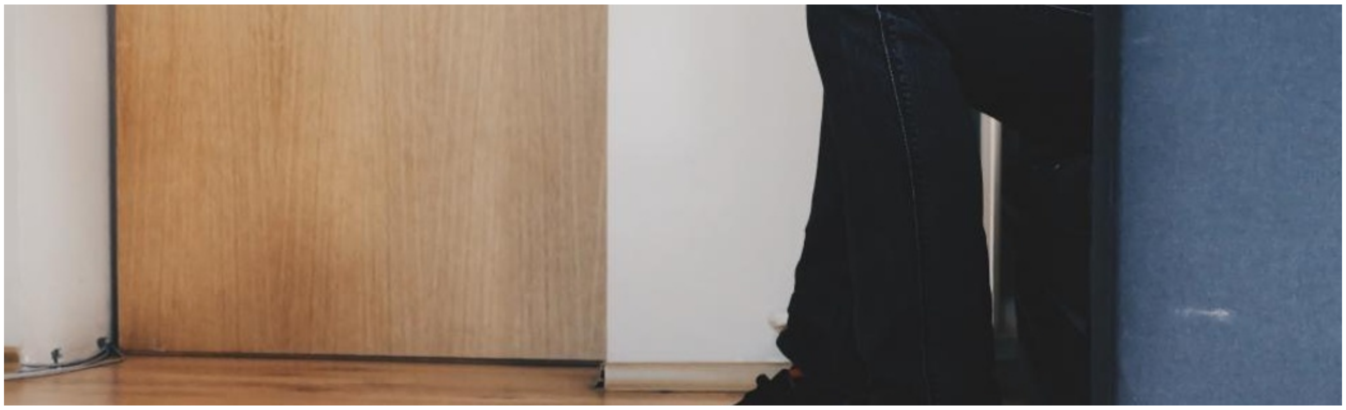
Nishant FOLLOW
Tech Author @Mozila | Software Consultant - Web & Mobile

21 Essential JavaScript Interview Questions

Published Sep 17, 2015



Question 1

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

```
> 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?

108 70

```
var y = 1;
  if (function f(){}) {
    y += typeof f;
  }
  console.log(y);
```

The output would be `1undefined`. The `if` condition statement evaluates using `eval`, so `eval(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

108 70

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 nul
    this.company = company || ""; //Public attribute default value is nul
    this.salary = salary || 5000; //Public attribute default value is nul

    // 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

♡ 108 70

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

♡ 108 70

```
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
```

♡ 108   70

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

```
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;
        };
    };
}
```

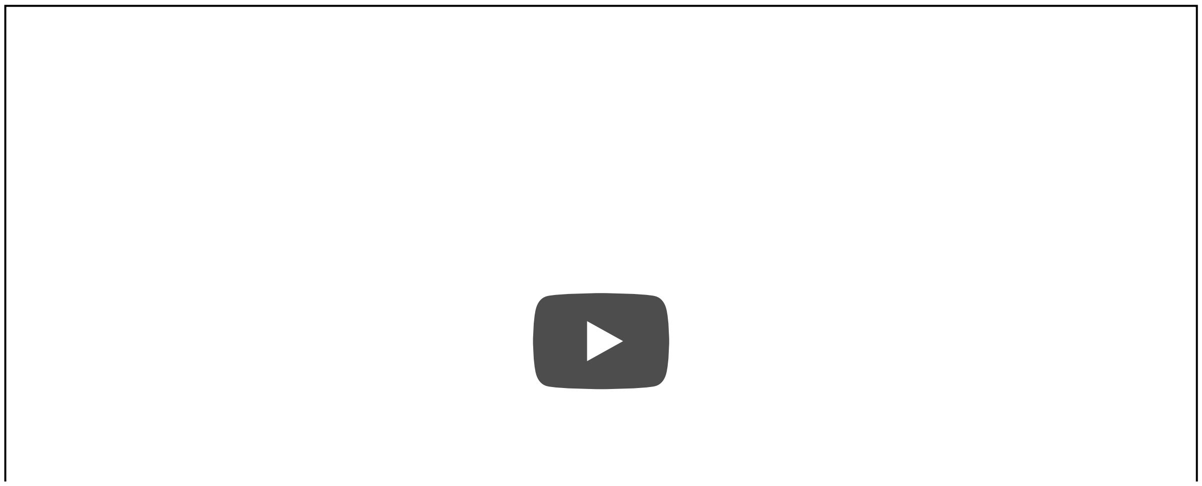
Here the `mul` function accepts the first argument and returns an anonymous function, which takes the second parameter and returns another anonymous function that will take the third parameter and return the multiplication of the arguments that have been passed.

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

108 70

- 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 to a new empty array. This is

108 70

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 var
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 var
arrayList.length = 0; // Empty the array by setting length to 0
console.log(anotherArrayList); // Output []
```

108 70

```
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 var
arrayList.splice(0, arrayList.length); // Empty the array by setting leng
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];
```

108 70

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 exec
        }
}
```

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 use

108 70

```
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?

108 70

```
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 `x` is 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?

108 70

```
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 is `undefined` .

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 `delete` operator doesn't delete prototype property.

`emp1` object doesn't have **company** as its own property. You can test it `console.log(emp1.hasOwnProperty('company')); //output : false` . However, we can delete the `company` property directly from the `Employee` object using

108 70

`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 check `trees[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 in `trees[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?

108 70

```
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 the `delete` 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** and `undefined` 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

♡ 108   70

- 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 `z` would 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:

♡ 108 70

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?

108 70

```
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>
```

108 70

```
<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:

```
var foo = undefined;
    foo(); // Here foo is undefined
           foo = function foo(){
              / Some code stuff
      }
```

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

♡ 108   70

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

108 70

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:

108 70

```
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

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`?

108 70

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:

~~~
We can also calculate the length of an object by iterating through an obj

```javascript
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` :

108 70

```
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.

Here's a live mock JavaScript interview you might find useful!

Interview JavaScript

Enjoy this post? Give **Nishant** a like if it's helpful.

108 70 SHARE

Nishant

Tech Author @Mozila | Software Consultant - Web & Mobile

6 + years of experience working as a software professional with substantial experience in

108 70

FOLLOW

70 Replies

Leave a reply

Nishank Dwivedi a month ago

In Question 15

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

console.log(y);

First 1 ==> z

Then, typeof y (i.e. undefined) ==> z ==> y #IF_YOU_CHECK

So, now both y and z ==> undefined

I think explanation is wrong.

Reply

Martin Carel a month ago

In question 2, a better explanation why `f` is `undefined` is the fact that we are dealing with a “named function expression”. And that name is local only to the function body (scope). Therefore, it will be `undefined` outside of the function body.

Show more

Reply

108 70

Martin - Did you also try the second snippet of question 2. I think the answer is wrong, because f is visible within the if’s scope.

Reply

Martin Carel a month ago

I also get the output of `1undefined` under Node v9.6.1, because `foo` is not visible except within `foo`'s function body, since it is interpreted as a “named function expression” again.

Reply

Dipak Raval a month ago

Clean code and clean to read.

Some point as too the point in short.

Reply

Show more replies

David Nguyen

next.js at Chotot

Overview

Everyday,Chợ tốt (Chotot) receives over 1 million of visits cross platforms, most of traffic comes from mobile devices. It is really important for us to develop products that can run across devices. Last year, we switched to a new stack to rebuild our products.

108 70

READ MORE

108

70