**Javascript interview questions – Part 2**

**1.** **Explain how ‘this’ keyword works in JS.**

The keyword 'this' **depends on the current context scope**. Generally spoken, it refers to the object upon which the method is called. The value of ‘this’ is determined by how a function is called.

**Examples:**

***Global context***

In the browser (global env) *this* refers to the window object

|  |
| --- |
| **function this\_scope() {**  **console.log(this);**  **return this === window**  **}**  **this\_scope(); // window object, true**  **console.log(this.document === document); // true** |

***Function context***

// inside of a function

|  |
| --- |
| **function do\_smth() {**  **this.style.color = '#cc0000';**  **}**  **do\_smth(); // here this refers to the global window object**  **element.onclick = do\_smth(); // here this refers to the element**  **element.addEventListener('click',do\_smth); // refers to the element** |

// here it refers to an instance of an object

|  |
| --- |
| **function Point(x,y) {**  **this.x = x;**  **this.y = y;**  **}**  **var pt = new Point(1,2);**  **console.log(pt); // {x:1,y:2}** |

// If you use the strict mode (ECMAscript5) this evaluates to 'undefined'

|  |
| --- |
| **function add2(x,y) {**  **"use strict";**  **console.log(this); // returns the global object**  **return x+y;**  **}**  **add2(2,3); // undefined, 5** |

// If you nest functions inside other methods -> this set to the global object

|  |
| --- |
| **Point.prototype = {**  **move: function(x,y) {**  **this.x += x;**  **this.y += y;**  **// nested functions**  **function checkBounds() {**  **// here 'this' refers to the inner function**  **if (this.x>100) {**  **console.log("this won't get executed");**  **}**  **}**  **checkBounds();**  **}**  **};**  **var pt1 = new Point(0,0);**  **pt1.move(105,100);** |

// Reference this by using another variable

|  |
| --- |
| **Point.prototype = {**  **move: function(x,y) {**  **var self = this;**  **this.x += x;**  **this.y += y;**    **function checkBounds() {**  **if (self.x>100) {**  **console.log("this will now be executed");**  **}**  **}**  **checkBounds();**  **}**  **};**  **var pt2 = new Point(0,0);**  **pt2.move(105,100);** |

<http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/>

<http://www.quirksmode.org/js/this.html>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>

<http://toddmotto.com/understanding-the-this-keyword-in-javascript/>

**2. What is a closure and how would you use one?**

When a **function is nested inside another function** and it accesses a variable from its parent's scope → a closure is created. It's a function to which the variables of the surrounding context are bound by reference; it basically preserves the state of the outer context.

Examples:

|  |
| --- |
| **function sayHello(name) {**  **var phrase = “Hello “ + name;**  **console.log(“From sayHello: “ + phrase);**  **var sayHi = function() {**  **console.log(“From sayHi: “ + phrase);**  **};**  **return sayHi;**  **}**  **var talk = sayHello(“Nena”); // “From sayHello: Hello Nena”**  **talk(); // “From sayHi: Hello Nena”** |

Example of a click handler:

|  |
| --- |
| **function Test() {  var self=this;  $('.list-wrapper').on('click', function(el) {  console.log(this); // returns the jQuery element  console.log(self); // returns the Test() object  });**  **}** |

Another typical example of a closure is when you want to attach an event listener in a loop. For example, you want to loop over an array of DOM elements and assign an event handler to each one of them. The simplest solution (and the one that doesn’t work) would be to do the following:

|  |
| --- |
| **for (var ind=0; ind<elements.length; ind++) {**  **elements[ind].onclick = function() {**  **alert(ind);**  **}**  **}** |

The value of this alert will be whatever the length of the elements is - the last value of ‘ind’. Why? Because anonymous functions that we define as event handlers ‘inherit’ the variable ‘ind’ from the outer scope, not from the for loop. However, by the time the click handlers are executed, for loop has completed and the value of ‘ind’ is set to elements.length. The problem is that anonymous functions (event handlers) don’t create a new scope for ‘ind’ until they are executed. There are a couple of ways to get around this.

1. Create a new scope for the current value of ***ind*** by executing a function inside of the loop

|  |
| --- |
| **for (var ind=0; ind<elements.length; ind++) {**  **elements[ind].onclick = (function(value) {**  **return alert(value);**  **})(ind)**  **}** |

Here an intermediate function is created that takes 1 argument (value) and passes it to the inner function. This function is then immediately called with ***ind*** as an argument. Calling this function creates a new variable scope, because variable scope is created at execution time.

2. You can put each event handler into a separate function and invoke them in a loop.

|  |
| --- |
| **function handler(i) {**  **document.getElementById(i).onclick = function() {**  **alert(i);**  **}**  **}**  **for (var ind=0; ind<elements.length; ind++) {**  **handler(ind);**  **}** |

This feature—being able to reference a specific instance of local variables in an enclosing function—is called ***closure***. A function that “closes over” some local variables is called *a* closure. This behavior not only frees you from having to worry about lifetimes of variables but also allows for some creative use of function values.

<http://javascriptissexy.com/understand-javascript-closures-with-ease/>

<http://mark-story.com/posts/view/picking-up-javascript-closures-and-lexical-scoping>

<http://jondavidjohn.com/javascript-closure-explained-using-events/>

<http://www.mennovanslooten.nl/blog/post/62>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

**3. What is a lexical scope in js?**

Lexical scope is a term that refers to a function remembering and preserving its state between and after execution. It describes how variables are resolved in nested functions. It’s called static scope or closure.

In short, each local scope can also see all the local scopes that contain it. The set of variables visible inside a function is determined by the place of that function in the program text. All variables from blocks around a function’s definition are visible—meaning both those in function bodies that enclose it and those at the top level of the program. This approach to variable visibility is called lexical scoping.

<http://www.smashingmagazine.com/2009/08/01/what-you-need-to-know-about-javascript-scope/>

<https://javascriptweblog.wordpress.com/2010/10/25/understanding-javascript-closures/>

<http://eloquentjavascript.net/03_functions.html#p_PCO8aj4Plj>

**4. What is the difference between .call and .apply?**

JS allows us to pass functions around, then execute them in a different context by taking advantage of ***call*** and ***apply***.

From ECMAScript:

**Function.prototype.apply(thisArg,argArray)**

→ apply called on an object function with arguments: thisArg is **this** value (refers to the current object) and argArray is an array like an object (which makes it much more flexible), specifying the arguments with which function should be called.

NOTE: according to ECMAScript5 spec the arguments can be a generic array-like object instead of an array.

**Function.prototype.call(this.Arg[,arg1[arg2...]])**

→ called on an object function with arguments: thisArg is this value (refers to the calling object) and arg1,arg2... is the argument list.

**Differences:**

- with call, you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.

- call() accepts an argument list, while apply() accepts a single array of args.

Examples:

// Use call and apply methods to resolve 'this' to its intended scope

|  |
| --- |
| var radius = 100;  var circle = {  radius: 5,  inner: {  radius: 30,  getRadius: function() {  return "Radius: " + this.radius;  }  }  };  var mycircle = point.inner.getRadius;  console.log(mycircle()); // Radius: 100, global scope (window.radius)  console.log(mycircle.apply(circle)); // Radius: 5, circle scope (circle.radius)  console.log(mycircle.apply(circle.inner)); //Radius:30,inner scope (circle.inner.radius) |

// Another example

|  |
| --- |
| function Point(x,y) {  this.x = x;  this.y = y;  }  Point.prototype = {  move: function(x,y) {  this.x += x;  this.y += y;    function checkBounds() {  if (this.x>100) {  console.log("this will get executed");  }  }  // here this refers to an instance of Point  checkBounds.call(this);  // checkBounds.apply(this);  }  };  var pt = new Point(1,2);  pt.move(105,100);    Point.prototype = {  move: function(x,y) {  this.x += x;  this.y += y;    function checkBounds(min,max) {  if (this.x>min || this.x<max) {  console.log("this will get executed again");  }  }  // here this refers to an instance of Point  **checkBounds.call(this,0,100);**  **checkBounds.apply(this,[0,100]);**  }  };  var pt1 = new Point(2,3);  pt1.move(50,60); |

**5.** **Explain Function.prototype.bind**

It's a function that allows you to change the context of another function. Whatever it’s defined as *thisArg* will be used as a new context to call a function upon it.

Function.prototype.bind(thisArg [,arg1[,arg2,...]])

→ it takes one or more args and returns a new function object

→ it creates a new function (a bound function) with the same function body as the function it is being called on (the bound function's target function) with this value bound to the first argument of bind, which cannot be overridden.

*NOTE*: Function objects created using Function.prototype.bind do not have a prototype property.

→ bind returns a function that when later executed will have the correct context set, while apply/call will call a function immediately ??!!

Example:

Create a bound function: no matter how it is called, it is called with a particular this value. Extract a function from an object, then bind it.

|  |
| --- |
| var x = 9;  var module = {  x: 81,  getX: function() {  return this.x;  }  }  console.log(module.getX()); //81  var get = module.getX;  console.log(get()); // 9, 'this' refers to the global(window) object  var boundgetX = get.bind(module);  console.log(boundgetX()); // 81 |

**6.** **Can you explain how inheritance works in JS? (prototypal inheritance)**

Object prototypes:

→ object prototype is JS way of sharing implementation across similar objects

→ prototype based system → no classes, all objects are created by adding properties and methods to an empty object or by cloning an existing one.

→ prototypes are implemented using the *prototype* property of construction function → any property or method that's added to constructor prototype becomes a part of every object created by that function

→ prototype is a template for objects created by constructor. It contains things that should be the same among all objects created by constructor

→ JS objects are dynamic, they can be changed after created.

Example:

|  |
| --- |
| function Pet(name, species, hello) {  this.name = name;  this.species = species;  this.hello = hello;  }  Pet.prototype.sayHello = function() {  console.log(this.hello);  }  // when this object is created, it maintains an invisible link back to Pet function that created it.  var ralf = new Pet('ralf','dog','bark');  ralf.sayHello(); //'bark' |

// the property resolution of prototype happens when it is called, any changes made to the prototype property will be reflected in all objects already created by constructor.

// when sayHello() is called, JS first looks in the constructor for this method, then in the prototype.

NOTE: when Pet function is created, its prototype property is automatically set to an empty object

***Prototype chaining:***

|  |
| --- |
| // Cat constructor  function Cat(name,hello,breed) {  this.name = name;  this.hello = hello;  this.species = species;  this.breed = breed;  }  // assign Pet object to the cat's prototype → chaining, inheritance  Cat.prototype = new Pet();  var rufus = new Cat('rufus','miaow','maine coin');  rufus.sayHello(); // invoke the method from Pet object |

→ used to build new types of objects based on existing ones

→ it's possible to create a function's prototype using another constructor function – all of the properties and methods from constructor function's prototype are automatically added to the new prototype object (inheritance).

|  |
| --- |
| // add another method to the prototype property  Cat.prototype.sleep = function() {  console.log("zzzzzz");  }  // overwrite sayHello method from Pet  Cat.prototype.sayHello = function() {  console.log(this.hello + "from a cat");  } |

Function resolution:

→ first check an object itself

rufus.sleep() { console.log(“zzzzz”);}

→ then check the constructor of the rufus, which is Cat

Cat.prototype.sleep = function() { console.log(“zzzzz”); }

→ if not found, walk up the chain to the Pet constructor (constructor function used to create prototype property)

Pet.prototype.sleep = function() { console.log(“zzzz”);}

<http://helephant.com/2009/01/18/javascript-object-prototype/>

<http://helephant.com/2009/08/17/javascript-prototype-chaining/>

<http://wildlyinaccurate.com/understanding-javascript-inheritance-and-the-prototype-chain/>

<http://www.crockford.com/javascript/inheritance.html>

**7. Explain event delegation.**

Event delegation allows you to avoid adding event listeners to specific nodes → event listener is added to the parent. It makes use of two JS features:

- event bubbling

- target element

**Bubbling** → when event is triggered on an element (mouseclick on button), the same event is triggered on all ancestors, it bubbles up from originating element to the top of the DOM tree.

Target element → the originating element.

Example: instead of adding event listener to each child element, you add it to <ul> parent element. When the event from child bubbles to UL, you check the *event's object target property* to gain reference to actual clicked node.

<ul>

<li> … </li>

<li> … </li>

…..

</ul>

|  |
| --- |
| document.getElementById("parent-list").addEventListener("click",function() {  // e.target is clicked element  if (e.target && e.target.nodeName == "LI") {  // list item found  console.log("List item: " + e.target.ID + " was clicked");  }  } |

<http://davidwalsh.name/event-delegate>

<http://stackoverflow.com/questions/1687296/what-is-dom-event-delegation>

**8. Difference between window.onload and document ready event**

***window.onload*** → will execute code **when browser has loaded the DOM tree and all other resources** (images, objects, etc.). \*Not cross-browser compatible.

***Document ready*** → **will execute code when only the DOM tree has been built**, without having to wait for images to load. You can start scripting against DOMmuch quicker. \*using jquery document.ready() method works on all browsers.

<http://stackoverflow.com/questions/3698200/window-onload-vs-document-ready>

**9.** **Difference between undefined, undeclared & null value**

***Undefined***→ variable declared, but not assigned a value.

>typeof(undefined) // undefined, its a type itself

***Undeclared*** → when you use a variable without declaration, it is implicitly declared as global.

***Null***→ only set up programmatically, represents intentional absence of any object value.

>typeof(null) // object

<http://lucybain.com/blog/2014/null-undefined-undeclared/>

**10.** **Difference between synchronous and asynchronous JS request**

***Synchronous*** → blocks the JS engine until the interaction with the server is complete. User can't click away, cancel request or go to another tab...

***Asynchronous*** → doesn't block the page. Communication with server asynchronous (in the background), without interfering with the display and behavior of the existing page.

<https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests>

**11. What is a typical use case for an anonymous function?**

- if no name is needed, don't add a name to whatever namespace you are in

- they are declared inline and they can access variables in the parent scope (closure)

- code is more self-contained and readable

- self-invoked function: an anonymous function that’s ran immediately after being created

- to create a new scope when invoking a function inside of the loop.

Example**:**

(function(name){ console.log("Hello " + name); })("Nena");

<http://stackoverflow.com/questions/1140089/why-do-you-need-to-invoke-an-anonymous-function-on-the-same-line>

<https://sarfraznawaz.wordpress.com/2012/01/26/javascript-self-invoking-functions/>

**12.** **Explain “JS module pattern” and when would you use one/what is the benefit?**

Modules divide programs into clusters of code. It can be used to provide publicly accessible sub-namespaces. Functions inside of modules can be used to create a private, isolated namespace inside of it. It promotes self-containing code and less clutter in the global namespace. Putting pieces of code into modules makes them easy to reuse them.

- singleton pattern (Crockford), only 1 instance of a class created

- anonymous closures

- global import

- module export

The fundamental construct that makes modules possible is a self-invoking anonymous function. All of the code that runs inside of the function lives in a closure, which provides **privacy** and **state** throughout the lifetime of the application.

|  |
| --- |
| (function() {  // some code here  }()); |

Anonymous function can also have parameters - you can pass globals to it and import them into the code.

|  |
| --- |
| (function($, Go) {  // you have access to jQuery and Google here  }(jQuery, Google)); |

Export a module - if you want to declare globals, you can easily do that by exporting them, using the anonymous function’s return value.

|  |
| --- |
| var MODULE =( function(){  // private variable  var private = {};  // private method  function privateMethod() {  // do something  }  my.moduleProperty = 2;  my.moduleMethod = function() {  };  return my;  }()); |

<http://toddmotto.com/mastering-the-module-pattern/>

<http://css-tricks.com/how-do-you-structure-javascript-the-module-pattern-edition/>

<http://eloquentjavascript.net/10_modules.html>

<http://addyosmani.com/resources/essentialjsdesignpatterns/book/>

**13.** **How do you organize your code (module pattern, classical inheritance)**

- have as little JS in HTML as possible

- break the code into logical units, keep them separate

- use a script to concatenate/minify files into a single bundle

- use JS namespace to avoid cluttering up the global namespace

- use front-end MVC frameworks such as Backbone-Marionette, Ember.js, AngularJS

- read about the module pattern here:<http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html>

**14.** **Difference between host objects and native objects**

According to the ECMAscript:

***Native object:*** object in an ECMAscript implementation whose semantics are fully defined by that spec rather than by host environment. NOTE: standard native objects are defined in ECMA spec. Some native objects are built-in; others may be constructed during the course of execution of an ECMAscript program.

For example, Number, String, Boolean, Array, Date ...etc

***Host object:*** object supplied by the host environment to complete the execution environment of ECMAscript. NOTE: any object that is not native is a host object.

For example, when JS code is ran in a web browser, its window object is a host object in that environment.

Built-in object: object supplied by ECMAScript implementation independent of host environment, that is present at the start of the execution of ES program.

NOTE: standard built-in objects are defined in ECMAspec and ES implementation may specify and define others. Every built-in object is a native object. A built-in constructor is a built-in object that is also a constructor.

<http://lucybain.com/blog/2014/host-vs-native/>

<https://programmerinnervoice.wordpress.com/2013/07/22/host-objects-vs-native-objects/>

**15. Difference between:**

**function Person() {}**

**var person = Person();**

**var person = new Person();**

- first one is defining a function Person that can represent a constructor (function declaration)

- second one is invoking a Person() function (factory constructor)

- third one is creating a new Person object

WARNING:

function Point(x,y) { this.x = x; this.y = y;}

var pt1 = new Point(20,30); // this is ok

var pt2 = Point(20,30); // pt2 undefined after the call. Two variables (x,y) added to global scope → because if you call a function without specifying the context, the browser passes in the global object, which is “window”

**16.** **When do you optimize your code?**

Code optimization:

→ strings: in IE6 & 7 concatenation causes major problem with garbage collection performance. In IE8 and Chrome more efficient.

→ build up long strings by passing string builders

→ defining class methods: don't do it in the constructor function, do it this way

baz.Bar = function() {

// constructor body

};

baz.Bar.prototype.foo = function() {

// method body

};

NOTE: now when you create 10000 instances of baz.Bar objects only a single function is ever created for foo (and no closures).

→ place instance variable declaration/initialization on the PROTOTYPE for instance variable with value type init values

foo.Bar = function() {

this.prop3 = [];

}

foo.prototype.prop1 = 4;

foo.prototype.prop2 = true;

foo.prototype.prop4 = “blah”;

→ avoid pitfalls with closures: most common source of memory leak and creating closure significantly slower than creating inner function without a closure.

**17. How do you go about testing in JS?**

There are different frameworks for testing JS; some of the popular ones are Jasmine, Mocha, QUnit. The basic idea is to make your code testable by breaking it into well contained modules and creating methods that can be easily unit-tested. For example, model’s methods are usually tested by calling them with different input arguments and asserting the expected output. For testing the UI elements, there are different plugins such as jasmine-jquery that include HTML fixtures that allow you to load HTML content to be used in tests.

Also frameworks such as JSdom or PhantomJS are used to mock the browser/DOM. You can then perform input actions such as clicking on stuff, and setting the value of input elements and then query the DOM and assert your expectation.

**18.** **When would you use document.write()?**

Document.write after DOM ready causes it to overwrite the existing DOM. It outputs a string directly into the page. It can only output to an open document. If document is closed (page finished loading), it implicitly calls document.open() which wipes the entire document.

This method is a part of legacy DOM.

Resources:

<http://javascript.info/tutorial/document-write#open-document-only>

<http://stackoverflow.com/questions/12376100/document-write-method-questions>

You might create a pop-up window and write some HTML to it with code like this:

|  |
| --- |
| **function hello() {**  **var w = window.open();**  **var d = w.document;**  **d.open(); //open a new document – optional**  **d.write("<h1>Hello world!</h1>");**  **d.close();**  **}** |

If you call the write() method on a document that has already been closed, JavaScript implicitly opens a new HTML document.

**19. What's the difference between feature detection, feature inference and using the UA string?**

***Feature detection*** is used to check if the browser supports a particular feature.

For example, check if the browser supports HTML5 canvas feature:

|  |
| --- |
| **var canvas = document.createElement("canvas");**  **if (canvas.getContext) {**  **// do something**  **}** |

Popular library for feature detection is Modernizr <http://modernizr.com/>

***Feature inference*** checks for a feature just like feature detection, but uses another function/property and it assumes it will also exist

Example:

|  |
| --- |
| **if (document.getElementsByTagName) {**  **element = document.getElementById("test");**  **}** |

*User agent* string is an old technique and shouldn’t be used anymore. It goes something like: if (navigator.userAgent.indexOf(‘asd’)) {}

<http://www.html5rocks.com/en/tutorials/detection/>

<http://learn.jquery.com/code-organization/feature-browser-detection/>

<http://stackoverflow.com/questions/20104930/whats-the-difference-between-feature-detection-feature-inference-and-using-th>

**20. Explain the cross-site HTTP request and the same-origin policy**

Cross-site HTTP requests are HTTP requests that make a request for a resource to a domain different than the domain of the resource. They are subject to restrictions due to security reasons and to the same-origin policy. As an example, an application using XMLHttpRequest could only make HTTP requests to the domain it was loaded from, and not to other domains. Same-origin policy is used as a means to prevent some of the cross-site request forgery attacks. Two pages have the same origin if the protocol, host and port (if specified) are the same.

This issue is commonly solved by sharing data through an API, REST interface.

The way to support the cross-site access control recommended by the W3C is to use Cross-Origin Resource Sharing mechanism (CORS). It works by adding new HTTP headers that allow servers to describe the set of origins that are permitted to read that information using a web browser.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS>

<https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy>

<http://en.wikipedia.org/wiki/Same-origin_policy>

**21. Explain AJAX as much detail as possible.**

Ajax stand for Asynchronous Javascript and XML. It’s a technique used on the client side to make asynchronous request to the server without doing a full page reload. The way it’s usually implemented is using success and error callbacks that get invoked once when the response from server is there.

There are 4 basic HTTP requests: GET, POST, PUT, DELETE you can make using AJAX.

Data can be retrieved from the server using XMLHttpRequest object. XML is not required and the most common data format is JSON. Request also doesn’t need to be asynchronous, despite its name.

There are a couple of steps when implementing AJAX in pure Javascript (though jQuery ajax() is mostly used for that nowadays).

Example:

1. Create an instance of the XMLHttpRequest class (originally introduced by IE as ActiveXObject)

|  |
| --- |
| **var httpRequest;**  **if (window.XMLHttpRequest) {**  **httpRequest = new XMLHttpRequest();**  **} else if (window.ActiveXObject) {**  **httpRequest = new ActiveXObject("Microsoft.XMLHTTP");**  **}** |

1. You need to setup a callback handler that it’s going to be executed once you receive a response from your call.

|  |
| --- |
| httpRequest.onreadystatechange = function() {  // process the server response  if (httpRequest.readyState == 4) {  // response received  } else {  // response not ready yet  }  }  OR  httpRequest.onreadystatechange = handleResponse; |

1. You need to make an actual request by calling open() and send() methods of the HTTP request object

|  |
| --- |
| httpRequest.open('GET', '<http://www.example.com>', true);  httpRequest.send(null); |

As a security feature, you can’t call pages on 3rd party domains (same-origin policy).

<http://en.wikipedia.org/wiki/Ajax_%28programming%29>

<https://developer.mozilla.org/en-US/docs/AJAX/Getting_Started>

**22. Explain JSONP and why is it used.**

JSONP is used for loading JS scripts (data files) from a server from a different domain. It takes advantage of the fact that browsers don’t enforce the same-origin policy on the **<script>** tags. JSONP requests are not dispatched using XMLHttpRequest, instead a <script> tag is created whose source is set to the target URL. This script tag is then added to the DOM (usually in the <head>)

Example:

|  |
| --- |
| **var tag = document.createElement("script");**  **tag.src = "**[**www.otherdomain.com?callback=bar**](http://www.otherdomain.com?callback=bar)**";**  **document.getElementsByTagName("head")[0].appendChild(tag);**  **Response:**  **bar({**  **"foo":"bar";**  **})** |

This function must exist in the global scope at the time the <script> tag is evaluated in the browser (once the request has completed).

<http://en.wikipedia.org/wiki/JSONP>

<http://www.sitepoint.com/jsonp-examples/>

<http://json-p.org/>

<http://www.jquery4u.com/function-demos/index.php?function=jsonp>

<http://stackoverflow.com/questions/3839966/can-anyone-explain-what-jsonp-is-in-layman-terms>

**23. Explain REST concept**

REST stands for Representational State Transfer and in simple words it means invoking a procedure from one machine (server) to another, remote location. Historically:

* CORBA was a standard for doing remote procedure calls
* then EJB replaced it as a newer standard
* SOAP became popular as XML became a big thing

Rather than using those complex mechanisms, simple HTTP is used to make calls between machines. REST is an architecture style for designing networked applications. It relies on stateless, client-server, cacheable communication protocol.

RESTful apps use HTTP requests to post data (create, update), read data (make queries) and delete data. REST uses HTTP for all four CRUD operations. REST service is platform-independent, language-independent, standards-based. They offer no security features, encryption, session management..etc, those can be added by building on top of HTTP.

<http://rest.elkstein.org/>

<http://en.wikipedia.org/wiki/Representational_state_transfer>

**24. Explain “hoisting” and scoping.**

In JS variables are hoisted to the top, regardless of where are they declared. Only the declaration will be hoisted, not initialization. If it’s initialized, the current value at the top of the scope will be initially set to undefined.

One thing to be aware of is that there is a difference in defining a function using ***function declaration*** and using ***function expression***. If you try to call a function defined using function expression, it will result in an error (there is no function with that name) - because the variable gets hoisted and set to undefined. ***In case of the function declaration - the entire function gets hoisted to the top of the scope***. That’s why you can call it before it’s definition.

Examples:

foo(); // this will work

bar(); // this won’t work (var a = undefined;)

function foo() {}; // function declaration

var a = function bar() {}; // function expression

<http://code.tutsplus.com/tutorials/javascript-hoisting-explained--net-15092>

<http://www.adequatelygood.com/JavaScript-Scoping-and-Hoisting.html>

<http://davidbcalhoun.com/2011/different-ways-of-defining-functions-in-javascript-this-is-madness/>

Regarding scoping, JS has only a function level scope. Blocks, such as if statements do not create a new scope. Scope refers to the current context of your code, and it can be globally and locally defined. As soon as you define a variable, it ends up in the global scope. *Global scope* is needed to create modules/APIs that are accessible across scopes. But it’s also bad because you can have namespace clashes with global scope.

You create a *local scope* using functions. Variable declared inside of functions are not visible in the global scope, unless you expose it.

Whenever you create a function inside of another function, this inner function has an access to it’s outer context, to the scope of the outer function. This is called *lexical scope* or *closure*.

<http://toddmotto.com/everything-you-wanted-to-know-about-javascript-scope/>

**25. What's the difference between an attribute and a property.**

Attributes are defined by HTML, and properties by the DOM. JS DOM objects have properties. These properties are kind of like instance variables for the particular element. A property can be different types (boolean, string..). They can be accessed using jQuery’s ***prop*** method.

Attributes are only strings, no other type. If an element has a default value, the attributes show the default value even if the value has changed. Attributes are useful when you want to set a custom attribute, when there is no property associated.

<http://lucybain.com/blog/2014/attribute-vs-property/>

<http://zanecorp.com/wiki/index.php/Javascript_Attributes_vs_Properties>

<http://stackoverflow.com/questions/19246714/html-attributes-vs-properties>

**26.** **Why is extending built-in objects prototypes not a good idea?**

The risk of breaking things is very high, since it changes the behaviour of an object used by other code. It can overwrite some methods defined in other places.

<http://stackoverflow.com/questions/14034180/why-is-extending-native-objects-a-bad-practice>

<http://perfectionkills.com/extending-native-builtins/>

<http://programmers.stackexchange.com/questions/104320/why-is-extending-the-dom-built-in-object-prototypes-a-bad-idea>

**27. Explain how would you get a query string parameter from the browser window's URL.**

Use the window.location object and it’s property ‘search’: location.search. It returns a string.

**28.** **Make this work: [1,2,3,4,5].duplicate(); //[1,2,3,4,5,1,2,3,4,5]**

You can extend Array.prototype to implement this method. One possible solution would be:

|  |
| --- |
| Array.prototype.duplicate = function() {  return this.concat(this);  } |

Another solution - using push:

|  |
| --- |
| **Array.prototype.duplicate = function() {**  **for (var i=0; i<this.length; i++) {**  **this.push(this[i]);**  **}**  **}** |

**29.** **Describe a strategy for memoization (avoiding calculation repetition) in JS.**

It’s like caching in a local variable. First time it gets set and next times it uses that set value.

Example (Coffeescript):

test = ||= 2;

<https://www.safaribooksonline.com/library/view/javascript-the-good/9780596517748/ch04s15.html>

<http://addyosmani.com/blog/faster-javascript-memoization/>

**30. Why is it called ternary operator, what does the word ternary indicate?**

It’s an operator that takes 3 arguments. condition ? value\_if\_true : value\_if\_false

Example:

(3>2) ? true : false

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator>

**31.** **What is “use strict”? Advantages vs disadvantages**

ECMAScript5 strict mode is a way to use more restricted variant of JS. It has a different semantics from normal code; it makes several changes, such as:

* eliminating some JS silent errors by changing them to throw errors (such as “bad syntax” that has been previously accepted)
* fixes mistakes that makes it difficult for JS engines to perform optimizations (hence can make code run faster)
* it makes it easier to write “secure” JS

To invoke the “strict mode” for an entire script, put the statement “use strict” before any other statements. To invoke it for a function, put that statement in the function’s body before any other statements.

Example:

“use strict”

x = 5; // will cause an error if x is not previously declared

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode>

<http://www.w3schools.com/js/js_strict.asp>

**32.** **Explain Web cache and what is it used for?**

Web cache saves copies of responses like HTML pages, images and files between two web servers/clients. If there is another request for the same thing, it can use the cached content instead of asking the origin server for it again. The reasons for that are:

* to reduce latency: because the response is retrieved from the cache, instead of making another server request and waiting
* to reduce network traffic: it reduces the amount of bandwidth used

There are a couple of kinds of web caches:

* **browser caches**: web browsers have a cache setting. This sets aside a section of your hard disk to store representations that you’ve seen. It will check that the representations are fresh, usually once a session (current browser invocation). This is particularly useful when the user hits a ‘back’ button.
* **proxy caches**: they work on the same principle, but on a much larger scale. They are usually deployed by network administrators to save bandwidth. They serve hundreds or thousands of users. Since they are out of network (not part of the client or the origin server), requests have to be routed. This is usually done with interception or using the browser’s proxy setting to manually tell it what proxy to use.
* **gateway caches**: they are typically deployed by webmasters themselves to make their sites more scalable, reliable and better performing. CDNs distribute gateway caches throughout the Internet and sell caching to interested websites (like Akamai)

How it works:

All caches have a set of rules that they use to determine when to server a representation from the cache (if available). Some of these rules are set in HTTP request protocols, and some are set by the administrator of the cache.

1. If the response headers tell the cache not to keep it, it won’t be cached. Example: **cache-control:** no-cache, no-store, max-age=0, must-revalidate
2. If the request is authenticated or secure like HTTPS, it won’t be cached.
3. Cached representation is considered fresh if it has an expiry time header and is still within the fresh period, or it has been cached recently and modified relatively long ago.
4. If representation is stale, the origin server would be asked to validate it or tell the cache that the cached copy is still good.
5. A cache can serve style response in some cases (e.g. disconnected from the network).

<https://www.mnot.net/cache_docs/>

**33. Explain dependency injection (DI) pattern in JS**

Even if you write your code and aim to have dependency-free modules, there will still be a part that combines everything together. When DI is used properly, your dependencies are given to your object, instead your object creating or explicitly referencing them. DI allows us to perform better testing because we can mock dependencies and test one thing at a time.

<https://docs.angularjs.org/guide/di>

<http://merrickchristensen.com/articles/javascript-dependency-injection.html>

<http://krasimirtsonev.com/blog/article/Dependency-injection-in-JavaScript>

**34.** **What is FOUC, how to avoid FOUC.**

FOUC stands for the Flash of Unstyled Content. It can occur when page content has been displayed to the user before all the styles have been applied to it. It can also occur if Javascript functions alter the page’s content/style after loading.

To counteract this, you can set a class (or an id) elements that are going to be modified later and attach some CSS (or hide them) until the JS has been loaded, then it should set the display to block/inline.

<http://www.paulirish.com/2009/avoiding-the-fouc-v3/>

<https://docs.google.com/presentation/d/1jt_VQC5LDF-e9j8Wtxu4KZPa8ItlmYmntGy5tdcbGOY/present?slide=id.g16b58378_0_17>

**35. Explain Single Page Apps (SPA)**

<http://en.wikipedia.org/wiki/Single-page_application>

<http://singlepageappbook.com/>

**36. What are polyfills and what are they used for?**