

# The Browser Object Model

PROGRAMMING WITH JAVASCRIPT



## Introduction

- What is the BOM?
  - Cross-browser issues
  - The 'core' BOM
- The window object
  - Global scope variables
- Other BOM objects
  - The location object
  - The history object
  - Document methods
  - Cookies
  - The screen object
  - The viewport
  - Window objects

## The Browser Object Model (BOM)

- The Browser Object Model (BOM) is the third part of JavaScript
  - With the core and the DOM making up the other two parts
- It allows interaction with the browser via JavaScript
  - Such as reading and writing to cookies
  - Storing data
  - Working with the browser history
- Many features actually disabled in modern browsers
  - Security and privacy issues lockdown functionality
  - As with status bar manipulation and popup windows

3

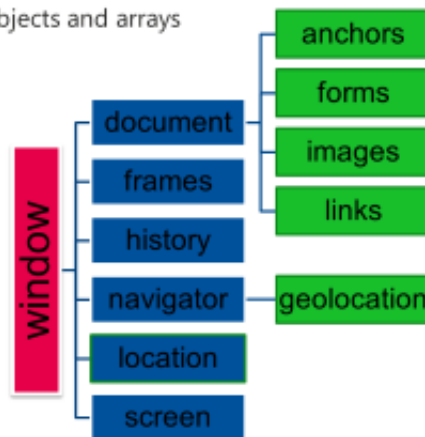
JavaScript consists of three parts. Two of these parts (the core component and Document Object Model) both have defined standards that all browsers are supposed to follow. The third part is the Browser Object Model and there is no official standard for this part of JavaScript. Fortunately for us, even though there is no actual standard for this part of JavaScript, most browsers have implemented the same commands to do the same (or almost the same) things; for those few areas where a browser does implement something in a different way, we can use feature sensing to test which is supported by the current browser.

The Browser Object Model (or BOM for short) is that part of JavaScript that allows JavaScript to interface and interact with the browser itself. As JavaScript can also run in environments other than web browsers, these are also the parts of JavaScript that may be unsupported or completely different when JavaScript is run somewhere other than a web browser. For the most part though, at least in web browsers, it is to the benefit of the browser writers themselves to implement the same processes in the same way as other web browsers, so that existing code written for other browsers will also work in theirs.

A lot of the BOM calls date back to the earliest days of JavaScript and have been available for JavaScript to use since it was first introduced in Netscape 2. If anything, there are fewer BOM commands that are usable now than there were back at the start, since there have been a number of them that have been disabled as browsers have introduced measures to protect people's security, since certain of the commands made scamming people visiting a web site much easier or have been overused to really annoy people.

## Browser object hierarchy

- Most BOM's expose the following objects and arrays
  - Arrays marked with an asterisk



4

Another thing we need to consider regarding the commands that make up the BOM portion of JavaScript is that a lot of them rely on information that is available to the browser. In some instances, the information that the browser has may be wrong (e.g. browsers do not know about operating systems introduced after the browser was released), or may be able to be changed by the person who owns the browser.

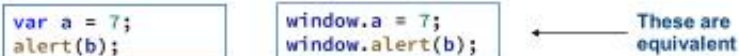
One final point to consider regarding the BOM commands is that you need to consider exactly what the particular value you are accessing means and its relevance to what you are trying to do. These commands are amongst the most misused commands in the JavaScript language (for example, testing the screen width and then using that to determine how wide to make something in the browser window where the browser window size is completely independent of the screen width).

The usefulness of the BOM commands varies from commands that are extremely useful to those that serve no useful purpose whatever.

The figure above shows the most common parts of the BOM that, with luck, will exist in most browsers you encounter. Divided between objects and arrays, we will examine some of the most useful functionalities in the next few slides.

## The global object

- Global object for client-side JavaScript is called window



```
var a = 7;
alert(b);
```

```
window.a = 7;
window.alert(b);
```

← These are equivalent

- Global variables created using the let keyword are NOT added as properties on the window
- Parent for all of the other browser objects
  - Holds references to other objects, e.g.
    - self/window – Reference to current window
    - Navigator – Reference to current browser info
    - top, parent - References to frames
    - Document – Reference to current document

3

The window object is the top-level object in client-side JavaScript. All other objects belong to it, either directly or indirectly. In particular, global variables created using the var keyword are implemented as properties of the window object containing the document in which they are declared. Similarly, global functions are simply methods of the containing window. Because of this, global variables (and functions) can be accessed from other windows (using dot-notation) provided that you have a reference to the window which owns them.

The current window can be defined as being the window to which the code currently executing belongs; note that this is not necessarily the same as the window in which the code is being executed. The current window reference is implicit in all identifiers, and thus, there is no need to prefix a window method call, function call, or global variable with a reference to the window – the current window is automatically assumed. Hence, earlier in the course we pretended that alert(), for example, was a top-level function. Strictly speaking this is not the case – it is a method of the window object – but there is no need to specify the window object, so it appears as if it is a top-level function.

Window objects have a property called self, which refers to the window object itself. self and window can thus be used interchangeably. One of the few circumstances in which you need an explicit reference to the window object is if you want to pass it to another function, as in someFunction(self).

Three further properties refer to other objects. navigator refers to the navigator object, there is only one navigator object and all windows refer to the same one. For top-level windows, top and parent both refer to the object itself (like window and self).

## The location object

- The href property gives you the full url
  - `http://www.somename.com:80/somepath/filename.html#top`
- Other properties can be used to access sub parts of the URL
  - `hash` (`#top`)
  - `host` (`www.somename.com:80`)
  - `hostname` (`www.somename.com`)
  - `pathname` (`somepath/filename.html`)
  - `port` (`80`)
  - `protocol` (`http`)
  - `search` (e.g. `?productnum=42`)
- Methods
  - `reload([true])` – To reload page
  - `replace(URL)` – To replace entry in history

6

The location property of a window is a location object that describes the window's current location. If you refer to the location object itself (e.g. `window.location`), it returns/sets the complete URL, and thus, behaves like its href property (strictly speaking, it calls the `toString()` method of the location object). The slide above shows the various properties of the object and the parts of the URL that they reflect.

Assigning to the object or any of its properties will cause the browser to jump to the given location, and is a very powerful technique.

Where present `reload()` causes the current URL to be reloaded. If the optional `true` parameter is specified, then the URL will be unconditionally reloaded from the server; otherwise, it may be reloaded from cache. The `replace()` method causes the given URL to be loaded but it will replace the entry for the current location in the history list, rather than adding a new entry (hence giving the impression the previous page has not been visited this browsing session).

## The history object

- Methods for navigating
  - `back()`
  - `forward()`
  - `go(int)`

```
<a href="JavaScript:window.history.back();">go back</a>
```

7

The history object contains the URLs of the places visited in the current session. This matches the URLs that can be selected from the Go menu. For security reasons, the history object does not expose the actual URLs in the browser history. This is a general security issue, and is handled at the window object level. Basically, JavaScript running in one window cannot access anything in another window that came from a different domain than it did itself. This includes frames, since frames are just window objects.

You can use this to provide the user with forward and back controls on your pages. Each window object has its own history object, and hence navigating forwards and backwards in a frame typically has a different effect to going forwards and backwards in the top-level window.

The history object has three methods: `go(n)`, `back()` and `forward()`. `back()` is equivalent to `go(-1)` and `forward()` is equivalent to `go(1)`.

## Manipulating windows

- It is possible to create 'popup' style windows via JavaScript

- Use these very sparingly
- Many browsers suppress them

- Open a window by calling

- `window.open(URL, windowname, [options])`
  - Options are specified as a single comma-separated list (no spaces)
  - Default is 'yes', if any value is specified, then all others default to 'no'

toolbar	(yes no)	(1 0)
location	(yes no)	(1 0)
directories	(yes no)	(1 0)
status	(yes no)	(1 0)
menubar	(yes no)	(1 0)
scrollbars	(yes no)	(1 0)
resizable	(yes no)	(1 0)
width	(pixels)	
height	(pixels)	
top	(pixels)	
left	(pixels)	

```
msgWin = window.open("", "Message", "toolbar=no,width=200,height=200");
```

8

The `window.open()` method enables us to open another browser window. It takes three parameters: the URL of the document to load, the window name, and a string describing the window features. The window features string takes the options shown above; typically, they are simple yes/no parameters, apart from width, height, top and left, which are in pixels. The Boolean options are set to true if they are specified without values, or as yes or 1, for example: `open("", "messageWindow", "toolbar")` and

`open("", "messageWindow", "toolbar=1")` both set the toolbar on.

If `windowname` does not specify an existing window and you do not specify options, the new window will have the same window furniture as the one which executes `window.open()`. If you specify any item in options, all other Boolean options are false unless you explicitly specify them.

In modern browsers (e.g. Firefox, IE7), you will find it is not possible to turn off some of the options (e.g. the status bar). This is for security reasons – a hacker can run script to open a browser that hides the status and address bars then you might not be aware that you are using http not https or that you are on a spoofed site. Instead, you might think you are on the genuine internet bank site and type in your bank account and credit card details, which the hacker can then read and use to remove money from your account.



**Pop-up windows should be used sparingly since they cause issue with accessibility, performance and scope. Consider iFrames and AJAX as an alternative.**



## Document methods

- Rarely used outside testing, DOM Programming has replaced them

- Use innerHTML instead

- Open a document for writing

- Creates a new data stream for writing, existing contents erased
  - Optionally specify the MIME-type
  - Using 'replace' causes history entry to be replaced

```
document.open([mime-type [, "replace"]]);
```

- Write to the open document

```
document.write(expression [, expression, ...]);
```

- Will open a stream, if not already open, existing contents erased

- Close the document

```
document.close();
```

- Flushes buffer

3

The `document.open()` method opens the data stream for a given document so that it can be written to. Note that this is opening a document object and not any underlying file or URL. The action of opening the stream erases the current contents. You can specify a MIME type for the stream to tell the browser what type of data you are planning to write to the stream. If you do not specify a MIME type, the stream defaults to text/html. If the MIME type is text/html then you can use the optional "replace" parameter to specify that the existing history entry for the document should be replaced with one for the new document, rather than a new entry being appended.

Beware of the difference between the `document.open()` method and the `window.open()` method. Although the window method could be called simply by writing `open()`, it is common to explicitly write `window.open()` in order to avoid confusion.

As we have already seen, we can write data into a document stream using the `write()` method, there is also a `writeln()` method. If the stream is closed, it must first be re-opened, and as we have seen, opening the stream erases the current contents. It is, therefore, very dangerous, and rarely useful, to write into the current document from an event handler, since this will overwrite the existing contents, including the event handler itself! Opening and writing to a stream for another window (or layer) is perfectly safe though, and is the most common use of this feature.

The `close()` method closes an open document stream, flushing any buffered data and causing rendering to be completed.

## Cookies

- Data maintained by browser
  - Held until end of session or expiry date
  - Useful for maintaining state between pages or visits to a site
- Cookie definition is made up of many parts

```
"name=value;expires=...;path=...;domain=...;secure"
```

- Value may not contain whitespace, semicolons, or commas
  - Use `escape()/unescape()` to encode/decode
- Date must be a UTC string
  - Such as returned by `toUTCString()`
- Accessible to JavaScript through `document.cookie`

10

One of the biggest problems in web programming, whether client-side or server-side, is that of maintaining state between pages. A cookie is a small chunk of data that is maintained by the browser and associated with a directory tree on either a single server or a whole internet domain. It is primarily intended for use by server-side applications; however, it is also possible to access cookies from client-side JavaScript, by manipulating the `cookie` property of `document`.

A cookie is simply a name /value pair. It may have an optional expiry date (in the exact format produced by `toUTCString()`) associated with it. In the absence of any such date, the cookie automatically expires at the end of the current browser session, and will be discarded at that point. Otherwise, it will be maintained across sessions up until the expiry date. It may have some other parameters also: `path` specifies the directory on the server with which this cookie is associated. `Domain` specifies the servers to which this cookie is available; normally, this is the hostname of the server from where the cookie originated, but it is possible to set it to the domain or super-domain. A cookie marked as `secure` will only be sent to a server if the connection is secure (e.g. https over SSL).

## Writing and reading cookies

- Cookies accessible through `document.cookie`
  - Writing adds, updates, or deletes a cookie for the page
    - Assign to `document.cookie` property
    - Will append automatically if you set more than one cookie

```
document.cookie = "name=" + escape(username);
document.cookie = "drink=" + escape(selectedDrink);
```

- Reading returns a semicolon-separated list of cookies
  - Write a function to extract value from string
  - Need to use `indexOf()`, `substring()` and `unescape()`

```
"name=darren; drink=tea"
```

11

When you write to the cookie property, you can add, modify, or delete a cookie. Only the name=value field is required – the expires, path, domain and secure parts are optional. If the name matches an existing cookie, then the existing cookie will be replaced, if the expiry date is some time in the past, then the cookie will be deleted.

When you read the cookie property of a document, you get back a semi-colon-separated list of all the cookies (name=value pairs), which apply to this document. To extract the values of a particular cookie, you will need to parse this string with code like:

```
var c = document.cookie;
var s = c.indexOf("name=");           // find the start
c = c.substring(s+5, c.length);      // discard before
var e = c.indexOf(";");              // find the end
if (e != -1)
{
    c = c.substring(0,e)              // discard after
}
c = unescape(c);                     // unescape the value
```

## The screen – widths and height

- The screen object contain information about the user's display
  - Useful for animation

Property	Description
availHeight	Returns the height of the screen (excluding the Windows Taskbar)
availWidth	Returns the width of the screen (excluding the Windows Taskbar)
colorDepth	Returns the bit depth of the colour palette for displaying images
Height	Returns the total height of the screen
pixelDepth	Returns the colour resolution (in bits per pixel) of the screen
width	Returns the total width of the screen

12

The Browser Object Model provides several properties that provide information about your visitor's screen. Most of these are only of use for statistical purposes and are meaningless in so far as any JavaScript processing is concerned.

There are only two properties that are at all useful with regard to JavaScript processing and these two are only relevant with respect to those window size and position options that we discussed in the last tutorial.

These are **screen.availWidth** and **screen.availHeight**, which give the width and height of the available area of the screen (after subtracting any fixed toolbars) into which you can open, move, or resize a browser window.

## The viewport

- The viewport is the term given to the visible part of the web browser
  - It does not include the 'chrome' of the browser
- Most browsers allow us just to access `innerWidth` or `outerWidth`

```
console.log(window.innerWidth);
```

- `window.outerWidth` gets the width of the outside of the browser
  - It represents the width of the whole browser window
    - including sidebar, window chrome
- `window.innerWidth` gets the width the browser window
  - including, if rendered, the vertical scrollbar.

13

Probably the most useful information that you can obtain from the browser is the size of the viewport. This is the area that the browser makes available in which to display your web page and is the size that you need to be concerned with in determining how your page will appear.

A lot of people make the mistake of retrieving the screen size and assuming that it will reflect the space that their page will display in but the relationship between screen size and browser viewport area is a tenuous one since – with some operating systems – the browser itself can at its biggest be the size of the screen less fixed toolbars, and the browser itself will have toolbars etc. That makes the viewport smaller. Other operating systems support multiple virtual screens and the browsers can easily span across several of these, allowing it to be opened much larger than the size of a single screen.

There are three different ways that browsers provide access to the size of the viewport. One of these is used by all of the common browsers, except for Internet Explorer, and that is to provide the information on the viewport width and height in **`window.innerWidth`** and **`window.innerHeight`**. How Internet Explorer provides the information depends on which doctype your page uses. A strict doctype means that the values get loaded into

**`document.documentElement.clientWidth`**

and

**`document.documentElement.clientHeight`**

while not having a strict doctype loads the values into

**`document.body.clientWidth`** and **`document.body.clientHeight`** instead.

## Review

- What is the BOM?
  - A browser-specific collection of objects and arrays
- The window object
  - Global scope variables
- Other BOM objects
  - The location object
  - The history object
  - Document methods
  - Cookies and how to create and read them
  - The screen object for browser width and height
  - The viewport as the visible part of the browser
  - Popup windows – and why to rarely use them

## Exercise

- Working with cookies using BOM programming
  - Creating temporary cookies
  - Checking for cookies
  - Reading from cookies
  - Conditionally displaying content
  - Debugging with the network panel