**Question 1**

**What is 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 script will stop executing. However, if you use typeof undeclared\_variable, then it will return undefined.

Before getting further into this, let's first understand the difference between declaration and definition.

Let's say var x is a declaration because you have not defined what value it holds yet, but you have declared its existence and the need for memory allocation.

> var x; // declaring x

> console.log(x); //output: undefined

Here var x = 1 is both a declaration and definition (also we can say we are doing an initialisation). In the example above, the declaration and assignment of value happen inline for variable x. In JavaScript, every variable or function declaration you bring to the top of its current scope is called hoisting.

The assignment happens in order, so when we try to access a variable that is declared but not defined yet, we will get the result undefined.

var x; // Declaration

if(typeof x === 'undefined') // Will return true

If a variable that is neither declared nor defined, when we try to reference such a variable we'd get the result not defined.

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

**Question 2**

**What will be the output of the code below?**

var y = 1;

if (function f(){}) {

y += typeof f;

}

console.log(y);

The output would be 1undefined. The if condition statement evaluates using eval, soeval(function f(){}) returns function f(){} (which is true). Therefore, inside the if statement, executing typeof f returns undefined because the if statement code executes at run time, and the statement inside the if condition is evaluated during run time.

var k = 1;

if (1) {

eval(function foo(){});

k += typeof foo;

}

console.log(k);

The code above will also output 1undefined.

var k = 1;

if (1) {

function foo(){};

k += typeof foo;

}

console.log(k); // output 1function

**Question 3**

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

One of the drawbacks of creating true private methods in JavaScript is that they are very memory-inefficient, as 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() {

increaseSlary();

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 its own copy of the increaseSalary private method.

So, as a recommendation, don’t use private methods unless it’s necessary.

**Question 4**

**What is a “closure” in JavaScript? Provide an example**

A closure is a function defined inside another function (called the parent function), and has access to variables that are declared and defined in the parent function scope.

The closure has access to variables in three scopes:

* Variables declared in their own scope
* Variables declared in a parent function scope
* Variables declared in the global namespace

var globalVar = "abc";

// Parent self invoking function

(function outerFunction (outerArg) { // begin of scope outerFunction

// Variable declared in outerFunction function scope

var outerFuncVar = 'x';

// Closure self-invoking function

(function innerFunction (innerArg) { // begin of scope innerFunction

// variable declared in innerFunction function scope

var innerFuncVar = "y";

console.log(

"outerArg = " + outerArg + "\n" +

"outerFuncVar = " + outerFuncVar + "\n" +

"innerArg = " + innerArg + "\n" +

"innerFuncVar = " + innerFuncVar + "\n" +

"globalVar = " + globalVar);

}// end of scope innerFunction)(5); // Pass 5 as parameter

}// end of scope outerFunction )(7); // Pass 7 as parameter

innerFunction is closure that is defined inside outerFunction and has access to all variables declared and defined in the outerFunction scope. In addition, the function defined inside another function as a closure will have access to variables declared in the global namespace.

Thus, the output of the code above would be:

outerArg = 7

outerFuncVar = x

innerArg = 5

innerFuncVar = y

globalVar = abc

[*Schedule a Mock Interview with a Lead Engineer*](https://www.codementor.io/gig/tqcbnk49/technical-interview-practice?utm_source=tutorial&utm_medium=article&utm_term=gigs&utm_content=tutorial_javascript&utm_campaign=cm_internal_ad)

**Question 5**

**Write a**mul**function which will produce the following outputs when invoked:**

javascript console.log(mul(2)(3)(4)); // output : 24 console.log(mul(4)(3)(4)); // output : 48

Below is the answer followed by an explanation to how it works:

function mul (x) {

return function (y) { // anonymous function

return function (z) { // anonymous function

return x \* y \* z;

};

};

}

Here the mul function accepts the first argument and returns an anonymous function, which takes the second parameter and returns another anonymous function that will take the third parameter and return the multiplication of the arguments that have been passed.

In JavaScript, a function defined inside another one has access to the outer function's variables. Therefore, a function is a first-class object that can be returned by other functions as well and be passed as an argument in another function.

* A function is an instance of the Object type
* A function can have properties and has a link back to its constructor method
* A function can be stored as a variable
* A function can be pass as a parameter to another function
* A function can be returned from another function

**Question 6**

**How to empty an array in JavaScript ?**

For instance,

var arrayList = ['a','b','c','d','e','f'];

**How can we empty the array above?**

There are a couple ways we can use to empty an array, so let's discuss them all.

**Method 1**

arrayList = []

Above code will set the variable arrayList to a new empty array. This is recommended if you don't have **references to the original array** arrayList anywhere else, because it will actually create a new, empty array. You should be careful with this method of emptying the array, because if you have referenced this array from another variable, then the original reference array will remain unchanged.

For Instance,

var arrayList = ['a','b','c','d','e','f']; // Created array

var anotherArrayList = arrayList; // Referenced arrayList by another variable

arrayList = []; // Empty the array

console.log(anotherArrayList); // Output ['a','b','c','d','e','f']

**Method 2**

arrayList.length = 0;

The code above will clear the existing array by setting its length to 0. This way of emptying the array also updates all the reference variables that point to the original array. Therefore, this method is useful when you want to update all reference variables pointing to arrayList.

For Instance,

var arrayList = ['a','b','c','d','e','f']; // Created array

var anotherArrayList = arrayList; // Referenced arrayList by another variable

arrayList.length = 0; // Empty the array by setting length to 0

console.log(anotherArrayList); // Output []

**Method 3**

arrayList.splice(0, arrayList.length);

The implementation above will also work perfectly. This way of emptying the array will also update all the references to the original array.

var arrayList = ['a','b','c','d','e','f']; // Created array

var anotherArrayList = arrayList; // Referenced arrayList by another variable

arrayList.splice(0, arrayList.length); // Empty the array by setting length to 0

console.log(anotherArrayList); // Output []

**Method 4**

while(arrayList.length){

arrayList.pop();

}

The implementation above can also empty arrays, but it is usually not recommended to use this method often.

**Question 7**

**How do you check if an object is an array or not?**

The best way to find out whether or not an object is an instance of a particular class is to use the toString method from Object.prototype:

var arrayList = [1,2,3];

One of the best use cases of type-checking an object is when we do method overloading in JavaScript. For example, let's say we have a method called greet, which takes one single string and also a list of strings. To make our greet method workable in both situations, we need to know what kind of parameter is being passed. Is it a single value or a list of values?

function greet(param){

if(){ // here have to check whether param is array or not

}else{

}

}

However, as the implementation above might not necessarily check the type for arrays, we can check for a single value string and put some array logic code in the else block. For example:

function greet(param){

if(typeof param === 'string'){

}else{

// If param is of type array then this block of code would execute

}

}

Now it's fine we can go with either of the aforementioned two implementations, but when we have a situation where the parameter can be single value, array, and object type, we will be in trouble.

Coming back to checking the type of an object, as mentioned previously we can useObject.prototype.toString

if( Object.prototype.toString.call( arrayList ) === '[object Array]' ) {

console.log('Array!');

}

If you are using jQuery, then you can also use the jQuery isArray method:

if($.isArray(arrayList)){

console.log('Array');

}else{

console.log('Not an array');

}

FYI, jQuery uses Object.prototype.toString.call internally to check whether an object is an array or not.

In modern browsers, you can also use

Array.isArray(arrayList);

Array.isArray is supported by Chrome 5, Firefox 4.0, IE 9, Opera 10.5 and Safari 5

**Question 8**

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

var output = (function(x){

delete x;

return x;

})(0);

console.log(output);

The output would be 0. The delete operator is used to delete properties from an object. Here xis not an object but a **local variable**. delete operators don't affect local variables.

**Question 9**

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

var x = 1;

var output = (function(){

delete x;

return x;

})();

console.log(output);

The output would be 1. The delete operator is used to delete the property of an object. Here x is not an object, but rather it's the **global variable** of type number.

**Question 10**

**What will be the output of the code below?**

var x = { foo : 1};

var output = (function(){

delete x.foo;

return x.foo;

})();

console.log(output);

The output would be undefined. The delete operator is used to delete the property of an object. Here, x is an object which has the property foo, and as it is a self-invoking function, we will delete the foo property from object x. After doing so, when we try to reference a deleted property foo, the result isundefined.

**Question 11**

**What will be the output of the code below?**

var Employee = {

company: 'xyz'

}

var emp1 = Object.create(Employee);

delete emp1.company

console.log(emp1.company);

The output would be xyz. Here, emp1 object has company as its **prototype** property. The deleteoperator doesn't delete prototype property.

emp1 object doesn't have **company** as its own property. You can test itconsole.log(emp1.hasOwnProperty('company')); //output : false. However, we can delete thecompany property directly from theEmployee object using delete Employee.company. Or, we can also delete the emp1 object using the \_\_proto\_\_ property delete emp1.\_\_proto\_\_.company.

**Question 12**

**What is**undefined x 1**in JavaScript?**

var trees = ["redwood","bay","cedar","oak","maple"];

delete trees[3];

When you run the code above and type console.log(trees); into your Chrome developer console, you will get ["redwood", "bay", "cedar", undefined × 1, "maple"]. When you run the code in Firefox's browser console, you will get["redwood", "bay", "cedar", undefined, "maple"]. Thus, it's clear that the Chrome browser has its own way of displaying uninitialised indexes in arrays. However, when you checktrees[3] === undefined in both browsers, you will get similar output as true.

**Note:** Please remember you do not need to check for the uninitialised index of array intrees[3] === 'undefined × 1', as it will give you an error. 'undefined × 1' is just way of displaying an array's uninitialised index in Chrome.

**Question 13**

**What will be the output of the code below?**

var trees = ["xyz","xxxx","test","ryan","apple"];

delete trees[3];

console.log(trees.length);

The output would be 5. When we use the delete operator to delete an array element, the array length is not affected from this. This holds even if you deleted all elements of an array using thedelete operator.

In other words, when the delete operator removes an array element, that deleted element is not longer present in array. In place of value at deleted index undefined x 1 in **chrome** andundefined is placed at the index. If you do console.log(trees) output["xyz", "xxxx", "test", undefined × 1, "apple"] in Chrome and in Firefox["xyz", "xxxx", "test", undefined, "apple"].

**Question 14**

**What will be the output of the code below?**

var bar = true;

console.log(bar + 0);

console.log(bar + "xyz");

console.log(bar + true);

console.log(bar + false);

The code will output 1, "truexyz", 2, 1. Here's a general guideline for addition operators:

* Number + Number -> Addition
* Boolean + Number -> Addition
* Boolean + Number -> Addition
* Number + String -> Concatenation
* String + Boolean -> Concatenation
* String + String -> Concatenation

**Question 15**

**What will be the output of the code below?**

var z = 1, y = z = typeof y;

console.log(y);

The output would be undefined. According to the associativity rule, operators with the same precedence are processed based on the associativity property of the operator. Here, the associativity of the assignment operator is Right to Left, so typeof y will evaluate first , which is undefined. It will be assigned to z, and then y would be assigned the value of z and then zwould be assigned the value 1.

**Question 16**

**What will be the output of the code below?**

// NFE (Named Function Expression

var foo = function bar(){ return 12; };

typeof bar();

The output would be Reference Error. To make the code above work, you can re-write it as follows:

**Sample 1**

var bar = function(){ return 12; };

typeof bar();

or

**Sample 2**

function bar(){ return 12; };

typeof bar();

A function definition can have only one reference variable as its function name. In **sample 1**,bar's reference variable points to anonymous function. In **sample 2**, the function's definition is the name function.

var foo = function bar(){

// foo is visible here

// bar is visible here

console.log(typeof bar()); // Work here :)

};

// foo is visible here

// bar is undefined here

**Question 17**

**What is the difference between the function declarations below?**

var foo = function(){

// Some code

};

function bar(){

// Some code

};

The main difference is the function foo is defined at run-time whereas function bar is defined at parse time. To understand this in better way, let's take a look at the code below:

Run-Time function declaration

<script>

foo(); // Calling foo function here will give an Error

var foo = function(){

console.log("Hi I am inside Foo");

};

</script>

<script>

Parse-Time function declaration

bar(); // Calling foo function will not give an Error

function bar(){

console.log("Hi I am inside Foo");

};

</script>

Another advantage of this first-one way of declaration is that you can declare functions based on certain conditions. For example:

<script>

if(testCondition) {// If testCondition is true then

var foo = function(){

console.log("inside Foo with testCondition True value");

};

}else{

var foo = function(){

console.log("inside Foo with testCondition false value");

};

}

</script>

However, if you try to run similar code using the format below, you'd get an error:

<script>

if(testCondition) {// If testCondition is true then

function foo(){

console.log("inside Foo with testCondition True value");

};

}else{

function foo(){

console.log("inside Foo with testCondition false value");

};

}

</script>

**Question 18**

**What is function hoisting in JavaScript?**

**Function Expression**

var foo = function foo(){

return 12;

};

In JavaScript, variable and functions are hoisted. Let's take function hoisting first. Basically, the JavaScript interpreter looks ahead to find all variable declarations and then hoists them to the top of the function where they're declared. For example:

foo(); // Here foo is still undefined

var foo = function foo(){

return 12;

};

Behind the scene of the code above looks like this:

javascript var foo = undefined; foo(); // Here foo is undefined foo = function foo(){ / Some code stuff }javascript var foo = undefined; foo = function foo(){ / Some code stuff } foo(); // Now foo is defined here

**Question 19**

**What will be the output of code below?**

var salary = "1000$";

(function () {

console.log("Original salary was " + salary);

var salary = "5000$";

console.log("My New Salary " + salary);

})();

The output would be undefined, 5000$. Newbies often get tricked by JavaScript's hoisting concept. In the code above, you might be expecting salary to retain its value from the outer scope until the point that salary gets re-declared in the inner scope. However, due to hoisting, the salary value was undefined instead. To understand this better, have a look of the code below:

var salary = "1000$";

(function () {

var salary = undefined;

console.log("Original salary was " + salary);

salary = "5000$";

console.log("My New Salary " + salary);

})();

salary variable is hoisted and declared at the top in the function's scope. The console.log inside returns undefined. After the console.log, salary is redeclared and assigned 5000$.

**Question 20**

**What is the**instanceof**operator in JavaScript? What would be the output of the code below?**

function foo(){

return foo;

}

new foo() instanceof foo;

Here, instanceof operator checks the current object and returns true if the object is of the specified type.

For Example:

var dog = new Animal();

dog instanceof Animal // Output : true

Here dog instanceof Animal is true since dog inherits from Animal.prototype.

var name = new String("xyz");

name instanceof String // Output : true

Here name instanceof String is true since dog inherits from String.prototype. Now let's understand the code below:

function foo(){

return foo;

}

new foo() instanceof foo;

Here function foo is returning foo, which again points to function foo.

function foo(){

return foo;

}

var bar = new foo();

// here bar is pointer to function foo(){return foo}.

So the new foo() instanceof foo return false;

[Ref Link](http://stackoverflow.com/questions/2449254/what-is-the-instanceof-operator-in-javascript)

**Question 21**

**If we have a JavaScript associative array**

var counterArray = {

A : 3,

B : 4

};

counterArray["C"] = 1;

**How can we calculate the length of the above associative array's**counterArray**?**

There are no in-built functions and properties available to calculate the length of associative array object here. However, there are other ways by which we can calculate the length of an associative array object. In addition to this, we can also extend an Object by adding a method or property to the prototype in order to calculate length. However, extending an object might break enumeration in various libraries or might create cross-browser issues, so it's not recommended unless it's necessary. Again, there are various ways by which we can calculate length.

Object has the keys method which can be used to calculate the length of an object:

Object.keys(counterArray).length // Output 2

We can also calculate the length of an object by iterating through an object and by counting the object's own property.

function getSize(object){

var count = 0;

for(key in object){

// hasOwnProperty method check own property of object

if(object.hasOwnProperty(key)) count++;

}

return count;

}

We can also add a length method directly on Object:

Object.length = function(){

var count = 0;

for(key in object){

// hasOwnProperty method check own property of object

if(object.hasOwnProperty(key)) count++;

}

return count;

}

//Get the size of any object using

console.log(Object.length(counterArray))

**Bonus**: We can also use Underscore (recommended, As it's lightweight) to calculate object length.

Question 22.

What is a potential pitfall with using typeof bar === "object" to determine if bar is an object? How can this pitfall be avoided?

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:

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)));

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!)

[Comment](https://www.toptal.com/javascript/interview-questions)

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?

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, perhaps 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).

[Comment](https://www.toptal.com/javascript/interview-questions)

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/).

View the answer →

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.

[Comment](https://www.toptal.com/javascript/interview-questions)

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);

});

[Comment](https://www.toptal.com/javascript/interview-questions)

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.

[Comment](https://www.toptal.com/javascript/interview-questions)

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);

View the answer →

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.

[Comment](https://www.toptal.com/javascript/interview-questions)

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).

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);

[Comment](https://www.toptal.com/javascript/interview-questions)

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)

View the answer →

\* There is more to interviewing than tricky technical questions, so these are intended merely as a guide. Not every “A” candidate worth hiring will be able to answer them all, nor does answering them all guarantee an “A” candidate. At the end of the day, [hiring remains an art, a science — and a lot of work](https://www.toptal.com/freelance/in-search-of-the-elite-few-finding-and-hiring-the-best-developers-in-the-industry).

**What is the difference between ViewState and SessionState?**

‘ViewState’ is specific to a page in a session.

‘SessionState’ is specific to user specific data that can be accessed across all pages in the web application.

**Explain how can you submit a form using JavaScript?**

To submit a form using JavaScript use document.form[0].submit();

document.form[0].submit();

**What are all the looping structures in JavaScript?**

Following are looping structures in Javascript:

* For
* While
* do-while loops

**What is the use of Void(0)?**

* Void(0) is used to prevent the page from refreshing and parameter “zero” is passed while calling.
* Void(0) is used to call another method without refreshing the page.

**What are escape characters?**

* Escape characters (Backslash) is used when working with special characters like single quotes, double quotes, apostrophes and ampersands. Place backslash before the characters to make it display.

|  |  |
| --- | --- |
| 1  2  3 | document.write "I m a "good" boy"    document.write "I m a \"good\" boy" |
|  |  |

**What are JavaScript Cookies?**

Cookies are the small test files stored in a computer and it gets created when the user visits the websites to store information that they need. Example could be User Name details and shopping cart information from the previous visits.

## The instanceof operator

The instanceof allows to check if the object is created by given constructor:

|  |  |
| --- | --- |
| 1 | function Rabbit() { } |
| 2 | var rabbit = new Rabbit | |

|  |  |
| --- | --- |
|  |  |
| 4 | alert(rabbit instanceof Rabbit) // true | |

**The instanceof has nothing to do with the constructor property. It follows the \_\_proto\_\_ chain instead.**

The logic behind obj instanceof F:

1. Get obj.\_\_proto\_\_
2. Compare obj.\_\_proto\_\_ against F.prototype
3. If no match then set temporarily obj = obj.\_\_proto\_\_ and repeat step 2 until either match is found or the chain ends.

* Give a high level overview of how JavaScript interacts with the DOM.
* What are some different ways that JavaScript code can be embedded into or linked into an XHTML page?
* Are objects passed by value or by reference? Does the type of object matter?
* Is JavaScript an object-oriented language? Why or why not?
* What is your favorite way to debug JavaScript?
* What is JSON? What are some common uses of JSON?
* What is the difference between undefined and null?
* What does the isNaN() function do?
* What will the following code do?
* function printMe() { var b = 5; alert (a+b); }
* var a = 3; var b = 2;
* printMe();

alert(a+b);

* What is a for…in loop?
* What is the difference between window.onload and onDocumentReady? Which are you (typically) more likely to use?
* What are a few different ways to attach handlers (e.g. click, blur, etc.) to a DOM element? Are any better than the others?
* What's unique about functions in JavaScript? (The answer you're looking for here is that they're first class objects; e.g. you can pass a function as an argument to another function, just like you could pass an object. Explore this concept with the interviewee to see how much they know.)
* What's an anonymous function?
* What is JavaScript namespacing? Why would you want to use it?

## jQuery

* Basic: what is jQuery?
* How do you select a DOM element (or elements) in jQuery?
* What does it mean to chain functions in jQuery?
* **1. Difference between window.onload and onDocumentReady?**
* The onload event does not fire until every last piece of the page is loaded, this includes css and images, which means there’s a huge delay before any code is executed.  
  That isnt what we want. We just want to wait until the DOM is loaded and is able to be manipulated. onDocumentReady allows the programmer to do that.
* **2. What is the difference between == and === ?**
* The == checks for value equality, but === checks for both type and value.
* **3. What does “1”+2+4 evaluate to? What about 5 + 4 + “3”?**
* Since 1 is a string, everything is a string, so the result is 124. In the second case, its 93.
* **4. What is the difference between undefined value and null value?**
* undefined means a variable has been declared but has not yet been assigned a value. On the other hand, null is an assignment value. It can be assigned to a variable as a representation of no value.  
  Also, undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object.  
  Unassigned variables are initialized by JavaScript with a default value of undefined. JavaScript never sets a value to null. That must be done programmatically.
* **5. How do you change the style/class on any element?**
* document.getElementById(“myText”).style.fontSize = “20”;  
  -or-  
  document.getElementById(“myText”).className = “anyclass”;
* **6. What are Javascript closures?When would you use them?**
* Two one sentence summaries:
* \* a closure is the local variables for a function – kept alive after the function has returned, or  
  \* a closure is a stack-frame which is not deallocated when the function returns.
* A closure takes place when a function creates an environment that binds local variables to it in such a way that they are kept alive after the function has returned. A closure is a special kind of object that combines two things: a function, and any local variables that were in-scope at the time that the closure was created.
* The following code returns a reference to a function:
* function sayHello2(name) {  
  var text = ‘Hello ‘ + name; // local variable  
  var sayAlert = function() { alert(text); }  
  return sayAlert;  
  }
* Closures reduce the need to pass state around the application. The inner function has access to the variables in the outer function so there is no need to store the information somewhere that the inner function can get it.
* This is important when the inner function will be called after the outer function has exited. The most common example of this is when the inner function is being used to handle an event. In this case you get no control over the arguments that are passed to the function so using a closure to keep track of state can be very convenient.
* **7. What is unobtrusive javascript? How to add behavior to an element using javascript?**
* Unobtrusive Javascript refers to the argument that the purpose of markup is to describe a document’s structure, not its programmatic behavior and that combining the two negatively impacts a site’s maintainability. Inline event handlers are harder to use and maintain, when one needs to set several events on a single element or when one is using event delegation.

|  |  |
| --- | --- |
| 1 | <input type="text" name="date" /> |

* Say an input field with the name “date” had to be validated at runtime:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | document.getElementsByName("date")[0].                     addEventListener("change", validateDate, false);    function validateDate(){  // Do something when the content of the 'input' element with the name 'date' is changed.  } |

* Although there are some browser inconsistencies with the above code, so programmers usually go with a javascript library such as JQuery or YUI to attach behavior to an element like above.
* **8.  What is Javascript namespacing? How and where is it used?**
* Using global variables in Javascript is evil and a bad practice. That being said, namespacing is used to bundle up all your functionality using a unique name. In JavaScript, a namespace is really just an object that you’ve attached all further methods, properties and objects. It promotes modularity and code reuse in the application.
* **9.  What datatypes are supported in Javascript?**  
  Number, String, Undefined, null, Boolean
* 10. What is the difference between innerHTML and append() in JavaScript?
* InnerHTML is not standard, and its a String. The DOM is not, and although innerHTML is faster and less verbose, its better to use the DOM methods like appendChild(), firstChild.nodeValue, etc to alter innerHTML content.

What is difference between jQuery.get() and jQuery.ajax() method? (ajax is more powerful and configurable, allows you to specify how long to wait and how to handle error, get() is a speciallization to over ajax just to retrieve some data)  
  
What is method chaining in jQuery? what is benefit of using method chaining ( clean and concise code, single search over DOM so better performance)  
  
what happen if you return false from a jQueyr event handler? (it used to stop the event bubbling up)   
  
which one is more efficient, document.getElementbyId( "myId") or $("#myId)? first one because its direct call to JavaScript engine