JS-1 jr

**1: What is the object type?**

The object type refers to a compound value where you can set properties (named locations) that each hold their own values of any type.

|  |
| --- |
| var obj = {  a: "hello world", // property  b: 42,  c: true  };  obj.a; // "hello world", accessed with doted notation  obj.b; // 42  obj.c; // true  obj["a"]; // "hello world", accessed with bracket notation  obj["b"]; // 42  obj["c"]; // true |

Bracket notation is also useful if you want to access a property/key but the name is stored in another variable, such as:

|  |
| --- |
| var obj = {  a: "hello world",  b: 42  };  var b = "a";  obj[b]; // "hello world"  obj["b"]; // 42 |

**2: Explain arrays in JavaScript**

An array is an object that holds values (of any type) not particularly in named properties/keys, but rather in numerically indexed positions:

|  |
| --- |
| var arr = [  "hello world",  42,  true  ];  arr[0]; // "hello world"  arr[1]; // 42  arr[2]; // true  arr.length; // 3  typeof arr; // "object" |

**3: What is typeof operator?**

JavaScript provides a typeof operator that can examine a value and tell you what type it is:

|  |
| --- |
| var a;  typeof a; // "undefined"  a = "hello world";  typeof a; // "string"  a = 42;  typeof a; // "number"  a = true;  typeof a; // "boolean"  a = null;  typeof a; // "object" -- weird, bug  a = undefined;  typeof a; // "undefined"  a = { b: "c" };  typeof a; // "object" |

**4: Explain equality in JavaScript**

JavaScript has both strict and type–converting comparisons:

Strict comparison (e.g., ===) checks for value equality without allowing coercion

Abstract comparison (e.g. ==) checks for value equality with coercion allowed

|  |
| --- |
| var a = "42";  var b = 42;  a == b; // true  a === b; // false |

Some simple equalityrules:

* if either value (aka side) in a comparison could be the true or false value, avoid == and use ===.
* If either value in a comparison could be of these specific values (0, "", or [] -- empty array), avoid == and use ===.
* In all other cases, you're safe to use ==. Not only is it safe, but in many cases it simplifies your code in a way that improves readability.

**5: What is Scope in JavaScript?**

In JavaScript, each function gets its own scope. Scope is basically a collection of variables as well as the rules for how those variables are accessed by name. Only code inside that function can access that function's scoped variables.

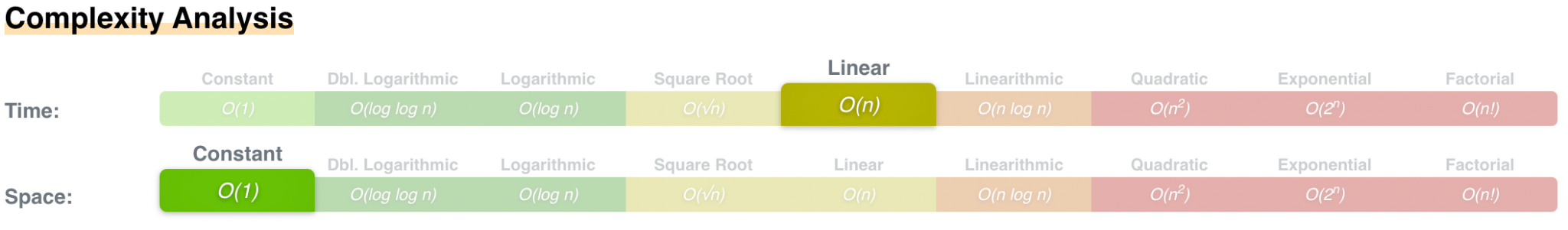
A variable name has to be unique within the same scope. A scope can be nested inside another scope. If one scope is nested inside another, code inside the innermost scope can access variables from either scope.

**6: Explain what is Linear (Sequential) Search and when may we use one?**

Linear (sequential) search goes through all possible elements in some array and compare each one with the desired element. It may take up to O(n) operations, where N is the size of an array and is widely considered to be horribly slow. In linear search when you perform one operation you reduce the size of the problem by one (when you do one operation in binary search you reduce the size of the problem by half). Despite it, it can still be used when:

* You need to perform this search only once,
* You are forbidden to rearrange the elements and you do not have any extra memory,
* The array is tiny, such as ten elements or less, or the performance is not an issue at all,
* Even though in theory other search algorithms may be faster than linear search (for instance binary search), in practice even on medium-sized arrays (around 100 items or less) it might be infeasible to use anything else. On larger arrays, it only makes sense to use other, faster search methods if the data is large enough, because the initial time to prepare (sort) the data is comparable to many linear searches,
* When the list items are arranged in order of decreasing probability, and these probabilities are geometrically distributed, the cost of linear search is only O(1)
* You have no idea what you are searching.

When you ask MySL something like SELECT x FROM y WHERE z = t, and z is a column without an index, linear search is performed with all the consequences of it. This is why adding an index to searchable columns is important.



* A linear search runs in at worst linear time and makes at most n comparisons, where n is the length of the list. If each element is equally likely to be searched, then linear search has an average case of (n+1)/2 comparisons, but the average case can be affected if the search probabilities for each element vary.
* When the list items are arranged in order of decreasing probability, and these probabilities are geometrically distributed, the cost of linear search is only O(1)

**Implementation**

|  |
| --- |
| function linearSearch(array, toFind){  for(let i = 0; i < array.length; i++){  if(array[i] === toFind) return i;  }  return -1;  } |

**7: Explain Values and Types in JavaScript**

JavaScript has typed values, not typed variables. The following built-in types are available:

* string
* number
* boolean
* null and undefined
* object
* symbol (new to ES6)

**8: What is let keyword in JavaScript?**

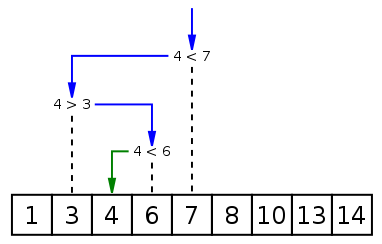
In addition to creating declarations for variables at the function level, ES6 lets you declare variables to belong to individual blocks (pairs of { .. }), using the let keyword.

**9: Explain what is Binary Search**

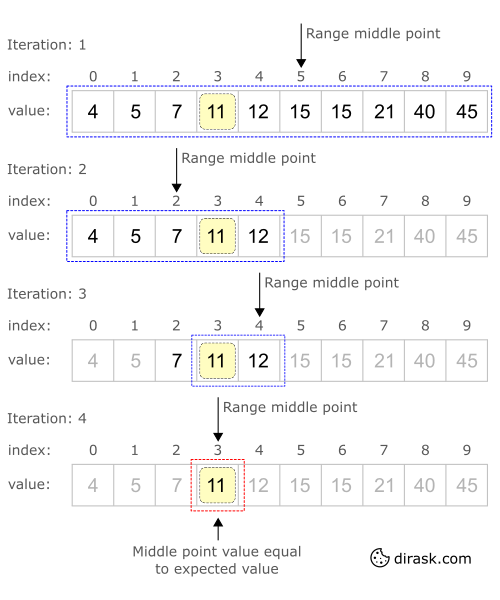
When the list is sorted we can use the binary search (also known as half-interval search, logarithmic search, or binary chop) technique to find items on the list. Here's a step-by-step description of using binary search:

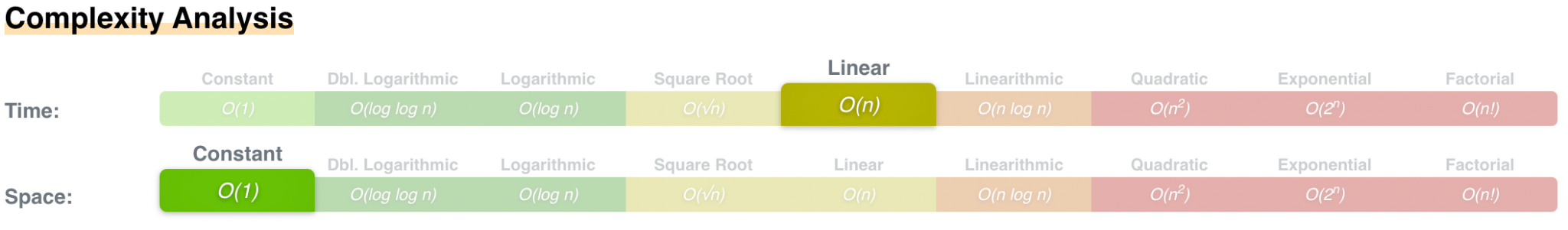
1. Let min = 1 and max = n.
2. Guess the average of max and min rounded down so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set min to be one larger than the guess.
5. If the guess was too high, set max to be one smaller than the guess.
6. Go back to step two.

In this example we looking for array item with value 4:



When you do one operation in binary search we reduce the size of the problem by half (look at the picture below how do we reduce the size of the problem area) hence the complexity of binary search is O(log n). The binary search algorithm can be written either recursively or iteratively.





|  |
| --- |
| var binarySearch = function(array, value) {  var guess,  min = 0,  max = array.length - 1;  while(min <= max){  guess = Math.floor((min + max) /2);  if(array[guess] === value)  return guess;  else if(array[guess] < value)  min = guess + 1;  else  max = guess - 1;  }  return -1;  } |

**10: Explain the same-origin policy with regards to JavaScript**.

The same-origin policy prevents JavaScript from making requests across domain boundaries. An origin is defined as a combination of URI scheme, hostname, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's Document Object Model.

**11: What is the difference between == and ===?**

== is the abstract equality operator while === is the strict equality operator. The == operator will compare for equality after doing any necessary type conversions. The === operator will not do type conversion, so if two values are not the same type === will simply return false. When using ==, funky things can happen, such as:

|  |
| --- |
| 1 == '1'; // true  1 == [1]; // true  1 == true; // true  0 == ''; // true  0 == '0'; // true  0 == false; // true |

My advice is never to use the == operator, except for convenience when comparing against null or undefined, where a == null will return true if a is null or undefined.

|  |
| --- |
| var a = null;  console.log(a == null); // true  console.log(a == undefined); // true |

**12: Is there anyway to force using strict mode in Node.js?**

you can place

"use strict";

at the top of your file in node >= 0.10.7, but if you want your whole app to run in strict (including external modules) you can do this

node --use\_strict

**13: Why would you use something like the load event? Does this event have disadvantages? Do you know any alternatives, and why would you use those?**

The load event fires at the end of the document loading process. At this point, all of the objects in the document are in the DOM, and all the images, scripts, links and sub-frames have finished loading.

The DOM event DOMContentLoaded will fire after the DOM for the page has been constructed, but do not wait for other resources to finish loading. This is preferred in certain cases when you do not need the full page to be loaded before initializing.

**14: What is strict mode?**

Strict Mode is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a "strict" operating context. This strict context prevents certain actions from being taken and throws more exceptions.

|  |
| --- |
| // Non-strict code...  (function(){  "use strict";  // Define your library strictly...  })();  // Non-strict code... |

**15: What's the difference between Host objects and Native objects?**

Native objects are objects that are part of the JavaScript language defined by the ECMAScript specification, such as String, Math, RegExp, Object, Function, etc.

Host objects are provided by the runtime environment (browser or Node), such as window, XMLHTTPRequest, etc.

**16: What language constructions do you use for iterating over object properties and array items?**

**For objects:**

* for loops - for (var property in obj) { console.log(property); }. However, this will also iterate through its inherited properties, and you will add an obj.hasOwnProperty(property) check before using it.
* Object.keys() - Object.keys(obj).forEach(function (property) { ... }). Object.keys() is a static method that will lists all enumerable properties of the object that you pass it.
* Object.getOwnPropertyNames() - Object.getOwnPropertyNames(obj).forEach(function (property) { ... }). Object.getOwnPropertyNames() is a static method that will lists all enumerable and non-enumerable properties of the object that you pass it.

**For arrays:**

* for loops - for (var i = 0; i < arr.length; i++). The common pitfall here is that var is in the function scope and not the block scope and most of the time you would want block scoped iterator variable. ES2015 introduces let which has block scope and it is recommended to use that instead. So this becomes: for (let i = 0; i < arr.length; i++).
* forEach - arr.forEach(function (el, index) { ... }). This construct can be more convenient at times because you do not have to use the index if all you need is the array elements. There are also the every and some methods which will allow you to terminate the iteration early.

Most of the time, I would prefer the .forEach method, but it really depends on what you are trying to do. for loops allow more flexibility, such as prematurely terminate the loop using break or incrementing the iterator more than once per loop.

**17: What are some of the advantages/disadvantages of writing JavaScript code in a language that compiles to JavaScript?**

Some examples of languages that compile to JavaScript include CoffeeScript, Elm, ClojureScript, PureScript, and TypeScript.

**Advantages**:

* Fixes some of the longstanding problems in JavaScript and discourages JavaScript anti-patterns.
* Enables you to write shorter code, by providing some syntactic sugar on top of JavaScript, which I think ES5 lacks, but ES2015 is awesome.
* Static types are awesome (in the case of TypeScript) for large projects that need to be maintained over time.

**Disadvantages**:

* Require a build/compile process as browsers only run JavaScript and your code will need to be compiled into JavaScript before being served to browsers.
* Debugging can be a pain if your source maps do not map nicely to your pre-compiled source.
* Most developers are not familiar with these languages and will need to learn it. There's a ramp up cost involved for your team if you use it for your projects.
* Smaller community (depends on the language), which means resources, tutorials, libraries, and tooling would be harder to find.
* IDE/editor support might be lacking.
* These languages will always be behind the latest JavaScript standard.
* Developers should be cognizant of what their code is being compiled to — because that is what would actually be running, and that is what matters in the end.

Practically, ES2015 has vastly improved JavaScript and made it much nicer to write. I don't really see the need for CoffeeScript these days.

**18: Explain event bubbling and how one may prevent it**

Event bubbling is the concept in which an event triggers at the deepest possible element, and triggers on parent elements in nesting order. As a result, when clicking on a child element one may exhibit the handler of the parent activating.

One way to prevent event bubbling is using event.stopPropagation() or event.cancelBubble on IE < 9.

**19: What does use strict do?**

The use strict literal is entered at the top of a JavaScript program or at the top of a function and it helps you write safer JavaScript code by throwing an error if a global variable is created by mistake. For example, the following program will throw an error:

|  |
| --- |
| function doSomething(val) {  "use strict";  x = val + 10;  }` |

It will throw an error because x was not defined and it is being set to some value in the global scope, which isn't allowed with use strict The small change below fixes the error being thrown:

|  |
| --- |
| function doSomething(val) {  "use strict";  var x = val + 10;  } |

**20: Why is it, in general, a good idea to leave the global scope of a website as-is and never touch it?**

Every script has access to the global scope, and if everyone uses the global namespace to define their variables, collisions will likely occur. Use the module pattern (IIFEs) to encapsulate your variables within a local namespace.

**21: What is a Polyfill?**

A polyfill is essentially the specific code (or plugin) that would allow you to have some specific functionality that you expect in current or “modern” browsers to also work in other browsers that do not have the support for that functionality built in.

Polyfills are not part of the HTML5 standard

Polyfilling is not limited to Javascript

22: Explain Null and Undefined in JavaScript

JavaScript (and by extension TypeScript) has two bottom types: null and undefined. They are intended to mean different things:

Something hasn't been initialised : undefined.

Something is currently unavailable: null.

**23: What's the difference between throw Error('msg') vs throw new Error('msg')?**

Problem

|  |
| --- |
| var err1 = Error('message');  var err2 = new Error('message');  Which one is correct and why? |

Answer

Both are fine; the function call Error(…) is equivalent to the object creation expression new Error(…) with the same arguments.

**24: What is Callback Hell and what is the main cause of it?**

Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

|  |
| --- |
| fs.readdir(source, function (err, files) {  if (err) {  console.log('Error finding files: ' + err)  } else {  files.forEach(function (filename, fileIndex) {  console.log(filename)  gm(source + filename).size(function (err, values) {  if (err) {  console.log('Error identifying file size: ' + err)  } else {  console.log(filename + ' : ' + values)  aspect = (values.width / values.height)  widths.forEach(function (width, widthIndex) {  height = Math.round(width / aspect)  console.log('resizing ' + filename + 'to ' + height + 'x' + height)  this.resize(width, height).write(dest + 'w' + width + '\_' + filename, function(err) {  if (err) console.log('Error writing file: ' + err)  })  }.bind(this))}})})}}) |

See the pyramid shape and all the }) at the end? This is affectionately known as callback hell.

The cause of callback hell is when people try to write JavaScript in a way where execution happens visually from top to bottom. Lots of people make this mistake! In other languages like C, Ruby or Python there is the expectation that whatever happens on line 1 will finish before the code on line 2 starts running and so on down the file.

JS-2 medm

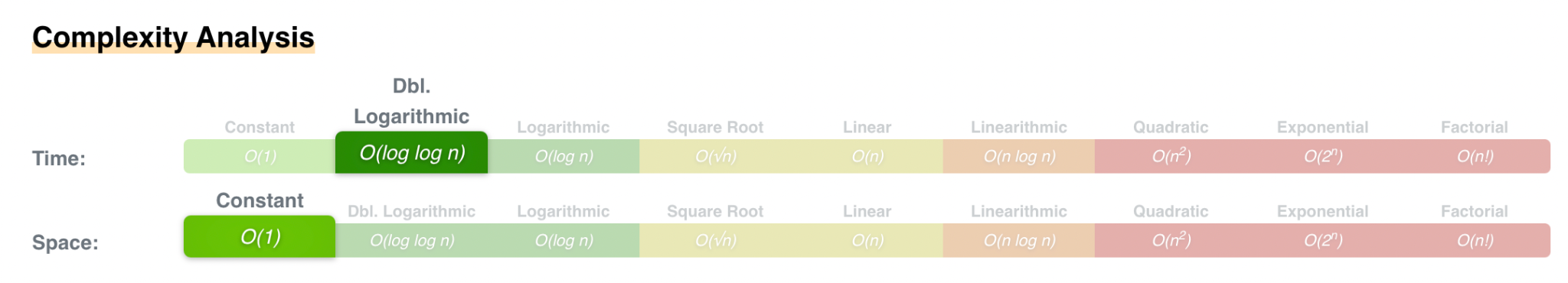
**25: Explain what is Interpolation Search**

**Interpolation Search** is an algorithm similar to Binary Search for searching for a given target value in a sorted array. It is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed.

The binary search always chooses the middle of the remaining search space. On a contrast Interpolation search, at each search step, calculates (using interpolation formula) where in the remaining search space the target might be present, based on the low and high values of the search space and the value of the target. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side. The value actually found at this estimated position is then compared to the target value. If it is not equal, then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position.

The idea of the interpolation formula or partitioning logic is to return higher value of pos when element to be searched is closer to arr[hi]. And smaller value when closer to arr[low].

|  |
| --- |
| arr[] ==> Array where elements need to be searched  x ==> Element to be searched  low ==> Starting index in arr[]  hi ==> Ending index in arr[]  pos = low + [ (x-arr[low])\*(hi-low) / (arr[hi]-arr[Low]) ] |



In the interpolation search algorithm, the midpoint is computed in such a way that it gives a higher probability of obtaining our search term faster. The worst case complexity of interpolation search is O(n). However, its average case complexity, under the assumption that the keys are uniformly distributed, is \_O(log log N).

It can be proven that Interpolation Search repeatedly reducing the problem to a subproblem of size that is the square root of the original problem size, and algorithm will terminate after O(log log n) steps (see proof here). When the values are equally dispersed into the interval this search algorithm can be extremely useful – way faster than the binary search (that divides searching space by half).

**Implementation**

|  |
| --- |
| const interpolationSearch = (array, key) => {  // if array is empty.  if (!array.length) {  return -1;  }  let low = 0;  let high = array.length - 1;  while (low <= high && key >= array[low] && x <= array[high]) {  // calculate position with  let pos = low + Math.floor(((high - low) \* (key - array[low])) / (array[high] - array[low]));  // if all elements are same then we'll have divide by 0 or 0/0  // which may cause NaN  pos = Number.isNaN(pos) ? low : pos;  if (array[pos] === key) {  return pos;  }  if (array[pos] > key) {  high = pos - 1;  } else {  low = pos + 1;  }  }  // not found.  return -1;  }; |

**26: What is IIFEs (Immediately Invoked Function Expressions)?**

It’s an Immediately-Invoked Function Expression, or IIFE for short. It executes immediately after it’s created:

|  |
| --- |
| (function IIFE(){  console.log( "Hello!" );  })();  // "Hello!" |

This pattern is often used when trying to avoid polluting the global namespace, because all the variables used inside the IIFE (like in any other normal function) are not visible outside its scope.

**27: What is Coercion in JavaScript?**

In JavaScript conversion between different two build-in types called coercion. Coercion comes in two forms in JavaScript: explicit and implicit.

Here's an example of explicit coercion:

|  |
| --- |
| var a = "42";  var b = Number( a );  a; // "42"  b; // 42 -- the number!  And here's an example of implicit coercion:  var a = "42";  var b = a \* 1; // "42" implicitly coerced to 42 here  a; // "42"  b; // 42 -- the number! |

**28: What is the difference between a shim and a polyfill?**

A shim is any piece of code that performs interception of an API call and provides a layer of abstraction. It isn't necessarily restricted to a web application or HTML5/CSS3.

A polyfill is a type of shim that retrofits legacy browsers with modern HTML5/CSS3 features usually using Javascript or Flash.

A shim is a library that brings a new API to an older environment, using only the means of that environment. Thus, a polyfill is a shim for a browser API.

**29: What is a Jump (or Block) Search?**

Jump Search (also referred to as Block Search) is an algorithm used to search for the position of a target element on a sorted data collection or structure.

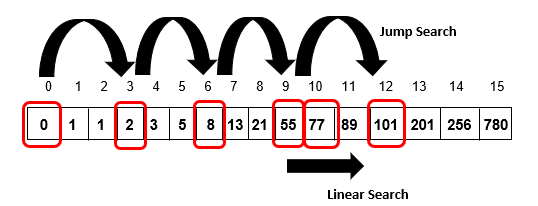
The fundamental idea behind this searching technique is to search fewer number of elements compared to linear search algorithm (which scans every element in the array to check if it matches with the element being searched or not). This can be done by skipping some fixed number of array elements or jumping ahead by fixed number of steps in every iteration.

Lets consider a sorted array A[] of size n, with indexing ranging between 0 and n-1, and element x that needs to be searched in the array A[]. For implementing this algorithm, a block of size m is also required, that can be skipped or jumped in every iteration. Thus, the algorithm works as follows:

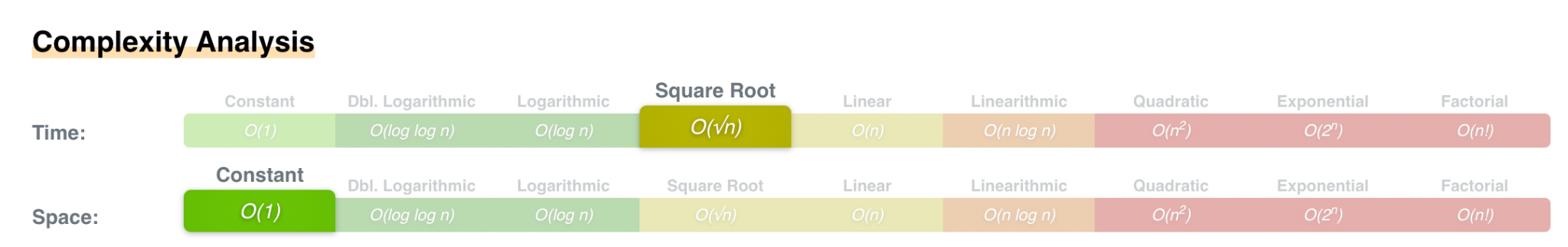
* Iteration 1: if (x==A[0]), then success, else, if (x > A[0]), then jump to the next block.
* Iteration 2: if (x==A[m]), then success, else, if (x > A[m]), then jump to the next block.
* Iteration 3: if (x==A[2m]), then success, else, if (x > A[2m]), then jump to the next block.
* At any point in time, if (x < A[km]), then a linear search is performed from index A[(k-1)m] to A[km]

Consider the following inputs:

* A[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 77, 89, 101, 201, 256, 780}
* item = 77



* The single most important advantage of jump search when compared to binary search is that it does not rely on the division operator (/).
* Jump Search is highly efficient for searching arrays especially when its only-seeks-forward behavior is favorable - its average performance makes it sit somewhere between Binary Search with its O(log n) complexity and Linear Search with an O(n) complexity.



* Jump Search would consecutively jump to its very last block searching for the target element, in turn causing an n/m number of jumps. Additionally, if the last element’s value of this block was greater than the target element, Jump Search would perform a linear search with m-1 iterations.
* This causes Jump Search to make (n/m) jumps with additional m-1 iterations. This value is minimum at m = √n. Therefore, the optimum block size is √n. Accordingly, Jump Search maintains a worst and average case efficiency of O(√n) complexity.
* The space complexity of this algorithm is O(1) since it does not require any additional temporary variables or data structures for its implementation.

**Implementation**

|  |
| --- |
| function jumpSearch(arrayToSearch, valueToSearch){  var length = arrayToSearch.length;  var step = Math.floor(Math.sqrt(length));  var index = Math.min(step, length)-1;  var lowerBound = 0;  while (arrayToSearch[Math.min(step, length)-1] < valueToSearch)  {  lowerBound = step;  step += step;  if (lowerBound >= length){  return -1;  }  }  var upperBound = Math.min(step, length);  while (arrayToSearch[lowerBound] < valueToSearch)  {  lowerBound++;  if (lowerBound == upperBound){  return -1;  }  }  if (arrayToSearch[lowerBound] == valueToSearch){  return lowerBound;  }  return -1;  } |

**30: What is the definition of a Higher-Order Function?**

A higher-order function is any function that takes one or more functions as arguments, which it uses to operate on some data, and/or returns a function as a result. Higher-order functions are meant to abstract some operation that is performed repeatedly. The classic example of this is map, which takes an array and a function as arguments. map then uses this function to transform each item in the array, returning a new array with the transformed data. Other popular examples in JavaScript are forEach, filter, and reduce. A higher-order function doesn't just need to be manipulating arrays as there are many use cases for returning a function from another function. Function.prototype.bind is one such example in JavaScript.

**Map**

Let say we have an array of names which we need to transform each string to uppercase.

|  |
| --- |
| const names = ['irish', 'daisy', 'anna']; |

The imperative way will be as such:

|  |
| --- |
| const transformNamesToUppercase = function(names) {  const results = [];  for (let i = 0; i < names.length; i++) {  results.push(names[i].toUpperCase());  }  return results;  };  transformNamesToUppercase(names); // ['IRISH', 'DAISY', 'ANNA'] |

Use .map(transformerFn) makes the code shorter and more declarative.

|  |
| --- |
| const transformNamesToUppercase = function(names) {  return names.map(name => name.toUpperCase());  };  transformNamesToUppercase(names); // ['IRISH', 'DAISY', 'ANNA'] |

**31: What do you think of AMD vs CommonJS?**

Both are ways to implement a module system, which was not natively present in JavaScript until ES2015 came along. CommonJS is synchronous while AMD (Asynchronous Module Definition) is obviously asynchronous. CommonJS is designed with server-side development in mind while AMD, with its support for asynchronous loading of modules, is more intended for browsers.

I find AMD syntax to be quite verbose and CommonJS is closer to the style you would write import statements in other languages. Most of the time, I find AMD unnecessary, because if you served all your JavaScript into one concatenated bundle file, you wouldn't benefit from the async loading properties. Also, CommonJS syntax is closer to Node style of writing modules and there is less context-switching overhead when switching between client side and server side JavaScript development.

I'm glad that with ES2015 modules, that has support for both synchronous and asynchronous loading, we can finally just stick to one approach. Although it hasn't been fully rolled out in browsers and in Node, we can always use transpilers to convert our code.

**32: Explain the differences on the usage of foo between function foo() {} and var foo = function() {}**

The former is a function declaration while the latter is a function expression. The key difference is that function declarations have its body hoisted but the bodies of function expressions are not (they have the same hoisting behavior as variables). If you try to invoke a function expression before it is defined, you will get an Uncaught TypeError: XXX is not a function error.

**Function Declaration**

|  |
| --- |
| foo(); // 'FOOOOO'  function foo() {  console.log('FOOOOO');  } |

**Function Expression**

|  |
| --- |
| foo(); // Uncaught TypeError: foo is not a function  var foo = function() {  console.log('FOOOOO');  }; |

**33: What is the drawback of creating true private in JavaScript?**

One of the drawback of creating a true private method in JavaScript is that they are very memory inefficient because a new copy of the method would be created for each instance.

|  |
| --- |
| var Employee = function (name, company, salary) {  this.name = name || ""; //Public attribute default value is null  this.company = company || ""; //Public attribute default value is null  this.salary = salary || 5000; //Public attribute default value is null  // Private method  var increaseSalary = function () {  this.salary = this.salary + 1000;  };  // Public method  this.dispalyIncreasedSalary = function() {  increaseSalary();  console.log(this.salary);  };  };  // Create Employee class object  var emp1 = new Employee("John","Pluto",3000);  // Create Employee class object  var emp2 = new Employee("Merry","Pluto",2000);  // Create Employee class object  var emp3 = new Employee("Ren","Pluto",2500); |

Here each instance variable emp1, emp2, emp3 has own copy of increaseSalary private method.So as recommendation don't go for a private method unless it's necessary.

**34: What's the difference between .call and .apply?**

Both .call and .apply are used to invoke functions and the first parameter will be used as the value of this within the function. However, .call takes in comma-separated arguments as the next arguments while .apply takes in an array of arguments as the next argument. An easy way to remember this is C for call and comma-separated and A for apply and an array of arguments.

|  |
| --- |
| function add(a, b) {  return a + b;  }  console.log(add.call(null, 1, 2)); // 3  console.log(add.apply(null, [1, 2])); // 3 |

**35: What is the preferred syntax for defining enums in JavaScript?**

Since 1.8.5 it's possible to seal and freeze the object, so define the above as:

|  |
| --- |
| var DaysEnum = Object.freeze({  "monday": 1,  "tuesday": 2,  "wednesday": 3,  ...  })  or  var DaysEnum = {  "monday": 1,  "tuesday": 2,  "wednesday": 3,  ...  }  Object.freeze(DaysEnum)  and voila! JS enums. |

However, this doesn't prevent you from assigning an undesired value to a variable, which is often the main goal of enums:

|  |
| --- |
| let day = DaysEnum.tuesday  day = 298832342 // goes through without any errors |

**36: Describe Closure concept in JavaScript as best as you could**

|  |
| --- |
| function makeAdder(x) {  // parameter `x` is an inner variable  // inner function `add()` uses `x`, so  // it has a "closure" over `x`  function add(y) {  return y + x;  };  return add;  } |

Reference to inner add function returned is able to remember what x value was passed to makeAdder function call.

|  |
| --- |
| var plusOne = makeAdder( 1 ); // x is 1, plusOne has a reference to add(y)  var plusTen = makeAdder( 10 ); // x is 10  plusOne(3); // 1 (x) + 3 (y) = 4  plusTen(13); // 10 (x) + 13 (y) = 23 |

In C and most other common languages, after a function returns, all the local variables are no longer accessible because the stack-frame is destroyed.

In JavaScript, if you declare a function within another function, then the local variables can remain accessible after returning from the function you called.

A closure is a stack frame which is allocated when a function starts its execution, and not freed after the function returns (as if a 'stack frame' were allocated on the heap rather than the stack!). In JavaScript, you can think of a function reference variable as having both a pointer to a function as well as a hidden pointer to a closure.

**37: Could you explain the difference between ES5 and ES6**

* ECMAScript 5 (ES5): The 5th edition of ECMAScript, standardized in 2009. This standard has been implemented fairly completely in all modern browsers
* ECMAScript 6 (ES6)/ ECMAScript 2015 (ES2015): The 6th edition of ECMAScript, standardized in 2015. This standard has been partially implemented in most modern browsers.
* Here are some key differences between ES5 and ES6:
* Arrow functions & string interpolation:

Consider:

|  |
| --- |
| const greetings = (name) => {  return `hello ${name}`;  } |

and even:

|  |
| --- |
| const greetings = name => `hello ${name}`; |

* Const.

Const works like a constant in other languages in many ways but there are some caveats. Const stands for ‘constant reference’ to a value. So with const, you can actually mutate the properties of an object being referenced by the variable. You just can’t change the reference itself.

|  |
| --- |
| const NAMES = [];  NAMES.push("Jim");  console.log(NAMES.length === 1); // true  NAMES = ["Steve", "John"]; // error |

* Block-scoped variables.

The new ES6 keyword let allows developers to scope variables at the block level. Let doesn’t hoist in the same way var does.

* Default parameter values Default parameters allow us to initialize functions with default values. A default is used when an argument is either omitted or undefined — meaning null is a valid value.

|  |
| --- |
| // Basic syntax  function multiply (a, b = 2) {  return a \* b;  }  multiply(5); // 10 |

* Class Definition and Inheritance

ES6 introduces language support for classes (class keyword), constructors (constructor keyword), and the extend keyword for inheritance.

* for-of operator

The for...of statement creates a loop iterating over iterable objects.

* Spread Operator For objects merging

|  |
| --- |
| const obj1 = { a: 1, b: 2 }  const obj2 = { a: 2, c: 3, d: 4}  const obj3 = {...obj1, ...obj2} |

* Promises

Promises provide a mechanism to handle the results and errors from asynchronous operations. You can accomplish the same thing with callbacks, but promises provide improved readability via method chaining and succinct error handling.

|  |
| --- |
| const isGreater = (a, b) => {  return new Promise ((resolve, reject) => {  if(a > b) {  resolve(true)  } else {  reject(false)  }  })  }  isGreater(1, 2)  .then(result => {  console.log('greater')  })  .catch(result => {  console.log('smaller')  }) |

* Modules exporting & importing Consider module exporting:

|  |
| --- |
| const myModule = { x: 1, y: () => { console.log('This is ES5') }}  export default myModule; |

and importing:

|  |
| --- |
| import myModule from './myModule'; |

**38: When should we use generators in ES6?**

To put it simple, generator has two features:

* one can choose to jump out of a function and let outer code to determine when to jump back into the function.
* the control of asynchronous call can be done outside of your code

The most important feature in generators — we can get the next value in only when we really need it, not all the values at once. And in some situations it can be very convenient.

**39: Explain Function.prototype.bind.**

Taken word-for-word from MDN:

The bind() method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

In my experience, it is most useful for binding the value of this in methods of classes that you want to pass into other functions. This is frequently done in React components.

**40: What are the benefits of using spread syntax in ES6 and how is it different from rest syntax?**

ES6's spread syntax is very useful when coding in a functional paradigm as we can easily create copies of arrays or objects without resorting to Object.create, slice, or a library function. This language feature is used often in Redux and rx.js projects.

|  |
| --- |
| function putDookieInAnyArray(arr) {  return [...arr, 'dookie'];  }  const result = putDookieInAnyArray(['I', 'really', "don't", 'like']); // ["I", "really", "don't", "like", "dookie"]  const person = {  name: 'Todd',  age: 29,  };  const copyOfTodd = { ...person }; |

ES6's rest syntax offers a shorthand for including an arbitrary number of arguments to be passed to a function. It is like an inverse of the spread syntax, taking data and stuffing it into an array rather than unpacking an array of data, and it works in function arguments, as well as in array and object destructuring assignments.

|  |
| --- |
| function addFiveToABunchOfNumbers(...numbers) {  return numbers.map(x => x + 5);  }  const result = addFiveToABunchOfNumbers(4, 5, 6, 7, 8, 9, 10); // [9, 10, 11, 12, 13, 14, 15]  const [a, b, ...rest] = [1, 2, 3, 4]; // a: 1, b: 2, rest: [3, 4]  const { e, f, ...others } = {  e: 1,  f: 2,  g: 3,  h: 4,  }; // e: 1, f: 2, others: { g: 3, h: 4 } |

**41: When should I use Arrow Functions in ES6?**

I'm now using the following rule of thumb for functions in ES6 and beyond:

* Use function in the global scope and for Object.prototype properties.
* Use class for object constructors.
* Use => everywhere else.

Why use arrow functions almost everywhere?

* Scope safety: When arrow functions are used consistently, everything is guaranteed to use the same thisObject as the root. If even a single standard function callback is mixed in with a bunch of arrow functions there's a chance the scope will become messed up.
* Compactness: Arrow functions are easier to read and write. (This may seem opinionated so I will give a few examples further on).
* Clarity: When almost everything is an arrow function, any regular function immediately sticks out for defining the scope. A developer can always look up the next-higher function statement to see what the thisObject is.

**42: Explain the difference between undefined and not defined in JavaScript**

In JavaScript if you try to use a variable that doesn't exist and has not been declared, then JavaScript will throw an error var name is not defined and the script will stop execute thereafter. But If you use typeof undeclared\_variable then it will return undefined.

Before starting further discussion let's understand the difference between declaration and definition.

var x is a declaration because you are not defining what value it holds yet, but you are declaring its existence and the need of memory allocation.

|  |
| --- |
| var x; // declaring x  console.log(x); //output: undefined |

var x = 1 is both declaration and definition (also we can say we are doing initialisation), Here declaration and assignment of value happen inline for variable x, In JavaScript every variable declaration and function declaration brings to the top of its current scope in which it's declared then assignment happen in order this term is called hoisting.

A variable that is declared but not define and when we try to access it, It will result undefined.

|  |
| --- |
| var x; // Declaration  if(typeof x === 'undefined') // Will return true |

A variable that neither declared nor defined when we try to reference such variable then It result not defined.

|  |
| --- |
| console.log(y); // Output: ReferenceError: y is not defined |

**43: What are the advantages and disadvantages of using use strict?**

'use strict' is a statement used to enable strict mode to entire scripts or individual functions. Strict mode is a way to opt into a restricted variant of JavaScript.

**Advantages**:

* Makes it impossible to accidentally create global variables.
* Makes assignments which would otherwise silently fail to throw an exception.
* Makes attempts to delete undeletable properties throw (where before the attempt would simply have no effect).
* Requires that function parameter names be unique.
* this is undefined in the global context.
* It catches some common coding bloopers, throwing exceptions.
* It disables features that are confusing or poorly thought out.

**Disadvantages**:

* Many missing features that some developers might be used to.
* No more access to function.caller and function.arguments.
* Concatenation of scripts written in different strict modes might cause issues.
* Overall, I think the benefits outweigh the disadvantages, and I never had to rely on the features that strict mode blocks. I would recommend using strict mode.

**44: What is Currying?**

Currying is when you break down a function that takes multiple arguments into a series of functions that take part of the arguments. Here's an example in JavaScript:

|  |
| --- |
| function add (a, b) {  return a + b;  }  add(3, 4); // returns 7 |

This is a function that takes two arguments, a and b, and returns their sum. We will now curry this function:

|  |
| --- |
| function add (a) {  return function (b) {  return a + b;  }  } |

In an algebra of functions, dealing with functions that take multiple arguments (or equivalent one argument that's an N-tuple) is somewhat inelegant. So how do you deal with something you'd naturally express as, say, f(x,y)? Well, you take that as equivalent to f(x)(y) - f(x), call it g, is a function, and you apply that function to y. In other words, you only have functions that take one argument - but some of those functions return other functions (which ALSO take one argument).

**45: What are the differences between ES6 class and ES5 function constructors?**

Let's first look at example of each:

|  |
| --- |
| // ES5 Function Constructor  function Person(name) {  this.name = name;  }  // ES6 Class  class Person {  constructor(name) {  this.name = name;  }  } |

For simple constructors, they look pretty similar.

The main difference in the constructor comes when using inheritance. If we want to create a Student class that subclasses Person and add a studentId field, this is what we have to do in addition to the above.

|  |
| --- |
| // ES5 Function Constructor  function Student(name, studentId) {  // Call constructor of superclass to initialize superclass-derived members.  Person.call(this, name);  // Initialize subclass's own members.  this.studentId = studentId;  }  Student.prototype = Object.create(Person.prototype);  Student.prototype.constructor = Student;  // ES6 Class  class Student extends Person {  constructor(name, studentId) {  super(name);  this.studentId = studentId;  }  } |

It's much more verbose to use inheritance in ES5 and the ES6 version is easier to understand and remember.

**46: Why should we use ES6 classes?**

Some reasons you might choose to use Classes:

* The syntax is simpler and less error-prone.
* It's much easier (and again, less error-prone) to set up inheritance hierarchies using the new syntax than with the old.
* class defends you from the common error of failing to use new with the constructor function (by having the constructor throw an exception if this isn't a valid object for the constructor).
* Calling the parent prototype's version of a method is much simpler with the new syntax than the old (super.method() instead of ParentConstructor.prototype.method.call(this) or Object.getPrototypeOf(Object.getPrototypeOf(this)).method.call(this)).

Consider:

|  |
| --- |
| // \*\*ES5\*\*  var Person = function(first, last) {  if (!(this instanceof Person)) {  throw new Error("Person is a constructor function, use new with it");  }  this.first = first;  this.last = last;  };  Person.prototype.personMethod = function() {  return "Result from personMethod: this.first = " + this.first + ", this.last = " + this.last;  };  var Employee = function(first, last, position) {  if (!(this instanceof Employee)) {  throw new Error("Employee is a constructor function, use new with it");  }.  Person.call(this, first, last);  this.position = position;  };  Employee.prototype = Object.create(Person.prototype);  Employee.prototype.constructor = Employee;  Employee.prototype.personMethod = function() {  var result = Person.prototype.personMethod.call(this);  return result + ", this.position = " + this.position;  };  Employee.prototype.employeeMethod = function() {  // ...  }; |

And the same with ES6 classes:

|  |
| --- |
| // \*\*\*ES2015+\*\*  class Person {  constructor(first, last) {  this.first = first;  this.last = last;  }  personMethod() {  / / ...  }  }  class Employee extends Person {  constructor(first, last, position) {  super(first, last);  this.position = position;  }  employeeMethod() {  // ...  }  } |

**47: Explain the difference between Object.freeze() vs const**

const and Object.freeze are two completely different things.

* const applies to bindings ("variables"). It creates an immutable binding, i.e. you cannot assign a new value to the binding.

|  |
| --- |
| const person = {  name: "Leonardo"  };  let animal = {  species: "snake"  };  person = animal; // ERROR "person" is read-only |

* Object.freeze works on values, and more specifically, object values. It makes an object immutable, i.e. you cannot change its properties.

|  |
| --- |
| let person = {  name: "Leonardo"  };    let animal = {  species: "snake"  };  Object.freeze(person);  person.name = "Lima"; //TypeError: Cannot assign to read only property 'name' of object  console.log(person); |

**48: How to compare two objects in JavaScript?**

Two non-primitive values, like objects (including function and array) held by reference, so both == and === comparisons will simply check whether the references match, not anything about the underlying values.

For example, arrays are by default coerced to strings by simply joining all the values with commas (,) in between. So two arrays with the same contents would not be == equal:

|  |
| --- |
| var a = [1,2,3];  var b = [1,2,3];  var c = "1,2,3";  a == c; // true  b == c; // true  a == b; // false |

Note : For deep object comparison use external libs like deep-equal or implement your own recursive equality algorithm.

**49: What will be the output of the following code?**

Problem

|  |
| --- |
| var x = 1;  var output = (function() {  delete x;  return x;  })();  console.log(output); |

Answer

Above code will output 1 as output. delete operator is used to delete property from object. Here x is not an object it's global variable of type number.

**50: What is a closure, and how/why would you use one?**

A closure is the combination of a function and the lexical environment within which that function was declared. The word "lexical" refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Closures are functions that have access to the outer (enclosing) function's variables—scope chain even after the outer function has returned.

Why would you use one?

* Data privacy / emulating private methods with closures. Commonly used in the module pattern.
* Partial applications or currying.

**51: What will be the output of the following code?**

|  |
| --- |
| var x = { foo : 1};  var output = (function() {  delete x.foo;  return x.foo;  })();  console.log(output); |

Answer

Above code will output undefined as output. delete operator is used to delete a property from an object. Here x is an object which has foo as a property and from self-invoking function we are deleting foo property of object x and after deletion we are trying to reference deleted property foo which result undefined.

**52: What's a typical use case for anonymous functions?**

They can be used in IIFEs to encapsulate some code within a local scope so that variables declared in it do not leak to the global scope.

|  |
| --- |
| (function() {  // Some code here.  })();    As a callback that is used once and does not need to be used anywhere else. The code will seem more self-contained and readable when handlers are defined right inside the code calling them, rather than having to search elsewhere to find the function body.  setTimeout(function() {  console.log('Hello world!');  }, 1000);    Arguments to functional programming constructs or Lodash (similar to callbacks).  const arr = [1, 2, 3];  const double = arr.map(function(el) {  return el \* 2;  });  console.log(double); // [2, 4, 6] |

**53: Suggest one simple way of removing duplicates from an array using ES6**

Let's use the Set() object. Sets are a new object type with ES6 (ES2015) that allow to create collections of unique values. The values in a set can be either simple primitives like strings or integers, but more complex object types like object literals or arrays can also be part of a set.

Consider:

|  |
| --- |
| const array = [1, 4, 99, 3, 1, 4, 15];  const noDups = Array.from(new Set(array));  console.log(noDups); //[1, 4, 99, 3, 15] |

**53: Suggest one simple way of removing duplicates from an array using ES6**

Let's use the Set() object. Sets are a new object type with ES6 (ES2015) that allow to create collections of unique values. The values in a set can be either simple primitives like strings or integers, but more complex object types like object literals or arrays can also be part of a set.

Consider:

|  |
| --- |
| const array = [1, 4, 99, 3, 1, 4, 15];  const noDups = Array.from(new Set(array));  console.log(noDups); //[1, 4, 99, 3, 15] |

**54: What is generator in JS?**

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances. Generator functions are written using the function\* syntax. When called initially, generator functions do not execute any of their code, instead returning a type of iterator called a Generator. When a value is consumed by calling the generator's next method, the Generator function executes until it encounters the yield keyword.

The function can be called as many times as desired and returns a new Generator each time, however each Generator may only be iterated once.

|  |
| --- |
| function\* makeRangeIterator(start = 0, end = Infinity, step = 1) {  let iterationCount = 0;  for (let i = start; i < end; i += step) {  iterationCount++;  yield i;  }  return iterationCount;  } |

**55: What is the difference between document load event and document DOMContentLoaded event?**

* The DOMContentLoaded event is fired when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading.
* window's load event is only fired after the DOM and all dependent resources and assets have loaded.

**56: What's the difference between using let and var to declare a variable in ES6?**

The difference is scoping. var is scoped to the nearest function block and let is scoped to the nearest enclosing block, which can be smaller than a function block. Both are global if outside any block. Also, variables declared with let are not accessible before they are declared in their enclosing block. This will throw a ReferenceError exception.

**57: What is the motivation for bringing Symbol to ES6?**

Symbols are a new, special kind of object that can be used as a unique property name in objects. Using Symbol instead of string's allows different modules to create properties that don't conflict with one another. Symbols can also be made private, so that their properties can't be accessed by anyone who doesn't already have direct access to the Symbol.

Symbols are a new primitive. Just like the number, string, and boolean primitives, Symbol have a function which can be used to create them. Unlike the other primitives, Symbols do not have a literal syntax (e.g how string have '') - the only way to create them is with the Symbol constructor in the following way:

|  |
| --- |
| let symbol = Symbol(); |

In reality, Symbol's are just a slightly different way to attach properties to an object - you could easily provide the well-known Symbols as standard methods, just like Object.prototype.hasOwnProperty which appears in everything that inherits from Object.

58: Why is extending built-in JavaScript objects not a good idea?

Extending a built-in/native JavaScript object means adding properties/functions to its prototype. While this may seem like a good idea at first, it is dangerous in practice. Imagine your code uses a few libraries that both extend the Array.prototype by adding the same contains method, the implementations will overwrite each other and your code will break if the behavior of these two methods is not the same.

The only time you may want to extend a native object is when you want to create a polyfill, essentially providing your own implementation for a method that is part of the JavaScript specification but might not exist in the user's browser due to it being an older browser.

**59: What advantages are using arrow functions?**

* Scope safety: Until arrow functions, every new function defined its own this value (a new object in the case of a constructor, undefined in strict mode function calls, the base object if the function is called as an "object method", etc.). An arrow function does not create its own this, the this value of the enclosing execution context is used.
* Compactness: Arrow functions are easier to read and write.
* Clarity: When almost everything is an arrow function, any regular function immediately sticks out for defining the scope. A developer can always look up the next-higher function statement to see what the thisObject is.

**60: What is the difference between Anonymous and Named functions?**

|  |
| --- |
| var foo = function() { // anonymous function assigned to variable foo  // ..  };  var x = function bar(){ // named function (bar) assigned to variable x  // ..  };  foo(); // actual function execution  x(); |

**61: What is export default in JavaScript?**

As the name says, it's used to export functions, objects, classes or expressions from script files or modules

**Utiliites.js**

|  |
| --- |
| export function cube(x) {  return x \* x \* x;  }  export const foo = Math.PI + Math.SRT2; |

This can be imported and used as

**App.js**

|  |
| --- |
| import { cube, foo } from 'Utilities';  console.log(cube(3)); // 27  console.log(foo); // 4.555806215962888 |

Or

|  |
| --- |
| import \* as utilities from 'Utilities';  console.log(utilities.cube(3)); // 27  console.log(utilities.foo); // 4.555806215962888 |

When export default is used, this is much simpler. Script files just exports one thing. cube.js

|  |
| --- |
| export default function cube(x) {  return x \* x \* x;  };  and used as App.js  import Cube from 'cube';  console.log(Cube(3)); // 27 |

JS-3 Senr

**62: What is the new keyword in JavaScript?**

1. It creates a new object. The type of this object is simply object.
2. It sets this new object's internal, inaccessible, [prototype] (i.e.\_\_proto\_\_) property to be the constructor function's external, accessible, prototype object (every function object automatically has a prototype property).
3. It makes the this variable point to the newly created object.
4. It executes the constructor function, using the newly created object whenever this is mentioned.
5. It returns the newly created object, unless the constructor function returns a non-null object reference. In this case, that object reference is returned instead.

Consider:

|  |
| --- |
| function New(func) {  var res = {};  if (func.prototype !== null) {  res.\_\_proto\_\_ = func.prototype;  }  var ret = func.apply(res, Array.prototype.slice.call(arguments, 1));  if ((typeof ret === "object" || typeof ret === "function") && ret !== null) {  return ret;  }  return res;  } |

**63: Explain Prototype Inheritance in JavaScript?**

In a language implementing classical inheritance like Java, C# or C++ you start by creating a class--a blueprint for your objects - and then you can create new objects from that class or you can extend the class, defining a new class that augments the original class.

In JavaScript you first create an object (there is no concept of class), then you can augment your own object or create new objects from it.

Every object in Javascript has a prototype. JavaScript's inheritance system is prototypical, and not class-based. When a messages reaches an object, JavaScript will attempt to find a property in that object first, if it cannot find it then the message will be sent to the object’s prototype and so on. That behavior called prototype chain or prototype inheritance.

Constructor functions are the most used way in JavaScript to construct prototype chains. When we use new, JavaScript injects an implicit reference to the new object being created in the form of the this keyword. It also returns this reference implicitly at the end of the function.

|  |
| --- |
| function Foo() {  this.kind = ‘foo’  }  var foo = new Foo();  foo.kind //=> ‘foo’ |

**64: What does the term Transpiling stand for?**

here's no way to polyfill new syntax that has been added to the language. So the better option is to use a tool that converts your newer code into older code equivalents. This process is commonly called transpiling, a term for transforming + compiling.

Typically you insert the transpiler into your build process, similar to your code linter or your minifier. There are quite a few great transpilers for you to choose from:

Babel: Transpiles ES6+ into ES5

Traceur: Transpiles ES6, ES7, and beyond into ES5

**65: Can you give an example for destructuring an object or an array in ES6?**

Destructuring is an expression available in ES6 which enables a succinct and convenient way to extract values of Objects or Arrays and place them into distinct variables.

**Array destructuring**

|  |
| --- |
| // Variable assignment.  const foo = ['one', 'two', 'three'];  const [one, two, three] = foo;  console.log(one); // "one"  console.log(two); // "two"  console.log(three); // "three" |

**// Swapping variables**

|  |
| --- |
| let a = 1;  let b = 3;  [a, b] = [b, a];  console.log(a); // 3  console.log(b); // 1 |

**Object destructuring**

|  |
| --- |
| // Variable assignment.  const o = { p: 42, q: true };  const { p, q } = o;  console.log(p); // 42  console.log(q); // true |

**66: Explain the Prototype Design Pattern**

The Prototype Pattern creates new objects, but rather than creating non-initialized objects it returns objects that are initialized with values it copied from a prototype - or sample - object. The Prototype pattern is also referred to as the Properties pattern.

An example of where the Prototype pattern is useful is the initialization of business objects with values that match the default values in the database. The prototype object holds the default values that are copied over into a newly created business object.

Classical languages rarely use the Prototype pattern, but JavaScript being a prototypal language uses this pattern in the construction of new objects and their prototypes.

**67: Describe the JS module design pattern**

JavaScript modules are the most prevalently used design patterns for keeping particular pieces of code independent of other components. This provides loose coupling to support well-structured code.

Modules should be Immediately-Invoked-Function-Expressions (IIFE) to allow for private scopes - that is, a closure that protect variables and methods (however, it will return an object instead of a function). This is what it looks like:

|  |
| --- |
| (function() {  // declare private variables and/or functions  return {  // declare public variables and/or functions  }  })(); |

Here we instantiate the private variables and/or functions before returning our object that we want to return. Code outside of our closure is unable to access these private variables since it is not in the same scope.

**68: Can you describe the main difference between a .forEach loop and a .map() loop and why you would pick one versus the other?**

To understand the differences between the two, let's look at what each function does.

**forEach**

* Iterates through the elements in an array.
* Executes a callback for each element.
* Does not return a value.

|  |
| --- |
| const a = [1, 2, 3];  const doubled = a.forEach((num, index) => {  // Do something with num and/or index.  });  // doubled = undefined |

**map**

* Iterates through the elements in an array.
* "Maps" each element to a new element by calling the function on each element, creating a new array as a result.

|  |
| --- |
| const a = [1, 2, 3];  const doubled = a.map(num => {  return num \* 2;  });  // doubled = [2, 4, 6] |

The main difference between .forEach and .map() is that .map() returns a new array. If you need the result, but do not wish to mutate the original array, .map() is the clear choice. If you simply need to iterate over an array, forEach is a fine choice.

**69: Explain what is Hoisting in Javascript**

Hoisting is the concept in which Javascript, by default, moves all declarations to the top of the current scope. As such, a variable can be used before it has been declared.

Note that Javascript only hoists declarations and not initializations.

**70: How can you share code between files?**

This depends on the JavaScript environment.

On the client (browser environment), as long as the variables/functions are declared in the global scope (window), all scripts can refer to them. Alternatively, adopt the Asynchronous Module Definition (AMD) via RequireJS for a more modular approach.

On the server (Node.js), the common way has been to use CommonJS. Each file is treated as a module and it can export variables and functions by attaching them to the module.exports object.

ES2015 defines a module syntax which aims to replace both AMD and CommonJS. This will eventually be supported in both browser and Node environments.

**71: What are the actual uses of ES6 WeakMap?**

WeakMaps provide a way to extend objects from the outside without interfering with garbage collection. Whenever you want to extend an object but can't because it is sealed - or from an external source - a WeakMap can be applied.

WeakMap is only available for ES6 and above. A WeakMap is a collection of key and value pairs where the key must be an object.

|  |
| --- |
| var map = new WeakMap();  var pavloHero = {  first: "Pavlo",  last: "Hero"  };  var gabrielFranco = {  first: "Gabriel",  last: "Franco"  };  map.set(pavloHero, "This is Hero");  map.set(gabrielFranco, "This is Franco");  console.log(map.get(pavloHero)); //This is Hero |

The interesting aspect of the WeakMaps is the fact that it holds a weak reference to the key inside the map. A weak reference means that if the object is destroyed, the garbage collector will remove the entire entry from the WeakMap, thus freeing up memory.

**72: Explain difference between: function Person(){}, var person = Person(), and var person = new Person()?**

This question is pretty vague. My best guess at its intention is that it is asking about constructors in JavaScript. Technically speaking, function Person(){} is just a normal function declaration. The convention is to use PascalCase for functions that are intended to be used as constructors.

var person = Person() invokes the Person as a function, and not as a constructor. Invoking as such is a common mistake if it the function is intended to be used as a constructor. Typically, the constructor does not return anything, hence invoking the constructor like a normal function will return undefined and that gets assigned to the variable intended as the instance.

var person = new Person() creates an instance of the Person object using the new operator, which inherits from Person.prototype. An alternative would be to use Object.create, such as: Object.create(Person.prototype).

|  |
| --- |
| function Person(name) {  this.name = name;  }  var person = Person('John');  console.log(person); // undefined  console.log(person.name); // Uncaught TypeError: Cannot read property 'name' of undefined  var person = new Person('John');  console.log(person); // Person { name: "John" }  console.log(person.name); // "john" |

**73: Check if a given string is a isomorphic**

For two strings to be isomorphic, all occurrences of a character in string A can be replaced with another character to get string B. The order of the characters must be preserved. There must be one-to-one mapping for ever char of string A to every char of string B.

|  |
| --- |
| paper and title would return true.  egg and sad would return false.  dgg and add would return true.  isIsomorphic("egg", 'add'); // true  isIsomorphic("paper", 'title'); // true  isIsomorphic("kick", 'side'); // false  function isIsomorphic(firstString, secondString) {  // Check if the same lenght. If not, they cannot be isomorphic  if (firstString.length !== secondString.length) return false  var letterMap = {};  for (var i = 0; i < firstString.length; i++) {  var letterA = firstString[i],  letterB = secondString[i];  // If the letter does not exist, create a map and map it to the value  // of the second letter  if (letterMap[letterA] === undefined) {  letterMap[letterA] = letterB;  } else if (letterMap[letterA] !== letterB) {  // Eles if letterA already exists in the map, but it does not map to  // letterB, that means that A is mapping to more than one letter.  return false;  }  }  // If after iterating through and conditions are satisfied, return true.  // They are isomorphic  return true;  } |

**74: What is Hoisting in JavaScript?**

Hoisting is the JavaScript interpreter's action of moving all variable and function declarations to the top of the current scope. There are two types of hoisting:

variable hoisting - rare

function hoisting - more common

Wherever a var (or function declaration) appears inside a scope, that declaration is taken to belong to the entire scope and accessible everywhere throughout.

|  |
| --- |
| var a = 2;  foo(); // works because `foo()`  // declaration is "hoisted"  function foo() {  a = 3;  console.log( a ); // 3  var a; // declaration is "hoisted"  // to the top of `foo()`  }  console.log( a ); // 2 |

**75: When should you NOT use arrow functions in ES6? Name three or more cases.**

Arrow functions should NOT be used:

* When we want function hoisting - as arrow functions are anonymous.
* When we want to use this/arguments in a function - as arrow functions do not have this/arguments of their own, they depend upon their outer context.
* When we want to use named function - as arrow functions are anonymous.
* When we want to use function as a constructor - as arrow functions do not have their own this.
* When we want to add function as a property in object literal and use object in it - as we can not access this (which should be object itself).

**76: What's the difference between a variable that is: null, undefined or undeclared? How would you go about checking for any of these states?**

Undeclared variables are created when you assign a value to an identifier that is not previously created using var, let or const. Undeclared variables will be defined globally, outside of the current scope. In strict mode, a ReferenceError will be thrown when you try to assign to an undeclared variable. Undeclared variables are bad just like how global variables are bad. Avoid them at all cost! To check for them, wrap its usage in a try/catch block.

|  |
| --- |
| function foo() {  x = 1; // Throws a ReferenceError in strict mode  }  foo();  console.log(x); // 1 |

A variable that is undefined is a variable that has been declared, but not assigned a value. It is of type undefined. If a function does not return any value as the result of executing it is assigned to a variable, the variable also has the value of undefined. To check for it, compare using the strict equality (===) operator or typeof which will give the 'undefined' string. Note that you should not be using the abstract equality operator to check, as it will also return true if the value is null.

|  |
| --- |
| var foo;  console.log(foo); // undefined  console.log(foo === undefined); // true  console.log(typeof foo === 'undefined'); // true  console.log(foo == null); // true. Wrong, don't use this to check!  function bar() {}  var baz = bar();  console.log(baz); // undefined |

A variable that is null will have been explicitly assigned to the null value. It represents no value and is different from undefined in the sense that it has been explicitly assigned. To check for null, simply compare using the strict equality operator. Note that like the above, you should not be using the abstract equality operator (==) to check, as it will also return true if the value is undefined.

|  |
| --- |
| var foo = null;  console.log(foo === null); // true  console.log(typeof foo === 'object'); // true  console.log(foo == undefined); // true. Wrong, don't use this to check! |

As a personal habit, I never leave my variables undeclared or unassigned. I will explicitly assign null to them after declaring if I don't intend to use it yet. If you use a linter in your workflow, it will usually also be able to check that you are not referencing undeclared variables.

**77: What is the Temporal Dead Zone in ES6? In ES6 let and const are hoisted (like var, class and function), but there is a period between entering scope and being declared where they cannot be accessed. This period is the temporal dead zone (TDZ).**

Consider:

|  |
| --- |
| //console.log(aLet) // would throw ReferenceError  let aLet;  console.log(aLet); // undefined  aLet = 10; console.log(aLet); // 10 |

In this example the TDZ ends when aLet is declared, rather than assigned.

**78: Explain how JSONP works (and how it's not really Ajax)**

JSONP (JSON with Padding) is a method commonly used to bypass the cross-domain policies in web browsers because Ajax requests from the current page to a cross-origin domain is not allowed.

JSONP works by making a request to a cross-origin domain via a <script> tag and usually with a callback query parameter, for example: https://example.com?callback=printData. The server will then wrap the data within a function called printData and return it to the client.

|  |
| --- |
| <!-- https://mydomain.com -->  <script>  function printData(data) {  console.log(`My name is ${data.name}!`);  }  </script>  <script src="https://example.com?callback=printData"></script> |

|  |
| --- |
| // File loaded from https://example.com?callback=printData  printData({ name: 'Yang Shun' }); |

The client has to have the printData function in its global scope and the function will be executed by the client when the response from the cross-origin domain is received.

JSONP can be unsafe and has some security implications. As JSONP is really JavaScript, it can do everything else JavaScript can do, so you need to trust the provider of the JSONP data.

These days, CORS is the recommended approach and JSONP is seen as a hack.

**79: Could you compare usage of Module Pattern vs Constructor/Prototype pattern?**

The module pattern is typically used for namespacing, where you'll have a single instance acting as a store to group related functions and objects. This is a different use case from what prototyping is good for. They're not really competing with each other; you can quite happily use both together (eg put a constructor-function inside a module and say new MyNamespace.MyModule.MyClass(arguments)).

Constructor-functions and prototypes are one of the reasonable ways to implement classes and instances. They don't quite correspond to that model so you typically need to choose a particular scheme or helper method to implement classes in terms of prototypes.

80: What tools can be used to assure consistent code style?

You have plenty of options to do so:

* JSLint by Douglas Crockford
* JSHint
* ESLint
* JSCS

These tools are really helpful when developing code in teams, to enforce a given style guide and to catch common errors using static analysis.

**81: Does JavaScript have a map function to iterate over an object properties?**

Consider solution that returns a new object and leaves the original object as it is

|  |
| --- |
| var myObject = { 'a': 1, 'b': 2, 'c': 3 };  // returns a new object with the values at each key mapped using mapFn(value)  function objectMap(object, mapFn) {  return Object.keys(object).reduce(function(result, key) {  result[key] = mapFn(object[key])  return result  }, {})  }  var newObject = objectMap(myObject, function(value) {  return value \* 2  })  console.log(newObject);  // => { 'a': 2, 'b': 4, 'c': 6 }  console.log(myObject);  // => { 'a': 1, 'b': 2, 'c': 3 } |

**Or with ES6 fromEntries, entries features:**

|  |
| --- |
| const objectMap = (obj, fn) =>  Object.fromEntries(  Object.entries(obj).map(  ([k, v], i) => [k, fn(v, k, i)]  )  )  const myObject = { a: 1, b: 2, c: 3 }  console.log(objectMap(myObject, v => 2 \* v)) |

**82 When would you use import \* as X from 'X' ?**

Modules with multiple (and even single) exports which do not define a default export will need to have all exports assigned to a named variable, like:

|  |
| --- |
| import \* as whatever from 'package'; |

**83: How would you prevent Callback Hell without using promises, async or generators?**

1. Don't nest functions. Give them names and place them at the top level of your program. Keep your code shallow.
2. Use function hoisting to your advantage to move functions 'below the fold'
3. Handle every single error in every one of your callbacks. Use a linter like standard to help you with this.
4. Modularize. Create reusable functions and place them in a module to reduce the cognitive load required to understand your code. Splitting your code into small pieces like this also helps you handle errors, write tests, forces you to create a stable and documented public API for your code, and helps with refactoring.

The most important aspect of avoiding callback hell is moving functions out of the way so that the flow of the program can be more easily understood without newcomers having to wade through all the detail of the functions to get to the meat of what the program is trying to do.

JS - 4 - Expert

**84: What's the difference between ES6 Map and WeakMap?**

They both behave differently when a object referenced by their keys/values gets deleted. Lets take the below example code:

|  |
| --- |
| var map = new Map(); var weakmap = new WeakMap();  (function() {  var a = {  x: 12  };  var b = {  y: 12  };  map.set(a, 1);  weakmap.set(b, 2);  })() |

The above IIFE is executed there is no way we can reference {x: 12} and {y: 12} anymore. Garbage collector goes ahead and deletes the key b pointer from “WeakMap” and also removes {y: 12} from memory. But in case of “Map”, the garbage collector doesn’t remove a pointer from “Map” and also doesn’t remove {x: 12} from memory.

WeakMap allows garbage collector to do its task but not Map. With manually written maps, the array of keys would keep references to key objects, preventing them from being garbage collected. In native WeakMaps, references to key objects are held "weakly", which means that they do not prevent garbage collection in case there would be no other reference to the object.

**85: What is the difference between the await keyword and the yield keyword?**

yield can be considered to be the building block of await. yield takes the value it's given and passes it to the caller. The caller can then do whatever it wishes with that value (1). Later the caller may give a value back to the generator (via generator.next()) which becomes the result of the yield expression (2), or an error that will appear to be thrown by the yield expression (3).

async-await can be considered to use yield. At (1) the caller (i.e. the async-await driver - similar to the function you posted) will wrap the value in a promise using a similar algorithm to new Promise(r => r(value) (note, not Promise.resolve, but that's not a big deal). It then waits for the promise to resolve. If it fulfills, it passes the fulfilled value back at (2). If it rejects, it throws the rejection reason as an error at (3).

So the utility of async-await is this machinery that uses yield to unwrap the yielded value as a promise and pass its resolved value back, repeating until the function returns its final value.

**86: Compare Async/Await and Generators usage to achive same functionality**

* Generator function are executed yield by yield i.e one yield-expression at a time by its iterator (the next method) where as Async-await, they are executed sequential await by await.
* Async/await makes it easier to implement a particular use case of Generators.
* The return value of Generator is always {value: X, done: Boolean} where as for Async function it will always be a promise that will either resolve to the value X or throw an error.
* Async function can be decomposed into Generator and promise implementation like:



87: How to deep-freeze object in JavaScript?

If you want make sure the object is deep frozen you have to create a recursive function to freeze each property which is of type object: **Without deep freeze:**

|  |
| --- |
| let person = {  name: "Leonardo",  profession: {  name: "developer"  }  };  Object.freeze(person); // make object immutable  person.profession.name = "doctor";  console.log(person); //output { name: 'Leonardo', profession: { name: 'doctor' } } |

**With deep freeze:**

|  |
| --- |
| function deepFreeze(object) {  let propNames = Object.getOwnPropertyNames(object);  for (let name of propNames) {  let value = object[name];  object[name] = value && typeof value === "object" ?  deepFreeze(value) : value;  }  return Object.freeze(object);  }  let person = {  name: "Leonardo",  profession: {  name: "developer"  }  };  deepFreeze(person);  person.profession.name = "doctor"; // TypeError: Cannot assign to read only property 'name' of object |

**88: Is it possible to reset an ECMAScript 6 generator to its initial state?**

Once a generator enters the "completed" state it never leaves it and its associated execution context is never resumed. Any execution state associated with generator can be discarded at this point.

**89: Is JavaScript a pass-by-reference or pass-by-value language?**

It's always pass by value, but for objects the value of the variable is a reference. Because of this, when you pass an object and change its members, those changes persist outside of the function. This makes it look like pass by reference. But if you actually change the value of the object variable you will see that the change does not persist, proving it's really pass by value.

Example:

|  |
| --- |
| function changeStuff(a, b, c)  {  a = a \* 10;  b.item = "changed";  c = {item: "changed"};  }  var num = 10;  var obj1 = {item: "unchanged"};  var obj2 = {item: "unchanged"};  changeStuff(num, obj1, obj2);  console.log(num);  console.log(obj1.item);  console.log(obj2.item); |

**90: In JavaScript, why is the this operator inconsistent?**

The most important thing to understand is that a function object does not have a fixed this value -- the value of this changes depending on how the function is called. We say that a function is invoked with some a particular this value -- the this value is determined at invocation time, not definition time.

* If the function is called as a "raw" function (e.g., just do someFunc()), this will be the global object (window in a browser) (or undefined if the function runs in strict mode).
* If it is called as a method on an object, this will be the calling object.
* If you call a function with call or apply, this is specified as the first argument to call or apply.
* If it is called as an event listener, this will be the element that is the target of the event.
* If it is called as a constructor with new, this will be a newly-created object whose prototype is set to the prototype property of the constructor function.

If the function is the result of a bind operation, the function will always and forever have this set to the first argument of the bind call that produced it. (This is the single exception to the "functions don't have a fixed this" rule -- functions produced by bind actually do have an immutable this.)

**91: Can you give an example of a curry function and why this syntax offers an advantage?**

Currying is a pattern where a function with more than one parameter is broken into multiple functions that, when called in series, will accumulate all of the required parameters one at a time. This technique can be useful for making code written in a functional style easier to read and compose. It's important to note that for a function to be curried, it needs to start out as one function, then broken out into a sequence of functions that each accepts one parameter.

|  |
| --- |
| function curry(fn) {  if (fn.length === 0) {  return fn;  }  function \_curried(depth, args) {  return function(newArgument) {  if (depth - 1 === 0) {  return fn(...args, newArgument);  }  return \_curried(depth - 1, [...args, newArgument]);  };  }  return \_curried(fn.length, []);  }  function add(a, b) {  return a + b;  }  var curriedAdd = curry(add);  var addFive = curriedAdd(5);  var result = [0, 1, 2, 3, 4, 5].map(addFive); // [5, 6, 7, 8, 9, 10] |

**92: Does JavaScript pass by references or pass by values?**

Primitives are passed by value, and Objects are passed by copy of a reference.

Specifically, when you pass an object (or array) you are (invisibly) passing a reference to that object, and it is possible to modify the contents of that object, but if you attempt to overwrite the reference it will not affect the copy of the reference held by the caller - i.e. the reference itself is passed by value:

|  |
| --- |
| function replace(ref) {  ref = {}; // this code does \_not\_ affect the object passed  }  function update(ref) {  ref.key = 'newvalue'; // this code \_does\_ affect the \_contents\_ of the object  }  var a = { key: 'value' };  replace(a); // a still has its original value - it's unmodfied  update(a); // the \_contents\_ of 'a' are changed |