The BOM Only Makes Sense in a Browser Environment

Remember that JavaScript can be run in different environments. The BOM only makes sense in a browser environment. This means that other environments (such as Node.js) probably won’t have a window object, although they will still have a global object; for example, Node.js has an object called global .

If you don’t know the name of the global object, you can also refer to it using the keyword this in the global scope. The following code provides a quick way of assigning the variable global to the global object:

// from within the global scope

const global = this;

### **Dialogs**

In Chapter 1, we introduced three functions that produced dialogs in the browsers: alert() , confirm() and prompt() . These are not part of the ECMAScript standard, although all major browsers support them as methods of the window object.

The window.alert() method will pause the execution of the program and display a message in a dialog box. The message is provided as an argument to the method, and undefined is always returned:

window.alert('Hello');

<< undefined

Confirm dialog

The window.prompt() method will stop the execution of the program. It displays a dialog that shows a message provided as an argument, as well as an input field that allows the user to enter text. This text is then returned as a string when the user clicks OK. If the user clicks Cancel, null is returned:

window.prompt('Please enter your name:');

### **Location, Location, Location**

The window.location property is an object that contains information about the URL of the current page. It contains a number of properties that provide information about different fragments of the URL.

The href property returns the full URL as a string:

window.location.href

<< "https://www.sitepoint.com/premium/books/javascript-novice-to-ninja"

This property (as well as most of the others in this section) is a read/write property, which means it can also be changed by assignment. If this is done, the page will be reloaded using the new property. For example, entering the following line into the browser console will redirect the page to the SitePoint JavaScript channel:

window.location.href = 'https://www.sitepoint.com/javascript/'

<< "https://www.sitepoint.com/javascript/"

The protocol property returns a string describing the protocol used (such as http , https , pop2 , ftp etc.). Note that there is a colon ( : ) at the end:

window.location.protocol

<< "https:"

The host property returns a string describing the domain of the current URLandthe port number (this is often omitted if the default port 80 is used):

window.location.host

<< "www.sitepoint.com"

The hostname property returns a string describing the domain of the current URL:

window.location.hostname

<< "www.sitepoint.com"

The port property returns a string describing the port number, although it will return an empty string if the port is not explicitly stated in the URL:

window.location.port

<< ""

The pathname property returns a string of the path that follows the domain:

window.location.pathname

<< "/premium/books/javascript-novice-to-ninja"

The search property returns a string that starts with a “?” followed by the query string parameters. It returns an empty string if there are no query string parameters. This is what I get when I search for “JavaScript” on SitePoint:

window.location.search

<< "?q=javascript&limit=24&offset=0&page=1&

content\_types[]=All&slugs[]=all&states[]=available&order="

The hash property returns a string that starts with a “#” followed by the fragment identifier. It returns an empty string if there is no fragment identifier:

window.location.hash

<< ""

The origin property returns a string that shows the protocol and domain where the current page originated from. This property is read-only, so cannot be changed:

window.location.origin

<< "https://www.sitepoint.com"

The window.location object also has the following methods:

* The reload() method can be used to force a reload of the current page. If it’s given a parameter of true , it will force the browser to reload the page from the server, instead of using a cached page.
* The assign() method can be used to load another resource from a URL provided as a parameter, for example:

window.location.assign('https://www.sitepoint.com/')

* The replace() method is almost the same as the assign() method, except the current page will not be stored in the session history, so the user will be unable to navigate back to it using the back button.
* The toString() method returns a string containing the whole URL:

window.location.toString();

<< "https://www.sitepoint.com/javascript/"

## **Controlling Windows – COOL STUFF!**

A new window can be opened using the window.open() method. This takes the URL of the page to be opened as its first parameter, the window title as its second parameter, and a list of attributes as the third parameter. This can also be assigned to a variable, so the window can then be referenced later in the code:

const popup = window.open('https://sitepoint.com','

SitePoint','width=400,height=400,resizable=yes');

## **Screen Information**

The window.screen object contains information about the screen the browser is displayed on. You can find out the height and width of the screen in pixels using the height and width properties respectively:

window.screen.height

<< 1024

window.screen.width

<< 1280

The availHeight and availWidth can be used to find the height and width of the screen, excluding any operating system menus:

window.screen.availWidth

<< 1280

window.screen.availHeight

<< 995

The colorDepth property can be used to find the color bit depth of the user’s monitor, although there are few use cases for doing this other than collecting user statistics:

window.screen.colorDepth;

<< 24

#### More Useful on Mobile. - IMPORTANT!!

The Screen object has more uses for mobile devices. It also allows you to do things like turn off the device’s screen, detect a change in its orientation or lock it in a specific orientation.

#### Use With Care

Many of the methods and properties covered in the previous section were abused in the past for dubious activities such as user-agent sniffing, or detecting screen dimensions to decide whether or not to display certain elements. These practices have (thankfully) now been superseded by better practices, such as media queries and feature detection, which is covered in the next chapter.

## **The Document Object**

Each window object contains a document object. This object has properties and methods that deal with the page that has been loaded into the window. In Chapter 6, we covered the Document Object Model and the properties and methods used to manipulate items on the page. The document object contains a few other methods that are worth looking at.

### **document.write()**

The write() method simply writes a string of text to the page. If a page has already loaded, it will completely replace the current document:

document.write('Hello, world!');

This would replace the whole document with the string Hello, world! . It is possible to include HTML in the string and this will become part of the DOM tree. For example, the following piece of code will create an <h1> tag node and a child text node:

document.write('<h1>Hello, world!</h1>');

The document.write() method can also be used within a document inside <script> tags to inject a string into the markup. This will not overwrite the rest of the HTML on the page. The following example will place the text "Hello, world!" inside the <h1> tags and the rest of the page will display as normal:

<h1>

<script>document.write("Hello, world!")</script>

</h1>

#### Creating Cookies

To create a cookie, you assign it to JavaScript’s “cookie jar”, using the document.cookie property, like so:

document.cookie = 'name=Superman';

<< "name=Superman"

The document.cookie property acts like a special type of string. Assigning another cookie to it won’t overwrite the entire property, it will just append it to the end of the string. So we can add more cookies by assigning them to document.cookie :

document.cookie = 'hero=true';

<< "hero=true"

document.cookie = 'city=Metropolis';

<< "city=Metropolis"

## **Timing Functions**

### **setTimeout()**

The window object provides some useful methods for scheduling the execution of a function, and for repeatedly executing functions at regular intervals.

The window.setTimeout() method accepts a callback to a function as its first parameter and a number of milliseconds as its second parameter. Try entering the following example into a console. It should show an alert dialog after three seconds (that’s 3000 milliseconds):

window.setTimeout( () => alert("Time's Up!"), 3000);

<< 4

Notice that the method returns an integer. This is an ID used to reference that particular timeout. It can also cancel the timeout using the window.clearTimeout() method. Try calling the code again and make a note of the number that is returned:

window.setTimeout( () => alert("Time's Up!"), 3000);

<< 5

## **Animation – very interesting**

The setTimeOut() and setInterval() methods can be used to animate elements on a web page. As an example, let’s create a web page that shows a colored square, and make it rotate. Create a folder called animation that contains files called index.html , styles.css and main.js . Place the following code inside index.html :

<!doctype html>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>Animation Example</title>

<link rel='stylesheet' href='styles.css'>

</head>

<body>

<div id='square'></div>

<script src='main.js'></script>

</body>

</html>

This places a div on the page with an ID of square .

Next, add the following styles.css :

#square {

margin: 100px;

width: 100px;

height: 100px;

background: #d16;

}

### **requestAnimationFrame**

This method of the window object works in much the same way as the window.setInterval() method, although it has a number of improvements to optimize its performance. These include making the most of the browser’s built-in graphics-handling capabilities, and not running the animation when the tab is inactive, resulting in a much smoother performance. It’s supported in all major browsers, including Internet Explorer from version 10 onwards. Change the code in main.js to the following:

const squareElement = document.getElementById('square');

let angle = 0;

function rotate() {

angle = (angle + 2)%360;

squareElement.style.transform = `rotate(${angle}deg)`

window.requestAnimationFrame(rotate);

}

const id = requestAnimationFrame(rotate);

This is similar to the earlier code, but this time we place the rotation code inside a function called rotate . The last line of this function uses the window.requestAnimationFrame() method and takes the rotate() function as an argument. This will then call the rotate() function recursively. The frame rate cannot be set using requestAnimationFrame() ; it’s usually 60 frames per second, although it’s optimized for the device being used.

To start the animation, we need to call the requestAnimationFrame() method, giving the rotate() function as an argument. This will return a unique ID that can be employed to stop the animation using the window.cancelAnimationFrame() method:

cancelAnimationFrame(id);

API :

### **Geolocation**

The Geolocation API is used to obtain the geographical position of the device. This means it can be used to find the user’s exact location, then link to nearby places or measure the speed at which the user is moving. This information can then be used to filter data based on the user's location or speed and direction of travel. An example of this might be a search function that returns results based on your location. Because of privacy concerns, permission to use this has to be granted by the user first.

If geolocation is available, it will be a property of the navigator object that we met in Chapter 9. This property has a method called getCurrentPosition() that will return a position object to a specified callback function, called youAreHere() in the example:

navigator.geolocation.getCurrentPosition(youAreHere);

function youAreHere(position) {

console.log(`Latitude: ${position.coords.latitude}, Longitude: ${position.coords.longitude}`);

}

### **Multimedia**

Before HTML5, it was notoriously difficult to display audio and video in browsers, and plugins such as Flash often had to be used. HTML5 introduced the <audio> and <video> tags used to insert audio and video clips into a web page. It also introduced a Media API for controlling the playback of the clips using JavaScript.

An audio clip can be inserted into a page with the <audio> tag, using the src attribute to point to the audio file:

<audio src='/song.mp3' controls>

Your browser does not support the audio element.

</audio>

A video clip can be inserted with the <video> tag, using the src attribute to point to the movie file:

<video src='http://movie.mp4' controls>

Your browser does not support the video element.

</video>

## **Drawing with Canvas**

The canvas element was introduced to allow graphics to be drawn onto a web page in real time using JavaScript. A canvas element is a rectangular element on the web page. It has a coordinate system that starts at (0,0) in the top-left corner. To add a canvas element to a page, the <canvas> tag is used specifying a height and width . Anything placed inside the tag will only display if the canvas element is unsupported:

<canvas id='canvas' width='400' height='400'>Sorry, but your browser does not support the canvas element</canvas>

This canvas can now be accessed in a JavaScript program using the document.getElementById() method:

const canvasElement = document.getElementById('canvas');

The next step is to access the context of the canvas. This is an object that contains all the methods used to draw onto the canvas. We'll be using a 2-D context, but it’s also possible to render in 3-D using[WebGL](https://developer.mozilla.org/en/docs/Web/API/WebGL_API/Tutorial/Getting_started_with_WebGL).

The getContext() method is used to access the context:

const context = canvasElement.getContext('2d');

Now we have a reference to the context, we can access its methods and draw onto the canvas. The fill and stroke colors can be changed by assigning a CSS color to the fillStyle and strokeStyle properties respectively:

context.fillStyle = "#0000cc"; // a blue fill color

context.strokeStyle = "#ccc"; // a gray stroke color

These colors will be utilized for everything that’s drawn onto the canvas until they’re changed.

The lineWidth property can be used to set the width of any line strokes drawn onto the canvas. It defaults to one pixel and remains the same until it’s changed:

context.lineWidth = 4;

#### A Factorizing Example

Back in Chapter 10, we created a function that found the factors of a given number. This works well, but can take a long time to find the factors of large numbers. If it was used in a website, it would stop any other code from running while it calculated the factors. To demonstrate this, save the following code in a file called 'factors.html':

<!doctype html>

<html lang='en'>

<head>

<meta charset='utf-8'>

<title>Factorizor</title>

</head>

<body>

<button id='rainbow'>Change Color</button>

<form>

<label for='number'>Enter a Number to Factorize:</label>

<input id='number' type='number' name='number' min=1 value='20'>

<button type='submit'>Submit</button>

</form>

<div id='output'></div>

<script src='main.js'></script>

</body>

</html>

This web page has a button that will change the background color of the page, and an input field where a number can be entered. The factors will be displayed inside the output div. To get this working, create a file called main.js in the same directory as 'factors.html' that contains the following code:

const btn = document.getElementById('rainbow');

const rainbow = ['red','orange','yellow','green','blue','rebeccapurple','violet'];

function change() {

document.body.style.background = rainbow[Math.floor(7\*

Math.random())];

}

btn.addEventListener('click', change);

This first piece of code was covered way back in Chapter 1 and uses an event listener to change the background color if the button is clicked. We also need to factorize the number entered in the form, so add this code to the end of main.js :

const form = document.forms[0];

form.addEventListener('submit', factorize, false);

function factorize(event) {

// prevent the form from being submitted

event.preventDefault();

const number = Number(form.number.value);

document.getElementById('output').innerText = factorsOf(number);

}

function factorsOf(n) {

if(Number.isNaN(Number(n))) {

throw new RangeError('Argument Error: Value must be an integer');

}

if(n < 0) {

throw new RangeError('Argument Error: Number must be positive');

}

if(!Number.isInteger(n)) {

throw new RangeError('Argument Error: Number must be an integer');

}

const factors = [];

for (let i=1 , max = Math.sqrt(n); i <= max ; i++) {

if (n%i === 0){

factors.push(i,n/i);

}

}

return factors.sort((a,b) => a - b);

}

This uses the same factorsof() function from Chapter 10 and adds a submit event listener to the form. When the form is submitted, it will find the factors of the number in the input field, then place the result inside the output div.