Learning Outcome

Learn about Event Programming

Exceptional Handling

Browser Object Model

Navigation

SettimeOut Function

TimeInterval Function

Date Functions

Ajax

JSON

# Introduction to JavaScript Event

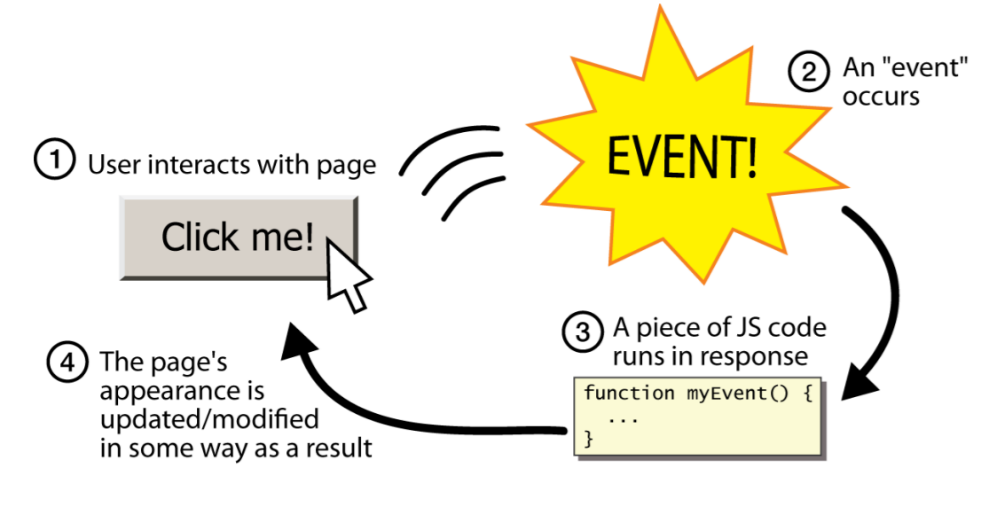
# What is an Event?

# JavaScript’s interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

# When the page loads, it is called an event. Many events can occur while a user is interacting with a Web page. For example, a user might click on a button, change some text, move the mouse pointer over a hyperlink or away from one, and, of course, cause a document to load, pressing any key, closing a window, resizing a window, etc.

# Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

# Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.



Event-driven programming is when parts of the programming are executed in an unpredictable sequence in response to specific events.

Events are objects in JavaScript with case-sensitive names, all of which are lower-case.

**What is Event handling?**

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

**Source -** The source is an object on which event occurs. Source is responsible for providing information of the occurred event to its handler. Java provides classes for source object.

**Listener -** It is also known as event handler. An event handler is a script that is implicitly executed in response to an event happening. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event a then returns.

**Functions of Event Handling**

* Event Handling identifies where an event should be forwarded.
* It makes the forward event.
* It receives the forwarded event.
* It takes some kind of appropriate action in response, such as writing to a log, sending an error or recovery routine or sending a message.
* The event handler may ultimately forward the event to an event consumer.

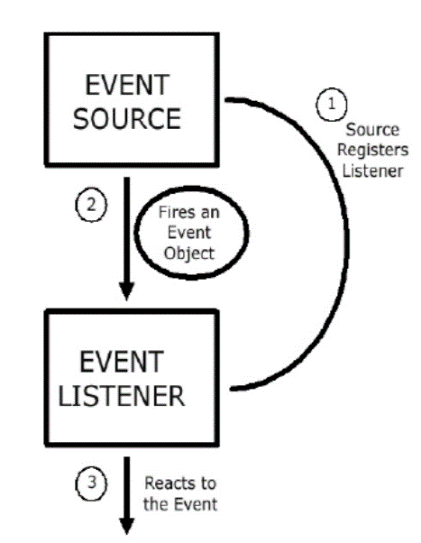


Figure 1- Delegation Event Model

JavaScript contains predetermined event handlers that deal with these events. Table shown below describes some commonly used JavaScript events.

|  |  |
| --- | --- |
| **Events** | **Description** |
| onabort | Occurs when the loading of an image is aborted |
| onblur | Occurs when input focus is removed from a form element (e.g., when a user clicks the mouse button outside of a particular field) |
| onclick | Occurs when the user clicks on a link or form element |
| onchange | Occurs when a user changes the value of a form field |
| onerror | Occurs when an error takes place while a page or image is loading |
| onfocus | Occurs when a user gives input or focus to a form element |
| onload | Occurs when a page is loaded into the browser (i.e., opened) |
| onmouseOver | Occurs when the user moves the mouse pointer over a link, image or other visible element on a page |
| onmouseOut | Occurs when the mouse pointer leaves a link, image or other visible element on a page |
| onreset | Occurs when a form's Reset button is clicked |
| onselect | Occurs when the user selects the text in a form field |
| onsubmit | Occurs when a form's Submit button is clicked |
| onunload | Occurs when a page is unloaded from the browser (i.e., closed) |

Table 1- JavaScript user event examples

## Example – onload event

<!DOCTYPE html>

<!-- load.html

An example to illustrate the load event

A document for load.js -->

<html lang = "en">

<head>

<title> onLoad event handler </title>

<meta charset = "utf-8" />

<script type = "text/javascript" src = "load.js" > </script>

</head>

<body onload = "loadGreeting();">

<p />

</body>

</html>

Load.js

// load.js

// An example to illustrate the load event

// The onload event handler

function loadGreeting() {

alert("You are visiting the home page of \n"+ "Pete's Pickled Peppers\n" + "WELCOME!!!");

}

# Exceptional handling

Exception handling is one of the powerful JavaScript features to handle errors and maintain a regular JavaScript code/program flow.

An exception is an object with an explanation of what went amiss. Also, it discovers where the problem occurred. Errors occur due to mistakes made by developers, wrong input, or unforeseeable things.

A few reasons why exceptions occur are listed below:

* Dividing a number by zero: This results in infinity, thus throwing an exception.
* When a requested file does not exist in the system.
* When the user provides the wrong input.
* When the network drops during communication.

# If a software engineer fails to plan for failure, then whatever project or code they are working on may not be successful (when an error does occur). That is where [exception handling](https://en.wikipedia.org/wiki/Exception_handling) comes into play.

# When JavaScript encounters an error and raises an exception. The JavaScript translator looks for an exception handling code rather than proceeding to the next statement. In a programming environment, exceptions can be handled, but errors cannot.

# JavaScript error types

# Different errors may occur while executing a JavaScript code. There are three types of errors.

1. [Syntax Errors:](https://developer.mozilla.org/en-US/docs/Glossary/Syntax_error#:~:text=An%20exception%20caused%20by%20the,you%20trigger%20a%20syntax%20error.) These are errors that cannot be interpreted by the computer. These errors stop the program from working.

# In JavaScript, these errors are:

* Spelling errors (wrong spelling such as fiction instead of function).
* The omission of important characters, such as not using a semicolon to end a statement.
* Use of the wrong indentation.

1. Runtime Errors: These errors take place during execution. The errors get detected when your program runs. It crashes or raises an exception. Thus, exception handlers handle exception errors.

These errors are often caused by:

* The program not being able to find data because it does not exist.
* The data being an invalid type of data.

1. Logical Errors: These types of errors do not throw an error or an exception at all. This is because they result from the code not doing what the developer intends it to. It’s challenging to find logical errors. They can only be found through thorough testing.

**Error objects**

When a runtime error occurs, it stops the code and raises an error object.

The error object has two properties:

* Name: It gives the error name.
* Message: It sets or returns the error message in the form of a string.

JavaScript uses six types of error objects. These error objects are the foundation of exception handling.

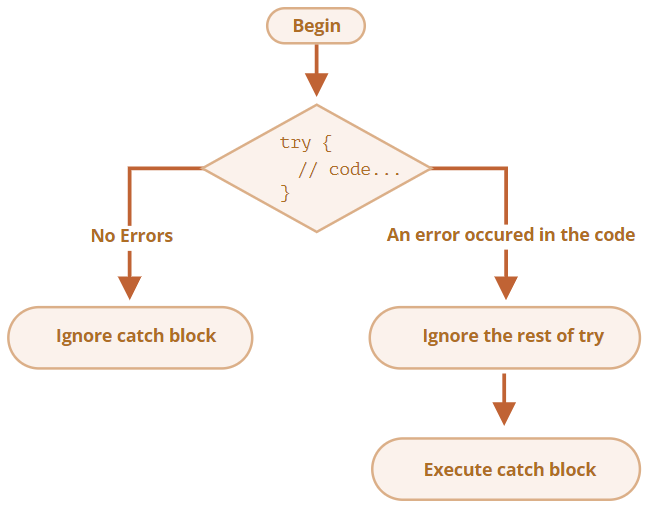
* EvalError: The EvalError function indicates the error that occurred in the eval() function. It’s a global function that evaluates the JavaScript string. JavaScript does not throw this exception anymore.
* RangeError: RangeError exceptions occur when a numeric value is outside the specified range.
* ReferenceError: A ReferenceError exception occurs when undeclared variables are used. These exceptions commonly occur due to spelling errors on variables.
* Syntax Error: A Syntax Error exception occurs when JavaScript language rules get broken.
* TypeError: A TypeError exception occurs when a value is different from the one expected.
* URIError: A URIError exception is raised by encodeURI() and decodeURI() methods.

# How to handle exceptions in JavaScript

# Know that we now understand what exceptions are. It’s time to learn how to handle them to stop our code from crashing. JavaScript handles exceptions in try-catch-finally statements and throw statements.

# Key Terms

* A *try-catch-finally* statement is a code or program that handles exceptions.
* The *try* clause runs the code that generates exceptions.
* The *catch* clause catches exceptions that are thrown.
* A *finally* clause always gets executed.
* The *throw* statement generates exceptions.



Insert the JavaScript code inside the script tag to understand how each exception handling statement works.

# <!DOCTYPE html>

# <html lang="en">

# <head>

# <meta charset="UTF-8">

# <meta name="viewport" content="width=device-width, initial-scale=1.0">

# <title>Try-Catch-Finally Statement</title>

# </head>

# <body>

# <script type="text/JavaScript"></script>

# <p>Click the button to see the output</p>

# <button type="button" onclick="myFunction()">Click Here</button>

# </body>

# </html>

# Throw statements - The throw statement is to raise your built-in exceptions.

# Below is an example of a throw statement:

# function myFunction() {

# const x = 50;

# const y = 0;

# try {

# if (y === 0) {

# throw ("This is division by zero error");

# } else {

# const z = x / y;

# }

# } catch (error) {

# alert("Error: " + error);

# }

# }

# Try catch statements - The try clause has the main code that may generate exceptions. If an exception is raised, the catch clause gets executed.

# Here is an example of a try-catch statement:

# function myFunction() {

# const j = 70;

# try {

# alert ("The value of j is: " + j);

# } catch (error) {

# alert("Error: " + error.message);

# }

# }

# In the example above, we have made a typo error while calling the in-built alert() function. We have misspelled alert to produce an error deliberately. The catch clause catches the error and executes the code.

# Try catch finally statements - The finally statement is the last block to be executed. It executes after try and catch clauses.

# Here is an example of try-catch-finally statements:

# function myFunction() {

# const j = 70;

# try {

# alert("The value of j is : " + j);

# } catch (error) {

# alert("Error: " + error.message);

# } finally {

# alert("Finally: Finally gets executed")

# }

# }

# The Browser Object Model (BOM)

A Web page is made dynamic by applying JavaScript processing to the HTML elements on that page. Up to this point you probably have considered HTML tags simply as markup codes providing structure to page content and supplying mechanisms through which styling is applied to that content. Importantly, though, HTML tags are also **software objects**. That is, all HTML tags have properties and methods that can be programmed. As is the case with all software objects, **properties**refer to structural, visual, or content characteristics of the element; methods refer to actions the object can perform. HTML tags, then, are programmable through JavaScript processing routines, or scripts, that set their properties and activate their methods in order to make Web pages **dynamic**.

The **browser object model (BOM)** is a hierarchy of browser objects that are used to manipulate methods and properties associated with the Web browser itself. Objects that make up the BOM include the window object, navigator object, screen object, history, location object, and the document object. The Document Object consists of objects that are used to manipulate methods and properties of the document or Web page loaded in the browser window. The document object represents the Web page currently loaded in the browser window. Each HTML element or tag that makes up the document is also considered an object. It is not necessary to explicitly create any of the objects that make up the browser object model. The objects are automatically created when a Web browser opens a Web page.

**The BOM Hierarchy**

The top-level object in the BOM is the window object. The **window object** represents the browser window. All other browser objects are contained within the window object. The window object includes a number of properties and methods that can be used to control the Web browser. The window object along with its properties and methods are discussed in more detail in a later section.

The **document object** represents the Web page displayed in the browser. All elements on a Web page including HTML tags are contained within the document object. Since the document object is often considered the most important part of the BOM, it is represented by its own object model called the **Document Object Model or DOM**. The DOM will be discussed in more detailed in later tutorials.

Other objects of the browser object model include the **navigator object**, the **screen object**, that contains information about the visitor's screen, the **history object**, that is part of the window object and contains the URLs that have been visited by the user, and the **location object**that contains information about the current URL. Within the window object are **document objects** representing elements within the Web pages. The general hierarchy of the BOM is shown in the illustration below.

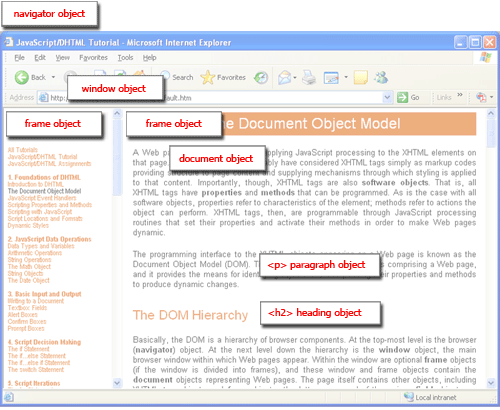


Figure 2 Components of the Browser Object Model (BOM)

**Identifying BOM Objects**

In order to program BOM objects, they must be identified to the scripts that manipulate their properties and methods. The following table summarizes several of the references used within scripts to identify common BOM objects. These and other reference notations are explained and illustrated throughout these tutorials.

|  |  |
| --- | --- |
| **Reference** | **Object** |
| window | The main browser window |
| window.navigator | Information about the browser itself |
| window.screen | The user's screen |
| window.history | URLs visited by a user |
| window.location | The current URL |
| window.document (document) | The document appearing in the main browser window |
| document.getElementById("id") | An HTML element appearing in a document and identified by its assigned *id* value. |

As you can see, script references to BOM objects use standard dotted notation to trace through the BOM hierarchy to identify particular objects. In some cases, there are short-cut notations that do not required the complete hierarchical path to an object. For example, **window.document** can be shorten to **document**.

A good portion of client-side Web develop is in working with the properties and methods associated with the browser itself, with its windows, and with the documents that occupy them. The largest part of client-side scripting, though, is in working with the properties and methods of HTML elements appearing on a Web page. In most cases, this involves detecting the **style settings** of HTML tags and changing these settings to change the appearance of, or to change the content enclosed by, these tags. In other cases, it involves calling up built-in methods to affect the behaviors of HTML tags. Being able to make proper script references to HTML elements is an important aspect of client-side scripting.

The most important object in the Browser Object Model is the window object. It helps in accessing information about the browser and its components. To access these features, it has various methods and properties.

|  |  |
| --- | --- |
| **Method** | **Description** |
| window.alert() | Creates dialog box with message and an OK button |
| window.blur() | Remove focus from window |
| window.close() | Closes a browser window |
| window.confirm() | Creates dialog box with message, an OK button and a cancel button |
| window.getComputedStyle() | Get CSS styles applied to an element |
| window.moveTo(x,y) | Move a window's left and top edge to supplied coordinates |
| window.open() | Opens new browser window with URL specified as parameter |
| window.print() | Tells browser that user wants to print contents of current page |
| window.prompt() | Creates dialog box for retrieving user input |
| window.scrollBy() | Scrolls the document by the specified number of pixels |
| window.scrollTo() | Scrolls the document to the specified coordinates |
| window.setInterval() | Do something repeatedly at specified intervals |
| window.setTimeout() | Do something after a specified amount of time |
| window.stop() | Stop window from loading |

The Window Object contains the following properties.

|  |  |
| --- | --- |
| **Property** | **Description** |
| window.closed | Whether the window has been closed |
| window.length | Number of <iframe> elements in window |
| window.name | Gets or sets the name of the window |
| window.innerHeight | Height of window |
| window.innerWidth | Width of window |
| window.screenX | X-coordinate of pointer, relative to top left corner of screen |
| window.screenY | Y-coordinate of pointer, relative to top left corner of screen |
| window.location | Current URL of window object (or local file path) |
| window.history | Reference to history object for browser window or tab. |
| window.screen | Reference to screen object |
| window.pageXOffset | Distance document has been scrolled horizontally |
| window.pageYOffset | Distance document has been scrolled vertically |

**Example of alert() in JavaScript**

It displays alert dialog box. It has message and ok button.

**<script** type="text/javascript"**>**

function msg(){

 alert("Hello Alert Box");

}

**</script>**

**<input** type="button" value="click" onclick="msg()"**/>**

**Example of confirm() in JavaScript**

It displays the confirm dialog box. It has message with ok and cancel buttons.

**<script** type="text/javascript"**>**

function msg(){

var v= confirm("Are u sure?");

if(v==true){

alert("ok");  }

else{

alert("cancel");

}

}  **</script>**

**<input** type="button" value="delete record" onclick="msg()"**/>**

**Example of prompt() in JavaScript**

It displays prompt dialog box for input. It has message and textfield.

**<script** type="text/javascript"**>**

function msg(){

var v= prompt("Who are you?");

alert("I am "+v);

}

**</script>**

**<input** type="button" value="click" onclick="msg()"**/>**

**Example of open() in JavaScript**

It displays the content in a new window.

**<script** type="text/javascript"**>**

function msg(){

open("http://www.edunetfoundation.com");

}

**</script>**

**<input** type="button" value="javascript1" onclick="msg()"**/>**

**Example of setTimeout() in javascript**

It performs its task after the given milliseconds.

**<script** type="text/javascript"**>**

function msg(){

setTimeout(

function(){

alert("Display JavaScript after 2 seconds")

},2000);

}

**</script>**

**<input** type="button" value="click" onclick="msg()"**/>**

**JavaScript Navigator Object**

The **JavaScript navigator object** is used for browser detection. It can be used to get browser information such as appName, appCodeName, userAgent etc.

The navigator object is the window property, so it can be accessed by:

window.navigator  Or, navigator

**Property of JavaScript navigator object**

There are many properties of navigator object that returns information of the browser.

|  |  |  |
| --- | --- | --- |
| **No.** | **Property** | **Description** |
| 1 | appName | returns the name |
| 2 | appVersion | returns the version |
| 3 | appCodeName | returns the code name |
| 4 | cookieEnabled | returns true if cookie is enabled otherwise false |
| 5 | userAgent | returns the user agent |
| 6 | language | returns the language. It is supported in Netscape and Firefox only. |
| 7 | userLanguage | returns the user language. It is supported in IE only. |
| 8 | plugins | returns the plugins. It is supported in Netscape and Firefox only. |
| 9 | systemLanguage | returns the system language. It is supported in IE only. |
| 10 | mimeTypes[] | returns the array of mime type. It is supported in Netscape and Firefox only. |
| 11 | platform | returns the platform e.g. Win32. |
| 12 | online | returns true if browser is online otherwise false. |

**Methods of JavaScript navigator object**

The methods of navigator object are given below.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | javaEnabled() | checks if java is enabled. |
| 2 | taintEnabled() | checks if taint is enabled. It is deprecated since JavaScript 1.2. |

**Example of navigator object**

Let’s see the different usage of history object.

<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>

**JavaScript screen object**

The Screen object provides the attributes of the screen on which the current window is being rendered.

To access the Screen object, you use the screen property of the window object: window.screen

The Screen object is typically used by the web analytic software like Google Analytics to collect information of the client device on which the web browsers are running.

# Property of JavaScript Screen Object

There are many properties of screen object that returns information of the browser.

|  |  |  |
| --- | --- | --- |
| No. | Property | Description |
| 1 | width | returns the width of the screen |
| 2 | height | returns the height of the screen |
| 3 | availWidth | returns the available width |
| 4 | availHeight | returns the available height |
| 5 | colorDepth | returns the color depth |
| 6 | pixelDepth | returns the pixel depth. |

# Example of JavaScript Screen Object

# <script>

# document.writeln("<br/>screen.width: "+screen.width);

# document.writeln("<br/>screen.height: "+screen.height);

# document.writeln("<br/>screen.availWidth: "+screen.availWidth);

# document.writeln("<br/>screen.availHeight: "+screen.availHeight);

# document.writeln("<br/>screen.colorDepth: "+screen.colorDepth);

# document.writeln("<br/>screen.pixelDepth: "+screen.pixelDepth);

# </script>

# JavaScript setTimeout() method

# The setTimeout() method in JavaScript is used to execute a function after waiting for the specified time interval. This method returns a numeric value that represents the ID value of the timer.

# The setTimeout() method executes the function only once. This method can be written with or without the *window* prefix.

# We can use the clearTimeout() method to stop the timeout or to prevent the execution of the function specified in the setTimeout() method. The value returned by the setTimeout() method can be used as the argument of the clearTimeout() method to cancel the timer.

# The commonly used syntax of the setTimeout() method is given below.

# window.setTimeout(function, milliseconds);

**Parameter values**

This method takes two parameter values function and milliseconds that are defined as follows.

**function:** It is the function containing the block of code that will be executed.

**milliseconds:** This parameter represents the time-interval after which the execution of the function takes place. The interval is in milliseconds. Its default value is 0. It defines how often the code will be executed. If it is not specified, the value 0 is used.

# <html>

# <head>

# <title> setTimeout() method </title>

# </head>

# <body>

# <h1> Hello World :) :) </h1>

# <h3> This is an example of using the setTimeout() method </h3>

# <p> Click the following button before 2 seconds to see the effect. </p>

# <button onclick = "stop()"> Stop </button>

# <script>

# var a = setTimeout(fun1, 2000);

# function fun1()  {

# var win1 = window.open();

# win1.document.write(" <h2> Learn Javascript </h2>");

# setTimeout(function(){win1.close()}, 2000);

# }

# function stop() {

# clearTimeout(a);

# }

# </script>

# </body>

# </html>

# JavaScript setInterval() method

# The setInterval() method in JavaScript is used to repeat a specified function at every given time-interval. It evaluates an expression or calls a function at given intervals. This method continues the calling of function until the window is closed or the clearInterval() method is called. This method returns a numeric value or a non-zero number that identifies the created timer.

# Unlike the setTimeout() method, the setInterval() method invokes the function multiple times. This method can be written with or without the *window* prefix.

# The commonly used syntax of setInterval() method is given below:

# window.setInterval(function, milliseconds);

# Parameter values

# This method takes two parameter values *function* and *milliseconds* that are defined as follows.

# function: It is the function containing the block of code that will be executed.

# milliseconds: This parameter represents the length of the time interval between each execution. The interval is in milliseconds. It defines how often the code will be executed. If its value is less than 10, the value 10 is used.

# <html>

# <head>

# <title> setInterval() method </title>

# </head>

# <body>

# <h1> Hello World :) :) </h1>

# <h3> This is an example of using the setInterval() method </h3>

# <p> Here, the background color changes on every 200 milliseconds. </p>

# <button onclick = "stop()"> Stop </button>

# 

# <script>

# var var1 = setInterval(color, 200);

# function color() {

# var var2 = document.body;

# var2var2.style.backgroundColor = var2.style.backgroundColor == "lightblue" ? "lightgreen”: "lightblue";

# }

# function stop() {

# clearInterval(var1);

# }

# </script>

# </body>

# </html>

# JavaScript Date

# The JavaScript date object can be used to get year, month and day. You can display a timer on the webpage by the help of JavaScript date object.

Syntax:

* new Date();
* new Date(value);
* new Date(dateAsString);
* new Date(year, month[, day[, hour[, minute[, second[, millisecond]]]]]);

Let's see the list of JavaScript date methods with their description.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| getDate() | It returns the integer value between 1 and 31 that represents the day for the specified date on the basis of local time. |
| getDay() | It returns the integer value between 0 and 6 that represents the day of the week on the basis of local time. |
| getFullYears() | It returns the integer value that represents the year on the basis of local time. |
| getHours() | It returns the integer value between 0 and 23 that represents the hours on the basis of local time. |
| getMilliseconds() | It returns the integer value between 0 and 999 that represents the milliseconds on the basis of local time. |
| getMinutes() | It returns the integer value between 0 and 59 that represents the minutes on the basis of local time. |
| getMonth() | It returns the integer value between 0 and 11 that represents the month on the basis of local time. |
| getSeconds() | It returns the integer value between 0 and 60 that represents the seconds on the basis of local time. |
| setDate() | It sets the day value for the specified date on the basis of local time. |
| setDay() | It sets the particular day of the week on the basis of local time. |
| setFullYears() | It sets the year value for the specified date on the basis of local time. |
| setHours() | It sets the hour value for the specified date on the basis of local time. |
| setMilliseconds() | It sets the millisecond value for the specified date on the basis of local time. |
| setMinutes() | It sets the minute value for the specified date on the basis of local time. |
| setMonth() | It sets the month value for the specified date on the basis of local time. |
| setSeconds() | It sets the second value for the specified date on the basis of local time. |
| toJSON() | It returns a string representing the Date object. It also serializes the Date object during JSON serialization. |
| toString() | It returns the date in the form of string. |
| toTimeString() | It returns the time portion of a Date object. |
| valueOf() | It returns the primitive value of a Date object. |

# Examples:

Current Date and Time: **<span** id="txt"**></span>**

**<script>**

var today=new Date();

document.getElementById('txt').innerHTML=today;

**</script>**

Current Time: **<span** id="txt"**></span>**

**<script>**

window.onload=function(){getTime();}

function getTime(){

var today=new Date();

var h=today.getHours();

var m=today.getMinutes();

var s=today.getSeconds();

// add a zero in front of numbers**<10**

m=checkTime(m);

s=checkTime(s);

document.getElementById('txt').innerHTML=h+":"+m+":"+s;

setTimeout(function(){getTime()},1000);

}

//setInterval("getTime()",1000);//another way

function checkTime(i){

if (i**<10**){

  i="0" + i;

 }

return i;

}

**</script>**

**Output:**

Current Time: 15:18:10

**Example - Convert to JSON**

var date1 = new Date();

date1.toJSON();

Output - "2022-02-23T15:49:08.596Z"

**What is Ajax?**

Ajax stands for **A**synchronous **J**avascript **A**nd **X**ml. Ajax is just a means of loading data from the server and selectively updating parts of a web page without reloading the whole page.

Basically, what Ajax does is make use of the browser's built-in XMLHttpRequest (XHR) object to send and receive information to and from a web server asynchronously, in the background, without blocking the page or interfering with the user's experience.

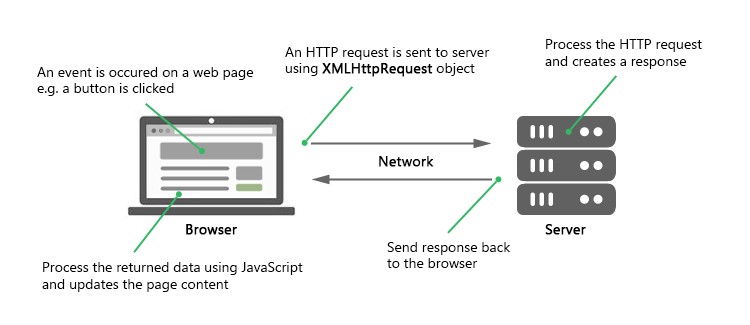
Ajax has become so popular that you hardly find an application that doesn't use Ajax to some extent. The example of some large-scale Ajax-driven online applications are: Gmail, Google Maps, Google Docs, YouTube, Facebook, Flickr, and so many other applications.

**Understanding How Ajax Works**

To perform Ajax communication JavaScript uses a special object built into the browser—an XMLHttpRequest (XHR) object—to make HTTP requests to the server and receive data in response.

All modern browsers (Chrome, Firefox, IE7+, Safari, Opera) support the XMLHttpRequest object.

The following illustrations demonstrate how Ajax communication works:



Since Ajax requests are usually asynchronous, execution of the script continues as soon as the Ajax request is sent, i.e., the browser will not halt the script execution until the server response comes back.

In the following section we'll discuss each step involved in this process one by one:

**Sending Request and Retrieving the Response**

Before you perform Ajax communication between client and server, the first thing you must do is to instantiate an XMLHttpRequest object, as shown below:

var request = new XMLHttpRequest();

Now, the next step in sending the request to the server is to instantiating the newly-created request object using the open() method of the XMLHttpRequest object.

The open() method typically accepts two parameters— the HTTP request method to use, such as "GET", "POST", etc., and the URL to send the request to, like this:

request.open("GET", "info.txt"); -Or- request.open("POST", "add-user.php");

And finally send the request to the server using the send() method of the XMLHttpRequest object.

request.send(); -Or- request.send(*body*);

The GET method is generally used to send small amount of data to the server. Whereas, the POST method is used to send large amount of data, such as form data.

In GET method, the data is sent as URL parameters. But, in POST method, the data is sent to the server as a part of the HTTP request body. Data sent through POST method will not visible in the URL.

In the following section we'll take a closer look at how Ajax requests actually works.

**Performing an Ajax GET Request**

The GET request is typically used to get or retrieve some kind of information from the server that doesn't require any manipulation or change in database, for example, fetching search results based on a term, fetching user details based on their id or name, and so on.

The following example will show you how to make an Ajax GET request in JavaScript.

**Example**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax GET Demo</title>

<script>

function displayFullName() {

// Creating the XMLHttpRequest object

var request = new XMLHttpRequest();

// Instantiating the request object

request.open("GET", "greet.php?fname=John&lname=Clark");

// Defining event listener for readystatechange event

request.onreadystatechange = function() {

// Check if the request is competed and was successful

if(this.readyState === 4 && this.status === 200) {

// Inserting the response from server into an HTML element

document.getElementById("result").innerHTML = this.responseText;

}

};

// Sending the request to the server

request.send();

}

</script>

</head>

<body>

<div id="result">

<p>Content of the result DIV box will be replaced by the server response</p>

</div>

<button type="button" onclick="displayFullName()">Display Full Name</button>

</body>

</html>

When the request is asynchronous, the send() method returns immediately after sending the request. Therefore, you must check where the response currently stands in its lifecycle before processing it using the readyState property of the XMLHttpRequest object.

The readyState is an integer that specifies the status of an HTTP request. Also, the function assigned to the onreadystatechange event handler called every time the readyState property changes. The possible values of the readyState property are summarized below.

|  |  |  |
| --- | --- | --- |
| **Value** | **State** | **Description** |
| 0 | UNSENT | An XMLHttpRequest object has been created, but the open() method hasn't been called yet (i.e. request not initialized). |
| 1 | OPENED | The open() method has been called (i.e. server connection established). |
| 2 | HEADERS\_RECEIVED | The send() method has been called (i.e. server has received the request). |
| 3 | LOADING | The server is processing the request. |
| 4 | DONE | The request has been processed and the response is ready. |

The status property returns the numerical HTTP status code of the XMLHttpRequest's response. Some of the common HTTP status codes are listed below:

* 200 — OK. The server successfully processed the request.
* 404 — Not Found. The server can't find the requested page.
* 503 — Service Unavailable. The server is temporarily unavailable.

Please check out the [HTTP status codes](https://www.tutorialrepublic.com/html-reference/http-status-codes.php) reference for a complete list of response codes.

Here's the code from our "greet.php" file that simply creates the full name of a person by joining their first name and last name and outputs a greeting message.

**Example**

<?php

if(isset($\_GET["fname"]) && isset($\_GET["lname"])) {

$fname = htmlspecialchars($\_GET["fname"]);

$lname = htmlspecialchars($\_GET["lname"]);

// Creating full name by joining first and last name

$fullname = $fname . " " . $lname;

// Displaying a welcome message

echo "Hello, $fullname! Welcome to our website.";

} else {

echo "Hi there! Welcome to our website.";

}

?>

**Performing an Ajax POST Request**

The POST method is mainly used to submit a form data to the web server.

The following example will show you how to submit form data to the server using Ajax.

**Example**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>JavaScript Ajax POST Demo</title>

<script>

function postComment() {

// Creating the XMLHttpRequest object

var request = new XMLHttpRequest();

// Instantiating the request object

request.open("POST", "confirmation.php");

// Defining event listener for readystatechange event

request.onreadystatechange = function() {

// Check if the request is compete and was successful

if(this.readyState === 4 && this.status === 200) {

// Inserting the response from server into an HTML element

document.getElementById("result").innerHTML = this.responseText;

}

};

// Retrieving the form data

var myForm = document.getElementById("myForm");

var formData = new FormData(myForm);

// Sending the request to the server

request.send(formData);

}

</script>

</head>

<body>

<form id="myForm">

<label>Name:</label>

<div><input type="text" name="name"></div>

<br>

<label>Comment:</label>

<div><textarea name="comment"></textarea></div>

<p><button type="button" onclick="postComment()">Post Comment</button></p>

</form>

<div id="result">

<p>Content of the result DIV box will be replaced by the server response</p>

</div>

</body>

</html>

If you are not using the FormData object to send form data, for example, if you're sending the form data to the server in the *query string* format, i.e. request.send(key1=value1&key2=value2) then you need to explicitly set the request header using setRequestHeader() method, like this:

request.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

The setRequestHeader() method, must be called after calling open(), but before calling send().

Some common request headers are: application/x-www-form-urlencoded, multipart/form-data, application/json, application/xml, text/plain, text/html, and so on.

Here's the code of our "confirmation.php" file that simply outputs the values submitted by the user.

**Example**

<?php

if($\_SERVER["REQUEST\_METHOD"] == "POST") {

$name = htmlspecialchars(trim($\_POST["name"]));

$comment = htmlspecialchars(trim($\_POST["comment"]));

// Check if form fields values are empty

if(!empty($name) && !empty($comment)) {

echo "<p>Hi, <b>$name</b>. Your comment has been received successfully.<p>";

echo "<p>Here's the comment that you've entered: <b>$comment</b></p>";

} else {

echo "<p>Please fill all the fields in the form!</p>";

}

} else {

echo "<p>Something went wrong. Please try again.</p>";

}

?>

For security reasons, browsers do not allow you to make cross-domain Ajax requests. This means you can only make Ajax requests to URLs from the same domain as the original page, for example, if your application is running on the domain "mysite.com", you cannot make Ajax request to "othersite.com" or any other domain. This is commonly known as *same origin policy*.

However, you can load images, style sheets, JS files, and other resources from any domain.

**JavaScript JSON**

JSON stands for JavaScript Object Notation. JSON is extremely lightweight data-interchange format for data exchange between server and client which is quick and easy to parse and generate.

Like XML, JSON is also a text-based format that's easy to write and easy to understand for both humans and computers, but unlike XML, JSON data structures occupy less bandwidth than their XML versions. JSON is based on two basic structures:

* Object: This is defined as an unordered collection of key/value pairs (i.e. key:value). Each object begins with a left curly bracket { and ends with a right curly bracket }. Multiple key/value pairs are separated by a comma ,.
* Array: This is defined as an ordered list of values. An array begins with a left bracket [ and ends with a right bracket ]. Values are separated by a comma ,.

# In JSON, property names or keys are always strings, while the value can be a string, number, true or false, null or even an object or an array. Strings must be enclosed in double quotes " and can contain escape characters such as \n, \t and \. A JSON object may look like this:

Example

{

"book": {

"name": "Harry Potter and the Goblet of Fire",

"author": "J. K. Rowling",

"year": 2000,

"genre": "Fantasy Fiction",

"bestseller": true

}}

JavaScript JSON Methods

Let's see the list of JavaScript **JSON** method with their description.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| [JSON.parse()](https://www.javatpoint.com/javascript-json-parse-method) | This method takes a JSON string and transforms it into a JavaScript object. |
| [JSON.stringify()](https://www.javatpoint.com/javascript-json-stringify-method) | This method converts a JavaScript value (JSON object) to a JSON string representation. |

Let's see an example to convert string in JSON format using parse() and stringify() method.

**<script>**

//JavaScript to illustrate JSON.parse() method.

var j = '{"Name":"Krishna","Email": "XYZ", "CN": "12345"}';

var data = JSON.parse(j);

document.write("Convert string in JSON format using parse() method**<br>**");

document.write(data.Email); //expected output: XYZ

//JavaScript to illustrate JSON.stringify() method.

var j = {Name:"Krishna",

Email: "XYZ", CN : 12345};

var data = JSON.stringify(j);

document.write("**<br>**Convert string in JSON format using stringify()  method**<br>**");

document.write(data); //expected output: {"Name":"Krishna","Email":"XYZ","CN":12345}

**</script>**

**Output:**

Convert string in JSON format using parse() method

XYZ

Convert string in JSON format using stringify() method

{"Name":"John","Email":"XYZ","CN":12345}

References

1. <https://www.section.io/engineering-education/exception-handling-in-javascript/>
2. <https://itwebtutorials.mga.edu/js/chp1/browser-object-model.aspx>
3. <https://riptutorial.com/javascript>
4. <https://www.javatpoint.com/javascript-tutorial>