubbling, noncancelable close event is fired on the WebSocket object and an associated CloseEvent object is passed to any registered event handlers.

Properties

readonly boolean **wasClean**

If the WebSocket connection closed in the controlled way specified by WebSocket protocol, with acknowledgment between client and server, the close is said to be *clean*, and this property will be true. If this property is false, the WebSocket may have closed as the result of a network error of some sort.

Comment

an HTML or XML comment

Node

A Comment node represents a comment in an HTML or XML document. The content of the comment (i.e., the text between `<!--` and `-->`) is available through the `data` property or through the `nodeValue` property inherited from Node. You can create a comment object with `Document.createComment()`.

Properties

string **data**

The text of the comment.

readonly unsigned long **length**

The number of characters in the comment.

Methods

void **appendData**(string *data*)

void **deleteData**(unsigned long *offset*, unsigned long *count*)

void **insertData**(unsigned long *offset*, string *data*)

void **replaceData**(unsigned long *offset*, unsigned long *count*, string *data*)

string **substringData**(unsigned long *offset*, unsigned long *count*)

Comment nodes have most of the methods of a Text node, and those methods work as they do on Text nodes. They are listed here, but see Text for documentation.

Console

debugging output

Modern browsers (and older ones with debugger extensions, such as Firebug, installed) define a global property `console` that refers to a `Console` object. The methods of this object define a API for simple debugging tasks, such as logging messages to a console window (the console may go by a name such as “Developer Tools” or “Web Inspector”).

There is no formal standard that defines the `Console` API, but the Firebug debugger extension for Firefox has established a de facto standard and browser vendors seem to be implementing the Firebug API, which is documented here. Support for the basic `console.log()` function is nearly universal, but the other functions may not be as well supported in all browsers.

Note that in some older browsers, the `console` property is only defined when the console window is open, and running scripts that use the `Console` API without the console open will cause errors.

See also `ConsoleCommandLine`.

Methods

void `assert`(any *expression*, string *message*)

Display an error *message* on the console if *expression* is `false` or a falsy value like `null`, `undefined`, `0`, or the empty string.

void `count`([string *title*])

Display the specified *title* string along with a count of the number of times that this method has been called with that string.

void `debug`(any *message*...)

Like `console.log()`, but mark the output as debugging information.

void `dir`(any *object*)

Display the JavaScript *object* on the console in a way that allows the developer to examine its properties or elements and interactively explore nested objects or arrays.

void `dirxml`(any *node*)

Display XML or HTML markup for the document *node* in the console.

void `error`(any *message*...)

Like `console.log()`, but mark the output as an error.

void `group`(any *message*...)

Display *message* in the same way that `log()` does, but display it as the title of a collapsible group of debug messages. All subsequent console output will be formatted as part of this group until a corresponding call to `groupEnd()` occurs.

void `groupCollapsed`(any *message*...)

Begin a new group of messages, but start it in its collapsed state, so that subsequent debugging output is hidden by default.

void groupEnd()

End the debugging output group most recently started with `group()` or `groupCollapsed()`.

void info(any message...)

Like `console.log()`, but mark the output as an informative message.

void log(string format, any message...)

This method displays its arguments in the console. In the simplest case, when *format* does not contain any % characters, it simply converts its arguments to strings and displays them with spaces in between. When an object is passed to this method, the string that is displayed in the console will be clickable to view the contents of the object.

For more complex log messages, this method supports a simple subset of the C language `printf()` formatting capabilities. The *message* arguments will be interpolated into the *format* argument in place of the character sequences “%s”, “%d”, “%i”, “%f”, and “%o”, and then the formatted string will be displayed in the console (followed by any unused *message* arguments). Arguments that replace “%s” are formatted as strings. Those that replace “%d” or “%i” are formatted as integers. Those that replace “%f” are formatted as floating-point numbers, and those that replace “%o” are formatted as clickable objects.

void profile([string title])

Start the JavaScript profiler, and display *title* at the start of its report.

void profileEnd()

Stop the profiler and display its code profiling report.

void time(string name)

Start a timer with the specified *name*.

void timeEnd(string name)

End the timer with the specified *name*, and display the name and the time elapsed since the corresponding call to `time()`

void trace()

Display a stack trace.

void warn(any message...)

Like `console.log()`, but mark the output as a warning.

ConsoleCommandLine

global utilities for the console window

Most web browsers support a JavaScript console (which may be known by a name like “Developer Tools” or “Web Inspector”) that allows you to enter individual lines of JavaScript code. In addition to the normal global variables and functions of client-side JavaScript, the console command line typically supports the useful properties and functions described here. See also the Console API.

Properties

readonly Element \$0

The document element most recently selected via some other feature of the debugger.

readonly Element \$1

The document element selected before \$0.

Methods

void cd(Window frame)

When a document includes nested frames, the `cd()` function allows you to switch global objects and execute subsequent commands in the scope of the specified *frame*.

void clear()

Clear the console window.

void dir(object o)

Display the properties or elements of *o*. Like `Console.dir()`.

void dirxml(Element elt)

Display an XML or HTML-based representation of *elt*. Like `Console.dirxml()`.

Element \$(string id)

A shortcut for `document.getElementById()`.

NodeList \$\$ (string selector)

Return an array-like object of all elements matching the specified CSS *selector*. This is a shortcut for `document.querySelectorAll()`. In some consoles, it returns a true array rather than a `NodeList`.

void inspect(any object, [string tabname])

Display the *object*, possibly switching from the console to a different tab of the debugger. The second argument is an optional hint about how you would like the object displayed. Supported values may include “html”, “css”, “script”, and “dom”.

string[] keys(any object)

Returns an array of the property names for *object*.

void monitorEvents(Element object, [string type])

Log events of the specified *type* dispatched to *object*. Values for *type* include “mouse”, “key”, “text”, “load”, “form”, “drag”, and “contextmenu”. If *type* is omitted, all events on *object* are logged.

void profile(string title)

Begin code profiling. See `Console.profile()`.

void profileEnd()

End profiling. See `Console.profileEnd()`.

`void unmonitorEvents(Element object, [string type])`

Stop monitoring *type* events on *object*.

`any[] values(any object)`

Returns an array of the property values for *object*.

CSS2Properties

see `CSSStyleDeclaration`

CSSRule

a rule in a CSS stylesheet

Description

A `CSSRule` object represents a rule in a `CSSStyleSheet`: it represents style information to be applied to a specific set of document elements. `selectorText` is the string representation of the element selector for this rule, and `style` is a `CSSStyleDeclaration` object that represents the set of style attributes and values to apply to the selected elements.

The CSS Object Model specification actually defines a hierarchy of `CSSRule` subtypes to represent different kinds of rules that can appear in a CSS stylesheet. The properties listed here are properties of the generic `CSSRule` type and of its `CSSStyleRule` subtype. Style rules are the most common and most important types of rules in a stylesheet, and the ones you are most likely to script.

In IE8 and before, `CSSRule` objects support only the `selectorText` and `style` properties.

Constants

`unsigned short STYLE_RULE = 1`

`unsigned short IMPORT_RULE = 3`

`unsigned short MEDIA_RULE = 4`

`unsigned short FONT_FACE_RULE = 5`

`unsigned short PAGE_RULE = 6`

`unsigned short NAMESPACE_RULE = 10`

These are the possible values of the `type` property below and they specify what kind of rule it is. If `type` is anything other than 1, the `CSSRule` object will have other properties that are not documented here.

Properties

`string cssText`

The complete text of this CSS rule.

`readonly CSSRule parentRule`

The rule, if any, within which this rule is contained.

readonly CSSStyleSheet parentStyleSheet

The stylesheet within which this rule is contained.

string selectorText

When **type** is **STYLE_RULE**, this property holds the selector text that specifies the document elements this style rule applies to.

readonly CSSStyleDeclaration style

When **type** is **STYLE_RULE**, this property specifies the styles that should be applied to elements specified by **selectorText**. Note that while the **style** property itself is read-only, the properties of the **CSSStyleDeclaration** object to which it refers are read/write.

readonly unsigned short type

The type of this rule. The value will be one of the constants defined above.

CSSStyleDeclaration

a set of CSS attributes and their values

A **CSSStyleDeclaration** object represents a set of CSS style attributes and their values, and it allows those style values to be queried and set using JavaScript property names that are similar to CSS property names. The **style** property of an **HTMLElement** is a read/write **CSSStyleDeclaration** object, as is the **style** property of a **CSSRule** object. The return value of **Window.getComputedStyle()**, however, is a **CSSStyleDeclaration** object whose properties are read-only.

A **CSSStyleDeclaration** object makes CSS style attributes available through JavaScript properties. The names of these JavaScript properties correspond closely to the CSS attribute names, with minor changes required to avoid syntax errors in JavaScript. Multiword attributes that contain hyphens, such as “font-family”, are written without hyphens in JavaScript, with each word after the first capitalized: **fontFamily**. Also, the “float” attribute conflicts with the reserved word **float**, so it translates to the property **cssFloat**. Note that you can use unmodified CSS attribute names if you use strings and square brackets to access the properties.

Properties

In addition to the properties described above, a **CSSStyleDeclaration** has two additional properties:

string cssText

The textual representation of a set of style attributes and their values. The text is formatted as in a CSS stylesheet, minus the element selector and the curly braces that surround the attributes and values.

readonly unsigned long length

The number of attribute/value pairs contained in this **CSSStyleDeclaration**. A **CSSStyleDeclaration** object is also an array-like object whose elements are the names of the CSS style attributes that are declared.

CSSStyleSheet

a CSS stylesheet

This interface represents a CSS stylesheet. It has properties and methods for disabling the stylesheet, and for querying, inserting, and removing `CSSRule` style rules. The `CSSStyleSheet` objects that apply to a document are members of the `styleSheets[]` array of the `Document` object and may also be available through the `sheet` property of the `<style>` or `<link>` element that defines or links to the stylesheet.

In IE8 and before, use the `rules[]` array instead of `cssRules[]`, and use `addRule()` and `removeRule()` instead of the DOM standard `insertRule()` and `deleteRule()`.

Properties

readonly CSSRule[] cssRules

A read-only, array-like object holding the `CSSRule` objects that compose the stylesheet. In IE, use the `rules` property instead.

boolean disabled

If `true`, the stylesheet is disabled and is not applied to the document. If `false`, the stylesheet is enabled and is applied to the document.

readonly string href

The URL of a stylesheet that is linked to the document, or `null` for inline stylesheets.

readonly string media

A list of media to which this stylesheet applies. You can query and set this property as a single string, or treat it as an array-like object of media types with `appendMedium()` and `deleteMedium()` methods. (Formally, the value of this property is a `MediaList` object, but that type is not covered in this reference.)

readonly Node ownerNode

The document element that “owns” this stylesheet, or `null` if there isn’t one. See [Link](#) and [Style](#).

readonly CSSRule ownerRule

The `CSSRule` (from a parent stylesheet) that caused this stylesheet to be included, or `null` if this stylesheet was included in some other way. (Note that the entry for `CSSRule` in this reference only documents style rules, not `@import` rules.)

readonly CSSStyleSheet parentStyleSheet

The stylesheet that included this one, or `null` if this stylesheet was included directly in the document.

readonly string title

The title of the stylesheet, if specified. A title can be specified by the `title` attribute of a `<style>` or `<link>` element that refers to the stylesheet.

readonly string type

The MIME type of this stylesheet. CSS stylesheets have a type of “text/css”

Methods

void deleteRule(unsigned long index)

This method deletes the rule at the specified *index* from the *cssRules* array. In IE8 and before, use the equivalent method *removeRule()* instead.

unsigned long insertRule(string rule, unsigned long index)

This method inserts (or appends) a new CSS *rule* (a single string that specifies selector and styles within curly braces) at the specified *index* of the *cssRules* array of this style-sheet. In IE8 and before, use *addRule()* instead, and pass the selector string and the styles string (without curly braces) as two separate arguments, passing the index as the third argument.

DataTransfer

a transfer of data via drag-and-drop

When the user performs a drag-and-drop operation, a sequence of events is fired on the drag source or the drop target (or both, if they are both in a browser window). These events are accompanied by an event object whose *dataTransfer* property (see *Event*) refers to a *DataTransfer* object. The *DataTransfer* object is the central object for any drag-and-drop operation: the drag source stores the data to be transferred in it, and the drop target extracts the transferred data from it. In addition, the *DataTransfer* object manages a negotiation between the drag source and drop target about whether the drag-and-drop should be a copy, move, or link operation.

The API described here was created by Microsoft for IE, and it has been at least partially implemented by other browsers. HTML5 standardizes the basic IE API. As this book goes to press, HTML5 has defined a new version of the API that defines the *items* property as an array-like object of *DataTransferItem* objects. This is an appealing and rational API, but since no browsers yet implement it, it is not documented here. Instead, this page documents the features that (mostly) work in current browsers. See §17.7 for further discussion of this quirky API.

Properties

string dropEffect

This property specifies the type of data transfer this object represents. It must have one of the values “none”, “copy”, “move”, or “link”. Typically, the drop target will set this property from a *dragenter* or *dragover* event. The value of this property may also be affected by the modifier keys that the user holds down while performing the drag, but that is platform-dependent.

string effectAllowed

This property specifies what combination of copy, move, and link transfers are allowed for this drag-and-drop operation. It is typically set by the drag source in response to the *dragstart* event. Legal values are “none”, “copy”, “copyLink”, “copyMove”, “link”, “linkMove”, “move”, and “all”. (As a mnemonic, note that in the values that specify two operations, the operation names always appear in alphabetical order.)

readonly File[] files

If the data being dragged is one or more files, this property will be set to an array or array-like object of File objects.

readonly string[] types

This is an array-like object of strings that specify the MIME types of the data that has been stored in this `DataTransfer` object (with `setData()` if the drag source is within the browser or by some other mechanism if the drag source is outside of the browser). The array-like object that holds the types is supposed to have a `contains()` method for testing whether a specific string is present. Some browsers just make this a true array, however, and in that case, you can use the `indexOf()` method instead.

Methods**void addElement(Element element)**

This method tells the browser to use *element* when creating the visual effect that the user sees while dragging. This method is generally called by the drag source, and it may not be implemented or have any effect in all browsers.

void clearData([string format])

Removes any data in the specified *format* that was previously set with `setData()`.

string getData(string format)

Return the transferred data in the specified *format*. If *format* is equal (ignoring case) to “text”, use “text/plain” instead. And if it is equal (ignoring case) to “url”, use “text/uri-list” instead. This method is called by the drop target in response to the drop event at the end of a drag-and-drop operation.

void setData(string format, string data)

Specify the *data* to be transferred, and the MIME type *format* of that data. The drag source calls this method in response to a dragstart event at the beginning of a drag-and-drop operation. It cannot be called from any other event handler. If the drag source can make its data available in more than one format, it can call this method multiple times to register values for each supported format.

void setDragImage(Element image, long x, long y)

This method specifies an *image* (typically an `<img>` element) that should be displayed to the user as a visual representation of the value being dragged. The *x* and *y* coordinates give mouse pointer offsets within the image. This method can only be called by the drag source in response to the dragstart event.

DataView

read and write values from an `ArrayBuffer`

`ArrayBufferView`

A `DataView` is an `ArrayBufferView` that wraps an `ArrayBuffer` (or a region of an array buffer) and defines methods for reading and writing 1-, 2-, and 4-byte signed and unsigned integers

and 4- and 8-byte floating-point numbers from or into the buffer. The methods support both big-endian and little-endian byte orders. See also `TypedArray`.

Constructor

```
new DataView(ArrayBuffer buffer,
              [unsigned long byteOffset], [unsigned long byteLength])
```

This constructor creates a new `DataView` object that allows read and write access to the bytes in *buffer* or a region of *buffer*. With one argument, it creates a view of the entire buffer. With two arguments, it creates a view that extends from byte number *byteOffset* to the end of the buffer. And with three arguments, it creates a view of the *byteLength* bytes starting at *byteOffset*.

Methods

Each of these method reads a numeric value from, or writes a numeric value into, the underlying `ArrayBuffer`. The method name specifies the type that is read or written. All methods that read or write more than one byte accept an optional final *littleEndian* argument. If this argument is omitted, or is `false`, big-endian byte ordering is used, with the most significant bytes being read or written first. If the argument is `true`, however, little-endian byte ordering is used.

```
float getFloat32(unsigned long byteOffset, [boolean littleEndian])
```

Interpret the 4 bytes starting at *byteOffset* as a floating-point number and return that number.

```
double getFloat64(unsigned long byteOffset, [boolean littleEndian])
```

Interpret the 8 bytes starting at *byteOffset* as a floating-point number and return that number.

```
short getInt16(unsigned long byteOffset, [boolean littleEndian])
```

Interpret the 2 bytes starting at *byteOffset* as a signed integer number and return that number.

```
long getInt32(unsigned long byteOffset, [boolean littleEndian])
```

Interpret the 4 bytes starting at *byteOffset* as a signed integer number and return that number.

```
byte getInt8(unsigned long byteOffset)
```

Interpret the byte at *byteOffset* as a signed integer number and return that number.

```
unsigned short getUint16(unsigned long byteOffset, [boolean littleEndian])
```

Interpret the 2 bytes starting at *byteOffset* as an unsigned integer number and return that number.

```
unsigned long getUint32(unsigned long byteOffset, [boolean littleEndian])
```

Interpret the 4 bytes starting at *byteOffset* as an unsigned integer number and return that number.

```
unsigned byte getUint8(unsigned long byteOffset)
```

Interpret the byte at *byteOffset* as an unsigned integer number and return that number.

```
void setFloat32(unsigned long byteOffset, float value, [boolean littleEndian])
```

Convert *value* to a 4-byte floating-point representation and write those bytes at *byteOffset*.

void setFloat64(unsigned long *byteOffset*, double *value*, [boolean *littleEndian*])
Convert *value* to an 8-byte floating-point representation and write those bytes at *byteOffset*.

void setInt16(unsigned long *byteOffset*, short *value*, [boolean *littleEndian*])
Convert *value* to a 2-byte integer representation and write those bytes at *byteOffset*.

void setInt32(unsigned long *byteOffset*, long *value*, [boolean *littleEndian*])
Convert *value* to a 4-byte integer representation and write those bytes at *byteOffset*.

void setInt8(unsigned long *byteOffset*, byte *value*)
Convert *value* to a 1-byte integer representation and write that byte at *byteOffset*.

void setUint16(unsigned long *byteOffset*, unsigned short *value*, [boolean *littleEndian*])
Convert *value* to a 2-byte unsigned integer representation and write those bytes at *byteOffset*.

void setUint32(unsigned long *byteOffset*, unsigned long *value*, [boolean *littleEndian*])
Convert *value* to a 4-byte unsigned integer representation and write those bytes at *byteOffset*.

void setUint8(unsigned long *byteOffset*, octet *value*)
Convert *value* to a 1-byte unsigned integer representation and write that byte at *byteOffset*.

Document

an HTML or XML document

Node

A Document object is a Node that serves as the root of a document tree. The `document` Element property is the root Element of the document. A Document node may have other children (such as Comment and DocumentType nodes), but it has only one Element child that holds all the content of the document.

You most commonly obtain a Document object via the `document` property of a Window. Document objects are also available through the `contentDocument` property of IFrame elements or the `ownerDocument` property of any Node.

Most of the properties of a Document object provide access to elements of the document or to other important objects associated with the document. A number of Document methods do the same thing: provide a way to look up elements within the document tree. Many other Document methods are “factory methods” that create elements and related objects.

Documents, like the Elements they contain, are event targets. They implement the methods defined by EventTarget, and they also support quite a few event handler properties.

You can create new Document objects by using the `createDocument()` and `createHTMLDocument()` methods of the DOMImplementation:

```
document.implementation.createHTMLDocument("New Doc");
```

It is also possible to load an HTML or XML file from the network and parse it into a Document object. See the `responseXML` property of the `XMLHttpRequest` object.

The reference entry for `HTMLDocument`, which appeared in previous versions of this book, has been merged into this page. Note that some of the properties, methods, and event handlers described here are HTML-specific and will not work for XML documents.

Properties

In addition to the properties listed here, you can also use the value of the `name` attribute of `<iframe>`, `<form>`, and `<img>` elements as document properties. The value of these properties is the named Element or a `NodeList` of such elements. For named `<iframe>` elements, however, the property refers to the Window object of the `<iframe>`. See §15.2.2 for details.

readonly Element `activeElement`

The document element that currently has the keyboard focus.

Element `body`

For HTML documents, this element refers to the `<body>` Element. (For documents that define framesets, this property refers to the outermost `<frameset>` instead.)

readonly string `characterSet`

The character encoding of this document.

string `charset`

The character encoding of this document. This is like `characterSet`, but you can set it to change the encoding.

readonly string `compatMode`

This property is the string “BackCompat” if the document is being rendered in CSS “quirks mode” for backward compatibility with very old browsers. Otherwise, this property is “CSS1Compat”.

string `cookie`

This property allows you to read, create, modify, and delete the cookie or cookies that apply to the current document. A *cookie* is a small amount of named data stored by the web browser. It gives web browsers a “memory” so they can use data input on one page in another page or recall user preferences across web browsing sessions. Cookie data is automatically transmitted between web browser and web server when appropriate so server-side scripts can read and write cookie values. Client-side JavaScript code can also read and write cookies with this property. Note that this is a read/write property but the value you read from it is not, in general, the same as the value you write. See §20.2 for details.

readonly string `defaultCharset`

The browser’s default character set.

readonly Window `defaultView`

The web browser Window object in which this document is displayed.

string designMode

If this property is “on”, the entire document is editable. If it is “off”, the entire document is not editable. (But elements with their `contentEditable` property set may still be editable, of course.) See §15.10.4.

string dir

For HTML documents, this property mirrors the `dir` attribute of the `<html>` element. It is the same, therefore, as `documentElement.dir`.

readonly DocumentType doctype

The `DocumentType` node that represents the document’s `<!DOCTYPE>`.

readonly Element documentElement

The root element of the document. For HTML documents, this property is always the `Element` object representing the `<html>` tag. This root element is also available through the `childNodes[]` array inherited from `Node`, but it is not generally the first element of that array. See also the `body` property.

string domain

The hostname of the server from which the document was loaded, or `null` if there is none. You can set this property to a suffix of itself in order to relax the same-origin policy and grant access to documents served from related domains. See §13.6.2.1 for details.

readonly HTMLCollection embeds

An array-like object of `<embed>` elements in the document.

readonly HTMLCollection forms

An array-like object of all `Form` elements in the document.

readonly Element head

For HTML documents, this property refers to the `<head>` element.

readonly HTMLCollection images

An array-like object of all `Image` elements in the document.

readonly DOMImplementation implementation

The `DOMImplementation` object for this document.

readonly string lastModified

Specifies the date and time of the most recent modification to the document. This value comes from the Last-Modified HTTP header that is optionally sent by the web server.

readonly HTMLCollection links

An array-like object of all hyperlinks in the document. This `HTMLCollection` contains all `<a>` and `<area>` elements that have `href` attributes, and does not include `<link>` elements. See `Link`.

readonly Location location

A synonym for `Window.location`.

readonly HTMLCollection plugins

A synonym for the `embeds` property.

readonly string `readyState`

This property is the string “loading” if the document is still loading and “complete” if it is fully loaded. The browser fires a `readystatechange` event at the Document when this property changes to “complete”.

readonly string `referrer`

The URL of the document that linked to this document, or `null` if this document was not accessed through a hyperlink or if the web server did not report the referring document. This property allows client-side JavaScript to access the HTTP `referer` header. Note the spelling difference, however: the HTTP header has three r's, and the JavaScript property has four r's.

readonly `HTMLCollection` `scripts`

An array-like object of all the `<script>` elements in the document.

readonly `CSSStyleSheet[]` `styleSheets`

A collection of objects representing all stylesheets embedded in or linked into a document. In HTML documents, this includes stylesheets defined with `<link>` and `<style>` tags.

string `title`

The plain-text content of the `<title>` tag for this document.

readonly string `URL`

The URL from which the document was loaded. This value is often the same as the `location.href` property, but if a script changes the fragment identifier (the `location.hash` property), the `location` property and the `URL` property will no longer refer to the same URL. Don't confuse `Document.URL` with `Window.URL`.

Methods**Node `adoptNode(Node node)`**

This method removes `node` from whatever document it is currently part of and changes its `ownerDocument` property to this document, making it ready for insertion into this document. Contrast this with `importNode()`, which copies a node from another document without removing it.

void `close()`

Closes a document stream opened with the `open()` method, forcing any buffered output to be displayed.

Comment `createComment(string data)`

Create and return a new Comment node with the specified content.

DocumentFragment `createDocumentFragment()`

Create and return a new, empty DocumentFragment node.

Element **createElement**(string *localName*)

Create and return a new, empty Element node with the specified tag name. In HTML documents, the tag name is converted to uppercase.

Element **createElementNS**(string *namespace*, string *qualifiedName*)

Create and return a new, empty, Element node. The first argument specifies the namespace URI for the element, and the second argument specifies the namespace prefix, a colon, and the tag name of the element.

Event **createEvent**(string *eventInterface*)

Create and return an uninitialized synthetic Event object. The argument must specify the type of event, and the argument should be a string such as “Event”, “UIEvent”, “MouseEvent”, “MessageEvent”, or so on. After creating an Event object, you can initialize its read-only properties by calling an appropriate event-initialization method on it, such as `initEvent()`, `initUIEvent()`, `initMouseEvent()`, or so on. Most of these event-specific initialization methods are not covered in this book, but see `Event.initEvent()` for the simplest one. When you have created and initialized an synthetic event object, you can dispatch it using the `dispatchEvent()` method of `EventTarget`. Synthetic events will always have an `isTrusted` property of false.

ProcessingInstruction **createProcessingInstruction**(string *target*, string *data*)

Creates and returns a new ProcessingInstruction node with the specified target and data string.

Text **createTextNode**(string *data*)

Creates and returns a new Text node to represent the specified text.

Element **elementFromPoint**(float *x*, float *y*)

Return the most deeply nested Element at window coordinates (*x*, *y*).

boolean **execCommand**(string *commandId*, [boolean *showUI*, [string *value*]])

Execute the editing command named by the *commandId* argument in whatever editable element has the insertion cursor. HTML5 defines the following commands:

<code>bold</code>	<code>insertLineBreak</code>	<code>selectAll</code>
<code>createLink</code>	<code>insertOrderedList</code>	<code>subscript</code>
<code>delete</code>	<code>insertUnorderedList</code>	<code>superscript</code>
<code>formatBlock</code>	<code>insertParagraph</code>	<code>undo</code>
<code>forwardDelete</code>	<code>insertText</code>	<code>unlink</code>
<code>insertImage</code>	<code>italic</code>	<code>unselect</code>
<code>insertHTML</code>	<code>redo</code>	

Some of these commands (such as “createLink”) require an argument value. If the second argument to `execCommand()` is false, the third argument gives the argument that the command is to use. Otherwise, the browser will prompt the user for the necessary value. See §15.10.4 for more on `execCommand()`.

Element `getElementById(string elementId)`

This method searches the document for an Element node with an `id` attribute whose value is *elementId* and returns that Element. If no such Element is found, it returns `null`. The value of the `id` attribute is intended to be unique within a document, but if this method finds more than one Element with the specified *elementId*, it returns the first. This is an important and commonly used method because it provides a simple way to obtain the Element object that represents a specific document element. Note that the name of this method ends with “Id”, not with “ID”.

NodeList `getElementsByClassName(string classNames)`

Returns an array-like object of elements that have a `class` attribute that includes all of the specified *classNames*. *classNames* may be a single class or a space-separated list of classes. The returned NodeList object is live and is automatically updated as the document changes. The elements in the returned NodeList appear in the same order as they appear in the document. Note that this method is also defined on Element.

NodeList `getElementsByName(string elementName)`

This method returns a live, read-only, array-like object of Elements that have a `name` attribute whose value is *elementName*. If there are no matching elements, it returns a NodeList with length 0.

NodeList `getElementsByTagName(string qualifiedName)`

This method returns a read-only array-like object that contains all Element nodes from the document that have the specified tag name, in the order in which they appear in the document source. The NodeList is “live”—its contents are automatically updated as necessary when the document changes. For HTML elements, tag name comparison is case-insensitive. As a special case, the tag name “*” matches all elements in a document.

Note that the Element interface defines a method by the same name that searches only a subtree of the document.

NodeList `getElementsByTagNameNS(string namespace, string localName)`

This method works like `getElementsByTagName()`, but it specifies the desired tag name as a combination of namespace URI and local name within that namespace.

boolean `hasFocus()`

This method returns `true` if this document’s Window has the keyboard focus (and, if that window is not a top-level window, all of its ancestors are focused).

Node `importNode(Node node, boolean deep)`

This method is passed a node defined in another document and returns a copy of the node that is suitable for insertion into this document. If *deep* is `true`, all descendants of the node are also copied. The original node and its descendants are not modified in any way. The returned copy has its `ownerDocument` property set to this document but has a `parentNode` of `null` because it has not yet been inserted into the document. Event-listener functions registered on the original node or tree are not copied. See also `adoptNode()`.

Window `open(string url, string name, string features, [boolean replace])`

When the `open()` method of a document is invoked with three or more arguments, it acts just like the `open()` method of the `Window` object. See `Window`.

Document `open([string type], [string replace])`

With two or fewer arguments, this method erases the current document and begins a new one (using the existing `Document` object, which is the return value). After calling `open()`, you can use the `write()` and `writeln()` methods to stream content to the document and `close()` to end the document and force its new content to be displayed. See §15.10.2 for details.

The new document will be an HTML document if `type` is omitted or is “text/html”. Otherwise, it will be a plain text document. If the `replace` argument is true, the new document replaces the old one in the browsing history.

This method should not be called by a script or event handler that is part of the document being overwritten, because the script or handler will itself be overwritten.

boolean `queryCommandEnabled(string commandId)`

Returns `true` if it is currently meaningful to pass `commandId` to `execCommand()` and `false` otherwise. The “undo” command, for example, is not enabled if there is nothing to undo. See §15.10.4.

boolean `queryCommandIndeterm(string commandId)`

Returns `true` if `commandId` is in an indeterminate state for which `queryCommandState()` cannot return a meaningful value. Commands defined by HTML5 are never indeterminate, but browser-specific commands might be. See §15.10.4.

boolean `queryCommandState(string commandId)`

Return the state of the specified `commandId`. Some editing commands, such as “bold” and “italic,” have a state `true` if the cursor or selection is in italic and `false` if it is not. Most commands are stateless, however, and this method always returns `false` for those. See §15.10.4.

boolean `queryCommandSupported(string commandId)`

Returns `true` if the browser supports the specified command and `false` otherwise. See §15.10.4.

string `queryCommandValue(string commandId)`

Returns the state of the specified command as a string. See §15.10.4.

Element `querySelector(string selectors)`

Returns the first element in this document that matches the specified CSS `selectors` (this may be a single CSS selector or a comma-separated group of selectors).

NodeList querySelectorAll(string selectors)

Returns an array-like object containing all Elements in this Document that match the specified *selectors* (this may be a single CSS selector or a comma-separated group of selectors). Unlike the NodeLists returned by `getElementsByTagName()` and similar methods, the NodeList returned by this method is not live: it is just a static snapshot of the elements that matched when the method was called.

void write(string text...)

This method appends its arguments to the document. You can use this method while the document is loading to insert content at the location of the `<script>` tag, or you can use it after calling the `open()` method. See §15.10.2 for details.

void writeln(string text...)

This method is like `HTMLDocument.write()`, except that it follows the appended text with a newline character, which may be useful when writing the content of a `<pre>` tag, for example.

Events

Browsers do not fire many events directly at Document objects, but Element events do bubble up to the Document that contains them. Therefore, Document objects support all of the event handler properties listed in Element. Like Elements, Document objects implement the `EventTarget` methods.

Browsers do fire two document readiness events at the Document object. When the `readyState` property changes, the browser fires a `readystatechange` event. You can register a handler for this event with the `onreadystatechange` property. The browser also fires a `DOMContentLoaded` event (see §17.4) when the document tree is ready (but before external resources have finished loading). You must use an `EventTarget` method to register a handler for those events, however, since there is an `onDOMContentLoaded` property.

DocumentFragment

adjacent nodes and their subtrees

Node

The `DocumentFragment` interface represents a portion—or fragment—of a document. More specifically, it is a list of adjacent nodes and all descendants of each, but without any common parent node. `DocumentFragment` nodes are never part of a document tree, and the inherited `parentNode` property is always `null`. `DocumentFragment` nodes exhibit a special behavior that makes them quite useful, however: when a request is made to insert a `DocumentFragment` into a document tree, it is not the `DocumentFragment` node itself that is inserted but instead each child of the `DocumentFragment`. This makes `DocumentFragment` useful as a temporary placeholder for nodes that you wish to insert, all at once, into a document.

You can create a new, empty `DocumentFragment` with `Document.createDocumentFragment()`.

You can search for elements in a `DocumentFragment` with `querySelector()` and `querySelectorAll()`, which work just like the same methods of the Document object.

Methods

Element **querySelector**(string *selectors*)

See `Document.querySelector()`.

NodeList **querySelectorAll**(string *selectors*)

See `Document.querySelectorAll()`.

DocumentType

the `<!DOCTYPE>` declaration of a document

Node

This infrequently used type represents the `<!DOCTYPE>` declaration of a document. The `doctype` property of a `Document` holds the `DocumentType` node for that document. `DocumentType` nodes are immutable and cannot be modified in any way.

`DocumentType` nodes are used to create new `Document` objects with `DOMImplementation.createDocument()`. You can create new `DocumentType` objects with `DOMImplementation.createDocumentType()`.

Properties

readonly string **name**

The name of the document type. This identifier immediately follows `<!DOCTYPE>` at the start of a document, and it is the same as the tag name of the document's root element. For HTML documents, this will be "html".

readonly string **publicId**

The public identifier of the DTD, or the empty string if none was specified.

readonly string **systemId**

The system identifier of the DTD, or the empty string if none was specified.

DOMException

an exception thrown by a Web API

Most client-side JavaScript APIs throw `DOMException` objects when they need to signal an error. The `code` and `name` properties of the object provide more details about the error. Note that a `DOMException` can be thrown when reading or writing a property of an object as well as when calling a method of an object.

`DOMException` is not a subclass of the core JavaScript `Error` type, but it functions like one, and some browsers include a `message` property for compatibility with `Error`.

Constants

```

unsigned short INDEX_SIZE_ERR = 1
unsigned short HIERARCHY_REQUEST_ERR = 3
unsigned short WRONG_DOCUMENT_ERR = 4
unsigned short INVALID_CHARACTER_ERR = 5
unsigned short NO_MODIFICATION_ALLOWED_ERR = 7
unsigned short NOT_FOUND_ERR = 8
unsigned short NOT_SUPPORTED_ERR = 9
unsigned short INVALID_STATE_ERR = 11
unsigned short SYNTAX_ERR = 12
unsigned short INVALID_MODIFICATION_ERR = 13
unsigned short NAMESPACE_ERR = 14
unsigned short INVALID_ACCESS_ERR = 15
unsigned short TYPE_MISMATCH_ERR = 17
unsigned short SECURITY_ERR = 18
unsigned short NETWORK_ERR = 19
unsigned short ABORT_ERR = 20
unsigned short URL_MISMATCH_ERR = 21
unsigned short QUOTA_EXCEEDED_ERR = 22
unsigned short TIMEOUT_ERR = 23
unsigned short DATA_CLONE_ERR = 25

```

These are the possible values of the `code` property. The constant names are verbose enough to indicate the approximate reason that the exception was thrown.

Properties

unsigned short `code`

One of the constant values listed above, indicating what type of exception occurred.

string `name`

The name of the specific exception type. This will be one of the constant names listed above, as a string.

DOMImplementation

global DOM methods

The `DOMImplementation` object defines methods that are not specific to any particular Document object but rather are “global” to an implementation of the DOM. You can obtain a reference to the `DOMImplementation` object through the `implementation` property of any Document object.

Methods

Document **createDocument**(string *namespace*, string *qualifiedName*, DocumentType *doctype*)

This method creates and returns a new XML Document object. If *qualifiedName* is specified, a root element with that name is created and added to the document as its `documentElement`. If *qualifiedName* includes a namespace prefix and a colon, *namespace* should be the URI that uniquely identifies the namespace. If the *doctype* argument is non-null, the `ownerDocument` property of this DocumentType object is set to the newly created document and the DocumentType node is added to the new document.

DocumentType **createDocumentType**(string *qualifiedName*, publicId, systemId)

This method creates and returns a new DocumentType node to represent a `<!DOCTYPE>` declaration that you can pass to `createDocument()`.

Document **createHTMLDocument**(string *title*)

This method creates a new HTMLDocument object with a skeletal document tree that includes the specified title. The `documentElement` property of the returned object is an `<html>` element, and this root element has `<head>` and `<body>` tags as its children. The `<head>` element in turn has a `<title>` child, which has the specified *title* string as its child.

DOMSettableTokenList

a token list with a settable string value

DOMTokenList

A DOMSettableTokenList is a DOMTokenList that also has a `value` property that can be set to specify the entire set of tokens at once.

The `classList` property of `Element` is a DOMTokenList that represents the set of tokens in the `className` property, which is a string. If you want to set all the `classList` tokens at once, you can simply set the `className` property to a new string. The `sandbox` property of the `IFrame` element is a little different. This property and the HTML attribute that it is based on was defined by HTML5 and so there is no need for an old string representation and a new DOMTokenList representation. In this case, the property is simply defined as a DOMSettableTokenList; you can read it and write it as if it were a string, or you can use its methods and use it as a set of tokens. The `htmlFor` property of `Output` and the `audio` property of `Video` are also DOMSettableTokenLists.

Properties

string **value**

The space-separated string representation of the set of tokens. You can read or write this property to treat the set as a single string value. You do not normally need to use this property explicitly, however: when you use a DOMSettableTokenList as a string, it is this string value that is returned. And if you assign a string to a DOMSettableTokenList, this property is implicitly set.

DOMTokenList

a set of space-separated tokens

A `DOMTokenList` is a parsed representation of a space-separated string of tokens, such as the `className` property of a `Element`. A `DOMTokenList` is, as its name implies, a list—it is an array-like object with a `length` property and you can index it to retrieve the individual tokens. But more importantly, it defines methods `contains()`, `add()`, `remove()`, and `toggle()` methods that allow you to treat it as a set of tokens. If you use a `DOMTokenList` as if it was a string, it evaluates to a space-separated string of tokens.

The HTML5 `classList` property of `Element` objects is a `DOMTokenList`, in browsers that support that property, and is the only `DOMTokenList` you are likely to use often. See also `DOMSettableTokenList`.

Properties

readonly unsigned long **length**

A `DOMTokenList` is an array-like object; this property specifies the number of unique tokens it contains.

Methods

void **add**(string *token*)

If the `DOMTokenList` does not already contain *token*, add it at the end of the list.

boolean **contains**(string *token*)

Returns `true` if the `DOMTokenList` contains *token*, or `false` otherwise.

string **item**(unsigned long *index*)

Return the token at the specified *index* or `null` if *index* is out of bounds. You can also index the `DOMTokenList` directly instead of calling this method.

void **remove**(string *token*)

If this `DOMTokenList` contains *token*, remove it. Otherwise, do nothing.

boolean **toggle**(string *token*)

If the `DOMTokenList` contains *token*, remove it. Otherwise add it.

Element

a document element

Node, EventTarget

An `Element` object represents an element in an HTML or XML document. The `tagName` property specifies the tag name or type of the element. Standard HTML attributes of the element are available through JavaScript properties of the `Element` object. Attributes, including XML attributes and nonstandard HTML attributes can also be accessed with the `getAttribute()` and `setAttribute()` methods. Element content is available through properties inherited from `Node`. If you are only interested in the `Element` relatives of an `Element`, you can use the `children` property or `firstElementChild`, `nextElementSibling`, and related properties.

There are a number of ways to obtain `Element` objects from documents. The `document` `Element` property of a `Document` refers to the root element for that document, such as the `<html>` element of an HTML document. For HTML documents, the `head` and `body` properties are similar: they refer to the `<head>` and `<body>` elements of the document. To locate a specific named element by its unique `id` attribute, use `Document.getElementById()`. As described in §15.2, you can also obtain `Element` objects with `Document` and `Element` methods such as `getElementsByTagName()`, `getElementsByClassName()`, and `querySelectorAll()`. Finally, you can create new `Element` objects for insertion into a document with `Document.createElement()`.

Web browsers fire many different kinds of events on document elements, and `Element` objects define many event handler properties. In addition, `Element` objects define the `EventTarget` methods (see `EventTarget`) for adding and removing event listeners.

The reference entry for `HTMLElement`, which appeared in previous versions of this book, has been merged with this section. Note that some of the properties, methods, and event handlers described here are HTML-specific and will not work with the elements of XML documents.

Properties

In addition to the properties listed here, the HTML attributes of HTML elements are accessible as JavaScript properties of the `Element` object. HTML tags and their legal attributes are listed at the end of this reference entry.

readonly Attr[] attributes

An array-like object of `Attr` objects that represent the HTML or XML attributes of this element. `Element` objects generally make attributes accessible through JavaScript properties, however, so it is never really necessary to use this `attributes[]` array.

readonly unsigned long childElementCount

The number of child elements (not child nodes) that this element has.

readonly HTMLCollection children

An array-like object of the `Element` children (excluding non-`Element` children, such as `Text` and `Comment` nodes) of this `Element`.

readonly DOMTokenList classList

The `class` attribute of an element is a space-separated list of class names. This property allows access to the individual elements of that list and defines methods for querying, adding, removing, and toggling class names. See `DOMTokenList` for details.

string className

This property represents the `class` attribute of the element. `class` is a reserved word in JavaScript, so the JavaScript property is `className` instead of `class`. Note that this property name is misleading, since the `class` attribute often includes more than one class name.

readonly long clientHeight

readonly long clientWidth

If this element is the root element (see `document.documentElement`), these properties return the dimensions of the Window. These are the inner or viewport dimensions that exclude

scrollbars and other browser “chrome”. Otherwise, these properties return the dimensions of the element’s content plus padding.

readonly long clientLeft

readonly long clientTop

These properties return the number of pixels between the left or top edge of the element’s border and the left or top edge of its padding. Normally this is just the left and top border width, but these amounts may also include the width of a scrollbar if one is rendered on the left or top of the element.

CSSStyleDeclaration currentStyle

This IE-specific property represents the cascaded set of all CSS properties that apply to the element. You can use it in IE8 and before as a substitute for the standard `Window.getComputedStyle()` method.

readonly object dataset

You can associate arbitrary values with any HTML element by assigning those values to attributes whose names begin with the special prefix “data-”. This `dataset` property is the set of data attributes for an element and makes it easy to set and query them.

The value of this property behaves like a regular JavaScript object. Each property of the object corresponds to one data attribute on the element. If the element has an attribute named `data-x`, the `dataset` object has a property named `x`, and `dataset.x` has the same value as `getAttribute("data-x")` does.

Querying and setting properties of the `dataset` object queries and sets the corresponding data attributes of this element. You can use the `delete` operator to remove data attributes, and you can use a `for/in` loop to enumerate the data attributes.

readonly Element firstElementChild

This property is like the `firstChild` property of `Node`, but it ignores `Text` and `Comment` nodes and only returns `Elements`.

string id

The value of the `id` attribute. No two elements within the same document should have the same value for `id`.

string innerHTML

A read/write string that specifies the HTML or XML markup that is contained within the element, not including the opening and closing tags of the element itself. Querying this property returns the content of the element as a string of HTML or XML text. Setting this property to a string of HTML or XML text replaces the content of the element with the parsed representation of that text.

readonly boolean isContentEditable

This property is `true` if the element is editable or `false` otherwise. An element may be editable because of the `contenteditable` property on it or an ancestor or because of the `designMode` property of the containing `Document`.

string lang

The value of the `lang` attribute, which specifies the language code for the element’s content.

readonly Element lastElementChild

This property is like the `lastChild` property of `Node`, but it ignores `Text` and `Comment` nodes and only returns `Elements`.

readonly string localName

The local, unprefixed name for this element. This differs from the `tagName` attribute, which includes the namespace prefix if there is one (and is converted to uppercase for HTML elements).

readonly string namespaceURI

The URL that formally defines the namespace for this element. This can be `null` or a string such as `"http://www.w3.org/1999/xhtml"`.

readonly Element nextElementSibling

This property is like the `nextSibling` property of `Node`, but it ignores `Text` and `Comment` nodes and only returns `Elements`.

readonly long offsetHeight**readonly long offsetWidth**

The height and width, in pixels, of the element and all its content, including the element's CSS padding and border, but not its margin.

readonly long offsetLeft**readonly long offsetTop**

The X and Y coordinates of the upper left corner of the CSS border of the element relative to the `offsetParent` container element.

readonly Element offsetParent

Specifies the container element that defines the coordinate system in which `offsetLeft` and `offsetTop` are measured. For most elements, `offsetParent` is the `<body>` element that contains them. However, if an element has a dynamically positioned container, the dynamically positioned element is the `offsetParent`, and if the element is in a table, a `<td>`, `<th>`, or `<table>` element may be the `offsetParent`. See §15.8.5.

string outerHTML

The HTML or XML markup that defines this element and its children. If you set this property to a string, you replace this element (and all of its content) with the result of parsing the new value as an HTML or XML document fragment.

readonly string prefix

The namespace prefix for this element. This is usually `null`, unless you are working with an XML document that uses namespaces.

readonly Element previousElementSibling

This property is like the `previousSibling` property of `Node`, but it ignores `Text` and `Comment` nodes and only returns `Elements`.

readonly long scrollHeight**readonly long scrollWidth**

The overall height and width, in pixels, of an element. When an element has scrollbars (because of the CSS `overflow` attribute, for example), these properties differ from `offset`

Height and `offsetWidth`, which simply report the size of the visible portion of the element.

long `scrollLeft`

long `scrollTop`

The number of pixels that have scrolled off the left edge of the element or off the top edge of the element. These properties are useful only for elements with scrollbars, such as elements with the CSS `overflow` attribute set to `auto`. When these properties are queried on the `<html>` element (see `Document.documentElement`), they specify the amount of scrolling for the document as a whole. Note that these properties do not specify the amount of scrolling in an `<iframe>` tag. You can set these properties to scroll an element or the entire document. See §15.8.5.

readonly `CSSStyleDeclaration style`

The value of the `style` attribute that specifies inline CSS styles for this element. Note that the value of this property is not a string but an object with read/write properties that correspond to CSS style attributes. See `CSSStyleDeclaration` for details.

readonly string `tagName`

The tag name of the element. For HTML documents, the tag name is returned in uppercase, regardless of its capitalization in the document source, so a `<p>` element would have a `tagName` property of "P". XML documents are case-sensitive, and the tag name is returned exactly as it is written in the document source. This property has the same value as the inherited `nodeName` property of the Node interface.

string `title`

The value of the `title` attribute of the element. Many browsers display the value of this attribute in a tool tip when the mouse hovers over the element.

Methods

void `blur()`

This method transfers keyboard focus to the `body` element of the containing Document object.

void `click()`

This method simulates a click on this element. If clicking on this element would normally make something happen (following a link, for example), this method makes that happen, too. Otherwise, calling this method just triggers a click event on the element.

void `focus()`

Transfer keyboard focus to this element.

string `getAttribute(string qualifiedName)`

`getAttribute()` returns the value of a named attribute of an element or `null` if no attribute with that name exists. Note that the `HTMLElement` object defines JavaScript properties that match each of the standard HTML attributes, so you need to use this method with HTML documents only if you are querying the value of nonstandard attributes. In HTML documents, attribute name comparisons are case-insensitive.

In XML documents, attribute values are not available directly as element properties and must be looked up by calling this method. For XML documents that use namespaces, include the namespace prefix and colon in the attribute name passed to this method or use `getAttributeNS()` instead.

string `getAttributeNS(string namespace, string localName)`

This method works just like the `getAttribute()` method, except that the attribute is specified by a combination of namespace URI and local name within that namespace.

ClientRect `getBoundingClientRect()`

Returns a **ClientRect** object that describes the bounding box of this element.

ClientRect[] `getClientRects()`

Returns an array-like object of **ClientRects** that describes one or more rectangles occupied by this element. (Inline elements that span more than one line usually require more than one rectangle to accurately describe their region of the window.)

NodeList `getElementsByClassName(string classNames)`

Returns an array-like object of descendant elements that have a `class` attribute that includes all of the specified `classNames`. `classNames` may be a single class or a space-separated list of classes. The returned **NodeList** object is live and is automatically updated as the document changes. The elements in the returned **NodeList** appear in the same order as they appear in the document. Note that this method is also defined on **Document**.

NodeList `getElementsByTagName(string qualifiedName)`

This method traverses all descendants of this element and returns an live array-like **NodeList** of **Element** nodes representing all document elements with the specified tag name. The elements in the returned array appear in the same order in which they appear in the source document.

Note that **Document** objects also have a `getElementsByTagName()` method that works just like this one but that traverses the entire document, rather than just the descendants of a single element.

NodeList `getElementsByTagNameNS(string namespace, string localName)`

This method works like `getElementsByTagName()`, except that the tag name of the desired elements is specified as a combination of a namespace URI and a local name defined within that namespace.

boolean `hasAttribute(string qualifiedName)`

This method returns `true` if this element has an attribute with the specified name and `false` otherwise. In HTML documents, attribute names are case-insensitive.

boolean `hasAttributeNS(string namespace, string localName)`

This method works like `hasAttribute()`, except that the attribute is specified by namespace URI and local name within that namespace.

void insertAdjacentHTML(string position, string text)

This method inserts the specified HTML markup *text* at the specified *position* relative to this element. The *position* argument must be one of these four strings:

Position	Meaning
beforebegin	Insert the text before the opening tag
afterend	Insert the text after the closing tag
afterbegin	Insert the text right after the opening tag
beforeend	Insert the text right before the closing tag

Element querySelector(string selectors)

Returns the first descendant of this element that matches the specified CSS *selectors* (this may be a single CSS selector or a comma-separated group of selectors).

NodeList querySelectorAll(string selectors)

Returns an array-like object containing all descendants of this Element that match the specified *selectors* (this may be a single CSS selector or a comma-separated group of selectors). Unlike the NodeList returned by `getElementsByTagName()`, the NodeList returned by this method is not live: it is just a static snapshot of the elements that matched when the method was called.

void removeAttribute(string qualifiedName)

`removeAttribute()` deletes a named attribute from this element. Attempts to remove non-existent attributes are silently ignored. In HTML documents, attribute names are case-insensitive.

void removeAttributeNS(string namespace, string localName)

`removeAttributeNS()` works just like `removeAttribute()`, except that the attribute to be removed is specified by namespace URI and local name.

void scrollIntoView([boolean top])

If an HTML element is not currently visible in the window, this method scrolls the document so that it becomes visible. The *top* argument is an optional hint about whether the element should be positioned near the top or bottom of the window. If *true* or omitted, the browser will attempt to position the element near the top. If *false*, the browser will attempt to position the element near the bottom. For elements that accept the keyboard focus, such as Input elements, the `focus()` method implicitly performs this same scroll-into-view operation. See also the `scrollTo()` method of Window.

void setAttribute(string qualifiedName, string value)

This method sets the specified attribute to the specified value. If no attribute by that name already exists, a new one is created. In HTML documents, the attribute name is converted to lowercase before being set. Note that HTML element objects of an HTML document define JavaScript properties that correspond to all standard HTML attributes, and you can set attributes directly with those properties. Thus, you need to use this method only if you want to set a nonstandard attribute.

```
void setAttributeNS(string namespace, string qualifiedName, string value)
```

This method is like `setAttribute()`, except that the attribute to be created or set is specified with a namespace URI and a qualified name that consists of a namespace prefix, a colon, and a local name within the namespace.

Event Handlers

Element objects that represent HTML elements define quite a few event handler properties. Set any of the properties listed below to a function, and that function will be invoked when a specific type of event occurs on (or bubbles up to) the element. You can also use the methods defined by `EventTarget` to register event handlers, of course.

Most events bubble up the document hierarchy to the Document node, and then on from there to the Window object. So each of the event handler properties listed here are also defined on the Document and Window object. The Window object has quite a few event handlers of its own, however, and the properties marked with an asterisk in the table below have a different meaning on the Window object. For historical reasons, event handlers registered as HTML attributes of the `<body>` element are registered on the Window object, and this means that the handler properties with asterisks have a different meaning on the `<body>` element than they do on other elements. See [Window](#).

Many of the events listed here are only triggered on certain types of HTML elements. But because most of those events bubble up the document tree, the event handler properties are defined generically for all elements. The HTML5 media events fired on `<audio>` and `<video>` tags do not bubble, so they are documented in [MediaElement](#). Similarly, some HTML5 form-related events do not bubble and are covered under [FormControl](#).

Event Handler	Invoked When...
<code>onabort</code>	resource loading canceled at user's request
<code>onblur*</code>	element loses input focus
<code>onchange</code>	user changes form control content or state (fired for complete edits, not individual keystrokes)
<code>onclick</code>	element activated by mouse click or other means
<code>oncontextmenu</code>	context menu is about to be displayed, usually because of a right-click
<code>ondblclick</code>	two rapid mouse clicks occur
<code>ondrag</code>	drag continues (triggered on drag source)
<code>ondragend</code>	drag ends (triggered on drag source)
<code>ondragenter</code>	drag enters (triggered on drop target)
<code>ondragleave</code>	drag leaves (triggered on drop target)
<code>ondragover</code>	drag continues (triggered on drop target)
<code>ondragstart</code>	user initiates drag-and-drop (triggered on drag source)
<code>ondrop</code>	user completes drag-and-drop (triggered on drop target)
<code>onerror*</code>	resource loading failed (usually because of a network error)
<code>onfocus*</code>	element gains keyboard focus
<code>oninput</code>	input occurs on a form element (triggered more frequently than <code>onchange</code>)

Event Handler	Invoked When...
onkeydown	the user presses a key
onkeypress	a keypress generates a printable character
onkeyup	the user releases a key
onload*	resource loading (e.g., for) has completed
onmousedown	the user presses a mouse button
onmousemove	the user moves the mouse
onmouseout	the mouse leaves an element
onmouseover	the mouse enters an element
onmouseup	the user releases a mouse button
onmousewheel	the user rotates the mouse wheel
onreset	a <form> is reset
onscroll*	an element with scrollbars is scrolled
onselect	the user selects text in a form element
onsubmit	when a <form> is submitted

HTML Elements and Attributes

This reference section includes individual reference pages for the following HTML element types:

Element(s)	Reference Page	Element(s)	Reference Page
<audio>	Audio	<output>	Output
<button>, <input type="button">	Button	<progress>	Progress
<canvas>	Canvas	<script>	Script
<fieldset>	FieldSet	<select>	Select
<form>	Form	<style>	Style
<iframe>	IFrame	<td>	TableCell
	Image	<tr>	TableRow
<input>	Input	<tbody>, <tfoot>, <thead>	TableSection
<label>	Label	<table>	Table
<a>, <area>, <link>	Link	<textarea>	TextArea
<meter>	Meter	<video>	Video
<option>	Option		

The HTML elements that do not have reference pages of their own are those whose only properties simply mirror the HTML attributes of the element. The following attributes are legal on any HTML element, and are therefore properties of all Element objects:

Attribute	Description
<code>accessKey</code>	keyboard shortcut
<code>class</code>	CSS class: see the <code>className</code> and <code>classList</code> properties above.
<code>contentEditable</code>	Whether element content is editable.
<code>contextMenu</code>	The ID of a <code><menu></code> element to display as a context menu. Supported only by IE at the time of this writing.
<code>dir</code>	Text direction: "ltr" or "rtl".
<code>draggable</code>	A boolean attribute set on elements that are drag sources for the Drag-and-Drop API.
<code>dropzone</code>	An attribute set on elements that are drop targets for the Drag-and-Drop API.
<code>hidden</code>	A boolean attribute set on elements that should not be displayed.
<code>id</code>	A unique identifier for the element.
<code>lang</code>	The primary language of the text in the element.
<code>spellcheck</code>	Whether element text should have spelling checked.
<code>style</code>	Inline CSS styles for the element. See the <code>style</code> property above.
<code>tabIndex</code>	Specifies the focus order of the element.
<code>title</code>	Tooltip text for the element.

The following HTML elements define no attributes other than the global attributes above:

<code><abbr></code>	<code><code></code>	<code><footer></code>	<code><hr></code>	<code><rt></code>	<code><sup></code>
<code><address></code>	<code><datalist></code>	<code><h1></code>	<code><i></code>	<code><ruby></code>	<code><tbody></code>
<code><article></code>	<code><dd></code>	<code><h2></code>	<code><kbd></code>	<code><s></code>	<code><tfoot></code>
<code><aside></code>	<code><dfn></code>	<code><h3></code>	<code><legend></code>	<code><samp></code>	<code><thead></code>
<code></code>	<code><div></code>	<code><h4></code>	<code><mark></code>	<code><section></code>	<code><title></code>
<code><bdi></code>	<code><dl></code>	<code><h5></code>	<code><nav></code>	<code><small></code>	<code><tr></code>
<code><bdo></code>	<code><dt></code>	<code><h6></code>	<code><noscript></code>	<code></code>	<code></code>
<code>
</code>	<code></code>	<code><head></code>	<code><p></code>	<code></code>	<code><var></code>
<code><caption></code>	<code><figcaption></code>	<code><header></code>	<code><pre></code>	<code><sub></code>	<code><wbr></code>
<code><cite></code>	<code><figure></code>	<code><hgroup></code>	<code><rp></code>	<code><summary></code>	

The remaining HTML elements, and the attributes they support, are listed below. Note that this table only lists attributes other than the global attributes described above. Also note that this table includes elements that also have their own reference page:

Element	Attributes
<code><a></code>	<code>href</code> , <code>target</code> , <code>ping</code> , <code>rel</code> , <code>media</code> , <code>hreflang</code> , <code>type</code>
<code><area></code>	<code>alt</code> , <code>coords</code> , <code>shape</code> , <code>href</code> , <code>target</code> , <code>ping</code> , <code>rel</code> , <code>media</code> , <code>hreflang</code> , <code>type</code>
<code><audio></code>	<code>src</code> , <code>preload</code> , <code>autoplay</code> , <code>loop</code> , <code>controls</code>

Element	Attributes
<base>	href, target
<blockquote>	cite
<body>	onafterprint, onbeforeprint, onbeforeunload, onblur, onerror, onfocus, onhashchange, onload, onmessage, onoffline, ononline, onpagehide, onpageshow, onpopstate, onredo, onresize, onscroll, onstorage, onundo, onunload
<button>	autofocus, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, name, type, value
<canvas>	width, height
<col>	span
<colgroup>	span
<command>	type, label, icon, disabled, checked, radiogroup
<del>	cite, datetime
<details>	open
<embed>	src, type, width, height,
<fieldset>	disabled, form, name
<form>	accept-charset, action, autocomplete, enctype, method, name, novalidate, target
<html>	manifest
<iframe>	src, srcdoc, name, sandbox, seamless, width, height
	alt, src, usemap, ismap, width, height
<input>	accept, alt, autocomplete, autofocus, checked, dirname, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, height, list, max, maxlength, min, multiple, name, pattern, placeholder, readonly, required, size, src, step, type, value, width
<ins>	cite, datetime
<keygen>	autofocus, challenge, disabled, form, keytype, name
<label>	form, for
	value
<link>	href, rel, media, hreflang, type, sizes
<map>	name
<menu>	type, label
<meta>	name, http-equiv, content, charset
<meter>	value, min, max, low, high, optimum, form
<object>	data, type, name, usemap, form, width, height
	reversed, start
<optgroup>	disabled, label
<option>	disabled, label, selected, value
<output>	for, form, name

Element	Attributes
<param>	name, value
<progress>	value, max, form
<q>	cite
<script>	src, async, defer, type, charset
<select>	autofocus, disabled, form, multiple, name, required, size
<source>	src, type, media
<style>	media, type, scoped
<table>	summary
<td>	colspan, rowspan, headers
<textarea>	autofocus, cols, disabled, form, maxlength, name, placeholder, readonly, required, rows, wrap
<th>	colspan, rowspan, headers, scope
<time>	datetime, pubdate
<track>	default, kind, label, src, srclang
<video>	src, poster, preload, autoplay, loop, controls, width, height

ErrorEvent

an uncaught exception from a worker thread

Event

When an uncaught exception occurs in a `Worker` thread, and the exception is not handled by the `onerror` function in the `WorkerGlobalScope`, that exception causes a nonbubbling error event to be triggered on the `Worker` object. The event has an `ErrorEvent` object associated with it to provide details about the exception that occurred. Calling `preventDefault()` on the `ErrorEvent` object (or returning `false` from the event handler) will prevent the error from propagating further to containing threads and may also prevent it from being displayed in an error console.

Properties

readonly string **filename**

The URL of the JavaScript file in which the exception was originally thrown.

readonly unsigned long **lineno**

The line number within that file at which the exception was thrown.

readonly string **message**

A message describing the exception.

Event

details for standard events, IE events and jQuery events

When an event handler is invoked, it is passed an Event object whose properties give details about the event, such as the type of event and the element on which it occurred. The methods of this Event object can control the propagation of the event. All modern browsers implement a standard event model, except IE, which, in version 8 and before, defines its own incompatible model. This page documents the standard event object properties and methods and the IE alternatives to them, and also covers the jQuery event object, which emulates a standard event object for IE. Read more about events in Chapter 17 and more about jQuery events in §19.4.

In the standard event model, different kinds of events have different kinds of event objects associated with them: mouse events have a MouseEvent object with mouse-related properties, for example, and keyboard events have a KeyEvent with key-related properties. Both the MouseEvent and KeyEvent types share a common Event superclass. In the IE and jQuery event models, however, a single Event object type is used for all events that can occur on Element objects. Event properties that are specific to keyboard events won't have a useful value when a mouse event occurs, but those properties will still be defined. For simplicity, this page collapses the event hierarchy and documents the properties for all events that can be delivered to Element objects (and that then bubble up to the Document and Window objects).

Originally, almost all client-side JavaScript events were triggered on document elements, and it is therefore natural to lump the properties of document-related event objects together like this. But HTML5 and related standards introduce a number of new event types that are triggered on objects that are not document elements. These event types often have Event types of their own, and those types are covered on their own reference pages. See BeforeUnloadEvent, CloseEvent, ErrorEvent, HashChangeEvent, MessageEvent, PageTransitionEvent, PopStateEvent, ProgressEvent, and StorageEvent.

Most of those event object types extend Event. Other new HTML5-related event types do not define an event object type of their own—the object associated with those events is just an ordinary Event object. This page documents that “ordinary” Event object plus the properties of some of its subtypes. The properties marked with an asterisk in the list below are the ones that are defined by the Event type itself. These are the properties that are inherited by event types like MessageEvent and are the properties that are defined for simple, ordinary events like the load event of the Window object and the playing event of a MediaElement object.

Constants

These constants define the values of the eventPhase property. That property, and these constants, are not supported in the IE event model.

unsigned short **CAPTURING_PHASE** = 1

The event is being dispatched to capturing event handlers registered on ancestors of its target.

unsigned short **AT_TARGET** = 2

The event is being dispatched at its target.

unsigned short **BUBBLING_PHASE** = 3

The event is bubbling and is being dispatched on ancestors of its target.

Properties

The properties listed here are defined by the standard event model for Event objects and also for the event objects associated with mouse and key events. Properties from the IE and jQuery event models are also listed. Properties with an asterisk are defined directly by Event and are universally available on any standard Event object, regardless of the event type.

readonly boolean **altKey**

Whether the Alt key was held down when the event occurred. Defined for mouse and key events and also by IE events.

readonly boolean **bubbles***

true if the event is of a type that bubbles (unless `stopPropagation()` is called); false otherwise. Not defined by IE events.

readonly unsigned short **button**

Which mouse button changed state during a mousedown, mouseup, or click event. A value of 0 indicates the left button, a value of 2 indicates the right button, and a value of 1 indicates the middle mouse button. Note that this property is defined when a button changes state; it is not used to report whether a button is held down during a mousemove event, for example. Also, this property is not a bitmap: it cannot tell you if more than one button is held down. Finally, some browsers only generate events for left button clicks.

IE events define an incompatible **button** property. In that browser, this property is a bit mask: the 1 bit is set if the left button was pressed, the 2 bit is set if the right button was pressed, and the 4 bit is set if the middle button (of a three-button mouse) was pressed. jQuery does not emulate the standard **button** property in IE, but see the **which** property instead.

readonly boolean **cancelable***

true if the default action associated with the event can be canceled with `preventDefault()`; false otherwise. Defined by all standard event types, but not by IE events.

boolean **cancelBubble**

In the IE event model, if an event handler wants to stop an event from being propagated up to containing objects, it must set this property to **true**. Use the `stopPropagation()` method for standard events.

readonly integer **charCode**

For keypress events, this property is the Unicode encoding of the printable character that was generated. This property is 0 for nonprinting function keys and is not used for keydown and keyup events. Use `String.fromCharCode()` to convert this value to a string. Most browsers set `keyCode` to the same value as this property for keypress events. In Firefox, however, `keyCode` is undefined for keypress events and you must use `charCode`. This property is nonstandard and is not defined in IE events or emulated by jQuery.

readonly long `clientX`

readonly long `clientY`

The X and Y coordinates of the mouse pointer relative to the *client area*, or browser window. Note that these coordinates do not take document scrolling into account; if an event occurs at the very top of the window, `clientY` is 0 regardless of how far down the document has been scrolled. These properties are defined for all types of mouse events. These properties are defined for IE events as well as standard events. See also `pageX` and `pageY`.

readonly boolean `ctrlKey`

Whether the Ctrl key was held down when the event occurred. Defined for mouse and key events and also by IE events.

readonly EventTarget `currentTarget`*

The Element, Document, or Window that is currently handling this event. During capturing and bubbling, this is different from `target`. Not defined by IE events but emulated by jQuery events.

readonly DataTransfer `dataTransfer`

For drag-and-drop events, this property specifies the `DataTransfer` object that coordinates the entire drag-and-drop operation. Drag-and-drop events are a kind of mouse event; any event that has this property set will also have `clientX`, `clientY`, and other mouse event properties. The drag-and-drop events are `dragstart`; `drag` and `drag end` on the drag source; and `dragenter`, `dragover`, `dragleave`, and `drop` on the drop target. See `DataTransfer` and §17.7 for details on drag-and-drop operations.

readonly boolean `defaultPrevented`*

`true` if `defaultPrevented()` has been called on this event or `false` otherwise. This is a new addition to the standard event model and may not be implemented in all browsers. (jQuery events define an `isDefaultPrevented()` method that works like this property.)

readonly long `detail`

A numeric detail about the event. For click, mousedown, and mouseup events, this field is the click count: 1 for a single-click, 2 for a double-click, 3 for a triple-click, and so on. In Firefox, `DOMMouseScroll` events use this property to report mousewheel scroll amounts.

readonly unsigned short `eventPhase`*

The current phase of event propagation. The value is one of the three constants defined above. Not supported by IE events.

readonly boolean `isTrusted`*

`true` if this event was created and dispatched by the browser or `false` if it is a synthetic event that was created and dispatched by JavaScript code. This is a relatively new addition to the standard event model and may not be implemented by all browsers.

readonly Element `fromElement`

For mouseover and mouseout events in IE, `fromElement` refers to the object from which the mouse pointer is moving. For standard events, use the `relatedTarget` property.

readonly integer `keyCode`

The virtual keycode of the key that was pressed. This property is used for all types of keyboard events. Keycodes may be browser-, OS-, and keyboard-hardware-dependent. Typically, when a key displays a printing character on it, the virtual keycode for that key is the same as the encoding of the character. Key codes for nonprinting function keys may vary more, but see Example 17-8 for a set of commonly used codes. This property has not been standardized but is defined by all browsers, including IE.

readonly boolean `metaKey`

Whether the Meta key was held down when the event occurred. Defined for mouse and key events and also by IE events.

readonly integer `offsetX`, `offsetY`

For IE events, these properties specify the coordinates at which the event occurred within the coordinate system of the event's source element (see `srcElement`). Standard events have no equivalent properties.

readonly integer `pageX`, `pageY`

These nonstandard, but widely supported, properties are like `clientX` and `clientY`, but use document coordinates rather than window coordinates. IE events do not define these properties, but jQuery emulates them for all browsers.

readonly EventTarget `relatedTarget`*

Refers to an event target (usually a document element) that is related to the `target` node of the event. For mouseover events, it is the element the mouse left when it moved over the target. For mouseout events, it is the element the mouse entered when leaving the target. This property is not defined by IE events but is emulated by jQuery events. See the IE properties `fromElement` and `toElement`.

boolean `returnValue`

For IE events, set this property to `false` to cancel the default action of the source element on which the event occurred. For standard events, use the `preventDefault()` method instead.

readonly long `screenX`, `screenY`

For mouse events, these properties specify the X and Y coordinates of the mouse pointer relative to the upper left corner of the user's monitor. These properties are not generally useful but are defined for all types of mouse events and are supported by standard events and IE events.

readonly boolean `shiftKey`

Whether the Shift key was held down when the event occurred. Defined for mouse and key events and also by IE events.

readonly EventTarget `srcElement`

For IE events, this property specifies the object on which the event was triggered. For standard events, use `target` instead.

readonly EventTarget target*

The target object for this event—i.e., the object on which the event was triggered. (All objects that can be event targets implement the methods of `EventTarget`.) This property is not defined for IE events, but it is emulated by jQuery events. See `srcElement`.

readonly unsigned long timeStamp*

A number that specifies the date and time at which the event occurred or that can at least be used to determine the order in which two events occurred. Many browsers return a timestamp that you can pass to the `Date()` constructor. In Firefox 4 and before, however, this property is some other kind of timestamp, such as the number of milliseconds since the computer was booted. IE events do not support it. jQuery sets this property to a timestamp in the format returned by `Date.getTime()`.

Element toElement

For mouseover and mouseout events in IE, `toElement` refers to the object into which the mouse pointer is moving. For standard events, use `relatedTarget` instead.

readonly string type*

The name of the event that this Event object represents. This is the name under which the event handler was registered or the name of the event-handler property with the leading “on” removed—for example, “click”, “load”, or “submit”. This property is defined by standard events and IE events.

readonly Window view

The window (called a “view” for historical reasons) in which the event was generated. This property is defined for all standard user-interface events, such as mouse and keyboard events. It is not supported in IE events.

readonly integer wheelDelta

For mousewheel events, this property specifies the amount of scrolling that has occurred in the Y axis. Different browsers set different values on this property: see §17.6 for details. This is a nonstandard property but is supported by all browsers, including IE8 and before.

readonly integer wheelDeltaX**readonly integer wheelDeltaY**

For mousewheel events in browsers that support two-dimensional mouse wheels, these properties specify the amount of scrolling in the X and Y dimensions. See §17.6 for an explanation of how to interpret these properties. If `wheelDeltaY` is defined, it will have the same value as the `wheelDelta` property.

readonly integer which

This nonstandard legacy property is supported by browsers other than IE and is emulated in jQuery. For mouse events, it is one more than the `button` property: 1 means the left button, 2 means the middle button, and 3 means the right button. For key events, it has the same value as `keyCode`.

Methods

All of these methods are defined by the Event class itself, so they are each available on any standard Event object.

void `initEvent`(string *type*, boolean *bubbles*, boolean *cancelable*)

This method initializes the *type*, *bubbles*, and *cancelable* properties of an Event object. Create a new event object by passing the string "Event" to the `createEvent()` method of Document. Then, after initializing it with this method, dispatch it on any EventTarget by passing it to the `dispatchEvent()` method of that target. The other standard event properties (besides *type*, *bubbles*, and *cancelable*) will be initialized by the dispatch. If you want to create, initialize, and dispatch a more complicated synthetic event, you'll have to pass a different argument (such as "MouseEvent") to `createEvent()` and then initialize the event object with a type-specific initializing function such as `initMouseEvent()` (not documented in this book).

void `preventDefault`()

Tells the web browser not to perform the default action associated with this event, if there is one. If the event is not of a type that is cancelable, this method has no effect. This method is not defined on IE event objects, but is emulated by jQuery. In the IE event model, set the `returnValue` property to false instead.

void `stopImmediatePropagation`()

Like `stopPropagation()`, but in addition, prevent the invocation of any other handlers registered on the same document element. This method is a new addition to the standard event model and may not be implemented in all browsers. It is not supported in the IE event model but is emulated by jQuery.

void `stopPropagation`()

Stops the event from propagating any further through the capturing, target, or bubbling phases of event propagation. After this method is called, any other event handlers for the same event on the same node are called, but the event is not dispatched to any other nodes. This method is not supported in the IE event model, but it is emulated by jQuery. In IE, set `cancelBubble` to true instead of calling `stopPropagation()`.

Proposed Properties

The properties listed here are proposed by the current draft of the DOM Level 3 Events specification. They address key areas of incompatibility among today's browsers but are not yet (at the time of this writing) implemented by any browsers. If implemented interoperably they will make it much easier to write portable code to handle text input events, key events, and mouse events.

readonly unsigned short `buttons`

This property is like IE's version of the `button` property described above.

readonly string `char`

For keyboard events, this property holds the character string (which may have more than one character) generated by the event.

readonly string `data`

For textinput events, this property specifies the text that was input.

readonly unsigned long `deltaMode`

For wheel events, this property specifies the appropriate interpretation of the `deltaX`, `deltaY`, and `deltaZ` properties. The value of this property will be one of these constants:

`DOM_DELTA_PIXEL`, `DOM_DELTA_LINE`, `DOM_DELTA_PAGE`. The value of this property is determined in a platform-dependent way. It may depend on system preferences or on keyboard modifiers held down during the wheel event.

readonly long `deltaX`, `deltaY`, `deltaZ`

For wheel events, these properties specify how much the mousewheel rotated around each of its three possible axes.

readonly unsigned long `inputMethod`

For textinput events, this property specifies how the text was input. The value will be one of these constants: `DOM_INPUT_METHOD_UNKNOWN`, `DOM_INPUT_METHOD_KEYBOARD`, `DOM_INPUT_METHOD_PASTE`, `DOM_INPUT_METHOD_DROP`, `DOM_INPUT_METHOD_IME`, `DOM_INPUT_METHOD_OPTION`, `DOM_INPUT_METHOD_HANDWRITING`, `DOM_INPUT_METHOD_VOICE`, `DOM_INPUT_METHOD_MULTIMODAL`, `DOM_INPUT_METHOD_SCRIPT`.

readonly string `key`

For keyboard events that generate characters, this property has the same value as `char`. If the keyboard event did not generate characters, this property holds the name of the key (such as “Tab” or “Down”) that was pressed.

readonly string `locale`

For keyboard events and textinput events, this property specifies a language code (such as “en-GB”) that identifies the locale for which the keyboard was configured, if that information is known.

readonly unsigned long `location`

For keyboard events, this property specifies the keyboard location of the key that was pressed. The value will be one of these constants: `DOM_KEY_LOCATION_STANDARD`, `DOM_KEY_LOCATION_LEFT`, `DOM_KEY_LOCATION_RIGHT`, `DOM_KEY_LOCATION_NUMPAD`, `DOM_KEY_LOCATION_MOBILE`, `DOM_KEY_LOCATION_JOYSTICK`.

readonly boolean `repeat`

For keyboard events, this property will be `true` if the event was caused because a key was held down long enough to begin repeating.

Proposed Method

Like the Proposed Properties listed above, the method listed here has been proposed in a draft standard but not yet implemented by any browsers.

boolean `getModifierState(string modifier)`

For mouse and keyboard events, this method returns `true` if the specified *modifier* key was held down when the event occurred, or `false` otherwise. *modifier* may be one of the strings “Alt”, “AltGraph”, “CapsLock”, “Control”, “Fn”, “Meta”, “NumLock”, “Scroll”, “Shift”, “SymbolLock”, and “Win”.

EventSource

a Comet connection to an HTTP server

EventTarget

An EventSource represents a long-lived HTTP connection through which a Web server can “push” textual messages. To use these “Server Sent Events”, pass the server URL to the EventSource() constructor and then register a message event handler on the resulting EventSource object.

Server Sent Events are new, and at the time of this writing, are not supported in all browsers.

Constructor

```
new EventSource(string url)
```

Creates a new EventSource object connected to the web server at the specified *url*. *url* is interpreted relative to the URL of the containing document.

Constants

These constants define the possible values of the readyState property.

```
unsigned short CONNECTING = 0
```

The connection is being set up, or the connection closed and the EventSource is re-connecting.

```
unsigned short OPEN = 1
```

The connection is open and events are being dispatched.

```
unsigned short CLOSED = 2
```

The connection was closed, either because close() was called or a fatal error occurred and it is not possible to reconnect.

Properties

```
readonly unsigned short readyState
```

The state of the connection. The constants above define the possible values.

```
readonly string url
```

The absolute URL to which the EventSource is connected.

Methods

```
void close()
```

This method closes the connection. Once this method is called, the EventSource object can no longer be used. If you need to connect again, create a new EventSource.

Event Handlers

Network communication is asynchronous, so EventSource triggers events when the connection opens, when an error occurs, and when messages arrive from the server. You can register event handlers on the properties listed here, or you can use the methods of EventTarget instead. EventSource events are all dispatched on the EventSource object itself. They do not bubble and have no default action that can be canceled.

onerror

Triggered when an error occurs. The associated event object is a simple *Event*.

onmessage

Triggered when a message arrives from the server. The associated event object is an *MessageEvent*, and the text of the server's message is available through the *data* property of that object.

onopen

Triggered when the connection opens. The associated event object is a simple *Event*.

EventTarget

an object that receives events

Objects that have events fired on them or objects to which events bubble need to have a way to define handlers for those events. These objects typically define event handler registration properties whose names begin with “on”, and they also typically define the methods described here. Event handler registration is a surprisingly complex topic. See §17.2 for details, and note, in particular, that IE8 and before use different methods, described in a special section following, than all other browsers do.

Methods

void addEventListener(string type, function listener, [boolean useCapture])

This method registers the specified *listener* function as an event handler for events of the specified *type*. *type* is an event name string and does not include an “on” prefix. The *useCapture* argument should be *true* if this is a capturing event handler (see §17.2.3) being registered on a document ancestor of the true event target. Note that some browsers still require you to pass a third argument to this function, and you must pass *false* to register an ordinary noncapturing handler.

boolean dispatchEvent(Event event)

This method dispatches a synthetic *event* to this event target. Create a new *Event* object with *document.createEvent(Events)*, passing the event name (such as “event” for simple events). Next, call the event initialization method for the *Event* object you created: for a simple event, this will be *initEvent()* (see *Event*). Next, pass the initialized event to this method to dispatch it. In modern browsers, every *Event* object has an *isTrusted* property. That property will be *false* for any synthetic event dispatched by JavaScript.

Every kind of event object defines a type-specific initialization method. Those methods are infrequently used, have long and cumbersome argument lists, and are not documented in this book. If you need to create, initialize, and dispatch synthetic events of some type more complex than a basic *Event*, you'll have to look up the initialization method online.

void removeEventListener(string type, function listener, [boolean useCapture])

This method removes a registered event *listener* function. It takes the same arguments as *addEventListener()*.

Internet Explorer Methods

IE8 and before do not support `addEventListener()` and `removeEventListener()`. Instead, they implement these two methods, which are quite similar. (§17.2.4 lists a few important differences.)

`void attachEvent(string type, function listener)`

Register the specified *listener* function as an event handler for events of the specified *type*. Note that this method expects *type* to include the prefix “on” before the event name.

`void detachEvent(string type, function listener)`

This method works like `attachEvent()` in reverse.

FieldSet

a `<fieldset>` in an HTML form

Node, Element, FormControl

The `FieldSet` object represents a `<fieldset>` in an HTML `<form>`. `FieldSets` implement most, but not all, of the properties and methods of `FormControl`.

Properties

boolean **disabled**

true if the `FieldSet` is disabled. Disabling a `FieldSet` disables the form controls it contains.

readonly `HTMLFormControlsCollection` **elements**

An array-like object of all form controls contained within this `<fieldset>`.

File

a file in the local filesystem

Blob

A `File` is a `Blob` that has a name and possibly also a modification date. It represents a file in the local file system. Obtain a user-selected file from the `files` array of an `<input type=file>` element, or from the `files` array of the `DataTransfer` object associated with the `Event` object that accompanies a drop event.

You can also obtain `File` objects that represent files in a private, sandboxed filesystem, as described in §22.7. The filesystem API is not stable at the time of this writing, however, and it is not documented in this reference section.

You can upload the contents of a file to a server with a `FormData` object or by passing the `File` to `XMLHttpRequest.send()`, but there is not much else you can do with the `File` object itself. Use `FileReader` to read the contents of a `File` (or of any `Blob`).

Properties

readonly `Date` **lastModifiedDate**

The modification date of the file, or `null` if it is not available.

readonly string name

The name of the file (but not its path).

FileError

error while reading a file

A FileError object represents an error that occurred when reading a file with FileReader or FileReaderSync. If the synchronous API is used, the FileError object is thrown. If the asynchronous API is used, the FileError object is the value of the error property of the FileReader object when the error event is dispatched.

Note that the FileWriter API (which is described in §22.7, but is not stable enough to document in this reference section) adds new error code constants to this object.

Constants

The FileError error codes are the following:

unsigned short NOT_FOUND_ERR = 1

The file does not exist. (Perhaps it was deleted after the user selected it, but before your program attempted to read it.)

unsigned short SECURITY_ERR = 2

Unspecified security issues prevent the browser from allowing your code to read the file.

unsigned short ABORT_ERR = 3

The attempt to read the file was aborted.

unsigned short NOT_READABLE_ERR = 4

The file is not readable, perhaps because its permissions have changed or because another process has locked it.

unsigned short ENCODING_ERR = 5

A call to readAsDataURL() failed because the file was too long to encode in a data:// URL.

Properties

readonly unsigned short code

This property specifies what kind of error occurred. Its value is one of the constants above.

FileReader

asynchronously read a File or Blob

EventTarget

A FileReader defines an asynchronous API for reading the content of a File or any Blob. To read a file, follow these steps:

- Create a FileReader with the FileReader() constructor.
- Define whichever event handlers you need.
- Pass your File or Blob object to one of the four read methods.

- When your `onload` handler is triggered, the file contents are available as the `result` property. Or, if the `onerror` handler is triggered, the `error` property refers to a `FileError` object that provides more information.
- When the read is complete, you can reuse the `FileReader` object or discard it and create new ones as needed.

See `FileReaderSync` for a synchronous API that you can use in worker threads.

Constructor

`new FileReader()`

Create a new `FileReader` object with the `FileReader()` constructor, which expects no arguments.

Constants

These constants are the values of the `readyState` property:

`unsigned short EMPTY = 0`

No read method has been called yet.

`unsigned short LOADING = 1`

A read is in progress.

`unsigned short DONE = 2`

A read has completed successfully or with an error.

Properties

`readonly FileError error`

If an error occurs during a read, this property will refer to a `FileError` that describes the error.

`readonly unsigned short readyState`

This property describes the current state of the `FileReader`. Its value will be one of the three constants listed above.

`readonly any result`

If the read completed successfully, this property will hold the File or Blob contents as a string or `ArrayBuffer` (depending on which read method was called). When `readyState` is `LOADING` or when a progress event is fired, this property may contain partial contents of the File or Blob. If no read method has been called or if an error has occurred, this property will be `null`.

Methods

`void abort()`

This method aborts a read. It sets `readyState` to `DONE`, sets `result` to `null`, and sets `error` to a `FileError` object with a code of `FileError.ABORT_ERR`. Then it fires an abort event and a loadend event.

void readAsArrayBuffer(Blob blob)

Asynchronously read the bytes of *blob* and make them available as an `ArrayBuffer` on the `result` property.

void readAsBinaryString(Blob blob)

Asynchronously read the bytes of *blob*, encode them as a JavaScript binary string, and set the `result` property to the resulting string. Each “character” in a JavaScript binary string has a character code between 0 and 255. Use `String.charCodeAt()` to extract these byte values. Note that binary strings are an inefficient representation of binary data: when possible you should use `ArrayBuffers` instead.

void readAsDataURL(Blob blob)

Asynchronously read the bytes of *blob*, encode them (along with the type of the `Blob`) into a `data://` URL, and set the `result` property to the resulting string.

void readAsText(Blob blob, [string encoding])

Asynchronously read the bytes of *blob* and decode them using the specified *encoding* into a string of Unicode text and then set the `result` property to that decoded string. If encoding is not specified, UTF-8 will be used (UTF-16 encoded text is also automatically detected and decoded if it begins with a Byte Order Mark).

Event Handlers

Like all asynchronous APIs, `FileReader` is event based. You can use the handler properties listed here to register event handlers, or you can use the `EventTarget` methods implemented by `FileReader`.

`FileReader` events are triggered on the `FileReader` object itself. They do not bubble and have no default action to cancel. `FileReader` event handlers are always passed a `ProgressEvent` object. A successful read begins with a `loadstart` event, followed by zero or more progress events, a `load` event, and a `loadend` event. A unsuccessful read begins with a `loadstart` event, followed by zero or more progress events, an `error` or `abort` event, and a `loadend` event.

onabort

Triggered if the read is aborted with the `abort()` method.

onerror

Triggered if an error of some sort occurs. The `error` property of the `FileReader` will refer to a `FileError` object that has an error code.

onload

Triggered when the File or Blob has been successfully read. The `result` property of the `FileReader` holds the File or Blob content, in a representation that depends on the read method that was called.

onloadend

Every call to a `FileReader` read method eventually produces a `load` event, an `error` event, or an `abort` event. The `FileReader` also triggers a `loadend` event after each of these events for the benefit of scripts that want to listen for only one event instead of listening for all three.

onloadstart

Triggered after a read method is invoked but before any data has been read.

onprogress

Triggered approximately 20 times a second while File or Blob data is being read. The `ProgressEvent` object will specify how many bytes have been read, and the `result` property of the `FileReader` may contain a representation of those bytes.

FileReaderSync

synchronously read a File or Blob

`FileReaderSync` is a synchronous version of the `FileReader` API, available only to Worker threads. The synchronous API is easier to use than the asynchronous one: simply create a `FileReaderSync()` object and then call one of its read methods, which will either return the contents of the File or Blob or throw a `FileError` object instead.

Constructor

`new FileReaderSync()`

Create a new `FileReaderSync` object with the `FileReaderSync()` constructor, which expects no arguments.

Methods

These methods throw a `FileError` object if the read fails for any reason.

`ArrayBuffer readAsArrayBuffer(Blob blob)`

Read the bytes of *blob* and return them as an `ArrayBuffer`.

`string readAsBinaryString(Blob blob)`

Read the bytes of *blob*, encode them as a JavaScript binary string (see `String.fromCharCode()`), and return that binary string.

`string readAsDataURL(Blob blob)`

Read the bytes of *blob*, and encode those bytes, along with the `type` property of *blob* into a `data://` URL, and then return that URL.

`string readAsText(Blob blob, [string encoding])`

Read the bytes of *blob*, decode them into text using the specified *encoding* (or using UTF-8 or UTF-16 if no encoding is specified), and return the resulting string.

Form

a `<form>` in an HTML document

Node, Element

The `Form` object represents a `<form>` element in an HTML document. The `elements` property is an `HTMLCollection` that provides convenient access to all elements of the form. The `submit()` and `reset()` methods allow a form to be submitted or reset under program control.

Each form in a document is represented as an element of the `document.forms[]` array. The elements of a form (buttons, input fields, checkboxes, and so on) are collected in the array-like object `Form.elements`. Named form controls can be referenced directly by name: the control name is used as a property name on the `Form` object. Thus, to refer to an `Input` element with a `name` attribute of “phone” within a form `f`, you might use the JavaScript expression `f.phone`.

See §15.9 for more on HTML forms. See `FormControl`, `FieldSet`, `Input`, `Label`, `Select`, and `TextArea` for more on the form controls that can appear in a form.

This page documents HTML5 form features which, at the time of this writing, were not yet widely implemented.

Properties

Most of the properties listed here simply mirror the HTML attributes of the same name.

`string acceptCharset`

A list of one or more allowed character sets in which the form data may be encoded for submission.

`string action`

The URL to which the form should be submitted.

`string autoComplete`

The string “on” or “off”. If “on”, the browser can prefill form controls with saved values from a previous visit to the page.

`readonly HTMLFormControlsCollection elements`

An array-like object of form controls contained by this form.

`string enctype`

Specifies the way the values of the form controls are encoded for submission. The legal values of this property are:

- “application/x-www-form-urlencoded” (the default)
- “multipart/form-data”
- “text/plain”

`readonly long length`

The number of form controls represented by the `elements` property. Form elements behave as if they themselves were array-like objects of form controls, and for a form `f` and an integer `n`, the expression `f[n]` is the same as `f.elements[n]`.

`string method`

The HTTP method used to submit the form to the action URL. Either “get” or “post”.

`string name`

The name of the form, as specified by the HTML `name` attribute. You can use the value of this property as a property name on the document object. The value of that document property will be this `Form` object.

boolean **noValidate**

true if the form is not to be validated before submission. Mirrors the HTML `novalidate` attribute.

string **target**

The name of a window or frame in which the document returned by form submission is to be displayed.

Methods

boolean **checkValidity()**

In browsers that support form validation, this method checks the validity of each form control. It returns **true** if they are all valid. If any controls are not valid, it fires an invalid event on that control and then returns **false**.

void **dispatchFormChange()**

This method triggers a `formchange` event on each control in this form. The form usually does this automatically when user input triggers a change event, so you do not normally need to call this method.

void **dispatchFormInput()**

This method triggers a `forminput` event on each control in this form. The form usually does this automatically when user input triggers an input event, so you do not normally need to call this method.

void **reset()**

Reset all form elements to their default values.

void **submit()**

Submit the form manually, without triggering an submit event.

Event Handlers

These form-related event handler properties are defined on `Element`, but they are documented in more detail here because they are triggered on `Form` elements.

onreset

Invoked just before the elements of the form are reset. Return **false** or cancel the event to prevent the reset.

onsubmit

Invoked just before the form is submitted. Return **false** or cancel the event to prevent the submission.

FormControl

common features of all form controls

Most HTML form controls are `<input>` elements, but forms can also contain `<button>`, `<select>`, and `<textarea>` controls. This page documents the features that those element types

have in common. See §15.9 for an introduction to HTML forms, and see `Form`, `Input`, `Select`, and `TextArea` for more on forms and form controls.

The `<fieldset>` and `<output>` elements implement most, but not all, of the properties described here. This reference treats `FieldSet` and `Output` objects as `FormControls` even though they do not implement every property.

This page documents certain HTML5 form features (particularly form validation) which, at the time of this writing, were not yet widely implemented.

Properties

boolean `autofocus`

`true` if the control should automatically receive keyboard focus as soon as the document is loaded. (`FieldSet` and `Output` controls do not implement this property.)

boolean `disabled`

`true` if the form control is disabled. Disabled controls do not respond to user input and are not subject to form validation. (`Output` elements do not implement this property; `FieldSet` elements use it to disable all of the controls they contain.)

readonly `Form form`

A reference to the `Form` that is the owner of this control, or `null` if it does not have one. If a control is contained within a `<form>` element, that is its form owner. Otherwise, if the control has an HTML `form` attribute that specifies the ID of a `<form>`, that named form is the form owner.

readonly `NodeList labels`

An array-like object of `Label` elements associated with this control. (`FieldSet` controls do not implement this property.)

string `name`

The value of the HTML `name` attribute for this control. A control's name can be used as a property of the `Form` element: the value of that property is the control element. Control names are also used when submitting a form.

string `type`

For `<input>` elements, the `type` property has the value of the `type` attribute, or the value "text" if no `type` attribute is specified on the `<input>` tag. For `<button>`, `<select>`, and `textarea` elements, the `type` property is "button", "select-one" (or "select-multiple", if the `multiple` attribute is set), and "textarea". For `<fieldset>` elements, the `type` is "fieldset", and for `<output>` elements the `type` is "output".

readonly string `validationMessage`

If the control is valid or is not subject to validation, this property will be the empty string. Otherwise, this property contains a localized string that explains why the user's input is invalid.

readonly `FormValidity validity`

This property refers to an object that specifies whether the user's input for this control is valid, and if not, why not.

string value

Every form control has a string value that is used when the form is submitted. For text input controls, the value is the user's input. For buttons, the value is just the value of the HTML `value` attribute. For output elements, this property is like the `textContent` property inherited from `Node`. `FieldSet` elements do not implement this property.

readonly boolean `willValidate`

This property is `true` if the control takes part in form validation, and `false` otherwise.

Event Handlers

Form controls define the following event handler properties. You can also register event handlers using the `EventTarget` methods implemented by all `Elements`:

Event Handler	Invoked when
<code>onformchange</code>	When a change event is fired on any control in the form, the form broadcasts a nonbubbling <code>formchange</code> event to all of its controls. Controls can use this handler property to detect changes to their sibling controls.
<code>onforminput</code>	When an input event is fired on any control in the form, the form broadcasts a nonbubbling <code>forminput</code> event to all of its controls. Controls can use this handler property to detect changes to their sibling controls.
<code>oninvalid</code>	If a form control does not validate, an <code>invalid</code> event will be fired on it. This event does not bubble, but if canceled, the browser will not display an error message for the control.

Methods**boolean `checkValidity()`**

Returns `true` if the control is valid (or if it is not subject to validation). Otherwise, it fires an `invalid` event at the control and returns `false`.

void `setCustomValidity(string error)`

If `error` is a nonempty string, this method marks the control as invalid and uses `error` as a localized message when reporting the element's invalidity to the user. If `error` is the empty string, any previous `error` string is removed and the control is considered valid.

FormData

an HTTP multipart/form-data request body

The `FormData` type is a feature of XMLHttpRequest Level 2 (XHR2) that makes it easy to perform HTTP PUT requests with multipart/form-data encoding using an `XMLHttpRequest`. Multipart encoding is necessary, for example, if you want to upload multiple `File` objects in a single request.

Create a `FormData` object with the constructor, and then add name/value pairs to it with the `append()` method. Once you have added all of the parts of your request body, you can pass the `FormData` to the `send()` method of an `XMLHttpRequest`.

Constructor

`new FormData()`

This no-argument constructor returns an empty `FormData` object.

Methods

`void append(string name, any value)`

This method adds a new part, with the specified *name* and *value*, to the `FormData`. The *value* argument can be a string or a `Blob` (recall that `File` objects are `Blobs`).

FormValidity

the validity of a form control

The `validity` property of a `FormControl` refers to a `FormValidity` object that is a live representation of the validity state of that control. If the `valid` property is `false`, the control is not valid, and at least one of the other properties will be `true` to indicate the nature of the validity error (or errors).

Form validation is an HTML5 feature that, at the time of this writing, is not yet widely implemented.

Properties

readonly boolean `customError`

A script called `FormControl.setCustomValidity()` on this element.

readonly boolean `patternMismatch`

The input does not match the `pattern` regular expression.

readonly boolean `rangeOverflow`

The input is too large.

readonly boolean `rangeUnderflow`

The input is too small.

readonly boolean `stepMismatch`

The input does not match the specified step.

readonly boolean `tooLong`

The input is too long.

readonly boolean `typeMismatch`

The input is of the wrong type.

readonly boolean `valid`

If this property is `true`, the form control is valid, and all the other properties are `false`.

If this property is `false`, the form control is not valid, and at least one of the other properties is `true`.

readonly boolean `valueMissing`

The form element was `required`, but no value was entered.

Geocoordinates

a geographical position

An object of this type represents a position on the surface of the earth.

Properties

readonly double **accuracy**

The accuracy of the latitude and longitude values, in meters.

readonly double **altitude**

The altitude, in meters above sea level, or `null` if altitude is not available.

readonly double **altitudeAccuracy**

The accuracy, in meters, of the altitude property. If altitude is `null`, altitude Accuracy will also be `null`.

readonly double **heading**

The user's direction of travel, in degrees clockwise from true north, or `null` if the heading is not available. If heading information is available, but **speed** is 0, heading will be NaN.

readonly double **latitude**

The user's latitude in decimal degrees north of the equator.

readonly double **longitude**

The user's longitude in decimal degrees east of the Greenwich Meridian.

readonly double **speed**

The user's speed in meters per second, or `null` if speed information is not available. This property will never be a negative number. See also **heading**.

Geolocation

obtain the user's latitude and longitude

The Geolocation object defines methods for determining the user's precise geographical location. In browsers that support it, the Geolocation object is available through the Navigator object as `navigator.geolocation`. The methods described here depend on a few other types: locations are reported in the form of a `Geoposition` object and errors are reported as `GeolocationError` objects.

Methods

void **clearWatch**(long *watchId*)

Stops watching the user's location. The *watchId* argument must be the value returned by the corresponding call to `watchPosition()`.

void **getCurrentPosition**(function *success*, [function *error*], [object *options*])

Asynchronously determines the user's location using any *options* (see the list of option properties below) that were specified. This method returns immediately, and when the user's location becomes available, it passes a `Geoposition` object to the specified *success* callback.

Or, if an error occurs (perhaps because the user did not grant permission to share her location), it passes a `GeolocationError` object to the `error` callback if one was specified.

long `watchPosition(function success, [function error], [object options])`

This method is like `getCurrentPosition()`, but after determining the user's current location, it continues to monitor the user's location and invokes `success` callback every time the position is found to have changed significantly. The return value is a number that you can pass to `clearWatch()` to stop tracking the user's location.

Options

The `options` argument to `getCurrentPosition()` and `watchPosition()` is a regular JavaScript object with zero or more of the following properties:

boolean `enableHighAccuracy`

This option is a hint that a high-accuracy position is desired, even if it would take longer to determine or would use more battery power, for example. The default is `false`. In devices that can determine position via WiFi signals or by GPS, setting this option to `true` will typically mean "use the GPS".

long `maximumAge`

This option specifies the largest acceptable age (in milliseconds) of the first `Geoposition` object passed to the `successCallback`. The default is 0, which means that each call to `getCurrentPosition()` or `watchPosition()` will have to request a new position fix. If you set this option to 60000, for example, the implementation is allowed to return any `Geoposition` determined in the last minute.

long `timeout`

This option specifies how long, in milliseconds, the requester is willing to wait for a position fix. The default value is `Infinity`. If more than `timeout` milliseconds elapse, the `errorCallback` will be invoked. Note that time spent asking the user for permission to share her location does not count against this `timeout` value.

GeolocationError

an error while querying the user's location

If an attempt to determine the user's geographical position fails, your error callback function will be invoked with a `GeolocationError` object that describes what went wrong.

Constants

These constants are the possible values of the `code` property:

unsigned short `PERMISSION_DENIED = 1`

The user did not grant permission to share her or his location.

unsigned short `POSITION_UNAVAILABLE = 2`

The location could not be determined for an unspecified reason. This could be caused by a network error, for example.

unsigned short **TIMEOUT** = 3

The location could not be determined within the time allotted (see the `timeout` option described in `Geolocation`).

Properties

readonly unsigned short **code**

This property will have one of the three values above.

readonly string **message**

A message that provides more details about the error. The message is intended to aid with debugging and is not suitable for display to end users.

Geoposition

a timestamped position report

A `Geoposition` object represents the user's geographical position at a specific time. Objects of this type have only two properties: a timestamp and a reference to a `Geocoordinates` object that holds the actual position properties.

Properties

readonly `Geocoordinates` **coords**

This property refers to a `Geocoordinates` object whose properties specify the user's latitude, longitude, etc.

readonly unsigned long **timestamp**

The time at which those coordinates were valid, in milliseconds since the epoch. You can use this value to create a `Date` object if desired.

HashChangeEvent

event object for hashchange events

Event

Browsers fire a hashchange event when the fragment identifier (the portion of a URL beginning with the hash mark #) of the document URL changes. This can happen because of a scripted change to the `hash` property of the `Location` object, or because the user used the browser's Back or Forward buttons to navigate through the browser's history. In either case, a hashchange event is triggered. The associated event object is a `HashChangeEvent`. See §22.2 for more on history management with `location.hash` and the hashchange event.

Properties

readonly string **newURL**

This property holds the new value of `location.href`. Note that this is the complete URL, not just the hash portion of it.

readonly string **oldURL**

This property holds the old value of `location.href`.

History

the browsing history of a Window

The History object represents the browsing history of a window. For privacy reasons, however, it does not allow scripted access to the actual URLs that have been visited. The methods of the History object allow scripts to move the window backward and forward through the browsing history and to add new entries to the browsing history.

Properties

readonly long **length**

This property specifies the number of entries in the browser's history list. Since there is no way to determine the index of the currently displayed document within this list, knowing the size of this list is not particularly helpful.

Methods

void back()

back() causes the window or frame to which the History object belongs to revisit the URL (if any) that was visited immediately before the current one. Calling this method has the same effect as clicking on the browser's Back button. It is also equivalent to:

```
history.go(-1);
```

void forward()

forward() causes the window or frame to which the History object belongs to revisit the URL (if any) that was visited immediately after the current one. Calling this method has the same effect as clicking on the browser's Forward button. It is also equivalent to:

```
history.go(1);
```

void go([long delta])

The **History.go()** method takes an integer argument and causes the browser to visit the URL that is the specified number of positions away in the browsing history list maintained by the History object. Positive arguments move the browser forward through the list, and negative arguments move it backward. Thus, calling **history.go(-1)** is equivalent to calling **history.back()** and produces the same effect as clicking on the Back button. With an argument of 0 or no argument at all, this method reloads the currently displayed document.

void pushState(any data, string title, [string url])

This method adds a new entry to the window's browsing history, storing a structured clone (see "Structured Clones" on page 672) of *data* as well as the specified *title* and *url*. If the user later uses the browser's history navigation mechanism to return to this saved state, a *popstate* event will be triggered on the window, and the *PopStateEvent* object will hold another clone of *data* in its *state* property.

The *title* argument provides a name for this state, and browsers may display it in their history UI. (At the time of this writing, browsers ignore this argument). If specified, the *url* argument is displayed in the location bar and gives this state a permanent state that can be bookmarked or shared with others. *url* is resolved relative to the current document location. If *url* is an

absolute URL, it must have the same origin as the current document. One common technique is to use URLs that are just fragment identifiers beginning with #.

```
void replaceState(any data, string title, [string url])
```

This method is like `pushState()`, except that instead of creating a new entry in the window's browsing history, it updates the current entry with a new state *data*, *title*, and *url*.

HTMLCollection

an element collection accessible by name or number

An HTMLCollection is a read-only array-like object of `Element` objects that also defines properties corresponding to the name and id values of the collected elements. The Document object defines HTMLCollection properties such as `forms` and `image`.

HTMLCollection objects define `item()` and `namedItem()` methods, for retrieving elements by position or name, but it is never necessary to use them: you can simply treat the HTMLCollection as a JavaScript object and access its properties and array elements. For example:

```
document.images[0]    // A numbered element of an HTMLCollection
document.forms.address // A named element of an HTMLCollection
```

Properties

readonly unsigned long **length**

The number of elements in the collection.

Methods

```
Element item(unsigned long index)
```

Returns the element at the specified *index* in the collection or `null` if *index* is out of bounds. You can also simply specify the position within array brackets instead of calling this method explicitly.

```
object namedItem(string name)
```

Returns the first element from the collection that has the specified *name* for its id or name attribute, or `null` if there is no such element. You can also place the element name within array brackets instead of calling this method explicitly.

HTMLDocument

see Document

HTML**Element**

see Element

HTMLFormControlsCollection

a array-like object of form controls

HTMLCollection

HTMLFormControlsCollection is a specialized HTMLCollection used by Form elements to represent collections of form controls. Like HTMLCollection, you can index it numerically, like an array, or treat it like an object and index it with the names or IDs of form controls. HTML forms often have multiple controls (usually radio buttons or checkboxes) that have the same value for their name attribute, and an HTMLFormControlsCollection handles this differently than an ordinary HTMLCollection would.

When you read a property of an HTMLFormControlsCollection, and the form contains more than one element that has that property as its name, the HTMLFormControlsCollection returns an array-like object of all form controls that share the name. In addition, the returned array-like object has a value property that returns the value attribute of the first checked radio button with that name. You can even set this value property to check the radio button with the corresponding value.

HTMLOptionsCollection

a collection of Option elements

HTMLCollection

HTMLOptionsCollection is a specialized HTMLCollection that represents the Option elements within a Select element. It overrides the namedItem() method to handle multiple Option elements with the same name, and it defines methods for adding and removing elements. For historical reasons, HTMLOptionsCollection defines a writable length property that you can set to truncate or extend the collection.

Properties

unsigned long length

This property returns the number of elements in the collection. Unlike the length property of a regular HTMLCollection, however, this one is not read-only. If you set it to a value smaller than its current value, the collection of Option elements is truncated, and those that are no longer in the collection are removed from the containing Select element. If you set length to a value larger than its current value, empty <option/> elements are created and added to the Select element and to the collection.

long selectedIndex

The index of the first selected Option in the collection, or -1 if no Option is selected. You can set this property to change the selected item.

Methods

void add(Element option, [any before])

Insert the *option* (which must be an <option> or <optgroup> element) into this collection (and into the Select element) at the position specified by *before*. If *before* is null, insert it at the end. If *before* is an integer index, insert it before the item that is currently at that index. If *before* is another Element, insert the *option* before that element.

Element **item**(unsigned long *index*)

HTMLOptionsCollection inherits this method from HTMLCollection. It returns the element at the specified *index* or null if *index* is out of bounds. You can also index the collection directly with square brackets instead of calling this method explicitly.

object **namedItem**(string *name*)

This method returns all Option elements in the collection that have the specified name or ID. If no elements match, it returns null. If one Option element matches, it returns that element. If more than one element matches, it returns a NodeList of those elements. Note that you can index an HTMLOptionsCollection directly, using *name* as a property name instead of calling this method explicitly.

void **remove**(long *index*)

This method removes the <option> element at the specified *index* in the collection. If invoked with no argument or with an argument that is out of bounds, it may remove the first element in the collection.

IFrame

an HTML <iframe>

Node, Element

An IFrame object represents an <iframe> element in an HTML document. If you look up an <iframe> using getElementById() or a similar query function, you'll get an IFrame object. If, however, you access the <iframe> through the frames property of the Window object, or by using the name of the <iframe> as a property of the containing window, the object you obtain is the Window object that the <iframe> represents.

Properties

readonly Document **contentDocument**

The document contained in this <iframe> element. If the document displayed in the <iframe> is from a different origin, the same-origin policy (§13.6.2) will prevent access to this document.

readonly Window **contentWindow**

The Window object of the <iframe>. (The frameElement of that Window object will be a reference back to this IFrame object.)

string **height**

The height, in CSS pixels, of the <iframe>. This property mirrors the HTML height attribute.

string **name**

The name of the <iframe>. This property mirrors the HTML name attribute, and its value can be used as the target of Link and Form objects.

readonly DOMSettableTokenList **sandbox**

This property mirrors the HTML5 sandbox attribute and allows it to be queried and set as a string or as a set of individual tokens.

The `sandbox` attribute specifies that the browser should impose additional security restrictions on untrusted content displayed in an `<iframe>`. If the `sandbox` attribute is present but empty, the `<iframe>` content will be treated as if it was from a distinct origin, will not be allowed to run scripts, will not be allowed to display forms, and will not be allowed to change the location of its containing window. The `sandbox` attribute can also be set to a space-separated list of tokens, each of which lifts one of those additional security restrictions. The valid tokens are “allow-same-origin”, “allow-scripts”, “allow-forms”, and “allow-top-navigation”.

The `sandbox` attribute is not yet widely implemented at the time of this writing. See an HTML reference for further details.

boolean `seamless`

This property mirrors the HTML `seamless` attribute. If `true`, the browser should render the content of the `<iframe>` so that it appears to be part of the containing document. This means, in part, that the browser must apply the CSS styles of the containing document to the content of the `<iframe>`.

The `seamless` attribute was introduced as part of HTML5 and is not yet widely implemented at the time of this writing.

string `src`

This property mirrors the `src` attribute of the `<iframe>`: it specifies the URL of the framed content.

string `srcdoc`

This property mirrors the `srcdoc` HTML attribute and specifies the content of the `<iframe>` as a string. The `srcdoc` attribute was recently introduced as part of HTML5 and is not yet implemented at the time of this writing.

string `width`

The width, in CSS pixels, of the `<iframe>`. This property mirrors the HTML `width` attribute.

Image

an `<img>` in an HTML document

Node, Element

An `Image` object represents an image embedded in an HTML document with an `<img>` tag. The images that appear in a document are collected in the `document.images[]` array.

The `src` property of the `Image` object is the most interesting one. When you set this property, the browser loads and displays the image specified by the new value. This allows visual effects such as image rollovers and animations. See §21.1 for examples.

You can create offscreen `Image` objects by simply creating new `<img>` elements with `document.createElement()` or with the `Image()` constructor. Note that this constructor does not have an argument to specify the image to be loaded: to load an image, simply set the `src` property of your `Image` object. To actually display the image, insert the `Image` object into the document.

Constructor

```
new Image([unsigned long width, unsigned long height])
```

You can create a new `Image` as you would create any HTML element with `document.createElement()`. For historical reasons, however, client-side JavaScript also defines the `Image()` constructor to do the same thing. If the `width` or `height` arguments are specified, they set the `width` and `height` attributes of the `<img>` tag.

Properties

In addition to the properties listed here, `Image` elements also expose the following HTML attributes as JavaScript properties: `alt`, `usemap`, `ismap`.

readonly boolean **complete**

true if no image `src` was specified or if the image has been completely downloaded.
false otherwise.

unsigned long **height**

The on-screen height at which the image is displayed, in CSS pixels. Set this to change the height of the image.

readonly unsigned long **naturalHeight**

The intrinsic height of the image.

readonly unsigned long **naturalWidth**

The intrinsic width of the image.

string **src**

The URL of the image. Setting this property causes the specified image to load. If the `Image` object has been inserted into the document, the new image will be displayed.

unsigned long **width**

The width, in CSS pixels, at which the image is actually displayed on the screen. You can set this to change the on-screen size of the image.

ImageData

an array of pixel data from a `<canvas>`

An `ImageData` object holds the red, green, blue, and alpha (transparency) components of a rectangular region of pixels. Obtain an `ImageData` object with the `createImageData()` or `getImageData()` methods of the `CanvasRenderingContext2D` object of a `<canvas>` tag.

The `width` and `height` properties specify the dimensions of the rectangle of pixels. The `data` property is an array that holds the pixel data. Pixels appear in the `data[]` array in left-to-right and top-to-bottom order. Each pixel consists of four byte values that represent the R, G, B, and A components, in that order. Thus, the color components for a pixel at (x,y) within an `ImageData` object `image` can be accessed like this:

```
var offset = (x + y*image.width) * 4;
var red = image.data[offset];
var green = image.data[offset+1];
```

```
var blue = image.data[offset+2];
var alpha = image.data[offset+3];
```

The `data[]` array is not a true JavaScript array, but an optimized array-like object whose elements are integers between 0 and 255. The elements are read/write, but the length of the array is fixed. For any `ImageData` object `i`, `i.data.length` will always equal `i.width * i.height * 4`.

Properties

readonly byte[] data

A read-only reference to a read/write array-like object whose elements are bytes.

readonly unsigned long height

The number of rows of image data.

readonly unsigned long width

The number of pixels per row of data.

Input

an HTML `<input>` element

Node, Element, FormControl

An `Input` object represents an HTML form `<input>` element. Its appearance and behavior depends on its `type` attribute: an `Input` element might represent a simple text input field, a checkbox, a radio box, a button, or a file selection element, for example. Because an `<input>` element can represent so many kinds of form controls, the `Input` element is one of the most complicated. See §15.9 for an overview of HTML forms and form elements. Note that some of the important properties of the `Input` element (such as `type`, `value`, `name`, and `form`) are documented in `FormControl`.

Properties

In addition to the properties listed here, `Input` elements also implement all of the properties defined by `Element` and `FormControl`. The properties marked with an asterisk in this list are newly defined by HTML5 and are not yet, at the time of this writing, widely implemented.

string accept

When `type` is “file”, this property is a comma-separated list of MIME types that specify the types of files that may be selected. The strings “audio/*”, “video/*”, and “image/*” are also legal. Mirrors the `accept` attribute.

string autocomplete

True if the browser can prefill this `Input` element with a value from a previous session. Mirrors that `autocomplete` attribute. See also the `autocomplete` property of `Form`.

boolean checked

For checkable input elements, this property specifies whether the element is “checked” or not. Setting this property changes the visual appearance of the input element.

boolean `defaultChecked`

For checkable input elements, this property specifies the initial checkedness of the element. When the form is reset, the `checked` property is restored to the value of this property. Mirrors the `checked` attribute.

string `defaultValue`

For elements with a textual value, this property holds the initial value displayed by the element. When the form is reset, the element is restored to this value. Mirrors the `value` attribute.

readonly File[] `files`

For elements whose `type` is “file”, this property is an array-like object of the `File` object or objects that the user selected.

string `formAction*`

For submit button elements, this property specifies a value that overrides the `action` property of the containing form. Mirrors the `formaction` attribute.

string `formEnctype*`

For submit button elements, this property specifies a value that overrides the `enctype` property of the containing form. Mirrors the `formenctype` attribute.

string `formMethod*`

For submit button elements, this property specifies a value that overrides the `method` property of the containing form. Mirrors the `formmethod` attribute.

boolean `formNoValidate*`

For submit button elements, this property specifies a value that overrides the `noValidate` property of the containing form. Mirrors the `formnovalidate` attribute.

string `formTarget*`

For submit button elements, this property specifies a value that overrides the `target` property of the containing form. Mirrors the `formtarget` attribute.

boolean `indeterminate`

For checkboxes, this property specifies whether the element is in an indeterminate (neither checked nor unchecked) state. This property does *not* mirror an HTML attribute; you can only set it with JavaScript.

readonly Element `list*`

A `<datalist>` element that contains `<option>` elements that a browser can use as suggestions or autocompletion values.

string `max*`

A maximum valid value for this Input element.

long `maxLength`

If `type` is “text” or “password”, this property specifies the maximum number of characters that the user is allowed to enter. Note that this is not the same as the `size` property. Mirrors the `maxlength` attribute.

string `min*`

A maximum valid value for this Input element.

Input

boolean multiple*

true if the input element should accept more than one value of the specified type. Mirrors the `multiple` attribute.

string pattern*

The text of a regular expression that the input must match in order to be considered valid. This property uses JavaScript regular expression syntax (without the leading and trailing slashes), but note that the property is a string, not a `RegExp` object. Also note that in order to be considered valid, the entire string of input must match the pattern, not just a substring. (This is as if the pattern begins with `^` and ends with `$`.) This property mirrors the `pattern` attribute.

string placeholder

A short string of text that will appear within the Input element as a prompt to the user. When the user focuses the element, the placeholder text will vanish and an insertion cursor will appear. This property mirrors the `placeholder` attribute.

boolean readOnly

If true, this Input element is not editable. Mirrors the `readonly` attribute.

boolean required*

If true, the containing form will not be considered valid if the user does not enter a value in this Input element. Mirrors the `required` attribute.

readonly Option selectedOption*

If the `list` property is defined and `multiple` is false, this property returns the selected Option element child of the `list`, if there is one.

unsigned long selectionEnd

Returns or sets the index of the first input character after the selected text. See also `setSelectionRange()`.

unsigned long selectionStart

Returns or sets the index of the first selected character in the `<textarea>`. See also `setSelectionRange()`.

unsigned long size

For Input elements that allow text input, this property specifies the width of the element in characters. Mirrors the `size` attribute. Contrast with `maxLength`.

string step*

For numeric input types (including date and time input), this property specifies the granularity or step size of the allowed input values. This property can be the string "any" or a floating-point number. Mirrors the `step` attribute.

Date valueAsDate*

Returns the element's value (see `FormControl`) as a `Date` object.

double valueAsNumber*

Returns the element's value (see `FormControl`) as a number.

Methods

In addition to the methods listed here, Input elements also implement all of the methods defined by `Element` and `FormControl`. The methods marked with an asterisk in this list are newly defined by HTML5 and are not yet, at the time of this writing, widely implemented.

void `select()`

This method selects all the text displayed by this Input element. In most browsers, this means that the text is highlighted and that new text entered by the user replaces the highlighted text instead of being appended to it.

void `setSelectionRange(unsigned long start, unsigned long end)`

This method selects text displayed in this Input element, starting with the character at position *start* and continuing up to (but not including) the character at *end*.

void `stepDown([long n])`*

For elements that support the `step` property, decrease the current value by *n* steps.

void `stepUp([long n])`*

For elements that support the `step` property, increase the current value by *n* steps.

jQuery

jQuery 1.4

the jQuery library

Description

This is a quick reference for the jQuery library. See Chapter 19 for complete details on the library and for examples of its use. This reference page is organized and formatted somewhat differently than the other pages in this reference section. It uses the following conventions in the method signatures. Arguments named *sel* are jQuery selectors. Arguments named *idx* are integer indexes. Arguments named *elt* or *elts* are document elements or array-like objects of document elements. Arguments named *f* are callback functions and nested parentheses are used to indicate the arguments that jQuery will pass to the function you supply. Square brackets indicate optional arguments. If an optional argument is followed by an equals sign and a value, that value will be used when the argument is omitted. The return value of a function or a method follows the close parenthesis and a colon. Methods with no return value specified return the jQuery object on which they are invoked.

jQuery Factory Function

The `jQuery` function is a namespace for a variety of utility functions, but it is also the factory function for creating jQuery objects. `jQuery()` can be invoked in all of the ways shown below, but it always returns a jQuery object that represents a collection of document elements (or the Document object itself). The symbol `$` is an alias for `jQuery`, and you can use `$()` instead of `jQuery()` in each of the forms following:

jQuery(sel [, context=document])

Returns a new jQuery object that represents the document elements that are descendants of *context* and match the selector string *sel*.

jQuery(elts)

Returns a new jQuery object that represents the specified elements. *elts* may be a single document element or an array or array-like object (such as a NodeList or another jQuery object) of document elements.

jQuery(html, [props])

Parses *html* as a string of HTML-formatted text and returns a new jQuery object that contains the one or more top-level elements in the string. If *html* describes a single HTML tag, *props* may be an object that specifies HTML attributes and event handlers for the newly created element.

jQuery(f)

Registers *f* as a function to be invoked when the document has loaded and is ready to be manipulated. If the document is already ready, *f* is invoked immediately as a method of the document object. Returns a jQuery object that contains only the document object.

jQuery Selector Grammar

The jQuery selector grammar is very similar to the CSS3 selector grammar, and it is explained in detail in §19.8.1. The following is a summary:

Simple tag, class and ID selectors

*	tagname	.classname	#id
---	---------	------------	-----

Selector Combinations

A B	B as a descendant of A
A > B	B as a child of A
A + B	B as a sibling following A
A ~ B	B as a sibling of A

Attribute Filters

[attr]	has attribute
[attr=val]	has attribute with value val
[attr!=val]	does not have attribute with value val
[attr^=val]	attribute begins with val
[attr\$=val]	attribute ends with val
[attr*=val]	attribute includes val
[attr~=val]	attribute includes val as a word
[attr =val]	attribute begins with val and optional hyphen

Element Type Filters

:button	:header	:password	:submit
:checkbox	:image	:radio	:text
:file	:input	:reset	

Element State Filters

<code>:animated</code>	<code>:disabled</code>	<code>:hidden</code>	<code>:visible</code>
<code>:checked</code>	<code>:enabled</code>	<code>:selected</code>	

Selection Position Filters

<code>:eq(n)</code>	<code>:first</code>	<code>:last</code>	<code>:nth(n)</code>
<code>:even</code>	<code>:gt(n)</code>	<code>:lt(n)</code>	<code>:odd</code>

Document Position Filters

<code>:first-child</code>	<code>:nth-child(n)</code>
<code>:last-child</code>	<code>:nth-child(even)</code>
<code>:only-child</code>	<code>:nth-child(odd)</code>
	<code>:nth-child(xn+y)</code>

Miscellaneous Filters

<code>:contains(text)</code>	<code>:not(selector)</code>
<code>:empty</code>	<code>:parent</code>
<code>:has(selector)</code>	

Basic jQuery Methods and Properties

These are the basic methods and properties of jQuery objects. They don't alter the selection or the selected elements in any way, but they allow you to query and iterate over the set of selected elements. See §19.1.2 for details.

context

The context, or root element, under which the selection was made. This is the second argument to `$()` or the Document object.

each(*f*(*idx*,*elt*))

Invoke *f* once as a method of each selected element. Stops iterating if the function returns false. Returns the jQuery object on which it was invoked.

get(*idx*):elt**get():array**

Return the selected element at the specified index in the jQuery object. You can also use regular square-bracket array indexing. With no arguments, `get()` is a synonym for `toArray()`.

index():int**index(*sel*):int****index(*elt*):int**

With no argument, return the index of the first selected element among its siblings. With a selector argument, return the index of the first selected element within the set of elements that match the selector *sel*, or -1 if it is not found. With an element argument, return the index of *elt* in the selected elements, or -1 if it is not found.

is(*sel*):boolean

Return true if at least one of the selected elements also matches *sel*.

length

The number of selected elements.

map(*f*(*idx*,*elt*)):jQueryInvoke *f* once as a method of each selected element and return a new jQuery object that holds the returned values, with *null* and *undefined* values omitted and array values flattened.**selector**The selector string originally passed to `$()`.**size():int**Return the value of the **length** property.**toArray():array**

Return a true array of the selected elements.

jQuery Selection Methods

The methods described in this section alter the set of selected elements, by filtering them, adding new elements, or using the selected elements as starting points for new selections. In jQuery 1.4 and later, jQuery selections are always sorted in document order and do not contain duplicates. See §19.8.2.

add(*sel*, [*context*])**add(*elts*)****add(*html*)**The arguments to `add()` are passed to `$()`, and the resulting selection is merged with the current selection.**andSelf()**

Add the previously selected set of elements (from the stack) to the selection.

children([*sel*])

Select children of the selected elements. With no argument, select all children. With a selector, select only matching children.

closest(*sel*, [*context*])Select the closest ancestor of each selected element that matches *sel* and is a descendant of *context*. If *context* is omitted, the context property of the jQuery object is used.**contents()**

Select all children of each selected element, including text nodes and comments.

end()

Pop the internal stack restoring the selection to the state it was in before the last selection-altering method.

eq(*idx*)

Select only the selected element with the specified index. In jQuery 1.4, negative indexes count from the end.

`filter(sel)`

`filter(elts)`

`filter(f(idx):boolean)`

Filter the selection so it only includes elements that also match the selector *sel*, that are included in the array-like object *elts*, or for which the predicate *f* returns *true* when invoked as a method of the element.

`find(sel)`

Select all descendants of any selected element that match *sel*.

`first()`

Select only the first selected element.

`has(sel)`

`has(elt)`

Filter the selection to include only those selected elements that have a descendant that matches *sel* or that are ancestors of *elt*.

`last()`

Select only the last selected element.

`next([sel])`

Select the next sibling of each selected element. If *sel* is specified, exclude those that do not match.

`nextAll([sel])`

Select all of the siblings following each selected element. If *sel* is specified, exclude those that do not match.

`nextUntil(sel)`

Select the siblings following each selected element up to (but not including) the first sibling that matches *sel*.

`not(sel)`

`not(elts)`

`not(f(idx):boolean)`

This is the opposite of `filter()`. It filters the selection to exclude elements that match *sel*, that are included in *elts*, or for which *f* returns *true*. *elts* may be a single element or an array-like object of elements. *f* is invoked as a method of each selected element.

`offsetParent()`

Select the nearest positioned ancestor of each selected element.

`parent([sel])`

Select the parent of each selected element. If *sel* is specified, exclude any that do not match.

`parents([sel])`

Select the ancestors of each selected element. If *sel* is specified, exclude any that do not match.

parentsUntil(*sel*)

Select the ancestors of each selected element up to (but not including) the first one that matches *sel*.

prev([*sel*])

Select the previous sibling of each selected element. If *sel* is specified, exclude those that do not match.

prevAll([*sel*])

Select all of the siblings before each selected element. If *sel* is specified, exclude those that do not match.

prevUntil(*sel*)

Select the siblings preceding each selected element up to (but not including) the first sibling that matches *sel*.

pushStack(*elts*)

Push the current state of the selection so that it can be restored with *end()*, and then select the elements in the *elts* array (or array-like object).

siblings([*sel*])

Select the siblings of each selected element, excluding the element itself. If *sel* is specified, exclude any siblings that do not match.

slice(*startidx*, [*endidx*])

Filter the selection to include only elements with an index greater than or equal to *startidx* and less than (but not equal to) *endidx*. Negative indexes count backward from the end of the selection. If *endidx* is omitted, the *length* property is used.

jQuery Element Methods

The methods described here query and set the HTML attributes and CSS style properties of elements. Setter callback functions with an argument named *current* are passed the current value of whatever it is they are computing a new value for. See §19.2.

addClass(*names*)**addClass(*f(idx, current)*):names)**

Add the specified CSS class name or names to the *class* attribute of each selected element. Or invoke *f* as a method of each element to compute the class name or names to add.

attr(*name*):value**attr(*name*, *value*)****attr(*name*, *f(idx, current)*):value)****attr(*obj*)**

With one string argument, return the value of the named attribute for the first selected element. With two arguments, set the named attribute of all selected elements to the specified *value* or invoke *f* as a method of each element to compute a value. With a single object argument, use property names as attribute names and property values as attribute values or attribute computing functions.

```
css(name):value
css(name, value)
css(name, f(idx,current):value)
css(obj)
```

Like `attr()`, but query or set CSS style attributes instead of HTML attributes.

```
data():obj
data(key):value
data(key, value)
data(obj)
```

With no arguments, return the data object for the first selected element. With one string argument, return the value of the named property of that data object. With two arguments, set the named property of the data object of all selected elements to the specified *value*. With one object argument, replace the data object of all selected elements.

```
hasClass(name):boolean
```

Returns true if any of the selected elements includes *name* in its `class` attribute.

```
height():int
height(h)
height(f(idx,current):int)
```

Return the height (not including padding, border, or margin) of the first selected element, or set the height of all selected elements to *h* or to the value computed by invoking *f* as a method of each element.

```
innerHeight():int
```

Return the height plus padding of the first selected element.

```
innerWidth():int
```

Return the width plus padding of the first selected element.

```
offset():coords
offset(coords)
offset(f(idx,current):coords)
```

Return the X and Y position (in document coordinates) of the first selected element, or set the position of all selected elements to *coords* or to the value computed by invoking *f* as a method of each element. Coordinates are specified as objects with `top` and `left` properties.

```
offsetParent():jQuery
```

Select the nearest positioned ancestor of each selected element and return them in a new jQuery object.

```
outerHeight([margins=false]):int
```

Return the height plus the padding and border, and, if *margins* is true, the margins of the first selected element.

```
outerWidth([margins=false]):int
```

Return the width plus the padding and border, and, if *margins* is true, the margins of the first selected element.

`position():coords`

Return the position of the first selected element relative to its nearest positioned ancestor. The return value is an object with `top` and `left` properties.

`removeAttr(name)`

Remove the named attribute from all selected elements.

`removeClass(names)`

`removeClass(f(idx,current)):names`

Remove the specified name or names from the `class` attribute of all selected elements. If a function is passed instead of a string, invoke it as a method of each element to compute the name or names to be removed.

`removeData([key])`

Remove the named property from the data object of each selected element. If no property name is specified, remove the entire data object instead.

`scrollLeft():int`

`scrollLeft(int)`

Return the horizontal scrollbar position of the first selected element or set it for all selected elements.

`scrollTop():int`

`scrollTop(int)`

Return the vertical scrollbar position of the first selected element or set it for all selected elements.

`toggleClass(names, [add])`

`toggleClass(f(idx,current)):names, [add]`

Toggle the specified class name or names in the `class` property of each selected element. If `f` is specified, invoke it as a method of each selected element to compute the name or names to be toggled. If `add` is `true` or `false`, add or remove the class names rather than toggling them.

`val():value`

`val(value)`

`val(f(idx,current)):value`

Return the form value or selection state of the first selected element, or set the value or selection state of all selected elements to `value` or to the value computed by invoking `f` as a method of each element.

`width():int`

`width(w)`

`width(f(idx,current)):int`

Return the width (not including padding, border, or margin) of the first selected element, or set the width of all selected elements to `w` or to the value computed by invoking `f` as a method of each element.

jQuery Insertion and Deletion Methods

The methods described here insert, delete, and replace document content. In the method signatures below, the *content* argument may be a jQuery object, a string of HTML, or an individual document element, and the *target* argument may be a jQuery object, an individual document element, or a selector string. See §19.2.5 and §19.3 for further details.

after(*content*)

after(*f*(*idx*):*content*)

Insert *content* after each selected element, or invoke *f* as a method of, and insert its return value after, each selected element.

append(*content*)

append(*f*(*idx*,*html*):*content*)

Append *content* to each selected element, or invoke *f* as a method of, and append its return value to, each selected element.

appendTo(*target*):jQuery

Append the selected elements to the end of each specified *target* element, cloning them as necessary if there is more than one target.

before(*content*)

before(*f*(*idx*):*content*)

Like *after()*, but make insertions before the selected elements instead of after them.

clone([*data*=false]):jQuery

Make a deep copy of each of the selected elements and return a new jQuery object representing the cloned elements. If *data* is *true*, also clone the data (including event handlers) associated with the selected elements.

detach([*sel*])

Like *remove()*, but does not delete any data associated with the detached elements.

empty()

Delete the content of all selected elements.

html():string

html(*htmlText*)

html(*f*(*idx*,*current*):*htmlText*)

With no arguments, return the content of the first selected element as an HTML-formatted string. With one argument, set the content of all selected elements to the specified *htmlText* or to the value returned by invoking *f* as a method of those elements.

insertAfter(*target*):jQuery

Insert the selected elements after each *target* element, cloning them as necessary if there is more than one target.

insertBefore(*target*):jQuery

Insert the selected elements before each *target* element, cloning them as necessary if there is more than one target.

`prepend(content)`

`prepend(f(idx,html):content)`

Like `append()`, but insert content at the beginning of each selected element instead of at the end.

`prependTo(target):jQuery`

Like `appendTo()`, except that the selected elements are inserted at the beginning of the target elements instead of at the end.

`remove([sel])`

Remove all selected elements or all selected elements that also match *sel*, from the document, removing any data (including event handlers) associated with them. Note that the removed elements are no longer part of the document, but are still members of the returned jQuery object.

`replaceAll(target)`

Insert the selected elements into the document so that they replace each *target* element, cloning the selected elements as needed if there is more than one target.

`replaceWith(content)`

`replaceWith(f(idx,html):content)`

Replace each selected element with *content*, or invoke *f* as a method of each selected element, passing the element index and current HTML content, and then replace that element with the return value.

`text():string`

`text(plainText)`

`text(f(idx,current):plainText)`

With no arguments, return the content of the first selected element as a plain-text string. With one argument, set the content of all selected elements to the specified *plainText* or to the value returned by invoking *f* as a method of those elements.

`unwrap()`

Remove the parent of each selected element, replacing it with the selected element and its siblings.

`wrap(wrapper)`

`wrap(f(idx):wrapper)`

Wrap *wrapper* around each selected element, cloning as needed if there is more than one selected element. If a function is passed, invoke it as a method of each selected element to compute the wrapper. The *wrapper* may be an element, a jQuery object, a selector, or a string of HTML, but it must have a single innermost element.

`wrapAll(wrapper)`

Wrap *wrapper* around the selected elements as a group by inserting *wrapper* at the location of the first selected element and then copying all selected elements into the innermost element of *wrapper*.

`wrapInner(wrapper)`

`wrapInner(f(idx):wrapper)`

Like `wrap()`, but inserts *wrapper* (or the return value of *f*) around the content of each selected element rather than around the elements themselves.

jQuery Event Methods

The methods in this section are for registering event handlers and triggering events. See §19.4.

`event-type()`

`event-type(f(event))`

Register *f* as a handler for *event-type*, or trigger an event of *event-type*. jQuery defines the following convenience methods that follow this pattern:

<code>ajaxComplete()</code>	<code>blur()</code>	<code>focusin()</code>	<code>mousedown()</code>	<code>mouseup()</code>
<code>ajaxError()</code>	<code>change()</code>	<code>focusout()</code>	<code>mouseenter()</code>	<code>resize()</code>
<code>ajaxSend()</code>	<code>click()</code>	<code>keydown()</code>	<code>mouseleave()</code>	<code>scroll()</code>
<code>ajaxStart()</code>	<code>dblclick()</code>	<code>keypress()</code>	<code>mousemove()</code>	<code>select()</code>
<code>ajaxStop()</code>	<code>error()</code>	<code>keyup()</code>	<code>mouseout()</code>	<code>submit()</code>
<code>ajaxSuccess()</code>	<code>focus()</code>	<code>load()</code>	<code>mouseover()</code>	<code>unload()</code>

`bind(type, [data], f(event))`

`bind(events)`

Register *f* as a handler for events of the specified *type* on each of the selected elements. If *data* is specified, add it to the event object before invoking *f*. *type* may specify multiple event types and may include namespaces.

If a single object is passed, treat it as a mapping of event types to handler functions, and register handlers for all the specified events on each selected element.

`delegate(sel, type, [data], f(event))`

Register *f* as a live event handler. *f* will be triggered when events of type *type* occur on an element matching *sel* and bubble up to any of the selected elements. If *data* is specified, it will be added to the event object before *f* is invoked.

`die(type, [f(event)])`

Deregister live event handlers registered with `live()` for events of type *type* on elements that match the selector string of the current selection. If a specific event handler function *f* is specified, only deregister that one.

`hover(f(event))`

`hover(enter(event), leave(event))`

Register event handlers for “mouseenter” and “mouseleave” events on all selected elements. If only one function is specified, it is used as the handler for both events.

`live(type, [data], f(event))`

Register *f* as a live event handler for events of type *type*. If *data* is specified, add it to the event object before invoking *f*. This method does not use the set of selected elements, but it does use the selector string and context object of the jQuery object. *f* will be triggered when *type* events bubble up to the context object (usually the document) and the event’s target element matches the selector. See `delegate()`.

`one(type, [data], f(event))`

`one(events)`

Like `bind()`, except that the registered event handlers are automatically deregistered after they are invoked once.

`ready(f())`

Register `f` to be invoked when the document becomes ready, or invoke it immediately if the document is ready. This method does not use the selected elements and is a synonym for `$(f)`.

`toggle(f1(event), f2(event), ...)`

Register an “click” event handler on all selected elements that alternates (or toggles) among the specified handler functions.

`trigger(type, [params])`

`trigger(event)`

Trigger a `type` event on all selected elements, passing `params` as extra parameters to event handlers. `params` may be omitted, or may be a single value or an array of values. If you pass an `event` object, its `type` property specifies the event type, and any other properties are copied into the event object that is passed to the handlers.

`triggerHandler(type, [params])`

Like `trigger()`, but do not allow the triggered event to bubble or to trigger the browser’s default action.

`unbind([type], [f(event)])`

With no arguments, deregister all jQuery event handlers on all selected elements. With one argument, deregister all event handlers for the `type` events on all selected elements. With two arguments, deregister `f` as a handler for `type` events on all selected elements. `type` may name multiple event types and may include namespaces.

`undelegate()`

`undelegate(sel, type, [f(event)])`

With no arguments, deregister all live event handlers delegated from the selected elements. With two arguments, deregister live event handlers for `type` events on elements matching `sel` that are delegated from the selected elements. With three arguments, only deregister the single handler `f`.

jQuery Effects and Animation Methods

The methods described here produce visual effects and custom animations. Most return the jQuery object on which they are called. See §19.5.

Animation options

complete duration easing queue specialEasing step

`jQuery.fx.off`

Set this property to `true` to disable all effects and animations.

animate(*props*, *opts*)

Animate the CSS properties specified by the *props* object on each selected element, using the options specified by *opts*. See §19.5.2 for details of both objects.

animate(*props*, [*duration*], [*easing*], [*f*()])

Animate the CSS properties specified by *props* on each selected element, using the specified *duration* and *easing* function. Invoke *f* as a method of each selected element when done.

clearQueue([*qname*="fx"])

Clear the effects queue or the named queue for each selected element.

delay(*duration*, [*qname*="fx"])

Add a delay of the specified duration to the effects queue or the named queue.

dequeue([*qname*="fx"])

Remove and invoke the next function on the effects queue or the named queue. It is not normally necessary to dequeue the effects queue.

fadeIn([*duration*=400], [*f*()])**fadeOut([*duration*=400], [*f*()])**

Fade the selected elements in or out by animating their opacity for *duration* ms. When complete, invoke *f*, if specified, as a method of each selected element.

fadeTo(*duration*, *opacity*, [*f*()])

Animate the CSS opacity of the selected elements to *opacity* over the specified *duration*. When complete, invoke *f*, if specified, as a method of each selected element.

hide()**hide(*duration*, [*f*()])**

With no arguments, hide each selected element immediately. Otherwise, animate the size and opacity of each selected element so that they are hidden after *duration* ms. When complete, invoke *f*, if specified, as a method of each selected element.

slideDown([*duration*=400], [*f*()])**slideUp([*duration*=400], [*f*()])****slideToggle([*duration*=400], [*f*()])**

Show, hide, or toggle the visibility of each selected element by animating its height for the specified *duration*. When complete, invoke *f*, if specified, as a method of each selected element.

show()**show(*duration*, [*f*()])**

With no arguments, show each selected element immediately. Otherwise, animate the size and opacity of each selected element so that they are fully visible after *duration* ms. When complete, invoke *f*, if specified, as a method of each selected element.

stop([*clear*=false], [*jump*=false])

Stop the current animation (if one is running) on all selected elements. If *clear* is true, also clear the effects queue for each element. If *jump* is true, jump the animation to its final value before stopping it.

`toggle([show])`

`toggle(duration, [f()])`

If *show* is true, `show()` the selected elements immediately. If *show* is false, `hide()` the selected elements immediately. If *show* is omitted, toggle the visibility of the elements.

If *duration* is specified, toggle the visibility of the selected elements with a size and opacity animation of the specified length. When complete, invoke *f*, if specified, as a method of each selected element.

`queue([qname="fx"]):array`

`queue([qname="fx"], f(next))`

`queue([qname="fx"], newq)`

With no arguments or just a queue name, return the named queue of the first selected element. With a function argument, add *f* to the named queue of all selected elements.

With an array argument, replace the named queue of all selected elements with the *newq* array of functions.

jQuery Ajax Functions

Most of the jQuery Ajax-related functionality takes the form of utility functions rather than methods. These are some of the most complicated functions in the jQuery library. See §19.6 for complete details.

Ajax status codes

success error notmodified timeout parsererror

Ajax Data Types

text html xml script json jsonp

Ajax Events

ajaxStart ajaxSend ajaxSuccess ajaxError ajaxComplete ajaxStop

Ajax Options

async	context	global	processData	type
beforeSend	data	ifModified	scriptCharset	url
cache	dataFilter	jsonp	success	username
complete	dataType	jsonpCallback	timeout	xhr
contentType	error	password	traditional	

jQuery.ajax(options):XMLHttpRequest

This is the complicated but fully general Ajax function on which all of jQuery's Ajax utilities are based. It expects a single object argument whose properties specify all details of the Ajax request and the handling of the server's response. The most common options are described in §19.6.3.1 and callback options are covered in §19.6.3.2.

jQuery.ajaxSetup(options)

This function sets default values for jQuery's Ajax options. Pass the same kind of options object you would pass to `jQuery.ajax()`. The values you specify will be used by any subsequent Ajax request that does not specify the value itself. This function has no return value.

jQuery.getJSON(*url*, [*data*], [*f(object,status)*]):XMLHttpRequest

Asynchronously request the specified *url*, adding any *data* that is specified. When the response is received, parse it as JSON, and pass the resulting object to the callback function *f*. Return the XMLHttpRequest object, if any, used for the request.

jQuery.getScript(*url*, [*f(text,status)*]):XMLHttpRequest

Asynchronously request the specified *url*. When the response arrives, execute it as a script, and then pass the response text to *f*. Return the XMLHttpRequest object, if any, used for the request. Cross-domain is allowed, but do not pass the script text to *f*, and do not return an XMLHttpRequest object.

jQuery.get(*url*, [*data*], [*f(data,status,xhr)*], [*type*]):XMLHttpRequest

Make an asynchronous HTTP GET request for *url*, adding *data*, if any, to the query parameter portion of that URL. When the response arrives, interpret it as data of the specified *type*, or according to the Content-Type header of the response, and execute it or parse it if necessary. Finally, pass the (possibly parsed) response data to the callback *f* along with the jQuery status code and the XMLHttpRequest object used for the request. That XMLHttpRequest object, if any, is also the return value of `jQuery.get()`.

jQuery.post(*url*, [*data*], [*f(data,status,xhr)*], [*type*]):XMLHttpRequest

Like `jQuery.get()`, but make an HTTP POST request instead of a GET request.

jQuery.param(*o*, [*old=false*]):string

Serialize the names and values of the properties of *o* in `www-form-urlencoded` form, suitable for adding to a URL or passing as the body of an HTTP POST request. Most jQuery Ajax functions will do this automatically for you if you pass an object as the *data* parameter. Pass `true` as the second argument if you want jQuery 1.3–style shallow serialization.

jQuery.parseJSON(*text*):object

Parse JSON-formatted *text* and return the resulting object. jQuery's Ajax functions use this function internally when you request JSON-encoded data.

load(*url*, [*data*], [*f(text,status,xhr)*])

Asynchronously request the *url*, adding any *data* that is specified. When the response arrives, interpret it as a string of HTML and insert it into each selected element, replacing any existing content. Finally, invoke *f* as a method of each selected element, passing the response text, the jQuery status code, and the XMLHttpRequest object used for the request.

If *url* includes a space, any text after the space is used as a selector, and only the portions of the response document that match that selector are inserted into the selected elements.

Unlike most jQuery Ajax utilities, `load()` is a method, not a function. Like most jQuery methods, it returns the jQuery object on which it was invoked.

serialize():string

Serialize the names and values of the selected forms and form elements, returning a string in `www-form-urlencoded` format.

jQuery Utility Functions

These are miscellaneous jQuery functions and properties (not methods). See §19.7 for more details.

`jQuery.boxModel`

A deprecated synonym for `jQuery.support.boxModel`.

`jQuery.browser`

This property refers to an object that identifies the browser vendor and version. The object has the property `msie` for Internet Explorer, `mozilla` for Firefox, `webkit` for Safari and Chrome, and `opera` for Opera. The `version` property is the browser version number.

`jQuery.contains(a,b):boolean`

Returns true if document element *a* contains element *b*.

`jQuery.data(elt):data`

`jQuery.data(elt, key):value`

`jQuery.data(elt, data)`

`jQuery.data(elt, key, value)`

A low-level version of the `data()` method. With one element argument, return the data object for that element. With an element and a string, return the named value from that element's data object. With an element and an object, set the data object for the element. With an element, string, and value, set the named value in the element's data object.

`jQuery.dequeue(elt, [qname="fx"])`

Remove and invoke the first function in the named queue of the specified element. Same as `$(elt).dequeue(qname)`.

`jQuery.each(o, f(name,value)):o`

`jQuery.each(a, f(index,value)):a`

Invoke *f* once for each property of *o*, passing the name and value of the property and invoking *f* as a method of the value. If the first argument is an array, or array-like object, invoke *f* as a method of each element in the array, passing the array index and element value as arguments. Iteration stops if *f* returns `false`. This function returns its first argument.

`jQuery.error(msg)`

Throw an exception containing *msg*. You can call this function from plug-ins or override (e.g. `jQuery.error = alert`) it when debugging.

`jQuery.extend(obj):object`

`jQuery.extend([deep=false], target, obj...):object`

With one argument, copy the properties of *obj* into the global `jQuery` namespace. With two or more arguments, copy the properties of the second and subsequent objects, in order, into the *target* object. If the optional *deep* argument is true, a deep copy is done and properties are copied recursively. The return value is the object that was extended.

`jQuery.globalEval(code):void`

Execute the specified JavaScript *code* as if it were a top-level `<script>`. No return value.

`jQuery.grep(a, f(elt,idx):boolean, [invert=false]):array`

Return a new array that contains only the elements of *a* for which *f* returns true. Or, if *invert* is true, return only those elements for which *f* returns false.

`jQuery.inArray(v, a):integer`

Search the array or array-like object *a* for an element *v*, and return the index at which it is found or -1.

`jQuery.isArray(x):boolean`

Return true only if *x* is a true JavaScript array.

`jQuery.isEmptyObject(x):boolean`

Return true only if *x* has no enumerable properties.

`jQuery.isFunction(x):boolean`

Return true only if *x* is a JavaScript function.

`jQuery.isPlainObject(x):boolean`

Return true only if *x* is a plain JavaScript object, such as one created by an object literal.

`jQuery.isXMLDoc(x):true`

Return true only if *x* is an XML document or an element of an XML document.

`jQuery.makeArray(a):array`

Return a new JavaScript array that contains the same elements as the array-like object *a*.

`jQuery.map(a, f(elt, idx)):array`

Return a new array that contains the values returned by *f* when invoked for each element in the array (or array-like object) *a*. Return values of null are ignored and returned arrays are flattened.

`jQuery.merge(a,b):array`

Append the elements of the array *b* to *a*, and return *a*. The arguments may be array-like objects or true arrays.

`jQuery.noConflict([radical=false])`

Restore the symbol *\$* to its value before the jQuery library was loaded and return jQuery. If *radical* is true, also restore the value of the jQuery symbol.

`jQuery.proxy(f, o):function`

`jQuery.proxy(o, name):function`

Return a function that invokes *f* as a method of *o*, or a function that invokes *o[name]* as a method of *o*.

`jQuery.queue(elt, [qname="fx"], [f])`

Query or set the named queue of *elt*, or add a new function *f* to that queue. Same as *\$(elt).queue(qname, f)*.

`jQuery.removeData(elt, [name]):void`

Remove the named property from the data object of *elt* or remove the data object itself.

jQuery.support

An object containing a number of properties describing the features and bugs of the current browser. Most are of interest only to plug-in writers. `jQuery.support.boxModel` is false in IE browsers running in quirks mode.

jQuery.trim(s):string

Return a copy of the string `s`, with leading and trailing whitespace trimmed off.

KeyEvent

see Event

Label

a `<label>` for a form control

Node, Element

A Label object represents a `<label>` element in an HTML form.

Properties

readonly Element control

The `FormControl` that this Label is associated with. If `htmlFor` is specified, this property is the control specified by that property. Otherwise, this property is the first `FormControl` child of the `<label>`.

readonly Form form

This property is a reference to the Form element that contains this label. Or, if the HTML form attribute is set, the Form element identified by that ID.

string htmlFor

This property mirrors the HTML `for` attribute. Since `for` is a reserved word in JavaScript, the name of this property is prefixed with “html” to create a legal identifier. If set, this property should specify the ID of the `FormControl` that this label is associated with. (It is usually simpler, however, to simply make that `FormControl` be a descendant of this Label.)

Link

an HTML hyperlink

Node, Element

HTML links are created with `<a>`, `<area>`, and `<link>` elements. `<a>` tags are used in the body of a document to create hyperlinks. `<area>` tags are a rarely used feature for creating “image maps.” `<link>` tags are used in the `<head>` of a document to refer to external resources such as stylesheets and icons. The `<a>` and `<area>` elements have the same representation in JavaScript. `<link>` elements have a somewhat different JavaScript representation, but, for convenience, these two types of links are documented together on this page.

When a Link object that represents an `<a>` element is used as a string, it returns the value of its `href` property.

Properties

In addition to the properties listed here, a Link object also has properties that reflect the underlying HTML attributes: `hreflang`, `media`, `ping`, `rel`, `sizes`, `target`, and `type`. Note that the URL decomposition properties (such as `host` and `pathname`) that return portions of the link's `href` are only defined for `<a>` and `<area>` elements, not for `<link>` elements, and that the `sheet`, `disabled`, and `relList` properties are only defined for `<link>` elements that refer to stylesheets.

boolean `disabled`

For `<link>` elements that refer to stylesheets, this property controls whether the stylesheet is applied to the document or not.

string `hash`

Specifies the fragment identifier of `href`, including the leading hash (#) mark—for example, “#results”.

string `host`

Specifies the hostname and port portions of `href`—for example, “http://www.oreilly.com:1234”.

string `hostname`

Specifies the hostname portion of `href`—for example, “http://www.oreilly.com”.

string `href`

Specifies the `href` attribute of the link. When an `<a>` or `<area>` element is used as a string, it is the value of this property that is returned.

string `pathname`

Specifies the path portion of `href`—for example, “/catalog/search.html”.

string `port`

Specifies the port portion of `href`—for example, “1234”.

string `protocol`

Specifies the protocol portion of `href`, including the trailing colon—for example, “http:”.

readonly DOMTokenList `relList`

Like the `classList` property of `Element`, this property makes it easy to query, set, and delete tokens from the HTML `rel` attribute of `<link>` elements.

string `search`

Specifies the query portion of `href`, including the leading question mark—for example, “?q=JavaScript&m=10”.

readonly `CSSStyleSheet` `sheet`

For `<link>` elements that reference stylesheets, this property represents the linked stylesheet.

string `text`

The plain-text content of an `<a>` or `<area>` element. A synonym for `Node.textContent`.

string title

All HTML elements allow a title attribute, and it usually specifies tooltip text for that element. Setting this attribute or property on a <link> element that has rel set to “alternate stylesheet” provides a name by which the user can enable or disable the stylesheet, and if the browser supports alternate stylesheets, the title you specify may appear within the browser UI in some fashion.

Location

represents and controls browser location

The `location` property of the Window and Document objects refers to a Location object that represents the web address (the “location”) of the current document. The `href` property contains the complete URL of that document, and the other properties of the Location object each describe a portion of that URL. These properties are much like the URL properties of the Link object. When a Location object is used as a string, the value of the `href` property is returned. This means that you can use the expression `location` in place of `location.href`.

In addition to representing the current browser location, the Location object also *controls* that location. If you assign a string containing a URL to the Location object or to its `href` property, the web browser loads and displays that URL. You can also make the browser load a new document by setting other Location properties to alter portions of the current URL. For example, if you set the `search` property, the browser reloads the current URL with a new query string appended. If you set the `hash` property, the browser does not load a new document, but it does create a new history entry. And if the `hash` property identifies an element of the document, the browser scrolls the document to make that element visible.

Properties

The properties of a Location object refer to the various portions of the current document’s URL. In each of the following property descriptions, the example given is a portion of this (fictitious) URL:

```
http://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results
```

string hash

The anchor portion of the URL, including the leading hash (#) mark—for example, “#results”. This portion of the document URL specifies the name of an anchor within the document.

string host

The hostname and port portions of the URL—for example, “http://www.oreilly.com:1234”.

string hostname

The hostname portion of a URL—for example, “http://www.oreilly.com”.

string href

The complete text of the document’s URL, unlike other Location properties that specify only portions of the URL. Setting this property to a new URL causes the browser to read

and display the contents of the new URL. Assigning a value directly to a `Location` object sets this property, and using a `Location` object as a string uses the value of this property.

string `pathname`

The pathname portion of a URL—for example, `"/catalog/search.html"`.

string `port`

The port portion of a URL—for example, `"1234"`. Note that this property is a string, not a number.

string `protocol`

The protocol portion of a URL, including the trailing colon—for example, `"http:"`.

string `search`

The query portion of a URL, including the leading question mark—for example, `"?q=JavaScript&m=10"`.

Methods

void `assign(string url)`

Load and display the contents of the specified `url`, as if the `href` property had been set to `url`.

void `reload()`

Reloads the document that is currently displayed.

void `replace(string url)`

Load and display the contents of the specified `url`, replacing the current document in the browsing history so that the browser's Back button will not take the browser back to the previously displayed document.

MediaElement

a media player element

Node, Element

`MediaElement` is the common superclass of the `<audio>` and `<video>` elements. Those two elements define almost exactly the same API, which is described here, but see [Audio](#) and [Video](#) for audio- and video-specific details. And see §21.2 for an introduction to these media elements.

Constants

The `NETWORK` constants are the possible values of the `networkState`, and the `HAVE` constants are the possible values of the `readyState` property.

unsigned short `NETWORK_EMPTY = 0`

The element has not started using the network. This would be the state before the `src` attribute was set.

unsigned short **NETWORK_IDLE** = 1

The element is not currently loading data from the network. It might have loaded the complete resource, or it might have buffered all the data it currently needs. Or it might have preload set to "none" and not yet have been asked to load or play the media.

unsigned short **NETWORK_LOADING** = 2

The element is currently using the network to load media data.

unsigned short **NETWORK_NO_SOURCE** = 3

The element is not using the network because it was not able to find a media source that it is able to play.

unsigned short **HAVE_NOTHING** = 0

No media data or metadata has been loaded.

unsigned short **HAVE_METADATA** = 1

The media metadata has been loaded, but no data for the current playback position has been loaded. This means that you can query the *duration* of the media or the dimensions of a video and you can seek by setting *currentTime*, but the browser cannot currently play the media at *currentTime*.

unsigned short **HAVE_CURRENT_DATA** = 2

Media data for *currentTime* has been loaded, but not enough data has been loaded to allow the media to play. For video, this typically means that the current frame has loaded, but the next one has not. This state occurs at the end of a sound or movie.

unsigned short **HAVE_FUTURE_DATA** = 3

Enough media data has been loaded to begin playing, but it is likely not enough to play to the end of the media without pausing to download more data.

unsigned short **HAVE_ENOUGH_DATA** = 4

Enough media data has been loaded that the browser is likely to be able to play to the end without pausing.

Properties

boolean **autoplay**

If *true*, the media element will automatically begin playing when it has loaded enough data. Mirrors the HTML *autoplay* attribute.

readonly TimeRanges **buffered**

The time ranges of the media data that are currently buffered.

boolean **controls**

If *true*, the media element should display a set of playback controls. Mirrors the HTML *controls* attribute.

readonly string **currentSrc**

The URL of the media data, from the *src* attribute or one of the *<source>* children of this element, or the empty string if no media data is specified.

double currentTime

The current playback time, in seconds. Set this property to make the media element skip to a new playback position.

double defaultPlaybackRate

The playback speed used for normal playback. The default is 1.0.

readonly double duration

The length, in seconds, of the media. If the duration is unknown (metadata has not been loaded, for example), this property will be NaN. If the media is a stream of indefinite duration, this property will be Infinity.

readonly boolean ended

True if the end of the media has been reached.

readonly MediaError error

This property is set when an error occurs and is null otherwise. It refers to an object whose code property describes the kind of error.

readonly double initialTime

The initial playback position, in seconds. This is usually 0, but some types of media (such as streaming media) may have a different starting point.

boolean loop

If true, the media element should automatically restart the media each time it reaches the end. This property mirrors the HTML `loop` attribute.

boolean muted

Specifies whether the audio is muted or not. You can set this property to mute and unmute audio. For `<video>` elements, you can use an `audio="muted"` attribute to mute the media by default.

readonly unsigned short networkState

Whether media data is currently loading or not. The legal values are listed in the Constants section above.

readonly boolean paused

true if playback is currently paused.

double playbackRate

The current playback speed. 1.0 is normal playback. Values greater than 1.0 are fast-forward. Values between 0 and 1.0 are slow-motion. Values less than 0 play the media backward. (Media is always muted when played backward, and it will also be muted when played particularly quickly or slowly.)

readonly TimeRanges played

The time ranges that have been played.

string preload

This property mirrors the HTML attribute of the same name, and you can use it to specify how much media data the browser should fetch before the user requests that the media be played. The value "none" means that no data should be preloaded. The value "metadata" means that the browser should fetch the media metadata (such as duration)

but not the actual data itself. The value "auto" (or just the empty string if the `preload` attribute is specified with no value) means that the browser is allowed to download the entire media resource, just in case the user decides to play it.

readonly unsigned short `readyState`

The media's readiness to play, based on the amount of data that has been buffered. The legal values are the `HAVE_` constants defined above.

readonly TimeRanges `seekable`

The range or ranges of times that you can set `currentTime` to. When playing back simple media files, this is typically any time between 0 and `duration`. But for streaming media, times in the past may no longer be buffered and times in the future may not yet be available.

readonly boolean `seeking`

This property is `true` while the media element is switching to a new `currentTime` playback position. If a new playback position is already buffered, this property will be `true` only for a short time. But if the media element must download new media data, seeking will remain `true` for a longer time.

string `src`

This property mirrors the HTML `src` attribute of the media element. You can set this property to make the media element load new media data. Note that this property is not the same as `currentSrc`.

readonly Date `startOffsetTime`

The real-world date and time of the playback position 0, if the media metadata includes that information. (A video file might include the time at which it was recorded, for example.)

double `volume`

This property queries and sets the volume for audio playback. It should be a value between 0 and 1. Also see the `muted` property.

Event Handlers

`<audio>` and `<video>` tags define the following event handlers, which can be set as HTML attributes or as JavaScript properties. At the time of this writing, some browsers do not support these properties and require you to register your event handlers using `addEventListener()` (see `EventTarget`). Media events do not bubble and have no default action to cancel. The associated event object is an ordinary `Event`.

Event Handler	Invoked When...
<code>onabort</code>	The element has stopped loading data, typically at the user's request. <code>error.code</code> is <code>error.MEDIA_ERR_ABORTED</code> .
<code>oncanplay</code>	Enough media data has loaded that playback can begin, but additional buffering is likely to be required.
<code>oncanplaythrough</code>	Enough media data has loaded that the media can probably be played all the way through without pausing to buffer more data.
<code>ondurationchange</code>	The duration property has changed

Event Handler	Invoked When...
<code>onemptied</code>	An error or abort has caused the <code>networkState</code> to return to <code>NETWORK_EMPTY</code> .
<code>onended</code>	Playback has stopped because the end of the media has been reached.
<code>onerror</code>	A network or other error prevented media data from being loaded. <code>error.code</code> is a value other than <code>MEDIA_ERR_ABORTED</code> (see <code>MediaError</code>).
<code>onloadeddata</code>	Data for the current playback position has loaded for the first time.
<code>onloadedmetadata</code>	The media metadata has been loaded, and the duration and dimensions of the media are ready.
<code>onloadstart</code>	The element begins requesting media data.
<code>onpause</code>	The <code>pause()</code> method was called and playback has been paused.
<code>onplay</code>	The <code>play()</code> method has been invoked, or the <code>autoplay</code> attribute has caused the equivalent.
<code>onplaying</code>	The media has begun to play.
<code>onprogress</code>	Network activity is continuing to load media data. Typically fired between 2 and 8 times per second. Note that the object associated with this event is a simple Event object, not the <code>ProgressEvent</code> object used by other APIs that fire events named "progress".
<code>onratechange</code>	The <code>playbackRate</code> or <code>defaultPlaybackRate</code> has changed.
<code>onseeked</code>	The seeking property has changed back to <code>false</code> .
<code>onseeking</code>	The script or user has requested that playback skip to an unbuffered portion of the media and playback has stopped while data loads. The seeking property is <code>true</code> .
<code>onstalled</code>	The element is trying to load data, but no data is arriving.
<code>onsuspend</code>	The element has buffered enough data and has temporarily stopped downloading.
<code>ontimeupdate</code>	The <code>currentTime</code> property has changed. During normal playback, this event is fired between 4 and 60 times per second.
<code>onvolumechange</code>	The <code>volume</code> or <code>muted</code> property has changed.
<code>onwaiting</code>	Playback cannot begin, or playback has stopped, because there is not enough data buffered. A playing event will follow when enough data is ready.

Methods

`string canPlayType(string type)`

This method asks the media element whether it can play media of the specified MIME *type*. If the player is certain it cannot play the type, it returns the empty string. If it is confident (but not certain) that it can play the type, it returns the string "probably". Media elements will generally not return "probably" unless *type* includes a `codecs=` parameter that lists specific media codecs. If the media element is not certain whether it will be able to play media of the specified *type*, this method will return "maybe".

`void load()`

This method resets the media element and makes it select a media source and start loading its data. This happens automatically when the element is first inserted into the document or whenever you set the `src` attribute. If you add, remove, or modify the `<source>` descendants of the media element, however, you should call `load()` explicitly.

void pause()

Pauses playback of the media.

void play()

Starts playback of the media.

MediaError

an `<audio>` or `<video>` error

When an error occurs on an `<audio>` or `<video>` tag, an error event is triggered and the `error` property is set to a `MediaError` object. The `code` property specifies what kind of error occurred. The following constants define the values of that property.

Constants

`unsigned short MEDIA_ERR_ABORTED = 1`

The user asked the browser to stop loading the media.

`unsigned short MEDIA_ERR_NETWORK = 2`

The media is of the right type, but a network error prevented it from being loaded.

`unsigned short MEDIA_ERR_DECODE = 3`

The media is of the right type, but an encoding error prevented it from being decoded and played.

`unsigned short MEDIA_ERR_SRC_NOT_SUPPORTED = 4`

The media specified by the `src` attribute is not a type that the browser can play.

Properties

readonly unsigned short code

This property describes the type of media error that occurred. Its value will be one of the constants above.

MessageChannel

a pair of connected `MessagePort`s

A `MessageChannel` is simply a pair of connected `MessagePort` objects. Calling `post Message()` on either one triggers a message event on the other. If you want to establish a private communication channel with a `Window` or `Worker` thread, create a `MessageChannel` and then pass one member of the `MessagePort` pair to the `Window` or `Worker` (using the `ports` argument of `postMessage()`).

`MessageChannel` and `MessagePort` types are an advanced feature of HTML5 and, at the time of this writing, some browsers support cross-origin messaging (§22.3) and worker threads (§22.4) without supporting private communication channels with `MessagePort`.

Constructor

`new MessageChannel()`

This no-argument constructor returns a new `MessageChannel` object.

Properties

readonly `MessagePort` **port1**

readonly `MessagePort` **port2**

These are the two connected ports that define the communication channel. The two are symmetrical: retain one or the other for your code, and pass the other to the Window or Worker you want to communicate with.

MessageEvent

a message from another execution context

Event

Various APIs use message events for asynchronous communication between unrelated execution contexts. The `Window`, `Worker`, `WebSocket`, `EventSource`, and `MessagePort` objects all define `onmessage` properties for handling message events. The message associated with a message event is any JavaScript value that can be cloned as described in “Structured Clones” on page 672. The message is wrapped in a `MessageEvent` object and available on the `data` property. The various APIs that rely on message events also define a few additional properties in the `MessageEvent` object. Message events do not bubble and have no default action to cancel.

Properties

readonly any **data**

This property holds the message that is being delivered. `data` can be of any type that can be cloned with the structured clone algorithm (“Structured Clones” on page 672): this includes core JavaScript values including objects and arrays but not functions. Client-side values such as `Document` and `Element` nodes are not allowed, although `Blobs` and `ArrayBuffers` are.

readonly string **lastEventId**

For message events on an `EventSource` (§18.3), this field contains the `lastEventId` string, if any, that was sent by the server.

readonly string **origin**

For message events on an `EventSource` (§18.3) or on a `Window` (§22.3), this property contains the origin URL of the message sender.

readonly `MessagePort`[] **ports**

For message events on a `Window` (§22.3), `Worker` (§22.4), or `MessagePort`, this property contains an array of `MessagePort` objects, if any were passed in the corresponding call to `postMessage()`.

readonly Window source

For message events on a Window (§22.3), this property refers to the Window from which the message was sent.

MessagePort

pass asynchronous messages

EventTarget

A MessagePort is used for asynchronous, event-based message passing, typically between JavaScript execution contexts, such as windows or worker threads. MessagePorts must be used in connected pairs: see MessageChannel. Calling `postMessage()` on a MessagePort triggers a message event on the MessagePort to which it is connected. The cross-origin messaging API (§22.3) and Web Workers (§22.4) also communicate using a `postMessage()` method and message events. Those APIs effectively use an implicit MessagePort object. Explicit use of MessageChannel and MessagePort enables the creation of additional private communication channels and can be used, for example, to allow direct communication between two sibling Worker threads.

MessageChannel and MessagePort types are an advanced feature of HTML5 and, at the time of this writing, some browsers support cross-origin messaging (§22.3) and worker threads (§22.4) without supporting private communication channels with MessagePort.

Methods

void close()

This method disconnects this MessagePort from the port to which it was connected (if any). Subsequent calls to `postMessage()` will have no effect, and no message events will be delivered in the future.

void postMessage(any message, [MessagePort[] ports])

Send a clone of the specified *message* through the port and deliver it in the form of a message event on the port to which this one is connected. If *ports* is specified, deliver those as part of the message event as well. *message* can be any value that is compatible with the structured clone algorithm (“Structured Clones” on page 672).

void start()

This method causes the MessagePort to start firing message events. Before this method is called, any data sent through the port is buffered. Delaying messages this way allows a script to register all of its event handlers before any messages are sent. Note, however, that you only need to call this method if you use the EventTarget method `addEventListener()`. If you simply set the `onmessage` property, `start()` will be called implicitly.

Event Handlers

onmessage

This property defines an event handler for message events. Message events are triggered on the MessagePort object. They do not bubble and have no default action. Note that setting this property calls the `start()` method to start the delivery of message events.

Meter

a graphical meter or gauge

Node, Element

A Meter object represents an HTML `<meter>` element that displays a graphical representation of a value within a range of possible values, where the range may optionally be annotated to indicate regions that are considered low, optimum, and high.

Most of the properties of this object simply mirror the HTML attributes with the same name. The JavaScript properties are numbers, however, while the HTML attributes are strings.

`<meter>` is an HTML5 element that, at the time of this writing, is not yet widely supported.

Properties

readonly `Form` **form**

The Form element, if there is one, that is the ancestor of this element or that was identified with the HTML `form` attribute.

double **high**

If specified, this property indicates that values between `high` and `max` should be graphically indicated as "high".

readonly `NodeList` **labels**

An array-like object of Label elements that are associated with this element.

double **low**

If specified, this property indicates that values between `min` and `low` should be graphically indicated as "low".

double **max**

The maximum value that can be displayed by the `<meter>`. The default is 1.

double **min**

The minimum value that can be displayed by the `<meter>`. The default is 0.

double **optimum**

If specified, the value that should be considered an optimum value.

double **value**

The value that is represented by this `<meter>`.

MouseEvent

see Event

Navigator

information about the web browser

The Navigator object contains properties that describe the web browser your code is running in. You can use its properties to perform platform-specific customization. The name of this object is a reference to the Netscape Navigator browser, but all browsers support it. There is only a single instance of the Navigator object, which you can reference through the `navigator` property of any Window object.

Historically, the Navigator object has been used for “client sniffing,” to run different code depending on what browser was in use. Example 14-3 shows a simple way to do this, and the accompanying text in §14.4 describes the pitfalls of relying on the Navigator object. A better approach to cross-browser compatibility is described in §13.4.3.

Properties

readonly string `appName`

The name of the browser. For Netscape-based browsers, the value of this property is “Netscape”. In IE, the value of this property is “Microsoft Internet Explorer”. For compatibility with existing code, many browsers return old, spoofed information.

readonly string `appVersion`

Version and platform information for the browser. For compatibility with existing code, most browsers return old out-of-date values for this property.

readonly Geolocation `geolocation`

A reference to the Geolocation object for this browser. The methods of that object allow a script to request the current geographical location of the user.

readonly boolean `onLine`

This property is `false` if the browser will not attempt to download anything from the network. This might be because the browser is certain that the computer is not connected to the network or because the user has configured the browser to perform no networking. If the browser will attempt downloads (because the computer might be online), this property is `true`. The browser fires online and offline events at the Window object when the state of this property changes.

readonly string `platform`

The operating system and/or hardware platform on which the browser is running. Although there is no standard set of values for this property, some typical values are “Win32”, “MacPPC”, and “Linux i586”.

readonly string `userAgent`

The value the browser uses for the user-agent header in HTTP requests. For example:

```
Mozilla/5.0 (X11; U; Linux i686; en-US)
AppleWebKit/534.16 (KHTML, like Gecko)
Chrome/10.0.648.45
Safari/534.16
```

Methods

void registerContentHandler(string mimeType, string url, string title)

This method requests the registration of the specified *url* as a handler for displaying content of the specified *mimeType*. *title* is a human-readable site title that the browser may display to the user. The *url* argument must contain the string “%s”. When this content handler is to be used to handle a web page of the specified *mimeType*, the URL of that web page will be encoded and inserted into the *url* in place of the “%s”. Then, the browser will visit the resulting URL. This is a new feature of HTML5 and may not be implemented in all browsers.

void registerProtocolHandler(string scheme, string url, string title)

This method is like `registerContentHandler()`, but it registers a website to use as a handler for the URL protocol *scheme*. *scheme* should be a string like “mailto” or “sms” without a colon. This is a new feature of HTML5 and may not be implemented in all browsers.

void yieldForStorageUpdates()

Scripts that use `Document.cookie` or `Window.localStorage` or `Window.sessionStorage` (see Storage and Chapter 20) are not supposed to be able to observe storage changes made by concurrently running (same-origin) scripts in other windows. Browsers can (though at the time of this writing, not all browsers do) prevent concurrent updates with a locking mechanism like those used for databases. In browsers that support it, this method explicitly releases the lock, potentially unblocking concurrent scripts in other windows. Stored values retrieved after calling this method may be different than those retrieved before calling it.

Node

All objects in a document tree (including the Document object itself) implement the Node interface, which provides the fundamental properties and methods for traversing and manipulating the tree. The `parentNode` property and `childNodes[]` array allow you to move up and down the document tree. You can enumerate the children of a given node by looping through the elements of `childNodes[]` or by using the `firstChild` and `nextSibling` properties (or the `lastChild` and `previousSibling` properties, to loop backward). The `appendChild()`, `insertBefore()`, `removeChild()`, and `replaceChild()` methods allow you to modify the document tree by altering the children of a node.

Every object in a document tree implements both the Node interface and a more specialized subinterface, such as `Element` or `Text`. The `nodeType` property specifies which subinterface a node implements. You can use this property to test the type of a node before using properties or methods of the more specialized interface. For example:

```
var n; // Holds the node we're working with
if (n.nodeType == 1) { // Or use the constant Node.ELEMENT_NODE
    var tagname = n.tagName; // If the node is an Element, this is the tag name
}
```

Constants

```

unsigned short ELEMENT_NODE = 1
unsigned short TEXT_NODE = 3
unsigned short PROCESSING_INSTRUCTION_NODE = 7
unsigned short COMMENT_NODE = 8
unsigned short DOCUMENT_NODE = 9
unsigned short DOCUMENT_TYPE_NODE = 10
unsigned short DOCUMENT_FRAGMENT_NODE = 11

```

These constants are possible values of the `nodeType` property. Note that these are static properties of the `Node()` constructor function; they are not properties of individual `Node` objects. Also note that they are not defined in IE8 and before. For compatibility, you can hardcode values or define your own constants.

```

unsigned short DOCUMENT_POSITION_DISCONNECTED = 0x01
unsigned short DOCUMENT_POSITION_PRECEDING = 0x02
unsigned short DOCUMENT_POSITION_FOLLOWING = 0x04
unsigned short DOCUMENT_POSITION_CONTAINS = 0x08
unsigned short DOCUMENT_POSITION_CONTAINED_BY = 0x10

```

These constants specify bits that may be on or off in the return value of `compareDocumentPosition()`.

Properties

readonly string `baseURI`

This property specifies the base URL of this `Node` against which relative URLs are resolved. For all nodes in HTML documents, this is the URL specified by the `<base>` element of the document, or just the `Document.URL` with the fragment identifier removed.

readonly `NodeList` `childNodes`

This property is an array-like object that contains the child nodes of the current node. This property should never be `null`: for nodes with no children, `childNodes` is an array with `length` zero. Note that the `NodeList` object is live: any changes to this element's list of children are immediately visible through the `NodeList`.

readonly `Node` `firstChild`

The first child of this node, or `null` if the node has no children.

readonly `Node` `lastChild`

The last child of this node, or `null` if the node has no children.

readonly `Node` `nextSibling`

The sibling node that immediately follows this one in the `childNodes[]` array of the `parentNode`, or `null` if there is no such node.

readonly string `nodeName`

The name of the node. For `Element` nodes, specifies the tag name of the element, which can also be retrieved with the `tagName` property of the `Element` interface. For most other types of nodes, the value is a constant string that depends on the node type.

readonly unsigned short `nodeType`

The type of the node—i.e., which subinterface the node implements. The legal values are defined by the previously listed constants. Since these constants are not supported by Internet Explorer, however, you may prefer to use hardcoded values instead of the constants. In HTML documents, the common values for this property are 1 for Element nodes, 3 for Text nodes, 8 for Comment nodes, and 9 for the single top-level Document node.

string `nodeValue`

The value of a node. For Text nodes, it holds the text content.

readonly Document `ownerDocument`

The Document object with which this node is associated. For Document nodes, this property is `null`. Note that nodes have owners even if they have not been inserted into a document.

readonly Node `parentNode`

The parent (or container) node of this node, or `null` if there is no parent. Note that the Document and DocumentFragment nodes never have parent nodes. Also, nodes that have been removed from the document, or that are newly created and have not yet been inserted into the document tree, have a `parentNode` of `null`.

readonly Node `previousSibling`

The sibling node that immediately precedes this one in the `childNodes[]` array of the `parentNode`, or `null` if there is no such node.

string `textContent`

For Text and Comment nodes, this property is just a synonym for the `data` property. For Element and DocumentFragment nodes, querying this property returns the concatenated text content of all Text node descendants. Setting this property on a Element or DocumentFragment replaces all descendants of that element or fragment with a single new Text node that holds the specified value.

Methods**Node `appendChild(Node newChild)`**

This method adds the node *newChild* to the document, inserting it as the last child of this node. If *newChild* is already in the document tree, it is removed from the tree and then reinserted at its new location. If *newChild* is a DocumentFragment node, it is not inserted itself; instead, all its children are appended, in order, to the end of this node's `childNodes[]` array. Note that a node from (or created by) one document cannot be inserted into a different document. That is, the `ownerDocument` property of *newChild* must be the same as the `ownerDocument` property of this node. (See `Document.adoptNode()`). This method returns the Node that was passed to it.

Node `cloneNode(boolean deep)`

The `cloneNode()` method makes and returns a copy of the node on which it is called. If passed the argument `true`, it recursively clones all descendants of the node as well. Otherwise, it clones only the node and none of its children. The returned node is not part of the document tree, and its `parentNode` property is `null`. When an Element node is cloned, all of its attributes

are also cloned. Note, however, that event-listener functions registered on a node are not cloned.

unsigned short **compareDocumentPosition**(Node *other*)

This method compares the document position of this node to the document position of the *other* node and returns a number whose set bits describe the relationship between the nodes. If the two nodes are the same, no bits are set and this method returns 0. Otherwise, one or more bits will be set in the return value. The `DOCUMENT_POSITION_` constants listed above give symbolic names for each of the bits, which have the following meanings:

<code>DOCUMENT_POSITION_</code>	Value	Meaning
<code>DISCONNECTED</code>	0x01	The two nodes are not in the same document, so their position cannot be compared.
<code>PRECEDING</code>	0x02	The <i>other</i> node appears before this node.
<code>FOLLOWING</code>	0x04	The <i>other</i> node comes after this node.
<code>CONTAINS</code>	0x08	The <i>other</i> node contains this node. When this bit is set, the <code>PRECEDING</code> bit is always also set.
<code>CONTAINED_BY</code>	0x10	The <i>other</i> node is contained by this node. When this bit is set, the <code>FOLLOWING</code> bit is always also set.

boolean **hasChildNodes**()

Returns true if this node has one or more children or false if it has none.

Node **insertBefore**(Node *newChild*, Node *refChild*)

This method inserts the node *newChild* into the document tree as a child of this node and then returns the inserted node. The new node is positioned within this node's `childNodes[]` array so that it comes immediately before the *refChild* node. If *refChild* is null, *newChild* is inserted at the end of `childNodes[]`, just as with the `appendChild()` method. Note that it is illegal to call this method with a *refChild* that is not a child of this node.

If *newChild* is already in the document tree, it is removed from the tree and then reinserted at its new position. If *newChild* is a `DocumentFragment` node, it is not inserted itself; instead, each of its children is inserted, in order, at the specified location.

boolean **isDefaultNamespace**(string *namespace*)

Returns true if the specified *namespace* URL is the same as the default namespace URL returned by `lookupNamespaceURI(null)` and false otherwise.

boolean **isEqualNode**(Node *other*)

Returns true if this node and *other* are identical, with equal type, tagname, attributes, and (recursively) children. Returns false if the two nodes are not equal.

boolean **isSameNode**(Node *other*)

Returns true if this node and *other* are the same node and false otherwise. You can also simply use the `==` operator.

string lookupNamespaceURI(string prefix)

This method returns the namespace URL associated with the specified namespace *prefix*, or null if there isn't one. If *prefix* is null, it returns the URL of the default namespace.

string lookupPrefix(string namespace)

This method returns the namespace prefix associated with the specified *namespace* URL, or null if there is none.

void normalize()

This method normalizes the text node descendants of this node, merging adjacent nodes and removing empty nodes. Documents do not normally have empty or adjacent text nodes, but this can occur when a script adds or removes nodes.

Node removeChild(Node oldChild)

This method removes *oldChild* from the `childNodes[]` array of this node. It is an error to call this method with a node that is not a child. `removeChild()` returns the *oldChild* node after removing it. *oldChild* continues to be a valid node and can be reinserted into the document later.

Node replaceChild(Node newChild, Node oldChild)

This method replaces *oldChild* with *newChild* and returns *oldChild*. *oldChild* must be a child of this node. If *newChild* is already part of the document, it is first removed from the document before being reinserted at its new position. If *newChild* is a `DocumentFragment`, it is not inserted itself; instead, each of its children is inserted, in order, at the position formerly occupied by *oldChild*.

NodeList

a read-only array-like object of Nodes

A `NodeList` is a read-only array-like object whose elements are `Node` objects (usually `Elements`). The `length` property specifies how many nodes are in the list, and you can retrieve those nodes from indexes 0 through `length-1`. You can also pass the desired index to the `item()` method instead of indexing the `NodeList` directly. The elements of a `NodeList` are always valid `Node` objects: `NodeLists` never contain null elements.

`NodeLists` are commonly used: the `childNodes` property of `Node`, and the return values of `Document.getElementsByTagName()`, `Element.getElementsByTagName()`, and `HTMLDocument.getElementsByTagName()` are all `NodeLists`, for example. Because `NodeList` is an array-like object, however, we often refer to those values informally as arrays, using language like “the `childNodes[]` array.”

Note that `NodeList` objects are usually live: they are not static snapshots but immediately reflect changes to the document tree. For example, if you have a `NodeList` that represents the children of a specific node and you then delete one of those children, the child is removed from your `NodeList`. Be careful when you are looping through the elements of a `NodeList`: the body of your loop can make changes to the document tree (such as deleting nodes) that can affect the contents of the `NodeList`!

Properties

readonly unsigned long **length**

The number of nodes in the `NodeList`.

Methods

`Node item(unsigned long index)`

Returns the `Node` at the specified *index* or `null` if the index is out of bounds.

Option

an `<option>` in a `Select` element

`Node`, `Element`

The `Option` object describes a single option displayed within a `Select` object. The properties of this object specify whether it is selected by default, whether it is currently selected, the position it has in the `options[]` array of its containing `Select` object, the text it displays, and the value it passes to the server if it is selected when the containing form is submitted.

For historical reasons, the `Option` element defines a constructor that you can use to create and initialize new `Option` elements. (You can also use the normal `Document.createElement()` method, of course.) Once a new `Option` object is created, it can be appended to the `options` collection of a `Select` object. See `HTMLOptionsCollection` for details.

Constructor

`new Option([string text, string value, boolean defaultSelected, boolean selected])`

The `Option()` constructor creates an `<option>` element. The four optional argument specify the `textContent` (see `Node`) of the element and the initial values of the `value`, `defaultSelected` and `selected` properties.

Properties

boolean **defaultSelected**

This property mirrors the HTML `selected` attribute. It defines the initial selectedness of the option, and also the value that is used when the form is reset.

boolean **disabled**

`true` if this option is disabled. Options are disabled if they or a containing `<optgroup>` has the HTML `disabled` attribute.

readonly Form **form**

The `<form>` element, if any, of which this `Option` element is a part.

readonly long **index**

The index of this `Option` element within its containing `Select` element. (See also `HTMLOptionsCollection`).

string **label**

The value of the HTML `label` attribute if there is one, or otherwise the `textContent` (see `Node`) of this `Option`.

boolean selected

true if this option is currently selected, or false otherwise.

string text

The `textContent` (see `Node`) of this `Option` element, with leading and trailing whitespace removed and runs of two or more spaces replaced with a single space character.

string value

The value of the HTML `value` attribute, if this `Option` has one, or the `textContent` of the element otherwise.

Output

an HTML form `<output>` element

`Node`, `Element`, `FormControl`

The `Output` object represents an HTML form `<output>` element. In browsers that support them, `Output` objects implement most of the properties of `FormControl`.

Properties

string defaultValue

This property is the initial value of the `Output` element's `textContent` (see `Node`). When the form is reset, its `value` is restored to this value. If this property is set and the `Output` element is currently displaying its previous `defaultValue`, the new default value will be displayed. Otherwise, the currently displayed value will not be changed.

readonly DOMSettableTokenList htmlFor

The HTML `for` attribute of an `<output>` element is a space-separated list of the IDs of elements whose values contributed to the computed content displayed by the `<output>` element. `for` is a reserved word in JavaScript, so this corresponding JavaScript property is named `htmlFor` instead. You can get and set this property as if it was an ordinary string value, or you can use the methods of `DOMTokenList` to query and set individual element IDs from the list.

PageTransitionEvent

event object for `pageshow` and `pagehide` events

`Event`

Browsers fire a `pageshow` event after the `load` event when a document first loads, and then fire another `pageshow` event each time the page is restored from the in-memory history cache. A `PageTransitionEvent` object is associated with each `pageshow` event, and its `persisted` property is `true` if the page is being restored rather than loaded or reloaded.

`Pagehide` events also have an associated `PageTransitionEvent` object, but the `persisted` property is always `true` for `pagehide` events.

`Pageshow` and `pagehide` events are triggered on the `Window` object. They do not bubble and have no default action to cancel.

Properties

readonly boolean **persisted**

For pageshow events, this property is **false** if the page was loaded (or reloaded) from the network or the disk cache. It is **true** if the page being shown was restored from the in-memory cache without being reloaded.

For pagehide events, this property is always **true**.

PopStateEvent

history transition event

Event

Web applications that manage their own history (see §22.2) use the `pushState()` method of `History` to create a new entry in the browsing history and associate a state value or object with it. When the user uses the browser's Back or Forward buttons to navigate between those saved states, the browser triggers a popstate event on the `Window` object and passes a copy of the saved application state in the associated `PopStateEvent` object.

Properties

readonly any **state**

This property holds a copy of the state value or object that was passed to the `History.pushState()` or `History.replaceState()` method. The **state** can be any value that can be cloned with the structured clone algorithm (see “Structured Clones” on page 672).

ProcessingInstruction

a processing instruction in an XML document

Node

This infrequently used interface represents a processing instruction (or PI) in an XML document. Programmers working with HTML documents will never encounter a `ProcessingInstruction` node.

Properties

string **data**

The content of the processing instruction (i.e., the first nonspace character after the target, up to but not including the closing `?>`).

readonly string **target**

The target of the processing instruction. This is the first identifier that follows the opening `<?;`; it specifies the “processor” for which the processing instruction is intended.

Progress

a progress bar

Node, Element

A `Progress` object represents an HTML `<progress>` element and displays a graphical representation of progress toward the completion of some kind of task.

When the amount of work or time required to complete the task is not known, the Progress element is said to be in an *indeterminate* state. In this state, it simply displays some kind of “working” animation to indicate that something is happening. When the total amount of work (or time or bytes) and the amount accomplished are known, the Progress element is a *determinate* state and can display progress with some kind of completion percentage graphic. `<progress>` is an HTML5 element that, at the time of this writing, is not yet widely supported.

Properties

readonly Form **form**

The Form element, if there is one, that is the ancestor of this element or that was identified with the HTML `form` attribute.

readonly NodeList **labels**

An array-like object of Label elements that are associated with this element.

double **max**

The total amount of work to be done. When using a Progress element to display upload or download progress of an XMLHttpRequest, for example, you might set this property to the total number of bytes to be transferred. This property mirrors the `max` attribute. The default value is 1.0.

readonly double **position**

If this is a determinate Progress element, this property is the computed value `value/max`. Otherwise this property will be -1.

double **value**

A value between 0 and `max` indicating how much progress has been made. This property mirrors the `value` attribute. If the attribute exists, the Progress element is a determinate element. If it does not exist, the Progress element is indeterminate. To switch from determinate to indeterminate mode (because of a stalled event from a MediaElement, for example), you can use the `removeAttribute()` method of Element.

ProgressEvent

downloading, uploading, or file reading progress

Event

The ApplicationCache, FileReader, and (level 2) XMLHttpRequest object all fire Progress events to inform interested applications of the progress of a data transfer process such as a network download or upload or a file read. Events of this sort are known generically as *Progress events*, but only one such event actually has the name “progress.” Other Progress events fired by FileReader and XMLHttpRequest are loadstart, load, loadend, error and abort. XMLHttpRequest also fires a timeout Progress event. ApplicationCache fires a number of events, but only the one named “progress” is a Progress event of the type described here.

Progress events are triggered in a sequence that begins with a loadstart event and always ends with a loadend event. The event immediately before loadend will be load, error, or abort, depending on whether the data transfer operation succeeded and if not, how it failed. Zero or more progress events (with the actual event name “progress”) are triggered between the initial loadstart and the final two events. (The ApplicationCache object fires a different se-

quence of events, but the progress event it fires as part of its cache update process is a Progress event.)

Event handlers for Progress events are passed a ProgressEvent object that specifies how many bytes of data have been transferred. This ProgressEvent object is unrelated to the HTML `<progress>` element described in Progress, but the ProgressEvent object passed to the `onprogress` event handler of an XMLHttpRequest (for example) could be used to update the state of a `<progress>` element to display a visual download completion percentage value to the user.

Properties

readonly boolean lengthComputable

true if the total number of bytes to transfer is known and false otherwise. If this property is true, the data transfer completion percentage for a ProgressEvent e can be computed as:

```
var percentComplete = Math.floor(100*e.loaded/e.total);
```

readonly unsigned long loaded

How many bytes have been transferred so far.

readonly unsigned long total

The total number of bytes to be transferred, if that value is known, or 0 otherwise. This information might come from the size property of a Blob or the Content-Length header returned by a web server, for example.

Screen

information about the display screen

The screen property of a Window refers to a Screen object. The properties of this global object contain information about the computer monitor on which the browser is displayed. JavaScript programs can use this information to optimize their output for the user's display capabilities. For example, a program can choose between large and small images based on the display size.

Properties

readonly unsigned long availHeight

Specifies the available height, in pixels, of the screen on which the web browser is displayed. This available height does not include vertical space allocated to permanent desktop features, such as a bar or dock at the bottom of the screen.

readonly unsigned long availWidth

Specifies the available width, in pixels, of the screen on which the web browser is displayed. This available width does not include horizontal space allocated to permanent desktop features.

readonly unsigned long colorDepth

readonly unsigned long pixelDepth

These synonymous properties both specify the color depth of the screen in bits per pixel.

readonly unsigned long height

Specifies the total height, in pixels, of the screen on which the web browser is displayed.
See also `availHeight`.

readonly unsigned long width

Specifies the total width, in pixels, of the screen on which the web browser is displayed.
See also `availWidth`.

Script

an HTML `<script>` element

Node, Element

A `Script` object represents an HTML `<script>` element. Most of its properties simply mirror the HTML attributes with the same name, but `text` works like the `textContent` property inherited from `Node`.

Note that a `<script>` will not run more than once. You cannot change the `src` or `text` property of an existing `<script>` element to make it run a new script. You can set these properties on a newly created `<script>` element to execute a script, however. Note, though, that a `<script>` tag must be inserted into a Document in order to run. The script will be executed when `src` or `type` is set or when it is inserted into the document, whichever comes last.

Properties

boolean `async`

`true` if the `<script>` element has an `async` attribute and `false` otherwise. See §13.3.1.

string `charset`

The character encoding of the script specified by `src` URL. This property is not normally specified, and the default is to interpret the script using the same encoding as the containing document.

boolean `defer`

`true` if the `<script>` element has a `defer` attribute and `false` otherwise. See §13.3.1.

string `src`

The URL of the script to be loaded.

string `text`

The text that appears between the `<script>` tag and the closing `</script>` tag.

string `type`

The MIME type of the scripting language. The default is “text/javascript”, and you do not need to set this property (or the HTML attribute) for ordinary JavaScript scripts. If you set this property to a custom MIME type of your own, you can embed arbitrary textual data within the `<script>` element for use by other scripts.

Select

a graphical selection list

Node, Element, FormControl

The **Select** element represents an HTML `<select>` tag, which displays a graphical list of choices to the user. If the HTML `multiple` attribute is present, the user may select any number of options from the list. If that attribute is not present, the user may select only one option, and options have a radio button behavior—selecting one deselects whichever was previously selected.

The options in a **Select** element can be displayed in two distinct ways. If the `size` attribute has a value greater than 1, or if the `multiple` attribute is present, they are displayed in a list box that is `size` lines high in the browser window. If `size` is smaller than the number of options, the listbox includes a scrollbar. On the other hand, if `size` is 1 and `multiple` is not specified, the currently selected option is displayed on a single line, and the list of other options is made available through a drop-down menu. The first presentation style displays the options clearly but requires more space in the browser window. The second style requires minimal space but does not display alternative options as explicitly. `size` defaults to 4 when the `multiple` attribute is set, and 1 otherwise.

The `options[]` property of the **Select** element is the most interesting. This is an array-like object of `<option>` elements (see **Option**) that describe the choices presented by the **Select** element. For historical reasons, this array-like object has some unusual behaviors for adding and removing new `<option>` elements. See **HTMLOptionsCollection** for details.

For a **Select** element without the `multiple` attribute specified, you can determine which option is selected with the `selectedIndex` property. When multiple selections are allowed, however, this property tells you the index of only the first selected option. To determine the full set of selected options, you must iterate through the `options[]` array and check the `selected` property of each **Option** object.

Properties

In addition to the properties listed here, **Select** elements also define the properties of **Element** and **FormControl** and mirror HTML attributes with the following JavaScript properties: `multiple`, `required`, and `size`.

unsigned long `length`

The number of elements in the `options` collection. **Select** objects are themselves array-like objects, and for a **Select** object `s` and a number `n`, `s[n]` is the same as `s.options[n]`.

readonly **HTMLOptionsCollection** `options`

An array-like object of **Option** elements contained by this **Select** element. See **HTMLOptionsCollection** for a description of the historical behavior of this collection.

long `selectedIndex`

The position of the selected option in the `options` array. If no options are selected, this property is `-1`. If multiple options are selected, this property holds the index of the first selected option.

Setting the value of this property selects the specified option and deselects all other options, even if the **Select** object has the `multiple` attribute specified. When you're doing

listbox selection (when `size > 1`), you can deselect all options by setting `selectedIndex` to `-1`. Note that changing the selection in this way does not trigger the `onchange()` event handler.

readonly `HTMLCollection` `selectedOptions`

A read-only array-like object of `Option` elements that are selected. This is a new property defined by HTML5 and, at the time of this writing, it is not yet widely supported.

Methods

The methods listed here all delegate to the same-named methods of the `options` property; see `HTMLOptionsCollection` for details. In addition to these methods, `Select` elements also implement the methods of `Element` and `FormControl`.

void `add`(`Element` *element*, [*any before*])

This method works just like `options.add()` to add a new `Option` element.

any `item`(`unsigned long` *index*)

This method works just like `options.item()` to return an `Option` element. It is also invoked when the user indexes the `Select` object directly.

any `namedItem`(`string` *name*)

This method is just like `options.namedItem()`. See `HTMLOptionsCollection`.

void `remove`(`long` *index*)

This method works just like `options.remove()` to remove an `Option` element. See `HTMLOptionsCollection`.

Storage

client-side storage of name/value pairs

The `localStorage` and `sessionStorage` properties of `Window` are both `Session` objects that represent persistent client-side associative arrays that map string keys to values. In theory, a `Session` object can store any value that can be cloned with the structured clone algorithm (see “Structured Clones” on page 672). At the time of this writing, however, browsers only support string values.

The methods of a `Storage` object allow you to add new key/value pairs, remove key/value pairs, and query the value associated with a specified key. You don’t have to call these methods explicitly, however: you can use array indexing and the `delete` operator in their place, and treat `localStorage` and `sessionStorage` as if they were ordinary JavaScript objects.

If you change the contents of a `Storage` object, any other `Windows` that have access to the same storage area (because they’re displaying a document from the same origin) will be notified of the change with a `StorageEvent`.

Properties

readonly `unsigned long` `length`

The number of key/value pairs stored.

Methods

void clear()

Removes all stored key/value pairs.

any getItem(string key)

Returns the value associated with *key*. (In implementations current at the time of this writing, the return value will always be a string.) This method is invoked implicitly when you index the Storage object to retrieve a property named *key*.

string key(unsigned long n)

Returns the *n*th key in this Storage object or *null* if *n* is greater than or equal to *length*. Note that the order of the keys may change if you add or remove key/value pairs.

void removeItem(string key)

Removes *key*, and its associated value, from this Storage object. This method is invoked implicitly if you use the *delete* operator to delete the property named *key*.

void setItem(string key, any value)

Add the specified *key* and *value* to this Storage object, replacing any value that is already associated with *key*. This method is invoked implicitly if you assign a *value* to the property named *key* of the Storage object. That is, you can use ordinary JavaScript property access and assignment syntax instead of explicitly calling *setItem()*.

StorageEvent

Event

The *localStorage* and *sessionStorage* properties of a Window object refer to Storage objects that represent client-side storage areas (see §20.1). If more than one window, tab, or frame is displaying documents from the same origin, multiple windows have access to the same storage areas. If a script in one window changes the contents of a storage area, a storage event is triggered on all other Window objects that share access to that storage area. (Note that the event is not triggered in the window that made the change.) Storage events are triggered on the Window object and do not bubble. They do not have any default action that can be canceled. The object associated with a storage event is a StorageEvent object, and its properties describe the change that occurred to the storage area.

Properties

readonly string key

This property is the key that was set or deleted. If the entire storage area was cleared with *Storage.clear()*, this property (as well as *newValue* and *oldValue*) will be *null*.

readonly any newValue

The new value of the specified key. This will be *null* if the key was removed. At the time of this writing, browser implementations only allow string values to be stored.

readonly any oldValue

The old value of the key that was changed, or null if this key was newly added to the storage area. At the time of this writing browser implementations only allow string values to be stored.

readonly Storage storageArea

This property will be equal to the `localStorage` or `sessionStorage` property of the Window that receives this event and indicates which storage area was changed.

readonly string url

This is the URL of the document whose script changed the storage area.

Style

an HTML `<style>` element

Node, Element

A Style object represents an HTML `<style>` tag.

Properties

boolean disabled

Set this property to `true` to disable the stylesheet associated with this `<style>` element, and set it to `false` to re-enable it.

string media

This property mirrors the HTML `media` attribute and specifies the mediums to which the specified styles apply.

boolean scoped

This property is `true` if the HTML `scoped` attribute is present on the `<style>` element, and `false` otherwise. At the time of this writing, browsers do not support scoped stylesheets.

readonly CSSStyleSheet sheet

The `CSSStyleSheet` defined by this `<style>` element.

string title

All HTML elements allow a `title` attribute. Setting this attribute or property on a `<style>` element may allow the user to select the stylesheet (as an alternate stylesheet) by title, and the title you specify may appear within the web browser UI in some fashion.

string type

Mirrors the HTML `type` attribute. The default value is `"text/css"`, and you do not normally need to set this attribute.

Table

an HTML `<table>`

Node, Element

The Table object represents an HTML `<table>` element and defines a number of convenience properties and methods for querying and modifying various sections of the table. These

methods and properties make it easier to work with tables, but their functionality can also be duplicated with core DOM methods.

HTML tables are composed of sections, rows, and cells. See `TableCell`, `TableRow`, and `TableSection`.

Properties

In addition to the properties listed here, Table elements also have a `summary` property that mirrors the HTML attribute of the same name.

Element `caption`

A reference to the `<caption>` element for the table, or `null` if there is none

readonly HTMLCollection `rows`

An array-like object of `TableRow` objects that represent all the rows in the table. This includes all rows defined within `<thead>`, `<tfoot>`, and `<tbody>` tags.

readonly HTMLCollection `tBodies`

An array-like object of `TableSection` objects that represent all the `<tbody>` sections in this table.

TableSection `tFoot`

The `<tfoot>` element of the table, or `null` if there is none.

TableSection `tHead`

The `<thead>` element of the table, or `null` if there is none.

Methods

Element `createCaption()`

This method returns an `Element` object representing the `<caption>` of this table. If the table already has a caption, this method simply returns it. If the table does not have an existing `<caption>`, this method creates a new (empty) caption and inserts it into the table before returning it.

TableSection `createTBody()`

This method creates a new `<tbody>` element, inserts it into the table, and returns it. The new element is inserted after the last `<tbody>` in the table, or at the end of the table.

TableSection `createTFoot()`

This method returns a `TableSection` representing the first `<tfoot>` element for this table. If the table already has a footer, this method simply returns it. If the table does not have an existing footer, this method creates a new (empty) `<tfoot>` element and inserts it into the table before returning it.

TableSection `createTHead()`

This method returns a `TableSection` representing the first `<thead>` element for this table. If the table already has a header, this method simply returns it. If the table does not have an existing header, this method creates a new (empty) `<thead>` element and inserts it into the table before returning it.

void deleteCaption()

Removes the first <caption> element from the table, if it has one.

void deleteRow(long index)

This method deletes the row at the specified position from the table. Rows are numbered in the order in which they appear in the document source. Rows in <thead> and <tfoot> sections are numbered along with all other rows in the table.

void deleteTFoot()

Removes the first <tfoot> element from the table, if it has one.

void deleteTHead()

Removes the first <thead> element from the table, if it has one.

TableRow insertRow([long index])

This method creates a new <tr> element, inserts it into the table at the specified *index*, and returns it.

The new row is inserted in the same section and immediately before the existing row at the position specified by *index*. If *index* is equal to the number of rows in the table (or to -1), the new row is appended to the last section of the table. If the table is initially empty, the new row is inserted into a new <tbody> section that is itself inserted into the table.

You can use the convenience method `TableRow.insertCell()` to add content to the newly created row. See also the `insertRow()` method of `TableSection`.

TableCell

a cell in an HTML table

Node, Element

A `TableCell` object represents a <td> or <th> element.

Properties

readonly long **cellIndex**

The position of this cell within its row.

unsigned long **colSpan**

The value of the HTML `colspan` attribute, as a number.

unsigned long **rowSpan**

The value of the HTML `rowspan` attribute, as a number.

TableRow

a <tr> element in an HTML table

Node, Element

A `TableRow` object represents a row (a <tr> element) in an HTML table and defines properties and methods for working with the `TableCell` elements of the row.

Properties

readonly HTMLCollection cells

An array-like object of TableCell objects representing the <td> and <th> elements in this row.

readonly long rowIndex

The index of this row in the table.

readonly long sectionRowIndex

The position of this row within its section (i.e., within the <thead>, <tbody>, or <tfoot> in which it is contained).

Methods

void deleteCell(long index)

This method deletes the cell at the specified *index* in the row.

Element insertCell([long index])

This method creates a new <td> element, inserts it into the row at the specified position, and then returns it. The new cell is inserted immediately before the cell that is currently at the position specified by *index*. If *index* is omitted, is -1, or is equal to the number of cells in the row, the new cell is appended to the end of the row.

Note that this convenience method inserts <td> data cells only. If you need to add a header cell into a row, you must create and insert the <th> element using `Document.createElement()` and `Node.insertBefore()` or related methods.

TableSection

a header, footer, or body section of a table

Node, Element

A TableSection object represents a <tbody>, <thead>, or <tfoot> element in an HTML table. The `tHead` and `tFoot` properties of a Table are TableSection objects, and the `tBodies` property is an HTMLCollection of TableSection objects.

A TableSection contains TableRow objects and is contained in a Table object.

Properties

readonly HTMLCollection rows

An array-like object of TableRow objects representing the rows in this section of the table.

Methods

void deleteRow(long index)

This method deletes the row at the specified position within this section.

TableRow insertRow([long index])

This method creates a new <tr> element, inserts it into this table section at the specified position, and returns it. If *index* is -1 or is omitted or equals the number of rows currently in

the section, the new row is appended to the end of the section. Otherwise, the new row is inserted immediately before the row that is currently at the position specified by *index*. Note that for this method, *index* specifies a row position within a single table section, not within the entire table.

Text

a run of text in a document

Node

A Text node represents a run of plain text in a document and typically appear in the document tree as children of Element. The textual content of a Text node is available through the `data` property or through the `nodeValue` and `textContent` properties inherited from Node. You can create a new Text node with `Document.createTextNode()`. Text nodes never have children.

Properties

`string data`

The text contained by this node.

`readonly unsigned long length`

The length, in characters, of the text.

`readonly string wholeText`

The text content of this node and any adjacent text nodes before or after this one. If you've called the `normalize()` method of the parent Node, this property will be the same as `data`.

Methods

Unless you are writing a web-based text editor application, these methods are not commonly used.

`void appendData(string text)`

This method appends the specified *text* to the end of this Text node.

`void deleteData(unsigned long offset, unsigned long count)`

This method deletes characters from this Text node, starting with the character at the position *offset* and continuing for *count* characters. If *offset* plus *count* is greater than the number of characters in the Text node, all characters from *offset* to the end of the string are deleted.

`void insertData(unsigned long offset, string text)`

This method inserts the specified *text* into the Text node at the specified *offset*.

`void replaceData(unsigned long offset, unsigned long count, string text)`

This method replaces *count* characters starting at position *offset* with the contents of the string *text*. If the sum of *offset* and *count* is greater than the length of the Text node, all characters from *offset* on are replaced.

Text `replaceAllText(string text)`

This method creates a new Text node that contains the specified *text*. It then replaces this node and any adjacent Text nodes with the single new node and returns the new node. See the `wholeText` property above and the `normalize()` method of `Node`.

Text `splitText(unsigned long offset)`

This method splits a Text node in two at the specified *offset*. The original Text node is modified so that it contains all text content up to, but not including, the character at position *offset*. A new Text node is created to contain all the characters from (and including) the position *offset* to the end of the string. This new Text node is the return value of the method. Additionally, if the original Text node has a `parentNode`, the new node is inserted into this parent node immediately after the original node.

string `substringData(unsigned long offset, unsigned long count)`

This method extracts and returns the substring that starts at position *offset* and continues for *count* characters from the text of a Text node. If a Text node contains a very large amount of text, using this method may be more efficient than using `String.substring()`.

TextArea

a multiline text input area

Node, Element, FormControl

A `TextArea` object represents an HTML `<textarea>` element that creates a multiline text input field, often within an HTML form. The initial content of the text area is specified as plain text between the `<textarea>` and `</textarea>` tags. You can query and set the displayed text with the `value` property.

`TextArea` is a form control element like `Input` and `Select`. Like those objects, it defines `form`, `name`, `type`, and `value` properties and the other properties and methods documented in `FormControl`.

Properties

In addition to the properties listed here, `TextArea` elements also define the properties of `Element` and `FormControl` and mirror HTML attributes with the following JavaScript properties: `cols`, `maxLength`, `rows`, `placeholder`, `readOnly`, `required`, and `wrap`.

string `defaultValue`

The initial plain-text content of the `<textarea>` element. When the form is reset, the text area is restored to this value. This property is the same as the `textContent` property inherited from `Node`.

unsigned long `selectionEnd`

Returns or sets the index of the first input character after the selected text. See also `setSelectionRange()`.

unsigned long `selectionStart`

Returns or sets the index of the first selected character in the `<textarea>`. See also `set Selection Range()`.

readonly unsigned long **textLength**

The length, in characters, of the **value** property (see **FormControl**).

Methods

In addition to the methods listed here, **TextArea** elements also implement the methods of **Element** and **FormControl**.

void select()

This method selects all the text displayed by this **<textarea>** element. In most browsers, this means that the text is highlighted and that new text entered by the user replaces the highlighted text instead of being appended to it.

void setSelectionRange(unsigned long start, unsigned long end)

Select text displayed in the **<textarea>**, starting with the character at position *start* and continuing up to (but not including) the character at *end*.

TextMetrics

measurements of a string of text

A **TextMetrics** object is returned by the **measureText()** method of **CanvasRenderingContext2D**. Its **width** property holds the width of the measured text, in CSS pixels. Additional metrics may be added in the future.

Properties

readonly double **width**

The width, in CSS pixels, of the measured text.

TimeRanges

a set of media time ranges

The **buffered**, **played**, and **seekable** properties of a **MediaElement** represent the portions of a media timeline that have data buffered, that have been played, and that playback can be started at. Each of these portions of the timeline may include multiple disjoint time ranges (this happens to the **played** property when the user skips to the middle of a video file, for example). A **TimeRanges** object represents zero or more disjoint time ranges. The **length** property specifies the number of ranges, and the **start()** and **end()** methods return the bounds of each range.

The **TimeRanges** objects returned by **MediaElements** are always *normalized*, which means that the ranges they contain are in order, nonempty and do not touch or overlap.

Properties

readonly unsigned long **length**

The number of ranges represented by this **TimeRanges** object.

Methods

double end(unsigned long *n*)

Returns the end time (in seconds) of time range *n*, or throws an exception if *n* is less than zero or greater than or equal to **length**.

double start(unsigned long *n*)

Returns the start time (in seconds) of time range *n*, or throws an exception if *n* is less than zero or greater than or equal to **length**.

TypedArray

fixed-size binary arrays

ArrayBufferView

A **TypedArray** is an **ArrayBufferView** that interprets the bytes of an underlying **ArrayBuffer** as an array of numbers and allows read and write access to the elements of that array. This page does not document a single type named **TypedArray**. Instead, it covers eight different kinds of *typed arrays*. These eight types are all subtypes of **ArrayBufferView**, and they differ from each other only in the number of bytes per array element and the way those bytes are interpreted. The eight actual types are:

Int8Array

One byte per array element, interpreted as a signed integer.

Int16Array

Two bytes per array element, interpreted as a signed integer, using platform byte order.

Int32Array

Four bytes per array element, interpreted as a signed integer, using platform byte order.

Uint8Array

One byte per array element, interpreted as an unsigned integer.

Uint16Array

Two bytes per array element, interpreted as an unsigned integer, using platform byte order.

Uint32Array

Four bytes per array element, interpreted as an unsigned integer, using platform byte order.

Float32Array

Four bytes per array element, interpreted as a floating-point number, using platform byte order.

Float64Array

Eight bytes per array element, interpreted as a floating-point number, using platform byte order.

As their name implies, these are array-like objects, and you can get and set element values using normal square-bracket array notation. Note, however, that objects of these types always have a fixed length.

As noted in the descriptions above, the **TypedArray** classes use the default byte ordering of the underlying platform. See **DataView** for an **ArrayBuffer** view that allows explicit control over byte order.

Constructor

```
new TypedArray(unsigned long length)
new TypedArray(TypedArray array)
new TypedArray(type[] array)
new TypedArray(ArrayBuffer buffer, [unsigned long byteOffset], [unsigned long length])
```

Each of the eight kinds of TypedArray has a constructor that can be invoked in the four ways shown above. The constructors work as follows:

- If the constructor is called with a single numeric argument, it creates a new typed array with the specified number of elements, and initializes each element to 0.
- If passed a single typed array object, the constructor creates a new typed array with the same number of arguments as the argument array, and then copies the elements of the argument array to the newly created array. The type of the argument array need not be the same as the type of the array being created.
- If passed a single array (a true JavaScript array), the constructor creates a new typed array with the same number of arguments and then copies element values from the argument array into the new array.
- Finally, if passed an ArrayBuffer object, along with optional offset and length arguments, the constructor creates a new typed array that is a view of the specified region of the specified ArrayBuffer. The length of the new TypedArray depends on the ArrayBuffer region and on the element size of the typed array.

Constants

long `BYTES_PER_ELEMENT`

The number of bytes that each element of this array occupies in the underlying ArrayBuffer. This constant will have the value 1, 2, 4, or 8, depending on what kind of TypedArray is in use.

Properties

readonly unsigned long `length`

The number of elements in the array. TypedArrays have fixed size, and the value of this property never changes. Note that this property is not, in general, the same as the `byteLength` property inherited from `ArrayBufferView`.

Methods

void `set(TypedArray array, [unsigned long offset])`

Copy elements of `array` into this typed array, starting at index `offset`.

void `set(number[] array, [unsigned long offset])`

This version of `set()` is like the one above, but it uses a true JavaScript array rather than a typed array.

TypedArray `subarray(long start, long end)`

Return a new TypedArray that uses the same underlying ArrayBuffer as this one does. The first element of the returned array is element `start` of this array. And the last element of the

returned array is element *end*-1 of this array. Negative values of *start* and *end* are interpreted as offsets from the end of this array rather than the beginning.

URL

Blob URL methods

The URL property of the Window object refers to this URL object. In the future, this object may become a constructor for a URL parsing and manipulation utility class. At the time of this writing, however, it simply serves as a namespace for the two Blob URL functions described below. See §22.6 and §22.6.4 for more on Blobs and Blob URLs.

The URL object is new at the time of this writing and the API is not yet stable. You may need to use it with a vendor-specific prefix: `webkitURL`, for example.

Functions

string `createObjectURL(Blob blob)`

Return a Blob URL for the specified *blob*. HTTP GET requests for that URL will return the contents of *blob*.

void `revokeObjectURL(string url)`

Revoke the specified Blob *url*, so that it is no longer associated with any Blob and can no longer be loaded.

Video

an HTML `<video>` element

Node, Element, `MediaElement`

A Video object represents an HTML `<video>` element. `<video>` and `<audio>` elements are very similar and their common properties and methods are documented in `MediaElement`. This page documents a handful of additional properties that are specific to Video objects.

Properties

DOMSettableTokenList `audio`

This property specifies audio options for the video. The options are specified as a space-separated list of tokens on the HTML `audio` attribute, and the set is mirrored in JavaScript as a `DOMSettableTokenList`. At the time of this writing, however, the HTML5 standard defines only one legal token (“muted”), and you can treat this property as a string.

unsigned long `height`

The onscreen height of the `<video>` element, in CSS pixels. Mirrors the HTML `height` attribute.

string `poster`

The URL of an image to be displayed as a “poster frame” before the video begins playing. Mirrors the HTML `poster` attribute.

readonly unsigned long **videoHeight**

readonly unsigned long **videoWidth**

These properties return the intrinsic width and height of the video (i.e., the size of its frames) in CSS pixels. These properties will be zero until the `<video>` element has loaded the video metadata (while `readyState` is still `HAVE_NOTHING` and the `loadedmetadata` event has not been dispatched).

unsigned long **width**

The desired onscreen width of the `<video>` element, in CSS pixels. Mirrors the HTML `width` attribute.

WebSocket

a bidirectional socket-like network connection

EventTarget

A `WebSocket` represents a long-lived, bidirectional, socket-like network connection to a server that supports the `WebSocket` protocol. This is a fundamentally different networking model than the request/response model of HTTP. Create a new connection with the `WebSocket()` constructor. Use `send()` to send textual messages to the server, and register a handler for message events to receive messages from the server. See §22.9 for further details.

WebSockets are a new Web API and, at the time of this writing, are not supported by all browsers.

Constructor

`new WebSocket(string url, [string[] protocols])`

The `WebSocket()` constructor creates a new `WebSocket` object and begins the (asynchronous) process of establishing a connection to a `WebSocket` server. The `url` argument specifies the server to connect to and must be an absolute URL that uses the `ws://` or `wss://` URL scheme. The `protocols` argument is an array of subprotocol names. If the argument is specified, it is the client's way of telling the server which communication protocols or which protocol versions it is able to "speak." The server must choose one and inform the client as part of the connection process. `protocols` may also be specified as a single string instead of an array: in this case, it is treated as an array of length 1.

Constants

These constants are the values of the `readyState` property.

unsigned short **CONNECTING** = 0

The connection process is underway.

unsigned short **OPEN** = 1

The `WebSocket` is connected to the server; messages can be sent and received.

unsigned short **CLOSING** = 2

The connection is closing.

unsigned short **CLOSED** = 3

The connection has closed.

Properties

readonly unsigned long **bufferedAmount**

The number of characters of text that have been passed to `send()` but not yet actually sent. If you need to send large amounts of data, you can use this property to ensure that you are not sending messages faster than they can be transmitted.

readonly string **protocol**

If an array of subprotocols was passed to the `WebSocket()` constructor, this property holds the one chosen by the server. Note that when the `WebSocket` is first created, the connection is not established and the server's choice is not known. So this property starts as the empty string. If you passed protocols to the constructor, this property will change to reflect the server's choice of subprotocol when the open event is triggered.

readonly unsigned short **readyState**

The current state of the `WebSocket` connection. This property holds one of the constant values listed above.

readonly string **url**

This property holds the URL that was passed to the `WebSocket()` constructor.

Methods

void close()

If the connection is not already closed or closing, this method begins the process of closing it, setting `readyState` to `CLOSING`. Message events may continue to be fired even after `close()` is called, until `readyState` changes to `CLOSED` and the close event is fired.

void send(string data)

This method sends the specified *data* to the server at the other end of the `WebSocket` connection. This method throws an exception if called before the open event has been triggered, while `readyState` is still `CONNECTING`. The `WebSocket` protocol supports binary data, but at the time of this writing, the `WebSocket` API only allows strings to be sent and received.

Event Handlers

Network communication is inherently asynchronous, and like `XMLHttpRequest`, the `WebSocket` API is event-based. `WebSocket` defines four event handler registration properties, and also implements `EventTarget`, so you can also register event handlers using the `EventTarget` methods. The events described below are all fired at the `WebSocket` object. None of them bubble, and none have a default action to cancel. Note, however, that they do have different event objects associated with them.

onclose

A close event is triggered when the `WebSocket` connection closes (and `readyState` changes to `CLOSED`). The associated event object is a `CloseEvent`, which specifies whether the connection closed cleanly or not.

onerror

An error event is triggered when a network or `WebSocket` protocol error occurs. The associated event object is a simple `Event`.

onmessage

When the server sends data through the WebSocket, the WebSocket fires a message event, with an associated `MessageEvent` object whose `data` property refers to the received message.

onopen

The `WebSocket()` constructor returns before the connection to the specified `url` is established. When the connection handshake completes and the WebSocket is ready to send and receive data, an open event is fired. The associated event object is a simple `Event`.

Window

a web browser window, tab, or frame

EventTarget

The `Window` object represents a browser window, tab, or frame. It is documented in detail in Chapter 14. In client-side JavaScript, the `Window` serves as the “global object,” and all expressions are evaluated in the context of the current `Window` object. This means that no special syntax is required to refer to the current window, and you can use the properties of that window object as if they were global variables. For example, you can write `document` rather than `window.document`. Similarly, you can use the methods of the current window object as if they were functions: e.g., `alert()` instead of `window.alert()`.

Some of the properties and methods of this object actually query or manipulate the browser window in some way. Others are defined here simply because this is the global object. In addition to the properties and methods listed here, the `Window` object also implements all the global properties and functions defined by core JavaScript. See `Global` in Part III for details.

Web browsers fire many kinds of events at windows. This means that the `Window` object defines quite a few event handlers, and that `Window` objects implement the methods defined by `EventTarget`.

The `Window` object has `window` and `self` properties that refer to the window object itself. You can use these to make the current window reference explicit rather than implicit.

A `Window` can contain other `Window` objects, typically in the form of `<iframe>` tags. Each `Window` is an array-like object of nested `Window` objects. Rather than indexing a `Window` object directly, however, you typically use its self-referential `frames` property as if it were the array-like object. The `parent` and `top` properties of a `Window` refer to the directly containing window and top-level ancestor window.

New top-level browser windows are created with the `window.open()` method. When you call this method, save the return value of the `open()` call in a variable and use that variable to reference the new window. The `opener` property of the new window is a reference back to the window that opened it.

Properties

In addition to the properties listed here, document content displayed within the window causes new properties to come into existence. As explained in §14.7, you can refer to an element within the document by using the value of the element’s `id` attribute as a property of the window (and since the window is the global object, its properties are global variables).

readonly ApplicationCache applicationCache

A reference to the `ApplicationCache` object. Cached and offline web apps can use this object to manage their cache updates.

readonly any dialogArguments

In Window objects created by the `showModalDialog()`, this property is the *arguments* value that was passed to `showModalDialog()`. In regular Window objects, this property does not exist. See §14.5 for more information.

readonly Document document

The Document object that describes the content of this window (see Document for details).

readonly Event event [IE only]

In Internet Explorer, this property refers to the Event object that describes the most recent event. In IE8 and before, the event object is not always passed to the event handler and must sometimes be accessed through this property. See Chapter 17 for further details.

readonly Element frameElement

If this Window is within an `<iframe>`, this property refers to that `IFrame` element. For top-level windows, this property is `null`.

readonly Window frames

This property, like the `self` and `window` properties, refers to the Window object itself. Every Window object is an array-like object of the frames contained within it. Instead of writing `w[0]` to refer to the first frame within a window `w`, this property allows you to more clearly write `w.frames[0]`.

readonly History history

The History object of this window. See History.

readonly long innerHeight**readonly long innerWidth**

The height and width, in pixels, of the document display area of this window. These properties are not supported in IE8 and before. See Example 15-9 for an example.

readonly unsigned long length

The number of frames contained in this window. See frames.

readonly Storage localStorage

This property refers to a `Storage` object that provides client-side storage for name/value pairs. Data stored through `localStorage` is visible to and shared with any documents with the same origin, and persists until deleted by the user or by a script. See also `sessionStorage` and §20.1.

readonly Location location

The Location object for this window. This object specifies the URL of the currently loaded document. Setting this property to a new URL string causes the browser to load and display the contents of that URL. See Location.

string name

The name of the window. The name is optionally specified when the window is created with the `open()` method or with the `name` attribute of a `<frame>` tag. The name of a window

can be used as the value of a `target` attribute of an `<a>` or `<form>` tag. Using the `target` attribute in this way specifies that the hyperlinked document or the results of form submission should be displayed in the named window or frame.

readonly Navigator navigator

A reference to the Navigator object, which provides version and configuration information about the web browser. See Navigator.

readonly Window opener

A read/write reference to the Window object that contained the script that called `open()` to open this browser window, or `null` for windows that were not created in that way. This property is valid only for Window objects that represent top-level windows, not those that represent frames. The `opener` property is useful so that a newly created window can refer to properties and functions defined in the window that created it.

readonly long outerHeight

readonly long outerWidth

These properties specify the total height and width, in pixels, of the browser window, including toolbars, scrollbars, window borders, and so on. These properties are not supported in IE8 and before.

readonly long pageXOffset

readonly long pageYOffset

The number of pixels that the current document has been scrolled to the right (`pageXOffset`) and down (`pageYOffset`). These properties are not supported in IE8 and before. See Example 15-8 for an example and compatibility code that works in IE.

readonly Window parent

The Window object that contains this one. If this window is a top-level window, `parent` refers to the window itself. If this window is a frame, the `parent` property refers to the window or frame that contains it.

string returnValue

This property does not exist in normal windows, but it is defined for Windows created by `showModalDialog()` and has the empty string as its default value. When a dialog Window is closed (see the `close()` method), the value of this property becomes the return value of `showModalDialog()`.

readonly Screen screen

The Screen object that specifies information about the screen: the number of available pixels and the number of available colors. See Screen for details.

readonly long screenX

readonly long screenY

The coordinates of the upper left corner of the window on the screen.

readonly Window self

A reference to this window itself. This is a synonym for the `window` property.

readonly Storage sessionStorage

This property refers to a Storage object that provides client-side storage for name/value pairs. Data stored through `sessionStorage` is visible only to same-origin documents with-

in the same top-level window or tab, and persists only for the duration of the browsing session. See also `localStorage` and §20.1.

readonly Window **top**

The top-level window that contains this window. If this window is a top-level window itself, the `top` property simply refers to the window itself. If this window is a frame, the `top` property refers to the top-level window that contains the frame. Contrast with the `parent` property.

readonly object **URL**

At the time of this writing, this property is simply a reference to a placeholder object that defines the functions documented at `URL`. In the future, this property may become a `URL()` constructor and define an API for parsing URLs and their query strings.

readonly Window **window**

The `window` property is identical to the `self` property; it contains a reference to this window. Since the `Window` object is the global object of client-side JavaScript, this property allows us to write `window` to refer to the global object.

Constructors

As the global object for client-side JavaScript, the `Window` object must define all the global constructors for the client-side environment. Although they are not listed here, all of the global constructors documented in this reference section are properties of the `Window` object. The fact that client-side JavaScript defines `Image()` and `XMLHttpRequest()` constructors, for example, means that every `Window` object has properties named `Image` and `XMLHttpRequest`.

Methods

The `Window` object defines the following methods and also inherits all the global functions defined by core JavaScript (see `Global` in Part III).

void `alert(string message)`

The `alert()` method displays the specified plain-text *message* to the user in a dialog box. The dialog box contains an OK button the user can click to dismiss it. The dialog box is typically modal (at least for the current tab), and the call to `alert()` blocks until the dialog is dismissed.

string `atob(string atob)`

This utility function accepts a base64-encoded string and decodes it into a JavaScript binary string in which each character represents a single byte. Use the `charCodeAt()` method of the returned string to extract byte values. See also `btoa()`.

void `blur()`

The `blur()` method removes keyboard focus from the top-level browser window specified by the `Window` object. It is unspecified which window gains keyboard focus as a result. In some browsers and/or platforms, this method may have no effect.

string `btoa(string btoa)`

This utility function accepts a JavaScript binary string (in which each character represents a single byte) as an argument and returns the base64 encoding of it. Use `String.fromCharCode()` to create a binary string from an arbitrary sequence of byte values. See also `atob()`.

void clearInterval(long handle)

clearInterval() stops the repeated execution of code that was started by a call to setInterval(). *intervalId* must be the value that was returned by a call to setInterval().

void clearTimeout(long handle)

clearTimeout() cancels the execution of code that has been deferred with the setTimeout() method. The *timeoutId* argument is a value returned by the call to setTimeout() and identifies which deferred code to cancel.

void close()

The close() method closes the top-level browser window on which it is invoked. Scripts are generally only allowed to close windows that they opened.

boolean confirm(string message)

This method displays the specified *question* as plain text in a modal dialog box. The dialog box contains OK and Cancel buttons that the user can use to answer the question. If the user clicks the OK button, confirm() returns true. If the user clicks Cancel, confirm() returns false.

void focus()

This method gives keyboard focus to the browser window. On most platforms, a top-level window is brought forward to the top of the window stack so that it becomes visible when it is given focus.

CSSStyleDeclaration getComputedStyle(Element elt, [string pseudoElt])

An element in a document can obtain style information from an inline style attribute and from any number of stylesheets in the stylesheet “cascade.” Before the element can actually be displayed in a window, its style information must be extracted from the cascade, and styles specified with relative units (such as percentages or “ems”) must be “computed” to convert to pixels. These computed values are sometimes also called “used” values.

This method returns a read-only CSSStyleDeclaration object that represents the CSS style values actually used to display the element. All dimensions will be in pixels.

The second argument to this method is usually omitted or null, but you can also pass the CSS pseudoelement “::before” or “::after” to determine the styles used for CSS-generated content.

Contrast getComputedStyle() with the style property of an HTMLElement, which gives you access only to the inline styles of an element, in whatever units they were specified, and tells you nothing about stylesheet styles that apply to the element.

This method is not implemented in IE8 and before, but similar functionality is available through the nonstandard *currentStyle* property of each HTMLElement object.

Window open([string url], [string target], [string features], [string replace])

The open() method loads and displays the specified *url* in a new or existing browser window or tab. The *url* argument specifies the URL of the document to load. If not specified, “about:blank” is used.

The *target* argument specifies the name of the window into which the *url* should be loaded. If not specified, “_blank” is used. If *target* is “_blank”, or if there is no existing window with

the specified name, a new window is created to display the contents of *url*. Otherwise, the *url* is loaded into the existing window with the specified name.

The *features* argument used to specify the window position, size, and features (such as menubar, toolbar, etc.). In modern browsers that support tabs, it is often ignored, and it is not documented here.

When you use `Window.open()` to load a new document into an existing window, the *replace* argument specifies whether the new document has its own entry in the window's browsing history or whether it replaces the history entry of the current document. If *replace* is `true`, the new document replaces the old. If this argument is `false` or is not specified, the new document has its own entry in the Window's browsing history. This argument provides functionality much like that of the `Location.replace()` method.

void `postMessage`(any *message*, string *targetOrigin*, [MessagePort[] *ports*])

Send a copy of the specified *message* and the optionally specified *ports* to this window, but only if the document displayed in this window has the specified *targetOrigin*.

message can be any object that can be cloned with the structured clone algorithm (see "Structured Clones" on page 672). *targetOrigin* should be an absolute URL that specifies the scheme, host, and port of the desired origin. Or *targetOrigin* can be `""` if any origin is acceptable or `"/"` to use the script's own origin.

Calling this method on a window causes a message event on that window. See `MessageEvent` and §22.3.

void `print`()

Calling `print()` causes the browser to behave as if the user had selected the browser's Print button or menu item. Usually, this brings up a dialog box that enables the user to cancel or customize the print request.

string `prompt`(string *message*, [string *default*])

The `prompt()` method displays the specified *message* in a modal dialog box that also contains a text input field and OK and Cancel buttons and blocks until the user clicks one of the buttons.

If the user clicks the Cancel button, `prompt()` returns `null`. If the user clicks the OK button, `prompt()` returns the text currently displayed in the input field.

The *default* argument specifies the initial value of the text input field

void `scroll`(long *x*, long *y*)

This method is a synonym for `scrollTo()`.

void `scrollBy`(long *x*, long *y*)

`scrollBy()` scrolls the document displayed in *window* by the relative amounts specified by *dx* and *dy*.

void `scrollTo`(long *x*, long *y*)

`scrollTo()` scrolls the document displayed within *window* so the point in the document specified by the *x* and *y* coordinates is displayed in the upper left corner, if possible.

long setInterval(function *f*, unsigned long *interval*, any *args*...)

`setInterval()` registers the function *f* to be invoked after *interval* milliseconds and then to be repeatedly invoked at that specified *interval*. *f* will be invoked with the Window as its this value, and will be passed any additional *args* that were passed to `setInterval()`.

`setInterval()` returns a number that can later be passed to `Window.clearInterval()` to cancel the execution of *code*.

For historical reasons, *f* may be a string of JavaScript code instead of a function. If so, the string will be evaluated (as if it were a `<script>`) every *interval* milliseconds.

Use `setTimeout()` when you want to defer the execution of code but do not want it to be repeatedly executed.

long setTimeout(function *f*, unsigned long *timeout*, any *args*...)

`setTimeout()` is like `setInterval()`, except that it invokes the specified function only once: it registers *f* to be invoked after *timeout* milliseconds have been elapsed and returns a number that can later be passed to `clearTimeout()` to cancel the pending invocation. When the specified time has passed, *f* will be invoked as a method of the Window and will be passed any specified *args*. If *f* is a string rather than a function, it will be executed after *timeout* milliseconds as if it were a `<script>`.

any showModalDialog(string *url*, [any *arguments*])

This method creates a new Window object, sets its `dialogArguments` property to *arguments*, loads *url* into the window, and blocks until the window is closed. Once closed, it returns the `returnValue` property of the window. See §14.5 and Example 14-4 for a discussion and example.

Event Handlers

Most events that occur on HTML elements bubble up the document tree, to the Document object and then on to the Window object. For this reason, you can use all of the event handler properties listed in Element on Window objects. And in addition, you can use the event handler properties listed below. For historical reasons, each of the event handler properties listed here can also be defined (as an HTML attribute or as a JavaScript property) on the `<body>` element.

Event Handler	Invoked...
<code>onafterprint</code>	After the window's contents are printed
<code>onbeforeprint</code>	Before the window's contents are printed
<code>onbeforeunload</code>	Before navigating away from the current page. If the return value is a string, or if the handler sets the <code>returnValue</code> property of its event object to a string, that string will be displayed in a confirmation dialog. See <code>BeforeUnloadEvent</code> .
<code>onblur</code>	When the window loses keyboard focus
<code>onerror</code>	When a JavaScript error occurs. This is not an ordinary event handler. See §14.6.
<code>onfocus</code>	When the window gains keyboard focus
<code>onhashchange</code>	When the fragment identifier (see <code>Location.hash</code>) of the document changes as the result of history navigation (see <code>HashChangeEvent</code>)

Event Handler	Invoked...
<code>onload</code>	When the document and its external resources are fully loaded
<code>onmessage</code>	When a script in another window sends a message by calling the <code>postMessage()</code> method. See <code>MessageEvent</code> .
<code>onoffline</code>	When the browser loses its connection to the Internet
<code>ononline</code>	When the browser regains a connection to the Internet
<code>onpagehide</code>	When the page is about to be cached and replaced by another page
<code>onpageshow</code>	When a page is first loaded, a <code>pageshow</code> event is fired right after the <code>load</code> event, and the event object has a persisted property of <code>false</code> . When a previously loaded page is restored from the browser's in-memory cache, however, no <code>load</code> event is fired (since the cached page is already in its loaded state) and a <code>pageshow</code> event is fired with an event object that has its persisted property set to <code>true</code> . See <code>PageTransitionEvent</code> .
<code>onpopstate</code>	When the browser loads a new page or restores a state saved with <code>History.pushState()</code> or <code>History.replaceState()</code> . See <code>PopStateEvent</code> .
<code>onresize</code>	When the user changes the size of the browser window
<code>onscroll</code>	When the user scrolls the browser window
<code>onstorage</code>	Content of <code>localStorage</code> or <code>sessionStorage</code> changes. See <code>StorageEvent</code> .
<code>onunload</code>	The browser navigates away from a page. Note that if you register an <code>onunload</code> handler for a page, that page will not be cacheable. To allow users to quickly return to your page without reloading, use <code>onpagehide</code> instead.

Worker

a worker thread

EventTarget

A Worker represents a background thread. Create a new Worker with the `Worker()` constructor, passing the URL of a file of JavaScript code for it to run. The JavaScript code in that file can use synchronous APIs or perform compute-intensive tasks without freezing up the main UI thread. Workers run their code in a completely separate execution context (see `WorkerGlobalScope`), and the only way to exchange data with a worker is via asynchronous events. Call `postMessage()` to send data to the Worker, and handle message events to receive data from the worker.

See §22.4 for an introduction to worker threads.

Constructor

```
new Worker(string scriptURL)
```

Constructs a new Worker object and causes it to run the JavaScript code in `scriptURL`.

Methods

```
void postMessage(any message, [MessagePort[] ports])
```

Send `message` to the worker, which will receive it as a `MessageEvent` object sent to its `onmessage` handler. `message` can be a JavaScript primitive value or object or array, but not a function.

Client-side types `ArrayBuffer`, `File`, `Blob`, and `ImageData` are allowed, but Nodes, such as `Document` and `Element`, are not allowed (see “Structured Clones” on page 672 for details).

The optional `ports` argument is an advanced feature that allows you to pass one or more direct communication channels to the Worker. If you create two Worker objects, for example, you can allow them to communicate directly with each other by passing them each one end of a `MessageChannel`.

`void terminate()`

Stop the worker and abort the script it is running.

Event Handlers

Because workers run code in a completely separate execution environment than the one that created them, the only way they can communicate with their parent thread is by events. You can register event handlers on these properties or use the `EventTarget` methods.

`onerror`

When an exception is thrown in the script being run by a Worker, and that error is not handled by the `onerror` handler of the `WorkerGlobalScope`, the error triggers an error event on the Worker object. The event object associated with this event is a `ErrorEvent`. The error event does not bubble. If this worker is owned by another worker, canceling an error event prevents it from being propagated up to the parent worker. If this Worker object is already in the main thread, canceling the event may prevent it from being displayed in the JavaScript console.

`onmessage`

When the script that the worker is running calls its global `postMessage()` function (see `WorkerGlobalScope`), a message event is triggered on the Worker object. The object passed to the event handler is a `MessageEvent`, and its `data` property contains a clone of the value that the worker script passed to `postMessage()`.

WorkerGlobalScope

`EventTarget`, `Global`

A Worker thread runs in a completely different execution environment than the parent thread that spawned it. The global object for a worker is a `WorkerGlobalScope` object, so this page describes the execution environment “inside” a Worker. Since `WorkerGlobalScope` is a global object, it inherits the `Global` object of core JavaScript.

Properties

In addition to the properties listed here, `WorkerGlobalScope` also defines all of the core JavaScript global properties, such as `Math` and `JSON`.

`readonly WorkerLocation location`

This property is like the `window.location` `Location` object: it allows a worker to inspect the URL it was loaded from and includes properties that return individual portions of the URL.

readonly WorkerNavigator navigator

This property is like the `window.navigator` Navigator object: it defines properties that allow a worker to determine what browser it is running in and whether it is currently online or not.

readonly WorkerGlobalScope self

This self-referential property refers to the WorkerGlobalScope global object itself. It is like the `window` property of the Window object in the main thread.

Methods

In addition to the properties listed here, WorkerGlobalScope also defines all of the core JavaScript global functions, such as `isNaN()` and `eval()`.

void clearInterval(long handle)

This method is just like the Window method of the same name.

void clearTimeout(long handle)

This method is just like the Window method of the same name.

void close()

This method puts the worker into a special “closing” state. Once in this state, it will not fire any timers or trigger any events. The script continues running until it returns to the worker’s event loop, at which point the worker stops.

void importScripts(string urls...)

For each of the specified *urls*, this method resolves the URL relative to the worker’s location, then loads the content of the URL, and then executes that content as JavaScript code. Note that this is a synchronous method. It loads and executes each file in turn and does not return until all scripts have executed. (If any script throws an exception, however, that exception will propagate and prevent any subsequent URLs from being loaded and executed.)

void postMessage(any message, [MessagePort[] ports])

Sends a *message* (and optionally an array of *ports*) to the thread that spawned this worker. Calling this method causes a message event to be triggered on the Worker object in the parent thread, and the associated MessageEvent object will include a clone of *message* as its *data* property. Note that in a worker, `postMessage()` is a global function.

long setInterval(any handler, [any timeout], any args...)

This method is just like the Window method of the same name.

long setTimeout(any handler, [any timeout], any args...)

This method is just like the Window method of the same name.

Constructors

WorkerGlobalScope includes all of the core JavaScript constructors, such as `Array()`, `Date()`, and `RegExp()`. It also defines key client-side constructors for `XMLHttpRequest`, `FileReaderSync`, and even the Worker object itself.

Event Handlers

You can register event handlers for workers by setting these global event handler properties, or you can use the `EventTarget` methods implemented by `WorkerGlobalScope`.

`onerror`

This is not a normal event handler; it is like the `onerror` property of `Window` rather than the `onerror` property of `Worker`. When an unhandled exception occurs in the worker, this function, if defined, will be invoked with three string arguments that specify an error message, a script URL, and a line number. If the function returns `false`, the error is considered handled and does not propagate. Otherwise, if this property is not set, or if the error handler does not return `false`, the error propagates and causes an error event on the `Worker` object in the parent thread.

`onmessage`

When the parent thread calls the `postMessage()` method of the `Worker` object that represents this worker, it causes a message event to be triggered on this `WorkerGlobalScope`. This event handler function will be passed a `MessageEvent` object, and the `data` property of that object will hold a clone of the `message` argument sent by the parent thread.

WorkerLocation

the URL of a worker's main script

The `WorkerLocation` object referenced by the `location` property of a `WorkerGlobalScope` is like the `Location` object referenced by the `location` property of a `Window`: it represents the URL of the worker's main script and defines properties that represent portions of that URL.

Workers differ from Windows in that they cannot be navigated or reloaded, so the properties of a `WorkerLocation` object are read-only, and the object does not implement the methods of the `Location` object.

The `WorkerLocation` object does not automatically convert to a string the way a regular `location` object does. In a worker, you cannot simply write `location` when you mean `location.href`.

Properties

These properties have the same meanings as the same-named properties of the `Location` object.

`readonly string hash`

The fragment identifier portion of the URL, including the leading hash mark.

`readonly string host`

The host and port portions of the URL.

`readonly string hostname`

The host portion of the URL.

readonly string href

The complete text of the URL that was passed to the `Worker()` constructor. This is the only value that the worker receives directly from its parent thread: all other values are received indirectly through message events.

readonly string pathname

The pathname portion of the URL.

readonly string port

The port portion of the URL.

readonly string protocol

The protocol portion of the URL.

readonly string search

The search or query portion of the URL, including the leading question mark.

WorkerNavigator

browser information for workers

The `navigator` property of a `WorkerGlobalScope` refers to a `WorkerNavigator` object that is a simplified version of the `Navigator` object of a `Window`.

Properties

Each of these properties has the same meaning that it does in the `Navigator` object.

readonly string appName

See the `appName` property of `Navigator`.

readonly string appVersion

See the `appVersions` property of `Navigator`.

readonly boolean onLine

true if the browser is online and false if it is not.

readonly string platform

A string that identifies the operating system and/or hardware platform on which the browser is running.

readonly string userAgent

The value the browser uses for the user-agent header in HTTP requests.

XMLHttpRequest

An HTTP request and response

EventTarget

The `XMLHttpRequest` object allows client-side JavaScript to issue HTTP requests and receive responses (which need not be XML) from web servers. `XMLHttpRequest` is the subject of Chapter 18, and that chapter contains many examples of its use.

Create an `XMLHttpRequest` object with the `XMLHttpRequest()` constructor (see the sidebar in §18.1 for information on how to create an `XMLHttpRequest` object in IE6) and then use it like this:

1. Call `open()` to specify the URL and method (usually “GET” or “POST”) for the request.
2. Set the `onreadystatechange` property to the function that will be notified of the progress of the request.
3. Call `setRequestHeader()`, if needed, to specify additional request parameters.
4. Call `send()` to send the request to the web server. If it is a POST request, you can also pass a request body to this method. Your `onreadystatechange` event handler function will be invoked as the request proceeds. When `readyState` is 4, the response is complete.
5. When `readyState` is 4, check the status code to ensure that the request was successful. If so, use `getResponseHeader()` or `getResponseHeaders()` to retrieve values from the response header, and use the `responseText` or `responseXML` properties to obtain the response body.

`XMLHttpRequest` defines a relatively high-level interface to the HTTP protocol. It takes care of details such as handling redirects, managing cookies, and negotiating cross-origin connections with CORS headers.

The `XMLHttpRequest` features described above are well supported by all modern browsers. At the time of this writing, an `XMLHttpRequest` Level 2 standard is under development and browsers are beginning to implement it. The properties, methods, and event handlers listed below include `XMLHttpRequest` Level 2 features, which may not yet be implemented by all browsers. These newer features are marked “XHR2.”

Constructor

```
new XMLHttpRequest()
```

This no-argument constructor returns a new `XMLHttpRequest` object.

Constants

These constants define the values of the `readyState` property. Prior to XHR2, these constants are not widely defined, and most code uses integer literals rather than these symbolic values.

```
unsigned short UNSENT = 0
```

This is the initial state. The `XMLHttpRequest` object has just been created or has been reset with the `abort()` method.

```
unsigned short OPENED = 1
```

The `open()` method has been called, but `send()` has not. The request has not yet been sent.

```
unsigned short HEADERS_RECEIVED = 2
```

The `send()` method has been called, and the response headers have been received, but the response body has not yet been received.

```
unsigned short LOADING = 3
```

The response body is being received but is not complete.

unsigned short **DONE** = 4

The HTTP response has been fully received or has stopped because of an error.

Properties

readonly unsigned short **readyState**

The state of the HTTP request and the server's response. The value of this property begins at 0 when an XMLHttpRequest is first created and increases to 4 when the complete HTTP response has been received. The constants listed above define the possible values.

The value of **readyState** never decreases, unless **abort()** or **open()** is called on a request that is already in progress.

In theory, a **readystatechange** event is dispatched every time the value of this property changes. In practice, however, an event is only really guaranteed when **readyState** changes to 4. (The XHR2 progress events provide a more reliable way of tracking the progress of a request.)

readonly any **response**

In XHR2, this property holds the server's response. Its type depends on the **responseType** property. If **responseType** is the empty string or "text", property will hold the response body as a string. If **responseType** is "document", this property will be a parsed representation of the response body as an XML or HTTP Document. If **responseType** is "arraybuffer", this property will be an **ArrayBuffer** that represents the bytes of the response body. And if **responseType** is "blob", this property will be a **Blob** that represents the bytes of the response body.

readonly string **responseText**

If **readyState** is less than 3, this property is the empty string. When **readyState** is 3, this property returns whatever portion of the response has been received so far. If **readyState** is 4, this property holds the complete body of the response.

If the response includes headers that specify a character encoding for the body, that encoding is used. Otherwise, the Unicode UTF-8 encoding is assumed.

string **responseType**

In XHR2, this property specifies the desired response type and determines the type of the response property. The legal values are "text", "document", "arraybuffer", and "blob". The default is the empty string, which is a synonym for "text". If you set this property, the **responseText** and **responseXML** properties will raise exceptions and you must use the XHR2 **response** property to get the server's response.

readonly Document **responseXML**

The response to the request, parsed as an XML or HTML Document object, or null if the response body is not ready or is not a valid XML or HTML document.

readonly unsigned short **status**

The HTTP status code returned by the server, such as 200 for success, 404 for "Not Found" errors, or 0 if the server has not set a status code yet.

readonly string `statusText`

This property specifies the HTTP status code of the request by name rather than by number. That is, it is “OK” when `status` is 200 and “Not Found” when `status` is 404. This property is the empty string if the server has not set a status code yet.

unsigned long `timeout`

This XHR2 property specifies a timeout value in milliseconds. If the HTTP request takes longer than this to complete, it will be aborted and `timeout` event will be triggered. You can only set this property after calling `open()` and before calling `send()`.

readonly XMLHttpRequestUpload `upload`

This XHR2 property refers to an `XMLHttpRequestUpload` object that defines a set of event handler registration properties for monitoring the progress of the HTTP request body upload.

boolean `withCredentials`

This XHR2 property specifies whether authentication credentials should be included in CORS requests and whether cookie headers in CORS responses should be processed. The default value is `false`.

Methods**void `abort()`**

This method resets the `XMLHttpRequest` object to a `readyState` of 0 and aborts any pending network activity. You might call this method, for example, if a request has taken too long, and the response is no longer necessary.

string `getAllResponseHeaders()`

This method returns the HTTP response headers (with cookie and CORS headers filtered out) sent by the server, or null if the headers have not been received yet. The headers are returned as a single string, with one header per line.

string `getResponseHeader(string header)`

Returns the value of a named HTTP response *header*, or null if headers have not been received yet or if the response does not include the specified *header*. Cookie and CORS-related headers are filtered out and cannot be queried. If the response includes more than one header with the specified name, the returned string will include the value of all of those headers, concatenated and separated by a comma and a space.

void `open(string method, string url, [boolean async, string user, string pass])`

This method resets the `XMLHttpRequest` object and stores its arguments for later use by the `send()`.

method is the HTTP method to be used for the request. Reliably implemented values include GET, POST, and HEAD. Implementations may also support the CONNECT, DELETE, OPTIONS, PUT, TRACE, and TRACK methods.

url is the URL being requested. Relative URLs are resolved in the normal way, using the URL of the document that contains the script. The same-origin security policy (see §13.6.2)

requires that this URL have the same hostname and port as the document that contains the script making the request. XHR2 allows cross-origin requests to servers that support CORS.

If the *async* argument is specified and is *false*, the request is performed synchronously and the *send()* method will block until the response is complete. This is not recommended except when XMLHttpRequest is used in a *Worker*.

The optional *user* and *pass* arguments specify a username and password to use with the HTTP request.

void overrideMimeType(string mime)

This method specifies that the server's response should be interpreted according to the specified *mime* type (and *charset* parameter, if that is included) instead of using the *Content-Type* header of the response.

void send(any body)

This method causes an HTTP request to be issued. If there has been no previous call to *open()*, or, more generally, if *readyState* is not 1, *send()* throws an exception. Otherwise, it issues an HTTP request that consists of:

- The HTTP method, URL, and authorization credentials (if any) specified in the previous call to *open()*.
- The request headers, if any, specified by previous calls to *setRequestHeader()*.
- The *body* argument passed to this method. The *body* may be a string or a Document object that specifies the request body, or may be omitted or *null* if the request has no body (such as GET requests that never have a body). In XHR2, the body may also be an *ArrayBuffer*, a *Blob*, or a *FormData* object.

If the *async* argument to the previous call to *open()* was *false*, this method blocks and does not return until *readyState* is 4 and the server's response has been fully received. Otherwise, *send()* returns immediately, and the server's response is processed asynchronously with notifications provided through event handlers.

void setRequestHeader(string name, string value)

setRequestHeader() specifies an HTTP request header *name* and *value* that should be included in the request issued by a subsequent call to *send()*. This method may be called only when *readyState* is 1—i.e., after a call to *open()*, but before a call to *send()*.

If a header with the specified *name* has already been specified, the new value for that header is the previously specified value, plus a comma, a space, and the *value* specified in this call.

If the call to *open()* specifies authorization credentials, XMLHttpRequest automatically sends an appropriate *Authorization* request header. You can also append to this header yourself with *setRequestHeader()*, however.

XMLHttpRequest automatically sets “Content-Length”, “Date”, “Referer”, and “User-Agent” and does not allow you to spoof them. There are a number of other headers, including cookie-related headers, that you cannot set with this method. The complete list is in §18.1.

Event Handlers

The original XMLHttpRequest object defined only a single event handler property: `onreadystatechange`. XMLHttpRequest2 expands the list with a set of progress event handlers that are much easier to use. You can register handlers by setting these properties or by using the methods of `EventTarget`. XMLHttpRequest events are always dispatched on the XMLHttpRequest object itself. They do not bubble and have no default action to cancel. `readystatechange` events have an associated Event object, and all of the other event types have an associated `ProgressEvent` object.

See the `upload` property and `XMLHttpRequestUpload` for a list of events you can use to monitor the progress of HTTP uploads.

`onabort`

Triggered when a request is aborted.

`onerror`

Triggered if the request fails with an error. Note that HTTP status codes such as 404 do not constitute an error since the response still completes successfully. A DNS failure while trying to resolve the URL or an infinite loop of redirects would both cause this event to occur, however.

`onload`

Triggered when the request completes successfully.

`onloadend`

Triggered when the request has succeeded or failed after the load, abort, error, or timeout event.

`onloadstart`

Triggered when the request starts.

`onprogress`

Triggered repeatedly (approximately every 50ms) while the response body is downloading.

`onreadystatechange`

Triggered when the `readyState` property changes, most importantly when the response is complete.

`ontimeout`

Triggered if the time specified by the `timeout` property has elapsed and the response is not complete.

XMLHttpRequestUpload

EventTarget

An XMLHttpRequestUpload object defines a set of event handler registration properties for monitoring the progress of an HTTP request body upload. In browsers that implement the XMLHttpRequest Level 2 specification, each XMLHttpRequest object has an `upload` property that refers to an object of this type. To monitor the progress of the request upload, simply set

these properties to appropriate event handler functions or call the `EventTarget` methods. Note that the upload progress event handlers defined here are exactly the same as the download progress event handlers defined on `XMLHttpRequest` itself, except that there is no `onreadyStatechange` property on this object.

Event Handlers

`onabort`

Triggered if the upload is aborted.

`onerror`

Triggered if the upload fails with a network error.

`onload`

Triggered when the upload succeeds

`onloadend`

Triggered when the upload finishes, whether successfully or not. A `loadend` event will always follow a `load`, `abort`, `error`, or `timeout` event.

`onloadstart`

Triggered when the upload starts.

`onprogress`

Triggered repeatedly (approximately every 50ms) while the upload is occurring.

`ontimeout`

Triggered if the upload is aborted because the `XMLHttpRequest` `timeout` expired.

Symbols

3D graphics for <canvas> element, 631, 688

& (ampersand)

- && (logical AND) operator, 41, 62, 75
- short-circuiting behavior of, 123

- &= (bitwise AND and assignment) operator, 62, 78

- bitwise AND operator, 62, 69

< > (angle brackets)

- < (less than) operator, 62, 73
- substituting compareTo() method, 223

- << (bitwise left shift) operator, 62, 70

- <= (bitwise left shift and assignment) operator, 62, 78

- <= (less than or equal to) operator, 62, 73
- substituting compareTo() method, 223

- > (greater than) operator, 62, 73
- substituting compareTo() method, 223

- >= (greater than or equal to) operator, 62, 73

- substituting compareTo() method, 223

- >> (bitwise right shift with sign extension) operator, 62, 70

- >>= (bitwise right shift with sign extension and assignment) operator, 62, 78

- >>> (bitwise right shift with zero fill) operator, 62, 70

- >>>= (bitwise right shift with zero fill and assignment) operator, 62, 78

* (asterisk)

- *= (multiplication and assignment) operator, 62, 78

- matching zero or more occurrences in regular expressions, 254

- multiplication operator, 33, 62

- wildcard character in E4X, 285

@ (at sign)

- @if, @else, and @end keywords in conditional comments, 331

- in attribute names, 285

\ (backslash)

- breaking multiline string literals, 37

- escape sequences in string literals, 37

- escaping special characters in regular expressions, 253

^ (caret)

- beginning-of-string matching in regular expressions, 258

- bitwise XOR operator, 62, 70

- negating character classes in regular expressions, 253

- ^= (bitwise XOR and assignment) operator, 62, 78

, (comma) operator, 85

{ } (curly braces)

- around function body, omitting in shorthand functions, 282

- enclosing object initializer expressions, 59

- enclosing statement blocks, 88

- escape characters in XML literal syntax of E4X, 284

- in function definitions, 164

- initializer expressions in, 5

\$ (dollar sign)

- \$\$() method, ConsoleCommandLine, 884

- \$() function, 11, 524, 525–528

- looking up elements by ID, 352

- using instead of querySelectorAll(), 529

- \$() method, ConsoleCommandLine, 884

- \$0 and \$1 properties,
 - ConsoleCommandLine, 884
- \$1, \$2 ... \$n variables in regular expressions, 260
- end-of-string matching in regular expressions, 252, 258
- (dot)
 - .. (descendant) operator, 285
 - dot operator, 5
 - property access in method invocations, 168
 - property access with, secure subsets and, 268
 - querying and setting properties, 120
 - matching any character except newlines in regular expressions, 254
 - in property access expressions, 60
- = (equals sign)
 - == (equality) operator, 62, 71
 - NaN values and, 34
 - object-to-primitive conversions, 51
 - substituting compareTo() method, 223
 - == and === operators
 - comparing null and undefined values, 42
 - comparing primitive values and wrapper objects, 44
 - type conversions and equality, 47
 - === (strict equality) operator, 62, 71
 - =? in URL or data string passed to jQuery.getJSON(), 563
- assignment operator, 62
- ! (exclamation mark)
 - != (inequality) operator, 62, 71
 - NaN comparisons, 34
 - substituting compareTo() method, 223
 - !== (strict inequality) operator, 62, 71
 - comparing variable to null, 40
 - testing for undefined properties, 125
 - logical NOT operator, 48, 62, 77
- (minus sign)
 - (decrement) operator, 62, 69
 - line breaks in statements and, 27
 - side effects, 64, 88
 - = (subtraction and assignment) operator, 78
 - Infinity (negative infinity), 33
 - negation operator, 62
 - statements beginning with, 26
 - subtraction operator, 33, 62
 - unary minus operator, 68
- () (parentheses)
 - enclosing expressions in if statements, 92
 - grouping in regular expressions, 256
 - in function and method invocations, 61
 - in function definitions, 59, 164
 - in generator expressions, 282
 - in object creation expressions, 61
 - statements beginning with (, 26
- % (percent sign)
 - %= (modulus and assignment) operator, 62, 78
 - modulus operator, 33, 62
- . (period) (see . (dot))
- + (plus sign)
 - ++ (increment) operator, 62, 68
 - line breaks in statements and, 27
 - side effects, 64, 88
 - += (addition and assignment) operator, 62, 78
 - appending text to innerHTML property, 379
 - addition and string concatenation operators, 67
 - object-to-primitive conversions, 51
 - addition operator, 33, 62
 - matching one or more occurrences in regular expressions, 254
 - statements beginning with, 26
 - string concatenation operator, 62
 - unary plus operator, 68
- ? (question mark)
 - ?! (negative lookahead assertion) in regular expressions, 258
 - ?: (conditional) operator, 62, 82
 - ?= (positive lookahead assertions) in regular expressions, 258
 - nongreedy repetition in regular expressions, 255
- "" (quotation marks, double) enclosing string literals, 36
 - quoting in stylesheet or style attribute strings, 432
- ' ' (quotation marks, single) enclosing string literals, 36
 - single-quoted HTML attribute, JavaScript code in, 317
- ;(semicolon)

- ending statements, 6, 87
- optional, ending statements, 25
- placing outside of style attribute or stylesheet strings, 432
- separating name-value pairs in style properties, 414
- statement blocks and, 89
- / (slash)
 - /* */ comments in javascript; URL code, 315
 - /*@cc_on and @*/ in conditional comments, 331
 - // and /**/ in comments, 23
 - // in comments, 4
 - /= (division and assignment) operator, 62, 78
- division operator, 33, 62
- enclosing regular expression literals, 251
- statements beginning with, 26
- [] (square brackets)
 - accessing array elements, 143
 - accessing individual string characters, 160
 - accessing values in multidimensional arrays, 148
- array operator, 5, 688
 - accessing individual string characters, 39
- creating arrays, 141
- enclosing array comprehensions, 281
- enclosing array initializers, 58
- enclosing character classes in regular expressions, 253
- in property access expressions, 60, 168
 - restriction in secure subsets, 268
- querying and setting properties, 120, 121
- statements beginning with [, 26
- ~ (tilde), bitwise NOT operator, 62, 70
- _ (underscore), in function names, 165
- | (vertical bar)
 - alternation in regular expression pattern matching, 256
- bitwise OR operator, 62, 70
- |= (bitwise OR and assignment) operator, 62, 78
- || (logical OR) operator, 62, 76, 171
- having name attribute rather than href attribute, 367
- href attribute, 315, 367
- abort() method
 - FileReader object, 925
 - XMLHttpRequest object, 507, 510
- about:blank URL, 354
- abs() function, Math object, 791
- absolute positioning of elements, 420
- abstract classes, 228
 - class hierarchies and, 234–238
- abstract methods, 228
- accept property, Input object, 942
- acceptCharset property, Form object, 928
- Access-Control-Allow-Origin response header, 335
- accessibility, 332
- accessor properties, 129, 815
 - adding to existing objects, 131
 - defining using object literal syntax, 129
 - inheritance and, 123
 - legacy API for, 134
 - property descriptor, 131
 - uses of, 130
- accuracy property, Geocoordinates, 933
- acos() function, Math object, 791
- action property, Form object, 399, 928
- activeElement property, Document, 892
- ActiveX controls, scripting, 336
- add() method
 - DOMTokenList object, 438, 902
 - HTMLOptionsCollection object, 938
 - jQuery, 580
 - Select element, 987
- addClass() method, jQuery, 532
- addColorStop(), CanvasGradient object, 647, 866
- addElement(), DataTransfer object, 475, 889
- addEventListener() method, 321, 456
 - capturing event handlers, 463
 - EventTarget object, 922
 - incompatibilities between attachEvent() and, 328
 - not implemented by Internet Explorer, 325
 - registering event handlers, 458
 - Worker object, 681
 - WorkerGlobalScope object, 683
- adoptNode() method, Document object, 894
- ADsafe security subset, 268

A

<a> elements

- affine transforms, 640
- after() method, jQuery, 538
- Ajax
 - defined, 491
 - functions in jQuery, 958
 - in jQuery 1.5, 565
 - transport mechanisms for, 492
 - utilities in jQuery, 559–571
 - ajax() function, 564–569
 - data types, 563
 - get() and post() functions, 563
 - getJSON() function, 561
 - getScript() function, 561
 - load() method, 559
 - passing data to, 562
 - XML with, 493
- ajax() function, 564–569
 - callbacks, 567
 - common options, 565
 - uncommon options and hooks, 568
- ajaxSetup() function, 565
- alert() method, Window object, 308, 348, 1004
- alpha values
 - specifying in canvas, 645
 - transparency of a color, 427
- altitude property, Geocoordinates object, 933
- altitudeAccuracy, Geocoordinates object, 933
- altKey property, 467, 484
 - Event object, 915
 - mouse events, 451
- anchors (regular expression), 258
- anchors property, HTMLDocument, 367
- andSelf() method, jQuery, 582
- angles, specifying in radians in canvas, 627, 638
- animate() method, jQuery, 551, 552
 - animation options object, 555
 - animation properties object, 554
 - custom animations with, 553
- animations
 - client-side libraries supporting, 435
 - creating using inline scripting of CSS, 433–435
 - creating using jQuery, 551–558
 - canceling, delaying, and queuing effects, 557
 - custom animations, 553–557
 - simple effects, 552
 - CSS Transitions and Animations, 419
 - jQuery methods for, 956
- API-specific events, 449
- apostrophes, 37
 - (see also ‘ ’ (quotation marks, single), under Symbols)
 - escaping in single-quoted strings, 37
- append() method
 - BlobBuilder object, 695, 864
 - FormData object, 932
 - jQuery, 538
- appendChild(), Node object, 383, 977
- appendData() method
 - Comment node, 881
 - Text node, 993
- appendTo() method, jQuery, 538
- Apple iPhone and iPad devices, gesture and touch events, 456
- <applet> elements, 537
- application cache, 601–607
 - creating application manifest file, 601
 - status property values, 606
 - updates, 603–607
- ApplicationCache object, 859
 - constants, values for status property, 859
 - event handlers, 860
 - status property, 606
 - swapCache() method, 860
 - update() method, 860
- applicationCache property, Window object, 604, 1002
- apply() and call(), Function object, 170, 187
 - restrictions in secure subsets, 267
- apply() method, Function object, 776
- appName property
 - Navigator object, 347, 974
 - WorkerNavigator object, 1012
- appVersion property
 - Navigator object, 347, 974
 - WorkerNavigator object, 1012
- arc(), CanvasRenderingContext2D, 631, 643, 873
- arcTo(), CanvasRenderingContext2D, 643, 873
- arguments (function), 171
 - object properties as, 174
 - variable-length argument lists, 172
- Arguments object, 172, 719
 - callee and caller properties, 173

- callee property, 720
 - length property, 186, 721
 - restriction in secure subsets, 267
- arguments property, Function object, 777
- arguments[] array, 719
- arithmetic expressions, 66–70
 - + operator, 67
 - bitwise operators, 69
 - unary arithmetic operators, 68
- arithmetic operators, 33, 62
- arity (functions), 173, 186
- array comprehensions, 280
 - changing to generator expressions, 282
 - syntax, 281
- Array() constructor, 142
- array-like objects, 158–160, 353
 - Arguments object, 172
 - DOMTokenList object, 438
 - frames property referring to, 357
 - HTMLCollection objects, 367
 - jQuery, 528
 - typed arrays, 687
- ArrayBuffer object, 689, 861
 - byte ordering endianness, 690
 - readAsArrayBuffer(), FileReader, 698, 699
- ArrayBufferView object, 861
- arrays, 5, 29, 141–161
 - adding and deleting elements, 145
 - Array class, 30
 - indexOf() and lastIndexOf(), 157
 - toString() method, 50
 - Array object, 722–740
 - concat() method, 150, 724
 - every() method, 154, 725
 - filter() method, 154, 726
 - forEach() method, 153, 726
 - indexOf() method, 727
 - isArray() method, 160
 - join() method, 149, 728
 - lastIndexOf() method, 729
 - length property, 722, 730
 - map() method, 154, 730
 - methods, 722
 - methods, invoking indirectly on
 - NodeLists or HTMLCollections, 367
 - pop() method, 731
 - push() and pop() methods, 151
 - push() method, 731
 - reduce() and reduceRight(), 155
 - reduce() method, 732
 - reduceRight() method, 733
 - reverse() method, 149, 734
 - shift() method, 734
 - slice() method, 150, 735
 - some() method, 155, 736
 - sort() method, 149, 224, 737
 - splice() method, 151, 737
 - toLocaleString() method, 738
 - toString() and toLocaleString(), 152
 - toString() method, 739
 - unshift() and shift() methods, 152
 - unshift() method, 739
 - of class instances, sorting, 224
 - comparing, 45
 - conversions, 51
 - to strings, 152
 - converting jQuery object to true array, 528
 - creating, 141
 - distinguishing from non-array objects, 157
 - ES5 Array methods, 328, 529
 - functions assigned to elements, 176
 - initializers, 58
 - iterating, 146–148
 - with for/each loops, 274
 - with for/in loops, 100
 - with jQuery.each() method, 572
 - Java, getting and setting elements with
 - JavaScript in Rhino, 292
 - length of, 144
 - methods, 148–152
 - ECMAScript 5, 153–157
 - multidimensional, 148
 - objects as associative arrays, 120
 - processing with functions, 192–193
 - property access expressions, 60
 - reading and writing elements, 142
 - sparse, 144
 - strings as, 39, 160
 - typed, 996
 - typed, and ArrayBuffers, 687–691
 - ArrayBuffers, 689
 - efficiency of typed arrays, 688
 - kinds of typed arrays, 687
 - set() method, 689
 - subarray() method, 689
 - of values, destructuring assignments, 272
- asin() function, Math object, 792

- assert() method, Console object, 882
 - assign() method, Location object, 344, 965
 - assignment
 - destructuring, 272, 277
 - object properties, 122
 - properties, 122
 - rules for success or failure of, 123
 - assignment expressions, 77
 - assignment with operation, 78
 - side effects, 88
 - assignment operators, 62
 - side effects, 64
 - assistive technology, 332
 - animation interfering with, 551
 - associative arrays, 120
 - associativity, operator, 65
 - async property, Script object, 985
 - asynchronous I/O
 - scripting with Node, 296–304
 - HTTP client utilities module (example), 302–304
 - HTTP server (example), 300–302
 - XMLHttpRequest and File API
 - specifications, Version 2, 455
 - asynchronous loading and execution of scripts, 319
 - asynchronous messaging between scripts of different origins, 677
 - atan() function, Math object, 792
 - atan2() function, Math object, 792
 - atob() method, Window object, 1004
 - attachEvent() method, 321, 456, 923
 - handlers registered with, this value, 461
 - incompatibilities between
 - addEventListener() and, 328
 - registering event handlers in IE, 459
 - attention, drawing to elements of a document, 437
 - Attr object, 378, 862
 - attr() method, jQuery, 531, 542
 - attribute names in E4X, 285
 - attributes, 375–378
 - as Attr nodes, 378
 - CSS attribute names for animation
 - properties object, 554
 - dataset, 377
 - event handlers registered as, scope, 461
 - getting and setting CSS attributes with
 - jQuery, 532
 - getting and setting HTML attributes with
 - jQuery, 531
 - getting and setting non-HTML attributes, 376
 - HTML attributes affecting audio and video playback, 618
 - HTML attributes mirrored by event handler
 - properties, 315
 - HTML elements, 910–913
 - HTML, as element properties, 375
 - object, 116, 135–138
 - class attribute, 136
 - extensible attribute, 137
 - property attributes, 116, 131–134
 - copying, 133
 - querying and setting, 131
 - setting for event handlers, 457
 - attributes property, Element object, 903
 - audio and video
 - MediaElement superclass, 965–970
 - MediaError objects, 970
 - scripting, 615
 - controlling playback, 617
 - media events, 620
 - querying media status, 618
 - type selection and loading, 617
 - Video object, 998
 - <audio> elements, 615
 - controls attribute, 616
 - events, 454
 - Audio object, 862
 - audio property, Video object, 998
 - Audio() constructor, 616
 - Authorization headers, 497
 - autocomplete property
 - Form object, 928
 - Input object, 942
 - autofocus property, FormControl object, 930
 - autoplay property, MediaElement object, 618, 966
 - availHeight property, Screen object, 984
 - availWidth property, Screen object, 984
- ## B
- \b (backspace character) in regular expressions, 254
 - back() method, History object, 345, 936
 - background CSS style property, 419
 - background, CSS style properties for, 427

- background-color property, 421, 426
- baseURL property, Node object, 976
- before() method, jQuery, 538
- BeforeUnloadEvent object, 862
- beginPath(), CanvasRenderingContext2D, 632, 873
- Bezier curves
 - drawing in canvas, 643–645
 - quadraticCurveTo() method, 878
- bezierCurveTo(), CanvasRenderingContext2D, 643, 874
- big-endian byte ordering, 690, 890
- binary data
 - Blobs and APIs that use them, 691
 - in HTTP responses, 501
- binary floating-point numbers, 35
- Binary Large Objects (see Blobs)
- bind() method, 188
 - event handler registration in jQuery, 544
 - Function object, 208, 778
 - similarity of jQuery.proxy() function, 573
- Function.bind() method for ECMAScript 3, 189
- live events and, 549
- partial application of functions, 194
- bitwise operators, 62, 69
- _blank window name, 354
- blank-page URL, about:blank, 354
- BlobBuilder object, 694, 863
- Blobs, 691–699, 863
 - building, 694
 - downloading, 694
 - files as, 693
 - obtaining, methods for, 691
 - reading, 698
 - URLs, 695–697
 - displaying dropped image files, 695
 - using, 692
- block scope, 54
 - lack of, addressing by using let keyword, 269
- blocking script execution, 319
- blur effects
 - motion blur with ImageData, 661
 - shadowBlur property in canvas, 653
- blur events, 401, 449
 - bubbling alternative to, 453
 - event handler registration with jQuery, 541
- blur() method
 - Element object, 906
 - Window object, 1004
- <body> elements, event handlers in, 457
- body property
 - Document object, 892
 - HTMLDocument object, 367
- bookmarklets
 - for currently selected text, 408
 - using javascript: URLs for, 316
- Boolean object, 740
 - toString() method, 741
 - valueOf() method, 741
- Boolean() constructor, 43
- Boolean() function, type conversions with, 47
- booleans, 29
 - boolean values, 40
 - conversions, 45
 - object-to-boolean conversions, 49
 - wrapper objects, 43
- border property, 416
- borders
 - in CSS box model, 424
 - specifying color of element's border, 426
 - specifying in CSS, 423
- borrowing methods, 224
- bottom and right style properties, 421, 425
- bottom property, ClientRect object, 880
- bottom, top, right, and left properties, 392
- box model (CSS), 424
 - border box model and box-sizing property, 425
 - jQuery.support.boxModel property, 574
- box-sizing property, 425
- break keyword, 96
- break statements, 103
 - line break interpreted as semicolon, 26
- browser property, 571
- browser sniffing, 330, 346
 - using navigator.userAgent, 347
- browsing contexts, 353
- browsing history, 345
 - history management mechanism in HTML5, 454, 671–676
 - History object, 936
 - transition, PopStateEvent, 982
- toa() method, Window object, 1004
- bubbles property, Event object, 915

- bubbling, 321, 446
 - event handling in jQuery, 541
 - keyboard events to document and window, 452
 - live events, 550
 - manually triggered events, 547
 - mouse events, 453
 - non-bubbling version of mouse events in IE, 452
 - place in event propagation after invoking event handlers, 463
 - buffer property, ArrayBuffer object, 861
 - buffered property, MediaElement, 619, 966
 - bufferedAmount property, WebSocket, 1000
 - buffers in Node interpreter, 298
 - bugs in browsers, 325
 - testing for, 330
 - button property, Event object, 451, 467, 915
 - buttons
 - <button> elements, 397
 - onclick event handler, 316
 - registering event handlers for click event, 458
 - Button object, 864
 - push buttons in forms, 401
 - toggle buttons in forms, 402
 - buttons property, Event object, 919
 - byte ordering, endianness, 690
 - byteLength property, ArrayBuffer object, 861
 - byteLength, ArrayBuffer object, 861
 - byteOffset, ArrayBuffer object, 861
- C**
- caching, 601
 - (see also application cache)
 - memoization of functions, 196
 - Caja secure subset, 268
 - calculated values for box dimensions (CSS), 426
 - call() and apply(), Function object, 170, 187
 - restrictions in secure subsets, 267
 - call() method, Function object, 136, 779
 - callable objects, 191
 - callbacks, 320
 - functions passed to setTimeout() and setInterval(), 322
 - jQuery.ajax() function, 567
 - passing to jQuery effects methods, 552
 - callee and caller properties
 - Arguments object, 173
 - restriction in secure subsets, 267
 - callee property, Arguments object, 720
 - caller property, Function object, 779
 - cancelable property, Event object, 915
 - cancelBubble property, Event object, 915
 - cancelBubble() method, Event object, 548
 - cancellation, events, 464
 - application cache updates, 606
 - textInput and keypress events, 481
 - canPlayType(), MediaElement object, 969
 - Canvas API, 630–665
 - array of bytes in CanvasPixelArray, 688
 - canvas dimensions and coordinates, 637
 - Canvas object, 865
 - getContext() method, 630, 865
 - toDataURL() method, 656, 865
 - clipping, 652
 - colors, transparency, gradients, and patterns, 645–648
 - compositing, 657
 - coordinate system transformations, 638
 - drawing and filling curves, 643–645
 - drawing lines and filling polygons, 632
 - drawing red square and blue circle, 631
 - drawing sparklines (example), 663–665
 - drawing text, 650
 - graphics attributes defined on context object, 635–637
 - hit detection, determining if point is in a path, 662–663
 - images, 655–657
 - line drawing attributes, 648
 - path, 631
 - pixel manipulation, 661–662, 870
 - rectangles, 645
 - reference, 865–880
 - shadows, 653
 - <canvas> elements, 630
 - context object, 636
 - canvas property, 871
 - CanvasGradient object, 645, 866
 - addColorStop() method, 866
 - creating, 874
 - fill or stroke with color gradient, 647
 - CanvasPattern object, 645, 867
 - creating, 875
 - fill or stroke using, 647
 - CanvasRenderingContext2D

- save() and restore() methods, 636
- CanvasRenderingContext2D object, 630, 867–880
 - arc() method, 873
 - arcTo() method, 873
 - beginPath() method, 873
 - bezierCurveTo() method, 874
 - clearRect() method, 874
 - clip() method, 652, 874
 - closePath() method, 874
 - compositing, 657–660
 - createImageData() method, 661, 874
 - createLinearGradient() method, 647, 874
 - createPattern() method, 647, 875
 - createRadialGradient() method, 647, 875
 - drawImage() method, 655, 875
 - drawing and filling curves, 643–645
 - drawing rectangles, 645
 - fill() method, 875
 - fillRect() method, 876
 - fillText() method, 650, 876
 - getImageData() method, 661, 876
 - graphics attributes, 635–637
 - graphics attributes for shadows, 653
 - isPointInPath() method, 662, 877
 - lineTo() method, 877
 - measureText() method, 651, 877
 - moveTo() method, 877
 - putImageData() method, 661, 878
 - quadraticCurveTo() method, 878
 - rect() method, 878
 - restore() method, 878
 - rotate() method, 878
 - save() and restore() methods, 870
 - save() method, 879
 - scale() method, 879
 - setTransform() method, 879
 - stroke() method, 879
 - strokeRect() method, 879
 - strokeText() method, 650, 879
 - transform() method, 880
 - translate() method, 880
- capability testing, 329
- caption property, Table object, 990
- capturing event handlers, 458, 463, 545
 - mouse events in IE, 468
- carriage returns, 22
- Cascading Style Sheets (see CSS)
- case clauses in switch statements, ending with break statement, 96
- case keyword, 95
- case sensitivity in JavaScript, 21
 - property names, 376
- case-insensitive sorting, arrays of strings, 150
- catch clauses (try/catch/finally), 106
 - multiple catch clauses, 283
- catching exceptions, 106
- cd() method, ConsoleCommandLine, 884
- CDATASection nodes, 381
- ceil() function, Math object, 793
- cellIndex property, TableCell object, 991
- cells property, TableRow object, 992
- chaining methods, 169
- chaining, constructor and method, from subclass to superclass, 231–233
- change() method, jQuery, 541
- changedTouches property, touch events, 456
- char property, Event object, 454, 919
- character classes in regular expressions, 253
- character sets, 21
- CharacterData object, 381
- characterSet property, Document, 892
- charAt() method, String object, 838
- charCode property, Event object, 915
- charCodeAt() method, String object, 501, 838
- charset property
 - Document object, 892
 - Script object, 985
- chat
 - custom Server-Sent Events chat server, 519
 - server using WebSockets and Node, 715
 - simple client using EventSource, 516
 - WebSocket-based chat client, 714
- <checkbox> elements, 402
- checked property
 - form elements, 399
 - Input object, 942
- checkValidity(), FormControl object, 931
- child properties, Element object, 372
- child windows, 356
 - browsing history and, 345
- childElementCount, Element object, 373, 903
- childNodes property, Node object, 371, 976
- children property, Element object, 372, 375, 903
- children() method, jQuery, 580
- Chrome, 471

- (see also web browsers)
- current version, 327
- implementation of Filesystem API, 700
- JavaScript in URLs, 316
- textInput event, 453
- class attribute, 10, 116, 136
 - Array, 158
 - HTML elements, 368, 376
 - testing for true function object, 191
- class fields and methods (Java classes), 205
- class members, in strongly-typed object-oriented languages, 205
- classes, 30, 199–246
 - (see also modules; objects)
 - augmenting by adding methods to
 - prototype, 208
 - built-in, automatically predefined in all windows, 359
 - constructors, 201
 - and class identity, 203
 - constructor property, 203
 - defining for CSS properties, 417
 - defining, example, 8
 - defining, multiple windows and, 359
 - determining class of an object, 210–215
 - duck-typing, 213
 - using constructor name as class identifier, 211
 - using constructor property, 211
 - using instanceof operator, 210
 - in ECMAScript 5, 238
 - defining immutable classes, 239–241
 - encapsulating object state, 241
 - making properties nonenumerable, 238
 - preventing class extensions, 242
 - property descriptors, 244–246
 - subclasses, 243
 - getting and setting CSS classes, 532
 - getting class of an object, 136
 - instanceof operator, 75
- Java
 - instantiating, 291
 - querying and setting static fields in Rhino, 291
- Java-style, in JavaScript, 205–208
 - defining complex class, 206
 - features not supported in JavaScript, 208
 - function defining simple classes, 205
- object-oriented programming with, 215–228
 - borrowing methods, 224
 - comparison methods, 221–224
 - constructor overloading and factory methods, 227
 - enumerated types (example), 217–219
 - private state, 226
 - Set class (example), 215–217
 - standard conversion methods, 219
- partial class hierarchy of document nodes, 363
- prototypes and, 200
- scripting CSS classes, 437–440
- subclasses, 228
 - class hierarchies and abstract classes, 234–238
 - composition versus subclassing, 233
 - constructor and method chaining, 231–233
 - defining, 229
- classList property, 438, 533
 - approximating functionality, example of, 438–440
- Element object, 903
- className property, 308, 368, 376, 438–440
- Element object, 308, 903
- clear() method
 - ConsoleCommandLine object, 884
 - Storage object, 988
- clearData(), DataTransfer object, 889
- clearInterval() function, 297
- clearInterval() method, 342
 - Window object, 1004
 - WorkerGlobalScope object, 1010
- clearQueue() method, jQuery, 558
- clearRect(), CanvasRenderingContext2D, 645, 874
- clearTimeout() function, 297
- clearTimeout() method
 - Window object, 1005
 - WorkerGlobalScope object, 1010
- clearWatch(), Geolocation object, 933
- click events, 466
 - detail property, 451
 - registering event handlers for, 11
 - registering event handlers on button element for, 458
- click() method, 540

- Element object, 906
- client properties, document elements, 395
- client sniffing, 330
- client-side JavaScript, 8–12, 307
 - building web applications with, 338
 - compatibility and interoperability issues, 325
 - keeping HTML content separate from JavaScript behavior, 458
 - loan calculator (example), 12–18
 - scripting HTML document content, 9
 - security, 333
 - threading model, 322
- client-side storage, 587–612
 - application storage and offline web applications, 601–607
 - application cache manifest, 601–603
 - cache updates, 603–607
 - cookies, 593–599
 - storage with cookies, 597–599
 - IE userData persistence, 599–601
 - localStorage and sessionStorage properties, 589
 - security, privacy, and, 589
- client-side storage and offline web applications
 - offline web applications, 608–612
- clientHeight and clientWidth properties, Element object, 392, 904
- clientLeft and clientTop properties, Element object, 904
- ClientRect object, 880
- clientX and clientY properties, Event object, 451, 467, 916
- clip property, 428
- clip(), CanvasRenderingContext2D, 648, 652, 874
- clone() method, jQuery, 539
- cloneNode() method, Node object, 382, 978
- clones, structured, 672
- close() method
 - distinguishing between Window and Document objects, 356
 - Document object, 894
 - EventSource object, 921
 - generators, 278
 - MessagePort object, 972
 - WebSocket object, 714, 1000
 - Window object, 1005
 - WorkerGlobalScope object, 682, 1010
- closed property, Window objects, 356
- CloseEvent object, 881
- closePath(), CanvasRenderingContext2D, 633, 874
- closest() method, jQuery, 581
- closing windows, 356
- Closure library, 339
- closures, 180–185
 - access to outer function arguments, 185
 - combining with property getters and setters, 183
 - lexical scoping rule for nested functions, 181
 - private property accessor methods using, 184
 - property accessor methods using, 226
 - shorthand functions (expression closures), 282
 - using in uniqueInteger() function (example), 182
- code property
 - DOMException object, 900
 - FileError object, 924
 - GeolocationError object, 934
 - MediaError object, 970
- codepoints (Unicode), 36, 481
- color stops, 647
- colorDepth or pixelDepth property, Screen object, 984
- colors
 - background-color property, 421
 - gradients in canvas, 646
 - shadowColor property, 653
 - specifying for paths in canvas, 868
 - specifying in canvas, 645
 - specifying with CSS, 426
- colSpan property, TableCell object, 991
- Comet, 491
 - Server-Sent Events with, 515–521
 - transport mechanisms for, 493
- comma operator (,), 85
- commands
 - ConsoleCommandLine objects, 883
 - text editing, 410
- comments, 4
 - Comment node, 881
 - conditional, 331
 - createComment(), Document object, 382
 - creating Comment node, 894

- CSS handling of, 414
- JavaScript code in URLs, 315
- styles of, 23
- `compareDocumentPosition()`, Node object, 978
- comparison methods, implementation in
 - classes, 221–224
 - `compareTo()` method, 222
 - `equals()` method, 221
- comparison operators, 62, 73, 223
- comparisons
 - objects, 45
 - primitive values, 44
- compatibility and interoperability, 325–332
 - browser testing, 330
 - compatibility libraries, 328
 - conditional comments in Internet Explorer, 331
 - feature testing, 329
 - graded browser support, 329
 - quirks mode and standards mode, 330
- `compatMode`, Document object, 330, 892
- complete property, Image object, 555, 941
- compositing in canvas, 657–660, 869
 - locally rather than globally, 659
- composition
 - subclassing versus, 233
 - using with partial application of functions, 196
- computed styles, 416, 532
 - querying, 436–437, 436
- `concat()` method
 - Array object, 150, 724
 - String object, 839
- concatenating strings, 38
- conditional comments in Internet Explorer, 331
- conditional operator (`?:`), 82
- conditional statements, 87, 92
 - in array comprehensions, 281
 - else if, 94
 - if, 92
 - switch, 95
- configurable attribute (properties), 116, 131
 - deletion of properties, 124
 - extensible attribute used with, 137
- `confirm()` method, Window object, 348, 1005
- console API, 3
- Console object, 882
 - methods, 882
- console tools, 3
- `console.log()` function, 686
- ConsoleCommandLine object, 883
- `const` keyword, 269
- constants
 - block scope and use of `let` keyword, 269
 - fields declared `final` in Java classes, 208
- constructor chaining, 228
 - from subclass to superclass, 231–233
- constructor objects, 205
- constructor property, 135
 - identifying class of an object, 211
 - restrictions in secure subsets, 267
- constructors, 163, 782
 - class, 201
 - and class identity, 203
 - constructor property, 203
 - client-side, in WorkerGlobalScope objects, 684
 - constructor property of an object, 811
 - defined, 30
 - interactive windows and, 359
 - invoking, 170
 - object prototypes and, 118
 - overloading, 227
 - prototype property, 135
 - for typed arrays, 997
 - using name of as class identifier, 211
- `contains()` function, 571
- `contains()` method, DOMTokenList, 438, 902
- content
 - generating document content at load time, 318
 - simple client-side script for revealing, 309
- content-box model, 425
- Content-Type headers
 - automatically set for request by XMLHttpRequest, 505
 - HTTP requests, 496
 - overriding incorrect MIME type in response, 501
- `contentDocument`, IFrame object, 939
- `contentEditable` property, Element object, 409
- `contents()` method, jQuery, 581
- `contentType` option, 562
- `contentWindow`, IFrame object, 357, 939
- context menus, 466
- context property, 947

- jQuery objects, 529
- contextmenu events, 451, 466
- continue statements, 104
 - semicolon interpreted as line break, 26
- contractions and possessives in English
 - language strings, 37
- control property, Label object, 962
- control structures, 6, 87
 - (see also statements)
- controls attribute, <audio> and <video>
 - elements, 616
- controls property, MediaElement object, 966
- cookie property, Document object, 405, 892
- cookies, 588, 593–599
 - attributes, lifetime and scope, 594
 - determining if they are enabled, 594
 - limitations on size and number of, 597
 - reading, 596
 - storage with, 597–599
 - storing, 595
- cookiesEnabled(), Navigator object, 348
- coordinate system transformations, 638–643, 869
- Coordinated Universal Time (UTC), 742
- coordinates
 - canvas, 637
 - images, 655
 - converting document to viewport
 - coordinates, 396
 - document and viewport, 390
 - document, as scrollbar offsets, 394
 - Geocoordinates object, 933
 - position of mouse in window coordinates, 451
- coords property, Geoposition object, 935
- CORS (“Cross-Origin Resource Sharing”)
 - headers, 511
- cos() function, Math object, 794
- count() method, Console object, 882
- create() function, 118, 811
 - adding properties to newly created object, 133
- createCaption() method, Table object, 990
- createComment(), Document object, 894
- createDocument(), DOMImplementation, 901
- createDocumentFragment(), Document object, 385, 894
- createDocumentType(), DOMImplementation object, 901
- createElement(), Document object, 382, 895
- createElementNS(), Document object, 382, 627, 895
- createEvent(), Document object, 895
- createHTMLDocument(),
 - DOMImplementation object, 901
- createImageData(),
 - CanvasRenderingContext2D, 661, 874
- createIndex() method, object store, 707
- createLinearGradient(),
 - CanvasRenderingContext2D, 647, 874
- createObjectStore() method, IndexedDB objects, 707
- createObjectURL() function, 695
- createObjectURL(), URL object, 998
- createPattern(), CanvasRenderingContext2D, 647, 875
- createProcessingInstruction(), Document object, 895
- createRadialGradient(),
 - CanvasRenderingContext2D, 647, 875
- createRange() method, 408
- createStyleSheet(), Document object, 442
- createTBody() method, Table object, 990
- createTextNode(), Document, 382, 895
- createTFoot() method, Table object, 990
- createTHead() method, Table object, 990
- Crockford, Douglas, 119, 266, 268
- cross-document messaging, 336, 676–680, 677
- Cross-Document Messaging API, 455
- “Cross-Origin Resource Sharing”, 335
- cross-origin HTTP requests, 496, 511–513
 - requesting link details with HEAD and CORS, 512
- cross-site scripting (XSS), 336
- CSS (Cascading Style Sheets), 413–443
 - box model and positioning details, 424
 - classes, selecting elements by, 368
 - color specifications, 646
 - color, transparency, and translucency, 426
 - element display and visibility, 426
 - getting and setting CSS attributes, 532
 - getting and setting CSS classes, 532

- important style properties, 419
- overlapping translucent windows (example), 429–431
- overview, 414
- partial visibility, overflow and clip, 428
- positioning elements, 420–423, 472–474
- querying computed styles, 436–437
- revolutionary new features of, 418
- scripting CSS classes, 437–440
- scripting inline styles, 431–435
 - animations, 433–435
- scripting stylesheets, 440–443
- selectors, 369
- styles, specified for Element object, 308
- use with JavaScript to style presentations of HTML, 10
- web page styled with CSS (example), 417
- css() method, jQuery, 532
- CSSOM-View Module, 390
- CSSRule object, 441, 885
- cssRules property, CSSStyleSheet, 441, 887
- CSSStyleDeclaration object, 886
 - computed styles, 436
 - describing styles associated with selector, 441
 - properties corresponding to shortcut properties, 432
 - using in scripting inline styles, 431
- CSSStyleSheet object, 440, 887
 - creating, 442
 - disabled property, 440
 - inserting and deleting rules, 441
 - querying, inserting, and deleting stylesheet rules, 441
- cssText property
 - CSSRule object, 885
 - CSSStyleDeclaration object, 433, 441, 886
- ctrlKey property, 451, 467, 484
 - Event object, 916
- currentSrc, MediaElement object, 966
- currentStyle property
 - Element object, 904
 - in Internet Explorer (IE), 437
- currentTarget property, Event object, 543, 916
- currentTime, MediaElement object, 617, 967
- currying, 188
- cursor, opening in IndexedDB, 706
- curves, drawing and filling in canvas, 643–645, 874

- custom events, using jQuery with, 549
- customError, FormValidity object, 932

D

- data properties, 129
 - attributes of, 131
- data property
 - CharacterData object, 381
 - Comment object, 881
 - Event object, 453, 481, 544, 919
 - ImageData object, 688, 942
 - MessageEvent object, 677, 971
 - ProcessingInstruction object, 982
 - Text node, 993
- data storage APIs for web applications, 311
- data types, 4, 29–56
 - booleans, 40
 - classes and, 210
 - determining class of an object with constructor property, 211
 - determining class of an object with instanceof, 210
 - duck-typing, 213
 - standard conversion methods, 219
 - using constructor name as class identifier, 211
- conversions, 31, 45–52
 - equality and, 47
 - explicit, 47–49
 - object to primitive, 49–52
 - summary listing of, 45
- function argument, 174
- Java, conversions in Rhino, 293
- jQuery's Ajax data types, 563
- methods, 30
- mutable and immutable types, 31
- numbers, 31–36
 - binary floating-point and rounding errors, 34
 - dates and times, 35
 - floating-point literals, 32
 - integer literals, 32
- objects and arrays, 5
- operand and result type, 64
- primitive and object types, 29
- text, 36–40
 - escape sequences in string literals, 37
 - pattern matching, 39
 - string literals, 36

- working with strings, 38
 - typeof operator, 82
 - variable declaration and, 52
- data() method, jQuery, 536, 584
- data:// URLs, 926, 927
- databases
 - client-side, 705–713
 - client-side database functionality in browsers, 588
- dataset attributes, 377
 - using to create image rollovers, 614
- dataset property, Element object, 377, 904
- DataTransfer object, 454, 475–481, 888
 - files property, 505, 694
- dataTransfer property, Event object, 454, 475, 916
- DataView object, 890
 - methods reading/writing values from ArrayBuffer, 690
- Date() constructor, 742
- datepicker() method, 586
- dates and time
 - Date class, 30
 - toString() method, 50
 - Date object, 742–766
 - conversions to strings and numbers, 51
 - getDate() method, 746
 - getDay() method, 747
 - getFullYear() method, 747
 - getHours() method, 747
 - getMilliseconds() method, 747
 - getMinutes() method, 748
 - getMonth() method, 748
 - getSeconds() method, 748
 - getTime() method, 748
 - getTimezoneOffset() method, 749
 - getUTCDate() method, 749
 - getUTCDay() method, 750
 - getUTCFullYear() method, 750
 - getUTCHours() method, 750
 - getUTCMilliseconds() method, 750
 - getUTCMinutes() method, 751
 - getUTCMonth() method, 751
 - getUTCSeconds() method, 751
 - getYear() method, 751
 - methods, 743
 - now() method, 752
 - parse() method, 752
 - prototype property, 118
 - seMilliseconds() method, 754
 - setDate() method, 753
 - setFullYear() method, 753
 - setHours() method, 754
 - setMinutes() method, 755
 - setMonth() method, 755
 - setSeconds() method, 756
 - setTime() method, 756
 - setUTCDate() method, 757
 - setUTCFullYear() method, 757
 - setUTCHours() method, 758
 - setUTCMilliseconds() method, 758
 - setUTCMinutes() method, 759
 - setUTCMonth() method, 759
 - setUTCSeconds() method, 760
 - setYear() (deprecated), 760
 - static methods, 745
 - toDate() method, 760
 - toGMTString() method, 761
 - toISOString() method, 761
 - toJSON() method, 762
 - toLocaleDateString() method, 762
 - toLocaleString() method, 763
 - toLocaleStringTime() method, 763
 - toString() method, 763
 - toTimeString() method, 764
 - toUTCString() method, 764
 - UTC() method, 765
 - valueOf() method, 50, 766
 - quick tutorial on, 35
 - serializing Date objects to ISO-formatted date strings, 138
- dblclick event, 451, 466
- dblclick() method, jQuery, 541
- debug() method, Console object, 882
- debugger statements, 110
- debugging Worker threads, 686
- decimal fractions, binary floating-point representation, 35
- declaration statements, 87, 89
 - var statement, 90
- decodeURI() function, 766
- decodeURIComponent() function, 343, 595, 767
- dedicated workers, 684
- default actions of events, 447
- defaultCharset, Document object, 892
- defaultChecked property
 - Checkbox object, 402

- Input object, 943
- defaultPlaybackRate, MediaElement object, 618, 967
- defaultPrevented, Event object, 464, 916
- defaultSelected, Option object, 980
- defaultValue property
 - Input object, 943
 - Output object, 981
 - TextArea object, 994
- defaultView property, Document object, 892
- defer property, Script object, 985
- deferred execution of scripts, 319
- __defineGetter__() and __defineSetter__() methods, 134, 375
- defineProperties() function, 812
 - creating or modifying multiple properties, 132
- defineProperty() function, 101, 134, 813
 - creating new property or setting attributes, 132
 - defining configurable, read-only properties, 124
 - making length property of array read-only, 145
 - making properties nonenumerable, 238
- delay() method, jQuery, 557
- delegate() method, jQuery, 549
- delete operator, 62, 84
 - deleting array elements, 146
 - deleting configurable properties of global object, 125
 - deleting properties, 124
 - removing XML attributes and tags in E4X, 286
 - side effects, 88
- deleteCaption() method, Table object, 991
- deleteCell() method, TableRow object, 992
- deleteData() method
 - Comment node, 881
 - Text node, 993
- deleteRow() method
 - Table object, 991
 - TableSection object, 992
- deleteRule(), CSSStyleSheet object, 441, 888
- deleteTFoot() method, Table object, 991
- deleteTHead() method, Table object, 991
- deletion methods in jQuery, 953
- deltaMode property, Event object, 920
- deltaX, deltaY, and deltaZ properties, Event object, 453, 471, 920
- denial-of-service attacks, 338
- dequeue() method, jQuery, 558
- descendant operator (.), 285
- designMode, Document object, 409, 893
- destructuring assignment, 272
 - using with Iterator() function in for/in loop, 277
- detach() method, jQuery, 540
- detachEvent() method, 459, 923
- detail property, Event object, 451, 471, 916
- device-dependent and -independent events, 448
- DHTML (Dynamic HTML), 310
- dialog boxes, 348–351
 - HTML dialog box displayed with showModalDialog(), 349
 - repeated, in cross-site scripting attacks, 337
 - repeated, in denial-of-service attacks, 338
- dialogArguments, Window object, 349, 1002
- die() method, jQuery, 550
- dimensions, canvas, 637
- dir property, Document object, 893
- dir() method
 - Console object, 882
 - ConsoleCommandLine object, 884
- directives, 110
- DirectoryEntry objects, 701
- dirxml() method
 - Console object, 882
 - ConsoleCommandLine object, 884
- disabled property
 - CSSStyleSheet object, 887
 - FieldSet object, 923
 - FormControl object, 930
 - Link object, 963
 - Option object, 980
 - Style object, 989
- disabling animations in jQuery, 551
- dispatchEvent(), EventTarget object, 922
- dispatchFormChange(), Form object, 929
- dispatchFormInput(), Form object, 929
- display property, 426
- division by zero, 34
- DnD (see drag and drop)
- do/while loops, 98
- doctype declarations

- quirks mode and standards mode, 330
- strictness of, 369
- doctype property, Document object, 893
- Document object, 308, 361, 891–898
 - addEventListener() method, 459
 - close() method, 356
 - compatMode property, 330
 - cookie property, 593
 - parsing, 596
 - createDocumentFragment() method, 385
 - createElement() method, 382
 - createElementNS() method, 382
 - createStyleSheet() method, 442
 - createTextNode() method, 382
 - designMode property, 409
 - elementFromPoint() method, 393
 - events, 898
 - forms property, 398
 - getElementById() method, 352, 364
 - getElementsByClassName() method, 368
 - getElementsByTagName() method, 159, 366
 - location property, 343
 - methods, 894–898
 - open() method, 346
 - parsed HTTP responses available as, 500
 - properties, 405, 892
 - queryCommandEnabled() method, 410
 - querySelector() method, 370
 - querySelectorAll() method, 370, 440
 - readyState property, 323, 465
 - removeEventListener() method, 459
 - same-origin policy applied to properties, 335
 - styleSheets property, 440, 441
 - URL property, 343
 - write() method, 318, 324, 346, 406
 - writeln() method, 407
- document property, Window object, 308, 1002
- document.all[] collection, 371
- documentElement, Document object, 367, 391, 893
- DocumentFragment object, 364, 385, 898
 - creating, 894
 - implementing insertAdjacentHTML()
 - using innerHTML, 386
 - querySelector() and querySelectorAll()
 - methods, 370
 - using to reverse order of children of a Node, 386
- documents, 361–411
 - altering structure using jQuery, 537–540
 - associating CSS stylesheet with, 415
 - attributes of elements, 375–378
 - creating, inserting, and deleting nodes, 382–387
 - creating nodes, 382
 - inserting nodes, 383
 - removing and replacing nodes, 384
 - using DocumentFragments, 385
 - Document properties, 405
 - editable content, 408–411
 - document and element geometry and scrolling, 390–396
 - element content, 378–382
 - elements as Window object properties, 351
 - generating content at load time, 318
 - generating table of contents (example), 387–390
 - HTML forms, 397–404
 - JavaScript in web documents, 310
 - load events, 465–466
 - loading new, 344
 - nested HTML documents, 353
 - origin of, 334, 590
 - overview of DOM, 362
 - querying selected text, 407
 - selecting elements in, 364–371
 - structure and traversal, 371–375
 - documents as trees of elements, 372–375
 - documents as trees of nodes, 371
 - write() method, 406
- DocumentType object, 899
- Dojo, 338
 - dojox.secure, 268
 - Sizzle library, 370
- DOM (Document Object Model)
 - DOM Level 2 Events specification, 325
 - DOM Level 3 Events specification, 447, 452, 453
 - key property, 485
 - proposed method, 920
 - proposed properties, 919
 - event cancellation in current Events module
 - draft, 464
 - implementation of types as classes, 374

- overview, 362
 - textInput event, 481
 - XML objects and E4X standard, 284
 - domain property, Document object, 405, 893
 - domains
 - domain attribute, cookies, 594
 - setting, 596
 - problems with same-origin policy and subdomains, 335
 - DOMContentLoaded event, 324, 450, 465
 - DOMException object, 899
 - DOMImplementation object, 900
 - DOMMouseScroll objects, 471
 - DOMSettableTokenList object, 901
 - DOMTokenList object, 438, 902
 - drag and drop, 474–481
 - access to files user drops over element, 505
 - custom drag source, 476
 - DataTransfer objects, 888
 - displaying dropped image files with Blob URLs, 695
 - drag source events, 475
 - drag() function evoked from mousedown event handler, 467
 - dragging document elements, 468–470
 - drop targets, 477
 - dropping local files into browser for script access, 694
 - HTML5 drag-and-drop API, 454
 - list as drop target and drag source (example), 478–481
 - obtaining files from DnD API and uploading via HTTP request, 508
 - draggable attribute, 475
 - drawImage(), CanvasRenderingContext2D, 655, 875
 - drawing context objects, 630
 - dropEffect, DataTransfer object, 476, 888
 - dropzone attribute, 477
 - duck-typing, 213
 - duration property, MediaElement object, 617, 618, 967
 - duration, animated effects, 551, 956
 - passing to jQuery effects methods, 552
 - Dynamic HTML (DHTML), 310
- ## E
- E constant (Math), 794
 - E4X (ECMAScript for XML), 274
 - introduction to, 283–287
 - each() function, 572
 - each() method, jQuery, 529
 - easing functions, 556
 - adding to jQuery, 584
 - ECMAScript, 2
 - JavaScript extensions, 265
 - ECMAScript 5
 - array methods, 153–157, 159, 529
 - classes in, 238
 - defining immutable classes, 239–241
 - defining nonenumerable properties, 238
 - encapsulating object state, 241
 - preventing class extensions, 242
 - property descriptors, 244–246
 - subclasses, 243
 - RegExp literals and object creation, 252
 - reserved words, 24
 - ECMAScript for XML (see E4X)
 - editable content in documents, 408–411
 - editor components in frameworks, 410
 - effectAllowed, DataTransfer object, 475, 477, 888
 - Element class, 364
 - element expressions in array initializers, 58
 - Element object, 308, 902–909
 - attributes property, 378
 - contentEditable property, 409
 - dataset property, 377
 - defining custom methods, 374
 - event handlers, 909
 - getAttribute() and setAttribute() methods, 376, 433
 - getBoundingClientRect() method, 396
 - hasAttribute() method, 377
 - implementing outerHTML property, using innerHTML, 384
 - innerHTML property, 379
 - methods, 906–909
 - Node objects, 371
 - offsetLeft and offsetTop properties, 394
 - offsetParent property, 395
 - outerHTML property, 379
 - properties, 903
 - properties, element-based document traversal API, 372
 - removeAttribute() method, 377
 - representing <style> and <link> elements, scripting stylesheets, 440

- style property, 431
- elementFromPoint(), Document, 393, 895
- elements
 - array, 141
 - attributes of, 375–378
 - attributes as Attr nodes, 378
 - dataset attributes, 377
 - non-HTML, getting and setting, 376
 - computed style, 416
 - content, 378
 - content as HTML, 379
 - content as plain text, 380
 - content as Text nodes, 381
 - copying with jQuery, 539
 - deleting using jQuery, 540
 - display and visibility properties (CSS), 426
 - document, as Window properties, 351
 - documents as trees of, 372–375
 - geometry and scrolling, 390–396
 - getting and setting content, 534
 - getting and setting element data, 536
 - getting and setting geometry, 534
 - HTML elements and attributes, 910–913
 - HTML form, 397
 - inserting and replacing in documents with jQuery, 537
 - jQuery element methods, 950
 - media, 965–970
 - positioning with CSS, 420
 - querying geometry of, 392
 - the selected elements in jQuery, 528
 - selecting document elements, 364
 - selecting HTML form elements, 398
 - size, positioning, and overflow, 394
 - wrapping around other elements, using jQuery, 539
- elements property
 - FieldSet object, 923
 - Form object, 399, 928
- else clauses in nested if statements, 93
- else if statements, 94
- <embed> elements, 537
- embedding JavaScript in HTML, 311–317
 - event handlers in HTML, 315
 - JavaScript in URLs, 315
 - <script> element, 312
 - script type, 314
 - scripts in external files, 313
- embeds property
 - Document object, 893
 - HTMLDocument object, 367
- empty statements, 89
- empty strings, 36
- empty() method, jQuery, 540
- enableHighAccuracy option, Geolocation methods, 934
- encapsulation
 - object state, in ECMAScript 5, 241
 - state variables, 226
- encodeURIComponent() function, 767
- encodeURIComponent() function, 595, 768
- encoding HTTP request body, 502–507
- encoding property, Form object, 399
- enctype property, Form object, 928
- end() method
 - jQuery, 582
 - TimeRanges object, 996
- ended property, MediaElement object, 967
- endianness, 690
- enumerable attribute (properties), 116, 131
 - propertyIsEnumerable() test, 125
- enumerable properties, 101
 - arrays, enumeration by for/each loop, 274
 - Object.propertyIsEnumerable() method, 822
 - returning own enumerable property names, 821
- enumerated types, 217–219
 - classes representing cards, 218
 - comparisons, 223
- enumerating properties, 126–128
 - order of, in for/in loops, 101
- eq() method, jQuery, 578
- equality comparisons
 - binary floating-point and rounding errors, 35
 - type conversions and, 47
- equality operators (see = (equals sign), under Symbols)
- equals() method, defining for classes, 221
- error property
 - FileReader object, 925
 - MediaElement object, 620, 967
- Error() constructor, 769
- error() method
 - Console object, 882
 - jQuery, 541
- ErrorEvent object, 913

errors

- Error class, 30, 106
- error handlers, similarity to events, 449
- Error object, 769–771
 - javaException property, 293
 - message property, 771
 - name property, 771
 - toString() method, 771
- handling in Window objects, 351
- property access, 123
- ES5 (see ECMAScript 5)
- escape sequences
 - in string literals, 37
 - Unicode, 22
- escape() function, 771
- eval() function, 79–81, 772
 - removal in secure subsets, 267
- EvalError object, 774
- evaluation expressions, 79–81
 - eval() function, 80
 - global eval(), 80
 - strict eval(), 81
- evaluation order, operators, 66
- evaluation, modules, 302
- event capturing, 447, 458, 463
 - Internet Explorer, setCapture() for mouse events, 468
 - jQuery event handlers and, 545
- event handlers
 - advanced event handler registration with jQuery, 544
 - ApplicationCache object, 860
 - defined, 10
 - defining for FileReader, 698, 926
 - defining for offline web application, 609
 - defining, onclick handler (example), 10
 - deregistering with jQuery, 546
 - Element objects, 909
 - EventSource object, 921
 - form and form element, 400
 - form controls, 931
 - Form objects, 929
 - functions for, 320
 - in HTML, 315
 - HTTP progress events, 507
 - HTTP upload progress event, 508
 - invoking, 460–465
 - event cancellation, 464
 - event handler argument, 460
 - event handler context, 460
 - event handler scope, 461
 - event propagation, 463
 - handler return value, 462
 - order of invocation, 462
 - jQuery, 542
 - mousewheel events, 472–474
 - onerror property, Window object, 351
 - properties defined by HTML elements, 375
 - properties of Window, Document, and Element objects, 309, 1007
 - registering (see registering event handlers)
 - text input elements, 403
 - WebSocket, 1000
 - Worker object, 1009
 - WorkerGlobalScope, 1011
 - XMLHttpRequest object, 1017
 - readystatechange events, 499
 - XMLHttpRequestUpload, 1018
- event listeners, 320
 - (see also event handlers)
 - same-origin policy, 334
- Event object, 542, 914–920
 - constants defining values of eventPhase property, 914
 - defaultPrevented property, 464
 - jQuery, 542
 - methods, 918
 - preventDefault() method, 464
 - properties, 915
 - proposed method, DOM Level 3 specification, 920
 - proposed properties, DOM Level 3 specification, 919
 - returnValue property, 464
 - stopImmediatePropagation() method, 465
 - stopPropagation() method, 464
- event propagation, 463
 - cancelling, 464
 - defined, 446
 - stopping, 919
- event property, Window object, 460, 1002
- event type, 445
- event-driven JavaScript, 320
- event-driven phase of execution, 318
- eventPhase property, Event object, 916
 - constants defining values of, 914
- events, 445–489
 - Ajax in jQuery, 570

- categories of, 448
- `createEvent()`, Document object, 895
- default actions associated with, 447
- defined, 445
- device-independent, 332
- document load, 465–466
- DOM (Document Object Model), 453
- drag and drop, 474–481
- ErrorEvent object, 913
- event emitters in Node, 297
- event objects, 446
- event propagation, 446
- event target, 445
- event type or event name, 445
- handling application cache events, 604–607
- handling with jQuery, 540–550
 - advanced event handler registration, 544
 - custom events, 549
 - deregistering event handlers, 546
 - event handler registrations methods, 540
 - Event object, 542
 - live events, 549
 - triggering events, 547
- hashchange event, 672
- HashChangeEvent object, 935
- HTML5, 454
- HTTP abort events, 510
- HTTP progress events, 507–510
- HTTP timeout events, 510
- implementing Java event listeners with
 - JavaScript in Rhino, 292
- invocation of event handlers, 460–465
- jQuery, 955
- keyboard, 484–489
- legacy event types, 449
- media, 620, 968
- message, 677, 681, 971, 972
- mouse, 466–470
- mousewheel, 471–474
- overview, 320
- page transition, 981
- popstate, 982
- progress, 983
- registering event handlers, 456–460
- Server-Sent Events with Comet, 515–521
- storage, 592, 988
- supported by Document objects, 898
- testing if mouse event is over current path in
 - canvas, 662
- testing if mouse event is over painted pixel
 - in canvas, 663
- text, 481–484
- touchscreen and mobile, 456
- types of, 447
- WebSocket, 713
- EventSource object, 515–521, 921
 - constants defining values of `readyState` property, 921
 - emulating with XMLHttpRequest, 517–519
 - using in simple chat client, 516
- EventTarget object, 922
- `every()` method, Array object, 154, 725
- excanvas.js library, 328
- exceptions
 - handling with try/catch/finally statements, 106
 - multiple catch clauses, 283
 - Java, handling as JavaScript exception in Rhino, 293
 - throwing, 106
 - thrown by Worker objects, 681
- `exec()` method, RegExp objects, 262, 585, 831
- `execCommand()`, Document object, 410, 895
- execution context, 353
 - event handlers, 460
- execution model, Worker threads, 683
- execution of JavaScript programs, 317–324
 - client-side threading model, 322
 - client-side timeline, 323
 - event-driven, 320
 - synchronous, asynchronous, and deferred scripts, 318
- `exp()` function, Math object, 794
- `expires` property, data saved with `userData`, 600
- explicit conversions of types, 47–49
- ExplorerCanvas project, 630
- exponential notation, 32, 48, 804
- expression closures, 282
- expression statements, 87
- expressions, 57
 - arithmetic, 66–70
 - + (addition or string concatenation) operator, 67

- bitwise operators, 69
 - unary arithmetic operators, 68
 - assignment, 77
 - assignment with operation, 78
 - comma operator (,), 85
 - conditional operator (?), 82
 - defined, 5
 - delete operator, 84
 - evaluation, 79–81
 - before for keyword in array
 - comprehensions, 281
 - function definition, 59, 164, 165
 - generator, 281
 - invocation, 61, 167
 - logical, 75
 - logical AND (&&) operator, 75
 - logical NOT (!) operator, 77
 - logical OR (||) operator, 76
 - object and array initializers, 58
 - object creation, 61
 - operator overview, 62–66
 - primary, 57
 - property access, 60, 168
 - relational, 71
 - comparison operators, 73
 - equality and inequality operators, 71
 - in operator, 74
 - instanceof operator, 75
 - statements versus, 6
 - typeof operator, 82
 - void operator, 85
 - extend() function, 127, 572
 - extensibility of objects, 818
 - extensible attribute, 116, 137
 - extensions, 269–287
 - constants and scoped variables, 269
 - destructuring assignment, 272
 - E4X (ECMAScript for XML), 283–287
 - iteration, 274–282
 - array comprehensions, 280
 - for/each loop, 274
 - generator expressions, 281
 - generators, 277–280
 - iterators, 274–277
 - preventing, 810, 821
 - preventing class extensions in ECMAScript 5, 242
 - shorthand functions (expression closures), 282
- ## F
- Facebook, FBJS secure subset, 269
 - factory functions
 - class factory function and method chaining, 231
 - creating and initializing new object, 200
 - factory methods, 227
 - fade effects
 - fadeIn() and fadeOut() methods, 551, 552
 - fadeOut(), 552
 - queued, animate() method and, 555
 - fadeout animation (example), 433–435
 - FALLBACK section, application cache
 - manifest, 603
 - false and true values, 45
 - FBJS secure subset, 269
 - feature testing for browsers, 329
 - Fibonacci numbers, generator function for, 278
 - FieldSet object, 923
 - File API specification, 455
 - FileEntry objects, 701
 - FileError object, 924
 - FileList object, 693
 - filename property, ErrorEvent object, 913
 - FileReader object, 924–927
 - constants defining values of readyState property, 925
 - event handlers, 926
 - events tracking progress of asynchronous I/O, 455
 - events triggered on, 455
 - methods, 925
 - properties, 925
 - reading Blobs, 698
 - FileReaderSync object, 699, 927
 - files
 - as Blobs, 693
 - File object, 923
 - local files and XMLHttpRequest, 495
 - monitoring HTTP upload progress, 508
 - Node file and filesystem API, 299
 - uploading with HTTP POST request, 505
 - files property
 - DataTransfer object, 478, 694, 889
 - Input object, 943
 - filesystems, 700–705
 - client-side storage by web applications, 588

- reading user-selected files with JavaScript, 333
- using asynchronous filesystem API, 701
- using synchronous filesystem API, 704
- working with files in local filesystem, 700
- FileUpload elements, value property, 334
- FileWriter objects, 701
- fill(), CanvasRenderingContext2D, 633, 875
- fillRect(), CanvasRenderingContext2D, 645, 876
- fills
 - clipped, 652
 - colors, gradients, and patterns in Canvas, 645–648, 868
- fillStyle property, CanvasRenderingContext2D, 635, 871
- fillText(), CanvasRenderingContext2D, 650, 876
- filter property (IE), 428
- filter() method
 - Array object, 154, 726
 - jQuery, 540, 579
- filtering user input, 481–483
- finally clauses (try/catch/finally), 106
- find() method, jQuery, 580
- finite numbers, 784
- Firebug extension (Firefox), 3
- Firefox, 347
 - (see also web browsers)
 - changes to History API in Firefox 4, 673
 - charCode property, 481
 - current version, 327
 - DOMMouseScroll events, 471
 - ECMAScript 5 array methods, 160
 - global compositing approach, 659
 - JavaScript versions and extensions, 265, 270
- first() and last() methods, jQuery, 578
- :first-line and :first-letter pseudo-elements (CSS), 370
- firstChild and lastChild properties, Node object, 371, 976
- firstElementChild, Element object, 373, 904
- fixed positioning of elements, 420
- fixed-point notation for numbers, 805
- flags in regular expressions, 259
- Flash plug-ins, scripting, 336
- Float32Array class, 996
- Float64Array class, 996
- floating-point literals, 32
- floating-point values, 31
 - binary floating-point and rounding errors, 34
- floor() function, Math object, 795
- fn prototype object (jQuery), 583
- focus events, 401, 449
 - bubbling focusin and focusout events, 453
 - event handler registration with jQuery, 541
 - keyboard focus for document elements, 452
- focus() method
 - Element object, 906
 - Window object, 1005
- font property, 416, 869, 871
 - text in canvas, 650
- fontFamily property, 437
- fonts
 - font-size and font-weight, and color style properties, 431
 - web fonts in CSS, 418
- for loops, 98
 - calling jQuery.each() instead, 529
 - continue statements in, 104
 - iterating arrays, 146
 - let keyword used as loop initializer, 270
 - variables declared with let keyword, 270
- for/each loops, 274
 - in array comprehensions, 281
 - defined in E4X, iteration through lists of XML tags and attributes, 286
- for/in loops, 100
 - in array comprehensions, 281
 - continue statements in, 104
 - enumerating properties, 126–128
 - extension to work with iterable objects, 274–277
 - iterating arrays, 147
 - iterating through methods, fields and properties of Java classes and objects, 292
 - let keyword in, 270
 - property enumeration order, 101
 - using with associative arrays, 121
- forEach() method, Array object, 148, 153, 726
- <form> elements

- action, encoding, method, and target
 - attributes, 399
 - method attribute, 399
 - setting form-submission attributes of, 375
 - triggering submit event on, 548
- Form object, 927–929
 - elements property, 399
 - event handlers, 929
 - methods, 929
 - properties, 928
 - referred to as `this.form` by form element
 - event handlers, 401
 - `submit()` and `reset()` methods, 400
- form property, 400, 401
 - FormControl object, 930
 - Label object, 962
 - Meter object, 973
 - Option object, 980
 - Progress object, 983
- form-encoded HTTP requests, 502–507
 - encoding an object, 502
 - file upload with POST request, 505
 - JSON-encoded body, 504
 - making GET request, 503
 - making HTTP POST request, 503
 - multipartform=data requests, 506
 - XML-encoded body, 504
- formAction property
 - Button object, 864
 - Input object, 943
- format control characters (Unicode), 22
- FormControl object, 929
 - event handlers, 931
 - methods, 931
 - properties, 930
- FormData object, 506, 931
- formEnctype property
 - Button object, 864
 - Input object, 943
- formMethod property
 - Button object, 864
 - Input object, 943
- formNoValidate property
 - Button object, 864
 - Input object, 943
- forms property, Document object, 367, 398, 893
- formTarget property
 - Button object, 865
- Input object, 943
- FormValidity object, 932
- forward() method, History object, 345, 936
- fractions, binary floating-point representations of, 35
- fragment identifiers in URLs, 672
- <frame> and <frameset> elements (deprecated), 353
- frameElement property, Window object, 357, 1002
- frames
 - inability to close, 356
 - multiple windows and frames, 353–360
 - relationships between frames, 356
 - viewport, 390
- frames property, Window object, 357, 1002
- frameworks, client-side, 328, 338
- freeze() function, 137, 814
- fromCharCode() method, String object, 839
- fromElement property, Event object, 916
- frozen objects, 814, 819
- fs (file and filesystem) module, Node, 299
- function calls, 96
 - (see also functions, invoking)
 - as expression statements, 88
- Function class, 30
 - toString() method, 50
- function declaration statements, 91, 165
- function definition expressions, 165
 - function declaration statements versus, 92
- function keyword, 59, 164
 - creating a variable, 358
- Function object, 775–781
 - apply() method, 776
 - arguments property, 777
 - bind() method, 189, 208, 778
 - call() method, 136, 779
 - caller property, 779
 - defining your own properties, 178
 - length property, 780
 - methods, 776
 - properties, 775
 - prototype property, 780
 - toString() method, 780
- function scope, 31
 - and hoisting, 54
 - as private namespace in modules, 248–250
- Function() constructor, 190
 - removal in secure subsets, 267

functions, 163–197

 Ajax utility functions in jQuery, 560

 arguments and parameters, 171–176

 argument types, 174

 optional parameters, 171

 using object properties as arguments,
 174

 variable-length argument lists, 172

 arguments[] array, 719

 bind() method, 188

 call() and apply() methods, 187

 callable objects, 191

 closures, 180–185

 constructor, 782

 defined, 6, 30

 defining, 164–166

 nested functions, 166

 definition expressions, 59

 demonstrating control structure statements,
 7

 Function() constructor, 190

 generator, 277–280

 global, 25, 781

 higher-order, 193

 invocation expressions, 61

 invoking, 166–170

 constructor invocation, 170

 indirect invocation, 170

 method invocation, 167

 invoking when document is ready, 465

 jQuery function (term), 528

 jQuery() function (\$()), 527

 Math, 790

 memoization, 196

 names, 165

 as namespaces, 178–180

 nested, 309

 partial application of, 194–196

 processing arrays, 192–193

 properties, 186

 length property, 186

 prototype, 186, 203

 restrictions in secure subsets, 267

 returning arrays of values, destructuring
 assignment with, 272

 setting event handler attributes to, 457

 sharing between frames or windows, 358

 shorthand for (expression closures), 282

 toString() method, 189

 as values, 176–178

 defining your own function properties,
 178

 variables declared with var or let, 271

G

g (global matching) in regular expressions,
 259

garbage collection, 30

generator expressions, 281

generators, 277–280

 pipeline of, 279

 restarting with send() or throw() method,
 280

Geocoordinates object, 933

Geolocation API, 668–671

 example demonstrating all features, 670

 using to display a map, 669

Geolocation object, 933

geolocation property, Navigator object, 348,
 668, 974

GeolocationError object, 934

geometrical, coordinate-based view of
 documents, 390

geometry and scrolling, document and element,
 390–396

 determining element positioned at a point,
 393

 document coordinates and viewport
 coordinates, 390

 element size, position, and overflow, 394

 getting and setting element geometry, 534

 handling mousewheel events (example),
 472–474

 querying geometry of an element, 392

 scrolling, 394

Geoposition object, 935

gesture and touch events, Safari on Apple
 iPhone and iPad, 456

GET method, 496

 making HTTP request with form-encoded
 data, 503

 no request body, 497

get() function, 563

getAllResponseHeaders(), XMLHttpRequest
 object, 1015

getAttribute() and setAttribute() methods,
 Element object, 376

getAttribute() method, Element object, 906

- getAttributeNS(), Element object, 907
- getBoundingClientRect(), Element object, 392, 396, 907
- getClientRects(), Element object, 393, 907
- getComputedStyle(), Window object, 436, 1005
- getContext() method, Canvas object, 630, 636
- getContext(), Canvas object, 865
- getCurrentPosition(), Geolocation object, 669, 934
- getData(), DataTransfer object, 478, 889
- getDate() method, Date object, 746
- getDay() method, Date object, 747
- getElementById(), Document object, 352, 364, 896
 - selecting forms and form elements, 398
- getElementsByClassName() method
 - Document object, 368, 896
 - Element object, 907
- getElementsByName() method,
 - HTMLDocument object, 365
- getElementsByTagName() method
 - Document object, 366, 896
 - selecting forms and form elements, 398
 - Element object, 907
- getElementsByTagName(), Document object, 159
- getElementsByTagNameNS() method
 - Document object, 896
 - Element object, 907
- getFloat32() method, DataView object, 890
- getFloat64() method, DataView object, 890
- getFullYear() method, Date object, 747
- getHours() method, Date object, 747
- getImageData(), CanvasRenderingContext2D, 661, 876
- getInt16() method, DataView object, 890
- getInt32() method, DataView object, 890
- getInt8() method, DataView object, 890
- getItem() method, Storage object, 988
- getJSON() function, 561
- getMilliseconds() method, Date object, 747
- getMinutes() method, Date object, 748
- getModifierState(), Event object, 920
- getMonth() method, Date object, 748
- getOwnPropertyDescriptor() function, 131, 134, 815
- getOwnPropertyNames() function, 128, 816
- getPrototypeOf() function, 135, 817
- getResponseHeader(), XMLHttpRequest object, 1015
- getScript() function, 561
- getSeconds() method, Date object, 748
- getSelection() method, Window object, 408
- getters and setters
 - jQuery, 531–537
 - for CSS attributes, 532
 - for CSS classes, 532
 - for element content, 534
 - for element data, 536
 - for element geometry, 534
 - for HTML attributes, 531
 - for HTML form values, 533
 - property, 128–130
 - combining closures with, 183
 - legacy API for, 134
- getTime() method, Date object, 748
- getTimezoneOffset() method, Date object, 749
- getUint16() method, DataView object, 890
- getUint32() method, DataView object, 890
- getUint8() method, DataView object, 890
- getUTCDate() method, Date object, 749
- getUTCDay() method, Date object, 750
- getUTCFullYear() method, Date object, 750
- getUTCHours() method, Date object, 750
- getUTCMilliseconds() method, Date object, 750
- getUTCMinutes() method, Date object, 751
- getUTCMonth() method, Date object, 751
- getUTCSeconds() method, Date object, 751
- getYear() method, Date object, 751
- global functions, 25, 781
 - defined in Rhino, 290
- global object, 42, 781–784, 781
 - properties referring to predefined JavaScript objects, 782
- global property, RegExp object, 262, 832
- global variables, 25, 31
 - avoiding by using functions as namespaces, 179
 - avoiding creation of, in modules, 246
 - as properties of global object, 55
 - restriction in secure subsets, 267
 - use of element IDs as, 352
- globalAlpha property, 645, 646, 871

- globalCompositeOperation property, 657, 869, 871
- globalEval() function, 572
- GMT (Greenwich Mean Time), 742
- go() method, History object, 345, 936
- Google
 - Caja secure subset, 268
 - Closure library, 339
 - promoting ability to load scripts from other sites, 313
 - V8 JavaScript engine and Node, 296
- Google Web Toolkit (GWT), 339
- graded browser support, 329
- gradients
 - CanvasGradient object, 866
- gradients in Canvas, 645–648
 - specifying for stroke or fill, 647
- graphics, 613, 630
 - (see also Canvas API)
 - scripting images, 613
 - SVG (Scalable Vector Graphics), 622–630
- graphics APIs for web applications, 311
- graphics attributes, canvas
 - graphics state management utilities, 636
- graphics attributes, Canvas API, 635
- graphics state (canvas), 635, 870
 - saving, 870, 879
 - state management utilities, 636
- grep() function, 572
- grep() method, jQuery, 579
- group() method, Console object, 882
- groupCollapsed() method, Console object, 882
- groupEnd() method, Console object, 883
- grouping in regular expressions, 256
- GUIs (graphical user interfaces), creating Java
 - GUI using JavaScript in Rhino (example), 293–296

H

- handler property, Event object, 544
- has() method, jQuery, 579
- hasAttribute() method, Element object, 377, 907
- hasAttributeNS(), Element object, 907
- hasChildNodes(), Node object, 978
- hasClass() method, jQuery, 532
- hasFocus() method, Document object, 896
- hash property

- Link object, 963
- Location object, 343, 672, 964
- WorkerLocation object, 1011
- hashchange events, 672
- HashChangeEvent object, 935
- hasOwnProperty() method, Object class, 125, 817
- HAVE constants, MediaElement object, 965
- HEAD method, HTTP request for link details with CORS, 512
- head property, Document object, 367, 893
- headers, HTTP request and response, 495
 - checking Content-Type response header, 501
 - Content-Length header, 508
 - CORS (Cross-Origin Resource Sharing) headers, 511
 - response headers, 498
 - setting for POST request file upload, 506
 - setting POST request Content-Type header, 502
 - setting request header, 496
- heading property, Geocoordinates object, 933
- heading variable, 352
- height and width style properties, 421, 425
- height property, 535
 - (see also width and height properties)
 - ClientRect object, 880
 - IFrame object, 939
- height() method, jQuery, 535
- hexadecimal digits, specifying RGB colors, 427
- hexadecimal values, 32
- hide() method, jQuery, 553, 555
- high property, Meter object, 973
- higher-order functions, 193
 - combined with partial application, 196
- history management in HTML5, 671–676
- History object, 345, 936
 - back(), forward() and go() methods, 345
 - pushState() method, 672
 - history management with (example), 673–676
 - replaceState() method, 673
- history property, Window object, 345, 1002
- hit detection, 662–663
- hoisting, 54, 165
- host objects, 116
- host property

- Link object, 963
- Location object, 343, 964
- WorkerLocation object, 1011
- hostname property
 - Link object, 963
 - Location object, 343, 964
 - WorkerLocation object, 1011
- :hover pseudoclass, 614
- hover() method, jQuery, 541, 545
- href attribute, <a> elements, 315, 367
- href property
 - CSSStyleSheet object, 887
 - Link object, 963
 - Location object, 343, 964
 - WorkerLocation object, 1012
- HSL (hue-saturation-value) color specification, 427
- HSLA (hue-saturation-value-alpha) color specification, 427
- HTML
 - attributes of elements, 375–378
 - case-insensitive tag names, 366
 - element content as, 379
 - elements and attributes, 910–913
 - embedding JavaScript code using <script> tags, 9
 - embedding JavaScript in, 311–317
 - escaping and inactivating HTML tags in untrusted data, 337
 - event handlers, 315
 - getting and setting attributes with jQuery, 531
 - including CSS stylesheet in HTML page, 415
 - keeping HTML content separate from JavaScript behavior, 458
 - not case-sensitive, 21
 - <script> element, 309
 - scripting document content, 9
 - String class methods, 836
 - strings, single- and double-quoted, 37
 - using JavaScript to script content, 10
 - <html> elements, manifest attribute, 601
- HTML forms, 397–404
 - elements, 397
 - events, 449
 - form and element event handlers, 400
 - form and element properties, 399
 - Form objects, 927–929
 - form-encoded HTTP requests, 502–507
 - FormControl objects, 929, 930
 - FormData objects, 931
 - FormValidity objects, 932
 - getting and setting values, 533
 - handler return value, preventing invalid input, 462
 - HTMLFormControlsCollection object, 938
 - new features in HTML5, 455
 - push buttons, 401
 - select and option elements, 403
 - selecting forms and form elements, 398
 - text fields, 402
 - toggle buttons, 402
- html() method, jQuery, 534
- HTML5, 667–716
 - APIs for web applications, 311
 - application cache, 601
 - Blobs, 691–699
 - classList property, 438, 533
 - client-side databases, 705–713
 - cross-origin messaging, 676–680
 - dataset attributes and dataset property, 377
 - DOMContentLoaded event, 465
 - drag and drop (DnD) API, 475
 - editing commands, 410
 - event handlers directed at browser as a whole, 457
 - events, 454
 - Filesystem API, 700–705
 - frames property as self-referential property, 357
 - Geolocation API, 668–671
 - getElementsByClassName() method, 368
 - history management, 671–676
 - innerHTML and outerHTML properties, 379
 - insertAdjacentHTML() method, 379
 - Offline Web Applications API, 588
 - placeholder attribute, text fields, 402
 - sandbox attribute for <iframe> element, 337
 - SVG markup appearing directly in HTML files, 624
 - typed arrays and ArrayBuffers, 687–691
 - “web workers”, 322
 - Web Workers specification, 680–687
 - WebSocket API, 713–716

- WindowProxy object, 359
 - HTMLCollection object, 367, 937
 - form elements, 398
 - overview of, 367
 - HTMLDocument object, 363
 - (see also Document object)
 - getElementsByName() method, 365
 - images, forms, and links properties, 367
 - HTMLElement object, 363
 - (see also Element object)
 - properties mirroring HTML attributes of elements, 375
 - representing HTML elements in documents, 351
 - text property, 314
 - htmlFor property
 - Label object, 962
 - Output object, 981
 - HTMLOptionsCollection object, 938
 - HTTP, 713
 - attributes of HTML elements, 375
 - client utilities module in Node (example), 302–304
 - HTTP server in Node (example), 300–302
 - scripted, 491–521
 - using <script> elements, 513–515
 - using Comet with Server-Sent Events, 515–521
 - using XMLHttpRequest, 494–513
 - HTTP methods, 496
 - hyperlinks, 316
 - bookmarking destination of, 317
 - Link objects, 962
 - onclick event handler, 401
 - hyphenated attribute names, 377
- I**
- i (case-insensitive matching) in regular expressions, 259
 - I/O
 - asynchronous, scripting with Node, 296–304
 - HTTP client (example), 302–304
 - HTTP server (example), 300–302
 - id attribute, HTML elements, 351
 - id property, Element object, 904
 - IDBRange object, 706
 - identifiers
 - defined, 23
 - format control characters and, 22
 - in labeled statements, 102
 - in property access expressions, 60
 - identity operator (see operators; strict equality operator)
 - if statements, 92
 - in array comprehensions, 281
 - in else if statements, 94
 - nested, with else clauses, 93
 - <iframe> elements, 334
 - Ajax transport with, 492
 - browsing history and, 345
 - contentWindow property, 357
 - deleting to close frames, 356
 - document properties for, referring to Window object, 366
 - making document editable in, 409
 - name attribute, 354
 - nested documents in HTML documents, 353
 - returning contents of, 581
 - sandbox attribute in HTML5, 337
 - using in history management, 346
 - with name and id attribute, becoming value of global variable, 353
 - IFrame object, 939
 - ignoreCase property, RegExp object, 262, 832
 - ImageData object, 661
 - ImageData objects, 941
 - copying onto the canvas, 878
 - creating, 874
 - data property, array of bytes, 688
 - passing to worker via postMessage(), 684
 - images
 - displaying dropped image files with Blob URLs, 695
 - drawing in canvas, 655–657, 868
 - extracting canvas content as, 656
 - Image object, 940
 - scripting, 613
 - unobtrusive image rollovers, 614
 - Web Worker for image processing (example), 684
 - images property, HTMLDocument object, 367
 - elements, 613
 - Ajax transport and, 492
 - displaying SVG images, 622
 - Immutable classes, defining, 239–241

- immutable objects, 137, 814, 819
- immutable types, 31
- implementation property, Document, 893
- importNode(), Document object, 382, 896
- importScripts(), WorkerGlobalScope object, 682, 1010
- in operator, 62, 74
 - testing for inherited or noninherited properties, 125
- inArray() method, 572
- increment expression (for loops), 98
- indeterminate property, Input object, 943
- index property, Option object, 980
- index() method, jQuery, 530
- IndexedDB API, 588, 705–713
 - database of U.S. postal codes (example), 708–713
- indexes
 - array, 141
 - indexOf() and lastIndexOf() methods, 157
 - object property names versus, 143
 - zero-based, string and array, 36
 - indexOf() method
 - Array object, 157, 727
 - String object, 840
- inequality operators (see ! (exclamation mark), under Symbols)
- infinite loops, 99
- Infinity property, 784
- infinity values, 33
- info() method, Console object, 883
- inheritance
 - accessor properties, 130
 - classes and prototypes, 200
 - creating new object that inherits from prototype, 119
 - enumeration of properties and, 126
 - favoring composition over, in object-oriented design, 233
 - object properties, 122
 - prototype constructor as prototype for new object, 201
 - subclasses, 230
- inherited properties, 116
- initEvent() method, Event object, 919
- initialize expression (for loops), 98
- initializer expressions, 5, 58
- initialTime, MediaElement object, 617, 967
- inline styles, scripting, 431–435
 - CSS animations, 433–435
 - setting while querying computed styles, 436
- innerHeight and innerWidth properties, Window object, 1002
- innerHTML property, Element object, 379, 386, 904
 - streaming API for, 407
 - use in jQuery to get element content, 534
 - using to implement outerHTML, 384
- innerText property, Element object, 380
- innerWidth() and innerHeight() methods, jQuery, 535
- <input> elements, buttons defined as, 401
 - file uploads with, 505
- input events
 - device-dependent and -independent, 448
 - triggered on text input form elements, 450
- Input object, 942
 - methods, 945
 - properties, 942
- input variable, 352
- inputMethod, Event object, 453, 481, 920
- insertAdjacentHTML(), Element object, 379, 908
 - implementing using innerHTML and DocumentFragment, 386
- insertAfter() method, jQuery, 538
- insertBefore() method, jQuery, 538
- insertBefore() method, Node object, 383, 978
- insertCell() method, TableRow object, 992
- insertData() method
 - Comment node, 881
 - Text node, 993
- insertion and deletion methods in jQuery, 953
- insertRow() method
 - Table object, 991
 - TableSection object, 993
- insertRule(), CSSStyleSheet object, 441, 888
- inspect(), ConsoleCommandLine, 884
- instance fields and methods (Java classes), 205
- instance objects, 205
- instanceof operator, 62, 75
 - determining class of an object, 210
 - inability to distinguish array type, 158
 - isPrototypeOf() method and, 135
 - not working across windows, 359

- using with constructors to test objects' class membership, 203
- working with Java objects and classes in Rhino, 291
- instances, 199
 - arrays of, sorting, 224
 - constructor property, 204
 - creating and initializing with factory function, 200
 - factory methods returning, 227
- Int16Array class, 996
- Int32Array class, 996
- Int8Array class, 688, 996
- integer literals, 32
- integers, 31
- interacting windows, JavaScript in, 358
- interfaces (see UI (user interface))
- Internet Explorer (IE)
 - <canvas> element, 630
 - clientInformation property, Window object, 346
 - computed styles, 437
 - conditional comments in, 331
 - CSS box model, 425
 - current version, 327
 - drag and drop (DnD) API, 475
 - event model
 - attachEvent() and addEventListener() methods, 321
 - attachEvent() and detachEvent() methods, 459, 923
 - event capturing and, 463
 - form events, 450
 - filter property, 428
 - innerText property instead of textContent, 380
 - nonbubbling mouse events, 452
 - not supporting
 - getElementsByClassName(), 369
 - propertychange event, using to detect text input, 484
 - querying selected text, 408
 - removing script tags and other executable content in IE8, 337
 - rules property instead of cssRules, 441
 - timer methods, 342
 - use of callable host objects rather than native Function objects, 191
 - userData API, 588
 - userData persistence, 599–601
 - XMLHttpRequest in IE 6, 494
- interpreters, 3
 - JavaScript versions, 269
 - support for E4X, 283
 - support for JavaScript extensions, 265
- invocation context, 163
- invocation expressions, 61
 - function invocation, 167
 - precedence, 65
- invocation order, event handlers, 462
- invoking functions, 166–170
 - constructor invocation, 170
 - event handlers, 460–465
 - indirectly, 170
 - jQuery() function, 526
 - method invocation, 167
- iPhone and iPad, gesture and touch events, 456
- is() method, jQuery, 530, 533
- isArray() function, 573
- isArray() method, Array object, 157, 160
- isContentEditable, Element object, 904
- isDefaultNamespace() method, Node object, 978
- isEmptyObject() function, 573
- isEqualNode() method, Node object, 978
- isExtensible() function, 137, 818
- isFinite() function, 34, 784
- isFrozen() function, 137, 819
- isFunction() function, 191, 573
- isNaN() function, 34
- isPlainObject() function, 573
- isPointInPath(), CanvasRenderingContext2D, 662, 877
- isPrototypeOf() method, 135, 819
- isSameNode() method, Node object, 978
- isSealed() function, 820
- isTrusted property, Event object, 916
- item() method
 - DOMTokenList object, 902
 - HTMLCollection object, 368, 937
 - HTMLOptionsCollection object, 939
 - NodeList object, 368, 980
 - Select element, 987
- items property, DataTransfer object, 478, 888
- iterable objects, 275
- iteration
 - arrays, 146–148

- for/each loop defined in E4X, 286
 - Java classes and objects with for/in loop, 292
 - language extensions for, 274–282
 - array comprehensions, 280
 - for/each loop, 274
 - generator expressions, 281
 - generators, 277–280
 - iterators, 274–277
 - Iterator () function, 276
 - __iterator__() method, 276
- J**
- Java
 - browser plug-ins, 336
 - scripting with Rhino, 289–296
 - GUI (graphical user interface) example, 293–296
 - Java-style classes in JavaScript, 205–208
 - javaEnabled(), Navigator object, 348
 - javaException property, Error object, 293
 - JavaScript
 - files ending in .js, 313
 - names and versions, 2
 - versions, 265, 269
 - JavaScript Object Notation (see JSON)
 - javascript: URLs, 315
 - using for bookmarklets, 316
 - join() method, Array object, 149, 728
 - jQuery, 11, 338, 523–586, 582–585, 945–962
 - Ajax functions, 958
 - Ajax in jQuery 1.5, 565
 - Ajax with, 559–571
 - Ajax events, 570
 - ajax() function, 564–569
 - load() method, 559
 - passing data to Ajax utilities, 562
 - utility functions, 560
 - altering document structure, 537–540
 - animated effects, 551–558
 - basic methods and properties, 947
 - basics of, 524
 - CSS selector-based queries, 370
 - effects and animation methods, 956
 - element methods, 950
 - event methods, 955
 - extending with plugins, 582–585
 - getters and setters, 531–537
 - handling events with, 540–550
 - insertion and deletion methods, 953
 - jQuery() function (\$()), 525–528
 - names of functions and methods in official documentation, 528
 - obtaining, 525
 - queries and query results, 528
 - selection methods, 578–582, 948
 - selector grammar, 946
 - selectors, 574–578
 - combinations of, 577
 - groups of, 578
 - simple, 575–577
 - terminology, 527
 - UI library, 585
 - utility functions, 571–574, 960
 - jquery property, 529
 - jQuery.easing object, 556
 - jQuery.fx.speeds, 551
 - jqXHR object, 565
 - JSON, 138, 786–789
 - jQuery.getJSON() function, 561
 - jQuery.parseJSON() function, 573
 - making HTTP POST request with JSON-encoded body, 504
 - methods, implementation in classes, 220
 - parsing HTTP response, 500
 - toJSON() method, 139
 - JSON.parse() function, 138, 786
 - JSON.stringify() function, 138, 672, 788
 - JSONP, 513–515
 - making request with script element, 515
 - request specified by URL or data string passed to jQuery.getJSON(), 563
 - jump statements, 87, 102
 - break, 103
 - continue, 104
 - labeled, 102
 - return, 105
 - throw, 106
 - try/catch/finally, 106
- K**
- key events, 452
 - key property, 485
 - Event object, 454, 920
 - StorageEvent object, 988
 - key() method, Storage object, 988
 - keyboard events, 484–489
 - event handler registration with jQuery, 541

- keydown and keyup, 485
- Keymap class for keyboard shortcuts
 - (example), 485–489
- text input, 481
- keyboard modifier keys for mouse events, 451, 467
- keyboard, using instead of a mouse, 332
- KeyboardEvent objects, 453
- keyCode property, 481, 484
 - Event object, 917
- keyIdentifier property, 485
- Keymap class for keyboard shortcuts
 - (example), 485–489
- keys() function, 821
 - enumerating property names, 128
- keys() method, ConsoleCommandLine, 884
- keywords, case sensitivity, 21
- Koch snowflakes, drawing (example), 641

L

- Label object, 962
- label property, Option object, 980
- labeled statements, 102
 - continue, 104
- labels property
 - FormControl object, 930
 - Meter object, 973
 - Progress object, 983
- lang property, Element object, 904
- lastChild property, Node object, 976
- lastElementChild, Element object, 373, 905
- lastEventId, MessageEvent object, 971
- lastIndex property, RegExp object, 262, 833
- lastIndexOf() method
 - Array object, 157, 729
 - String object, 841
- lastModified property, Document object, 405, 893
- lastModifiedDate, File object, 693, 923
- latitude property, Geocoordinates object, 933
- left and top style properties, 421, 425
- left, right, top, and bottom properties, 392
 - ClientRect object, 880
- legacy event types, 449
 - form events, 449
 - key events, 452
 - mouse events, 451
 - window events, 450
- length of a string, 36

- determining, 38
- length property
 - Arguments object, 172, 721
 - Array object, 730
- arrays, 143
 - manipulating, 144
 - sparse arrays, 144
- Comment node, 881
- CSSStyleDeclaration object, 886
- DOMTokenList object, 902
- Form object, 928
- Function object, 780
- functions, 186
- History object, 936
- HTMLCollection object, 937
- HTMLOptionsCollection object, 938
- jQuery objects, 528
- NodeList object, 980
- Select element, 986
- Storage object, 987
- String object, 38, 842
- Text node, 993
- TimeRanges object, 995
- TypedArray object, 997
- Window object, 357, 1002
- lengthComputable, ProgressEvent object, 984
- let keyword, 269
 - destructuring assignment used to initialize variables, 272
 - used as loop initializer, 270
 - using as variable declaration, 270
- lexical scoping, 31, 55, 180
 - and functions shared between frames or windows, 358
 - rules for nested functions, 181
- libraries
 - client-side, supporting visual effects, 435
 - compatibility, 328
 - history management, 346
- lifecycle notification events, 454
- line breaks
 - interpreted as semicolon, exceptions, 26
 - in JavaScript code, 22
 - treatment as semicolons in JavaScript, 25
- linear gradients, 647, 874
- lineCap property, 650
- lineCap property,
 - CanvasRenderingContext2D, 649, 871

- lineCap, CanvasRenderingContext2D, 868
- lineJoin, CanvasRenderingContext2D, 650, 868, 872
- lineno property, ErrorEvent object, 913
- lines
 - drawing in Canvas API, 632–635
 - line drawing attributes in canvas, 648
- lineTo(), CanvasRenderingContext2D, 632, 641, 649, 877
- lineWidth property,
 - CanvasRenderingContext2D, 635, 648
- lineWidth, CanvasRenderingContext2D, 868, 872
- <link> elements, Element objects representing, 440
- links
 - Link object, 962
 - requesting details about with HTTP HEAD and CORS, 512
- links property, Document object, 367, 893
- list property, Input object, 943
- lists, filtering in E4X, 285
- literals, 58
 - defined, 23
 - numeric, 31
 - object, 117
 - regular expression, 251
 - string, 36
 - XML literals included in JavaScript code, 284
- little-endian byte ordering, 690, 890
- live events, 549
- live() method, jQuery, 550
- LN10 constant (Math), 795
- LN2 constant (Math), 795
- load events, 318, 450
 - browser support for, 324
 - document, 465–466
 - event propagation, 463
 - FileReader object, 698
 - onload event handler, Window object, 309
 - registering event handler for, 11
- load() function (Rhino), 290
- load() method
 - Ajax utility in jQuery, 559
 - jQuery, 541
 - media elements, 617
 - MediaElement object, 969
- loaded property, ProgressEvent, 508, 984
- loading and executing scripts asynchronously, 319
- loan calculator web application (example), 12–18
- local files and XMLHttpRequest, 495
- locale property, Event object, 920
- localeCompare() method, String object, 842
- localName property, Attr object, 862
- localName property, Element object, 905
- localStorage property, Window objects, 589, 1002
 - offline web applications, 608
 - storage API, 591
 - storage events, 592
 - storage lifetime and scope, 590
- Location object, 343, 964
 - assign() and replace() methods, 344
 - hash property, 672
 - href property, 343
 - reload() method, 344
 - URL decomposition properties, 343
- location property
 - Document object, 343, 405, 893
 - Event object, 920
 - Window object, 307, 343, 1002
 - WorkerGlobalScope object, 683, 1009
- log() function, Math object, 796
- log() method, Console object, 686, 883
- LOG10E constant (Math), 796
- LOG2E constant (Math), 796
- logical expressions, 75
 - logical AND (&&) operator, 75
 - logical NOT (!) operator, 77
 - logical OR (||) operator, 76
- logical operators, 62
- long-running scripts, 338
- longitude property, Geocoordinates object, 933
- lookahead assertions in regular expressions, 258
- __lookupGetter__() and __lookupSetter__() methods, 134
- lookupNamespaceURI(), Node object, 979
- lookupPrefix() method, Node object, 979
- loop property, MediaElement object, 618, 967
- loops, 87, 97
 - continue statements in, 104
 - do/while, 98

- for, 98
- for/each, 274
- let keyword as variable declaration or loop initializer, 270
- while, 97
- low property, Meter object, 973
- lvalues, 64, 100

M

- m (multiline mode) in regular expression
 - pattern matching, 259
- makeArray() function, 573
- malicious scripts, containment with same-origin policy, 334
- manifest files, application cache, 601
 - complex manifests, 603
 - updates, 603–607
- map() function, 573
- map() method
 - Array and jQuery, 530
 - Array object, 154, 192, 730
- map, displaying using geolocation, 669
- margin property, 416, 432
- margins
 - in CSS box model, 424
 - specifying for elements with CSS, 423
- match position, specifying for regular expressions, 257
- match() method, String object, 260, 843
- Math object, 789–800
 - abs() function, 791
 - acos() function, 791
 - asin() function, 792
 - atan() function, 792
 - atan2() function, 792
 - ceil() function, 793
 - constants, 789
 - cos() function, 794
 - exp() function, 794
 - floor() function, 795
 - functions and constants defined as properties, 33
 - log() function, 796
 - max() function, 187
 - max() method, 797
 - min() function, 797
 - pow() function, 798
 - random() function, 798
 - round() function, 798

- sin() function, 799
- sqrt() function, 799
- static functions, 790
- tan() function, 800
- max property
 - Input object, 943
 - Meter object, 973
 - Progress object, 983
- max() function, Math object, 187, 797
- max-age attribute, cookies, 596
- maximumAge option, Geolocation methods, 934
- maxLength property, Input object, 943
- MAX_VALUE (Number), 803
- measureText(), CanvasRenderingContext2D, 651, 877
- media
 - controlling playback, 617
 - events, 620
 - querying status, 618
 - scripting audio and video, 615
 - type selection and loading, 617
- media property
 - CSSStyleSheet object, 887
 - Style object, 989
- MediaElement object, 965–970
 - constants defining values for networkState and readyState, 965
 - event handlers, 968
 - methods, 969
 - properties, 966
- MediaError object, 970
- memoization, 196
- merge() function, 573
- message property
 - Error object, 769, 771
 - ErrorEvent object, 913
 - EvalError object, 774
 - GeolocationError object, 935
 - ReferenceError object, 829
 - URIError object, 856
- MessageChannel object, 684, 970
- MessageEvent object, 971
- MessagePort object, 684, 972
- messaging
 - cross-document, 336, 676–680
 - message event used for asynchronous communication, 455

- sending data to Worker objects with `postMessage()`, 681
 - support by WebSocket API, 714
 - metadata affecting media playback, 618
 - metaKey property, 451, 467, 484
 - Event object, 917
 - in jQuery, 543
 - Meter object, 973
 - method attribute and property, form elements, 399
 - method chaining, 169
 - from subclass to superclass, 231–233
 - method property, Form object, 928
 - methods, 30, 43, 163
 - adding to prototype objects for classes, 208
 - array
 - ECMAScript 3, 148–152
 - ECMAScript 3 and ECMAScript 5, 159
 - ECMAScript 5, 153–157
 - borrowing, 224
 - class, 199
 - common object methods, 138
 - `toJSON()`, 139
 - `toLocaleString()`, 139
 - `toString()`, 139
 - `valueOf()`, 140
 - comparison, 221–224
 - defined, 7
 - invocation expressions, 61
 - invoking, 167
 - Java, invoking with JavaScript programs in Rhino, 291
 - jQuery, 528
 - special, restriction in secure subsets, 267
 - type conversion, 219
 - Microsoft Web Sandbox, 269
 - MIME types
 - application cache manifest files, 602
 - form data encoding, 502
 - media, 617
 - overriding incorrect type in HTTP response, 501
 - specifying in HTTP Content-Type headers, 496
 - text, 500
 - min property
 - Input object, 943
 - Meter object, 973
 - `min()` function, Math object, 797
 - MIN_VALUE (Number), 803
 - miterLimit property, 650, 872
 - mobile devices, events, 448, 456
 - modal dialog boxes, 349
 - (see also dialog boxes)
 - modules, 246–250
 - evaluation of, 302
 - function scope as private namespace, 248–250
 - objects as namespaces, 246
 - `monitorEvents()`, ConsoleCommandLine object, 884
 - motion blur or smear effect on graphics in canvas, 661
 - mouse
 - coordinates of pointer, 390
 - two-dimensional mouse wheels and wheel event, 453
 - using keyboard instead of, 332
 - mouse events, 451, 466–470
 - default actions that can be prevented, 466
 - DOM Level 3 Events specification, 453
 - registering event handlers with jQuery, 540
 - testing if event is over painted pixel in canvas, 663
 - testing whether event is over current path in canvas, 662
 - mousewheel events, 452, 453, 471–474
 - handling, 472–474
 - working with, interoperably, 471
 - `moveTo()`, CanvasRenderingContext2D, 632, 877
 - Mozilla
 - downloading Rhino from, 290
 - JavaScript versions, 265, 269
 - multidimensional arrays, 148
 - multiline property, RegExp object, 262, 830
 - multipart/form-data HTTP requests, 506
 - multiple property
 - Input object, 944
 - Select element, 986
 - mutable types, 31
 - muted property
 - MediaElement object, 967
 - setting for audio playback, 617
- ## N
- `\n` (newline character), 37
 - name attribute, HTML elements, 352, 399

- selecting elements by name, 365
- setting, creating properties on Document object, 366
- name property
 - Attr object, 862
 - DocumentType object, 899
 - DOMException object, 900
 - Error object, 770, 771
 - EvalError object, 774
 - File object, 693, 924
 - form elements, 400
 - Form object, 928
 - FormControl object, 930
 - IFrame object, 939
 - ReferenceError object, 829
 - URIError object, 856
 - Window object, 354, 1003
- namedItem() method
 - HTMLCollection object, 368, 937
 - HTMLOptionsCollection object, 939
 - Select element, 987
- names
 - CSS properties in JavaScript, 431
 - of functions, 165
 - window name, importance of, 354
- namespace URI property, Attr object, 862
- namespaces
 - createElementNS() method, 382
 - createElementNS() method, Document, 895
 - function scope as private namespace, in modules, 248–250
 - functions as, 178–180
 - getAttributeNS(), Element object, 907
 - getElementsByTagNameNS() method, 896
 - isDefaultNamespace(), Node object, 978
 - jQuery, 527
 - jQuery plug-in binding event handlers, 584
 - lookupPrefix(), Node object, 979
 - namespaceURI property, Element object, 905
 - objects as, 246
 - process namespace, Node, 297
 - specifying for event handlers with jQuery, 545
 - SVG, 627
 - XML documents including attributes from other namespaces, 377
 - XML, working with in E4X, 286
- NaN (not-a-number), 34, 800
 - equality comparisons and, 34
 - isNaN() function, 785
 - Number.NaN, 803
- native objects, 116
- naturalHeight and naturalWidth properties, Image object, 941
- Navigator object, 330, 346, 974
 - browser sniffing properties, 346
 - cookieEnabled property, 594
 - geolocation property, 668
 - miscellaneous properties and methods, 348
 - onLine property, 608
- navigator property
 - Window object, 346, 1003
 - WorkerGlobalScope object, 683, 1009
- negative infinity, 33, 804
- negative lookahead assertions in regular expressions, 258
- negative zero value, 33
 - equality comparisons to positive zero, 34
- nested functions, 166, 309
- NETWORK section, application cache manifest, 603
- networking
 - client-side JavaScript and, 333
 - TCP-based, net module in Node, 300
- networkState, MediaElement object, 620, 965, 967
- new keyword
 - instantiating Java classes, 291
 - preceding object creation expressions, 61
- new operator
 - constructor invocation with, 202
 - creating objects, 117
- newURL property, HashChangeEvent object, 935
- newValue property, StorageEvent object, 988
- next() and prev() methods, jQuery, 581
- next() method
 - generators, 278
 - iterators, 275
- nextAll() and prevAll() methods, jQuery, 581
- nextElementSibling, Element object, 373, 905
- nextSibling property, Node object, 371, 976
- nextUntil() and prevUntil() methods, jQuery, 581
- noConflict() function, 527, 583
- Node interpreter, 289

- chat server using WebSockets and, 715
 - online documentation, 296
 - scripting asynchronous I/O with, 296–304
 - buffers, 298
 - client-side timer functions, 297
 - event emitters, 297
 - file and filesystem API in `fs` module, 299
 - functions, 296
 - HTTP client utilities module (example), 302–304
 - HTTP server (example), 300–302
 - streams, 298
 - TCP-based networking, 300
 - Node object, 975–979
 - `appendChild()` method, 383
 - `attributes` property, 378
 - `cloneNode()` method, 382
 - constants, possible values of `nodeType` property, 976
 - constants, return value of `compareDocumentPosition()`, 976
 - documents as trees of, 371
 - `insertBefore()` method, 383
 - methods, 977
 - properties, 976
 - `removeChild()` method, 384
 - `replaceChild()` method, 384
 - `textContent` property, 380
 - NodeList object, 365, 896, 979
 - overview of, 367
 - returned by `getElementsByTagName()`, 366
 - `nodeName` property, Node object, 372, 976
 - nodes, 363
 - creating, inserting, and deleting in documents, 382–387
 - creating nodes, 382
 - inserting nodes, 382, 383
 - removing and replacing nodes, 384
 - using DocumentFragments, 385
 - `nodeType` property, Node object, 371, 977
 - `nodeValue` property, Node object, 372, 381, 977
 - nonzero winding rule, 635
 - normalization
 - normalized, defined, 995
 - Unicode encodings, 23
 - `normalize()` method, Node object, 979
 - `not()` method, jQuery, 579
 - not-a-number (see NaN)
 - `noValidate` property, Form object, 929
 - `now()` method, Date object, 752
 - null values, 29, 41
 - properties with, access errors, 123
 - property access expressions and, 60
 - Number object, 801–808
 - constants, 801
 - `MAX_VALUE`, 802
 - methods, 801
 - `MIN_VALUE`, 803
 - NaN, 803
 - `NEGATIVE_INFINITY`, 803
 - `POSITIVE_INFINITY`, 804
 - read-only values for Infinity and NaN global variables, 34
 - `toExponential()` method, 804
 - `toFixed()` method, 805
 - `toLocaleString()` method, 806
 - `toPrecision()` method, 806
 - `toString()` method, 48, 807
 - `valueOf()` method, 808
 - `Number()` constructor, 43, 801
 - `Number()` function, type conversions with, 47
 - numbers, 31–36
 - arithmetic in JavaScript, 33
 - binary floating-point and rounding errors, 34
 - conversions, 45, 47
 - number-to-string, 48
 - object-to-number, 49, 51
 - dates and times, 35
 - defining a complex number class (example), 206
 - floating-point literals, 32
 - integer literals, 32
 - wrapper objects, 43
 - numeric literals, 31
 - numeric types, 29
- ## O
- Object class, 75, 808–826
 - constructor property, 809, 811
 - `create()` function, 118, 133, 811
 - `defineProperties()` function, 132, 812
 - `defineProperty()` function, 101, 132, 134, 145, 813
 - `freeze()` function, 137, 814

- getOwnPropertyDescriptor() function, 131, 134, 815
- getOwnPropertyNames() function, 128, 816
- getPrototypeOf() function, 135, 817
- hasOwnProperty() method, 125, 817
- isExtensible() function, 137, 818
- isFrozen() function, 137, 819
- isPrototypeOf() method, 135, 819
- isSealed() function, 820
- keys() function, 128, 821
- methods, 138, 809
- preventExtensions() function, 137, 821
- propertyIsEnumerable() method, 125, 822
- prototype property, 118
- seal() function, 137, 823
- static methods, 809
- toLocaleString() method, 139, 823
- toString() method, 139, 824
- valueOf() method, 140, 825
- object creation expressions, 61
- object databases, 705
- <object> elements, 537
 - displaying SVG images, 622
- object literals, 117
- object types, 29
- Object() function, type conversions with, 47
- object-oriented programming
 - favoring composition over inheritance, 233
 - separating interface from implementation, 234
 - strongly typed languages, 199
- object-oriented techniques in JavaScript, 215
 - borrowing methods, 224
 - constructor overloading and factory methods, 227
 - emulating private instance fields, 226
 - enumerated types (example), 217–219
 - implementing comparison methods in classes, 221–224
 - Set class (example), 215–217
 - standard conversion methods, 219
- objects, 5, 115–140
 - array-like, 158
 - arrays of, 141
 - attributes of, 116, 135–138
 - class attribute, 136
 - extensible attribute, 137
 - prototype attribute, 135
 - callable, 191
 - conversion to primitives, 49–52
 - conversion to strings, 672
 - conversion to strings for Ajax, 559
 - conversions, 45
 - creating, 116–119
 - using new operator, 117
 - using object literals, 117
 - using Object.create() function, 118
 - using prototypes, 118
 - deleting properties, 124
 - determining class of, 210
 - using constructor name as class identifier, 211
 - using constructor property, 211
 - using duck-typing, 213
 - using instanceof operator, 210
 - enumerating properties, 126–128
 - form-encoding properties for HTTP request, 502
 - global object, 42
 - initializers, 58
 - instanceof operator, 75
 - iterable, 275
 - iterating through properties with
 - jQuery.each() function, 572
 - Java, querying and setting fields in Rhino, 291
 - jQuery, 527
 - methods, 7, 163
 - universal, 138
 - mutable object references, 44
 - properties, 115
 - property access expressions, 60
 - property attributes, 131–134
 - property getters and setters, 128–130
 - querying and setting properties, 120–124
 - inheritance and, 122
 - objects as associative arrays, 120
 - property access errors, 123
 - serializing, 138
 - structured clones of, 672
 - testing properties, 125
 - wrapper, 43
 - XML, 284
- octal values, 32
- offline web applications, 588, 608–612
 - application storage and, 601
 - events, 455

- offset properties, document elements, 394
- offset() method, jQuery, 534
- offsetHeight and offsetWidth properties,
 - Element object, 394, 905
- offsetLeft and offsetTop properties, Element object, 394, 905
- offsetParent property, Element object, 395, 905
- offsetParent() method, jQuery, 535
- offsetX and offsetY properties, Event object, 917
- oldURL property, HashChangeEvent object, 935
- oldValue property, StorageEvent object, 988
- onbeforeunload event handler, windows, 462
- onchange attribute, 315
- onchange event handlers
 - text fields, 403
 - toggle buttons in forms, 402
- onclick event handlers, 316
 - button elements in forms, 401
- ondragstart event handler, 476
- one() method, jQuery, 545
- onerror property
 - Window object, 351, 450
 - setting to a function, 321
 - WorkerGlobalScope object, 683
- onhashchange property, Window object, 672
- onLine property
 - Navigator object, 348, 974
 - WorkerNavigator object, 1012
- onload event handlers, 318
 - in client-side script revealing content, 309
 - in digital clock program, 312
- onload() function, defining (example), 322
- onmessage event handlers, 336
- onmessage property, WorkerGlobalScope, 682
- onmousedown attribute, <div> element, 470
- onreadystatechange property,
 - XMLHttpRequest object, 499, 1013, 1017
- onreset event handlers, form elements, 400
- onstorage property, Window objects, 592
- onsubmit event handlers, form elements, 400
- ontimeout property, XMLHttpRequest object, 510
- opacity property, 428, 551
 - animating, 552, 556
 - animating changes for fadeable elements, 435
- open() method
 - Document object, 897
 - Window object, 354, 1006
 - javascript:URL as argument, 315
 - XMLHttpRequest object, 1015
- opener property, Window object, 356, 1003
- opening and closing tags of elements, 379
- opening and closing windows, 354–356
 - closing windows, 356
- OpenSocial API, 268
- Opera, 455
 - (see also web browsers)
 - current version, 327
 - global compositing approach, 659
- operating systems
 - drag-and-drop operations based on, 454
 - web browsers as, 310
- operators, 5, 57, 62–66
 - + (addition or string concatenation)
 - operator, 67
 - arithmetic, 66
 - assignment, 78
 - associativity, 65
 - bitwise, 69
 - comparison, 73
 - conditional (?:), 82
 - delete, 84
 - equality and inequality, 71
 - in, 74
 - instanceof, 75
 - list of JavaScript operators, 62
 - logical AND (&&), 75
 - logical NOT (!), 77
 - logical OR (||), 76
 - lvalues, 64
 - number of operands, 64
 - operand and result type, 64
 - order of evaluation, 66
 - precedence, 65
 - side effects, 64, 88
 - typeof, 82
 - unary arithmetic operators, 68
 - void, 85
- optimum property, Meter object, 973
- <option> elements, 938
- Option object, 404, 980
- options property, Select element, 986

- orientation property, Window object, 456
- orientationchanged event, 456
- origin of a document, 334, 590
- origin property, MessageEvent object, 677, 971
- Origin: request header, 335
- originalEvent property, Event object, 544
- outerHeight and outerWidth properties, Window object, 1003
- outerHTML property, Element object, 379, 905
 - implementing using innerHTML, 384
- outerWidth() and outerHeight() methods, jQuery, 535
- Output object, 981
- overflow, 33
- overflow property, 428, 451
- overflowing content, scrollable elements in documents, 396
- overlapping translucent windows (CSS example), 429–431
- overloading constructors, 227
- overrideMimeType(), XMLHttpRequest object, 501, 1016
- own properties, 116
- ownerDocument property, Node object, 977
- ownerNode property, CSSStyleSheet object, 887
- ownerRule property, CSSStyleSheet object, 887

P

- padded-JSON (see JSONP)
- padding
 - in CSS box model, 424
 - specifying for elements with CSS, 423
- padding property, 416
- PageTransitionEvent object, 981
- pageX and pageY properties, Event object, 917
 - in jQuery, 543
- pageXOffset and pageYOffset properties, Window object, 391, 1003
- paragraph separators, 22
- param() function, 562
- parameters (function), 163, 171
 - optional, 171
- parent property, 356
 - Window object, 1003

- parent window, 354
- parent() and parents() methods, jQuery, 581
- parentNode property, Node object, 371, 977
- parentRule property, CSSRule object, 885
- parentStyleSheet property
 - CSSRule object, 886
 - CSSStyleSheet object, 887
- parentsUntil() method, jQuery, 581
- parse() function, 138, 786
- parse() method, Date object, 752
- parseFloat() function, 49, 826
- parseInt() function, 49, 827
- parseJSON() function, 573
- partial application of functions, 188, 194–196
- path attribute, cookies, 594
 - setting, 596
- pathname property
 - Link object, 963
 - Location object, 343, 965
 - WorkerLocation object, 1012
- paths
 - canvas, 631
 - beginPath() method, 873
 - closePath() method, 874
 - creating and rendering, 867
 - curved paths, 643
 - defining, 632
 - determining if point is on, 877
 - determining whether point is in, 635
 - line drawing attributes, 648
 - testing if mouse event is over current path, 662
 - SVG, 627
- pattern matching, 39
 - (see also regular expressions)
 - string methods for, 259
- pattern property, Input object, 944
- patternMatch property, FormValidity object, 932
- patterns
 - fill or stroke using, 647, 867
 - specifying for fill or stroke in canvas, 647
- pause() method, MediaElement object, 617, 970
- paused property, MediaElement object, 967
- period (.) (see . (dot), under Symbols)
- persisted property, PageTransitionEvent object, 982
- persistent filesystem, 700

- PI constant (Math), 797
- pie chart, drawing with JavaScript and SVG, 624–627
- pipeline of generators, 279
- pixelDepth property, Screen object, 984
- pixels
 - compositing in canvas, 657–660, 869
 - enlarged withdrawImage() in canvas, 656
 - manipulation in canvas, 661–662, 870
 - testing if mouse event is over painted pixel in canvas, 663
- placeholder attribute, text fields, 402
- placeholder property, Input object, 944
- platform property
 - Navigator object, 347, 974
 - WorkerNavigator object, 1012
- play() method, MediaElement object, 617, 970
- playbackRate property, MediaElement object, 618, 967
- played property, MediaElement object, 619, 967
- plug-ins
 - scripting in browsers, security implications, 336
- plugins property
 - Document object, 893
 - HTMLDocument object, 367
- PNG image format, canvas content returned in, 656
- point, determining element at, 393
- polygons, drawing with Canvas methods, 633
- pop ups, blocking by browsers, 355
- pop() method, Array object, 151, 731
- popstate events, 672
- PopStateEvent object, 982
- port property
 - Link object, 963
 - Location object, 343, 965
 - WorkerLocation object, 1012
- port1 and port2 properties, MessageChannel object, 971
- ports property, MessageEvent object, 971
- position property, 420
 - Progress object, 983
- position() method, jQuery, 535
- positioning elements
 - CSS box model, 424
 - CSS example, shadowed text, 422
 - document and element geometry and scrolling, 390–396
 - getting and setting element geometry in jQuery, 534
 - handling mousewheel events (example), 472–474
 - using CSS, 420–423
- positive infinity, 33, 804
- positive lookahead assertions in regular expressions, 258
- POSIX (Unix) API, use in Node, 289
- POST method, 696
 - encoding HTTP request body with XML document, 504
 - file upload with HTTP request, 505
 - making HTTP request with form-encoded data, 503
 - making HTTP request with JSON-encoded body, 504
 - posting plain text to a server, 497
 - request body, 497
- post() function, 563
- poster property, Video object, 998
- postMessage() method
 - MessagePort object, 972
 - Window object, 336, 677, 1006
 - Twitter search gadget controlled by, 678–680
 - Worker object, 681, 1008
 - WorkerGlobalScope object, 1010
- pow() function, Math object, 798
- precedence, operator, 65
- precision for numbers, 807
- prefix property
 - Attr object, 862
 - Element object, 905
- preload property, MediaElement object, 618, 967
- prepend() method, jQuery, 538
- prependTo() method, jQuery, 538
- prev() method, jQuery, 581
- prevAll() method, jQuery, 581
- preventDefault() method, Event object, 464, 548, 919
- preventExtensions() function, 137, 821
- previousElementSibling, Element object, 373, 905
- previousSibling property, Node object, 371, 977

- prevUntil() method, 581
- primary expressions, 57
- primitive types, 29
 - conversion of JavaScript primitives to Java, 289, 293
 - conversions to other types, 46
 - immutable primitive values and mutable object references, 44
 - object to primitive conversions, 49–52
- print() function (Rhino), 290
- print() method, Window object, 1006
- printing
 - beforeprint and afterprint events, 455
 - defining jQuery.fn.println() function, 583
- private properties, 208
 - emulating private instance fields in JavaScript, 226
- procedures, 166
- processData option, 562
- ProcessImageInstruction object, 982
- ProcessingInstruction object, creating, 895
- profile() method
 - Console object, 883
 - ConsoleCommandLine object, 884
- profileEnd() method
 - Console object, 883
 - ConsoleCommandLine object, 884
- programs, JavaScript, 317
- progress events, 455, 498, 507–510
 - HTTP, 507
 - upload, 508
 - XMLHttpRequest Level 2 (XHR2), 1017
- Progress object, 982
- ProgressEvent object, 983
- prompt() method, Window object, 348, 1006
- propagation of events (see events, event propagation)
- properties
 - attributes of, 116, 131–134
 - class, 199
 - computed style, 436
 - converting HTML attribute names to, 376
 - CSS style properties, 414
 - dataset attributes converted to, 377
 - defined, 115
 - deleting, 124
 - enumerating, 126–128
 - event handler, 457
 - mirroring HTML attributes, 315
 - naming convention, 321
 - form and form element, 399
 - function, 186
 - Function object, defining your own, 178
 - functions assigned to, 176
 - getters and setters, 128–130
 - global, 781
 - HTMLElement, mirroring HTML element attributes, 375
 - important CSS style properties, 419
 - inherited, checking for, 817
 - iterating through, with for/in loops, 100
 - enumeration order, 101
 - making nonenumerable, 238
 - names and values, 117
 - naming conventions for CSS properties in JavaScript, 431
 - nonstandard CSS properties, 416
 - Object.defineProperty() method, 812
 - private, in Java-style classes, 208
 - property names versus array indexes, 143
 - prototype, inheritance of, 118
 - querying and setting, 120–124
 - access errors, 123
 - inherited properties, 122
 - objects as associative arrays, 120
 - shortcut properties in CSS, 416
 - special, restriction in secure subsets, 267
 - testing, 125
 - using as function arguments, 174
 - variables as, 55
- property access expressions, 60
 - method invocations, 168
 - precedence, 65
- property descriptors, 131, 815
 - ECMAScript 5 properties utilities, 244–246
 - getting for named property of an object, 131
 - utility functions for, 239
- propertyIsEnumerable() method, 125, 822
- __proto__ property, 136
- protocol property
 - Link object, 963
 - Location object, 343, 965
 - WebSocket object, 714, 1000
 - WorkerLocation object, 1012
- prototype attribute, 135
- prototype chain, 118

- Prototype framework, Scriptaculous library, 435
 - Prototype library, 338
 - prototype property
 - Function object, 780
 - functions, 186
 - restrictions in secure subsets, 267
 - prototypes, 118, 205, 384, 811
 - adding methods to augment classes
 - inheriting from, 208
 - Array.prototype, 141
 - classes and, 200
 - constructor function, 203
 - constructor property and, 204
 - constructor, used as prototype of new object, 201
 - defined, 116
 - duck-typing and, 213
 - inheritance and, 115
 - jQuery.fn, 583
 - making nonextensible, 242
 - in multiple, interacting windows, 359
 - Object.getPrototypeOf() method, 817
 - Object.isPrototypeOf() method, 819
 - proper initialization, key to subclassing, 229
 - testing prototype chain of an object, 210
 - proxy objects, 359
 - proxy() function, 573
 - pseudo-class filters, 584
 - publicId property, DocumentType object, 899
 - push buttons, 401
 - push() method, Array object, 151, 731
 - adding elements to end of array, 145
 - pushStack() method, jQuery, 582
 - pushState() method, History object, 672, 936
 - history management with (example), 673–676
 - PUT method, 609
 - putImageData(),
 - CanvasRenderingContext2D, 661, 878
- ## Q
- quadraticCurveTo(),
 - CanvasRenderingContext2D, 643, 878
 - queryCommandEnabled(), Document object, 410, 897
 - queryCommandIndeterminate(), Document object, 897
 - queryCommandState(), Document object, 410, 897
 - queryCommandSupported(), Document object, 410, 897
 - queryCommandValue(), Document object, 897
 - querySelector() method, 370
 - Document object, 897
 - Element object, 908
 - querySelectorAll() method, 370
 - Document object, 440, 898
 - \$() function versus, 529
 - Element object, 908
 - selecting form elements, 398
 - queue() method, jQuery, 558
 - queues, jQuery animation, 552
 - canceling, delaying, and queuing effects, 557
 - queue property, animation options object, 555
 - quirks mode, 330, 425
 - HTML document display, 369
 - quotation marks (see under Symbols section)
- ## R
- radial gradients, 647, 875
 - radians, specifying for angles in Canvas API, 638
 - <radio> elements, 402
 - random() function, Math object, 798
 - Range object, 408
 - RangeError object, 828
 - rangeOverflow property, FormValidity object, 932
 - rangeUnderflow property, FormValidity object, 932
 - read operations, 925
 - (see also FileReader object)
 - events triggered on XMLHttpRequest or FileReader, 455
 - readAsArrayBuffer() method
 - FileReader object, 698, 699, 926
 - FileReaderSync object, 927
 - readAsBinaryString() method
 - FileReader object, 698, 926
 - FileReaderSync object, 927
 - readAsDataURL() method

- FileReader object, 698, 926
- FileReaderSync object, 927
- readAsText() method
 - FileReader object, 698, 926
 - FileReaderSync object, 927
- readOnly property, Input object, 944
- readyState property
 - Document object, 323, 465, 894
 - EventSource object, 921
 - FileReader object, 698, 925
 - MediaElement object, 619, 965, 968
 - WebSocket object, 999
 - XMLHttpRequest object, 498, 1013
 - values, 498
- readystatechange events, 465, 499
- Really Simple History (RSH) library, 346
- rect() method
 - CanvasRenderingContext2D, 878
- rect(), CanvasRenderingContext2D, 645
- rectangles
 - clearRect() method, 874
 - ClientRect objects, 880
 - drawing in canvas, 645, 868
 - fillRect() method, 876
- reduce() method, Array object, 155, 192, 732
- reduceRight() method, Array object, 155, 733
- by reference, object comparisons, 45
- reference types, 45
- ReferenceError objects, 56, 829
- references, 45
- references to previous subexpression of regular expression, 256
- referrer property, Document object, 405, 894
- RegExp class, 30
 - toString() method, 50
- RegExp object, 39, 251, 261, 829–835
 - as callable object, 191
 - exec() method, 262, 585, 831
 - instance properties, 830
 - lastIndex property, 833
 - properties, 262
 - source property, 833
 - test() method, 263, 834
 - toString() method, 834
- registerContentHandler(), Navigator object, 975
- registering event handlers, 320, 456–460
 - advanced event handler registration with jQuery, 544
 - deregistering event handlers with jQuery, 546
 - for HTTP progress events, 507
 - for load and click events, 11
 - setting event handler attributes, 457
 - setting event handler properties, 457
 - simple event handler registration with jQuery, 540
 - using addEventListener(), 458
- registerProtocolHandler(), Navigator object, 975
- regular expressions, 39, 251–263
 - defining, 251
 - alternation, grouping, and references, 256
 - character classes, 253
 - flags, 259
 - literal characters, 252
 - repetition, 254
 - specifying match position, 257
 - RegExp objects, 261–263
 - string methods using, 259–261
- relatedTarget property, Event object, 452, 543, 917
- relational databases, 705
- relational expressions, 71
 - comparison operators, 73
 - equality and inequality operators, 71
 - in operator, 74
 - instanceof operator, 75
- relational operators, object-to-primitive conversions with, 51
- relative positioning of elements, 420
- relative URLs, 344
- relative values for numeric properties'
 - animation, 554
- relList property, Link object, 963
- reload() method, Location object, 344, 965
- remove() method
 - DOMTokenList object, 438, 902
 - jQuery, 540
 - Select element, 987
- removeAttr() function, 531
- removeAttribute(), Element object, 377, 908
- removeAttributeNS(), Element object, 908
- removeChild() method, Node object, 384, 979
- removeClass() method, jQuery, 532
- removeData() method, jQuery, 536

- removeEventListener() method
 - Document object, 459
 - EventTarget object, 922
 - Worker object, 681
 - WorkerGlobalScope object, 683
- removeItem() method, Storage object, 988
- repeat property, Event object, 920
- repetition in regular expressions, 254
 - nongreedy, 255
- replace() method
 - Location object, 344, 965
 - String object, 260, 844
- replaceAll() method, jQuery, 538
- replaceChild() method, Node object, 384, 979
- replaceData() method
 - Comment node, 881
 - Text node, 993
- replaceState() method, History object, 673, 937
- replaceWholeText(), Text node, 994
- replaceWith() method, jQuery, 537
- requests and responses (HTTP), 495
 - aborting requests and timeouts, 510
 - decoding the response, 500
 - encoding request body, 502–507
 - getting onreadystatechange response, 499
 - order of request parts, 497
 - response components, 498
 - specifying the request, 495
 - synchronous responses, 499
- required property
 - Input object, 944
 - Select element, 986
- reserved words, 24
 - CSS properties having reserved word in name, 432
 - HTML attribute names, 376
 - primary expressions, 58
 - using as property names, 117
 - querying and accessing the properties, 120
- reset() method, Form object, 400, 929
- resize events, windows, 451
- resize() method, jQuery, 541
- response property, XMLHttpRequest object, 1014
- responseText, XMLHttpRequest object, 501, 1014
 - parsing response, 500
- responseType, XMLHttpRequest object, 1014
- responseXML property, XMLHttpRequest objects, 501
- responseXML, XMLHttpRequest object, 500, 1014
- restore(), CanvasRenderingContext2D, 636, 878
- restricted features in browsers, 333
- result property
 - Event object, 542, 544
 - FileReader object, 698, 925
- return keyword, use by constructor functions, 170
- return statements, 61, 105, 166
 - line break interpreted as semicolon, 26
 - use by generator functions, 277
- return values
 - event handler, 462
 - jQuery event handler functions, 542
 - setting returnValue property of event to false, 464
- returnValue property
 - BeforeUnloadEvent object, 863
 - Event object, 464, 917
 - Window object, 1003
- reverse() method, Array object, 149, 734
- revokeObjectURL(), method, URL object, 998
- revokeObjectURL(), URL object, 697
- RGBA color space, 427
- Rhino
 - JavaScript versions, 269
 - scripting Java with, 289–296
 - GUI (graphical user interface) example, 293–296
 - support for E4X, 283
 - support for JavaScript extensions, 265
- rich-text editing functionality, 410
- right and bottom style properties, 421, 425
- right property, ClientRect object, 880
- right, left, top, and bottom properties, 392
- rollovers, image, 614
- rotate(), CanvasRenderingContext2D, 638, 878
- rotation property, 456
- rotations, 641
- round() function, Math object, 798

- rounding errors, binary floating-point numbers and, 35
- rowIndex property, TableRow object, 992
- rows property
 - Table object, 990
 - TableSection object, 992
- rowSpan property, TableCell object, 991
- RSH (Really Simple History) library, 346
- rules, style, 414, 885
 - querying, inserting, and deleting in stylesheets, 441

S

- \s (space) character in regular expressions, 254
- Safari
 - current version, 327
 - gesture and touch events on iPhone and iPad, 456
 - JavaScript in URLs, 316
 - mouse, two-dimensional trackball, 471
 - textInput event, 453
- same-origin policy, 334
 - Canvas objects, toDataURL() method, 657
 - filesystems and, 700
 - origin of a document, 590
 - preventing cookie swapping across sites, 595
 - relaxing, 335
- sandbox attribute, <iframe> element, 337
- sandbox property, IFrame object, 939
- sandboxing systems, 267
- save(), CanvasRenderingContext2D, 636, 879
- Scalable Vector Graphics (see SVG)
- scale property, 456
- scale(), CanvasRenderingContext2D, 638, 879
- scope, variable, 31, 53–56
 - closures, 180–185
 - cookies, 594
 - event handler functions, 461
 - function scope and hoisting, 54, 165
 - functions as namespaces, 178
 - IE userData, 600
 - JavaScript functions and, 163
 - nested functions, 166
 - scope chain, 55
 - sessionStorage, 592
 - storage scope, 589, 590

- variables as properties, 55
- variables defined with let keyword, 270
- Worker threads, 682
- WorkerGlobalScope object, 1009
- scoped property, Style object, 989
- Screen object, 348, 984
- screen property, Window object, 348, 1003
- screen readers, 332
- screenX and screenY properties
 - Event object, 917
 - Window object, 1003
- <script> elements, 309
 - Ajax transport with, 492
 - async and defer attributes, 319, 324
 - embedding JavaScript in HTML, 9, 312
 - HTTP by, JSONP, 513–515
 - HTTP scripting with, 501
 - src attribute, 313
 - text in, 381
 - type attribute, 314
- Scriptaculous library, Prototype framework, 435
- scripts
 - in external files, 313
 - jQuery.getScript() function, 561
 - Script object, 985
 - synchronous, asynchronous, and deferred, 318
 - type attribute, specifying MIME type, 314
- scripts property, Document object, 367, 894
- scroll offsets, 467
- scroll properties, document elements, 395
- scroll() method
 - jQuery, 541
 - Window object, 394, 1006
- scrollbar positions of a window, 391
- scrollBy() method, Window object, 394, 1006
- scrollHeight and scrollWidth properties, Element object, 906
- scrolling, 394
 - scroll events in windows, 451
- scrollIntoView() method, Element object, 394, 908
- scrollLeft and scrollTop properties, 391
 - Element object, 906
- scrollLeft() method, jQuery, 536
- scrollTo() method, Window object, 394, 1006
- scrollTop() method, jQuery, 536

- seal() function, 137, 823
- sealed objects, 820
- seamless property, IFrame object, 940
- search property
 - Link object, 963
 - Location object, 343, 965
 - WorkerLocation object, 1012
- search() method, String object, 259, 845
- secure attribute, cookies, 595, 596
- security, 333–338
 - client-side storage and, 589
 - cookie data and, 593
 - cross-origin HTTP requests, 511
 - cross-site scripting (XSS), 336
 - denial-of-service attacks, 338
 - same-origin policy, 334
 - scripting plug-ins and ActiveX controls, 336
 - scripts and, 513
 - subsets for, 267
 - features removed, 267
 - listing of important subsets, 268
 - toDataURL() method, Canvas objects, 657
 - what JavaScript can't do, 333
- seekable property, MediaElement object, 619, 968
- seeking property, MediaElement object, 968
- Select element, 403, 986
- select() method
 - Input object, 945
 - jQuery, 541
 - TextArea object, 995
- selected property, Option object, 404, 981
- selected text, querying in a document, 407
- selectedIndex property
 - HTMLCollection object, 938
 - Select element, 404, 986
- selectedOption property, Input object, 944
- selectedOptions property, Select element, 987
- selecting document elements, 364–371
 - by CSS class, 368, 369
 - document.all[] collection, 371
 - by id, 364
 - by name, 365
 - by type, 366
- selection methods in jQuery, 578–582, 948
 - reverting to previous selection, 581
 - using selection as context, 580
- Selection object, 408
- selectionEnd property
 - Input object, 944
 - TextArea object, 994
- selectionRowIndex, TableRow object, 992
- selectionStart property
 - Input object, 944
 - TextArea object, 994
- selector property, jQuery objects, 529
- selectors
 - CSS, 369, 414
 - for style rules, 441
 - using to invoke jQuery() function, 526
 - jQuery, 574–578, 946
 - combinations of, 577
 - filters for, 575–577
 - groups, 578
 - selecting part of document to display with jQuery, 559
- Selectors API, 370
- selectorText property, CSSRule object, 886
- self property
 - Window object, 1003
 - WorkerGlobalScope object, 683, 1009
- self variable, using with nested functions, 169
- self-referential Window objects, 356
- send() method
 - generators, 280
 - WebSocket object, 1000
 - XMLHttpRequest object, 497, 1016
- serialize() method, 562
- serializing objects, 138, 677
 - example, JSON.stringify() function, 789
- Server-Sent Events, 515–521
 - custom chat server, 519
 - emulating EventSource with XMLHttpRequest, 517–519
 - simple chat client using EventSource, 516
- server-side JavaScript, 289–304
 - asynchronous I/O with Node, 296–304
 - scripting Java with Rhino, 289–296
- sessionStorage property, Window objects, 589, 1004
 - cookies versus, 594
 - storage API, 591
 - storage events, 592
 - storage lifetime and scope, 590
- set() method, TypedArray object, 689, 997
- setAttribute() method, Element object, 908

- setAttributeNS() method, Element object, 909
- setCapture() method (IE), 468
- setCustomValidity(), FormControl object, 931
- setData(), DataTransfer object, 475, 889
- setDate() method, Date object, 753
- setDragImage(), DataTransfer object, 475, 889
- setFloat32() method, DataView object, 890
- setFloat64() method, DataView object, 891
- setFullYear() method, Date object, 753
- setHours() method, Date object, 754
- setInt16() method, DataView object, 891
- setInt32() method, DataView object, 891
- setInt8() method, DataView object, 891
- setInterval() function, 297, 322
- setInterval() method
 - use in malicious code, 338
 - Window object, 342, 1007
 - WorkerGlobalScope object, 1010
- setItem() method, Storage object, 988
- setMilliseconds() method, Date object, 754
- setMinutes() method, Date object, 755
- setMonth() method, Date object, 755
- setRequestHeader(), XMLHttpRequest, 496
- setRequestHeader(), XMLHttpRequest object, 1016
- sets, Set class (example), 215–217
- setSeconds() method, Date object, 756
- setSelectionRange(), Input object, 945
- setSelectionRange(), Textarea object, 995
- setTime() method, Date object, 756
- setTimeout() function, 297, 322
 - implementing timeouts for
 - XMLHttpRequest, 510
- setTimeout() method
 - Window object, 308, 342, 1007
 - WorkerGlobalScope object, 1010
- setTransform(), CanvasRenderingContext2D, 638, 641, 879
- setUTCDate() method, Date object, 757
- setUTCFullYear() method, Date object, 757
- setUTCHours() method, Date object, 758
- setUTCMilliseconds() method, Date object, 758
- setUTCMinutes() method, Date object, 759
- setUTCMonth() method, Date object, 759
- setUTCSeconds() method, Date object, 760
- setVersion() method, IndexedDB objects, 707
- setYear() method, Date object, 760
- seUint16() method, DataView object, 891
- seUint18() method, DataView object, 891
- seUint32() method, DataView object, 891
- shadowBlur property, 653, 872
- shadowColor property, 653, 872
- shadowed text, CSS positioning example, 422
- shadowOffsetX and shadowOffsetY properties, 653
- shadowOffsetX and shadowOffsetY,
 - CanvasRenderingContext2D, 872
- shadows, drawing in canvas, 653, 869
- shaking an element side to side (animation example), 433–435
- shared workers, 684
- shear transforms, 641
- sheet property, Link object, 963
- sheet property, Style object, 989
- shift() method, Array object, 146, 152, 734
- shiftKey property, Event object, 451, 467, 484, 917
- shortcut properties in CSS, 416
 - corresponding properties in
 - CSSStyleDeclaration object, 432
- show() method, jQuery, 553
- showModalDialog(), Window object, 349, 1007
- sibling properties, Element object, 372
- side effects
 - case expressions containing, 96
 - expressions with, 87
 - operator, 64
- Sieve of Eratosthenes algorithm, 688
- sin() function, Math object, 799
- size of elements, setting with CSS properties, 421
- size property
 - Blob object, 863
 - Input object, 944
 - Select element, 986
- Sizzle library, 370
- slice() method
 - Array object, 150, 735
 - Blob object, 691, 695, 863
 - jQuery, 579
 - String object, 846
- slideDown(), slideUp(), and slideToggle()
 - methods, jQuery, 553

- `some()` method, Array object, 154, 736
- `sort()` method, Array object, 149, 737
 - functions as arguments, 177
- sorting objects for comparison, 222
- <source> elements, 615
- source property
 - MessageEvent object, 677, 972
 - RegExp object, 833
- sparklines, 377
 - drawing in canvas (example), 663–665
- sparse arrays, 141, 144
- speed property, Geocoordinates object, 933
- spell-checking in browsers, 409
- Spidermonkey
 - destructuring assignment, 272
 - JavaScript versions, 269
 - support for E4X, 283
 - support for JavaScript extensions, 265
- `splice()` method, Array object, 151, 737
- `split()` method
 - splitting cookie property into name/value pairs, 596
 - String object, 261, 847
- `splitText()` method, Text node, 994
- SQL databases, 705
- `sqrt()` function, Math object, 799
- `SQRT1_2` constant (Math), 799
- `SQRT2` constant (Math), 800
- src attribute, <script> element, 313
- src property
 - IFrame object, 940
 - Image object, 941
 - MediaElement object, 968
 - Script object, 985
- srcdoc property, IFrame object, 940
- srcElement property, Event object, 917
- standards mode, 330
 - HTML document display, 369
- `start()` method
 - MessagePort object, 972
 - TimeRanges object, 996
- `startOffsetTime`, MediaElement object, 968
- state property
 - History object, 673
 - PopStateEvent object, 673, 982
- state-change events, 449
- statement blocks, 88
- statements, 87–113
 - compound and empty, 88
 - conditional, 92
 - else if, 94
 - if, 92
 - switch, 95
 - control structure, in function body, 7
 - debugger, 110
 - declaration, 89
 - defined, 6
 - expression statements, 88
 - function declaration, 91, 164
 - jumps, 102
 - break statements, 103
 - continue statements, 104
 - labeled statements, 102
 - return statements, 105
 - throw statements, 106
 - try/catch/finally statements, 106
 - loops, 97
 - do/while loops, 98
 - for loops, 98
 - for/in loops, 100
 - while loops, 97
 - summary of, 112
 - terminating, optional semicolons and, 25
 - with statement, 108
- static positioning of elements, 420
- status codes, Ajax in jQuery, 560
- status property
 - ApplicationCache object, 606
 - XMLHttpRequest object, 1014
- statusText, XMLHttpRequest object, 1015
- step property, Input object, 944
- `stepDown()` method, Input object, 945
- stepMismatch property, FormValidity object, 932
- `stepUp()` method, Input object, 945
- `stop()` method, jQuery, 557
- `stopImmediatePropagation()`, Event object, 465, 919
- StopIteration exceptions, 275
 - thrown by generators' `next()` method, 278
- `stopPropagation()`, Event object, 464, 919
- storage, 587
 - (see also client-side storage)
- Storage object, 987
- storageArea property, StorageEvent object, 988
- StorageEvent object, 988
- streaming API for innerHTML property, 407

- streaming media, initialTime property, 618
- streams, in Node, 298
- strict equality operator (===), 71
- strict inequality operator (!==), 71
- strict mode, 111
 - reserved words, 24
- strictness
 - event handlers defined in non-strict mode, 458
- string concatenation operator (+), 67, 74
- string literals, 36
 - escape sequences in, 37
- String() constructor, 43
- String() function, type conversions with, 47
- stringify() function, 138, 788
- strings, 29, 36
 - as arrays, 160
 - comparisons, 44, 74
 - conversions, 45, 47
 - between JavaScript and Java, 293
 - object to string, 50
 - converting arrays to, 152
 - encoding application state as, 672
 - immutability of, 31
 - methods for pattern matching, 259–261
 - match() method, 260
 - replace() method, 260
 - search() method, 259
 - object conversion to, for Ajax, 559
 - parsing HTTP response, 500
 - pattern matching, 39
 - properties accessed with [] notation, 121
- String object, 835–852
 - charAt() method, 838
 - charCodeAt() method, 838
 - concat() method, 839
 - fromCharCode() method, 481, 839
 - HTML methods, 836
 - indexOf() method, 840
 - lastIndexOf() method, 841
 - length property, 842
 - localeCompare() method, 74
 - localeCompare() method, 842
 - match() method, 843
 - methods, listed, 835
 - replace() method, 844
 - search() method, 845
 - slice() method, 846
 - split() method, 847
 - static method, fromCharCode(), 836
 - substr() method (deprecated), 849
 - substring() method, 849
 - toLocaleLowerCase() method, 850
 - toLocaleUpperCase() method, 851
 - toLowerCase() method, 74, 851
 - toString() method, 851
 - toUpperCase() method, 74, 852
 - trim() method, 852
 - valueOf() method, 852
- Unicode characters, codepoints, and, 36
- values in stylesheet or style attribute, 432
- working with, 38
- wrapper objects, 43
- stroke
 - colors, gradients, and patterns in Canvas, 645–648, 868
 - defined, 879
 - pattern and gradient stroke, 647
 - unclipped, 652
- stroke(), CanvasRenderingContext2D, 633, 648, 879
- strokeRect(), CanvasRenderingContext2D, 645, 879
- strokeStyle property,
 - CanvasRenderingContext2D, 635, 872
- strokeText(), CanvasRenderingContext2D, 650, 879
- strongly typed languages
 - classes in, 199
 - objects in, 120
- structure and traversal (Document objects), 371–375
 - documents as trees of elements, 372–375
 - documents as trees of nodes, 371
- structured clones, 672
- <style> elements
 - Element objects representing, 440
 - enclosing CSS stylesheet in, 415
- style attribute, 10
- Style object, 989
- style properties, 414
 - combining using shortcut properties, 416
 - important, 419
 - units for settings, 432
- style property, 308
 - CSSRule object, 886
 - Element object, 308, 431, 906

- style rules, 414
 - styles, cascade of, 415
 - stylesheets, 440–443
 - associating with HTML documents, 415
 - creating new, 442
 - CSSStyleSheet object, 887
 - defined, 414
 - enabling and disabling, 440
 - querying, inserting, and deleting rules, 441
 - styleSheets property, Document object, 440, 441, 894
 - sub-statements, 89
 - subarray(), TypedArray object, 689, 998
 - subclasses, 228
 - class hierarchies and abstract classes, 234–238
 - composition versus subclassing, 233
 - constructor and method chaining, 231–233
 - creating using ECMAScript 5 features, 243
 - defining, 229
 - subdomains, problems posed by same-origin policy, 335
 - submit() method
 - Form object, 400, 929
 - jQuery, 541
 - triggering events, 547
 - subpaths (Canvas), 632
 - subsets of JavaScript, 265–269
 - The Good Parts, 266
 - for security, 267
 - listing of important subsets, 268
 - substr() method (deprecated), String object, 849
 - substring() method, String object, 849
 - substringData() method
 - Comment node, 881
 - Text node, 994
 - support property, 574
 - SVG (Scalable Vector Graphics), 622–630
 - <canvas> element versus, 630
 - displaying time by manipulating image, 628
 - pie chart built with JavaScript, 624–627
 - <svg:path> elements, 627
 - swapCache(), ApplicationCache object, 607, 860
 - switch statements, 95
 - case clauses, 96
 - synchronous event triggering in jQuery, 547
 - synchronous execution of scripts, 319
 - synchronous HTTP responses, 499
 - SyntaxError object, 852
 - raised in strict mode when deleting properties, 125
 - systemId property, DocumentType object, 899
- ## T
- Table object, 990
 - table of contents, generating for a document (example), 387–390
 - TableCell object, 991
 - TableRow object, 991
 - TableSection object, 992
 - tabs in browser windows, 353
 - tag names (XML), 285
 - tag names, getting elements by, 366
 - tagName property
 - Element object, 906
 - tan() function, Math object, 800
 - target property
 - Event object, 918
 - in jQuery, 543
 - events, 446
 - Form object, 399, 929
 - ProcessingInstruction object, 982
 - tBodies property, Table object, 990
 - temporary filesystem, 700
 - terminate() method, Worker object, 682, 1009
 - test expression (for loops), 98
 - test() method, RegExp object, 263, 834
 - text, 36–40
 - CharacterData methods for manipulating, 382
 - conversion to speech in screen readers, 332
 - drawing in canvas, 650, 869, 876
 - element content as plain text, 380
 - embedding arbitrary textual data using script element, 314
 - escape sequences in string literals, 37
 - pattern matching with regular expressions, 39
 - querying selected text in documents, 407
 - reading text files with FileReader, 698
 - in <script> elements, 381
 - string literals, 36
 - text fields in forms, 402

- working with strings, 38
- Text and Textarea elements, forms, 400
- text input events, 481–484
 - filtering user input (example), 481–483
 - support for, DOM Events specification, 453
 - using propertychange event to detect, 484
- Text nodes, 371, 993
 - creating, 382
 - element content as, 381
- text property
 - HTMLElement object, 314
 - Link object, 963
 - Option object, 404, 981
 - Script object, 985
- time() method, jQuery, 534
- text-input form elements
 - browsers triggering input event on, 450
- text-shadow property, 422
- textAlign property,
 - CanvasRenderingContext2D, 651, 872
- <textarea> elements, 403
- TextArea object, 994
- textBaseline property,
 - CanvasRenderingContext2D, 651, 873
- textContent property, Node object, 380, 977
- textLength property
 - TextArea object, 995
- TextMetrics object, 651, 877, 995
- TextRange object (IE), 408
- tFoot property, Table object, 990
- tHead property, Table object, 990
- third dimension, z-index property, 422
- this keyword
 - in event handlers, 401
 - function invocation context, 163
 - functions used as methods, 168
 - in method invocation, 168
 - nested function invoked as method or function, 169
 - as primary expression, 58
 - referring to global object, 42, 783
 - referring to target of event handlers, 461
 - removal or restriction in secure subsets, 267
 - use in property getters and setters, 130
- threading
 - in client-side JavaScript, 322
 - FileReaders and, 698
 - IndexedDB operations and, 707
 - Web Workers specification, 680–687
 - debugging Worker threads, 686
 - Worker execution model, 683
 - Worker threads, 1008
 - WorkerGlobalScope object, 1009
- 3D graphics for <canvas> element, 631, 688
- throw statements, 106
- throw() method, generators, 280
- time, 752
 - (see also dates and time)
 - Date.getTime() function, 766
 - UTC and GMT, 742
- time() method, Console object, 883
- timeEnd() method, Console object, 883
- timeline, client-side JavaScript program execution, 323
- timeout option, Geolocation methods, 934
- timeout property, XMLHttpRequest object, 510, 1015
- TimeRanges object, 619, 995
- timers, 342–343
 - client-side timer functions implemented by Node, 297
 - methods available to WorkerGlobalScope object, 683
 - similarity to events, 449
 - using in inline scripting of CSS animation, 433–435
 - utility function for (example), 342
- timeStamp property, Event object, 543, 918
- timestamp property, Geoposition object, 935
- title property
 - CSSStyleSheet object, 440, 887
 - Document object, 405, 894
 - Element object, 906
 - Link object, 963
 - Style object, 989
- toArray() method, jQuery, 528
- toDataURL(), Canvas object, 656, 865
- toDateString() method, Date object, 760
- toElement property, Event object, 918
- toExponential(), Number object, 48, 804
- toFixed(), Number object, 48, 805
- toggle buttons, 402
- toggle() method
 - DOMTokenList object, 902

- jQuery, 541, 553
- toggleClass() method, jQuery, 532
- toGMTString() method, Date object, 761
- toISOString() method, Date object, 761
- toJSON() method, 139
 - Date object, 138, 762
 - implementation in classes, 220
- token lists, DOM, 901
- toLocaleDateString(), Date object, 762
- toLocaleLowerCase(), String object, 850
- toLocaleString() method, 139
 - Array object, 152, 738
 - Date object, 763
 - implementation in classes, 220
 - Number object, 806
 - Object class, 823
- toLocaleTimeString(), Date object, 763
- toLocaleUpperCase(), String object, 851
- toLowerCase() method, String object, 851
- tooLong property, FormValidity object, 932
- top and left style properties, 421, 425
- top property
 - ClientRect object, 880
 - Window object, 356, 1004
- top, bottom, right, and left properties, 392
- top, left, width, and height style properties, 424
- top-level ancestor window, 354
- top-level windows
 - null frameElement property, 357
 - parent property referring to self, 356
- toPrecision(), Number object, 48, 807
- toStaticHTML() method (IE), 337
- toString() method, 139
 - Array object, 152, 739
 - Boolean object, 741
 - converting boolean values to strings, 41
 - Date object, 763
 - Error object, 770, 771
 - Function object, 780
 - functions, 189
 - implementation in classes, 220
 - Location object, 343
 - Number object, 48, 807
 - Object class, 824
 - querying class attribute, 136
 - RegExp object, 834
 - Selection object, 408
 - String object, 851
 - type conversions with, 49
- total property, ProgressEvent, 508, 984
- toTimeString(), Date object, 764
- touchscreens, events, 456
- toUpperCase() method, String object, 852
- toUTCString() method, Date object, 764
- trace() method, Console object, 883
- transaction management in IndexedDB, 707
- transform() method,
 - CanvasRenderingContext2D objects, 641, 880
- transformations
 - canvas coordinate system, 638–643, 869
 - transformation example, 641
 - understanding mathematically, 640
 - coordinate system in canvas
 - shadows and, 655
 - setTransform() method in canvas, 879
- Transforms (CSS), 419
- Transitions module (CSS), 419, 435
- translate() method,
 - CanvasRenderingContext2D, 638, 880
- translucency, specifying with opacity style property, 428
- translucent windows, overlapping with CSS (example), 429–431
- transparency
 - compositing operations with hard and soft transparency, 657
 - specifying alpha values in canvas, 646
 - specifying for colors in CSS, 427
- transports, 492
- traversal of documents element-by-element, portable functions for, 373
- tree structure
 - documents as trees of elements, 372–375
 - documents as trees of nodes, 371
 - geometrical, coordinate-based view of document versus, 390
 - representation of HTML documents, 362
- trigger() function, 549
- trigger() method, jQuery, 542, 547
- triggerHandler() method, jQuery, 548
- triggering events, 547
 - preventing jQuery from triggering Ajax-related events, 571
- trim() function, 574
- trim() method, String object, 852

- true and false values, 45
- try/catch/finally statements, 106
 - multiple catch clauses in, 283
- Tufte, Edward, 663
- Twitter search gadget, controlled by
 - postMessage(), 678–680
- type (tag name), selecting HTML or XML
 - elements by, 366
- type attribute, <script elements>, 314
- type property
 - Blob object, 863
 - CSSRule object, 886
 - CSSStyleSheet object, 887
 - Event object, 918
 - events, 446
 - form elements, 397, 400
 - FormControl object, 930
 - Script object, 985
 - Select element, 403
 - Style object, 989
- TypedArray object, 996
- TypeError object, 42, 853
 - property access errors, 123
 - raised by attempts to create or modify
 - properties, 133
 - resulting from attempts to delete properties, 124
 - thrown in type conversions, 47
- typeMismatch property, FormValidity object, 932
- typeof operator, 62, 82, 210
 - applied to null and undefined values, 41
 - using with XML objects, 284
- types (see data types)
- types property, DataTransfer object, 477, 889
- undefined values, 29, 41, 854
 - properties set to, testing, 125
 - properties with, access errors, 123
 - property access expressions and, 60
 - returned by functions, 166
 - variables declared without initializer, 90
- undelegate() method, jQuery, 549
 - deregistering event handlers for live events, 550
- underflow, 33
- unescape() function (deprecated), 855
- Unicode, 21
 - characters, codepoints, and JavaScript
 - strings, 36
 - codepoints, 481
 - escape sequences for, 22
 - format control characters, 22
 - in identifiers, 24
 - normalization of character encodings, 23
- unit specification for CSS style properties, 429, 432
- unlabeled continue statements, 104
- unload events, 450
 - BeforeUnloadEvent object, 863
- unmonitorEvents(), ConsoleCommandLine
 - object, 885
- unobtrusive JavaScript, 311
- unshift() method, Array object, 146, 152, 739
- untyped variables, 31
- unwrap() method, jQuery, 540
- update() method, ApplicationCache object, 607, 860
- upload progress events, 508
- upload property, XMLHttpRequest object, 508, 1015
- URIError object, 855
- URIs
 - decodeURI() function, 766
 - decodeURIComponent() function, 767
 - encodeURI() function, 767
 - encodeURIComponent() function, 768
- URL objects, 998
- URL property
 - Document object, 343, 405, 894
 - Window object, 1004
- url property
 - EventSource object, 921
 - StorageEvent object, 989
 - WebSocket object, 1000

U

- UI (user interface)
 - events, 448
 - Java, implementing using JavaScript in
 - Rhino, 292
 - GUI example, 293–296
 - jQuery UI library, 585
- Uint16Array class, 996
- Uint32Array class, 996
- Uint8Array class, 996
- Uint8Array objects, 688
- unary arithmetic operators, 68
- unbind() method, jQuery, 546

URLs

- blank-page URL, *about:blank*, 354
 - Blob, 695–697
 - displayed as current state of web application, 672
 - `importScripts()` function arguments, 682
 - JavaScript, 315
 - loading document and displaying portions of it with jQuery, 559
 - parsing, 343
 - relative, 344
 - subject of HTTP request, 496
 - URL object, 998
 - WebSocket, 713
 - wildcard, in network section of application cache manifest, 603
- use strict directive, 110
- user input
- filtering, 481–483
 - text events, 481
- user interface (see UI)
- user-defined objects, 116
- userAgent property
- Navigator object, 347, 974
 - WorkerNavigator object, 1012
- userData API (IE), 588
- userData property
- persistence, 599–601
- UTC (Coordinated Universal Time), 742
- UTC() method, Date object, 765

V

- val() method, jQuery, 533
- valid property, FormValidity object, 932
- validation of forms
- FormValidity objects, 932
 - mechanism in HTML5, 455
- validationMessage, FormControl object, 930
- validity property, FormControl object, 930
- by value
- comparisons of primitives, 44
- value attribute (properties), 131
- value attribute, cookies, 595, 596
- value property
- DOMSettableTokenList object, 901
 - form elements, 400
 - FormControl object, 931
 - Meter object, 973
 - Option object, 981
 - Progress object, 983
 - valueAsDate property, Input object, 944
 - valueAsNumber property, Input object, 944
 - valueMissing property, FormValidity object, 932
- valueOf() method, 140
- Boolean object, 741
 - Date object, 766
 - implementation in classes, 220
 - Number object, 808
 - Object class, 825
 - object comparisons, 223
 - String object, 852
 - type conversions with, 50
- values, 4
- functions as, 176–178
 - defining your own function properties, 178
- values() method, ConsoleCommandLine, 885
- var keyword, 52
- replacing with let, 270
- var statements, 90
- varargs functions, 173
- variable-length argument lists, 172
- variables, 4
- in array comprehensions, 281
 - creating with function keyword, 358
 - data types and, 31
 - declaring, 52
 - repeated and omitted declarations, 52
 - declaring with let, 270
 - destructuring assignment, 272
 - functions assigned to, 176
 - global, 25
 - scope, 53–56
 - function scope and hoisting, 54
 - scope chain, 55
 - variables as properties, 55
 - variable reference as primary expression, 58
- versions, 269
- <video> elements, 615
- controls attribute, 616
 - events, 454
- Video object, 998
- videoHeight and videoWidth properties, Video object, 999
- view property, Event object, 918
- viewport

- defined, 390
- determining size of, 391
- viewport coordinates, 390
 - element positions in, 392
- visibility
 - partial visibility with overflow and clip style properties, 428
 - visibility property, 419, 426
- visual effects
 - image rollovers, 614
 - jQuery methods for, 551, 956
 - jQuery methods for simple effects, 552
 - motion blur of canvas graphics with
 - ImageData, 661
- visual effects libraries, 435
- void operator, 62, 85
 - forcing invocation or assignment
 - expressions to be undefined, 316
- volume
 - setting for media playback, 617
 - volume property, MediaElement, 968

W

W3C

- XMLHttpRequest Level 2 standard (XHR2), 494

- WAI-ARIA (Web Accessibility Initiative–Accessible Rich Internet Applications), 332

- warn() method, Console object, 883
- wasClean property, CloseEvent, 881
- watchPosition() method, Geolocation object, 934

- watchPosition(), Geolocation object, 669
- Web Accessibility Initiative–Accessible Rich Internet Applications (WAI-ARIA), 332

- web applications, 12
 - JavaScript in, 310
 - loan calculator (example), 12–18
 - offline, 608–612

- web browsers
 - asynchronous script execution, 324
 - <canvas> element support, 630
 - children property of Element objects, support of, 372
 - client-side databases integrated into, 588
 - compatibility and interoperability issues, 325

- compositing operations in, 659
- console tools, 3
- CORS (“Cross-Origin Resource Sharing”)
 - headers support, 511
- CSS box model, 425
- CSS box-sizing property, 426
- CSS color specifications support, 427
- CSS selector support, 370
- CSS support, 416
- current versions, 327
- database support, 705
- dataset property, not implemented, 378
- DOMContentLoaded event, 465
- editing document content, 409
- element-by-element document traversal support, 373
- event handlers directed at browser as a whole, 457
- event listener registration methods, 321
- EventSource support, 517
- HTML5 form features and events, 455
- HTTP progress events implementation, 507
- implementation of CSS Transitions, 435
- information about browsers and their screens, 346–348
- input event type fired after text insertion into element, 483
- insertAdjacentHTML() method support, 380
- JavaScript in, 307–339
 - accessibility, 332
 - client-side frameworks, 338
 - compatibility and interoperability, 325–332
 - embedding JavaScript in HTML, 311–317
 - execution of JavaScript programs, 317–324
 - security, 333–338
 - URLs, 315
 - web applications, 310
 - web documents, 310
- JavaScript tools, 3
- javascript: URLs, 316
- jQuery.browser property, 571
- keyIdentifier property in Chrome and Safari, 485
- location and navigation, 343–345

- mousewheel event support, 471
- Navigator object, 346
- opacity specifications, 428
- prefixes for nonstandard CSS properties, 416
- RegExp literals and object creation, 252
- Safari on iPhone and iPad, 456
- scope of localStorage, 590
- scripting a modal dialog (example), 9
- string indexing, 39
- support for E4X, 274
- support for object literal get and set syntax, 134
- support of `getElementsByClassName()`, 369
- SVG support, 622
- timeouts for HTTP requests, XHR2, 510
- transformed into simple operating systems, 310
- triggering input event on text-input form elements, 450
- use of callable objects, 191
- using JavaScript, 9
- web applications stored on, 601
- websites for compatibility information, 326
- XMLHttpRequest `readyState` values, 498
- web bugs, 492
- web databases, 588
- web documents
 - JavaScript in, 310
- web fonts, 418
- web pages, 12
 - scripting content, presentation, and behavior, 9
- web sockets, 713–716
 - chat server using WebSockets and Node, 715
 - creating socket and registering event handlers, 713
 - creating WebSocket-based chat client, 714
- Web Storage API, 455, 587, 705
- web workers
 - Worker object, 1008
 - WorkerGlobalScope object, 1009
 - WorkerLocation object, 1011
 - WorkerNavigation object, 1012
- Web Workers specification, 680–687
 - access to Blob URLs, 695
 - advanced worker features, 684
 - debugging workers, 686
 - making synchronous XMLHttpRequests in (example), 686
 - using synchronous filesystem API with worker threads, 704
 - worker for image processing (example), 684
 - Worker objects, 681
 - Worker scope, 682
- WebGL, 631
- `webkitURL.revokeObjectURL()` method, 697
- WebSocket object, 999
- wheel events (see mousewheel events)
- wheelDelta properties, 471
- wheelDelta property, Event object, 918
- wheelDeltaX and wheelDeltaY properties, Event object, 918
- which property, Event object, 451, 544, 918
- while loops, 97
 - continue statements in, 104
- whitespace
 - in JavaScript code, 22
 - matching with regular expressions, 254
- wholeText property, Text node, 993
- width and height properties
 - Canvas context objects, 637
 - Canvas object, 865
 - Image object, 941
 - ImageData object, 942
 - querying for element in jQuery, 535
 - Screen object, 985
 - Video object, 998
- width and height style properties, 421, 425
- width property
 - ClientRect object, 881
 - IFrame object, 940
 - TextMetrics object, 995
- `width()` method, jQuery, 535
- wildcard URL in application cache manifest, 603
- `willValidate`, FormControl object, 931
- window coordinates, 390
- Window object, 42, 1001–1008
 - `alert()`, `confirm()`, and `prompt()` methods, 348
 - `applicationCache` property, 604
 - beforeprint and afterprint events, 455
 - `close()` method, 356
 - closed property, 356

- constructors, 1004
- dialogArguments property, 349
- document property, 308, 361
- entry point to client-side JavaScript, 307
- event handlers, 1007
- event handlers directed at browser as a whole, 457
- event property, 460
- features available to WorkerGlobalScope objects, 683
- frameElement property, 357
- frames property, 357
- getComputedStyle() method, 436
- getSelection() method, 408
- length property, 357
- load event, 465
- localStorage and sessionStorage properties, 589
- location property, 343
- methods, 1004
- name property, 354
- navigator property, 346
- offline and online events, 455
- onbeforeunload event handler, 462
- onerror property, 351, 450
 - setting to a function, 321
- onhashchange property, 672
- onload event handler, 309
- open() method, 354
 - javascript: URL as argument, 315
- opener property, 356
- orientation property, 456
- pageXOffset and pageYOffset properties, 391
- parent property, 356
- postMessage() method, 336, 677
 - Twitter search gadget controlled by, 678–680
- properties, 1001
- screen property, 348, 984
- scroll() method, 394
- scrollTo() and scroll() methods, 394
- setInterval() method, 342
- setTimeout() method, 342
- setting onload property to event handler function, 457
- storage event, 455
- top property, 356
- URL property, 998
- window property, Window object, 308, 1004
- WindowProxy object, 359
- windows, 341–360
 - browser and screen information, 346–348
 - browser location and navigation, 343–345
 - loading new documents, 344
 - parsing URLs, 343
 - browser, JavaScript opening and closing, 333
 - browser, same-origin policy and, 334
 - browsing history, 345
 - dialog boxes, 348–351
 - document elements as properties of, 351
 - error handling, 351
 - events, 450
 - multiple windows and frames, 353–360
 - JavaScript in interacting windows, 358
 - opening and closing windows, 354–356
 - relationship between frames, 356
 - overlapping translucent windows (CSS example), 429–431
 - querying scrollbar positions, 390
 - querying viewport size, 391
 - timers, 342–343
- with statements, 108
 - removal in secure subsets, 267
- withCredentials, XMLHttpRequest, 511, 1015
- Worker object, 681, 1008
- Worker() constructor, 684
- WorkerGlobalScope object, 682, 1009
 - properties, 683
- WorkerLocation object, 1011
- WorkerNavigator object, 1012
- wrap(), wrapAll(), and wrapInner() methods, jQuery, 539
- wrapper objects, 43
- writable attribute (properties), 116, 131
 - extensible attribute used with, 137
- write() method, Document object, 318, 324, 406, 898
- writeFile() function, 299
- writeFileSync() function, 299
- writeln() method, Document object, 407, 898
- ws:// or wss:// protocol, 713

X

- XHR2, 494
 - downloading URL contents as Blob, 694
 - FormData API, 506

- handling binary responses, 501
 - overrideMimeType() method, 501
 - progress events, 1017
 - timeout property, 510
 - XHTML
 - case sensitivity, 21
 - <script> elements, 312
 - SVG embedded in document, 623
 - XML
 - E4X (ECMAScript for XML)
 - introduction to, 283–287
 - ECMAScript for XML (E4X), 274
 - HTTP request encoded with XML
 - document as body, 504
 - innerHTML property, use with XML elements, 379
 - namespaces, createElementNS() method, 382
 - optional in scripted HTTP, 493
 - outerHTML property, use of, 379
 - parsing HTTP response document with responseXML property, 500
 - processing instruction in a document, 982
 - querying and setting elements of documents, 376
 - SVG (Scalable Vector Graphics), 622
 - Text nodes in documents, 381
 - XML objects, 284
 - XMLHttpRequest API, 493
 - core portions and Level 2 draft (XHR2), 494
 - XMLHttpRequest object, 311, 1012
 - aborting requests and setting timeouts, 510
 - constants defining values of readyState property, 1013
 - cross-origin HTTP requests, 511–513
 - emulating EventSource with, 517–519
 - encoding request body, 502–507
 - event handlers, 1017
 - HTTP progress events, 507–510
 - HTTP requests and responses, 495
 - instantiating, 494
 - local files and, 495
 - making synchronous requests in Web Worker, 686
 - methods, 1015
 - POSTing plain text to a server, 497
 - posting user's chat messages to server, 516
 - properties, 1014
 - responseText property, 500
 - retrieving the response, 498–502
 - simulation by jqXHR object in jQuery 1.5, 565
 - specifying the request, 495
 - use by Ajax utility functions in jQuery, 561
 - Version 2 of specification, 455
 - XMLHttpRequest() constructor, 684
 - XMLHttpRequestUpload object, 1018
 - XMLList objects, 284
 - XSS (cross-site scripting), 336
- ## Y
- yield expressions, 280
 - yield keyword, 277
 - yieldForStorageUpdates(), Navigator object, 975
 - YUI (in-house library of Yahoo!), 339
- ## Z
- z-index property, 422
 - zero-based indexing (strings and arrays), 36, 141

About the Author

David Flanagan is a programmer and writer with a website at <http://davidflanagan.com>. His other O'Reilly books include *JavaScript Pocket Reference*, *The Ruby Programming Language*, and *Java in a Nutshell*. David has a degree in computer science and engineering from the Massachusetts Institute of Technology. He lives with his wife and children in the Pacific Northwest between the cities of Seattle, Washington, and Vancouver, British Columbia.

Colophon

The animal on the cover of *JavaScript: The Definitive Guide*, sixth edition, is a Javan rhinoceros. All five species of rhinoceros are distinguished by their large size, thick armor-like skin, three-toed feet, and single or double snout horn. The Javan rhinoceros, along with the Sumatran rhinoceros, is one of two forest-dwelling species. The Javan rhinoceros is similar in appearance to the Indian rhinoceros, but smaller and with certain distinguishing characteristics (primarily skin texture).

Rhinoceroses are often depicted standing up to their snouts in water or mud. In fact, they can frequently be found just like that. When not resting in a river, rhinos will dig deep pits in which to wallow. Both of these resting places provide a couple of advantages. First, they give the animal relief from the tropical heat and protection from blood-sucking flies. (The mud that the wallow leaves on the skin of the rhinoceros also provides some protection from flies.) Second, mud wallows and river water help support the considerable weight of these huge animals, thereby relieving the strain on their legs and backs.

Folklore has long held that the horn of the rhinoceros possesses magical and aphrodisiac powers, and that humans who gain possession of the horns will also gain those powers. This is one of the reasons why rhinos are a prime target of poachers. All species of rhinoceros are in danger, and the Javan rhino population is the most precarious. Fewer than 100 of these animals are still living. At one time, Javan rhinos could be found throughout southeastern Asia, but they are now believed to exist only in Indonesia and Vietnam.

The cover image is a 19th-century engraving from the Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed.

Get even more for your money.

Join the O'Reilly Community, and register the O'Reilly books you own. It's free, and you'll get:

- \$4.99 ebook upgrade offer
- 40% upgrade offer on O'Reilly print books
- Membership discounts on books and events
- Free lifetime updates to ebooks and videos
- Multiple ebook formats, DRM FREE
- Participation in the O'Reilly community
- Newsletters
- Account management
- 100% Satisfaction Guarantee

Signing up is easy:

- 1. Go to: oreilly.com/go/register**
- 2. Create an O'Reilly login.**
- 3. Provide your address.**
- 4. Register your books.**

Note: English-language books only

To order books online:

oreilly.com/store

For questions about products or an order:

orders@oreilly.com

To sign up to get topic-specific email announcements and/or news about upcoming books, conferences, special offers, and new technologies:

elists@oreilly.com

For technical questions about book content:

booktech@oreilly.com

To submit new book proposals to our editors:

proposals@oreilly.com

O'Reilly books are available in multiple DRM-free ebook formats. For more information:

oreilly.com/ebooks

O'REILLY®

Spreading the knowledge of innovators

oreilly.com

©2010 O'Reilly Media, Inc. O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 00000

Have it your way.

O'Reilly eBooks



- Lifetime access to the book when you buy through oreilly.com
- Provided in up to four DRM-free file formats, for use on the devices of your choice: PDF, .epub, Kindle-compatible .mobi, and Android .apk
- Fully searchable, with copy-and-paste and print functionality
- Alerts when files are updated with corrections and additions

oreilly.com/ebooks/

Safari Books Online

- Access the contents and quickly search over 7000 books on technology, business, and certification guides
- Learn from expert video tutorials, and explore thousands of hours of video on technology and design topics
- Download whole books or chapters in PDF format, at no extra cost, to print or read on the go
- Get early access to books as they're being written
- Interact directly with authors of upcoming books
- Save up to 35% on O'Reilly print books



See the complete Safari Library at safari.oreilly.com

O'REILLY®

Spreading the knowledge of innovators.

oreilly.com

©2011 O'Reilly Media, Inc. O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 000

CPSIA information can be obtained at www.ICGtesting.com
Printed in the USA
BVOW061613200613

323876BV00001B/1/P



JavaScript: The Definitive Guide

This book is a programmer's guide and comprehensive reference to the core JavaScript language and to the client-side JavaScript APIs defined by web browsers.

The sixth edition covers HTML5 and ECMAScript 5. Many chapters have been completely rewritten to bring them in line with today's best web development practices. New chapters in this edition document jQuery and server-side JavaScript.

Recommended for experienced programmers who want to learn the programming language of the Web, and for current JavaScript programmers who want to master it.

David Flanagan is also the author of *The Ruby Programming Language*, *Java in a Nutshell*, and *JavaScript Pocket Reference*.

Previous programming experience is recommended.

The Essential JavaScript Reference

Since 1996, *JavaScript: The Definitive Guide* has been the bible for JavaScript programmers. With more than 500,000 copies in print, web developers are still raving about it:

"A must-have reference for expert JavaScript programmers...well-organized and detailed."

—Brendan Eich
creator of JavaScript, CTO of Mozilla

"I made a career of what I learned from JavaScript: The Definitive Guide."

—Andrew Hedges
Tapulous

"The Definitive Guide taught me JavaScript."

—Tom Robinson
cofounder of 280 North, cocreator of Cappuccino

"I know which parts of JavaScript matter, based on how crinkled the spine of my copy of The Definitive Guide is in that section."

—J. Chris Anderson
cofounder of CouchBase, Apache CouchDB committer,
and author of CouchDB: The Definitive Guide

"...an indispensable reference for all JavaScript developers. If there's something I need to know about JavaScript, I trust The Definitive Guide will have the right answer for me. It's that good."

—Rey Bango
Microsoft Client-Web Community Program Manager and jQuery Team member



X003T1AKWB

New - JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)



Twitter: @oreillymedia
facebook.com/oreilly

O'REILLY®
oreilly.com