THE BROWSER OBJECT MODEL (BOM)

In JavaScript, BOM stands for Browser Object Model. It represents a set of objects provided by the web browser to interact with browser functionality beyond just manipulating the content of a web page. It allow developers to access and control the browser window, handle user interactions, manipulate URLs, manage cookies, display alerts and dialogs, and more. ***The BOM is not standardized, and different browsers may implement it differently, which can lead to inconsistencies across browsers.***

1. **Window Object:** At the top of the hierarchy is the window object, which represents the browser window or tab. It provides properties and methods for controlling the browser window, such as resizing, navigating to URLs, and accessing the document loaded in the window.

2. **Document Object:** The document object represents the HTML document loaded in the browser window. It provides access to the content and structure of the web page, allowing manipulation of HTML elements, their attributes, and the content within them.

3. **Location Object:** The location object contains information about the current URL loaded in the browser window. It provides properties like href, hostname, pathname, etc., which allow you to read or manipulate the URL.

4. **Navigator Object:** The navigator object provides information about the browser itself, such as its name, version, and platform. It allows developers to perform feature detection to determine the capabilities of the browser.

5. **History Object:** The history object represents the browser's session history, allowing you to navigate backward and forward through the user's browsing history.

6. **Screen Object:** The screen object provides information about the user's screen, such as its dimensions and color depth. It can be used to optimize the layout or presentation of content based on the user's screen resolution.

7. **Event Object:** The BOM also includes the Event object, which is used to handle events triggered by user interactions or other actions within the browser window. Event handling is a fundamental part of JavaScript programming for creating interactive web applications.

8. **Timers:** JavaScript within the browser environment provides methods for creating timers using setTimeout() and setInterval(). These methods are part of the BOM and allow developers to execute code asynchronously after a specified delay or at regular intervals.

9. **Cookies:** The BOM includes support for working with cookies, which are small pieces of data stored on the client-side by the browser. JavaScript can read, write, and delete cookies using the document.cookie property, enabling state management and user tracking across multiple sessions.

10. **Storage:** With the introduction of HTML5, the BOM expanded to include mechanisms for client-side storage. This includes localStorage and sessionStorage, which allow web applications to store data locally within the browser, persisting even after the user navigates away from the page or closes the browser.

11. **Console Object:** The BOM provides access to the browser's console through the console object. Developers can use methods like log(), error(), warn(), etc., to output messages and debug their JavaScript code directly in the browser's developer tools.

The Window Object

The window object is a fundamental part of the Browser Object Model (BOM) in JavaScript. It **represents the browser window or tab in which the web page is loaded.** The window object serves as the global object for JavaScript code running within the browser environment.

- **Global Scope:** All global variables, functions, and objects are defined as properties and methods of the window object. This means that you can access global variables directly as properties of the window object. **e.g.**

```
<script>
  var globalVariable = "Hello";
  console.log(window.globalVariable); // Outputs: "Hello"
</script>
```

- **Browser Window Properties:** The window object provides properties that represent various attributes and features of the browser window. Some common properties include **window.innerWidth** and **window.innerHeight** for the inner width and height of the browser window, window.location for the URL of the current page, **window.document** for the Document object representing the loaded web page, and **window.navigator** for information about the browser.

    - **window.innerHeight and window.innerWidth** are properties of the window object in JavaScript that provide information about the dimensions of the browser's viewport, which is the visible area of the web page within the browser window or tab.
      e.g.

```
<html>
  <body>
    <p id="demo"></p>
    <button onclick="fun()">Hit</button>
    <script>
      function fun() {
        document.getElementById("demo").innerHTML =
          "Browser inner window width: " +
          window.innerWidth +
          "px<br>" +
          "Browser inner window height: " +
          window.innerHeight +
          "px";
      }
    </script>
  </body>
</html>
```

- **Browser Window Methods:** The window object also provides methods for interacting with the browser window. These methods include **window.open()** to open a new browser window or tab, **window.close()** to close the current window, **window.alert()** to display an alert dialog, **window.confirm()** to display a confirmation dialog, and **window.prompt()** to display a prompt dialog for user input.
    - **window.open():** The window.open() method is a feature of the Browser Object Model (BOM) in JavaScript that allows you to open a new browser window or tab. It provides a way to dynamically create a new browsing context, which can display a different web page or content from the current page.

      e.g.

      ```
      <!DOCTYPE html>
      <html lang="en">
      <head>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
      <title>Window Open Example</title>
      </head>
      <body>

      <button onclick="openNewWindow()">Open New Window</button>

      <script>
      function openNewWindow() {
        // Open a new window with the URL https://www.facebook.com"
        var newWindow = window.open("https://www.facebook.com",
      "facebookWindow", "width=600,height=400");

        // Check if the new window was successfully opened
        if (newWindow) {
          console.log("New window opened successfully!");
        } else {
          console.error("Failed to open new window!");
        }
      }
      </script>
      </body>
      </html>
      ```

    - **window.close():** The window.close() method in JavaScript is used to close the current browser window or tab that the script is running in. This method works only for windows that were opened via JavaScript using the window.open() method.

      Note:

      However, it's important to note that most modern browsers have restrictions on closing windows that were not explicitly opened by JavaScript. This is a security measure to prevent malicious websites from closing browser windows without the user's consent.

      e.g.

      ```
      <!DOCTYPE html>
      <html>
      <head>
        <title>Close Window Example</title>
      ```

```
</head>
<body>

<button onclick="openWindow()">Open Window</button>
<button onclick="closeWindow()">Close Window</button>

<script>
  var myWindow;

  function openWindow() {
    // Open a new window
    myWindow = window.open("", "_blank", "width=300,height=200");
  }

  function closeWindow() {
    // Close the previously opened window
    if (myWindow) {
      myWindow.close();
    } else {
      alert("Window is not opened yet!");
    }
  }
</script>

</body>
</html>
```

- **window.alert() :** The window.alert() method is a feature in JavaScript that allows you to display a dialog box, commonly known as an alert box, to the user. This dialog box typically contains a message and an "OK" button, and it is useful for presenting information or getting the user's attention.
  e.g.
  ```
  window.alert("This is an alert message!");
  ```

- **window.confirm() :** The **window.confirm()** method is a built-in function in JavaScript that displays a dialog box with a specified message and two buttons: "OK" and "Cancel". It is primarily used to prompt the user for confirmation before proceeding with an action.
  e.g.
  ```
  <script>
    var result = window.confirm("Are you sure you want to delete this item?")
    console.log(result)
    if (result) {
      // Code to execute if the user clicks "OK"
      console.log("Item deleted.");
    } else {
      // Code to execute if the user clicks "Cancel"
      console.log("Deletion canceled.");
    }
  ```

```
</script>
```

- **window.prompt()** :The window.prompt() method is a built-in function in JavaScript that displays a dialog box with a message prompting the user to input text. It typically includes a text input field and two buttons: "OK" and "Cancel". This method is commonly used to interactively request information from the user.
  e.g.
  ```
  <script>
      var userInput = window.prompt("Please enter your name:", "bhoomika");
      if (userInput !== null) {
        console.log("Hello, " + userInput + "!");
      } else {
        console.log("No name entered.");
      }
    </script>
  ```

- **Timers:** Timers in JavaScript, specifically setTimeout() and setInterval(), are mechanisms that allow you to execute code asynchronously after a certain delay or at regular intervals, respectively. These methods are indeed part of the window object in JavaScript, making them accessible globally within the browser environment.
  - **setTimeout():**
    - setTimeout() is used to execute a function or a piece of code after a specified delay, measured in **milliseconds.**
    - **It takes two parameters**: a function to execute and the delay in milliseconds.After the specified delay, the function provided as the first parameter is executed.The method returns a unique identifier (an integer value) that can be used to cancel the execution of the function using clearTimeout() if necessary.
      e.g.
      ```
      <script>
         const task=  function() {
         console.log("This message will be logged after 3 seconds.");
      }
      setTimeout(task , 3000);
      </script>
      ```

    - **setInterval()** is used to repeatedly execute a function or a piece of code at a specified interval.
      - It takes two parameters: a function to execute and the interval between executions in milliseconds.
      - The function provided as the first parameter is executed repeatedly at the specified interval until it is explicitly stopped using clearInterval().
      - It also returns a unique identifier (an integer value) that can be used to cancel the interval using clearInterval() if needed.
        e.g.
        ```
        <script>
              const task=  function() {
                console.log("This message will be logged after 3 seconds.");
        ```

```
        }
        setInterval(task , 3000);
      </script>
```

- **clearTimeout()** :The clearTimeout() function takes this identifier as an argument and cancels the execution of the corresponding timeout, preventing the associated function from being executed if the timeout hasn't already occurred.

  e.g.

```
<body>
   <button onclick="calcelTimeout()">calcel timeout </button>
   <script>
      // Setting a timeout and storing the timer identifier
      var timerID = setInterval(function () {
         console.log("This message will never be logged because clearTimeout()
is called before the timeout.");
      }, 2000);

      // Canceling the timeout before it triggers
      function calcelTimeout(){
         clearTimeout(timerID)
      }
   </script>
</body>
```

Note:

Both setTimeout() and setInterval() are commonly used in JavaScript for various purposes such as implementing animations, updating content dynamically, fetching data from servers at regular intervals, and handling asynchronous tasks.

- **Console object :** The Console object is a part of the Browser Object Model (BOM) in JavaScript, providing developers with a way to interact with the browser's developer console. This object exposes methods that allow you to log various types of messages, debug code, and inspect objects directly within the browser's developer tools.
  1. **Logging Messages**:
     - The Console object provides several methods for logging messages of different types. The most commonly used methods include:
       - **console.log():** Outputs a message to the console.
       - **console.error():** Outputs an error message to the console.
       - **console.warn():** Outputs a warning message to the console.
       - **console.info()**: Outputs an informational message to the console.
       - **console.debug():** Outputs a debug message to the console (if supported).
  2. **Interpolation:**
     - The Console object supports string interpolation, allowing you to include variables or expressions within logged messages using placeholders.

       e.g.

```
 <script>
       var name = "John";
       console.log("Hello, %s!", name);
```

```
</script>
```

- **Formatting:** The Console object allows you to format logged messages using CSS-like syntax to style the output.
  ```
  <script>
   console.log("%cStyled Message", "color: blue; font-size: 18px;");
  </script>
  ```

- **Grouping Messages:** The Console object provides methods for grouping related messages together, making it easier to organize and understand complex logs.
  e.g.
  ```
  <script>
   console.group("Group 1");
  console.log("Message 1");
  console.log("Message 2");
  console.groupEnd();
  </script>
  ```
- **Timing:** The Console object includes methods for measuring the time taken by operations using console.time() and console.timeEnd().
  e.g.
  ```
  <script>
    console.time("Timer");
    console.group("Group 1");
    console.log("Message 1");
    console.log("Message 2");
    console.groupEnd();
    console.timeEnd("Timer");
  </script>
  ```

WINDOW LOCATION

In JavaScript, the **window.location object** represents the current URL of the web page being viewed in the browser. It provides information about various parts of the URL, such as the protocol, host, pathname, search parameters, and hash fragment.

properties and methods available in the window.location object:

- **href:** A string containing the entire URL of the current page, including the protocol, host, pathname, search parameters, and hash fragment.
- **protocol:** A string containing the protocol (e.g., "http:", "https:", "file:") of the current URL.
- **host**: A string containing the host and port number of the current URL.
- **hostname:** A string containing the hostname of the current URL.
- **port:** A string containing the port number of the current URL.
- **pathname:** A string containing the pathname (the part of the URL after the host) of the current URL.
- **search:** A string containing the query parameters of the current URL, including the leading "?" character.

- **hash:** A string containing the hash fragment of the current URL, including the leading "#" character.
- **assign(url):** A method that loads a new URL in the current browsing context, replacing the current page in the history.
- **reload():** A method that reloads the current page.
- **replace(url):** A method that replaces the current URL with a new URL, without adding an entry to the browser's history.

e.g.

```
<script>
    // Accessing properties
    console.log(window.location.href);      // Entire URL
    console.log(window.location.protocol); // Protocol (e.g., "https:")
    console.log(window.location.hostname); // Hostname
    console.log(window.location.pathname); // Pathname
    console.log(window.location.search);    // Query parameters
    console.log(window.location.hash);      // Hash fragment

    // Methods
    window.location.assign('https://example.com'); // Load a new URL
    window.location.reload();                   // Reload the current page
    window.location.replace('https://example.com'); // Replace the current URL
  </script>
```

The window.location object can be written without the window prefix.

- window.location.href
  - returns the href (URL) of the current page
- window.location.hostname
  - returns the domain name of the web host
- window.location.pathname
  - returns the path and filename of the current page
- window.location.protocol
  - returns the web protocol used (http: or https:)
- window.location.assign()
  - loads a new document

e.g.

```
<!DOCTYPE html>
<html>
  <body>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
        "The full URL of this page is:<br>" + window.location +
        "<br> hostname " + window.location.hostname +
        "<br> Pathname " + window.location.pathname +
        "<br> Protocol " + window.location.protocol
    </script>
  </body>
</html>
```

Window History

In JavaScript, the window.history object provides access to the browsing history of the current window or tab. It allows developers to interact with the browser's history stack, enabling navigation between previously visited pages and controlling the browser's history behavior.

properties and methods available in the window.history object:

- **length:** A read-only property that returns the number of entries in the browsing history stack.
- **state:** A read-only property that returns an object representing the state object associated with the current history entry.
- **back():** A method that navigates the browser back one step in the history stack. This is equivalent to clicking the browser's back button.
- **forward()**: A method that navigates the browser forward one step in the history stack. This is equivalent to clicking the browser's forward button.
- **go(delta):** A method that navigates the browser to a specific step in the history stack relative to the current entry. The delta parameter specifies the number of steps to go back (negative value) or forward (positive value).
- **pushState(state, title, url):** A method that adds a new entry to the history stack with the specified state object, title, and URL. This method allows developers to manipulate the browser's history without triggering a page reload.
- **replaceState(state, title, url):** A method that modifies the current entry in the history stack with the specified state object, title, and URL. This method allows developers to update the URL and state of the current page without adding a new entry to the history stack.

  e.g.
  ```
  // Accessing properties
      console.log(window.history.length); // Number of entries in history stack
      console.log(window.history.state);  // State object of current history entry

      // Navigating history
      window.history.back();      // Go back one step in history
      window.history.forward(); // Go forward one step in history
      window.history.go(-2);      // Go back two steps in history

      // Modifying history
      window.history.pushState({ page: 1 }, "Page 1", "page1.htm "); // Add a new entry to history
      window.history.replaceState({ page: 2 }, "Page 2", "page2.html"); // Replace current entry in history
  ```

  e.g.-1
  ```
   <html>
    <body>
      <button onclick="history.back()">back</button>
  ```

```
    <button onclick="history.forward()">forward</button>
    <button onclick="history.go(2)">go</button>
    <button onclick="length()">length</button>
     console.log(window.history.state);  // State object of current history entry

        // Modifying history
        window.history.pushState({ page: 1 }, "Page 1", "page1.html");  // Add a new entry to
history
        window.history.replaceState({ page: 2 }, "Page 2", "page2.html"); // Replace current entry
in history

    <script>
        function length()
        {
            document.write("length is : " + window.history.length)
        }
    </script>
  </body>
</html>
```

Navigator Object

The Navigator object in JavaScript provides information about the web browser that is executing the script. It represents the user agent of the browser and exposes various properties and methods to query information about the browser and its capabilities.

properties and methods available in the Navigator object:

- **appName:** Returns the name of the browser. This property is deprecated and is not recommended for use in modern web development.

- **appVersion:** Returns the version of the browser.

- **platform:** Returns the platform on which the browser is running (e.g., "Win32", "MacIntel", "Linux x86_64").

- **userAgent:** Returns the user agent string for the browser, which contains information about the browser, its version, and the operating system.

- **cookieEnabled:** Returns a boolean indicating whether cookies are enabled in the browser.

- **language:** Returns the language of the browser, as specified in the browser's settings.

- **onLine:** Returns a boolean indicating whether the browser is currently online (connected to the Internet).

- **javaEnabled():** Returns a boolean indicating whether Java is enabled in the browser. This method is deprecated and is not recommended for use due to security concerns associated with Java applets.

- **geolocation:** Returns a Geolocation object, which can be used to retrieve the current geographical location of the device running the browser (if the user grants permission).

- **mediaDevices**: Returns a MediaDevices object, which can be used to access connected media devices such as cameras and microphones.

- **getBattery():** Returns a Promise that resolves to a BatteryManager object, which provides information about the device's battery status (if available).

- **serviceWorker:** Returns a ServiceWorkerContainer object, which provides access to Service Worker registration, updates, and messaging.

e.g.

```
  <script>
    document.writeln("<br/>navigator.appCodeName: " + navigator.appCodeName);
    document.writeln("<br/>navigator.appName: " + navigator.appName);
    document.writeln("<br/>navigator.appVersion: " + navigator.appVersion);
    document.writeln("<br/>navigator.cookieEnabled: " + navigator.cookieEnabled);
    document.writeln("<br/>navigator.language: " + navigator.language);
    document.writeln("<br/>navigator.userAgent: " + navigator.userAgent);
    document.writeln("<br/>navigator.platform: " + navigator.platform);
    document.writeln("<br/>navigator.onLine: " + navigator.onLine);

  </script>
```

WINDOW SCREEN

The window.screen object in JavaScript provides information about the user's screen or monitor. It represents the physical characteristics and properties of the display device, such as its dimensions, color depth, and pixel density. Developers can use the properties of the window.screen object to optimize the layout and presentation of content based on the user's screen resolution and capabilities.

properties available in the window.screen object:

- width: Returns the width of the screen in pixels.
- height: Returns the height of the screen in pixels.
- availWidth: Returns the width of the available screen space in pixels, excluding taskbars, docked windows, and other system UI elements.
- availHeight: Returns the height of the available screen space in pixels, excluding taskbars, docked windows, and other system UI elements.
- colorDepth: Returns the color depth of the screen, typically represented as the number of bits per pixel used to display colors (e.g., 24-bit color depth).
- pixelDepth: Similar to colorDepth, returns the color depth of the screen but expressed in bits per pixel.
- orientation: Returns the orientation of the screen, indicating whether it is in portrait or landscape mode.

screen.width and screen.height: These properties provide the width and height of the user's screen in pixels, respectively. They give you the total dimensions of the screen, including areas that might not be available due to operating system elements like taskbars.

- **screen.availWidth and screen.availHeight:** These properties give you the width and height of the available screen space in pixels, respectively. They take into account any areas that might be occupied by operating system elements such as taskbars, docked panels, or other persistent screen elements. Essentially, these properties provide the dimensions of the screen minus any space taken up by such elements.
- **screen.colorDepth:**screen.colorDepth represents the number of bits used to represent the color of a single pixel on the screen. It's a measure of the number of distinct colors that can be displayed. For example, a color depth of 24 bits per pixel can display up to 16.7 million colors (2^24).
- **screen.pixelDepth** is similar to screen.colorDepth and also represents the color depth of the screen, typically in bits per pixel. These two properties generally return the same value.

  e.g.

```
<html>
  <body>
    <p id="demo"></p>
    <button onclick="fun()">hit</button>
    <script>
      function fun() {
        document.getElementById("demo").innerHTML =
          "Browser screen width: " +
          window.screen.width +
          "px<br>" +
          "Browser screen height: " +
          window.screen.height +
          "px <br>" +
          "screen.availWidth " +
          screen.availWidth +
          "px <br>" +
          "screen.availHeight " +
          screen.availHeight +
          "px <br>" +
          "screen.colorDepth " +
          screen.colorDepth +
          "px<br>" +
          "screen.pixelDepth " +
          screen.pixelDepth +
          "px";
      }
    </script>
  </body>
</html>
```

Akhilesh Kumar Gupta
(Technical Training specialist)
TechCarrel LLP