JavaScript Events

Understanding Events and Event Handlers

An event is something that happens when user interact with the web page, such as when he clicked a link or button, entered text into an input box or textarea, made selection in a select box, pressed key on the keyboard, moved the mouse pointer, submits a form, etc. In some cases, the Browser itself can trigger the events, such as the page load and unload events.

When an event occur, you can use a JavaScript event handler (or an event listener) to detect them and perform specific task or set of tasks. By convention, the names for event handlers always begin with the word "on", so an event handler for the click event is called `onclick`, similarly an event handler for the load event is called `onload`, event handler for the blur event is called `onblur`, and so on.

There are several ways to assign an event handler. The simplest way is to add them directly to the start tag of the HTML elements using the special event-handler attributes. For example, to assign a click handler for a button element, we can use `onclick` attribute, like this:

Example
»

```
<button type="button" onclick="alert('Hello World!')">Click Me</button>
```

However, to keep the JavaScript seperate from HTML, you can set up the event handler in an external JavaScript file or within the `<script>` and `</script>` tags, like this:

Example
»

```
<button type="button" id="myBtn">Click Me</button>
<script>
    function sayHello() {
        alert('Hello World!');
    }
    document.getElementById("myBtn").onclick = sayHello;
```

```
</script>
```

Note: Since HTML attributes are case-insensitive so `onclick` may also be written as `onClick`, `OnClick` or `ONCLICK`. But its *value* is case-sensitive.

In general, the events can be categorized into four main groups — mouse events, keyboard events, form events and document/window events. There are many other events, we will learn about them in later chapters. The following section will give you a brief overview of the most useful events one by one along with the real life practice examples.

Mouse Events

A mouse event is triggered when the user click some element, move the mouse pointer over an element, etc. Here're some most important mouse events and their event handler.

The Click Event (onclick)

The click event occurs when a user clicks on an element on a web page. Often, these are form elements and links. You can handle a click event with an `onclick` event handler.

The following example will show you an alert message when you click on the elements.

Example
»

```
<button type="button" onclick="alert('You have clicked a
button!');">Click Me</button>
<a href="#" onclick="alert('You have clicked a
link!');">Click Me</a>
```

The Contextmenu Event (oncontextmenu)

The contextmenu event occurs when a user clicks the right mouse button on an element to open a context menu. You can handle a contextmenu event with an `oncontextmenu` event handler.

The following example will show an alert message when you right-click on the elements.

Example
»

```
<button type="button" oncontextmenu="alert('You have right-clicked a button!');">Right Click on Me</button>
<a href="#" oncontextmenu="alert('You have right-clicked a link!');">Right Click on Me</a>
```

The Mouseover Event (onmouseover)

The mouseover event occurs when a user moves the mouse pointer over an element.

You can handle the mouseover event with the `onmouseover` event handler. The following example will show you an alert message when you place mouse over the elements.

Example
»

```
<button type="button" onmouseover="alert('You have placed mouse pointer over a button!');">Place Mouse Over Me</button>
<a href="#" onmouseover="alert('You have placed mouse pointer over a link!');">Place Mouse Over Me</a>
```

The Mouseout Event (onmouseout)

The mouseout event occurs when a user moves the mouse pointer outside of an element.

You can handle the mouseout event with the `onmouseout` event handler. The following example will show you an alert message when the mouseout event occurs.

Example
»

```
<button type="button" onmouseout="alert('You have moved out of the button!');">Place Mouse Inside Me and Move Out</button>
<a href="#" onmouseout="alert('You have moved out of the link!');">Place Mouse Inside Me and Move Out</a>
```

Keyboard Events

A keyboard event is fired when the user press or release a key on the keyboard. Here're some most important keyboard events and their event handler.

The Keydown Event (onkeydown)

The keydown event occurs when the user presses down a key on the keyboard.

You can handle the keydown event with the `onkeydown` event handler. The following example will show you an alert message when the keydown event occurs.

Example
»

```
<input type="text" onkeydown="alert('You have pressed a key inside text input!')">
<textarea onkeydown="alert('You have pressed a key inside textarea!')"></textarea>
```

The Keyup Event (onkeyup)

The keyup event occurs when the user releases a key on the keyboard.

You can handle the keyup event with the `onkeyup` event handler. The following example will show you an alert message when the keyup event occurs.

Example
»

```
<input type="text" onkeyup="alert('You have released a key inside text input!')">
<textarea onkeyup="alert('You have released a key inside textarea!')"></textarea>
```

The Keypress Event (onkeypress)

The keypress event occurs when a user presses down a key on the keyboard that has a character value associated with it. For example, keys like Ctrl, Shift, Alt, Esc, Arrow keys, etc. will not generate a keypress event, but will generate a keydown and keyup event.

You can handle the keypress event with the `onkeypress` event handler. The following example will show you an alert message when the keypress event occurs.

Example
»

```
<input type="text" onkeypress="alert('You have pressed a key inside text input!')">
<textarea onkeypress="alert('You have pressed a key inside textarea!')"></textarea>
```

Form Events

A form event is fired when a form control receive or loses focus or when the user modify a form control value such as by typing text in a text input, select any option in a select box etc. Here're some most important form events and their event handler.

The Focus Event (onfocus)

The focus event occurs when the user gives focus to an element on a web page.

You can handle the focus event with the `onfocus` event handler. The following example will highlight the background of text input in yellow color when it receives the focus.

Example
»

```
<script>
    function highlightInput(elm){
        elm.style.background = "yellow";
    }
</script>
<input type="text" onfocus="highlightInput(this)">
<button type="button">Button</button>
```

Note: The value of `this` keyword inside an event handler refers to the element which has the handler on it (i.e. where the event is currently being delivered).

The Blur Event (onblur)

The blur event occurs when the user takes the focus away from a form element or a window.

You can handle the blur event with the `onblur` event handler. The following example will show you an alert message when the text input element loses focus.

Example
»

```
<input type="text" onblur="alert('Text input loses
focus!')">
<button type="button">Submit</button>
```

To take the focus away from a form element first click inside of it then press the tab key on the keyboard, give focus on something else, or click outside of it.

The Change Event (onchange)

The change event occurs when a user changes the value of a form element.

You can handle the change event with the `onchange` event handler. The following example will show you an alert message when you change the option in the select box.

Example
»

```
<select onchange="alert('You have changed the
selection!');">
    <option>Select</option>
    <option>Male</option>
    <option>Female</option>
</select>
```

The Submit Event (onsubmit)

The submit event only occurs when the user submits a form on a web page.

You can handle the submit event with the `onsubmit` event handler. The following example will show you an alert message while submitting the form to the server.

Example
»

```
<form action="action.php" method="post"
onsubmit="alert('Form data will be submitted to the
server!');">
    <label>First Name:</label>
    <input type="text" name="first-name" required>
    <input type="submit" value="Submit">
</form>
```

Document/Window Events

Events are also triggered in situations when the page has loaded or when user resize the browser window, etc. Here're some most important document/window events and their event handler.

The Load Event (onload)

The load event occurs when a web page has finished loading in the web browser.

You can handle the load event with the `onload` event handler. The following example will show you an alert message as soon as the page finishes loading.

Example
»

```
<body onload="window.alert('Page is loaded
successfully!');">
    <h1>This is a heading</h1>
    <p>This is paragraph of text.</p>
</body>
```

The Unload Event (onunload)

The unload event occurs when a user leaves the current web page.

You can handle the unload event with the `onunload` event handler. The following example will show you an alert message when you try to leave the page.

Example
»

```
<body onunload="alert('Are you sure you want to leave this
page?');">
    <h1>This is a heading</h1>
    <p>This is paragraph of text.</p>
</body>
```

The Resize Event (onresize)

The resize event occurs when a user resizes the browser window. The resize event also occurs in situations when the browser window is minimized or maximized.

You can handle the resize event with the `onresize` event handler. The following example will show you an alert message when you resize the browser window to a new width and height.

Example
»

```
<p id="result"></p>
<script>
    function displayWindowSize() {
        var w = window.outerWidth;
        var h = window.outerHeight;
        var txt = "Window size: width=" + w + ", height=" +
h;
        document.getElementById("result").innerHTML = txt;
    }
    window.onresize = displayWindowSize;
</script>
```

JavaScript DOM Nodes

Understanding the Document Object Model

The Document Object Model, or DOM for short, is a platform and language independent model to represent the HTML or XML documents. It defines the logical structure of the documents and the way in which they can be accessed and manipulated by an application program.

In the DOM, all parts of the document, such as elements, attributes, text, etc. are organized in a hierarchical tree-like structure; similar to a family tree in real life that consists of parents and children. In DOM terminology these individual parts of the document are known as *nodes*.

The Document Object Model that represents HTML document is referred to as HTML DOM. Similarly, the DOM that represents the XML document is referred to as XML DOM.

In this chapter we'll cover the HTML DOM which provides a standard interface for accessing and manipulating HTML documents through JavaScript. With the HTML DOM, you can use JavaScript to build HTML documents, navigate their hierarchical structure, and add, modify, or delete elements and attributes or their content, and so on. Almost anything found in an HTML document can be accessed, changed, deleted, or added using the JavaScript with the help of HTML DOM.

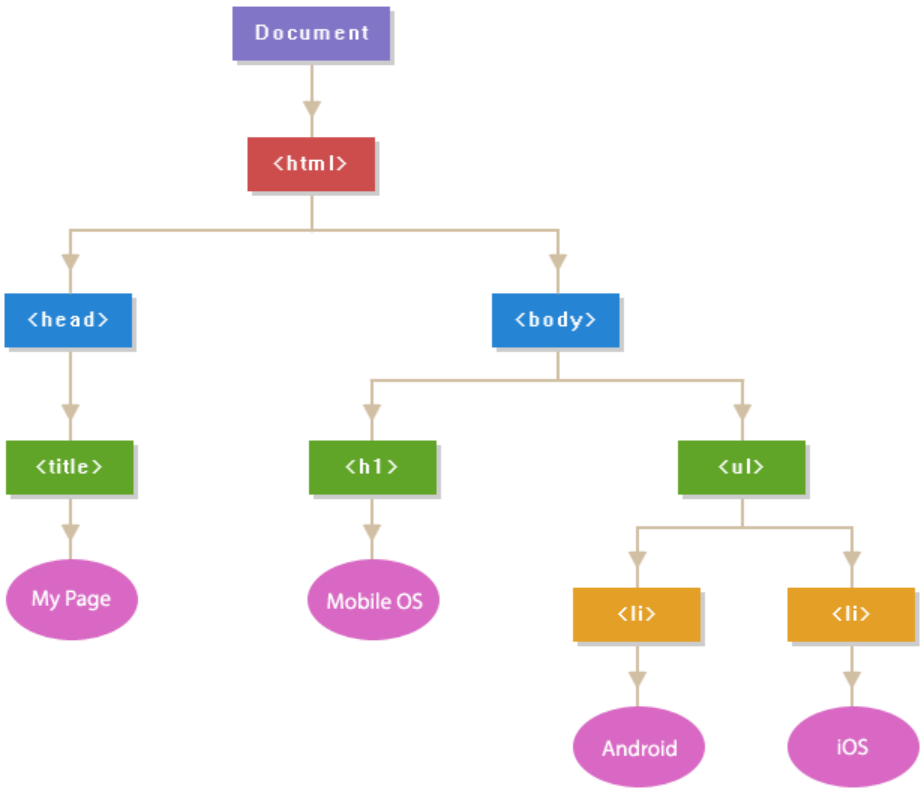
To understand this more clearly, let's consider the following simple HTML document:

Example
»

```
<!DOCTYPE html>
<html>
<head>
    <title>My Page</title>
</head>
<body>
    <h1>Mobile OS</h1>
```

```
    <ul>
        <li>Android</li>
        <li>iOS</li>
    </ul>
</body>
</html>
```

The above HTML document can be represented by the following DOM tree:



The above diagram demonstrates the parent/child relationships between the nodes. The topmost node i.e. the Document node is the root node of the DOM tree, which has one child, the `<html>` element. Whereas, the `<head>` and `<body>` elements are the child nodes of the `<html>` parent node.

The `<head>` and `<body>` elements are also siblings since they are at the same level. Further, the text content inside an element is a child node of the parent element. So, for example, "Mobile OS" is considered as a child node of the `<h1>` that contains it, and so on.

Comments inside the HTML document are nodes in the DOM tree as well, even though it doesn't affect the visual representation of the document in any way. Comments are useful for documenting the code, however, you will rarely need to retrieve and manipulate them.

HTML attributes such as `id`, `class`, `title`, `style`, etc. are also considered as nodes in DOM hierarchy but they don't participate in parent/child relationships like the other nodes do. They are accessed as properties of the element node that contains them.

Each element in an HTML document such as image, hyperlink, form, button, heading, paragraph, etc. is represented using a JavaScript object in the DOM hierarchy, and each object contains properties and methods to describe and manipulate these objects. For example, the `style` property of the DOM elements can be used to get or set the inline style of an element.

In the next few chapters we'll learn how to access individual elements on a web page and manipulate them, for example, changing their style, content, etc. using the JavaScript program.

Tip: The Document Object Model or DOM is, in fact, basically a representation of the various components of the browser and the current Web document (HTML or XML) that can be accessed or manipulated using a scripting language such as JavaScript.

JavaScript DOM Selectors

Selecting DOM Elements in JavaScript

JavaScript is most commonly used to get or modify the content or value of the HTML elements on the page, as well as to apply some effects like show, hide, animations etc. But, before you can perform any action you need to find or select the target HTML element.

In the following sections, you will see some of the common ways of selecting the elements on a page and do something with them using the JavaScript.

Selecting the Topmost Elements

The topmost elements in an HTML document are available directly as `document` properties. For example, the `<html>` element can be accessed with `document.documentElement` property, whereas the `<head>` element can be accessed with `document.head` property, and the `<body>` element can be accessed with `document.body` property. Here's an example:

Example »

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Select Topmost Elements</title>
</head>
<body>
    <script>
    // Display lang attribute value of html element
    alert(document.documentElement.getAttribute("lang"));
// Outputs: en

    // Set background color of body element
    document.body.style.background = "yellow";

    // Display tag name of the head element's first child
```

```
    alert(document.head.firstElementChild.nodeName); //
Outputs: meta
    </script>
</body>
</html>
```

But, be careful. If `document.body` is used before the `<body>` tag (e.g. inside the `<head>`), it will return `null` instead of the body element. Because the point at which the script is executed, the `<body>` tag was not parsed by the browser, so `document.body` is truly `null` at that point.

Let's take a look at the following example to better understand this:

Example »

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Document.body Demo</title>
    <script>
    alert("From HEAD: " + document.body); // Outputs: null
(since <body> is not parsed yet)
    </script>
</head>
<body>
    <script>
    alert("From BODY: " + document.body); // Outputs:
HTMLBodyElement
    </script>
</body>
</html>
```

Selecting Elements by ID

You can select an element based on its unique ID with the `getElementById()` method. This is the easiest way to find an HTML element in the DOM tree.

The following example selects and highlight an element having the ID attribute `id="mark"`.

Example
»

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Select Element by ID</title>
</head>
<body>
    <p id="mark">This is a paragraph of text.</p>
    <p>This is another paragraph of text.</p>

    <script>
    // Selecting element with id mark
    var match = document.getElementById("mark");

    // Highlighting element's background
    match.style.background = "yellow";
    </script>
</body>
</html>
```

The `getElementById()` method will return the element as an object if the matching element was found, or `null` if no matching element was found in the document.

Note: Any HTML element can have an `id` attribute. The value of this attribute must be unique within a page i.e. no two elements in the same page can have the same ID.

Selecting Elements by Class Name

Similarly, you can use the `getElementsByClassName()` method to select all the elements having specific class names. This method returns an array-like object of all child elements which have all of the given class names. Let's check out the following example:

Example
»

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Select Elements by Class Name</title>
</head>
<body>
    <p class="test">This is a paragraph of text.</p>
    <div class="block test">This is another paragraph of
text.</div>
    <p>This is one more paragraph of text.</p>

    <script>
    // Selecting elements with class test
    var matches = document.getElementsByClassName("test");

    // Displaying the selected elements count
    document.write("Number of selected elements: " +
matches.length);

    // Applying bold style to first element in selection
    matches[0].style.fontWeight = "bold";

    // Applying italic style to last element in selection
    matches[matches.length - 1].style.fontStyle = "italic";

    // Highlighting each element's background through loop
    for(var elem in matches) {
        matches[elem].style.background = "yellow";
    }
    </script>
</body>
</html>
```

Selecting Elements by Tag Name

You can also select HTML elements by tag name using the `getElementsByTagName()` method. This method also returns an array-like object of all child elements with the given tag name.

Example

»

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Select Elements by Tag Name</title>
</head>
<body>
    <p>This is a paragraph of text.</p>
    <div class="test">This is another paragraph of
text.</div>
    <p>This is one more paragraph of text.</p>

    <script>
    // Selecting all paragraph elements
    var matches = document.getElementsByTagName("p");

    // Printing the number of selected paragraphs
    document.write("Number of selected elements: " +
matches.length);

    // Highlighting each paragraph's background through
loop
    for(var elem in matches) {
        matches[elem].style.background = "yellow";
    }
    </script>
</body>
</html>
```

Selecting Elements with CSS Selectors

You can use the `querySelectorAll()` method to select elements that matches the specified CSS selector. CSS selectors provide a very powerful and efficient way of selecting HTML elements in a document.

This method returns a list of all the elements that matches the specified selectors. You can examine it just like any array, as shown in the following example:

Example

»

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Select Elements with CSS Selectors</title>
</head>
<body>
    <ul>
        <li>Bread</li>
        <li class="tick">Coffee</li>
        <li>Pineapple Cake</li>
    </ul>

    <script>
    // Selecting all li elements
    var matches = document.querySelectorAll("ul li");

    // Printing the number of selected li elements
    document.write("Number of selected elements: " +
matches.length + "<hr>")

    // Printing the content of selected li elements
    for(var elem of matches) {
        document.write(elem.innerHTML + "<br>");
    }

    // Applying line through style to first li element with
class tick
    matches = document.querySelectorAll("ul li.tick");
    matches[0].style.textDecoration = "line-through";
    </script>
</body>
</html>
```

Note: The `querySelectorAll()` method also supports CSS pseudo-classes like `:first-child`, `:last-child`, `:hover`, etc. But, for CSS pseudo-elements such as `::before`, `::after`, `::first-line`, etc. this method always returns an empty list.

JavaScript DOM Styling

Styling DOM Elements in JavaScript

You can also apply style on HTML elements to change the visual presentation of HTML documents dynamically using JavaScript. You can set almost all the styles for the elements like, fonts, colors, margins, borders, background images, text alignment, width and height, position, and so on.

In the following section we'll discuss the various methods of setting styles in JavaScript.

Setting Inline Styles on Elements

Inline styles are applied directly to the specific HTML element using the style attribute. In JavaScript the `style` property is used to get or set the inline style of an element.

The following example will set the color and font properties of an element with `id="intro"`.

Example
»

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Set Inline Styles Demo</title>
</head>
<body>
    <p id="intro">This is a paragraph.</p>
    <p>This is another paragraph.</p>

    <script>
    // Selecting element
    var elem = document.getElementById("intro");

    // Appling styles on element
    elem.style.color = "blue";
    elem.style.fontSize = "18px";
    elem.style.fontWeight = "bold";
```

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

Naming Conventions of CSS Properties in JavaScript

Many CSS properties, such as `font-size`, `background-image`, `text-decoration`, etc. contain hyphens (`-`) in their names. Since, in JavaScript hyphen is a reserved operator and it is interpreted as a minus sign, so it is not possible to write an expression, like: `elem.style.font-size`

Therefore, in JavaScript, the CSS property names that contain one or more hyphens are converted to intercapitalized style word. It is done by removing the hyphens and capitalizing the letter immediately following each hyphen, thus the CSS property `font-size` becomes the DOM property `fontSize`, `border-left-style` becomes `borderLeftStyle`, and so on.

Getting Style Information from Elements

Similarly, you get the styles applied on the HTML elements using the `style` property.

The following example will get the style information from the element having `id="intro"`.

Example »

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>JS Get Element's Style Demo</title>
</head>
<body>
    <p id="intro" style="color:red; font-size:20px;">This
is a paragraph.</p>
    <p>This is another paragraph.</p>

    <script>
```

```
    // Selecting element
    var elem = document.getElementById("intro");

    // Getting style information from element
    alert(elem.style.color);  // Outputs: red
    alert(elem.style.fontSize);  // Outputs: 20px
    alert(elem.style.fontStyle);  // Outputs nothing
    </script>
</body>
</html>
```

The `style` property isn't very useful when it comes to getting style information from the elements, because it only returns the style rules set in the element's style attribute not those that come from elsewhere, such as style rules in the embedded style sheets, or external style sheets.

To get the values of all CSS properties that are actually used to render an element you can use the `window.getComputedStyle()` method, as shown in the following example:

Example »

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JS Get Computed Style Demo</title>
<style type="text/css">
    #intro {
        font-weight: bold;
        font-style: italic;
    }
</style>
</head>
<body>
    <p id="intro" style="color:red; font-size:20px;">This
is a paragraph.</p>
    <p>This is another paragraph.</p>

    <script>
    // Selecting element
    var elem = document.getElementById("intro");

    // Getting computed style information
    var styles = window.getComputedStyle(elem);
```

```
    alert(styles.getPropertyValue("color"));  // Outputs:
rgb(255, 0, 0)
    alert(styles.getPropertyValue("font-size"));  //
Outputs: 20px
    alert(styles.getPropertyValue("font-weight"));  //
Outputs: 700
    alert(styles.getPropertyValue("font-style"));  //
Outputs: italic
    </script>
</body>
</html>
```

Tip: The value `700` for the CSS property `font-weight` is same as the keyword `bold`. The color keyword `red` is same as `rgb(255,0,0)`, which is the rgb notation of a color.

Adding CSS Classes to Elements

You can also get or set CSS classes to the HTML elements using the `className` property.

Since, `class` is a reserved word in JavaScript, so JavaScript uses the `className` property to refer the value of the HTML class attribute. The following example will show to how to add a new class, or replace all existing classes to a `<div>` element having `id="info"`.

Example »

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>JS Add or Replace CSS Classes Demo</title>
<style>
    .highlight {
        background: yellow;
    }
</style>
</head>
<body>
```

```
    <div id="info" class="disabled">Something very
important!</div>

    <script>
    // Selecting element
    var elem = document.getElementById("info");

    elem.className = "note";  // Add or replace all classes
with note class
    elem.className += " highlight";  // Add a new class
highlight
    </script>
</body>
</html>
```

There is even better way to work with CSS classes. You can use the `classList` property to get, set or remove CSS classes easily from an element. This property is supported in all major browsers except Internet Explorer prior to version 10. Here's an example:

Example »

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>JS classList Demo</title>
<style>
    .highlight {
        background: yellow;
    }
</style>
</head>
<body>
    <div id="info" class="disabled">Something very
important!</div>

    <script>
    // Selecting element
    var elem = document.getElementById("info");

    elem.classList.add("hide");  // Add a new class
    elem.classList.add("note", "highlight");  // Add
multiple classes
    elem.classList.remove("hide"); // Remove a class
```

```
    elem.classList.remove("disabled", "note"); // Remove
multiple classes
    elem.classList.toggle("visible"); // If class exists
remove it, if not add it

    // Determine if class exist
    if(elem.classList.contains("highlight")) {
        alert("The specified class exists on the
element.");
    }
    </script>
</body>
</html>
```

JavaScript DOM Get Set Attributes

Working with Attributes

The attributes are special words used inside the start tag of an HTML element to control the tag's behavior or provides additional information about the tag.

JavaScript provides several methods for adding, removing or changing an HTML element's attribute. In the following sections we will learn about these methods in detail.

Getting Element's Attribute Value

The `getAttribute()` method is used to get the current value of a attribute on the element.

If the specified attribute does not exist on the element, it will return `null`. Here's an example:

Example
»

```
<a href="https://www.google.com/" target="_blank"
id="myLink">Google</a>

<script>
    // Selecting the element by ID attribute
    var link = document.getElementById("myLink");

    // Getting the attributes values
    var href = link.getAttribute("href");
    alert(href); // Outputs: https://www.google.com/

    var target = link.getAttribute("target");
    alert(target); // Outputs: _blank
</script>
```

JavaScript provides several different ways to select elements on a page.

Setting Attributes on Elements

The `setAttribute()` method is used to set an attribute on the specified element.

If the attribute already exists on the element, the value is updated; otherwise a new attribute is added with the specified name and value. The JavaScript code in the following example will add a `class` and a `disabled` attribute to the `<button>` element.

Example »

```
<button type="button" id="myBtn">Click Me</button>

<script>
    // Selecting the element
    var btn = document.getElementById("myBtn");

    // Setting new attributes
    btn.setAttribute("class", "click-btn");
    btn.setAttribute("disabled", "");
</script>
```

Similarly, you can use the `setAttribute()` method to update or change the value of an existing attribute on an HTML element. The JavaScript code in the following example will update the value of the existing `href` attribute of an anchor (`<a>`) element.

Example »

```
<a href="#" id="myLink">Tutorial</a>

<script>
    // Selecting the element
    var link = document.getElementById("myLink");

    // Changing the href attribute value
    link.setAttribute("href", "https://www.google.com");
</script>
```

Removing Attributes from Elements

The `removeAttribute()` method is used to remove an attribute from the specified element.

The JavaScript code in the following example will remove the `href` attribute from an anchor element.

Example »

```
<a href="https://www.google.com/" id="myLink">Google</a>

<script>
    // Selecting the element
    var link = document.getElementById("myLink");

    // Removing the href attribute
    link.removeAttribute("href");
</script>
```

JavaScript DOM Manipulation

Manipulating DOM Elements in JavaScript

Now that you've learnt how to select and style HTML DOM elements. In this chapter we will learn how to add or remove DOM elements dynamically, get their contents, and so on.

Adding New Elements to DOM

You can explicitly create new element in an HTML document, using the `document.createElement()` method. This method creates a new element, but it doesn't add it to the DOM; you'll have to do that in a separate step, as shown in the following example:

Example
»

```
<div id="main">
    <h1 id="title">Hello World!</h1>
    <p id="hint">This is a simple paragraph.</p>
</div>

<script>
// Creating a new div element
var newDiv = document.createElement("div");

// Creating a text node
var newContent = document.createTextNode("Hi, how are you
doing?");

// Adding the text node to the newly created div
newDiv.appendChild(newContent);

// Adding the newly created element and its content into
the DOM
var currentDiv = document.getElementById("main");
document.body.appendChild(newDiv, currentDiv);
</script>
```

The `appendChild()` method adds the new element at the end of any other children of a specified parent node. However, if you want to add the new

element at the beginning of any other children you can use the `insertBefore()` method, as shown in example below:

Example »

```
<div id="main">
    <h1 id="title">Hello World!</h1>
    <p id="hint">This is a simple paragraph.</p>
</div>

<script>
// Creating a new div element
var newDiv = document.createElement("div");

// Creating a text node
var newContent = document.createTextNode("Hi, how are you doing?");

// Adding the text node to the newly created div
newDiv.appendChild(newContent);

// Adding the newly created element and its content into the DOM
var currentDiv = document.getElementById("main");
document.body.insertBefore(newDiv, currentDiv);
</script>
```

Getting or Setting HTML Contents to DOM

You can also get or set the contents of the HTML elements easily with the `innerHTML` property. This property sets or gets the HTML markup contained within the element i.e. content between its opening and closing tags. Checkout the following example to see how it works:

Example »

```
<div id="main">
    <h1 id="title">Hello World!</h1>
    <p id="hint">This is a simple paragraph.</p>
</div>
```

```
<script>
// Getting inner HTML conents
var contents = document.getElementById("main").innerHTML;
alert(contents); // Outputs inner html contents

// Setting inner HTML contents
var mainDiv = document.getElementById("main");
mainDiv.innerHTML = "<p>This is <em>newly inserted</em>
paragraph.</p>";
</script>
```

As you can see how easily you can insert new elements into DOM using the `innerHTML` property, but there is one problem, the `innerHTML` property replaces all existing content of an element. So if you want to insert the HTML into the document without replacing the existing contents of an element, you can use the `insertAdjacentHTML()` method.

This method accepts two parameters: the position in which to insert and the HTML text to insert. The position must be one of the following values: `"beforebegin"`, `"afterbegin"`, `"beforeend"`, and `"afterend"`. This method is supported in all major browsers.

The following example shows the visualization of position names and how it works.

Example »

```
<!-- beforebegin -->
<div id="main">
    <!-- afterbegin -->
    <h1 id="title">Hello World!</h1>
    <!-- beforeend -->
</div>
<!-- afterend -->

<script>
// Selecting target element
var mainDiv = document.getElementById("main");

// Inserting HTML just before the element itself, as a
previous sibling
mainDiv.insertAdjacentHTML('beforebegin', '<p>This is
paragraph one.</p>');
```

```
// Inserting HTML just inside the element, before its first child
mainDiv.insertAdjacentHTML('afterbegin', '<p>This is paragraph two.</p>');

// Inserting HTML just inside the element, after its last child
mainDiv.insertAdjacentHTML('beforeend', '<p>This is paragraph three.</p>');

// Inserting HTML just after the element itself, as a next sibling
mainDiv.insertAdjacentHTML('afterend', '<p>This is paragraph four.</p>');
</script>
```

Note: The `beforebegin` and `afterend` positions work only if the node is in the DOM tree and has a parent element. Also, when inserting HTML into a page, be careful not to use user input that hasn't been escaped, to prevent XSS attacks.

Removing Existing Elements from DOM

Similarly, you can use the `removeChild()` method to remove a child node from the DOM. This method also returns the removed node. Here's an example:

Example »

```
<div id="main">
    <h1 id="title">Hello World!</h1>
    <p id="hint">This is a simple paragraph.</p>
</div>

<script>
var parentElem = document.getElementById("main");
var childElem = document.getElementById("hint");
parentElem.removeChild(childElem);
</script>
```

It is also possible to remove the child element without exactly knowing the parent element. Simply find the child element and use

the `parentNode` property to find its parent element. This property returns the parent of the specified node in the DOM tree. Here's an example:

Example »

```
<div id="main">
    <h1 id="title">Hello World!</h1>
    <p id="hint">This is a simple paragraph.</p>
</div>

<script>
var childElem = document.getElementById("hint");
childElem.parentNode.removeChild(childElem);
</script>
```

Replacing Existing Elements in DOM

You can also replace an element in HTML DOM with another using the `replaceChild()` method. This method accepts two parameters: the node to insert and the node to be replaced. It has the syntax like `parentNode.replaceChild(newChild, oldChild);`. Here's an example:

Example »

```
<div id="main">
    <h1 id="title">Hello World!</h1>
    <p id="hint">This is a simple paragraph.</p>
</div>

<script>
var parentElem = document.getElementById("main");
var oldPara = document.getElementById("hint");

// Creating new elememt
var newPara = document.createElement("p");
var newContent = document.createTextNode("This is a new
paragraph.");
newPara.appendChild(newContent);

// Replacing old paragraph with newly created paragraph
```

```
parentElem.replaceChild(newPara, oldPara);
</script>
```

JavaScript DOM Navigation

Navigating Between DOM Nodes

In the previous chapters you've learnt how to select individual elements on a web page. But there are many occasions where you need to access a child, parent or ancestor element. See the JavaScript DOM nodes chapter to understand the logical relationships between the nodes in a DOM tree.

DOM node provides several properties and methods that allow you to navigate or traverse through the tree structure of the DOM and make changes very easily. In the following section we will learn how to navigate up, down, and sideways in the DOM tree using JavaScript.

Accessing the Child Nodes

You can use the `firstChild` and `lastChild` properties of the DOM node to access the first and last direct *child node* of a node, respectively. If the node doesn't have any child element, it returns `null`.

Example
»

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p>
</div>

<script>
var main = document.getElementById("main");
console.log(main.firstChild.nodeName); // Prints: #text

var hint = document.getElementById("hint");
console.log(hint.firstChild.nodeName); // Prints: SPAN
</script>
```

Note: The `nodeName` is a read-only property that returns the name of the current node as a string. For example, it returns the tag name for element node, `#text` for text node, `#comment` for comment node, `#document` for document node, and so on.

If you notice the above example, the `nodeName` of the first-child node of the main DIV element returns #text instead of H1. Because, whitespace such as spaces, tabs, newlines, etc. are valid characters and they form #text nodes and become a part of the DOM tree. Therefore, since the `<div>` tag contains a newline before the `<h1>` tag, so it will create a #text node.

To avoid the issue with `firstChild` and `lastChild` returning #text or #comment nodes, you could alternatively use the `firstElementChild` and `lastElementChild` properties to return only the first and last *element node*, respectively. But, it will not work in IE 9 and earlier.

Example »

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p>
</div>

<script>
var main = document.getElementById("main");
alert(main.firstElementChild.nodeName); // Outputs: H1
main.firstElementChild.style.color = "red";

var hint = document.getElementById("hint");
alert(hint.firstElementChild.nodeName); // Outputs: SPAN
hint.firstElementChild.style.color = "blue";
</script>
```

Similarly, you can use the `childNodes` property to access all child nodes of a given element, where the first child node is assigned index 0. Here's an example:

Example »

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p>
</div>

<script>
var main = document.getElementById("main");

// First check that the element has child nodes
```

```
if(main.hasChildNodes()) {
    var nodes = main.childNodes;

    // Loop through node list and display node name
    for(var i = 0; i < nodes.length; i++) {
        alert(nodes[i].nodeName);
    }
}
</script>
```

The `childNodes` returns all child nodes, including non-element nodes like text and comment nodes. To get a collection of only elements, use `children` property instead.

Example »

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p>
</div>

<script>
var main = document.getElementById("main");

// First check that the element has child nodes
if(main.hasChildNodes()) {
    var nodes = main.children;

    // Loop through node list and display node name
    for(var i = 0; i < nodes.length; i++) {
        alert(nodes[i].nodeName);
    }
}
</script>
```

Accessing the Parent Nodes

You can use the `parentNode` property to access the parent of the specified node in the DOM tree.

The `parentNode` will always return `null` for document node, since it doesn't have a parent.

Example
»

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p>
</div>

<script>
var hint = document.getElementById("hint");
alert(hint.parentNode.nodeName); // Outputs: DIV
alert(document.documentElement.parentNode.nodeName); //
Outputs: #document
alert(document.parentNode); // Outputs: null
</script>
```

Tip: The topmost DOM tree nodes can be accessed directly as `document` properties. For example, the `<html>` element can be accessed with `document.documentElement` property, whereas the `<head>` element can be accessed with `document.head` property, and the `<body>` element can be accessed with `document.body` property.

However, if you want to get only element nodes you can use the `parentElement`, like this:

Example
»

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p>
</div>

<script>
var hint = document.getElementById("hint");
alert(hint.parentNode.nodeName); // Outputs: DIV
hint.parentNode.style.backgroundColor = "yellow";
</script>
```

Accessing the Sibling Nodes

You can use the `previousSibling` and `nextSibling` properties to access the previous and next node in the DOM tree, respectively. Here's an example:

Example
»

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p><hr>
</div>

<script>
var title = document.getElementById("title");
alert(title.previousSibling.nodeName); // Outputs: #text

var hint = document.getElementById("hint");
alert(hint.nextSibling.nodeName); // Outputs: HR
</script>
```

Alternatively, you can use the `previousElementSibling` and `nextElementSibling` to get the previous and next sibling element skipping any whitespace text nodes. All these properties returns `null` if there is no such sibling. Here's an example:

Example
»

```
<div id="main">
    <h1 id="title">My Heading</h1>
    <p id="hint"><span>This is some text.</span></p>
</div>

<script>
var hint = document.getElementById("hint");
alert(hint.previousElementSibling.nodeName); // Outputs: H1
alert(hint.previousElementSibling.textContent); // Outputs:
My Heading

var title = document.getElementById("title");
alert(title.nextElementSibling.nodeName); // Outputs: P
alert(title.nextElementSibling.textContent); // Outputs:
This is some text.
</script>
```

The `textContent` property represents the text content of a node and all of its descendants.

Types of DOM Nodes

The DOM tree is consists of different types of nodes, such as elements, text, comments, etc.

Every node has a `nodeType` property that you can use to find out what type of node you are dealing with. The following table lists the most important node types:

Constant	Value	Description
`ELEMENT_NODE`	1	An element node such as `<p>` or `<img>`.
`TEXT_NODE`	3	The actual text of element.
`COMMENT_NODE`	8	A comment node i.e. `<!-- some comment -->`
`DOCUMENT_NODE`	9	A document node i.e. the parent of `<html>` element.
`DOCUMENT_TYPE_NODE`	10	A document type node e.g. `<!DOCTYPE html>` for HTML5 documents.

JavaScript Window

The Window Object

The `window` object represents a window containing a DOM document. A window can be the main window, a frame set or individual frame, or even a new window created with JavaScript.

If you remember from the preceding chapters we've used the `alert()` method in our scripts to show popup messages. This is a method of the `window` object.

In the next few chapters we will see a number of new methods and properties of the `window` object that enables us to do things such as prompt user for information, confirm user's action, open new windows, etc. which lets you to add more interactivity to your web pages.

Calculating Width and Height of the Window

The `window` object provides the `innerWidth` and `innerHeight` property to find out the width and height of the browser window viewport (in pixels) including the horizontal and vertical scrollbar, if rendered. Here's is an example that displays the current size of the window on button click:

Example »

```
<script>
function windowSize(){
    var w = window.innerWidth;
    var h = window.innerHeight;
    alert("Width: " + w + ", " + "Height: " + h);
}
</script>
```

```
<button type="button" onclick="windowSize();">Get Window Size</button>
```

However, if you want to find out the width and height of the window excluding the scrollbars you can use the `clientWidth` and `clientHeight` property of any DOM element (like a `div`), as follow:

Example »

```
<script>
function windowSize(){
    var w = document.documentElement.clientWidth;
    var h = document.documentElement.clientHeight;
    alert("Width: " + w + ", " + "Height: " + h);
}
</script>

<button type="button" onclick="windowSize();">Get Window Size</button>
```

Note: The `document.documentElement` object represents the root element of the document, which is the `<html>` element, whereas the `document.body` object represents the `<body>` element. Both are supported in all major browsers.

JavaScript Window Screen

The Screen Object

The `window.screen` object contains information about the user's screen such as resolution (i.e. width and height of the screen), color depth, pixel depth, etc.

Since window object is at the top of the scope chain, so properties of the `window.screen` object can be accessed without specifying the `window.` prefix, for example `window.screen.width` can be written as `screen.width`. The following section will show you how to get information of the user's display using the screen object property of the window object.

Getting Width and Height of the Screen

You can use the `screen.width` and `screen.height` property obtains the width and height of the user's screen in pixels. The following example will display your screen resolution on button click:

Example »

```
<script>
function getResolution() {
    alert("Your screen is: " + screen.width + "x" +
screen.height);
}
</script>

<button type="button" onclick="getResolution();">Get
Resolution</button>
```

Getting Available Width and Height of the Screen

The `screen.availWidth` and `screen.availHeight` property can be used to get the width and height available to the browser for its use on user's screen, in pixels.

The screen's available width and height is equal to screen's actual width and height minus width and height of interface features like the taskbar in Windows. Here's an example:

Example »

```
<script>
function getAvailSize() {
    alert("Available Screen Width: " + screen.availWidth +
", Height: " + screen.availHeight);
}
</script>

<button type="button" onclick="getAvailSize();">Get
Available Size</button>
```

Getting Screen Color Depth

You can use the `screen.colorDepth` property to get the color depth of the user's screen. Color depth is the number of bits used to represent the color of a single pixel.

Color depth indicates how many colors a device screen is capable to produce. For example, screen with color depth of 8 can produce 256 colors (2^8).

Currently, most devices has screen with color depth of 24 or 32. In simple words more bits produce more color variations, like 24 bits can produce 2^{24} = 16,777,216 color variations (*true colors*), whereas 32 bits can produce 2^{32} = 4,294,967,296 color variations (*deep colors*).

Example »

```
<script>
```

```
function getColorDepth() {
    alert("Your screen color depth is: " +
screen.colorDepth);
}
</script>

<button type="button" onclick="getColorDepth();">Get Color
Depth</button>
```

Tip: As of now virtually every computer and phone display uses 24-bit color depth. 24 bits almost always uses 8 bits of each of R, G, B. Whereas in case of 32-bit color depth, 24 bits are used for the color, and the remaining 8 bits are used for transparency.

Getting Screen Pixel Depth

You can get the pixel depth of the screen using the `screen.pixelDepth` property. Pixel depth is the number of bits used per pixel by the system display hardware.

For modern devices, color depth and pixel depth are equal. Here's an example:

Example »

```
<script>
function getPixelDepth() {
    alert("Your screen pixel depth is: " +
screen.pixelDepth);
}
</script>

<button type="button" onclick="getPixelDepth();">Get Pixel
Depth</button>
```

JavaScript Window Location

The Location Object

The location property of a window (i.e. `window.location`) is a reference to a Location object; it represents the current URL of the document being displayed in that window.

Since window object is at the top of the scope chain, so properties of the `window.location` object can be accessed without `window.` prefix, for example `window.location.href` can be written as `location.href`. The following section will show you how to get the URL of page as well as hostname, protocol, etc. using the location object property of the window object.

Getting the Current Page URL

You can use the `window.location.href` property to get the entire URL of the current page.

The following example will display the complete URL of the page on button click:

Example
»

```
<script>
function getURL() {
    alert("The URL of this page is: " +
window.location.href);
}
</script>

<button type="button" onclick="getURL();">Get Page
URL</button>
```

Getting Different Part of a URL

Similarly, you can use other properties of the location object such as `protocol`, `hostname`, `port`, `pathname`, `search`, etc. to obtain different part of the URL.

Try out the following example to see how to use the location property of a window.

Example »

```
// Prints complete URL
document.write(window.location.href);

// Prints protocol like http: or https:
document.write(window.location.protocol);

// Prints hostname with port like localhost or
localhost:3000
document.write(window.location.host);

// Prints hostname like localhost or www.example.com
document.write(window.location.hostname);

// Prints port number like 3000
document.write(window.location.port);

// Prints pathname like /products/search.php
document.write(window.location.pathname);

// Prints query string like ?q=ipad
document.write(window.location.search);

// Prints fragment identifier like #featured
document.write(window.location.hash);
```

Note: When you visit a website, you're always connecting to a specific port (e.g. http://localhost:3000). However, most browsers will simply not display the default port numbers, for example, 80 for HTTP and 443 for HTTPS.

Loading New Documents

You can use the `assign()` method of the location object i.e. `window.location.assign()` to load another resource from a URL provided as parameter, for example:

Example »

```
<script>
function loadHomePage() {
    window.location.assign("https://www.google.com");
}
</script>

<button type="button" onclick="loadHomePage();">Load Home Page</button>
```

You can also use the `replace()` method to load new document which is almost the same as `assign()`. The difference is that it doesn't create an entry in the browser's history, meaning the user won't be able to use the back button to navigate to it. Here's an example:

Example »

```
<script>
function loadHomePage(){
    window.location.replace("https://www.google.com");
}
</script>

<button type="button" onclick="loadHomePage();">Load Home Page</button>
```

Alternatively, you can use the `window.location.href` property to load new document in the window. It produce the same effect as using `assign()` method. Here's is an example:

Example »

```
<script>
function loadHomePage() {
    window.location.href = "https://www.google.com";
}
</script>
```

```
<button type="button" onclick="loadHomePage();">Load Home
Page</button>
```

Reloading the Page Dynamically

The `reload()` method can be used to reload the current page dynamically.

You can optionally specify a Boolean parameter `true` or `false`. If the parameter is `true`, the method will force the browser to reload the page from the server. If it is `false` or not specified, the browser may reload the page from its cache. Here's an example:

Example »

```
<script>
function forceReload() {
    window.location.reload(true);
}
</script>

<button type="button" onclick="forceReload();">Reload
Page</button>
```

Note: The result of calling `reload()` method is different from clicking browser's Reload/Refresh button. The `reload()` method clears form control values that otherwise might be retained after clicking the Reload/Refresh button in some browsers.

JavaScript Window History

The History Object

The history property of the Window object refers to the History object. It contains the browser session history, a list of all the pages visited in the current frame or window.

Since Window is a global object and it is at the top of the scope chain, so properties of the Window object i.e. `window.history` can be accessed without `window.` prefix, for example `window.history.length` can be written as `history.length`.

The following section will show you how to get the information of user's browsing history. However, for security reasons scripts are not allowed to access the stored URLs.

Getting the Number of Pages Visited

The `window.history.length` property can be used to get the number of pages in the session history of the browser for the current window. It also includes the currently loaded page.

You can use this property to find out how many pages a user has visited during the current browser session, as demonstrated in the following example:

Example
»

```
<script>
function getViews() {
    alert("You've accessed " + history.length + " web pages
in this session.");
}
</script>

<button type="button" onclick="getViews();">Get Views
Count</button>
```

Going Back to the Previous Page

You can use the `back()` method of the History object i.e. `history.back()` to go back to the previous page in session history. It is same as clicking the browser's back button.

Example »

```
<script>
function goBack() {
    window.history.back();
}
</script>

<button type="button" onclick="goBack();">Go Back</button>
```

If your browser back button is active then clicking this Go Back link takes you one step back.

Going Forward to the Next Page

You can use the `forward()` method of the History object i.e. `history.forward()` to go forward to the next page in session history. It is same as clicking the browser's forward button.

Example »

```
<script>
function goForward() {
    window.history.forward();
}
</script>

<button type="button" onclick="goForward();">Go
Forward</button>
```

If your browser forward button is active then clicking this Go Forward link takes you one step forward.

Going to a Specific Page

You can also load specific page from the session history using the `go()` method of the History object i.e. `history.go()`. This method takes an integer as a parameter. A negative integer moves backward in the history, and a positive integer moves forward in the history.

Example »

```
window.history.go(-2);  // Go back two pages
window.history.go(-1); // Go back one page
window.history.go(0);  // Reload the current page
window.history.go(1);  // Go forward one page
window.history.go(2);  // Go forward two pages
```

Tip: If you attempt to access the page that does not exist in the window's history then the methods `back()`, `forward()` and `go()` will simply do nothing.

JavaScript Window Navigator

The Navigator Object

The navigator property of a window (i.e. `window.navigator`) is a reference to a Navigator object; it is a read-only property which contains information about the user's browser.

Since Window is a global object and it is at the top of the scope chain, so properties of the Window object such as `window.navigator` can be accessed without `window.` prefix, for example `window.navigator.language` can be written as `navigator.language`.

The following section will show you how to get various information about user's browser.

Detect Whether the Browser is Online or Offline

You can use the `navigator.onLine` property to detect whether the browser (or, application) is online or offline. This property returns a Boolean value `true` meaning online, or `false` meaning offline.

Example
»

```
<script>
function checkConnectionStatus() {
    if(navigator.onLine) {
        alert("Application is online.");
    } else {
        alert("Application is offline.");
    }
}
</script>

<button type="button"
onclick="checkConnectionStatus();">Check Connection
Status</button>
```

Browser fires online and offline events when a connection is establish or lost. You can attach handler functions to these events in order to customize your application for online and offline scenarios.

Let's take a look at the following JavaScript code to see how this works:

Example
»

```
<script>
function goOnline() {
    // Action to be performed when your application goes online
    alert("And we're back!");
}

function goOffline() {
    // Action to be performed when your application goes offline
    alert("Hey, it looks like you're offline.");
}

// Attaching event handler for the online event
window.addEventListener("online", goOnline);

// Attaching event handler for the offline event
window.addEventListener("offline", goOffline);
</script>

<p>Toggle your internet connection on/off to see how it works.</p>
```

The `goOffline()` function in the above example will be called automatically by the browser whenever the connection goes offline, whereas the `goOnline()` function will be called automatically by the browser when the connection status changes to online.

Check Whether Cookies Are Enabled or Not

You can use the `navigator.cookieEnabled` to check whether cookies are enabled in the user's browser or not. This property returns a Boolean value `true` if cookies are enabled, or `false` if it isn't.

Example
»

```
<script>
function checkCookieEnabled() {
    if(navigator.cookieEnabled) {
        alert("Cookies are enabled in your browser.");
    } else {
        alert("Cookies are disabled in your browser.");
    }
}
</script>

<button type="button" onclick="checkCookieEnabled();">Check
If Cookies are Enabled</button>
```

Tip: You should use the `navigator.cookieEnabled` property to determine whether the cookies are enabled or not before creating or using cookies in your JavaScript code.

Detecting the Browser Language

You can use the `navigator.language` property to detect the language of the browser UI.

This property returns a string representing the language, e.g. "en", "en-US", etc.

Example
»

```
<script>
function checkLanguage() {
    alert("Your browser's UI language is: " +
navigator.language);
}
</script>

<button type="button" onclick="checkLanguage();">Check
Language</button>
```

Getting Browser Name and Version Information

The Navigator object has five main properties that provide name and version information about the user's browser. The following list provides a brief overview of these properties:

- `appName` — Returns the name of the browser. It always returns "Netscape", in any browser.
- `appVersion` — Returns the version number and other information about the browser.
- `appCodeName` — Returns the code name of the browser. It returns "Mozilla", for all browser.
- `userAgent` — Returns the user agent string for the current browser. This property typically contains all the information in both `appName` and `appVersion`.
- `platform` — Returns the platform on which browser is running (e.g. "Win32", "WebTV OS", etc.)

As you can see from the above descriptions, the value returned by these properties are misleading and unreliable, so don't use them to determine the user's browser type and version.

Example »

```
<script>
function getBrowserInformation() {
       var info = "\n App Name: " + navigator.appName;
          info += "\n App Version: " + navigator.appVersion;
          info += "\n App Code Name: " +
navigator.appCodeName;
          info += "\n User Agent: " + navigator.userAgent;
          info += "\n Platform: " + navigator.platform;

    alert("Here're the information related to your browser:
" + info);
}
</script>
```

```
<button type="button"
onclick="getBrowserInformation();">Get Browser
Information</button>
```

Check Whether the Browser is Java Enabled or Not

You can use the method `javaEnabled()` to check whether the current browser is Java-enabled or not.

This method simply indicates whether the preference that controls Java is on or off, it does not reveal whether the browser offers Java support or Java is installed on the user's system or not.

Example »

```
<script>
function checkJavaEnabled() {
    if(navigator.javaEnabled()) {
        alert("Your browser is Java enabled.");
    } else {
        alert("Your browser is not Java enabled.");
    }
}
</script>

<button type="button" onclick="checkJavaEnabled();">Check
If Java is Enabled</button>
```

JavaScript Dialog Boxes

Creating Dialog Boxes

In JavaScript you can create dialog boxes or popups to interact with the user. You can either use them to notify a user or to receive some kind of user input before proceeding.

You can create three different types of dialog boxes *alert*, *confirm*, and *prompt* boxes.

The appearance of these dialog boxes is determined by the operating system and/or browser settings, they cannot be modified with the CSS. Also, dialog boxes are modal windows; when a dialog box is displayed the code execution stops, and resumes only after it has been dismissed.

In the following section we will discuss each of these dialog boxes in detail.

Creating Alert Dialog Boxes

An alert dialog box is the most simple dialog box. It enables you to display a short message to the user. It also includes OK button, and the user has to click this OK button to continue.

You can create alert dialog boxes with the `alert()` method. You've already seen a lot of alert examples in the previous chapters. Let's take a look at one more example:

Example
»

```
var message = "Hi there! Click OK to continue.";
alert(message);

/* The following line won't execute until you dismiss
previous alert */
alert("This is another alert box.");
```

Creating Confirm Dialog Boxes

A confirm dialog box allows user to confirm or cancel an action. A confirm dialog looks similar to an alert dialog but it includes a Cancel button along with the OK button.

You can create confirm dialog boxes with the `confirm()` method. This method simply returns a Boolean value (`true` or `false`) depending on whether the user clicks OK or Cancel button. That's why its result is often assigned to a variable when it is used.

The following example will print some text in the browser depending on which button is clicked.

Example
»

```
var result = confirm("Are you sure?");

if(result) {
    document.write("You clicked OK button!");
} else {
    document.write("You clicked Cancel button!");
}
```

Creating Prompt Dialog Box

The prompt dialog box is used to prompt the user to enter information. A prompt dialog box includes a text input field, an OK and a Cancel button.

You can create prompt dialog boxes with the `prompt()` method. This method returns the text entered in the input field when the user clicks the OK button, and `null` if user clicks the Cancel button. If the user clicks OK button without entering any text, an empty string is returned. For this reason, its result is usually assigned to a variable when it is used.

The following example will print the value entered by you when you click the OK button.

Example
»

```
var name = prompt("What's your name?");

if(name.length > 0 && name != "null") {
    document.write("Hi, " + name);
} else {
    document.write("Anonymous!");
}
```

The value returned by the `prompt()` method is always a string. This means if the user enters 10 in the input field, the string "10" is returned instead of the number 10.

Therefore, if you want to use the returned value as a number you must covert it or cast to Number, like this: `var age = Number(prompt("What's your age?"));`

Tip: To display line breaks inside the dialog boxes, use newline character or line feed (\n); a backslash followed by the character n.

JavaScript Timers

Working with Timers

A timer is a function that enables us to execute a function at a particular time.

Using timers you can delay the execution of code so that it does not get done at the exact moment an event is triggered or the page is loaded. For example, you can use timers to change the advertisement banners on your website at regular intervals, or display a real-time clock, etc. There are two timer functions in JavaScript: `setTimeout()` and `setInterval()`.

The following section will show you how to create timers to delay code execution as well as how to perform one or more actions repeatedly using these functions in JavaScript.

Executing Code After a Delay

The `setTimeout()` function is used to execute a function or specified piece of code just once after a certain period of time. Its basic syntax is `setTimeout(function, milliseconds)`.

This function accepts two parameters: a *function*, which is the function to execute, and an optional *delay* parameter, which is the number of milliseconds representing the amount of time to wait before executing the function (1 second = 1000 milliseconds). Let's see how it works:

Example »

```
<script>
function myFunction() {
    alert('Hello World!');
}
</script>

<button onclick="setTimeout(myFunction, 2000)">Click Me</button>
```

The above example will display an alert message after 2 seconds on click of the button.

Note: If the *delay* parameter is omitted or not specified, a value of `0` is used, that means the specified function is executed "immediately", or, as soon as possible.

Executing Code at Regular Intervals

Similarly, you can use the `setInterval()` function to execute a function or specified piece of code repeatedly at fixed time intervals. Its basic syntax is `setInterval(function, milliseconds)`.

This function also accepts two parameters: a *function*, which is the function to execute, and *interval*, which is the number of milliseconds representing the amount of time to wait before executing the function (1 second = 1000 milliseconds). Here's an example:

Example »

```
<script>
function showTime() {
    var d = new Date();
    document.getElementById("clock").innerHTML =
d.toLocaleTimeString();
}
setInterval(showTime, 1000);
</script>

<p>The current time on your computer is: <span
id="clock"></span></p>
```

The above example will execute the `showTime()` function repeatedly after 1 second. This function retrieves the current time on your computer and displays it in the browser.

Stopping Code Execution or Cancelling a Timer

Both `setTimeout()` and `setInterval()` method return an unique ID (a positive integer value, called timer identifier) which identifies the timer created by the these methods.

This ID can be used to disable or clear the timer and stop the execution of code beforehand. Clearing a timer can be done using two functions: `clearTimeout()` and `clearInterval()`.

The `setTimeout()` function takes a single parameter, an ID, and clear a `setTimeout()` timer associated with that ID, as demonstrated in the following example:

Example »

```
<script>
var timeoutID;

function delayedAlert() {
  timeoutID = setTimeout(showAlert, 2000);
}

function showAlert() {
  alert('This is a JavaScript alert box.');
}

function clearAlert() {
  clearTimeout(timeoutID);
}
</script>

<button onclick="delayedAlert();">Show Alert After Two Seconds</button>

<button onclick="clearAlert();">Cancel Alert Before It Display</button>
```

Similarly, the `clearInterval()` method is used to clear or disable a `setInterval()` timer.

Example
»

```
<script>
var intervalID;

function showTime() {
    var d = new Date();
    document.getElementById("clock").innerHTML =
d.toLocaleTimeString();
}

function stopClock() {
    clearInterval(intervalID);
}

var intervalID = setInterval(showTime, 1000);
</script>

<p>The current time on your computer is: <span
id="clock"></span></p>

<button onclick="stopClock();">Stop Clock</button>
```

Note: You can technically use `clearTimeout()` and `clearInterval()` interchangeably. However, for clarity and code maintainability you should avoid doing so.