JavaScript:

Advanced JavaScript Coding from The Ground Up

The information in the following pages is broadly considered to be a truthful and accurate account of facts, and as such any inattention, use or misuse of the information in question by the reader will render any resulting actions solely under their purview. There are no scenarios in which the publisher or the original author of this work can be in any fashion deemed liable for any hardship or damages that may befall them after undertaking information described herein.

Additionally, the information in the following pages is intended only for informational purposes and should thus be thought of as universal. As befitting its nature, it is presented without assurance regarding its prolonged validity or interim quality.

Table of Contents

Introduction

Thank you for downloading JavaScript: Advanced JavaScript Coding from The Ground Up! This book is the third entry of the DIY JavaScript series. It is preceded by the books:

- JavaScript: Beginner JavaScript Coding from The Ground Up
- JavaScript: Intermediate JavaScript Coding from The Ground Up

…and assumes that the user is familiar with the contents of those books, including JavaScript code, syntax, and beginner to intermediate concepts. The books should be taken in chronological order for optimal results. This book will cover advanced JavaScript manipulation techniques, with code examples to match the topics explained.

Thank you again for downloading this book! You have many choices available to you for furthering your JavaScript coding knowledge. Thank you for selecting the DIY JavaScript series as your tool of choice!

Chapter 1: Introduction to DOM

What is DOM?

DOM stands for Document Object Model; it is an API (Application Programming Interface) for accessing and modifying HTML or XML documents. It represents the document and elements in the document as objects and properties; this helps in connecting a scripting language with the document. In our case, that scripting language is JavaScript. DOM is roughly structured in the same way as the HTML or XML document. Thus it helps to analyze and manipulate the document closely.

The standards for DOM are specified by the W3C (World Wide Web Consortium). The main objective of DOM as stated by W3C is to provide a standard programming based interface for HTML and XML based documents for a variety of programming languages; it is not limited to JavaScript only. DOM was designed in such a way that it provides a consistent API for all languages that it is intended to be used

on. Its general syntax is independent of the language that it is being used.

JavaScript and DOM

With the help of DOM, JavaScript can access and modify the HTML document; DOM is not a programming language itself! Just as you access objects and modify their properties in JavaScript, in the same manner, you can access and edit DOM and the changes made will be reflected in the HTML document in real time. DOM, along with JavaScript, helps us to create a more dynamic web page and give the user a more delightful experience.

Things you can do with DOM

You can:

- Search for an element
- Modify an element's content
- Add an element
- Delete an element
- Modify an element's attribute
- Interact with the events emitted by elements, such as click, hover, focus, etc. events.

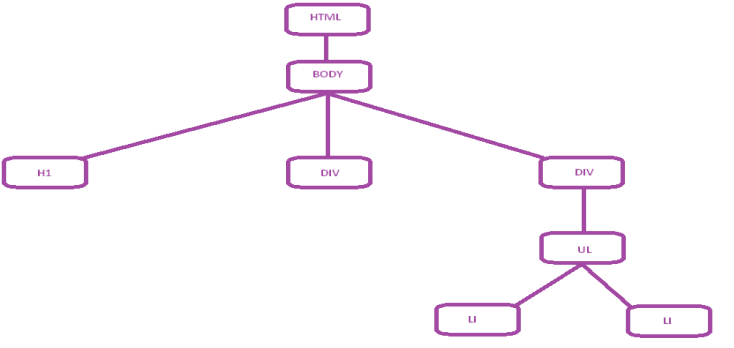
- Create custom events based on the document
- Change CSS styling of elements

Understanding DOM structure

DOM contains objects linked with each other in a logical way, such that they closely resemble the structure of a tree. Let us understand this with an example. Consider the following HTML document:

```
<html>
<body>
     <h1>My page</h1>
     <div>My name is Linus</div>
     <div>My hobbies are:
          <ul>
               <li>Programming</li>
               <li>Listening to music</li>
          </ul>
     </div>
</body>
</html>
```

The above HTML code would represent a DOM of the following structure:



You can see this structure closely resembles a tree data structure, but as the size of documents can grow drastically, the size may sometimes refers to as a ‘forest.'

The ‘document’ object

As in the example above, the ‘HTML’ is the root of all the elements. The ‘document’ object is in fact, the root of all elements and methods in a DOM. All elements in the documents are either children or sub-children of this object. You can also access all the document manipulation related methods using this object.

The ‘window’ object

The ‘window’ object represents the current window; it is considered the top-level object since even the ‘document’ object is a child of this object which can be accessed by the referring to ‘document’ property of the ‘window’ object, example: window.document. This object generally gives you access to window related methods such as closing the window, setting window’s scroll offset, etc. Note that the ‘window’ object is not considered a part of DOM. The DOM is a child of this object.

Understanding DOM is a crucial part of becoming a front-end developer. The main purpose of JavaScript in the front end is to be able to use DOM. Armed with the knowledge of what the DOM is, you can develop more dynamic and interactive websites.

Chapter 2: An overview of DOM Document object

In the previous chapter, we briefly discussed the document object; we shall study it in more detail in this chapter. Previously, we discussed that the document object is the root of the DOM, and every other element and methods are its child or sub-child.

In layman's language, the document object represents our full HTML page. Our HTML page in JavaScript is the document object. It contains the content of the whole page. The document object is supported by all the major browsers.

What exactly does document object contain?

The document object contains all sorts of methods and properties that you may need to interact with the HTML document. Some of the great functionality that document object provides are:

- It contains methods to access various elements of the document.

- You can search for elements under various criteria, using methods provided by the document object.
- It contains a method to create new HTML elements.
- It contains a method to remove HTML elements.
- It contains a method to replace HTML elements with other elements.
- It contains a method to add plain text to HTML document.
- You can check the current state of the document – i.e., whether it's in the stage of being loaded, or has loaded already using a property of the document object.
- The document object's properties contain some useful information. Some of the useful information you can retrieve with the document object are:
    - The domain name of the server on which the HTML document is located.
    - The exact URL to access the HTML document.
    - The URL of HTML document.
    - The title of the HTML document.
    - The last modification date of the HTML document.

    - The encoding of the HTML document
    - and much more.
- You can access cookies associated with the current domain with the help of the document object.
- With the help of properties of the document object, you can also retrieve the list of all:
    - <form> elements
    - <img> elements
    - <a> elements
    - <script> elements
    - <embed> elements
- You can get the <html>, <head> and the <head> element using properties of the document object.
- You can close the current HTML document using a method provided by the document object.

Major properties of the document object

Here is the list of all the major properties of the document object, along with their description:

- document.domain

  This property contains the domain name of the server on which the current document is hosted on.

Example, assuming that the URL of the document is ‘www.mysite.com/doc.html’:

```
console.log( document.domain ); // prints ‘www.mysite.com’ on the console screen
```

- document.URL

  This property contains the exact URL used to access the document.

  Example, assuming that the URL of the document is ‘www.mysite.com/doc.html’:

```
console.log( document.URL ); // prints ‘www.mysite.com/doc.html’ on the console screen
```

- document.cookie

  This property contains the list of all the cookies, stored in the browser which is associated with the current domain, separated by a semicolon(;) in string format.

  Example:

```
console.log( document.cookie ); //
example output
‘user=MyName;sessionid=SESSION_ID’
```

- document.title

  This property contains the title of the document, i.e., the text present between <title> tag in the head section of the HTML document.

  Example:

```
<html>
<head>
      <title>MyTest</title>
      <script>
            console.log( document.title
      );
            // prints ‘MyTest’ on the
      console screen
      </script.
</head>
</html>
```

If you modify this property, the title of the page changes to:

```
document.title = 'New title'; //
changes the document title to ‘New
title’
```

- document.documentElement

  This property is a reference to the <html> element’s object of the DOM. Once you retrieve the element as an object, you can do all sorts of things with it.

  Example changing the background of the <html> element of the document:

```
var htmlElement =
document.documentElement;
htmlElement.style.background = 'red';
// changing background color to red
```

- document.body

  This property is a reference to the <body> element’s object of the DOM.

Example changing the background of the <body> element of the document:

```
var bodyElement = document.body;
bodyElement.style.background = 'green'; // changing background color to green
```

- document.head

  This property is a reference to the <head> element's object of the DOM.

- document.images

  This property is essentially an array of all the <img> elements present in the current HTML document.

  Example:

```
<body>

<img scr='imagesource1' />
<img scr='imagesource2' />
<img scr='imagesource3' />

<script>
```

```
    console.log(
document.images.length ); // prints '3'
on the console screen

    document.images[0].style.width =
'300px'; // changes width of first
image

    document.images[2].style.height
= '100px'; // changes height of the
THIRD image

</script>
</body>
```

- document.forms

  This property is basically an array of all the <form> elements present in the current HTML document.

- document.links

  This property is basically an array of all the <a> elements which have the 'href' attribute present in the current HTML document.

Example:

```
<body>

<a href='link1' />
<a />
<a href='link2' />


<script>
     console.log(
document.links.length ); // prints ‘2’
on the console screen, since one <a>
element doesn’t have the ‘href’
attribute


     // now let’s change the textcolor
of all the links:
     for( var i = 0; i <
document.links.length; i++) {
```

```
    document.links[i].style.color = 'red';
    }

</script>
</body>
```

- document.anchors

  This property is similar to the .links property of the document object. The difference between them is that this property is an array of all the <a> elements which have a 'name' attribute.

  Example:

```
<body>

<a href='link1' />
<a name='myName'/>
<a href='link2' />

<script>
    console.log( document.anchors.length ); // prints
```

```
'1' on the console screen, since only
one <a> element has the 'name'
attribute


</script>
</body>
```

- document.scripts
  This property is essentially an array of all the <script> elements present in the current HTML document.

With the help of the information in this chapter, you can start building HTML pages which are somewhat dynamic, like a page that changes its title every few seconds.

The document object also grants access to useful methods which are not discussed in this chapter, but WILL be discussed in future chapters.

Chapter 3: Finding HTML elements using DOM

DOM provides us with some awesome functions to search for elements and retrieve them as an object. Using an element's retrieved object, we can modify its content, attributes, change its CSS styling, delete it or replace it with another element. All of these search functions can be accessed through document object. These functions are infinitely easier to use compared to older methods, where developers used to loop through every element to search for specific elements. In this chapter, we will learn how to use these search methods.

Discussed below are the methods to search for elements in HTML.

The document.getElementById() method

This is the most frequently, and perhaps the easiest function used to find elements in DOM. This method takes the ID of the element as its argument in string format, which we

specify along with the element's tag name in the HTML code.

Syntax:

```
document.getElementById( 'ELEMENT_ID' );
```

Example:

```
<body>

<p id='gtext'> Color of this text will be green</p>
<p id='rtext'> Color of this text will be red</p>

<script>

      var rtext= document.getElementById('rtext');
      var gtext= document.getElementById('gtext');

      rtext.style.color = 'red'; // changing text color to red
```

```
      gtext.style.color = 'green'; //
changing text color to green

</script>
</body>
```

This method only returns one object. If multiple elements have the same ID, only the first element with that ID is returned.

Example:

```
<body>

<p id='rtext'> Color of this text will be
red</p>
<p id='rtext'> This element’s color won’t
change!</p>

<script>

      var rtext=
document.getElementById('rtext');
```

```
      rtext.style.color = 'red'; // only
changes first element’s text color

</script>
</body>
```

If no element with the specified ID is found, then this method returns null.

Example:

```
<body>
<script>

      var rtext=
document.getElementById('rtext');

      console.log(rtext); // prints ‘null’
on the console screen

</script>
</body>
```

The document.getElementsByTagName() method

This method is used to perform a search based on the tag name of the element. You may have noticed the ‘s’ behind the word ‘Element’ in this method. That is because this method returns an array of element objects matching the given criteria. Unlike the .getElementById() method, this method can be used to get a list of all the elements with the same tag name. This method returns ‘null’ if no element was found matching the given criteria.

Syntax:

```
document.getElementsByTagName( 'TAG NAME' );
```

Example:

```
<body>

<p id='para1'> paragraph 1 </p>
<p id='para2'> paragraph 2 </p>
<p id='para3'> paragraph 3 </p>
<div id='div1'> Div 1 </div>
<div id='div2'> Div 2 </div>
```

```
<script>

      var ptags =
document.getElementsByTagName('p');
      var divtags=
document.getElementsByTagName('div');

      console.log( ptags.length ); prints
‘3’ on the console screen
      console.log( divtags.length ); prints
‘2’ on the console screen

      // now let’s change text color of each
p tag name element
      for( var i = 0; i < ptags.length; i++)
            ptags[i].style.color = 'red';

</script>
</body>
```

If you use an asterisk (*) symbol instead of a tag name in this method then this method returns an array of all the elements in the current page, including the <html>, <body> and all the <scripts> element.

The document.getElementsByClassName() method

This method is used to perform a search based on the class name of the element. Just like document.getElementsByTagName() method this method also returns an array of objects of all the elements matching the criteria. This method returns 'null' if no element was found matching the given criteria.

Syntax:

```
document.getElementsByClassName( 'CLASS NAME' );
```

Example:

```
<body>
```

```
<p id='classA classB'> paragraph 2 </p>
<p id='classB'> paragraph 3 </p>

<div id='classA'> Div 1 </div>
<div id='classB classC'> Div 2 </div>

<script>

      var aclass=
document.getElementsByClassName('classA');
      var bclass=
document.getElementsByClassName('classB');


      console.log( aclass.length ); prints
‘2’ on the console screen
      console.log( bclass.length ); prints
‘3’ on the console screen

      // now let’s change text color of
elements with classA
```

```
      for( var i = 0; i < aclass.length;
i++)
            aclass[i].style.color = 'red';

</script>
</body>
```

The document.querySelector() method

This method is used to perform a search based on the CSS selector, which is passed to this method as its argument. This method returns the object of the element matching the given CSS selector or returns null if no such element is found.

Syntax:

```
document.querySelector( 'CSS SELECTOR' );
```

Example:

```
<body>

<p class='classA'> This text color will be
red </p>
<p class='classB'> This text will have the
default color </p>
```

```
<script>

     var pWithaclass=
document.querySelector('p.classA'); // here
we are selecting the element with <p> tag
and class named ‘classA’

     pWithaclass.style.color = 'red';

</script>
</body>
```

This element returns the first element matching the given CSS selector, so if multiple elements are present which match the given CSS selector, only the object of the first matching element is returned.

The document.querySelectorAll() method

This method is similar to the document.querySelector() method as this method also performs a search based on the CSS selector that is passed to this method as its argument.

The difference between document.querySelector() method and this is that this method returns an array of ALL the elements matching the given criteria, whereas the former returned the object of the FIRST element matching the criteria. This method returns ‘null’ if no element was found matching the given CSS selector.

Syntax:

```
document.querySelectorAll( 'CSS SELECTOR' );
```

Example:

```
<body>

<p class='classA'> This text color will be
red </p>
<p class='classB'> This text will have the
default color </p>
<p class='classA'> This text color will be
red too </p>

<script>

     var pWithaclass=
document.querySelectorAll('p.classA');
```

```
      console.log( pWithaclass.length ); //
prints ‘2’ on the console screen


      //now let’s change the text color of
the elements that we searched for

      for(var i = 0; i < pWithaclass.length;
i++)
            pWithaclass[i].style.color =
      'red';

</script>
</body>
```

Knowing how to search for elements in DOM is very important, as everything that you will do in JavaScript will require searching for elements.

Chapter 4: Changing the content of HTML elements using DOM

We need to change the content of the HTML elements very often for making a dynamic web application or website. Some example scenarios include:

- Giving user feedback. Replacing an HTML element's content instead of using the 'alert' function is a better alternative to give feedback, as you can modify how an element looks and thus make your feedback display better visually. This will, in turn, lead to a more pleasurable user experience.
- Have you seen a live clock, or countdown, on a web application? They both replace their HTML element's content when the unit of time they function on passes by.
- Changing a page's content on a button click without refreshing.

The ‘.innerHTML’ property

This is perhaps the easiest and the most frequently used way of changing the content of an HTML element. This property of an element’s object in the DOM represents the current content of that element, and can be used to get or set the content of that element. If a value is assigned to this property of an element's object, then that element’s content changes to the newly assigned value.

Syntax:

```
elementObject.innerHTML = ‘new value’;//
setting
var x = elementObject.innerHTML;      //
getting
```

Note that this property represents everything within the element's tag, even it’s children and sub-children.

Example demonstrating getting an element's content:

```
<body>
<div id='outer'>
     <div id='inner'>
          Hey there
```

```
      </div>
</div>

<script>
      var innerContent =
document.getElementById('inner').innerHTML;
      var outerContent =
document.getElementById('outer').innerHTML;
      console.log( innerContent );
      console.log( outerContent );

</script>
</body>
```

The above code produces the following output on the console screen:

```
            Hey there

      <div id="inner">
            Hey there
      </div>
```

As you can see, the output contains the white spaces and the tabs.

TIP: If you wish to get rid of the extra white spaces and tab spaces at the start and the end of the content you can use the .trim() method from the string class on them!

Example demonstrating the changing of an element's content:

```
<body>
<div id='outer'>
     <div id='inner'>
          Hey there
     </div>
</div>

<script>

     var innerElement =
document.getElementById('inner');

     innerElement.innerHTML = 'Hola
amigos';
```

```
     // the content of <div> with id
‘inner’ has been changed to ‘Hola amigos’

</script>
</body>
```

The ‘.textContent’ property

This property is almost same as the ‘.innerHTML’ property. The difference between this property and ‘.innerHTML.’ is that this property is only used to get or set the textual content (meaning the children and sub-children element’s tag name is not included when getting the content of the element), whereas in the case of the ‘.inner.HTML’ property, the full content of the element including the children and sub-children element’s tag names and attribute were also returned.

Let us understand this by using the same example as we used in the case of the ‘.innerHTML.’ property – but instead, we will use ‘.textContent.’ property in place of the ‘.innerHTML’ property here.

```
<body>
<div id='outer'>
      <div id='inner'>
            Hey there
      </div>
</div>

<script>
      var innerContent =
document.getElementById('inner').textConten
t;
      var outerContent =
document.getElementById('outer').textConten
t;
      console.log( innerContent );
      console.log( outerContent );

</script>
</body>
```

The above code produces the following output on the console screen:

```
            Hey there

            Hey there

```

The white spaces and the tab spaces at the start and at the ending are included in the output of this result as well, but note that the tag is not!

Another difference between the two properties is that the content that we get using the '.innerHTML' property has the characters '&', '<', or '>' replaced by '&amp', '&lt' and '&gt' respectively, whereas the in the content obtained using '.textContent' property these characters remain as they are.

The setting action of both properties will work the same. Here is an example:

```
<body>
<div id='outer'>
     <div id='inner'>
          Hey there
     </div>
</div>
```

```
<script>

      var innerElement =
document.getElementById('inner');

      innerElement.textContent = 'Hola
amigos';
      // the content of <div> with id
‘inner’ has been changed to ‘Hola amigos’

</script>
</body>
```

Yet another major difference between the two properties is that when you set an element’s content using .innerHTML property, it is interpreted as HTML code whereas in the case of .textContent, it is interpreted as raw text.

Example:

```
<body>
<div id='outer'>
      <div id='inner'>
            Hey there
```

```
      </div>
</div>

<script>

      var innerElement =
document.getElementById('inner');
      // no alert box will pop up in this
case:
      innerElement.textContent =
"<script>alert('hey')</script>";

      // A alert box will pop up in this
case:
      innerElement.innerHTML =
"<script>alert('hey')</script>";

      // this was because, in the latter,
the text was interpreted as HTML code

</script>
</body>
```

Therefore, it is safer to use the .textContent property to set the content of an element instead of the .innerHTML property when you are expecting the new content to be plain text only.

Chapter 5: Changing CSS using DOM

CSS is used to style our web pages. Almost everything you see on modern sites has been styled using CSS. Thus, it becomes important for us to learn how to change CSS, which can help us to dynamically change the styling of our web page. Doing so will make our web page more interactive, giving the user a better experience. An example scenario – when a user is writing in a text box, you can make textbox background red to indicate that input is invalid, and green to indicate that it is valid. Have you ever noticed that while registering on some sites, when filling in the password, the password input box border color changes from red to orange and then to green to indicate the strength of password? This is done with the help of JavaScript.

The element.style property

As an attentive reader, you must have seen the usage of this property in some examples from the previous chapters. The '.style.' property of an HTML element's object is used to get and set the CSS style of the element.

We don’t directly assign a string containing CSS style definition to this property. Instead, we assign values to “sub-properties” of this property. Technically, this property is an object itself. You can access a property by using the convention “element.style.propertyName.” The naming convention of these “sub-properties” is almost the same as in CSS. For example: to change text color, we use the property “color” in CSS. To access or modify it in DOM, we can use element.style.color. Also, note that the naming convention of these CSS “sub-properties” in DOM is in camelcase not in kebab-case as in CSS, so the property ‘border-bottom’ becomes ‘borderBottom.'

The values assigned to these “sub-properties” should be in the same format as in CSS.

Discussed below are some of the important CSS properties and how you can get and set them using DOM.

Changing an element's background

You can get or set the background of an element using the '.background.' sub-property of the '.style.' property of an element.

Syntax:

```
var getVal = element.style.background //
GET
element.style.background = set;       //
SET
```

Example usage:

```
<body>

<p style='background:red' id='changeBg'>
      This paragraph background color was
      originally red but will change to blue
      by the JavaScript code!
</p>

<script>
var pElement =
document.getElementById('changeBg');
```

```
console.log( pElement.style.background );
// prints ‘red’ on the console screen

// now let's change its background to blue
pElement.style.background = 'blue';

console.log( pElement.style.background );
// prints ‘blue’ on the console screen

</script>
</body>
```

You can directly assign values to the .background sub-property in shortland format too, as you do in CSS. Example:

```
pElement.style.background = "#00ff00
url('image.png') no-repeat fixed center";
```

NOTE: If a color is specified in HEX format(#RRGGBB) in any of the CSS property, then while accessing it from DOM it will appear in ‘rgb(rrr, ggg, bbb)’ format.

Example:

```
var pElement =
document.getElementById('changeBg');
pElement.style.background = '#0000ff'; //
HEX representation of blue color

console.log( pElement.style.background );
// prints 'rgb(0, 0, 255)' on the console
screen
```

Changing an element's text color

You can get or set the text color of an element using the '.color' sub-property of the '.style' property of an element.

Syntax:

```
var getVal = element.style.color // GET
element.style.color = set;       // SET
```

Changing an element's text size

You can get or set the text size of an element using the '.fontSize' sub-property(note the camel case here) of the '.style' property of an element.

Syntax:

```
var getVal = element.style.fontSize   //
GET
element.style.fontSize = set;         //
SET
```

NOTE: When you assign a size to any size-related property like font size, height width, etc., the sizes should be assigned along with their units (px, em, vh, etc). Not doing so will result in an error.

Example:

```
element.style.fontSize = 40;     // WRONG
element.style.fontSize = '40';   // WRONG

element.style.fontSize = '40px'; // RIGHT
```

The ‘float’ property, an exception in property naming convention in DOM

The ‘float’ property in CSS is used to specify the position of an element in the layout. But ‘float’ is also a reserved word in JavaScript. Owing to this, we cannot use ‘float’ as a

property name in JavaScript. We use the name ‘cssFloat’ in JavaScript to refer to the ‘float’ property.

Syntax:

```
var getVal = element.style.cssFloat   //
GET
element.style.cssFloat = set;         //
SET
```

Modifying CSS style using element.setAttribute() method

The .setAttribute() method is used to change attributes of elements. We can change the CSS styling of an element by modifying their ‘style’ attribute. This attribute’s value is plain CSS.

Syntax:

```
element.setAttribute('style',   'PLAIN   CSS
HERE')
```

Example:

```
element.setAttribute('style',
'background:red; height:15px');
```

Modifying CSS style by adding a CSS class to it

You can also modify CSS styling of an element through JavaScript by adding a predefined CSS class to it. You can add classes to an element by adding them to 'className' property of the element.

Syntax for adding a CSS new class:

```
element.className += ' newClassName'
```

Note that there must be a space present in the string before the CSS class name as this property is merely a string containing all the name of all the CSS classes of the element separated by a space.

Example:

```
<body>

<p style='background:red' id='changeClass'>
      This paragraph's styling will be
      changed by JavaScript by adding a CSS
      class to it!
```

```
</p>

<style>
# our custom CSS class:
.myClass {
      background:blue;
      height:10px;
}
</style>
<script>

var pElement =
document.getElementById('changeClass');
pElement.className += ' myClass';

</script>
</body>
```

You may use any way you like to change the CSS styling, but doing it via CSS class names is considered a better practice. This helps by making a more structured codebase

similar to stateful architecture, where each class conceptually represents a state. Consider the scenario of weak and strong passwords as described in the beginning of this chapter. When the password has weak strength, you can add a class named ‘weakPassword’ to the password textbox and a class named ‘mildPassword’ when it is of mild strength. This helps us to write more organized code.

Chapter 6: Working with cookies

What are cookies?

Cookies are basically the data that is stored on a user's local machine by a website and that data can later be accessed by the same website, even after you close your browser or restart your computer.

Have you ever noticed that once you log into your account on a website, you stay logged in even when you close the browser and go to the site again? This is because when you logged on the website, it stored some information on your browser in the form of a cookie which can be used to identify your account. In layman's language, it is a way by which a website remembers your browser.

Due to security reasons, a cookie stored by a website on your browser is only accessible by that website only. If you go to a website – say x.com – and it stores a cookie on your browser, then go to y.com? The cookies stored by the website x.com would not be accessible by the y.com website.

Cookies have an expiration time. The expiration time can be set never to expire, and are always stored as a name-value pairing. This is similar to how we use a variable in JavaScript, where a variable has a name, and that name is used to refer to the value that variable stores. All cookie-based operations in JavaScript are done with the help of the document.cookie object.

Getting all cookies

If you try to access the document.cookie object, and do not modify it, it will return a string containing all the cookies in the name-value pair in the format 'name=value' where each cookie is separated by a semicolon (;).

Example:

```
var myCookies = document.cookie;

console.log( myCookies ); // prints a string containing all the cookies separated by a semicolon (;)
```

Example output of the above code:

```
username=John;email=john21@example.com
```

Getting specific cookies

If you want to get a specific cookie, the first step is to get a list of all cookies with the help of document.cookie, and then parse that object to get the specific cookie you are searching for. There are two main approaches to do it, which are:

- Using the split() method and then looping and comparing:

  In this method, we will use the fact that the cookies list we get from the document.cookie object contains the cookies separated by a semicolon(;), and then search for our desired cookie. Here is a function for doing so:

```
function cookieVal(c) {
     var cookies =
document.cookie.split(';'); //
splitting into separate cookies
     for(var i = 0; i <
cookies.length; i++) {
          var name =
cookies[i].substring(0,
cookies[i].indexOf('=')).trim();
```

```
        if(name == c) return
cookies[i].substring(cookies[i].indexO
f('=') + 1);
      }
      return '':
}
```

The function mentioned above will take the cookie name as its argument and returns its value if found. If no value is found, it returns an empty string.

- Using a regular expression to extract a cookie's value directly.

  As the title states, we will use a regular expression and directly extract the value of the cookie. Here is the function that uses a regular expression to extract a cookie's value:

```
function cookieVal(c) {
      var regex = new RegExp('.*' + c
+ '=(.*?);' , 'g');
      var result =
regex.exec(document.cookie);
      if(result == null)return '';
```

```
        return result[1];
    }
```

The function mentioned above will take the cookie name as its argument and returns its value if found, else it returns an empty string.

.

Both of the above-mentioned functions perform the same task, but do so in different ways.

Creating new cookies

This is done by assigning a string to a document.cookie object. The string must be in the format 'cookieName=myValue;', this will create a new cookie with the name 'cookieName' and the value 'myValue'. Note that you can create only one cookie during a single assignment operation.

Here is an example demonstrating how to create a new cookie with the name 'username' and the value 'Linus':

```
document.cookie = 'name=Linus;'
```

There are some optional parameters that you can use while creating a new cookie. These optional parameters are mentioned after the semicolon(;) of the name-value pair and are separated by a semicolon(;) from each other. The optional parameters are:

- path:

  This parameter is used to specify a path from which the newly created cookie can be accessed. If the path is specified, the cookie will be accessible from that path only. The default value of the path is the current path.

  Example:

  ```
  document.cookie = 'name=Linus;
  path=/users'
  ```

  The above example will create a cookie with the path parameter set to ‘users’ so this cookie can be accessed by the website when we go to a path like ‘currentDomain.com/users/something’. It will not be accessible from a path like ’currentDomain.com/NOTUSERS/something’.

- max-age:

  This parameter is used to set the 'living' time of the cookie in seconds. After the number of seconds specified as the value of this parameter has elapsed – starting from the moment of the creation of the cookie – the cookie will expire and get deleted.

  Example:

  ```
  document.cookie = 'name=Linus; max-age=3600'
  ```

  The above example will create a cookie that will expire after 1 hour(= 3600 seconds)

- expires:

  This parameter is similar to the above example. However, in this parameter we specify the exact date and time of the expiration of the cookie in GMT string format.

  Example:

  ```
  document.cookie = 'name=Linus; expires=Sat, 06 Sep 2017 00:00:00 GMT'
  ```

The above example will create a cookie that will expire at 6th of September, 2017 at the time 00:00.

- domain:
  This parameter is used to specify from which domain the cookie being created will be accessible. If not mentioned, it defaults to the current domain name.
  Example:
  ```
  document.cookie = 'name=Linus;
  domain=otherdomian.com'
  ```

  The above example will create a cookie that will accessible from the domain ‘otherdomain.com’.

Example demonstrating the use of multiple parameters at once:

```
document.cookie = 'name=Linus;
domain=otherdomian.com; max-age=3600'
```

Note that if neither the ‘max-age’ nor the ‘expires’ are specified then the cookie will expire at the end of the current session, i.e., when the browser is closed.

Modifying a pre-existing cookie

Modifying a cookie is a simple process. Create a new cookie of the same type as the cookie that you need to modify. The newly created cookie will be overwritten onto the existing one.

Deleting a cookie

This is also a rather simple process. To delete a cookie, you need to modify the cookie and set its expiry date to a date that has passed already.

Example:

```
document.cookie = 'name=Linus; expires=Wed, 29 Dec 9999 00:00:00 GMT;
```

The above example will delete the cookie with the name ‘Linus’ if it exists.

Cookies are a very important part of web applications. A good web application should remember its user’s data, and won’t ask them to perform the same tasks again and again. Cookies can also be used to store user preferences, such as

the theme they want to use when using the web application, YouTube also uses cookies to store a user's preferences.

Chapter 7: Introduction to BOM

BOM stands for Browser Object Model. Just as the DOM (Document Object Model) deals with the content of the page, the BOM deals with the items related to browser window other than the content of the page. Unlike DOM, no standards are defined for the BOM. BOM implementation may vary from browser to browser as a result, but it is generally same for all major and modern web browsers.

In JavaScript, everything in the window – including the content of the page, the menu bar, the address bar, the history, the window size etc. – is parsed as objects. The hierarchy of all these objects is known as the BOM. In layman's term, all these objects are collectively referred to as BOM. BOM helps us to manipulate and interact with the browser and do things such as redirecting the user to another URL, getting the window size, changing the text in the status bar, opening a new window etc. So technically the DOM, about which you have studied earlier, is a part of the BOM.

Major Components of BOM

The major components of the BOM include:

- The ‘window’ object
- The DOM (Document Object Model)
- Arrays
- The ‘navigator’ object
- The ‘history’ object
- The ‘location’ object
- The ‘screen’ object

You will learn about the above-mentioned components of the BOM in the coming chapter, except for the DOM and arrays about which you probably have learned already if you have been reading attentively!

Chapter 8: The window object

What is the ‘window’ object?

You must have read about the BOM in the last chapter, which is the collection of all the objects of the current window. The ‘window’ object is on the top of the hierarchy of the BOM. In other words, it is the root of the BOM object tree.

As the name suggests, this object represents the current window. All major parts of the BOM are the direct children of this object. For example, window.document (The DOM), window.history, etc. Note that each tab on the browser has a unique window object. They don’t share the same object! Some properties like window size, which is the same in all tabs, have the same value and technically those properties are shared. The window object contains references to useful properties and functions that may not strictly be related to the window only.

Even though there is no strict standard for the ‘window’ object, it is supported by all browsers.

The default references to the ‘window’ object

Since the ‘window’ object is present on top of hierarchy with no other object present at its level, all references to the window object’s methods and properties can be made without writing the starting of the dotted notation part, i.e., the ‘window.’ part. Example:

```
window.alert()
alert()                // same as above

var x = window.length;
var y = length  ;
// x will be equal to y since the refer to
the same property
```

All variables declared are actually ‘window’ object’s properties

As stated above, even the variables declared in the program are the direct child of the ‘window’ object, and are its properties.

Example:

```
var x = 21;
console.log( window.x ); // prints ‘21’ on the console screen
```

The ‘window’ object’s method references

Here is the list of the ‘window’ object’s major methods and their description:

- .alert("message")

  This method is used to display a dialog box on the screen, a type of pop-up, with a message that is passed as the argument of this function. The dialog box this will open has only one button, which is “OK”

  Example:

  ```
  window.alert( " Hello from JavaScript! " )
  ```

  OR

```
alert( " Hello from JavaScript! " )
```

Both of the above statements are equivalent.

- .confirm("message")

  This method is used to display a dialog box on the screen, with a message that is passed as the argument of this function, along with two buttons which are “OK” and “Cancel”. This method returns a boolean value. It returns true if the “OK” button was clicked and false if the “Cancel” button was clicked.

  Example:

```
var val = confirm( " Do you accept our terms and agreement? " );
if( val == true) console.log("User pressed OK");
else console.log("User pressed Cancel");
```

- .prompt("message", "default text")

  This method is used to display a dialog box on the screen, with a message and an input box where the user can enter a value, along with the “OK” and

"Cancel" button. This method is used to take an input from the user. The 'default text' parameter is the default value of the input box in the dialog box. This method returns the value entered by the user in the input box if the user presses the "OK" button. If the user presses the "Cancel" button, this method returns 'null'.

Example:

```
var val = prompt("What is your name?", "Enter your name here" );
if( val === null) console.log("User pressed the cancel button");
else console.log("User's name is " + val);
```

- .open([URL,] [name,] [specs,] [replace])

  This method is used to open a new window. All the parameters in this method are optional. This method returns the window object of the newly created window. Here is the description of the parameters:

  ➔ URL

  This parameter is used to specify the URL of the page to be opened in the new window.

The default value of this parameter is 'about:blank' which opens a window with a blank page.

➔ name

This parameter is used to specify how the URL is opened in the window or the name of the window. The following values are used as this parameter:

- **'_blank'**: This is the default value of the 'name' parameter. If this value is used, the URL is loaded in a new window.
- **'_parent'**: If this value is used, the URL is loaded in the parent frame.
- **'_top'**: If this value is used, the URL replaces any of the framesets that have been loaded previously.
- **'_self'**: If this value is used, the URL replaces the current page and is loaded in the current page.
- **Any other value**: If any other value is used, it acts as the name of the window. Note that the name of the

window is not the same as the title of the window.

- ➔ specs

  This parameter is a string which contains some attributes, separated by a comma, of the new window to be opened. You can specify things such as width, height etc. of the new window using this parameter.

- ➔ replace

  This parameter specifies whether the new URL to be opened will replace the current URL in the history object list or not. It takes boolean values, true or false. If 'true' is passed as the value of this parameter, the new URL replaces the current URL in the history object list. If 'false' is passed, it does not.

Example:

```
var newWindow =
window.open("http://mySite.com", "",
"height=210,width=700"); // creates a
new window with 210px height and 700px
width.
```

- .close()
  This method is used to close a window.

  Example:

```
var newWindow =
window.open("http://mySite.com");
newWindow.close(); // closes the
window as soon as it opens
```

- .scrollTo(xCoords, yCoords)
  This method is used to set the scroll of the window to the specified coordinates in the document.

- .resizeTo(height, width)
  This method is used to resize the window to a specified height and width. The height and width passed as the parameters in this argument are in pixels(px).

There are other important functions such as .setInterval(), .setTimeout() etc. which are related to timers in JavaScript, and will be explained in coming chapters.

The ‘window’ object’s properties

Here is the list of the ‘window’ object’s major properties and their description:

- .pageXOffset

  This property returns the current horizontal scroll distance in pixels(px). It is basically the **horizontal** distance between the actual left corner of the page and the current window left corner.

- .scrollX

  Same as the window.pageXOffset property.

- .pageYOffset

  This property returns the current vertical scroll distance in pixels(px). It is basically the **vertical** distance between the actual left corner of the page and the current window left corner.

- .scrollY

  Same as the window.pageYOffset property.

- .outerHeight

  This property returns the full height of the window, including the document, the toolbar, and the scrollbar.

- .outerWidth

  This property returns the full width of the window, including the document, the toolbar, and the scrollbar.

- .innerHeight

  This property returns the height of the content area of the window where the HTML document is displayed. It does **not** include the scrollbar or toolbar height.

- .innerWidth

  This property returns the width of the content area of the window where the HTML document is displayed. It does **not** include the scrollbar or toolbar height.

- .frames

  Returns an array of all the <iframe> element's in the current window if any.

  Example:

```
<body>
      <iframe
src="https://mySite1.com"></iframe>
      <iframe
src="https://mySite2.com"></iframe>
      <script>

            console.log(
            window.frames.length ); //
            prints '2' on the console
            screen

      </script>
</body>
```

- .closed

  Returns a boolean value indicating whether a window has been closed or not. If the returned value

is true, the window has been closed. If false, the window has not been closed.

Example:

```
var newWindow =
window.open("http://mySite.com");

function isNewWindowOpen()
{
      if(newWindow.closed ==
true)return "NO";
      else return "YES";
}
```

There are other important properties such as .history, .navigator etc., which will be explained in the coming chapters.

Chapter 9: Pop-up boxes in JavaScript

The pop-up boxes are an important way by which a message can be shown on the webpage. You may have seen on some websites where entering the wrong password will generate a pop-up window that says, “You have entered a wrong password”. You might have seen a dialog box appear telling you that you need to confirm something. No matter what site you visit, these pop-up boxes are implemented one way or another. Even pop-up ads are a type of pop-up box. Let us understand how JavaScript implements the concept of pop-up boxes.

There are three important types of dialog box in JavaScript – alert box, confirm box and prompt box. Each of them has different functionalities. Let us discuss them in detail.

1. Alert dialog box

As the name suggests, an alert box in most cases is used to alert the user to something important.

This can be a warning to say that the password was incorrect. Alert boxes are also used to validate the fields in a form. If one of the fields in the form has the wrong input, an alert can be shown on the screen detailing the same. It can also be used to show a normal message on the screen. There is only a single button “OK” present in the alert box, which is used to confirm the reading of the message. For example:

```
var age =17;

if (age>=18)

   alert (“You are eligible to vote. Go make your vote count!”);

else

   alert (“You are not eligible to vote!!”);
```

In the above example, since the age is less than 18, an alert box with message “You are not eligible to vote!!” will pop up. You can also bind the alert

box to a button which will make the dialog box appear when clicked.

2. Confirm dialog box

Let us assume a situation where an important file is present on the webpage. Next to that file, there is a button to “delete”, and you accidentally clicked it. If the web page has no option to confirm that action, that important file will get deleted. Therefore, there should be a mechanism which lets the user confirm some critical actions before performing them. That is where the confirm dialog box come into the picture. A confirmation dialog box contains two buttons- **OK** and **Cancel.** If the user clicks the OK button, the confirm() method will return “true”, which means the user wishes to proceed. If the user clicks the “cancel” button, the confirm() method returns “false” which indicates the user’s refusal to continue further. Let us look at an example:

```
varuserChoice= confirm(“Are you sure you want to delete the file?”);

if (userChoice==true)

  alert (“You chose to delete the file!”);

else

  alert (“You chose to skip the deleting process!”);
```

In the above example, depending on the user’s action on the confirm dialog box, the appropriate message will be displayed in the alert box. If the user selects the OK option, an alert will pop up with the message “You chose to delete the file!”. If he/she clicks on the CANCEL option, an alert will pop up with the message “You chose to skip the deleting process!” In either case, the dialog will close after the user has performed the operation.

3. Prompt dialog box

The prompt dialog box is useful when an input has to be taken from the user. This pop-up box in a way allows your web page to interact with the user. This

has various applications. Most of the websites ask for your name so that it can refer to you with the same. In those cases, prompt dialog boxes are used.

A prompt dialog box contains a field which the user has to fill out, and two buttons: OK to confirm the input, and Cancel to cancel the operation. The prompt() method is used to display the prompt dialog box. This method takes two parameters – a label which is to be displayed in the text box, and a default string which acts as a placeholder in the input box. If the user clicks the Ok button, the prompt() method will return the value entered in the text field by the user. In case the user wishes to cancel the operation for some reason, the prompt() method returns null. Let us look at an example to understand it better.

```
varuserInput= prompt(“What is your name?”, “Enter your name here”);
if (userInput==null || userInput== “”)
     alert (“You did not enter any name!”);
else
     alert (“Welcome to our website” + userInput);
```

In the above example, a prompt dialog box is displayed which asks the user to enter his/her name. The text field also has a placeholder saying “Enter your name here”. In case the user leaves it blank or cancels it, then an alert box is shown to the user with the message “You did not enter any name”. If the user enters the name “Keith” and clicks OK, an alert box with message “Welcome to our website Keith” is displayed.

All three dialog boxes mentioned above can be used as per the need. Make use of them whenever your website requires their functionality. These can turn a normal website into a dynamic, interactive website.

Chapter 10: Timers in JavaScript

As the name “timers” suggests, timers in JavaScript are used to run a task after a specific period of time. For example, say you have a website that displays minutes of the hour, and updates it as the minutes change. With a timer, you can schedule an event that runs every minute and updates the text on the website.

Till now, you have only seen the synchronous JavaScript. If you run a task synchronously, you can move on to the next task only when the synchronous task finishes executing. Timers in JavaScript demonstrate its asynchronous ability. When a task is executed asynchronously, the execution control can move onto next task without waiting for the current task to finish.

Discussed below are the various methods related to timers in JavaScript.

The setTimeOut() method

This method is basically used to execute a particular function after a specified amount of time.

Syntax:

```
setTimeOut(function, delay, [,arg1, arg2...])
```

Here are the details of each parameter:

- function

  This parameter contains the function or the function name that is to be executed after the specified amount of times.

- delay

  This parameter contains the delay, in milliseconds (1 second equals 1000 milliseconds), after which the function specified in the first parameter is to be executed.

- [,arg1, arg2…]

  These are the optional parameters, which are in turn passed as parameter(s) to the function that is executed after the specified delay. You can specify

as many parameters as you want to pass to the function to be executed, separated by a comma.

This function returns a timer ID, which can be used to stop the timer later if needed.

Examples:

```
<body>
<button onclick='buttonClick'> Click Me !
</button>

<script>
delayedFunction() {
     alert(' Hello from the other side ');
}
function buttonClick() {
     setTimeOut(delayedFunction, 5000);
}

</script>
</body>
```

In the above example, after 5 seconds (5000 milliseconds) of the button with the text “Click Me!” being clicked, we see an alert dialog box with the text “Hello” from the other side'.

Let us see an example demonstrating the use of the additional parameters in the setTimeOut() method:

```
<body>
<button onclick='buttonClick'> Click Me !
</button>

<script>
delayedSum(x, y) {
     alert(' Sum is ' + (x + y) );
}
function buttonClick() {
     setTimeOut(delayedSum, 5000, 14, 7);
}

</script>
</body>
```

In the above example, after 5 seconds (5000 milliseconds) of the button with the text “Click Me!” being clicked, we see an alert dialog box with the text “Sum is **21**”.

The clearTimeout() method

This method is used to stop a timer that was started using the setTimeOut() method. This method works as long as the function that is to be executed after the specified delay has not executed yet. If the function has already been executed, this method has no effect.

Syntax:

```
clearTimeout( timerID )
```

You remember that the setTimeOut() method returns a timer ID? We use that returned ID in this function in case you want to stop the timer from executing the function. Note that stopping the timer using this method won’t alter the original timer ID that you obtained from the setTimeOut() method, and stored in a variable.

Example:

```
<body>
```

```
<button onclick='buttonClick'> Start Timer!
</button>
<button onclick='buttonStop'> Stop </button>
<script>

var timerID = -1; // our variable to store
the timer ID

delayedFunction() {
     alert( 'Hello!' );
}
function buttonClick() {
     timerID  =  setTimeOut(delayedFunction,
5000);
}

function buttonStop() {
     clearTimeout( timerID ); // stop the
timer
}

</script>
```

```
</body>
```

In the above example, when you click the button with the text “Start Timer!”, the function buttonClick() is scheduled to run after 5 seconds. If within 5 seconds, the button with the text “Stop” is clicked, the function buttonClick() will never be executed. If the “Stop” button is clicked after 5 seconds, it will have no effect.

The setInterval() method

This method is used to execute a function repeatedly after every interval of the specified time has passed. With this method, you can make a function execute every minute, every 5 seconds, or whatever time interval suits your needs.

Syntax:

```
setInterval(function, delay, [,arg1,
arg2...])
```

Here are the details of each parameter:

- function

This parameter contains the function or the function name that is to be executed again and again after every interval of specified delay.

- delay
  This parameter contains the time delay, in milliseconds(1 second equals 1000 milliseconds), after which every interval of the function specified as the first parameter is executed.

- [,arg1, arg2…]
  These are the optional parameters which are passed as the parameter to the function that is executed after every interval of specified time. You can specify as many parameters as you want to pass to the function to be executed, separated by comma(s).

This function returns a timer ID, which can be used to stop the timer later if needed.

Example:

```
<body>
<button onclick='buttonClick'> Click Me !
</button>
```

```
<script>
delayedFunction() {
     alert(' Hello from the other side ');
}
function buttonClick() {
     setInterval(delayedFunction, 5000);
}

</script>
</body>
```

In the above example, after 5 seconds (5000 milliseconds) of the button with text “Click Me!” being clicked, we see an alert dialog box with the text “Hello from the other side”. We see the same alert again after 5 seconds. We will see that dialog box every 5 seconds until the timer is stopped.

The clearInterval() method

This method is used to stop a timer that was started using the setInterval() method.

Syntax:

```
clearInterval( timerID )
```

After using this method, the function that is to be repeatedly executed after each fixed interval of time won't be executed anymore.

Example:

```
<body>
<button onclick='buttonClick'> Start Timer!
</button>
<button onclick='buttonStop'> Stop </button>
<script>

var timerID = -1; // our variable to store
the timer ID

delayedFunction() {
     alert( 'Hello!' );
}
function buttonClick() {
     timerID = setInterval(delayedFunction,
2000);
}
```

```
function buttonStop() {
     clearInterval( timerID ); // stop the
timer
}

</script>
</body>
```

In the above example, when you click on the button with the text ‘Start Timer!’ you will start seeing an alert dialog box pop-up after every 2 seconds. If you click on the button with the text ‘Stop’, those dialog boxes will stop appearing.

The setImmediate() method

This method has the same function as using `setTimeout(fucntionname, 0),` which leads to execution of the function occurring almost directly after this method is used, but in an **asynchronous** manner.

This function is used to break a heavy and a long-running task into smaller parts. When one operation is running, the

UI of the web application becomes completely unresponsive. Usually, tasks execute so quickly that you don't notice the unresponsiveness of the UI. When long-running task execute however, you will notice it. Dividing that heavy task into parts and making it asynchronous leads to UI being responsive, along with the task being completed. This function can also be used for real-time tracking of the user's mouse.

Syntax:

```
setImmediate(function, [,arg1, arg2...])
```

Note that this method doesn't contain the 'delay' parameter as in other timer related methods. This is because the function passed as the first parameter of this argument is set to be executed the very next moment.

Example usage scenario:

Let's say you have a very long loop, such as:

```
for (var i = 0; i < 99999999; i++) {
     // doing some task here
}
```

While this loop is running, nothing else will run. The web application's frontend (the UI) will become unresponsive during this process. In order to solve this problem, we will use the setImmediate() method. Here is the converted version of the loop:

```
function longLoop(i) {
      // doing some task here

      if( i < 99999999)
setImmediate(longLoop, i + 1); //
increasing value of i and calling function
again to create a loop
}

setImmediate(longLoop, 0); // starting the
loop
```

Running the above code won't cause the UI to go unresponsive, as the call to each iteration of the loop is asynchronous, thereby allowing other functions to be executed too.

Timers are very important, and a very useful part of JavaScript. They are used extensively in modern web application or websites.

Chapter 11: The window.screen object

What is a window.screen object?

The window.screen object is one of the most important parts of the BOM. This object helps us to find information about the screen on which the browser is being displayed. As you have learned, we do not need to write the '**window.**' of the dot-notation method when referencing a child of the window object, so both of the following ways are correct for referencing to this object:

```
var screenObject = window.screen;      //
correct
```

```
var screenObject = screen;             //
also correct
```

It is to be noted that there are no standards for this particular object, nonetheless, they are implemented in all major browsers and in the same fashion.

The properties of the window.screen object

All the properties of this object are read-only properties, i.e., you cannot set a custom value for the properties of this object. Here is a list of all properties and their description of the window.screen object:

- .pixelDepth

  This property returns the pixel depth of the user's screen. Pixel depth is basically the color resolution in bpp (bits per pixel). The more color resolution exists, the more tones of color it can display.

  Syntax for using this property:

  ```
  var colorReso =
  window.screen.pixelDepth
  OR
  var colorReso = screen.pixelDepth
  ```

- .colorDepth

  This is same as the screen.pixelDepth in the modern browsers. The difference between these two can be noticed on the older UNIX machines which allowed the application to define their own color scheme. In

that case, this property returned the color resolution of the custom color scheme, whereas .pixelDepth returned the actual color resolution of the screen. In modern browsers, an application cannot define its custom color scheme. It is therefore safe to assume that the value of these two properties will always be the same.

Syntax for using this property:

```
var colorReso = window.screen.colorDepth
OR
var colorReso = screen.colorDepth
```

- .height

  This property returns the total height of the user's screen in pixels (px). Note that it returns the height of the screen, not of the browser!

  Syntax for using this property:

```
var screenHeight = window.screen.height
OR
var screenHeight = screen.height
```

- .width

  This property returns the total width of the user's screen in pixels (px). Note that it returns the width of the screen not of the browser!

  Syntax for using this property:

  ```
  var screenWidth = window.screen.width
  ```

  OR

  ```
  var screenWidth = screen.width
  ```

- .availHeight

  This property returns the available or max height for the browser that can be achieved, in pixels(px). This height is calculated by subtracting the height of taskbar from the total height of the screen. If the taskbar is not at top or bottom of the screen, the value of this property is equal to the value of the screen.height property.

  Syntax for using this property:

  ```
  var availableHeight =
  window.screen.availHeight
  ```

  OR

  ```
  var availableHeight =
  screen.availHeight
  ```

- .availWidth

  This property returns the available or max width for the browser that can be achieved, in pixels(px). This width is calculated by subtracting the width of taskbar from the total width of the screen. If the taskbar is not at left or right of the screen, the value of this property is equal to the value of the screen.width property.

  Syntax for using this property:

  ```
  var availableWidth =
  window.screen.availWidth
  OR
  var availableWidth = screen.availWidth
  ```

The properties of this object can come in handy when you have two different versions of the site for two different type of screens. For example, a website that has elements with many tones of colors can't be viewed as it was designed to be seen on a screen with a lower color resolution. You can show a site with a different color scheme based on what the color resolution of the user's screen is.

Chapter 12: The window.history object

What is the window.history object?

The window.history object is also one of the most important parts of the BOM. This object essentially helps us to manipulate the current history session of the current tab in which this object is being used. That means if the same web page is opened in the same browser – but in different tabs at the same time – their window.history object will show different results.

Just like the window.screen object, this object also has no standard. However, it is implemented by all the major browsers in a similar fashion, and is also a read-only object in the versions of HTML older than 5. In HTML version 5, this property is not just a read-only property. It also contains some methods that can be used to modify history.

Note that for security reasons, you can only navigate through the current history session. You cannot get the URL's of the page that user visited in the current session!

The properties of the window.history object

This object has only one property which is '**.length**', so the word 'properties' in the title is technically incorrect. It is in plural form. This property returns the number of entries in the current history session of the current tab, i.e., total number of pages visited by the user in the current session of the current tab.

Example:

```
console.log( ' The number of pages you visited in the current session of the current tab is ' + window.history.length ); // using just 'history.length' will work too
```

The window.history object's methods

Here is the list of all the window.history object's methods along with their description:

- .back()

  This method is used to load the previous page in the current history session of the current tab. Using this

method is same as clicking the back button of the browser.

Example:

```
function onCustomBackButtonClick() {

     alert( ' You just pressed back button, the previous page in history will now be loaded! ');

     window.history.back();
}
```

- .forward()

  This method is used to load the very next page in the current history session of the current tab. Using this method is same as clicking the forward button of the browser.

  Example:

```
function onCustomBackForwardClick() {
     alert( ' You just pressed forward button, the next page in history will now be loaded! ');
```

```
    window.history.forward();
}
```

- .go(relativeIndex|URL)
  This method is used to load a specific page from the current history session of the current tab, using either the relative index or a partial or full URL of that page.

  To understand the concept of the relative index, imagine a number-line where the current page is the 0 of the number line. +1 is the page that succeeds the current page. -1 is the previous page and so on with the other numbers.

  When using the URL, we give the partial or full URL of the page we want to load, and the first page in the history that matches the URL that we passed as the parameter is loaded.

  Using history.go(-1) is same as using history.back() or clicking on the back button of the browser. Similarly, using history.go(1) is same as using

history.forward or clicking on the forward button of the browser.

Examples:

```
window.history.go( -4 ); // goes back by 4 pages in the current history session
```

```
window.history.go( 3 ); // goes forward by 3 pages in the current history session
```

The properties and the methods that have been described till now show the read-only nature of the history object, but the methods described below show that history object can be modified. The methods described below were added in HTML version 5, and won't work in previous versions of HTML.

- .pushState(stateObject, title[, path|URL])
  This method is used to create a new entry in the current history session of the current tab.

This method replaces the current URL with the path or URL passed as the third parameter of this function, and makes a new entry with that URL in the current session history. Please note that it only replaces the current URL. This method doesn't loads that URL or path, neither checks whether it even exists or not. The third parameter is optional. Its default value is the current URL. If the URL passed as this parameter is not of the current origin, i.e., has a different domain, then this method throws an exception. The method also replaces the current title with the second parameter of this method. Note that in Firefox the second parameter is ignored, so the title remains unchanged.

Whenever a new entry is created by this method, the current history session list is loaded via the history navigation options. Like using the forward or back button of the browser, using the history.back() or the history.forward() or the history.go() method, the window.onpopstate event is emitted along the with

‘**stateObject**’ parameter, the first parameter of this method.

Understanding this method will be easier with the help of an example:

```
history.pushState({info:'hey'},   'New
title', '/newPath');

window.onpopstate = function(event) {
     var          objPassed          =
event.originalEvent.state || null; //
this line sets object value to null if
it was not passed

     if(objPassed) { // if object is
passed
          alert('The  info  is  '  +
objPassed.info);
     }
};
```

Say the current URL is **mysite.com/somePath**. Running the above line of code will replace the current URL with **mysite.com/newPath,** and the current title with 'New title', but it won't really reload or load that page. The page remains same.

Now let's say user went to some other site from here, and then clicked on the 'Back' button/ The URL **mysite.com/newPath** will be loaded, but the actual path being loaded will be **mysite.com/somePath.** When this path loads, the window.onpopstate event will be called with the object that was passed to the .pushState() method earlier (which can be accessed by using event.originalEvent.state). You will see an alert dialog box with the text '**The info is hey**'. If the 'Back' button is clicked again, the URL **mysite.com/somePath** will be loaded.

Note that if the user tries to load the URL made by the .pushState() method in the history list by trying to click on the URL directly by selecting it from the history list, or tries to load it after browser has been

closed and reopened, then the browser will try to load the URL directly. It may not exist.

- .replace(stateObject, title[, path|URL])
  This method is exactly like the .pushState() method but instead of creating a new entry in the history list, this method replaces the current URL.

  If this method was used in place of the .pushState() method in the example given in the .pushState() method description, the whole code will work exactly as described there EXCEPT for the last sentence. It was detailed that when you clicked on the 'Back' button the **mysite.com/somePath** will be loaded again, but in this case it won't be. That URL will have been replaced by the **mysite.com/newPath** entry.

This object is really helpful when creating custom back or forward options, or for skipping to a specific point in the history. The last two methods described can be used in a dynamic web application, where the URL changes when a

major aspect of the application changes, perhaps due to some user action.

Chapter 13: The window.location object

What is the window.location object?

The window.location is yet another important part of the BOM. This object contains methods and properties that are related to the current URL of the page on which the code is running. This object helps us to manipulate and perform tasks based on the current URL.

Just like the window.screen object, this object also has no specific standards set.It is implemented in all of the major browsers in a similar fashion.

The properties of the window.location object

Here is the list of all the window.location object's properties along with their description:

- hash:

This property is used to get or set the anchor part of the current page's URL. The anchor in the URL is used to point to a specific element of the page that the URL points to. The anchor part of the URL is always specified at the end of the URL, and is preceded by a hash(#) symbol.The syntax is the property name 'hash', followed by the element's ID the user desires to point to. For example, in the URL 'www.mysite.com/somepage.html**#myElement**' the '**#myElement**' is the anchor section of the URL.

Whenever you set the anchor of the URL, the scroll level of the current tab changes to that element's position in the current page so that the element in the anchor part is visible.

Example to get to the anchor part using the 'hash' property:

```
// assume that the current URL is
//
www.mysite.com/somepage.html#element1
```

```
console.log( location.hash ); // prints ‘#element1’ on the console screen
```

Example to set the anchor part:

```
<body>
<button onclick='changeAnchor'> Click Me </button>

<div style='height:1000px'></div> <!-- This div is just for this example’s sake so that you can see the change in scroll level when you click on the button -->

<div id='myElement'> Hey there! </div>

<script>
function changeAnchor() {
     location.hash = 'myElement'; // changing anchor, will also change scroll level
}
```

```
</script>
</body>
```

In the above example, when you click on the 'Click me' button, the scroll level of the current tab is automatically set to the element ID 'myElement' position. You will see the text 'Hey there!' at the top of the current tab's content viewing area. We don't use the hash(#) symbol while setting the anchor through this property.

- protocol:
  This property is used to set or get the protocol of the current URL. The protocol specifies how the selected resource in the URL is transferred. The protocol is the very first thing you see in the URL, and is followed by a colon(:)'. For example, in the URL **https:**//www.mysite.com/somepage.html the '**https:**' part is the protocol of the URL.
  Syntax for usage of this property:
  ```
  var value = location.property; // get
  location.protocol = 'someProtocol:';
  // set
  ```

Unlike the .hash property, here you have to include the colon(:) symbol while setting the value of this property.

- href:
  This property is used to set or get the URL of the current page. Not only can you set the URL using this method, but also set the anchor part of URL. You can also specify the URL with different protocols.

  Examples:

```
// assume that the current URL is
// http://www.mysite.com/somepage.html

console.log( location.href ); // prints
‘http://www.mysite.com/somepage.html’
on the console screen

location.href = '#myElement'; //
setting only the anchor part of the
```

```
URL, unlike .hash property you have to use the hash(#) symbol here

location.href = 'ftp://myserver.com'; // setting URL with a different protocol;
```

Note that when you use the location object directly without specifying any property or method after it using the dot notation, this object points to its .href property. In other words, location object name is an alias of the location.href property.

For example, both the lines below are equivalent:

```
location = '#myElement';
location.href = '#myElement';
```

- port:

  This property is used to set or get the port of the current URL. The port is the part of the URL specified just after the hostname of the URL, preceded by a colon(:). For example, in the URL 'http://mySite.com**:9292**/myfile.html' the '**9292**' part is the port.

This part is optional and not specified. Certain protocols have default ports. Using those protocols results in the default port being used, without being displayed in the URL. For example, the default port for the ‘http’ protocol is 80. When the port is not specified in the URL, this property may return 0 or an empty string, depending on the browser.

Syntax for usage of this property:

```
var value = location.port; // get
location.port = '443'; // set
```

- origin:

  This property is used to get or set the origin of the current URL. The origin is the protocol, along with the host of the current URL combined. For example, in the URL **‘http://mySite.com:9292**/myfile.html’ the ‘**http://mySite.com:9292**’ is the origin part of the URL.

  Syntax for usage of this property:

```
var value = location.origin; // get
location.origin                    =
'ftp://myserver.com:7777'; // set
```

- host:

  This property is used to get or set the hostname along with the port (if specified in the URL) of the current URL. For example, in the URL 'http://**mySite.com:9292**/myfile.html' the '**mySite.com:9292**' is the host part of the URL.

  Syntax for usage of this property:

  ```
  var value = location.host; // get
  location.host = 'othersite.com:2017';
  // set
  ```

- hostname:

  This property is used to get or set the hostname of the current URL. For example, in the URL 'http://**mysite.com**:9292/myfile.html' the '**mysite.com**' is the hostname part of the URL.

  Syntax for usage of this property:

  ```
  var value = location.hostname; // get
  ```

```
location.hostname = 'othersite.com';
// set
```

- pathname:

  This property is used to get or set the pathname of the current property. The pathname is the part of the URL after the origin and before the anchor (if there is any). For example, in the URL 'http://mysite.com:9292/**myfile.html**#myElement' the '**myfile.html**' is the pathname part of the URL.

  Syntax for usage of this property:

```
var value = location.pathname; // get
location.pathname =
'/some/other/path.html'; // set
```

- search:

  This property is used to set or get the query string of the URL. The query string part of URL is optional, and is the last part of the URL, defined even after the anchor. It is preceded by a question mark(?) symbol. For example, in the URL 'http://mysite.com:9292/myfile.html#myElement**?n**

ame=john' the '**?name=john**' is the query string part of the URL.

This part of the URL is basically used to pass information to backend server of the host through URL.

Syntax for usage of this property:

```
var value = location.search; // get
location.search = 'name=linus'; // set
```

The methods of the window.location object

Here is the list of all the window.location object's methods, along with their descriptions:

- .assign('NEW_URL')

  This method is used to load a new page in the current tab of the browser.

  Syntax:

  ```
  location.assign(
  'https://mysite.com/test.html' );
  ```

- .replace('NEW_URL')

  This method is also used to load a new page in the current tab of the browser. The difference between this method and the .assign() method is that this method replaces the current page with the new page in the browsing history, whereas the .assign() method creates a new entry in the browsing history of the browser.

  Syntax:

```
location.replace(
'https://mysite.com/test.html' );
```

- .reload([getFromHost])

  This method is used to reload the current page of the current tab. This method has an optional parameter which is used to specify whether to reload the current page from the cache, or get it from the host again. By default, the current page is reloaded from the cache. To load the page from the host again, you can pass the boolean value 'true' as the parameter of this method.

Syntax:

```
location.reload(); // reloads the page from the cache
location.reload(true); // gets the page again from the host
```

This completes the window.location object chapter. I hope you found it useful!

Chapter 14: The window.navigator object

What is the window.navigator object?

This object is yet another important part of the BOM. This object contains methods to get the information about user's browsers, and its different settings and capabilities. This object and its properties are read-only and cannot be modified.

This object has no specific standard, but is implemented by all the major modern browsers in a similar fashion.

The properties of the window.navigator object

Here is the list of all the properties of the window.navigator object, along with their descriptions:

- userAgent

  This property returns the user-agent string of the current browser. This string is unique for every

browser. It contains information about the browser, its version, the operating system on which the browser is running, etc.

Example:

```
var myUserAgent = navigator.userAgent;
console.log( myUserAgent ); // prints user's useragent on the console screen
```

An example output of the above code is '`User-agent header sent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.91 Safari/537.36`'

- platform

  This property returns the platform name for the browser in use, which code is running, what it was compiled for, etc. You can say it returns the name of the platform on which the browser is currently running, since a program compiled for a particular platform can be run on that platform only (unless you use an emulator to run it on another platform).

  Example:

```
console.log( navigator.platform ); // prints the platform for which the user's browser was compiled for
```

The output of the above code is '`Win32`' if the user's browser was compiled for Windows 32-bit version.

- cookieEnabled

  This object returns a boolean value, telling the user whether the cookies are enabled on the user's browser or not. If this property returns 'true' the cookies are enabled on the user's browser; 'false', they are not.

  Example:

```
if( navigator.cookieEnabled ) alert( " Cookies are enabled! " );
else alert( " Cookies are not enabled :( " );
```

- language

  This property returns the user's browser language. You can use this property to determine the preferred language of the user.

  Example:

```
console.log( navigator.language ); //
prints the user’s browser language
```

The output of the above code is ‘`en-US`’ if the user’s browser language is set to the English US version.

- languages

  This property returns an array which contains the user’s preferred languages, in order of their preference.

  Example:

```
console.log( navigator.languages ); //
prints the user’s preferred language
and in the order of user’s preference
```

  The output of the above code be ‘["en-US", "en", "ja-JP", "zh-CN"]’.

  Note that this property is not supported by Internet Explorer.

- geolocation

  This property returns an object that contains information about the device on which the browser is running.

  Example usage:

```
var userLocation = navigator.geolocation;
console.log( userLocation.coords.latitude ); // prints user’s latitude on the console screen
console.log( userLocation.coords.longitude ); // prints user’s longitude on the console screen
```

  For security reasons, when you use the .geolocation property, some browsers notify the user that the website is trying to access the location and ask user’s permission for the same. If the user denies permission, this property returns null.

- onLine

  This property returns a boolean value that tells whether the user's browser is online or not, i.e., has access to the internet or not. This property returns 'true' when user's browser is online, else it returns 'false'.

  Example:

```
if( navigator.onLine ) alert( " You are online! " );
else alert( " You are not online :( " );
```

- oscpu

  This property returns a string which tells the user the device's operating system on which the browser is running.

  Example:

```
console.log( navigator.oscpu ); // prints the operating system on which the browser is running on
```

The output of the above code is '`Windows NT x.y; Win64; x64`' if the browser is running on Windows 64bit version.

The navigator.javaEnabled() method

The window.navigator object has only one method and that is the .javaEnabled() method. This method returns a boolean value telling whether the current browser has java enabled or not. This method returns 'true' if the current browser is java enabled, otherwise it returns 'false'.

Example usage:

```
if( navigator.javaEnabled() ) alert( " Java is enabled! " );
else alert( " Java is not enabled :( " );
```

The window.navigator object can be useful when you have different versions of JavaScript code for different browsers. Since all browsers are not the same and handle code differently, it's always better to have different settings for different browsers based on their capabilities. For example, you must check whether the browser supports cookies before setting cookies using JavaScript.

Chapter 15: Developers' best practices

Learning a new language correctly is very important. One should be careful while designing a web application, as it can lead to failure even when the mistakes are very small. Let us look at some of the best practices while creating a web application using JavaScript.

1. Proper use of variables

The global variables form an important aspect of a program. However, one should be very careful in utilizing them. The use of global variables must be minimized as much as possible. This is because you might be unaware when these global variables have the potential to be overwritten in the later parts of the code. Use local variables wherever possible.

When the scope of some variables is limited to a function only, those variables must be declared as local variables.

Always put all the declarations at the beginning of the program or function. Doing so not only ensures that your

code is clean, but also make it easier to locate the intended variable. It reduces the possibility of unwanted re-declarations in your code.

```
//    Declarations    at    the    start
var name,      roll,      markSecured,
maximumMarks,              percentage;


name                          = "Deepak";
roll=                                21;
markSecured= 80;
maximumMarks=100;
percentage=
(markSecured/maximumMarks)*100;
```

It is also good practice to initialize the variables at the time of declaration. You can initialize the variables with default values. This practice is highly recommended. This way, you can track the intended data type that the variable was supposed to hold.

```
// declaring and initializing with
default values

var name = "",
    roll = 0,
    markSecured = 0,
```

```
    maximumMarks = 0,
    percentage=0;
```

Use of ‘new’ to create new objects is always frowned upon. Declaring primitive types as objects not only slows down the execution, but also produces various side effects. For example:

```
var name = "Dev"; //a string
var name = new String("Dev"); // an
object
(x === y) // false as their types are
different
```

Or

```
var x = new String("Dev ");
var y = new String("Dev ");
(x == y) // false since different
objects
```

2. Be careful with automatic type conversion

Automatic type conversion is an area which is greatly neglected. This often leads to unambiguous situations during the execution. While operating on non-similar datatypes, the result might be a datatype which may not be the originally intended one. Since JavaScript variables

are loosely typed, one must make sure that their integrity is not lost during execution. For example:

```
var name= “Dev”;
name = 56; // This replaces the old value of the variable name
```

The mathematical operations too can result in automatic type conversion. For example:

```
var a = 2 + 5;       // 7
var b = 2 + "5";     // 25
var c = "2" + 5;     // 25
var d = 2 - 5;       // -3
var e = 7 - "5";     // 2
var f = "2" - 5;     // -3
```

If you carefully observe the above examples, you will notice that variables b and c are string type. The value they store is not the added value, but the concatenated value. The same goes with other variables in the above example. Make sure the results produced by those operations are the intended one.

3. Use of == and ===

The '==' operator compares the value of two variables. In order to do so, it also converts their data types to a common data type. This might often lead to an incorrect data type situation. Two variables whose types are different might still be considered equal, simply because they have the same value. On the other hand, '===', compares not only the value, but also the TYPES of the two variables. Let us look at an example:

```
2 == "2";       // true since values
are same
1 == true;      // true since 1 is
considered true

2 === "2";      // false since types
are different (num and string)
1 === true;     // false since 1 is
num and true is a Boolean type
```

4. Undefined function parameters

If an argument is missing while calling a function, its value is set to undefined. This can create a problem, since the function might not be equipped to handle such asituation. Therefore, always assign default values to the parameters of a function.

```
function add(num1, num2)
{
    if (num2 === undefined) {
        num2 = 0;
    }
}
```

5. Add default case in switch statement

The Switch statement is an excellent replacement for repeating if statements. However, if not used properly, it might create problems. Even if you are extremely sure that all the possible cases are taken care of, it is always recommended to have a default case in the switch statement.

```
var animal = ‘cow';
switch (animal) {
case 'Cow':
case 'dog':
case 'goat':
case 'cat':
console.log('This is a pet animal');
break;
case ‘lion':
case ‘tiger’:
case ‘fox’:
console.log(“This is a wild animal”);

default:
console.log('Invalid entry');
}
```

6. Avoid the use of eval( )

The eval() function is used to run normal scripts as code. This might be necessary sometimes, but it also poses high-security risks. In cases where the string contains

malicious code, it can create havoc. Therefore, it is always a good idea to avoid this function.

7. Use comments

One of the most important traits of a good developer is to use meaningful comments in the code. This not only enhances the readability of the code, but also makes it easy for the coder to understand it. This assists greatly in maintaining the code where developer and maintainer are two different people.

Conclusion

The above-mentioned practices are important in order to be a good developer. I highly recommend using these practices while developing your website or websites.

Thank you very much for downloading JavaScript: Advanced JavaScript Coding from The Ground Up! Please be on the lookout for the next book in this series, JavaScript: Elite JavaScript Coding from The Ground Up. There is more yet left to learn to further your coding knowledge base. The journey does not stop here for us.

If you have enjoyed this book, please leave a positive review on Amazon to show your support. Your feedback is greatly appreciated!