**Difference between null and undefined in javaScript?**

In JavaScript, undefined means a variable has been declared but has not yet been assigned a value, such as:

var TestVar;

alert(TestVar); //shows undefined

alert(typeof TestVar); //shows undefined

null is an assignment value. It can be assigned to a variable as a representation of no value:

var TestVar = null;

alert(TestVar); //shows null

alert(typeof TestVar); //shows object

From the preceding examples, it is clear that undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object.

null === undefined // false

null == undefined // true

null === null // true

and

null = 'value' // ReferenceError

undefined = 'value' // 'value'

**Difference between =, == and === in javaScript?**

By using = you assign a value to something.

**x = 1 //x now equals 1  
x = 2 //x now equals 2**By using == you check if something is equal to something else. This is not strict

**x == 1 //is x equal to 1? (False)  
x == 2 //is x equal to 2? (True)  
true == 1 //does the boolean value of true equal 1? (True)**By using === you check if something is equal to something else. This is also strict.

**x === 1 //is x equal to 1? (False)**

**x === 2 //is x equal to 2? (True)  
true === 1 //does the boolean value of true equal 1? (False)**

**What happens if we declare variable without var?**

If you declare a variable, without using "var", the variable always becomes GLOBAL.

**What is use of let in javaScript?**

let allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. This is unlike the [var](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var) keyword, which defines a variable globally, or locally to an entire function regardless of block scope.

Variables declared by **let** have as their scope the block in which they are defined, as well as in any contained sub-blocks. In this way, **let** works very much like **var**. The main difference is that the scope of a **var** variable is the entire enclosing function:

function varTest() {  
 var x = 1;  
 if (true) {  
 var x = 2; // same variable!  
 console.log(x); // 2  
 }  
 console.log(x); // 2  
}  
  
function letTest() {  
 let x = 1;  
 if (true) {  
 let x = 2; // different variable  
 console.log(x); // 2  
 }  
 console.log(x); // 1  
}

**What is use of DOCTYPE in html?**

* The <!DOCTYPE> declaration must be the very first thing in your HTML document, before the <html> tag.
* The <!DOCTYPE> declaration is not an HTML tag; it is an instruction to the web browser about what version of HTML the page is written in.
* In HTML 4.01, the <!DOCTYPE> declaration refers to a DTD, because HTML 4.01 was based on SGML. The DTD specifies the rules for the markup language, so that the browsers render the content correctly.
* HTML5 is not based on SGML, and therefore does not require a reference to a DTD.
* There are three different <!DOCTYPE> declarations in HTML 4.01. In HTML5 there is only one: <!DOCTYPE html>

**What is type of null in javaScript?**

In JavaScript null is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the datatype of null is an object.

**How can we declare string in javaScript?**

Strings are written with quotes. You can use single or double quotes: both are equal

Var a=’a’;

Var b=”a”;

a==b //true

a===b /true

**What is javaScript hoisting?**

Hoisting is JavaScript's default behavior of moving declarations to the top.

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

Example

x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element

elem.innerHTML = x; // Display x in the element

var x; // Declare x

**What is strict-mode in javaScript?**

* "use strict"; Defines that JavaScript code should be executed in "strict mode".
* The "use strict" directive is new in JavaScript 1.8.5 (ECMAScript version 5).
* It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.
* The purpose of "use strict" is to indicate that the code should be executed in "strict mode".
* With strict mode, you can not, for example, use undeclared variables.
* Strict mode is declared by adding "use strict"; to the beginning of a script or a function.
* Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

**Example**

**"use strict";**

**x = 3.14;// This will cause an error because x is not declared**

**What is a closure?**

A closure is an inner function that has access to the outer (enclosing) function’s variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables. The inner function has access not only to the outer function’s variables, but also to the outer function’s parameters. Note that the inner function cannot call the outer function’s arguments object, however, even though it can call the outer function’s parameters directly. You create a closure by adding a function inside another function.

A Basic Example of Closures in JavaScript:

function showName (firstName, lastName) {

​var nameIntro = "Your name is ";

// this inner function has access to the outer function's variables, //including the parameter​

​

function makeFullName () {

​ return nameIntro + firstName + " " + lastName;

}

​ ​return makeFullName ();

}

​showName ("Michael", "Jackson"); // Your name is Michael Jackson

### **What is noConflict in Jquery?**

As you already know; jQuery uses the **$** sign as a shortcut for jQuery.

There are many other popular JavaScript frameworks like: Angular, Backbone, Ember, Knockout, and more.

**What if other JavaScript frameworks also use the $ sign as a shortcut?**

If two different frameworks are using the same shortcut, one of them might stop working.

The jQuery team have already thought about this, and implemented the noConflict() method.

The noConflict() method releases the hold on the $ shortcut identifier, so that other scripts can use it.

You can of course still use jQuery, simply by writing the full name instead of the shortcut:

$.noConflict();

jQuery(document).ready(function(){

jQuery("button").click(function(){

jQuery("p").text("jQuery is still working!");

});

});

You can also create your own shortcut very easily. The noConflict() method returns a reference to jQuery, that you can save in a variable, for later use. Here is an example:

var jq = $.noConflict();

jq(document).ready(function(){

jq("button").click(function(){

jq("p").text("jQuery is still working!");

});

});

**What is callback function in javaScript?**

JavaScript statements are executed line by line. However, with effects, the next line of code can be run even though the effect is not finished. This can create errors.

To prevent this, you can create a callback function.

A callback function is executed after the current effect is finished.

Typical syntax: $(selector).hide(speed,callback);

Examples

The example below has a callback parameter that is a function that will be executed after the hide effect is completed:

**$("button").click(function(){**

**$("p").hide("slow", function(){**

**alert("The paragraph is now hidden");**

**});**

**});**

The example below has no callback parameter, and the alert box will be displayed before the hide effect is completed:

$("button").click(function(){

$("p").hide(1000);

alert("The paragraph is now hidden");

});

**What is difference between display:none & visibility:hidden?**

**display:none** removes the element from the normal flow of the page, allowing other elements to fill in.

**visibility:hidden** leaves the element in the normal flow of the page such that is still occupies space.

**What are the jquery selectors?**

jQuery selectors allow you to select and manipulate HTML element(s).

jQuery selectors are used to "find" (or select) HTML elements based on their **name, id, classes, types, attributes, values of attributes and much more**. It's based on the existing [CSS Selectors](https://www.w3schools.com/cssref/css_selectors.asp), and in addition, it has some own custom selectors.

All selectors in jQuery start with the dollar sign and parentheses: **$().**

**How to create dom object?**

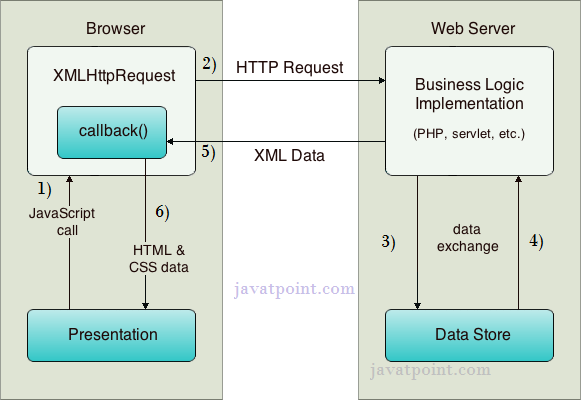
We can create any dom element by calling createElement method on document object

**If we want to create button we can do as follows:**

**var btn = document.createElement("BUTTON");**

**How AJAX works?**

AJAX communicates with the server using XMLHttpRequest object. Let's try to understand the flow of ajax or how ajax works by the image displayed below.

****

As you can see in the above example, XMLHttpRequest object plays a important role.

1. User sends a request from the UI and a javascript call goes to XMLHttpRequest object.
2. HTTP Request is sent to the server by XMLHttpRequest object.
3. Server interacts with the database using JSP, PHP, Servlet, ASP.net etc.
4. Data is retrieved.
5. Server sends XML data or JSON data to the XMLHttpRequest callback function.
6. HTML and CSS data is displayed on the browser.

**What is the difference between jQuery.ajax(), jQuery.get() and jQuery.post() ?**

**jQuery.ajax()**

**jQuery.ajax({**

**url: 'test.txt',**

**dataType: 'text',**

**type: “GET”,**

**success: function(data) {**

**console.log(data);**

**}});**

We just pass an object into the jQuery.ajax() method. That object has four properties: “url,” “dataType,” “type,” and “success”, and here are the details for each property:

* **url:** This is the URL of the file that you want to grab via your ajax call.
* **dataType:** This determines how the return data will be treated (i.e. pure text, html, XML, etc.).
* **type:** This the the request type. Choose “GET” or “POST”. This is actually optional; if you omit it, jQuery will default to “GET”.
* **success:** This is a callback function that is fired after a successful http request has completed. The first argument to the function is the data returned from the server. There are other arguments that can be passed in as well.

**jQuery.get()**

**jQuery.get('test.txt',function(data){**

**console.log(data)**

**},'text');**

The .get() method, which is a kind of shorthand for jQuery.ajax(). When using jQuery.get(), instead of passing in an object, you pass in arguments. At minimum, you’ll need the first two arguments: the URL of the file you want to retrieve (i.e. ‘test.txt’), and a success callback. In this example, we also passed in a third argument: ‘text,’ which told jQuery that we wanted to treat the return message as text.

The jQuery.get() method is recommended when you want to make a quick and dirty Ajax call and you are sure it will be a GET.

**jQuery.load()**

**$( "#result" ).load( "ajax/test.html", function() {**

**alert( "Load was performed." );**

**});**

.load() is similar to $.get() but adds functionality which allows you to define where in the document the returned data is to be inserted. Therefore really only usable when the call only will result in HTML. It is called slightly differently than the other, global, calls, as it is a method tied to a particular jQuery-wrapped DOM element. Therefore, one would do: $('#divWantingContent').load(...)

**jQuery.post()**

**jQuery.post('test.txt',function(data){**

**console.log('post response-&gt; ' + data)**

**},'text');**

This method is like using jQuery.ajax(), and specifying a “type” of “POST.” So, this method is recommended if you need to make a quick Ajax call via POST, and don’t need to make a lot of configuration decisions.

**How to achieve inheritance in javaScript?**

**What is promise in javaScript?**

A Promise object represents a value that may not be available yet, but will be resolved at some point in the future. It allows you to write asynchronous code in a more synchronous fashion. For example, if you use the promise API to make an asynchronous call to a remote web service you will create a Promise object which represents the data that will be returned by the web service in future. The caveat being that the actual data is not available yet. It will become available when the request completes and a response comes back from the web service. In the meantime the Promise object acts like a proxy to the actual data. Furthermore, you can attach callbacks to the Promise object which will be called once the actual data is available.

Ex.

if (window.Promise) {  
 console.log('Promise found');  
  
 var promise = new Promise(function(resolve, reject) {  
 var request = new XMLHttpRequest();  
  
 request.open('GET', 'http://api.icndb.com/jokes/random');  
 request.onload = function() {  
 if (request.status == 200) {  
 resolve(request.response); // we got data here, so resolve the Promise  
 } else {  
 reject(Error(request.statusText)); // status is not 200 OK, so reject  
 }  
 };  
  
 request.onerror = function() {  
 reject(Error('Error fetching data.')); // error occurred, reject the Promise  
 };  
  
 request.send(); //send the request  
 });  
  
 console.log('Asynchronous request made.');  
  
 promise.then(function(data) {  
 console.log('Got data! Promise fulfilled.');  
 document.getElementsByTagName('body')[0].textContent = JSON.parse(data).value.joke;  
 }, function(error) {  
 console.log('Promise rejected.');  
 console.log(error.message);  
 });  
} else {  
 console.log('Promise not available');  
}