Table of Contents

1.

2. What is a closure?

A closure is the combination of a function and the lexical environment within which that function was declared.

```
function init() {
  var name = 'Mozilla'; // name is a local variable created by init
  function displayName() { // displayName() is the inner function, a closure
    alert(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

This will result in an alert with the name variable. This is an example of lexical scoping, which describes how a parser resolves variable names when functions are nested.

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

the displayName() inner function is returned from the outer function before being executed. Functions in JavaScript form closures. In some programming languages, the local variables within a function exist only for the duration of that function's execution. In javascript the closure is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created.

3. How to empty an array in JavaScript?

```
A = [1,2,3,4];

A = [];
```

```
A.length = 0

A.splice(0,A.length)- returns a copy of the A array

while(A.length > 0) {
   A.pop();
}
```

4. What is the difference between const, let and var declarations?

const means that the identifier can't be reassigned. a `const` object can have properties mutated.

let is a signal that the variable may be reassigned, such as a counter in a loop, or a value swap in an algorithm. It also signals that the variable will be used only in the block it's defined in, which is not always the entire containing function.

The let statement allows you to create a variable with the scope limited to the block on which it is used.

It is similar to the variable we declare in other languages like Java, .NET, etc.

```
function nodeSimplified(){
  let a =10;
  console.log(a);  // output 10
  if(true){
   let a=20;
   console.log(a); // output 20
  }
  console.log(a);  // output 10
}
```

var is now the weakest signal available when you define a variable in JavaScript. The variable may or may not be reassigned, and the variable may or may not be used for an entire function, or just for the purpose of a block or loop.

Variable declarations are processed before the execution of the code.

The scope of a JavaScript variable declared with var is its current execution context.

The scope of a JavaScript variable declared outside the function is global.

If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable. Global variables are not created automatically in "Strict Mode".

```
function nodeSimplified(){
  var a =10;
```

```
  console.log(a);  // output 10
  if(true){
   var a=20;
   console.log(a); // output 20
  }
  console.log(a);  // output 20
}

function nodeSimplified(){
  let a =10;
  let a =20; //throws syntax error
  console.log(a);
}

function nodeSimplified(){
  var a =10;
  var a =20;
  console.log(a);  //output 20
}

function nodeSimplified(){
  const MY_VARIABLE =10;
  console.log(MY_VARIABLE);  //output 10
  MY_VARIABLE =20;          //throws type error
  console.log(MY_VARIABLE);
}
```

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

The Array.isArray() method determines whether the passed value is an Array.

Or

```
arr.constructor === Array
```

6. What is the significance of, and reason for, wrapping the entire content of a JavaScript source file in a function block?

This technique creates a closure around the entire contents of the file which creates a private namespace and thereby helps avoid potential name clashes between different JavaScript modules and libraries.

```
/* Variables placed outside of self-executed function will be considered as global */

var globalVar = 'bar';

(function() {
  /* Variables placed inside of this function will be considered private */

  /* You won't see privateVar in global scope */
  var privateVar = 'foo';
})();

var myPlugin = (function() {
 var private_var;

 function private_function() {
 }

 return {
   public_function1: function() {
   },
   public_function2: function() {
   }
 }
})()
```

Now you can call myPlugin.public_function1(), but you cannot access private_function()!

7. What is the significance, and what are the benefits, of including 'use strict' at the beginning of a JavaScript source file?

The use strict directive is new in JavaScript 1.8.5 (ECMAScript 5). It is not a statement, but a literal expression, ignored by earlier versions of JavaScript. The short 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.

- Makes debugging easier.
- Prevents accidental globals. Without strict mode, assigning a value to an undeclared variable automatically creates a global variable with that name.
- Eliminates this coercion. Without strict mode, a reference to a this value of null or undefined is automatically coerced to the global.
- 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) {})
- 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

```
console.log(eval('2 + 2'));
// expected output: 4

// eval executes a string as code.
```

- 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.

- **configurable**

  true if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.

  Defaults to false.

```
'use strict';

const object1 = {};

Object.defineProperty(object1, 'property1', {
  value: 42,
  writable: true,
  configurable: false,
});

delete object1.property1;

console.log(object1.property1);
// will throw an error in strict mode because configurable is false!
```

- It is supported by all modern browsers.

8. What is NaN? What is its type? How can you reliably test if a value is equal to NaN?

The global NaN property is a value representing Not-A-Number. It is the returned value when Math functions fail (Math.sqrt(-1)) or when a function trying to parse a number fails (parseInt("blabla")).

Since NaN is the only JavaScript value that is treated as unequal to itself, you can always test if a value is NaN by checking it for equality to itself:

```
var a = NaN;
a !== a; // true
```

9. Discuss possible ways to write a function isInteger(x) that determines if x is an integer.

The Number.isInteger() method determines whether the passed value is an integer.

```
if(typeof data===='number' && (data%1)===0) {
   // data is an integer
}
```

10. What is the difference between the function declarations below?

```
var foo = function(){
   // Some code
};
 function bar(){
   // Some code
};
```

Function declarations load before any code is executed.

Function expressions load only when the interpreter reaches that line of code.

So if you try to call a function expression before it's loaded, you'll get an error! If you call a function declaration instead, it'll always work, because no code can be called until all declarations are loaded.

11.What is the difference between a method and a function in javascript?

In javascript every function is an object. An object is a collection of key:value pairs. If a value is a primitive (integer, string, boolean), or another object, the value is considered a property. If a value is a function, it is called a 'method'.

Within the scope of an object, a function is referred to as a method of that object. It is invoked from the object namespace 'MyObj.theMethod()'. Since we said a function is an object, a function within a function is considered a method of that function. You can say I am going to use the save method of my object.

12. Difference between undefined and not defined in JavaScript

Variables that are actually 'not defined', i.e. they don't exist as a given name isn't bound in the current lexical environment. Accessing such a variable will throw an error, but using typeof won't and will return 'undefined'. In contrast, accessing non-existing properties will not throw an error and return undefined instead (and you may use the in operator or the **hasOwnProperty()** method to check if properties actually do exist).

ReferenceError: "x" is not defined- error when a variable doesn't exist.

Existing variables which have not been assigned a value (which is common because of var hoisting) or which have been explicitly set to undefined. Accessing such a variable will return undefined, typeof will return 'undefined'.

Existing variables which have been explicitly set to null. Accessing such a variable will return null, typeof will return 'object'. Note that this is misleading: null is not an object, but a primitive value of type Null (which has the consequence that you can't return null from constructor functions - you

have to throw an error instead to denote failure).

13. Error types in javascript

The Error constructor creates an error object. Instances of Error objects are thrown when **runtime errors** occur. The Error object can also be used as a base object for user-defined exceptions.

EvalError

Creates an instance representing an error that occurs regarding the global function eval().

InternalError

Creates an instance representing an error that occurs when an internal error in the JavaScript engine is thrown. E.g. "too much recursion".

RangeError

Creates an instance representing an error that occurs when a numeric variable or parameter is outside of its valid range.

ReferenceError

Creates an instance representing an error that occurs when de-referencing an invalid reference.

SyntaxError

Creates an instance representing a syntax error that occurs while parsing code in eval().

TypeError

Creates an instance representing an error that occurs when a variable or parameter is not of a valid type.

URIError

Creates an instance representing an error that occurs when encodeURI() or decodeURI() are passed invalid parameters.

How to throw an error:

```
function getRectArea(width, height) {
  if (isNaN(width) || isNaN(height)) {
    throw "Parameter is not a number!";
  }
}

try {
  getRectArea(3, 'A');
}
catch(e) {
  console.log(e);
  // expected output: "Parameter is not a number!"
}
```

14. What is the drawback of creating true private in JavaScript?

One of the drawback of creating true private method in javascript is that they are very memory

inefficient because a new copy of the method would be created for each instance.

```
var Employee = function (name, company, salary) {

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

   // Private method
   var increaseSlary = 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);
```

Here each instance variable emp1, emp2, emp3 has own copy of increaseSalary private method.

15. What is undefined x 1 in JavaScript

This is how Chrome used to display empty arrays if you'd print them.

```
var arr = new Array(2);
console.log(arr);
```

now it does like:

```
(2) [empty × 2]
```

in vscode:

```
[ <2 empty items> ]
```

16. What is function hoisting in JavaScript?

Variable hoisting

Variables and functions are "hoisted." Rather than being available after their declaration, they might actually be available beforehand.

It only works with "var" declaration.

```
// Outputs: undefined
console.log(declaredLater);

var declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```

this is equivalent to this:

```
var declaredLater;

// Outputs: undefined
console.log(declaredLater);

declaredLater = "Now it's defined!";
```

```
// Outputs: "Now it's defined!"
console.log(declaredLater);
```

in this case you could expect that name retains its value, but due to hoisting it doesn't:

```
var name = "Baggins";

(function () {
    // Outputs: "Original name was undefined"
    console.log("Original name was " + name);

    var name = "Underhill";

    // Outputs: "New name is Underhill"
    console.log("New name is " + name);
})();
```

Function hoisting

A function declaration doesn't just hoist the function's name. It also hoists the actual function definition.

```
// Outputs: "Yes!"
isItHoisted();

function isItHoisted() {
    console.log("Yes!");
}
```

function definition hoisting **only occurs for function declarations, not function expressions**. It doesn't work neither if it's unnamed nor a named function.

17. What is the instanceof operator in JavaScript?

The instanceof operator tests whether the **prototype property of a constructor appears anywhere in the prototype chain of an object**.

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

```
//Car: prototype property of the constructor
var auto = new Car('Honda', 'Accord', 1998);

console.log(auto instanceof Car);
// expected output: true

console.log(auto instanceof Object);
// expected output: true
```

18. Difference between Function, Method and Constructor calls in JavaScript.

In JavaScript, these are just three different usage patterns of one single construct.

functions : The simplest usages of function call:

```
function helloWorld(name) {
return "hello world, " + name;
}
```

Methods in JavaScript are nothing more than object properties that reference to a function.

```
var obj = {
helloWorld : function() {
return "hello world, " + this.name;
},
name: 'John Carter'
}
obj.helloWorld();  // "hello world John Carter"
```

we can copy a reference to the same function helloWorld in another object and get a difference answer.

```
var obj2 = {
helloWorld : obj.helloWorld,
name: 'John Doe'
}
```

The third use of functions is as constructors. Like function and method, constructors are defined with function.

```
function Employee(name, age) {
this.name = name;
this.age = age;
}

var emp1 = new Employee('John Doe', 28);
emp1.name; // "John Doe"
emp1.age; // 28
```

a **constructor call new Employee('John Doe', 28) creates a brand new object and passes it as the value of this**, and implicitly returns the new object as its result.

19. What is JavaScript Self-Invoking anonymous function or Self-Executing anonymous function

the part before the () is the anonymous function, and the () makes it self executing. Those two brackets cause everything contained in the preceding parentheses to be executed immediately. What's useful here is that JavaScript has function level scoping. All variables and functions defined within the anonymous function aren't available to the code outside of it, effectively using closure to seal itself from the outside world.

```
(function(){
  console.log('Hello World!');
})();
```

an use example:

```
(function(window){
  var foo = 'Hello';
  var bar = 'World!'

  function baz(){
     return foo + ' ' + bar;
}

  //In this context, 'window' refers to the parameter
  window.baz = baz;
})(window); //Pass in a reference to the global window object
```

20. Describe Singleton Pattern In JavaScript?

a design pattern that restricts the instantiation of a class to one object. A singleton should be immutable by the consuming code, and there should be no danger of instantiating more than one of them. Old implementation, when this code is interpreted, UserStore will be set to the result of that immediately invoked function — an object that exposes two functions, but that does not grant direct access to the collection of data.

```
var UserStore = (function(){
  var _data = [];

  function add(item){
    _data.push(item);
  }

  function get(id){
    return _data.find((d) => {
      return d.id === id;
    });
  }
  return {
    add: add,
    get: get
  };
}());
```

This doesn't give us the immutability we desire when making use of singletons. Code executed later could modify either one of the exposed functions, or even redefine UserStore altogether.

Newer/better implementation (its methods cannot be changed, nor can new methods or properties be added to it. Furthermore, because we're taking advantage of ES6 modules, we know exactly where UserStore is used).

Here's how the implementation would look if we wanted to utilize ES6 classes. By adding the extra step of holding a reference to the instance, we can check whether or not we've already instantiated a UserStore, and if we have, we won't create a new one. As you can see, this also makes good use of the fact that we've made UserStore a class.

```
class UserStore {
  constructor(){
   if(! UserStore.instance){
     this._data = [];
     UserStore.instance = this;
```

```
  }

  return UserStore.instance;
 }

 add(item){
   this._data.push(item);
 }

 get(id){
   return this._data.find(d => d.id === id);
 }

}

const instance = new UserStore();
Object.freeze(instance);

export default instance;
```

The Object.freeze() method freezes an object: that is, prevents new properties from being added to it; prevents existing properties from being removed; and prevents existing properties, or their enumerability, configurability, or writability, from being changed, it also prevents the prototype from being changed.

21. What are the ways by which we can create object in JavaScript ?

Creates empty object:

```
var d = new Object();
```

This method creates a new object extending the prototype object passed as a parameter.

```
var a = Object.create(null);
```

This is equivalent to Object.create(null) method, using a null prototype as an argument.

```
var b = {};
```

What the new operator does is call a function and setting this of the function to a fresh new Object, and binding the prototype of that new Object to the function's prototype.

```
var Obj = function(name) {
  this.name = name
}
var c = new Obj("hello");
```

Using the function constructor + prototype:

```
function myObj(){};
myObj.prototype.name = "hello";
var k = new myObj();
```

Using ES6 class syntax:

```
class myObject  {
  constructor(name) {
    this.name = name;
  }
}
var e = new myObject("hello");
```

22. Write a function called deepClone which takes an object and creates a object copy of it.

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated.

Half-solutions:

shallow copy

In ECMAScript 6 there is Object.assign method, which copies values of all enumerable own properties from one object to another.

```
var x = {myProp: "value"};
var y = Object.assign({}, x);
```

deep copy

```
JSON.parse(JSON.stringify(a))
```

Full solution:

```
function cloneSO(obj) {
    // Handle the 3 simple types, and null or undefined
    if (null == obj || "object" != typeof obj) {
            return obj;
        }

    // Handle Date
    else if (obj instanceof Date) {
        var copy = new Date();
        copy.setTime(obj.getTime());
        return copy;
    }


    // Handle Array
    else if (obj instanceof Array) {
        var copy = [];
        for (var i = 0, len = obj.length; i < len; i++) {
            copy[i] = cloneSO(obj[i]);
        }
        return copy;
    }

    // Handle Object
    else if (obj instanceof Object) {
        var copy = {};
        for (var attr in obj) {
            if (obj.hasOwnProperty(attr)) copy[attr] = cloneSO(obj[attr]);
        }
```

```
        return copy;
    } else {
        throw new Error("Unable to copy obj! Its type isn't supported.");
    }
}
```

23. Best way to detect undefined object property in JavaScript.

```
if (typeof my_obj.someproperties === "undefined"){
    console.log('the property is not available...'); // print into console
}
```

24. How to check whether a key exists in a JavaScript object or not

test the particular object instance for a property:

```
obj.hasOwnProperty("key")

// note the three equal signs so that null won't be equal to undefined
if( aa["goodbye"] === undefined ) {
    // do something
}
```

25. What is Passed By Reference

Most things in JavaScript are Objects. The only elements that are not objects are the Primitive Data Types : string, number, boolean, null and undefined. These Primitive Data Types also are immutable, which means that once created they cannot be modified.

Primitive Data Types are passed By Value and Objects are passed By Reference.

By Value means creating a COPY of the original.

```
var name = "Carlos";
var firstName = name;
name = "Carla";
console.log(name); // "Carla"
console.log(firstName); // "Carlos"
```

A variable name is created and given the value of "Carlos". JavaScript allocates a memory spot for it.

A variable firstName is created and is given a copy of name's value. firstName has its own memory spot and is independent of name . At this moment in the code, firstName also has a value of "Carlos".

We then change the value of name to "Carla". But firstName still holds its original value, because it lives in a different memory spot.

When working with primitives, the =operator creates a copy of the original variable. That's what by value means.

By Reference means creating an ALIAS to the original.

```
var myName = {
  firstName: "Carlos"
};
var identity = myName;
myName.firstName = "Carla";
console.log(myName.firstName); // "Carla"
console.log(identity.firstName); // "Carla"
```

A variable myName is created and is given the value of an object which has a property called firstName. firstName has the value of "Carlos". JavaScript allocates a memory spot for myName and the object it contains.

A variable identity is created and is pointed to myName. There is no dedicated memory space to identity's value'. It only points to myName's value.

We change the value of myName's firstName property to "Carla" instead of "Carlos".

With functions:

with primitives:

```
var myName = "Carlos";
function myNameIs(aName){
  aName = "Carla";
}
myNameIs(myName);
console.log(myName); // "Carlos"
```

with objects:

```
var myName = {};
function myNameIs(aName){
  aName.firstName = "Carla";
```

```
}
myNameIs(myName);
console.log(myName); // Object {firstName: "Carla"}
```

reassigning an object doesn't work:

```
var myName = {
  firstName: "Carla"
};
function myNameIs(aName){
  aName = {
    nickName: "Carlita"
  };
}
myNameIs(myName);
console.log(myName); // Object {firstName: "Carla"}
```

but changing properties does work:

```
var myName = {
  firstName: "Carla"
};
function myNameIs(aName){
  aName.nickName = "Carlita";
}
myNameIs(myName);
console.log(myName); // Object {firstName: "Carla", nickName: "Carlita"}
```

26. Describe Object-Based inheritance in JavaScript.

A prototype-based language, such as JavaScript simply has objects. A prototype-based language has the notion of a prototypical object, an object used as a template from which to get the initial properties for a new object. Any object can specify its own properties, either when you create it or at run time. In addition, any object can be associated as the prototype for another object, allowing the second object to share the first object's properties.

In class-based languages, you define a class in a separate class definition. In that definition you can specify special methods, called constructors, to create instances of the class. A constructor method can specify initial values for the instance's properties and perform other processing appropriate at creation time. You use the new operator in association with the constructor method to create class instances.

JavaScript follows a similar model, but does not have a class definition separate from the constructor. Instead, you define a constructor function to create objects with a particular initial set

of properties and values. Any JavaScript function can be used as a constructor. You use the new operator with a constructor function to create a new object.

In class-based languages, you typically create a class at compile time and then you instantiate instances of the class either at compile time or at run time. You cannot change the number or the type of properties of a class after you define the class. In JavaScript, however, at run time you can add or remove properties of any object. If you add a property to an object that is used as the prototype for a set of objects, the objects for which it is the prototype also get the new property.

Class-based (Java)	Prototype-based (JavaScript)
Class and instance are distinct entities.	All objects can inherit from another object.
Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Create a single object with the new operator.	Same.
Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Class definition specifies *all* properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an *initial set* of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

```
function Employee() {
  this.name = '';
  this.dept = 'general';
}
```

similar to whatever extends shit terminology:

```
function Manager() {
  Employee.call(this);
  this.reports = [];
}
Manager.prototype =
    Object.create(Employee.prototype);

function WorkerBee() {
  Employee.call(this);
```

```
  this.projects = [];
}
WorkerBee.prototype =
   Object.create(Employee.prototype);
```

27. How we can prevent modification of an object in JavaScript ?

Difference between the Object.freeze(), Object.seal(), Objet.preventExtensions() methods in terms of modifying existing properties.

	CREATE	READ	UPDATE	DELETE
Object.freeze	✗	✓	✗	✗
Object.seal	✗	✓	✓	✗
Object.preventExtensions	✗	✓	✓	✓

28. Write a log function which will add prefix (your message) to every message you log using console.log

```
function log(message){
  const prefix = 'whatever: ';
  console.log(prefix, message);
}
```

29. typical use case for anonymous function in JavaScript ?

Anonymous Functions are function expressions rather than the regular function declaration which are statements. Function expressions are more flexible. We can assign functions to variables, object properties, pass them as arguments to other functions, and even write a simple one line code enclosed in an anonymous functions.

```
var squaredArray = inputArray.map(x => x * x);
```

typical: self executing anonymous function

```
(function() { })();
```

typical use for anonymous functions in JS is to pass them as arguments to other functions

```
setTimeout(function() {
  alert('hello');
}, 1000);
```

30. How to set a default parameter value ?

```
function multiply(a, b = 1) {
  return a * b;
}
```

31. Write code for merge two JavaScript Object dynamically.

```
/** There's no limit to the number of objects you can merge.
 *  All objects get merged into the first object.
 *  Only the object in the first argument is mutated and returned.
 *  Later properties overwrite earlier properties with the same name. */
const allRules = Object.assign({}, obj1, obj2, obj3, etc);
```

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

```
const object1 = {
  a: 1,
  b: 2,
  c: 3
};

const object2 = Object.assign({c: 4, d: 5}, object1);

console.log(object2.c, object2.d);
// expected output: 3 5
```

Cloning an object

```
var obj = { a: 1 };
var copy = Object.assign({}, obj);
```

```
console.log(copy); // { a: 1 }
```

32. What is non-enumerable property in JavaScript and how can you create one?

```
var person = { age: 18 };
Object.defineProperty(person, 'name', { value: 'Joshua', enumerable: false });

person.name; // 'Joshua'
Object.keys(person); // ['age']
```

name property didn't show up because it wasn't enumerable. Consequently, doing JSON.stringify(person) would not include the name property either.

33. What is Function binding ?

Example: When using setTimeout with object methods or passing object methods along, there's a known problem: this stops working right.

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

Same as:

```
let f = user.sayHi;
setTimeout(f, 1000); // lost user context

var module = {
  x: 42,
  getX: function() {
    return this.x;
  }
}

var retrieveX = module.getX;
console.log(retrieveX()); // The function gets invoked at the global scope
// expected output: undefined
```

```
var boundGetX = retrieveX.bind(module);
console.log(boundGetX());
// expected output: 42
```

function binding:

```
let boundFunc = func.bind(context);

let user = {
  firstName: "John"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John
```

or:

```
let user = {
  firstName: "John",
  sayHi() {
    alert(`Hello, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)

sayHi(); // Hello, John!

setTimeout(sayHi, 1000); // Hello, John!
```

34.

35. 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);
})();
```

1

4

3

2

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.

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).

36. Write a simple function (less than 160 characters) that returns a boolean indicating whether or not a string is a palindrome.

```
function isPalindrome(str) {
  str = str.replace(/\W/g, '').toLowerCase();
  return (str == str.split('').reverse().join(''));
}
```

37. Write a sum method which will work properly when invoked using either syntax below.

```
function sum(x) {
  if (arguments.length == 2) {
    return arguments[0] + arguments[1];
  } else {
    return function(y) { return x + y; };
  }
```

```
}
```

38. 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.

39. 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.

```
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
  }
}
```

40. How do you add an element at the begining of an array? How do you add one at the end?

```
var myArray = ['a', 'b', 'c', 'd'];
myArray.push('end');
myArray.unshift('start');
console.log(myArray); // ["start", "a", "b", "c", "d", "end"]
```

With ES6, one can use the spread operator:

```
myArray = ['start', ...myArray];
myArray = [...myArray, 'end'];
```

Or, in short:

```
myArray = ['start', ...myArray, 'end'];
```

41. What happens when you reference out of bounds elements of an array?

```
var a = [1, 2, 3];
```

a) Will this result in a crash?

```
A[10] = 99;
```

b) What will this output?

```
console.log(a[6])
```

a) It will not crash. The JavaScript engine will make array slots 3 through 9 be “empty slots.”

b) Here, a[6] will output undefined, but the slot still remains empty rather than filled with undefined. This may be an important nuance in some cases. For example, when using map(), empty slots will remain empty in map()’s output, but undefined slots will be remapped using the function passed to it:

```
var b = [undefined];
b[2] = 1;
console.log(b);              // (3) [undefined, empty × 1, 1]
console.log(b.map(e => 7)); // (3) [7,         empty × 1, 7]
```

42. What is the value of typeof undefined == typeof NULL?

The expression will be evaluated to true, since NULL will be treated as any other undefined variable.

Note: JavaScript is case-sensitive and here we are using NULL instead of null.

43. What gets printed out:

1.

A will be undefined, b not

```
(function(){
   var a = b = 3;
})();
```

```
console.log(typeof a);//"undefined"
console.log(b);//3
```

var a = b = 3; is actually shorthand for:

```
b = 3;
var a = b;
```

Therefore, b ends up being a global variable (since it is not preceded by the var keyword) and is still in scope even outside of the enclosing function. The reason a is undefined is that a is a local variable to that self-executing anonymous function.

2.

```
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();
```

printed out:

```
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.

3.

```
console.log(0.1 + 0.2);
console.log(0.1 + 0.2 == 0.3);
```

prints:

```
0.30000000000000004
false
```

Numbers in JavaScript are all treated with floating point precision, and as such, may not always yield the expected results.

4.

```
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);
}
```

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.

It works properly if you use let instead of var.

5.

will these return the same thing?

```
function foo1()
{
 return {
    bar: "hello"
 };
}

function foo2()
{
 return
 {
    bar: "hello"
 };
}
```

foo1 returns:

```
Object {bar: "hello"}
```

foo2 returns:

```
undefined
```

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").

6.

```
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));
```

"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 (i.e. ['n','h','o','j', ['j','o','n','e','s']]) 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.

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).

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.

7.

```
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);
```

"122"

"32"

"02"

"112"

"NaN2"

NaN

8.

```
for (var i = 0; i < 5; i++) {
        setTimeout(function() { console.log(i); }, i * 1000 );
}
```

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 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);
}
```

Or I can just use let instead of var.

9.

```
console.log("0 || 1 = "+(0 || 1));
console.log("1 || 2 = "+(1 || 2));
console.log("0 && 1 = "+(0 && 1));
console.log("1 && 2 = "+(1 && 2));
```

```
0 || 1 = 1
1 || 2 = 1
0 && 1 = 0
1 && 2 = 2
```

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).

10.

```
console.log(false == '0')
console.log(false === '0')
```

```
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 !=.

11.

```
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].

12.

```
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).

The named function f() calls itself recursively, until it gets down to calling f(1)

13.

```
(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, 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.

14.

```
var length = 10;
function fn() {
        console.log(this.length);
}

var obj = {
  length: 5,
```

```
  method: function(fn) {
    fn();
    arguments[0]();
  }
};

obj.method(fn, 1);
```

```
10
2
```

In the first place, as fn is passed as a parameter to the function method, the scope (this) of the function fn is window. var length = 10; is declared at the window level. It also can be accessed as window.length or length or this.length (when this === window.)

method is bound to Object obj, and obj.method is called with parameters fn and 1. Though method is accepting only one parameter, while invoking it has passed two parameters; the first is a function callback and other is just a number.

When fn() is called inside method, which was passed the function as a parameter at the global level, this.length will have access to var length = 10 (declared globally) not length = 5 as defined in Object obj.

Now, we know that we can access any number of arguments in a JavaScript function using the arguments[] array.

Hence arguments[0]() is nothing but calling fn(). Inside fn now, the scope of this function becomes the arguments array, and logging the length of arguments[] will return 2.

14.

```
(function () {
    try {
        throw new Error();
    } catch (x) {
        var x = 1, y = 2;
        console.log(x);
    }
    console.log(x);
    console.log(y);
})();
```

```
1
```

```
undefined
2
```

Var statements are hoisted (without their value initialization) to the top of the global or function scope it belongs to, even when it's inside a with or catch block. However, the error's identifier is only visible inside the catch block. It is equivalent to:

```
(function () {
    var x, y; // outer and hoisted
    try {
        throw new Error();
    } catch (x /* inner */) {
        x = 1; // inner x, not the outer one
        y = 2; // there is only one y, which is in the outer scope
        console.log(x /* inner */);
    }
    console.log(x);
    console.log(y);
})();
```

15.

```
var x = 21;
var girl = function () {
    console.log(x);
    var x = 20;
};
girl ();
```

Neither 21, nor 20, the result is undefined

It's because JavaScript initialization is not hoisted.

(Why doesn't it show the global value of 21? The reason is that when the function is executed, it checks that there's a local x variable present but doesn't yet declare it, so it won't look for global one.)

16.

```
console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
```

The first statement returns true which is as expected.

The second returns false because of how the engine works regarding operator associativity for < and >. It compares left to right, so 3 > 2 > 1 JavaScript translates to true > 1. true has value 1, so it then compares 1 > 1, which is false.

17.

```
console.log(typeof typeof 1);
```

string

typeof 1 will return "number" and typeof "number" will return string.

18.

```
var b = 1;
function outer(){
        var b = 2
   function inner(){
      b++;
      var b = 3;
      console.log(b)
   }
   inner();
}
outer();
```

Output to the console will be “3”.

There are three closures in the example, each with it’s own var b declaration. When a variable is

invoked closures will be checked in order from local to global until an instance is found. Since the inner closure has a b variable of its own, that is what will be output.

Furthermore, due to hoisting the code in inner will be interpreted as follows:

```
function inner () {
    var b; // b is undefined
    b++; // b is NaN
    b = 3; // b is 3
    console.log(b); // output "3"
}
```