**JAVASCRIPT – Errors and Exceptions**

* **Syntax Errors**

Syntax errors, also called parsing errors, occur at compile time in traditional programming languages and at interpret time in JavaScript.

For example, the following line causes a syntax error because it is missing a closing parenthesis.

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

**<!--**

**window.print(;**

**//-->**

**</script>**

* **Runtime Errors**

Runtime errors, also called exceptions, occur during execution (after compilation/interpretation).

For example, the following line causes a runtime error because here the syntax is correct, but at runtime, it is trying to call a method that does not exist.

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

**<!--**

**window.printme();**

**//-->**

**</script>**

* **Logical Errors**

Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result you expected.

You cannot catch those errors, because it depends on your business requirement what type of logic you want to put in your program.

## **The try...catch...finally Statement**

The latest versions of JavaScript added exception handling capabilities. JavaScript implements the try...catch...finally construct as well as the throw operator to handle exceptions.

You can catch programmer-generated and runtime exceptions, but you cannot catch JavaScript syntax errors.

Here is the try...catch...finally block syntax −

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

**<!--**

**try {**

**// Code to run**

**[break;]**

**}**

**catch ( e ) {**

**// Code to run if an exception occurs**

**[break;]**

**}**

**[ finally {**

**// Code that is always executed regardless of**

**// an exception occurring**

**}]**

**//-->**

**</script>**

* **The throw Statement**

You can use a throw statement to raise your built-in exceptions or your customized exceptions. Later these exceptions can be captured and you can take appropriate action.

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

**<!--**

**function myFunc() {**

**var a = 100;**

**var b = 0;**

**try {**

**if ( b == 0 ) {**

**throw( "Divide by zero error." );**

**} else {**

**var c = a / b;**

**}**

**}**

**catch ( e ) {**

**alert("Error: " + e );**

**}**

**}**

**//-->**

**</script>**

* **The onerror( ) Method**

The onerror event handler was the first feature to facilitate error handling in JavaScript. The error event is fired on the window object whenever an exception occurs on the page.

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

**<!--**

**window.onerror = function () {**

**alert("An error occurred.");**

**}**

**//-->**

**</script>**

The onerror event handler provides three pieces of information to identify the exact nature of the error −

* Error message − The same message that the browser would display for the given error
* URL − The file in which the error occurred
* Line number− The line number in the given URL that caused the error

Here is an example to show how to extract this information.

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

**<!--**

**window.onerror = function (msg, url, line) {**

**alert("Message : " + msg );**

**alert("url : " + url );**

**alert("Line number : " + line );**

**}**

**//-->**

**</script>**

**JAVASCRIPT – Form Validation**

* **Basic Form Validation**

HTML form validation can be done by JavaScript.

If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

**function validateForm() {**

**let x = document.forms["myForm"]["fname"].value;**

**if (x == "") {**

**alert("Name must be filled out");**

**return false;**

**}**

**}**

* **Data Format Validation**

Data validation is the process of ensuring that user input is clean, correct, and useful.

Typical validation tasks are:

* Has the user filled in all required fields?
* Has the user entered a valid date?
* Has the user entered text in a numeric field?

Most often, the purpose of data validation is to ensure correct user input.

Validation can be defined by many different methods, and deployed in many different ways.

**Server side validation** is performed by a web server, after input has been sent to the server.

**Client side validation** is performed by a web browser, before input is sent to a web server.

**JAVASCRIPT – Animation**

* **Manual Animation**

So let's implement one simple animation using DOM object properties and JavaScript functions as follows. The following list contains different DOM methods.

* We are using the JavaScript function getElementById() to get a DOM object and then assigning it to a global variable imgObj.
* We have defined an initialization function init() to initialize imgObj where we have set its position and left attributes.
* We are calling the initialization function at the time of window load.
* Finally, we are calling the moveRight() function to increase the left distance by 10 pixels. You could also set it to a negative value to move it to the left side.

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

**<!--**

**var imgObj = null;**

**function init() {**

**imgObj = document.getElementById('myImage');**

**imgObj.style.position= 'relative';**

**imgObj.style.left = '0px';**

**}**

**function moveRight() {**

**imgObj.style.left = parseInt(imgObj.style.left) + 10 + 'px';**

**}**

**window.onload = init;**

**//-->**

**</script>**

* **Automated Animation**

In the above example, we saw how an image moves to right with every click. We can automate this process by using the JavaScript function setTimeout() as follows −

Here we have added more methods. So let's see what is new here −

* The moveRight() function is calling setTimeout() function to set the position of *imgObj*.
* We have added a new function stop() to clear the timer set by setTimeout() function and to set the object at its initial position.

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

**<!--**

**var imgObj = null;**

**var animate ;**

**function init() {**

**imgObj = document.getElementById('myImage');**

**imgObj.style.position= 'relative';**

**imgObj.style.left = '0px';**

**}**

**function moveRight() {**

**imgObj.style.left = parseInt(imgObj.style.left) + 10 + 'px';**

**animate = setTimeout(moveRight,20); // call moveRight in 20msec**

**}**

**function stop() {**

**clearTimeout(animate);**

**imgObj.style.left = '0px';**

**}**

**window.onload = init;**

**//-->**

**</script>**

* **Rollover with a Mouse Event**

With the help of JavaScript and html we create a "Rollover with a mouse event" function. Here we have two images. when we place mouse over image(1), the image(1) changes to image (2) and when we move the mouse away from that image, the original image,( i.e. image(1) ) gets displayed again.

Code Sample:

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

**if(document.images){**

**var image1 = new Image();**

**image1.src = "img/images (1).jpg";**

**var image2 = new Image();**

**image2.src = "img/images (2).jpg";**

**}**

**</script>**

**JAVASCRIPT – Multimedia**

* **Checking for Plug-Ins**

Each plug-in has an entry in the array. Each entry has the following properties −

* name − is the name of the plug-in.
* filename − is the executable file that was loaded to install the plug-in.
* description − is a description of the plug-in, supplied by the developer.
* mimeTypes − is an array with one entry for each MIME type supported by the plug-in.

You can use these properties in a script to find out the installed plug-ins, and then using JavaScript, you can play appropriate multimedia files. Take a look at the following example.

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

**media = navigator.mimeTypes["video/quicktime"];**

**if (media) {**

**document.write("<embed src = 'quick.mov' height = 100 width = 100>");**

**} else {**

**document.write("<img src = 'quick.gif' height = 100 width = 100>");**

**}**

**</script>**

## **Controlling Multimedia**

Let us take one real example which works in almost all the browsers −

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

**<!--**

**function play() {**

**if (!document.demo.IsPlaying()) {**

**document.demo.Play();**

**}**

**}**

**function stop() {**

**if (document.demo.IsPlaying()) {**

**document.demo.StopPlay();**

**}**

**}**

**function rewind() {**

**if (document.demo.IsPlaying()) {**

**document.demo.StopPlay();**

**}**

**document.demo.Rewind();**

**}**

**//-->**

**</script>**

**JAVASCRIPT – Debugging**

* **Error Messages in IE**

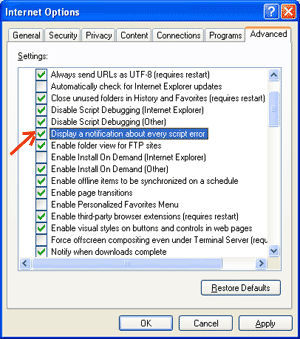
The most basic way to track down errors is by turning on error information in your browser. By default, Internet Explorer shows an error icon in the status bar when an error occurs on the page.



Double-clicking this icon takes you to a dialog box showing information about the specific error that occurred.

Since this icon is easy to overlook, Internet Explorer gives you the option to automatically show the Error dialog box whenever an error occurs.

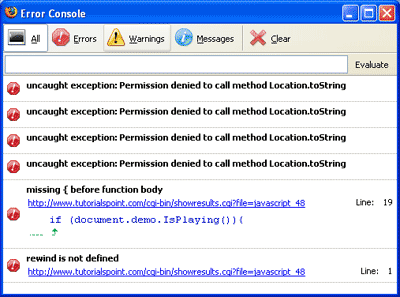
To enable this option, select Tools → Internet Options → Advanced tab. and then finally check the "Display a Notification About Every Script Error" box option as shown below −



* **Error Messages in Firefox or Mozilla**

Other browsers like Firefox, Netscape, and Mozilla send error messages to a special window called the JavaScript Console or Error Console. To view the console, select Tools → Error Console or Web Development.

Unfortunately, since these browsers give no visual indication when an error occurs, you must keep the Console open and watch for errors as your script executes.



* **Error Notifications**

Error notifications that show up on Console or through Internet Explorer dialog boxes are the result of both syntax and runtime errors. These error notifications include the line number at which the error occurred.

If you are using Firefox, then you can click on the error available in the error console to go to the exact line in the script having error.

* **How to Debug a Script**

One way to check your JavaScript code for strange bugs is to run it through a program that checks it to make sure it is valid and that it follows the official syntax rules of the language. These programs are called validating parsers or just validators for short, and often come with commercial HTML and JavaScript editors.

The most convenient validator for JavaScript is Douglas Crockford's JavaScript Lint, which is available for free at [Douglas Crockford's JavaScript Lint](http://www.jslint.com/).

Simply visit that web page, paste your JavaScript (Only JavaScript) code into the text area provided, and click the jslint button. This program will parse through your JavaScript code, ensuring that all the variable and function definitions follow the correct syntax. It will also check JavaScript statements, such as if and while, to ensure they too follow the correct format

### Add Debugging Code to Your Programs

You can use the alert() or document.write() methods in your program to debug your code. For example, you might write something as follows −

**var debugging = true;**

**var whichImage = "widget";**

**if( debugging )**

**alert( "Calls swapImage() with argument: " + whichImage );**

**var swapStatus = swapImage( whichImage );**

**if( debugging )**

**alert( "Exits swapImage() with swapStatus=" + swapStatus );**

By examining the content and order of the alert() as they appear, you can examine the health of your program very easily.

### Use a JavaScript Debugger

A debugger is an application that places all aspects of script execution under the control of the programmer. Debuggers provide fine-grained control over the state of the script through an interface that allows you to examine and set values as well as control the flow of execution.

Once a script has been loaded into a debugger, it can be run one line at a time or instructed to halt at certain breakpoints. Once execution is halted, the programmer can examine the state of the script and its variables in order to determine if something is amiss. You can also watch variables for changes in their values.

* **Useful Tips for Developers**

You can keep the following tips in mind to reduce the number of errors in your scripts and simplify the debugging process −

* Use plenty of comments. Comments enable you to explain why you wrote the script the way you did and to explain particularly difficult sections of code.
* Always use indentation to make your code easy to read. Indenting statements also makes it easier for you to match up beginning and ending tags, curly braces, and other HTML and script elements.
* Write modular code. Whenever possible, group your statements into functions. Functions let you group related statements, and test and reuse portions of code with minimal effort.
* Be consistent in the way you name your variables and functions. Try using names that are long enough to be meaningful and that describe the contents of the variable or the purpose of the function.
* Use consistent syntax when naming variables and functions. In other words, keep them all lowercase or all uppercase; if you prefer Camel-Back notation, use it consistently.
* Test long scripts in a modular fashion. In other words, do not try to write the entire script before testing any portion of it. Write a piece and get it to work before adding the next portion of code.
* Use descriptive variable and function names and avoid using single-character names.
* Watch your quotation marks. Remember that quotation marks are used in pairs around strings and that both quotation marks must be of the same style (either single or double).
* Watch your equal signs. You should not use a single = for comparison purposes.
* Declare variables explicitly using the var keyword.

**JAVASCRIPT – Image Map**

You can use JavaScript to create client-side image map. Client-side image maps are enabled by the usemap attribute for the <img /> tag and defined by special <map> and <area> extension tags.

The image that is going to form the map is inserted into the page using the <img /> element as normal, except that it carries an extra attribute called usemap. The value of the usemap attribute is the value of the name attribute on the <map> element, which you are about to meet, preceded by a pound or hash sign.

The <map> element actually creates the map for the image and usually follows directly after the <img /> element. It acts as a container for the <area /> elements that actually define the clickable hotspots. The <map> element carries only one attribute, the name attribute, which is the name that identifies the map. This is how the <img /> element knows which <map> element to use.

The <area> element specifies the shape and the coordinates that define the boundaries of each clickable hotspot.

The following code combines image maps and JavaScript to produce a message in a text box when the mouse is moved over different parts of an image.

**<html>**

**<head>**

**<title>Using JavaScript Image Map</title>**

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

**<!--**

**function showTutorial(name) {**

**document.myform.stage.value = name**

**}**

**//-->**

**</script>**

**</head>**

**<body>**

**<form name = "myform">**

**<input type = "text" name = "stage" size = "20" />**

**</form>**

**<!-- Create Mappings -->**

**<img src = "/images/usemap.gif" alt = "HTML Map" border = "0" usemap = "#tutorials"/>**

**<map name = "tutorials">**

**<area shape="poly"**

**coords = "74,0,113,29,98,72,52,72,38,27"**

**href = "/perl/index.htm" alt = "Perl Tutorial"**

**target = "\_self"**

**onMouseOver = "showTutorial('perl')"**

**onMouseOut = "showTutorial('')"/>**

**<area shape = "rect"**

**coords = "22,83,126,125"**

**href = "/html/index.htm" alt = "HTML Tutorial"**

**target = "\_self"**

**onMouseOver = "showTutorial('html')"**

**onMouseOut = "showTutorial('')"/>**

**<area shape = "circle"**

**coords = "73,168,32"**

**href = "/php/index.htm" alt = "PHP Tutorial"**

**target = "\_self"**

**onMouseOver = "showTutorial('php')"**

**onMouseOut = "showTutorial('')"/>**

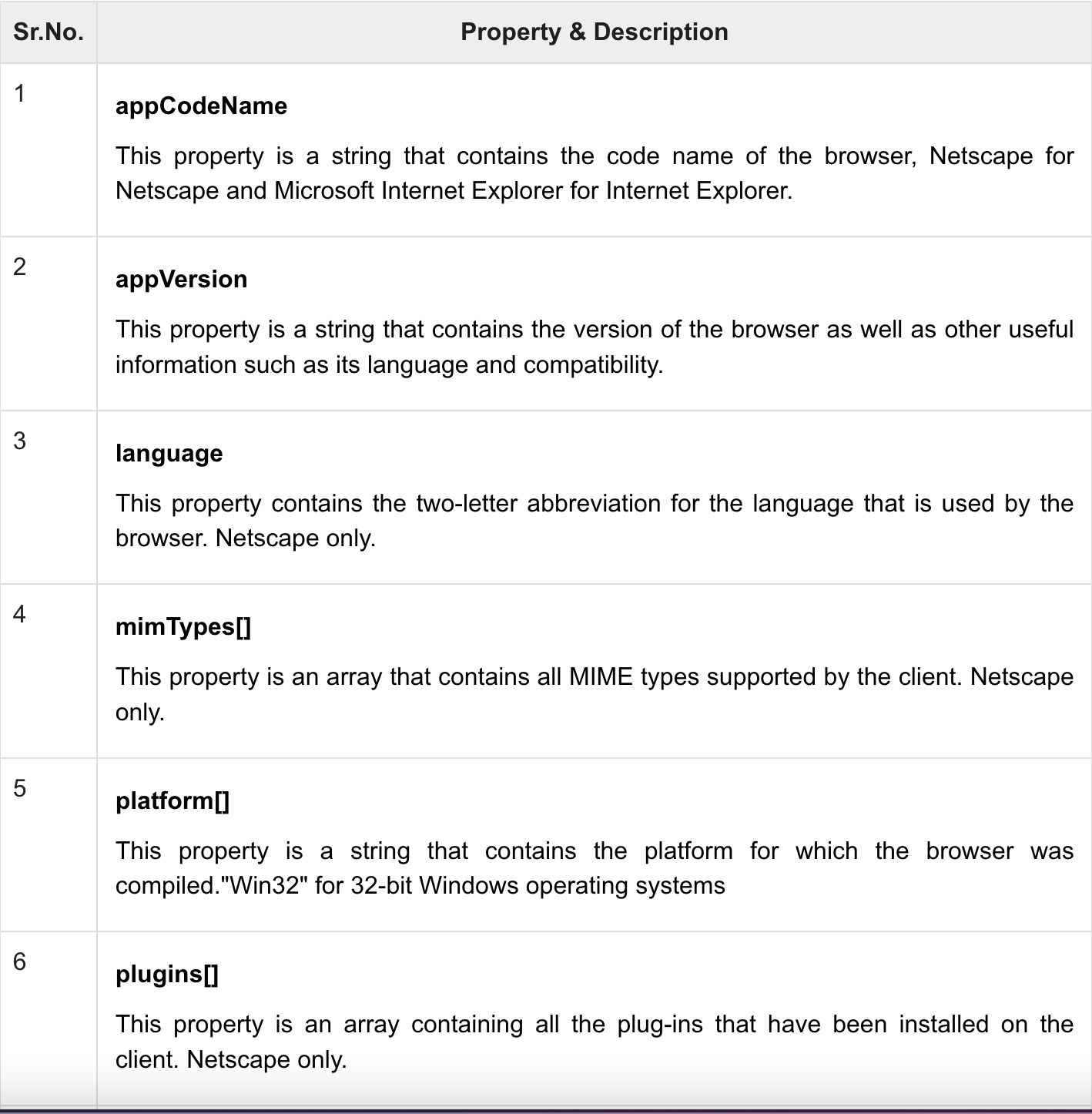
**</map>**

**</body>**

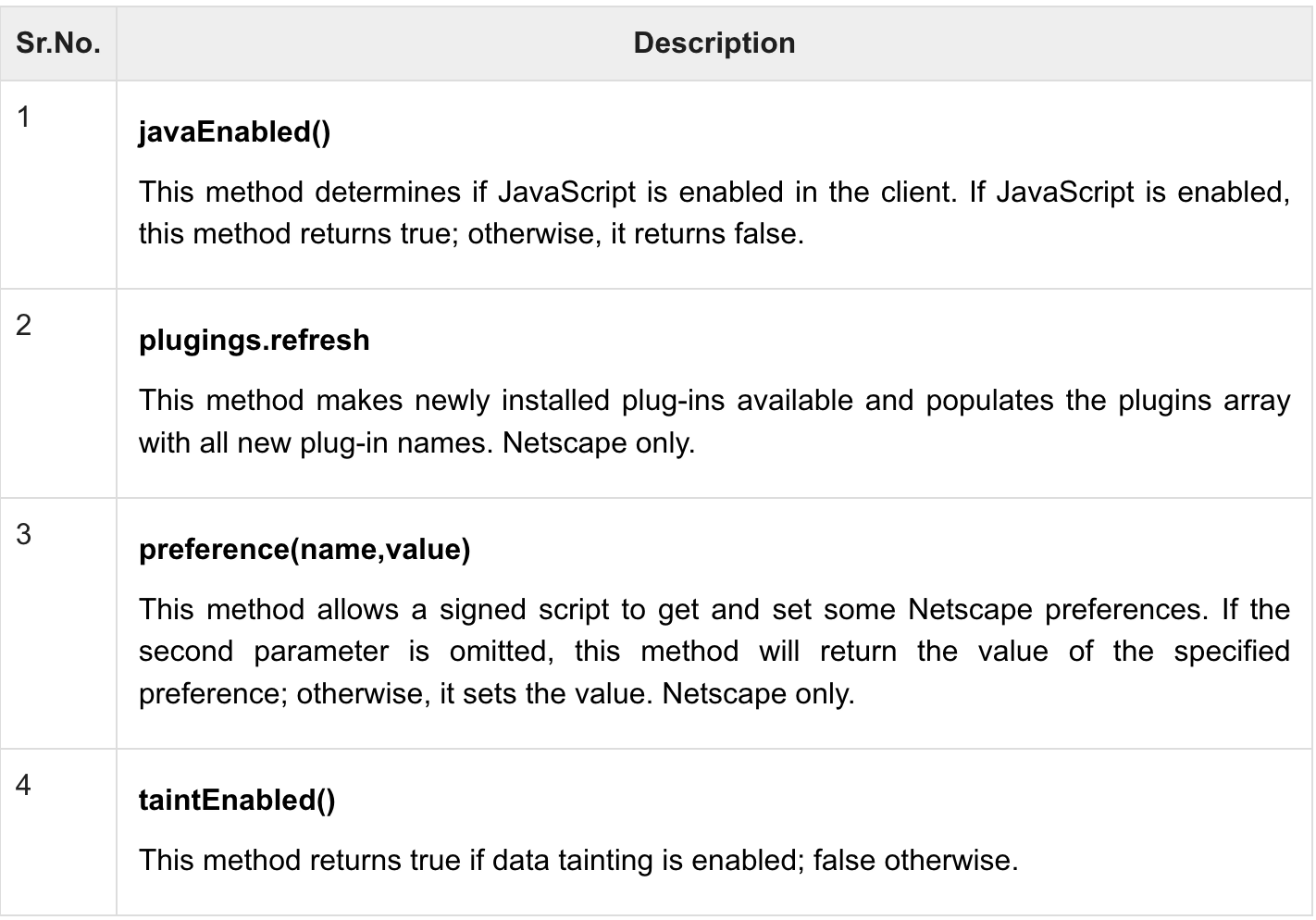
**</html>**

**JAVASCRIPT – Browsers**

* **Navigator Properties**

There are several Navigator related properties that you can use in your Web page. The following is a list of the names and descriptions of each.

* **Navigator Methods**

There are several Navigator-specific methods. Here is a list of their names and descriptions.

* **Browser Detection**

There is a simple JavaScript which can be used to find out the name of a browser and then accordingly an HTML page can be served to the user.

**<html>**

**<head>**

**<title>Browser Detection Example</title>**

**</head>**

**<body>**

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

**<!--**

**var userAgent = navigator.userAgent;**

**var opera = (userAgent.indexOf('Opera') != -1);**

**var ie = (userAgent.indexOf('MSIE') != -1);**

**var gecko = (userAgent.indexOf('Gecko') != -1);**

**var netscape = (userAgent.indexOf('Mozilla') != -1);**

**var version = navigator.appVersion;**

**if (opera) {**

**document.write("Opera based browser");**

**// Keep your opera specific URL here.**

**} else if (gecko) {**

**document.write("Mozilla based browser");**

**// Keep your gecko specific URL here.**

**} else if (ie) {**

**document.write("IE based browser");**

**// Keep your IE specific URL here.**

**} else if (netscape) {**

**document.write("Netscape based browser");**

**// Keep your Netscape specific URL here.**

**} else {**

**document.write("Unknown browser");**

**}**

**// You can include version to along with any above condition.**

**document.write("<br /> Browser version info : " + version );**

**//-->**

**</script>**

**</body>**

**</html>**

**JAVASCRIPT – Classes**

* **Inheritance**

To create a class inheritance, use the extends keyword.

A class created with a class inheritance inherits all the methods from another class:

**class Car {**

**constructor(brand) {**

**this.carname = brand;**

**}**

**present() {**

**return 'I have a ' + this.carname;**

**}**

**}**

**class Model extends Car {**

**constructor(brand, mod) {**

**super(brand);**

**this.model = mod;**

**}**

**show() {**

**return this.present() + ', it is a ' + this.model;**

**}**

**}**

**let myCar = new Model("Ford", "Mustang");**

**document.getElementById("demo").innerHTML = myCar.show();**

* **Static**

Static class methods are defined on the class itself.

You cannot call a static method on an object, only on an object class.

**class Car {**

**constructor(name) {**

**this.name = name;**

**}**

**static hello() {**

**return "Hello!!";**

**}**

**}**

**let myCar = new Car("Ford");**

**// You can call 'hello()' on the Car Class:**

**document.getElementById("demo").innerHTML = Car.hello();**

**// But NOT on a Car Object:**

**// document.getElementById("demo").innerHTML = myCar.hello();**

**// this will raise an error.**

**JAVASCRIPT – ES6 (Features)**

* **The let keyword**
* **The const keyword**
* **Arrow Functions**
* **For/of**
* **Map Objects**
* **Set Objects**
* **Classes**
* **Promises**
* **Symbol**
* **Default Parameters**
* **Function Rest Parameter**
* **String.includes()**
* **String.startsWith()**
* **String.endsWith()**
* **Array.from()**
* **Array keys()**
* **Array find()**
* **Array findIndex()**
* **New Math Methods**
* **New Number Properties**
* **New Number Methods**
* **New Global Methods**
* **Object entries**
* **JavaScript Modules**