# JavaScript -Toptal

|  |
| --- |
| Best Way to iterate ‘Array’ --in JS |
| Native For Loop var myArray = [1, 2, 3, 4, 5];  for (var i = 0, len = myArray.length; i < len; i++) {  console.log(myArray[i] +'--'+ i);  } angular.forEach var myArray = [1, 2, 3, 4, 5];  angular.forEach(myArray, function(value, index){  console.log(value +'--'+ index)  }); |
| Best Way to break the loop |
| Native For Loop var myArray = [1, 2, 3, 4, 5];  for (var i = 0, len = numbers.length; i < len; i++) {  if (numbers[i] === 3) {  console.log('Loop is going to break.');  break;  }  console.log('Loop will continue.');  } angular.forEach var myArray = [1, 2, 3, 4, 5];  var keepGoing = true;  angular.forEach(myArray, function(value){  if(keepGoing) {  if(value == 3){  keepGoing = false;  }  }  }); |
| Best Way to iterate **object** -properties |
| for/in - loops through the properties of an object  var obj = {name: 'Jagadeesh', gender: 'male', age: 26};  for (var key in obj) {  console.log(key + " = " + obj[key]);  }  To avoid getting inherited properties, check with **hasOwnProperty** :  for (var key in obj) {  if (obj.hasOwnProperty(key)) {  console.log(key + " = " + obj[key]);  }  }  var obj = {name: 'Jagadeesh', gender: 'male', age: 26};  angular.forEach(obj, function(value, key) {  console.log(key + ': ' + value);  });  // angular.forEach() does not iterate over inherited properties. Because it internally filters using the hasOwnProperty() method. |
|  |
| How to check object is ‘null’ -or- ‘undefined’ ? |
| Difference between the 'null' and 'undefined'?   * Both indicates the absence of something * **undefined**   + variable is (not declared) -or- (value -not initialized)   + means that the variable has not been declared,or has not been given a value * **null**   + means --null object reference   + because typeof null returns 'object'   **var** testVar**;**  console**.**log**(**testVar**);** //undefined  console**.**log**(typeof** testVar**);** //undefined  console**.**log**(typeof** testVar === 'undefined'**);** //true  **var** testVar **=** null**;**  console**.**log**(**testVar**);** //null  console**.**log**(**testVar === null**);** //true  console**.**log**(typeof** testVar**);** //object |
| What is the difference between "==" and "===" |
| * === is strict --comparison operators * == is **type-converting** equality -- comparison operators   0 == false // true  0 === false // false, because they are of a different type  1 == "1" // true, automatic type conversion for value only  1 === "1" // false, because they are of a different type  null == undefined // true  null === undefined // false  '0' == false // true  '0' === false // false |
| Simple, valid variable check? |
| You can just check if the *variable* has a truthy value or not. That means  if( value ) {  // will evaluate to true if value is not:  // [ null, undefined, NaN, "" (empty string) , 0, false ]  } |

|  |
| --- |
| How to check variable -- isObject? --or-- isArray? |
| Although typeof bar === "object" *is* a reliable way of checking if bar is an object, the surprising gotcha in JavaScript is that null is *also* considered an object!  Therefore, the following code will, to the surprise of most developers, log true (not false) to the console:  var bar = null;  console.log(typeof bar === "object"); // logs true!  As long as one is aware of this, the problem can easily be avoided by also checking if bar is null:  console.log((bar !== null) && (typeof bar === "object")); // logs false  To be entirely thorough in our answer, there are two other things worth noting:  First, the above solution will return false if bar is a function. In most cases, this is the desired behavior, but in situations where you want to also return true for functions, you could amend the above solution to be:  console.log((bar !== null) && ((typeof bar === "object") || (typeof bar === "function")));  Second, the above solution will return true if bar is an array (e.g., if var bar = [];). In most cases, this is the desired behavior, since arrays are indeed objects, but in situations where you want to also false for arrays, you could amend the above solution to be: isObject? console.log((bar !== null) && (typeof bar === "object") && (toString.call(bar) !== "[object Array]"));  Or, if you’re using jQuery:  console.log((bar !== null) && (typeof bar === "object") && (! $.isArray(bar))); isArray? console.log(toString.call(bar) !== "[object Array]");  Or, if you’re using jQuery:  console.log($.isArray(bar));  Or, if you’re using AngularJs:  console.log(angular.isArray(bar)); |

|  |
| --- |
| slice vs splice |
| myArray.slice(startIndex, endIndex); // will not remove the array  myArray.splice(startIndex, noOfValsTobeRemoved); // will remove from the array  'abc'.slice(1,2) // "b"  [1, 14, 3, 77, 2, 5].slice(2, 3) // [3]  [1, 14, 3, 77, 2, 5].splice(2, 3) // [3, 77, 2] Splice  |  | | --- | | var array=[1,2,3,4,5];  console.log(array.splice(2));  // shows [3, 4, 5], returned removed item(s) as a new array object.    console.log(array);  // shows [1, 2], original array altered.    var array2=[6,7,8,9,0];  console.log(array2.splice(2,1));  // shows [8]    console.log(array2.splice(2,0));  //shows [] , as no item(s) removed.    console.log(array2);  // shows [6,7,9,0]    var array3=[11,12,13,14,15];  console.log(array3.splice(2,1,"Hello","World"));  // shows [13]    console.log(array3);  // shows [11, 12, "Hello", "World", 14, 15]    -5 -4 -3 -2 -1  | | | | |  var array4=[16,17,18,19,20];  | | | | |  0 1 2 3 4    console.log(array4.splice(-2,1,"me"));  // shows [19]    console.log(array4);  // shows [16, 17, 18, "me", 20] |  Slice  |  | | --- | | var array=[1,2,3,4,5]  console.log(array.slice(2));  // shows [3, 4, 5], returned selected element(s).    console.log(array.slice(-2));  // shows [4, 5], returned selected element(s).  console.log(array);  // shows [1, 2, 3, 4, 5], original array remains intact.    var array2=[6,7,8,9,0];  console.log(array2.slice(2,4));  // shows [8, 9]    console.log(array2.slice(-2,4));  // shows [9]    console.log(array2.slice(-3,-1));  // shows [8, 9]    console.log(array2);  // shows [6, 7, 8, 9, 0] |  substring  |  | | --- | | **var** numbers **=** "0123456789"**;**  slice**(** 3**,** 7 **)** **:** 3456  slice**(** **-**7**,** 7 **)** **:** 3456  slice**(** **-**7**,** **-**3 **)** **:** 3456  slice**(** 3 **)** **:** 3456789  slice**(** **-**7 **)** **:** 3456789  slice**(** 100**,** 101 **)** **:**  substring**(** 3**,** 7 **)** **:** 3456  substring**(** **-**7**,** 7 **)** **:** 0123456  substring**(** **-**7**,** **-**3 **)** **:**  substring**(** 3 **)** **:** 3456789  substring**(** **-**7 **)** **:** 0123456789  substring**(** 100**,** 101 **)** **:**  substr**(** 3**,** 4 **)** **:** 3456  substr**(** **-**7**,** 4 **)** **:** 3456  substr**(** 3 **)** **:** 3456789  substr**(** **-**7 **)** **:** 3456789  substr**(** 100**,** 1 **)** **:** | |
| call vs apply |
| syntax: theFunction.call(valueForThis, arg1, arg2, ...)  theFunction.apply(valueForThis, arrayOfArgs)  function theFunction(name, profession) {  console.log(name + "--" + profession + "--" + this.**age**);  }  theFunction("John", "fireman");  theFunction.call({age:20}, "Claude", "mathematician");  theFunction.apply({age:30}, ["Susan", "school teacher"]);  // John –fireman --undefined  // Susan --school teacher --20  // Claude --mathematician --30 |
| OOPS |
|  |
|  |

|  |
| --- |
| Event - bubbling (default) |
| * EVENT - BUBBLING (default) * --when u click 'grandChild' * event delegation goes: grandChild -> child -> parent -> grandParent  |  | | --- | | <!DOCTYPE html>  <html>  <head>  <style>  div{ padding: 20px; } #grand-parent{ background-color:yellow; }  #parent{ background-color:blue; } #child{ background-color:red; }  </style>  <script>  function load(){  var grandParent = document.getElementById('grand-parent');  grandParent.addEventListener("click", function(){ alert('grandParent');});  var parent = document.getElementById('parent');  parent.addEventListener("click", function(){ alert('parent');});  var child = document.getElementById('child');  child.addEventListener("click", function(){ alert('child');});  var grandChild = document.getElementById('grand-child');  grandChild.addEventListener("click", function(){ alert('grandChild');});  }  var alertFn = function(text){  alert(text);  };  </script>  </head>  <body onload="load()">  <div id="grand-parent">  <div id="parent" >    <div id="child">      <button type="button" id="grand-child">Button</button>      <div>      </div>  </div>  </body>  </html> | |

|  |
| --- |
| Event - capturing |
| * EVENT - CAPTURING * --when u click 'grandChild' * event delegation goes: grandParent -> parent -> child -> grandChild  |  | | --- | | <!DOCTYPE html>  <html>  <head>  <style>  div{ padding: 20px; } #grand-parent{ background-color:yellow; }  #parent{ background-color:blue; } #child{ background-color:red; }  </style>  <script>  function load(){  var grandParent = document.getElementById('grand-parent');  grandParent.addEventListener("click", function(){ alert('grandParent');}, true);  var parent = document.getElementById('parent');  parent.addEventListener("click", function(){ alert('parent');}, true);  var child = document.getElementById('child');  child.addEventListener("click", function(){ alert('child');}, true);  var grandChild = document.getElementById('grand-child');  grandChild.addEventListener("click", function(){ alert('grandChild');}, true);  }  var alertFn = function(text){  alert(text);  };  </script>  </head>  <body onload="load()">  <div id="grand-parent">  <div id="parent" >    <div id="child">      <button type="button" id="grand-child">Button</button>      <div>      </div>  </div>  </body>  </html> | |
|  |

|  |
| --- |
| Event - bubbling (default) (stop propogation) |
| * EVENT - BUBBLING (stop Propagation) * --when u click 'grandChild' * event delegation goes: grandChild -> will not be propagated  |  | | --- | | <!DOCTYPE html>  <html>  <head>  <style>  div{ padding: 20px; } #grand-parent{ background-color:yellow; } #parent{ background-color:blue; } #child{ background-color:red; }  </style>  <script>  function load(){  var grandParent = document.getElementById('grand-parent');  grandParent.addEventListener("click", function(event){ event.stopPropagation(); alert('grandParent');});  var parent = document.getElementById('parent');  parent.addEventListener("click", function(event){ event.stopPropagation(); alert('parent');});  var child = document.getElementById('child');  child.addEventListener("click", function(event){ event.stopPropagation(); alert('child');});  var grandChild = document.getElementById('grand-child');  grandChild.addEventListener("click", function(event){ event.stopPropagation(); alert('grandChild');});  }  var alertFn = function(text){  alert(text);  };  </script>  </head>  <body onload="load()">  <div id="grand-parent">  <div id="parent" >    <div id="child">      <button type="button" id="grand-child">Button</button>      <div>      </div>  </div>  </body>  </html> | |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| What will the code below output to the console and why?  (function(){  var a = b = 3;  })();  console.log("a defined? " + (typeof a !== 'undefined'));  console.log("b defined? " + (typeof b !== 'undefined')); |
| Since both a and b are defined within the enclosing scope of the function, and since the line they are on begins with the varkeyword, most JavaScript developers would expect typeof a and typeof b to both be *undefined* in the above example.  However, that is *not* the case. The issue here is that most developers *incorrectly* understand the statement var a = b = 3; to be shorthand for:  var b = 3;  var a = b;  But in fact, var a = b = 3; is actually shorthand for:  b = 3;  var a = b;  As a result (if you are *not* using strict mode), the output of the code snippet would be:  a defined? false  b defined? true  But how can b be defined *outside* of the scope of the enclosing function? Well, since the statement var a = b = 3; is shorthand for the statements b = 3; and var a = b;, b ends up being a global variable (since it is not preceded by the var keyword) and is therefore still in scope even outside of the enclosing function.  Note that, in strict mode (i.e., with [use strict](http://www.w3schools.com/js/js_strict.asp)), the statement var a = b = 3; will generate a runtime error of ReferenceError: b is not defined, thereby avoiding any headfakes/bugs that might othewise result.  (Yet another prime example of why you should use use strict as a matter of course in your code!) |
| What will the code below output to the console and why?  var myObject = {  foo: "bar",  func: function() {  var self = this;  console.log("outer func: this.foo = " + this.foo);  console.log("outer func: self.foo = " + self.foo);  (function() {  console.log("inner func: this.foo = " + this.foo);  console.log("inner func: self.foo = " + self.foo);  }());  }  };  myObject.func(); |
| The above code will output the following to the console:  outer func: this.foo = bar  outer func: self.foo = bar  inner func: this.foo = undefined  inner func: self.foo = bar  In the outer function, both this and self refer to myObject and therefore both can properly reference and access foo.  In the inner function, though, this no longer refers to myObject. As a result, this.foo is undefined in the inner function, whereas the reference to the local variable self remains in scope and is accessible there.  (Prior to ECMA 5, this in the inner function would refer to the global window object; whereas, as of ECMA 5, this in the inner function would be undefined.) |
| What is the significance of, and reason for, wrapping the entire content of a JavaScript source file in a function block? (Closure) |
| This is an increasingly common practice, employed by many popular JavaScript libraries (jQuery, Node.js, etc.). This technique creates a closure around the entire contents of the file which,  most importantly, creates a private namespace and thereby helps avoid potential name clashes between different JavaScript modules and libraries.  Another feature of this technique is to allow for an easily referenceable (presumably shorter) alias for a global variable. This is often used, for example, in jQuery plugins. jQuery allows you to disable the $ reference to the jQuery namespace, using jQuery.noConflict(). If this has been done, your code can still use $ employing this closure technique, as follows:  (function($) { /\* jQuery plugin code referencing $ \*/ } )(jQuery); |
| What is the significance, and what are the benefits, of including 'use strict'; at the beginning of a JavaScript source file? |
| the short and most important answer here is that use strict is a way to voluntarily enforce stricter parsing and error handling on your JavaScript code at runtime. Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions. In general, it is a good practice.  Some of the key benefits of strict mode include:   * **Makes debugging easier.** Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions, alerting you sooner to problems in your code and directing you more quickly to their source. * **Prevents accidental globals.** Without strict mode, assigning a value to an undeclared variable automatically creates a global variable with that name. This is one of the most common errors in JavaScript. In strict mode, attempting to do so throws an error. * **Eliminates this coercion**. Without strict mode, a reference to a this value of null or undefined is automatically coerced to the global. This can cause many headfakes and pull-out-your-hair kind of bugs. In strict mode, referencing a a this value of null or undefined throws an error. * **Disallows duplicate property names or parameter values.** Strict mode throws an error when it detects a duplicate named property in an object (e.g., var object = {foo: "bar", foo: "baz"};) or a duplicate named argument for a function (e.g., function foo(val1, val2, val1){}), thereby catching what is almost certainly a bug in your code that you might otherwise have wasted lots of time tracking down. * **Makes eval() safer.** There are some differences in the way eval() behaves in strict mode and in non-strict mode. Most significantly, in strict mode, variables and functions declared inside of an eval() statement are *not* created in the containing scope (they *are* created in the containing scope in non-strict mode, which can also be a common source of problems). * **Throws error on invalid usage of delete.** The delete operator (used to remove properties from objects) cannot be used on non-configurable properties of the object. Non-strict code will fail silently when an attempt is made to delete a non-configurable property, whereas strict mode will throw an error in such a case. |
| Consider the two functions below. Will they both return the same thing? Why or why not?  function foo1()  {  return {  bar: "hello"  };  }  function foo2()  {  return  {  bar: "hello"  };  } |
| Surprisingly, these two functions will *not* return the same thing. Rather:  console.log("foo1 returns:");  console.log(foo1());  console.log("foo2 returns:");  console.log(foo2());  will yield:  foo1 returns:  Object {bar: "hello"}  foo2 returns:  undefined  Not only is this surprising, but what makes this particularly gnarly is that foo2() returns undefined without any error being thrown.  The reason for this has to do with the fact that semicolons are technically optional in JavaScript (although omitting them is generally really bad form). As a result, when the line containing the return statement (with nothing else on the line) is encountered in foo2(), a semicolon is automatically inserted immediately after the return statement.  No error is thrown since the remainder of the code is perfectly valid, even though it doesn’t ever get invoked or do anything (it is simply an unused code block that defines a property bar which is equal to the string "hello").  This behavior also argues for following the convention of placing an opening curly brace at the end of a line in JavaScript, rather than on the beginning of a new line. As shown here, this becomes more than just a stylistic preference in JavaScript. |
|  |
| What is NaN? What is its type? How can you reliably test if a value is equal to NaN? |
| The NaN property represents a value that is “not a number”. This special value results from an operation that could not be performed either because one of the operands was non-numeric (e.g., "abc" / 4), or because the result of the operation is non-numeric (e.g., an attempt to divide by zero).  While this seems straightforward enough, there are a couple of somewhat surprising characteristics of NaN that can result in hair-pulling bugs if one is not aware of them.  For one thing, although NaN means “not a number”, its type is, believe it or not, Number:  console.log(typeof NaN === "number"); // logs "true"  Additionally, NaN compared to anything – even itself! – is false:  console.log(NaN === NaN); // logs "false"  A *semi-reliable* way to test whether a number is equal to NaN is with the built-in function isNaN(), but even using [isNaN() is an imperfect solution](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN#Confusing_special-case_behavior).  A better solution would either be to use value !== value, which would *only* produce true if the value is equal to NaN. Also,  ES6 offers a new [Number.isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN) function, which is a different and more reliable than the old global isNaN() function. |
| What will the code below output? Explain your answer.  console.log(0.1 + 0.2);  console.log(0.1 + 0.2 == 0.3); |
| An educated answer to this question would simply be: “You can’t be sure. it might print out “0.3” and “true”, or it might not. Numbers in JavaScript are all treated with floating point precision, and as such, may not always yield the expected results.”  The example provided above is classic case that demonstrates this issue. Surprisingly, it will print out:  0.30000000000000004  false |
| Discuss possible ways to write a function isInteger(x) that determines if x is an integer. |
| This may sound trivial and, in fact, it is trivial with ECMAscript 6 which introduces a new Number.isInteger() function for precisely this purpose. However, prior to ECMAScript 6, this is a bit more complicated, since no equivalent of the Number.isInteger()method is provided.  The issue is that, in the ECMAScript specification, integers only exist conceptually; i.e., numeric values are *always* stored as floating point values.  With that in mind, the *simplest and cleanest* pre-ECMAScript-6 solution (which is also sufficiently robust to return false even if a non-numeric value such as a string or null is passed to the function) would be the following:  function isInteger(x) { return (x^0) === x; }  The following solution would also work, although not as elegant as the one above:  function isInteger(x) { return Math.round(x) === x; }  Note that Math.ceil() or Math.floor() could be used equally well (instead of Math.round()) in the above implementation.  Or alternatively:  function isInteger(x) { return (typeof x === 'number') && (x % 1 === 0); }  One fairly common **incorrect** solution is the following:  function isInteger(x) { return parseInt(x, 10) === x; }  While this parseInt-based approach will work well for *many* values of x, once x becomes quite large, it will fail to work properly. The problem is that parseInt() coerces its first parameter to a string before parsing digits. Therefore, once the number becomes sufficiently large, its string representation will be presented in exponential form (e.g., 1e+21). Accordingly, parseInt() will then try to parse 1e+21, but will stop parsing when it reaches the e character and will therefore return a value of 1. Observe:  > String(1000000000000000000000)  '1e+21'  > parseInt(1000000000000000000000, 10)  1  > parseInt(1000000000000000000000, 10) === 1000000000000000000000  false |
| In what order will the numbers 1-4 be logged to the console when the code below is executed? Why?  (function() {  console.log(1);  setTimeout(function(){console.log(2)}, 1000);  setTimeout(function(){console.log(3)}, 0);  console.log(4);  })(); |
| The values will be logged in the following order:  1  4  3  2  Let’s first explain the parts of this that are presumably more obvious:   * 1 and 4 are displayed first since they are logged by simple calls to console.log() without any delay * 2 is displayed after 3 because 2 is being logged after a delay of 1000 msecs (i.e., 1 second) whereas 3 is being logged after a delay of 0 msecs.   OK, fine. But if 3 is being logged after a delay of 0 msecs, doesn’t that mean that it is being logged right away? And, if so, shouldn’t it be logged *before* 4, since 4 is being logged by a later line of code?  The answer has to do with properly understanding [JavaScript events and timing](http://javascript.info/tutorial/events-and-timing-depth).  The browser has an event loop which checks the event queue and processes pending events. For example, if an event happens in the background (e.g., a script onload event) while the browser is busy (e.g., processing an onclick), the event gets appended to the queue. When the onclick handler is complete, the queue is checked and the event is then handled (e.g., the onload script is executed).  Similarly, setTimeout() also puts execution of its referenced function into the event queue if the browser is busy.  When a value of zero is passed as the second argument to setTimeout(), it attempts to execute the specified function “as soon as possible”. Specifically, execution of the function is placed on the event queue to occur on the next timer tick. Note, though, that this is *not* immediate; the function is not executed until the next tick. That’s why in the above example, the call to console.log(4)occurs before the call to console.log(3) (since the call to console.log(3) is invoked via setTimeout, so it is slightly delayed). |
| Write a simple function (less than 80 characters) that returns a boolean indicating whether or not a string is a [palindrome](http://www.palindromelist.net/). |
| The following one line function will return true if str is a palindrome; otherwise, it returns false.  function isPalindrome(str) {  str = str.replace(/\W/g, '').toLowerCase();  return (str == str.split('').reverse().join(''));  }  For example:  console.log(isPalindrome("level")); // logs 'true'  console.log(isPalindrome("levels")); // logs 'false'  console.log(isPalindrome("A car, a man, a maraca")); // logs 'true'  My Solution:  var a= "level";  var aArray= a.split("");  var aReverseArray= aArray.reverse();  var b = aReverseArray.join("");  console.log(a.toLowerCase()===b.toLowerCase()); //true  console.log(a);  console.log(b); |
| Write a sum method which will work properly when invoked using either syntax below.  console.log(sum(2,3)); // Outputs 5  console.log(sum(2)(3)); // Outputs 5 |
| There are (at least) two ways to do this:  **METHOD 1**  function sum(x) {  if (arguments.length == 2) {  return arguments[0] + arguments[1];  } else {  return function(y) { return x + y; };  }  }  In JavaScript, functions provide access to an arguments object which provides access to the actual arguments passed to a function. This enables us to use the length property to determine at runtime the number of arguments passed to the function.  If two arguments are passed, we simply add them together and return.  Otherwise, we assume it was called in the form sum(2)(3), so we return an anonymous function that adds together the argument passed to sum() (in this case 2) and the argument passed to the anonymous function (in this case 3).  **METHOD 2**  function sum(x, y) {  if (y !== undefined) {  return x + y;  } else {  return function(y) { return x + y; };  }  }  When a function is invoked, JavaScript does not require the number of arguments to match the number of arguments in the function definition. If the number of arguments passed exceeds the number of arguments in the function definition, the excess arguments will simply be ignored. On the other hand, if the number of arguments passed is less than the number of arguments in the function definition, the missing arguments will have a value of undefined when referenced within the function. So, in the above example, by simply checking if the 2nd argument is undefined, we can determine which way the function was invoked and proceed accordingly. |
| Consider the following code snippet:  for (var i = 0; i < 5; i++) {  var btn = document.createElement('button');  btn.appendChild(document.createTextNode('Button ' + i));  btn.addEventListener('click', function(){ console.log(i); });  document.body.appendChild(btn);  }  (a) What gets logged to the console when the user clicks on “Button 4” and why?  (b) Provide one or more alternate implementations that will work as expected. |
| (a) No matter what button the user clicks the number 5 will *always* be logged to the console. This is because, at the point that the onclick method is invoked (for *any* of the buttons), the for loop has already completed and the variable i already has a value of 5. (Bonus points for the interviewee if they know enough to talk about how execution contexts, variable objects, activation objects, and the internal “scope” property contribute to the closure behavior.)  (b) The key to making this work is to capture the value of i at each pass through the for loop by passing it into a newly created function object. Here are three possible ways to accomplish this:  for (var i = 0; i < 5; i++) {  var btn = document.createElement('button');  btn.appendChild(document.createTextNode('Button ' + i));  btn.addEventListener('click', (function(i) {  return function() { console.log(i); };  })(i));  document.body.appendChild(btn);  }  Alternatively, you could wrap the entire call to btn.addEventListener in the new anonymous function:  for (var i = 0; i < 5; i++) {  var btn = document.createElement('button');  btn.appendChild(document.createTextNode('Button ' + i));  (function (i) {  btn.addEventListener('click', function() { console.log(i); });  })(i);  document.body.appendChild(btn);  }  Or, we could replace the for loop with a call to the array object’s native forEach method:  ['a', 'b', 'c', 'd', 'e'].forEach(function (value, i) {  var btn = document.createElement('button');  btn.appendChild(document.createTextNode('Button ' + i));  btn.addEventListener('click', function() { console.log(i); });  document.body.appendChild(btn);  });  (function(a,b){ console.log(a+'----'+b); })(1,2); |
| What will the code below output to the console and why?  var arr1 = "john".split('');  var arr2 = arr1.reverse();  var arr3 = "jones".split('');  arr2.push(arr3);  console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));  console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1)); |
| The logged output will be:  "array 1: length=5 last=j,o,n,e,s"  "array 2: length=5 last=j,o,n,e,s"  arr1 and arr2 are the same after the above code is executed for the following reasons:   * Calling an array object’s reverse() method doesn’t only *return* the array in reverse order, it also reverses the order of the array *itself* (i.e., in this case, arr1). * The reverse() method returns a reference to the array itself (i.e., in this case, arr1). As a result, arr2 is simply a reference to (rather than a copy of) arr1. Therefore, when anything is done to arr2 (i.e., when we invoke arr2.push(arr3);), arr1 will be affected as well since arr1 and arr2 are simply references to the same object.   And a couple of side points here that can sometimes trip someone up in answering this question:   * Passing an array to the push() method of another array pushes that *entire* array as a *single* element onto the end of the array. As a result, the statement arr2.push(arr3); adds arr3 in its entirety as a single element to the end of arr2 (i.e., it does *not*concatenate the two arrays, that’s what the concat() method is for). * Like Python, JavaScript honors negative subscripts in calls to array methods like slice() as a way of referencing elements at the end of the array; e.g., a subscript of -1 indicates the last element in the array, and so on. |
| What will the code below output to the console and why ?  console.log(1 + "2" + "2");  console.log(1 + +"2" + "2");  console.log(1 + -"1" + "2");  console.log(+"1" + "1" + "2");  console.log( "A" - "B" + "2");  console.log( "A" - "B" + 2); |
| The above code will output the following to the console:  "122"  "32"  "02"  "112"  "NaN2"  NaN  Here’s why…  The fundamental issue here is that JavaScript (ECMAScript) is a loosely typed language and it performs automatic type conversion on values to accommodate the operation being performed. Let’s see how this plays out with each of the above examples.  **Example 1:** 1 + "2" + "2" **Outputs:** "122" **Explanation:** The first operation to be performed in 1 + "2". Since one of the operands ("2") is a string, JavaScript assumes it needs to perform string concatenation and therefore converts the type of 1 to "1", 1 + "2" yields "12". Then, "12" + "2" yields "122".  **Example 2:** 1 + +"2" + "2" **Outputs:** "32" **Explanation:** Based on order of operations, the first operation to be performed is +"2"(the extra + before the first "2" is treated as a unary operator). Thus, JavaScript converts the type of "2" to numeric and then applies the unary + sign to it (i.e., treats it as a positive number). As a result, the next operation is now 1 + 2 which of course yields 3. But then, we have an operation between a number and a string (i.e., 3 and "2"), so once again JavaScript converts the type of the numeric value to a string and performs string concatenation, yielding "32".  **Example 3:** 1 + -"1" + "2" **Outputs:** "02" **Explanation:** The explanation here is identical to the prior example, except the unary operator is - rather than +. So "1" becomes 1, which then becomes -1 when the - is applied, which is then added to 1yielding 0, which is then converted to a string and concatenated with the final "2" operand, yielding "02".  **Example 4:** +"1" + "1" + "2" **Outputs:** "112" **Explanation:** Although the first "1" operand is typecast to a numeric value based on the unary + operator that precedes it, it is then immediately converted back to a string when it is concatenated with the second "1" operand, which is then concatenated with the final "2" operand, yielding the string "112".  **Example 5:** "A" - "B" + "2" **Outputs:** "NaN2" **Explanation:** Since the - operator can not be applied to strings, and since neither "A" nor "B" can be converted to numeric values, "A" - "B" yields NaN which is then concatenated with the string "2" to yield “NaN2”.  **Example 6:** "A" - "B" + 2 **Outputs:** NaN **Explanation:** As exlained in the previous example, "A" - "B" yields NaN. But any operator applied to NaN with any other numeric operand will still yield NaN. |
| The following recursive code will cause a stack overflow if the array list is too large. How can you fix this and still retain the recursive pattern?  var list = readHugeList();  var nextListItem = function() {  var item = list.pop();  if (item) {  // process the list item...  nextListItem();  }  }; |
| The potential stack overflow can be avoided by modifying the nextListItem function as follows:  var list = readHugeList();  var nextListItem = function() {  var item = list.pop();  if (item) {  // process the list item...  setTimeout( nextListItem, 0 );  }  };  The stack overflow is eliminated because the event loop handles the recursion, not the call stack. When nextListItem runs, if item is not null, the timeout function (nextListItem) is pushed to the event queue and the function exits, thereby leaving the call stack clear. When the event queue runs its timed-out event, the next item is processed and a timer is set to again invoke nextListItem. Accordingly, the method is processed from start to finish without a direct recursive call, so the call stack remains clear, regardless of the number of iterations. |

|  |
| --- |
| What is a “closure” in JavaScript? Provide an example. |
| A closure is an inner function that has access to the variables in the outer (enclosing) function’s scope chain. The closure has access to variables in three scopes; specifically: (1) variable in its own scope, (2) variables in the enclosing function’s scope, and (3) global variables.  Here is a simple example:  var globalVar = "xyz";  (function outerFunc(outerArg) {  var outerVar = 'a';    (function innerFunc(innerArg) {  var innerVar = 'b';    console.log(  "outerArg = " + outerArg + "\n" +  "innerArg = " + innerArg + "\n" +  "outerVar = " + outerVar + "\n" +  "innerVar = " + innerVar + "\n" +  "globalVar = " + globalVar);    })(456);  })(123);  In the above example, variables from innerFunc, outerFunc, and the global namespace are **all** in scope in the innerFunc. The above code will therefore produce the following output:  outerArg = 123  innerArg = 456  outerVar = a  innerVar = b  globalVar = xyz |
| What will be the output of the following code:  for (var i = 0; i < 5; i++) {  setTimeout(function() { console.log(i); }, i \* 1000 );  }  Explain your answer. How could the use of closures help here? |
| The code sample shown will ***not*** display the values 0, 1, 2, 3, and 4 as might be expected; rather, it will display 5, 5, 5, 5, and 5.  The reason for this is that each function executed within the loop will be executed *after* the entire loop has completed and *all* will therefore reference the *last* value stored in i, which was 5.  [**Closures**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures) can be used to prevent this problem by creating a unique scope for each iteration, storing each unique value of the variable within its scope, as follows:  for (var i = 0; i < 5; i++) {  (function(x) {  setTimeout(function() { console.log(x); }, x \* 1000 );  })(i);  }  This will produce the presumably desired result of logging 0, 1, 2, 3, and 4 to the console. |
| What would the following lines of code output to the console?  console.log("0 || 1 = "+(0 || 1));  console.log("1 || 2 = "+(1 || 2));  console.log("0 && 1 = "+(0 && 1));  console.log("1 && 2 = "+(1 && 2)); |
| The code will output the following four lines:  0 || 1 = 1  1 || 2 = 1  0 && 1 = 0  1 && 2 = 2  In JavaScript, both || and && are logical operators that return the first fully-determined “logical value” when evaluated from left to right.  **The or (||) operator.** In an expression of the form X||Y, X is first evaluated and interpreted as a boolean value. If this boolean value is true, then true (1) is returned and Y is not evaluated, since the “or” condition has already been satisfied. If this boolean value is “false”, though, we still don’t know if X||Y is true or false until we evaluate Y, and interpret it as a boolean value as well.  Accordingly, 0 || 1 evaluates to true (1), as does 1 || 2.  **The and (&&) operator.** In an expression of the form X&&Y, X is first evaluated and interpreted as a boolean value. If this boolean value is false, then false (0) is returned and Y is not evaluated, since the “and” condition has already failed. If this boolean value is “true”, though, we still don’t know if X&&Y is true or false until we evaluate Y, and interpret it as a boolean value as well.  However, the interesting thing with the && operator is that when an expression is evaluated as “true”, then the expression itself is returned. This is fine, since it counts as “true” in logical expressions, but also can be used to return that value when you care to do so. This explains why, somewhat surprisingly, 1 && 2 returns 2 (whereas you might it expect it to return true or 1). |
| What will be the output when the following code is executed? Explain.  console.log(false == '0')  console.log(false === '0') |
| The code will output:  true  false  In JavaScript, there are two sets of equality operators. The triple-equal operator === behaves like any traditional equality operator would: evaluates to true if the two expressions on either of its sides have the same type and the same value. The double-equal operator, however, tries to coerce the values before comparing them. It is therefore generally good practice to use the === rather than ==. The same holds true for !== vs !=. |
| What is the output out of the following code? Explain your answer.  var a={},  b={key:'b'},  c={key:'c'};  a[b]=123;  a[c]=456;  console.log(a[b]);  The output of this code will be 456 (*not* 123).  The reason for this is as follows: When setting an object property, JavaScript will implicitly **stringify** the parameter value. In this case, since b and c are both objects, they will *both* be converted to "[object Object]". As a result, a[b] anda[c] are both equivalent to a["[object Object]"] and can be used interchangeably. Therefore, setting or referencing a[c] is precisely the same as setting or referencing a[b]. |
| What will the following code output to the console:  console.log((function f(n){  return ((n > 1) ? n \* f(n-1) : n)}  )(10)); |
| The code will output the value of 10 factorial (i.e., 10!, or 3,628,800).  function f(n){  if(n > 1){  return n \* f(n-1);  }else{  return n;  }  )(10));  1! = 1  2! = 1×2 = 2  3! = 1×2×3 = 6  4! = 1×2×3×4 = 24  5! = 1×2×3×4×5 = 120  --  10! = 1×2×3×4×5x6x7x8x9x10 = 120  Here’s why:  The named function f() calls itself recursively, until it gets down to calling f(1) which simply returns 1. Here, therefore, is what this does:  f(1): returns n, which is 1  f(2): returns 2 \* f(1), which is 2  f(3): returns 3 \* f(2), which is 6  f(4): returns 4 \* f(3), which is 24  f(5): returns 5 \* f(4), which is 120  f(6): returns 6 \* f(5), which is 720  f(7): returns 7 \* f(6), which is 5040  f(8): returns 8 \* f(7), which is 40320  f(9): returns 9 \* f(8), which is 362880  f(10): returns 10 \* f(9), which is 3628800 |
| Consider the code snippet below. What will the console output be and why?  (function(x) {  return (function(y) {  console.log(x);  })(2)  })(1); |
| The output will be 1, even though the value of x is never set in the inner function. Here’s why:  As explained in our [JavaScript Hiring Guide](https://www.toptal.com/javascript#hiring-guide), a **closure** is a function, along with all variables or functions that were in-scope at the time that the closure was created. In JavaScript, a closure is implemented as an “inner function”; i.e., a function defined within the body of another function. An important feature of closures is that an inner function still has access to the outer function’s variables.  Therefore, in this example, since x is not defined in the inner function, the scope of the outer function is searched for a defined variable x, which is found to have a value of 1. |
| What will the following code output to the console and why:  var hero = {  \_name: 'John Doe',  getSecretIdentity: function (){  return this.\_name;  }  };  var stoleSecretIdentity = hero.getSecretIdentity;  console.log(stoleSecretIdentity());  console.log(hero.getSecretIdentity());  What is the issue with this code and how can it be fixed. |
| The code will output:  undefined  John Doe  The first console.log prints undefined because we are extracting the method from the hero object, so stoleSecretIdentity() is being invoked in the global context (i.e., the window object) where the \_name property does not exist.  One way to fix the stoleSecretIdentity() function is as follows:  var stoleSecretIdentity = hero.getSecretIdentity.bind(hero); |
| Create a function that, given a DOM Element on the page, will visit the element itself and *all* of its descendents (*not just its immediate children*). For each element visited, the function should pass that element to a provided callback function.  The arguments to the function should be:  • a DOM element  • a callback function (that takes a DOM element as its argument) |
| Visiting all elements in a tree (DOM) is a classic [Depth-First-Search algorithm](https://en.wikipedia.org/wiki/Depth-first_search) application. Here’s an example solution:  function Traverse(p\_element,p\_callback) {  p\_callback(p\_element);  var list = p\_element.children;  for (var i = 0; i < list.length; i++) {  Traverse(list[i],p\_callback); // recursive call  }  } |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

|  |
| --- |
| DOM (Document Object Model) |
| HTML DOM |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |