Javascript

The prototype object is special type of enumerable object to which additional properties can be attached to it which will be shared across all the instances of it's constructor function.

eCMA International publishes the specification for scripting languages is called 'ECMAScript'. ECMAScript specification defined in ECMA-262 for creating a general-purpose scripting language.

Use the online editor to quickly execute the JavaScript code without any installation. The followings are free online editors:

jsfiddle.net

jsbin.com

playcode.io

Case Sensitive

JavaScript is a case-sensitive scripting language. So, name of functions, variables and keywords are case sensitive. For example, myfunction and MyFunction are different, Name is not equal to nAme, etc.

Variables

In JavaScript, a variable is declared with or without the var keyword

JavaScript ignores multiple spaces and tabs

Write comment after double slashes // or write multiple lines of comments between /\* and \*/

JavaScript Message Boxes: alert(), confirm(), prompt()

Sometimes you may need to take the user's input to do further actions.

The alert(), confirm(), and prompt() functions are global functions. So, they can be called using the window object e.g. window.alert(), window.confirm(), and window.prompt().

JavaScript is a loosely typed language. It means it does not require a data type to be declared. You can assign any literal values to a variable, e.g., string, integer, float, boolean, etc.

The variables declared without the var keyword becomes global variables, irrespective of where they are declared. Visit Variable Scope in JavaScript to learn about it.

It is Not Recommended to declare a variable without var keyword because it can accidentally overwrite an existing global variable.

Points to Remember

Variable stores data value that can be changed later on.

Variables can be defined using var keyword. Variables defined without var keyword become global variables.

Variables must be initialized before accessing it.

JavaScript allows multiple white spaces and line breaks in a variable declaration.

Multiple variables can be defined in a single line separated by a comma.

JavaScript is a loosely-typed language, so a variable can store any type value.

Variable names are case-sensitive.

Variable names can contain letters, digits, or the symbols $ and \_. It cannot start with a digit 0 - 9.

Variables can have local or global scope. Local variables cannot be accessed out of the function where they are declared, whereas the global variables can be accessed from anywhere.

Points to Remember :

JavaScript object is a standalone entity that holds multiple values in terms of properties and methods.

Object property stores a literal value and method represents function.

An object can be created using object literal or object constructor syntax.

Object literal:

var person = {

firstName: "James",

lastName: "Bond",

age: 25,

getFullName: function () {

return this.firstName + ' ' + this.lastName

}

};

Object constructor:

var person = new Object();

person.firstName = "James";

person["lastName"] = "Bond";

person.age = 25;

person.getFullName = function () {

return this.firstName + ' ' + this.lastName;

};

Object properties and methods can be accessed using dot notation or [ ] bracket.

An object is passed by reference from one function to another.

An object can include another object as a property.

The ++ and -- operators are unary operators.

=== Compares equality of two operands with type.

JavaScript provides a special operator called ternary operator :? that assigns a value to a variable based on some condition

Points to Remember:

JavaScript includes operators that perform some operation on single or multiple operands (data value) and produce a result.

JavaScript includes various categories of operators: Arithmetic operators, Comparison operators, Logical operators, Assignment operators, Conditional operators.

Ternary operator ?: is a short form of if-else condition.

JavaScript is dynamic and loosely typed language. It means you don't require to specify a type of a variable. A variable in JavaScript can be assigned any type of value, as shown in the following example.

var myvariable = 1; // numeric value

myvariable = 'one'; // string value

myvariable = 1.1; // decimal value

myvariable = true; // Boolean value

myvariable = null; // null valuePrimitive Data Types

The primitive data types are the lowest level of the data value in JavaScript. The typeof operator can be used with primitive data types to know the type of a value.

Data Type Description

String String is a textual content wrapped inside ' ' or " " or ` ` (tick sign).

Example: 'Hello World!', "This is a string", etc.

Number Number is a numeric value.

Example: 100, 4521983, etc.

BigInt BigInt is a numeric value in the arbitrary precision format.

Example: 453889879865131n, 200n, etc.

Boolean Boolean is a logical data type that has only two values, true or false.

Null A null value denotes an absense of value.

Example: var str = null;

Undefined undefined is the default value of a variable that has not been assigned any value.

Example: In the variable declaration, var str;, there is no value assigned to str. So, the type of str can be check using typeof(str) which will return undefined.

Structural Data Types

The structural data types contain some kind of structure with primitive data.

Data Type Description

Object An object holds multiple values in terms of properties and methods.

Example:

var person = {

firstName: "James",

lastName: "Bond",

age: 15

};

Date Date object represents date & time including days, months, years, hours, minutes, seconds and milliseconds.

Example: var today = new Date("25 July 2021");

Array An array stores multiple values using special syntax.

Example: var nums = [1, 2, 3, 4];

Learn about each data type in detail in the next section.

If you want to include same quotes in a string value as surrounding quotes then use backward slash (\) before quotation mark inside string value.

Points to Remember:

JavaScript string must be enclosed in double or single quotes (" " or ' ').

String can be assigned to a variable using = operator.

Multiple strings can be concatenated using + operator.

A string can be treated as character array.

Use back slash (\) to include quotation marks inside string.

String objects can be created using new keyword. e.g. var str = new String();

String methods are used to perform different task on strings.

Points to Remember :

An array is a special type of variable that stores multiple values using a special syntax.

An array can be created using array literal or Array constructor syntax.

Array literal syntax: var stringArray = ["one", "two", "three"];

Array constructor syntax:var numericArray = new Array(3);

A single array can store values of different data types.

An array elements (values) can be accessed using zero based index (key). e.g. array[0].

An array index must be numeric.

Array includes length property and various methods to operate on array objects.The following table lists all the Array methods.

Method Description

concat() Returns new array by combining values of an array that is specified as parameter with existing array values.

every() Returns true or false if every element in the specified array satisfies a condition specified in the callback function. Returns false even if single element does not satisfy the condition.

filter() Returns a new array with all the elements that satisfy a condition specified in the callback function.

forEach() Executes a callback function for each elements of an array.

indexOf() Returns the index of the first occurrence of the specified element in the array, or -1 if it is not found.

join() Returns string of all the elements separated by the specified separator

lastIndexOf() Returns the index of the last occurrence of the specified element in the array, or -1 if it is not found.

map() Creates a new array with the results of calling a provided function on every element in this array.

pop() Removes the last element from an array and returns that element.

push() Adds one or more elements at the end of an array and returns the new length of the array.

check

reduce() Pass two elements simultaneously in the callback function (till it reaches the last element) and returns a single value.

reduceRight() Pass two elements simultaneously in the callback function from right-to-left (till it reaches the last element) and returns a single value.

reverse() Reverses the elements of an array. Element at last index will be first and element at 0 index will be last.

shift() Removes the first element from an array and returns that element.

slice() Returns a new array with specified start to end elements.

some() Returns true if at least one element in this array satisfies the condition in the callback function.

sort() Sorts the elements of an array.

splice() Adds and/or removes elements from an array.

toString() Returns a string representing the array and its elements.

unshift() Adds one or more elements to the front of an array and returns the new length of the array.

Undefined is also a primitive value in JavaScript. A variable or an object has an undefined value when no value is assigned before using it. So you can say that undefined means lack of value or unknown value.

Points to Remember :

null and undefined are primitive values in JavaScript.

A null value means absence.

An undefined value means lack of value.

A null or undefined value evalutes to false in conditional expression.

Anonymous function is useful in passing callback function, creating closure or Immediately invoked function expression.

var showMessage = function (){

alert("Hello World!");

};

Points to Remember :

JavaScript a function allows you to define a block of code, give it a name and then execute it as many times as you want.

A function can be defined using function keyword and can be executed using () operator.

A function can include one or more parameters. It is optional to specify function parameter values while executing it.

JavaScript is a loosely-typed language. A function parameter can hold value of any data type.

You can specify less or more arguments while calling function.

All the functions can access arguments object by default instead of parameter names.

A function can return a literal value or another function.

A function can be assigned to a variable with different name.

JavaScript allows you to create anonymous functions that must be assigned to a variable.

points to Remember :

Use if-else conditional statements to control the program flow.

JavaScript includes three forms of if condition: if condition, if else condition and else if condition.

The if condition must have conditional expression in brackets () followed by single statement or code block wrapped with { }.

'else if' statement must be placed after if condition. It can be used multiple times.

'else' condition must be placed only once at the end. It must come after if or else if statement.

Points to Remember :

The switch is a conditional statement like if statement.

A switch statement includes literal value or is expression based

A switch statement includes multiple cases that include code blocks to execute.

A break keyword is used to stop the execution of case block.

A switch case can be combined to execute same code block for multiple cases.

for (; ;) {

if (i >= 5)

break;

console.log(arr[i]);

i++;

}

Points to Remember :

JavaScript while loop & do-while loop execute the code block repeatedly till conditional expression returns true.

do-while loop executes the code at least once even if condition returns false.

Points to Remember :

JavaScript has global scope and local scope.

Variables declared and initialized outside any function become global variables.

Variables declared and initialized inside function becomes local variables to that function.

Variables declared without var keyword inside any function becomes global variables automatically.

Global variables can be accessed and modified anywhere in the program.

Local variables cannot be accessed outside the function declaration.

Global variable and local variable can have same name without affecting each other.

JavaScript does not allow block level scope inside { } brackets

eval() is a global function in JavaScript that evaluates a specified string as JavaScript code and executes it.

var str = '({"firstName":"Bill","lastName":"Gates"})';

var obj = eval(str);

obj.firstName; // Bill

It is not recommended to use eval() because it is slow, not secure, and makes code unreadable and maintainable.

JavaScript is a loosely-typed language. It does not give compile-time errors. So some times you will get a runtime error for accessing an undefined variable or calling undefined function etc.

try

{

// code that may throw an error

}

catch(ex)

{

// code to be executed if an error occurs

}

finally{

// code to be executed regardless of an error occurs or not

}

throw

Use throw keyword to raise a custom error.

try

{

throw "Error occurred";

}

catch(ex)

{

alert(ex);

}

"use strict";

var x = 1; // valid in strict mode

y = 1; // invalid in strict mode

The strict mode in JavaScript does not allow following things:

Use of undefined variables

Use of reserved keywords as variable or function name

Duplicate properties of an object

Duplicate parameters of function

Assign values to read-only properties

Modifying arguments object

Octal numeric literals

with statement

eval function to create a variable

Strict mode can be applied to function level in order to implement strictness only in that particular function.

x = 1; //validfunction sum(val1, val2){

"use strict";

result = val1 + val2; //error

return result;

}

JavaScript Hoisting

Hoisting is a concept in JavaScript, not a feature. In other scripting or server side languages, variables or functions must be declared before using it.

In JavaScript, variable and function names can be used before declaring it. The JavaScript compiler moves all the declarations of variables and functions at the top so that there will not be any error. This is called hoisting.

x = 1;

alert('x = ' + x); // display x = 1

var x;

Example: Hoisting not applicable for initialized variables

alert('x = ' + x); // display x = undefined

var x = 1;

Hoisting of Function

JavaScript compiler moves the function definition at the top in the same way as variable declaration.

Example: Function Hoisting

alert(Sum(5, 5)); // 10

function Sum(val1, val2)

{

return val1 + val2;

}

Add(5, 5); // error

var Add = function Sum(val1, val2)

{

return val1 + val2;

}

Points to Remember :

JavaScript compiler moves variables and function declaration to the top and this is called hoisting.

Only variable declarations move to the top, not the initialization.

Functions definition moves first before variables.

JavaScript ECMAScript 5, does not have class type. So it does not support full object oriented programming concept as other languages like Java or C#. However, you can create a function in such a way so that it will act as a class.

the following example demonstrates how a function can be used like a class in JavaScript.

Example: Class in JavaScript

function Person() {

this.firstName = "unknown";

this.lastName = "unknown";

}

var person1 = new Person();

person1.firstName = "Steve";

person1.lastName = "Jobs";

alert(person1.firstName + " " + person1.lastName);var person2 = new Person();

person2.firstName = "Bill";

person2.lastName = "Gates";

alert(person2.firstName + " " + person2.lastName );Add Methods in a Class

We can add a function expression as a member variable in a function in JavaScript. This function expression will act like a method of class.Example: Method in Class

function Person() {

this.firstName = "unknown";

this.lastName = "unknown";

this.getFullName = function(){

return this.firstName + " " + this.lastName;

}

};var person1 = new Person();person1.firstName = "Steve";

person1.lastName = "Jobs";

alert(person1.getFullName());

var person2 = new Person();

person2.firstName = "Bill";

person2.lastName = "Gates";

alert(person2.getFullName());

In the above example, the Person function includes function expression that is assigned to a member variable getFullName. So now, getFullName() will act like a method of the Person class. It can be called using dot notation e.g. person1.getFullName().

Multiple Properties

Specify more than one property in defineProperties() method as shown below.

Example: Multiple Properties

function Person(firstName, lastName, age) {

var \_firstName = firstName || "unknown";

var \_lastName = lastName || "unknown";

var \_age = age || 25;

Object.defineProperties(this, {

"FirstName": {

get: function () { return \_firstName },

set: function (value) { \_firstName = value }

},

"LastName": {

get: function () { return \_lastName },

set: function (value) { \_lastName = value }

},

"Age": {

get: function () { return \_age },

set: function (value) { \_age = value }

}

});

this.getFullName = function () {

return this.FirstName + " " + this.LastName;

}

};var person1 = new Person();

person1.FirstName = "John";

person1.LastName = "Bond";

alert(person1.getFullName());As you know, object in JavaScript can be created using object literal, object constructor or constructor function. Object includes properties. Each property can either be assigned a literal value or a function.// object literal

var person = {

firstName:'Steve',

lastName:'Jobs'

};// Constructor function

function Student(){

this.name = "John";

this.gender = "Male";

this.sayHi = function(){

alert('Hi');

}

}

var student1 = new Student();

console.log(student1.name);

console.log(student1.gender);

student1.sayHi();

https://www.tutorialsteacher.com/javascript/javascript-object-in-depthhttps://www.tutorialsteacher.com/javascript/this-keyword-in-javascripthttps://www.tutorialsteacher.com/javascript/new-keyword-in-javascriptSo, use prototype property of a function in the above example in order to have age properties across all the objects as shown below.Example: prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}Student.prototype.age = 15;var studObj1 = new Student();

alert(studObj1.age); // 15var studObj2 = new Student();

alert(studObj2.age); // 15The magic of this is closure. In other words, the sayHi() function is a closure.A closure is a function that preserves the outer scope in its inner scope.

function greeting() {

let message = 'Hi';

function sayHi() {

console.log(message);

} return sayHi;

}

let hi = greeting();

hi(); //

Prototype in JavaScript

JavaScript is a dynamic language. You can attach new properties to an object at any time as shown below.Example: Attach property to object

function Student() {

this.name = 'John';

this.gender = 'Male';

}var studObj1 = new Student();

studObj1.age = 15;

alert(studObj1.age); // 15var studObj2 = new Student();

alert(studObj2.age); // undefined

As you can see in the above example, age property is attached to studObj1 instance. However, studObj2 instance will not have age property because it is defined only on studObj1 instance.So what to do if we want to add new properties at later stage to a function which will be shared across all the instances?The answer is Prototype.The prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.Every function includes prototype object by default.The prototype object is special type of enumerable object to which additional properties can be attached to it which will be shared across all the instances of it's constructor function.So, use prototype property of a function in the above example in order to have age properties across all the objects as shown below.Example: prototype

function Student() {

this.name = 'John';

this.gender = 'M';

}Student.prototype.age = 15;var studObj1 = new Student();

alert(studObj1.age); // 15var studObj2 = new Student();

alert(studObj2.age); // 15Both for..in and for..of are looping constructs which are used to iterate over data structures. The only difference between them is the entities they iterate over:for..in iterates over all enumerable property keys of an object

for..of iterates over the values of an iterable object. Examples of iterable objects are arrays, strings, and NodeLists.

JavaScript | Promises

Difficulty Level : Easy

Last Updated : 06 Dec, 2021

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code. Prior to promises events and callback functions were used but they had limited functionalities and created unmanageable code.

Multiple callback functions would create callback hell that leads to unmanageable code. Also it is not easy for any user to handle multiple callbacks at the same time.

Benefits of Promises

Improves Code Readability

Better handling of asynchronous operations

Better flow of control definition in asynchronous logic

Better Error Handling

A Promise has four states:

fulfilled: Action related to the promise succeeded

rejected: Action related to the promise failed

pending: Promise is still pending i.e. not fulfilled or rejected yet

settled: Promise has fulfilled or rejected

A promise can be created using Promise constructor.

Syntax

var promise = new Promise(function(resolve, reject){

//do something

});A promise can be created using Promise constructor.

Syntax

var promise = new Promise(function(resolve, reject){

//do something

});

Parameters

Promise constructor takes only one argument which is a callback function (and that callback function is also referred as anonymous function too).

Callback function takes two arguments, resolve and reject

Perform operations inside the callback function and if everything went well then call resolve.

If desired operations do not go well then call reject.

var promise = new Promise(function(resolve, reject) {

const x = "geeksforgeeks";

const y = "geeksforgeeks"

if(x === y) {

resolve();

} else {

reject();

}

});

promise.

then(function () {

console.log('Success, You are a GEEK');

}).

catch(function () {

console.log('Some error has occurred');

});

Output:Success, You are a GEEK

Promise Consumers

Promises can be consumed by registering functions using .then and .catch methods.1. then() Promise Consumers

Promises can be consumed by registering functions using .then and .catch methods.1. then()

then() is invoked when a promise is either resolved or rejected. It may also be defined as a career which takes data from promise and further executes it successfully.Parameters:

then() method takes two functions as parameters. First function is executed if promise is resolved and a result is received.

Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to handle error using .catch() method

2. catch() catch() is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error.

Done…………………………

map() vs forEach()

Some of the difference between map() and forEach() methods are listed below −

* The map() method returns a new array, whereas the forEach() method does not return a new array.
* The map() method is used to transform the elements of an array, whereas the forEach() method is used to loop through the elements of an array.
* The map() method can be used with other array methods, such as the filter() method, whereas the forEach() method cannot be used with other array methods.

**Using new keyword:**Regular functions created using function declarations or expressions are ‘constructible’ and ‘callable’. Since regular functions are constructible, they can be called using the ‘new’ keyword. However, the arrow functions are only ‘callable’ and not constructible. Thus, we will get a run-time error on trying to construct a non-constructible arrow function using the new

let x = function(){

    console.log(arguments);

    };

    var y= new x(1,2,3);

let x = ()=> {

        console.log(arguments);

    };

    new x(1,2,3);

error

**Availability of arguments objects:**Arguments objects are not available in arrow functions, but are available in regular functions. Example using **regular** ():

let user = {

        show(){

            console.log(arguments);

        }

    };

    user.show(1, 2, 3);

    let user = {

            show\_ar : () => {

            console.log(...arguments);

        }

    };

    user.show\_ar(1, 2, 3);

error

Answer

**setTimeout** is simply like calling the function after delay has finished. Whenever a function is called it is not executed immediately, but queued so that it is executed after all the executing and currently queued eventhandlers finish first. setTimeout(,0) essentially means execute after all current functions in the present queue get executed. No guarantees can be made about how long it could take.

**setImmediate** is similar in this regard except that it doesn't use queue of functions. It checks queue of I/O eventhandlers. If all I/O events in the current snapshot are processed, it executes the callback. It queues them immediately after the last I/O handler somewhat like process.nextTick. So it is faster.

Basically, a function which takes another function as an argument or returns a function is known as a higher order function. Let's deep dive a bit to see both types of implementation, that is: Passing a function as an argument to another function. Returning a function from another function.09-

**Call** is a function that helps you change the context of the invoking function. In layperson's terms, it helps you replace the value of this inside a function with whatever value you want.

**Apply** is very similar to the call function. The only difference is that in apply you can pass an array as an argument list.

**Bind** is a function that helps you create another function that you can execute later with the new context of this that is provided.

Now we will look at some basic examples of the call, apply, and bind functions. Then we will look at an example were we will be constructing our own function similar to the map function.

## How to Use the Call Function in JavaScript

call is a function that you use to change the value of this inside a function and execute it with the arguments provided.

Here is the syntax of the call function:

func.call(thisObj, args1, args2, ...)

Where,

* **func** is a function that needs to be invoked with a different this object
* **thisObj** is an object or a value that needs to be replaced with the this keyword present inside the function func
* **args1, args2** are arguments that are passed to the invoking function with the changed this object.

Note that if you invoke a function without any thisObj argument, then JavaScript considers this property to be a global object.

Now that we have some context around what the call function is, let's start off by understanding it in more detail with some examples.

### How to call a function with different contexts in JS

Consider the below example. It consists of 3 classes – Car, Brand1, and Brand2.

function Car(type, fuelType){

this.type = type;

this.fuelType = fuelType;

}

function setBrand(brand){

Car.call(this, "convertible", "petrol");

this.brand = brand;

console.log(`Car details = `, this);

}

function definePrice(price){

Car.call(this, "convertible", "diesel");

this.price = price;

console.log(`Car details = `, this);

}

const newBrand = new setBrand('Brand1');

const newCarPrice = new definePrice(100000);

//

Car details = setBrand { type: 'convertible', fuelType: 'petrol', brand: 'Brand1' }

Car details = definePrice { type: 'convertible', fuelType: 'diesel', price: 100000 }

If you look carefully, you can see that we use the call function to invoke the Car function on two occasions. Firstly, in the setBrand and then in the definePrice functions.

In both of these functions, we invoke the Car function with this object representing to the respective functions themselves. For example, inside setBrand, we call the Car function with the this object belonging to its context. The case is similar for definePrice.

How to call a function with no arguments in JS

Consider the below example:

const newEntity = (obj) => console.log(obj);

function mountEntity(){

this.entity = newEntity;

console.log(`Entity ${this.entity} is mounted on ${this}`);

}

mountEntity.call();

In this example, we invoked the function mountEntity with no thisObj argument. In such cases, JavaScript refers to the global object.

How to Use the Apply Function in JavaScript

The Apply function is very similar to the Call function. The only difference between call and apply is the difference in how arguments are passed.

In apply, arguments you can pass an argument as an array literal or a new array object.

Here is the syntax for the apply function:

func.apply(thisObj, argumentsArray);

Where,

func is a function that needs to be invoked with a different this object

thisObj is an object or a value that needs to be replaced with the this keyword present inside the function func

argumentsArray can be an array of arguments, array object, or the arguments keyword itself.

As you can see above, the apply function has different types of syntaxes.

The first syntax is a simple one. You can pass in an array of arguments like below:

func.apply(thisObj, [args1, args2, ...]);

The second syntax is where we can pass in the new array object to it:

func.apply(thisObj, new Array(args1, args2));

The third syntax is where we can pass in the arguments keyword:

func.apply(thisObj, arguments);

arguments is a special object available inside a function. It contains values of the arguments that are passed to a function. You can use this keyword with the apply function to take any number of arbitrary arguments.

The best part about apply is we don’t need to take care of the number of arguments that are passed to the invoking function. Because of its dynamic and versatile nature, you can use it in complicated situations.

Let’s look at the same example as above, but this time we'll use the apply function.

function Car(type, fuelType){

this.type = type;

this.fuelType = fuelType;

}

function setBrand(brand){

Car.apply(this, ["convertible", "petrol"]); //Syntax with array literal

this.brand = brand;

console.log(`Car details = `, this);

}

function definePrice(price){

Car.apply(this, new Array("convertible", "diesel")); //Syntax with array object construction

this.price = price;

console.log(`Car details = `, this);

}

const newBrand = new setBrand('Brand1');

const newCarPrice = new definePrice(100000);

And here is an example that showcases how you'd use the arguments keyword:

function addUp(){

//Using arguments to capture the arbitrary number of inputs

const args = Array.from(arguments);

this.x = args.reduce((prev, curr) => prev + curr, 0);

console.log("this.x = ", this.x);

}

function driverFunc(){

const obj = {

inps: [1,2,3,4,5,6]

}

addUp.apply(obj, obj.inps);

}

driverFunc();

How to Use the Bind Function in JavaScript

The bind function creates a copy of a function with a new value to the this present inside the calling function.

Here is the syntax for the bind function:

func.bind(thisObj, arg1, arg2, ..., argN);

Where,

func is a function that needs to be invoked with a different this object

thisObj is an object or a value that needs to be replaced with the this keyword present inside the function func

arg1, arg2…argN – you can pass 1 argument to the calling function or more than that, similar to the call function.

The bind function then returns a new function that consists of a new context to the this variable present inside the calling function:

func(arg1, arg2);

Now this function func can be executed later with the arguments.

Let's look at a classic example of how to use a bind function with the help of a class-based React component:

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

counter: 1

};

}

handleCode() {

console.log("HANDLE CODE THIS = ", this.state);

}

render() {

return <button onClick={this.handleCode}>Click Me</button>;

}

}

Consider the above App component. It constitutes the following things:

constructor is a function that gets called a class and is instantiated with a new keyword.

render is a function that executes/renders the JSX code.

handleCode is a class method that logs the state of the component.

If we click on the Click Me button then we will receive an error stating: Cannot read properties of undefined (reading 'state').

Have you ever wondered why this issue occurs? 🤔🤔

You might be expecting that we should be able to access the state of the class since handleCode is a class method. But here is the catch:

this inside the handleCode is not same as that of the class’s this.

Inside a class, this is a regular object that has non-static class methods as its properties. But this inside the handleCode will refer to a different context.

To be honest, the value of this in this scenario depends on from where the functions is being called. If you see, the handleCode is being called on onClick event.

But at this stage, we will get undefined for the context of this present inside the handleCode function.

We're trying to call the state property of an undefined value. Therefore, this leads to the above error.

We can fix this by providing the right context of this inside the handleCode method. You can do this with the bind method.

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

counter: 1

};

this.handleCode = this.handleCode.bind(this); //bind this function

}

handleCode() {

console.log("HANDLE CODE THIS = ", this.state);

}

render() {

return <button onClick={this.handleCode}>Click Me</button>;

}

}

The bind will create a new function and store it inside the this object with a new property as handleCode. Bind will make sure that the class’s this context gets applied to the this present inside the handleCode function.

How to Create Your Own map Function

Now that we have all the necessary things, let's start off by creating our own map function. Let's first understand the things that we will need to build our own map function.

Here is the syntax of the map function:

arr.map(func)

Where,

arr is an array on which the map is called.

func is the function that needs to run on each element of an array.

The basic functionality of a map function is simple:

It is a function that walks through each element of an array and applies the function that is passed as an argument. The return type of a map is again an array with func being applied on each element.

Now we understand the requirements, so we can move on to create our own map function. Here is the code of our new map function:

function newMap(func){

let destArr = [];

const srcArrLen = this.length;

for(let i = 0; i < srcArrLen; i++){

destArr.push(func.call(this, this[i]));

}

return destArr;

}

Let's understand the above function bit-by-bit:

This function accepts an argument called func. It's nothing but a function that needs to be called on each element of an array.

The other parts of the code are pretty self explanatory. We will focus on the following line: destArr.push(func.call(this, this[i]));

This line does two things:

1. Pushes the changes into the destArr

2. Executes the func with the help of call method. Here the call method (as explained in the previous sections) will execute the func method with a new value to the this object present inside the func method.

Now let's take a look at how we are going to execute our newMap function. The below approach of adding a new method to the existing primitive data type is not recommended but still we will do it for the sake of this article.

1. What are logical operators in JavaScript?

Logical operators allow developers to compare variables and perform tasks based on the result of the comparison. As a hiring manager, you’d ask this question to gauge the candidate’s familiarity with the language and its fundamental features. Your candidate should be able to explain each logical operator and their behavior – stepping through each operand and computing its output.

There are four logical operators in JavaScript:

|| – OR

&& – AND

! – NOT

?? – Nullish Coalescing (see next question)

OR

The “OR” operator is represented by two vertical lines (||). In JavaScript, the “OR” operator evaluates the values from left to right and returns the first truthy value. If none of the values are truthy, the “OR” operator will return the last operand.

let x = 'Hello' || false; // x is equal to 'Hello' (first truthy value)

let y = false || 'Yes' || 1; // y is equal to 'Yes' (first truthy value)

let z = false || undefined || 0; // since all are false, z is equal to 0 (the last value)

AND

The “AND” operator is represented by two ampersands (&&). In JavaScript, the “AND” operator evaluates the values from left to right and returns the first falsy value. If all the operands are true, the “AND” operator will return the last operand.

let x = 'Hello' && false; // x is equal to 'false' (first falsy value)

let y = 0 && 'Yes' && true; // y is equal to 0 (first falsy value)

let z = true && 'Hello' && 10; // since all are truthy, z is equal to 10 (the last value)

NOT

The “NOT” operator is represented by an exclamation mark (!). the “NOT” operator accepts a single argument and returns the inverse of its boolean value. The argument is first converted into a boolean (true or false). The argument’s boolean value is then inverted and returned (true becomes false and vice versa).

let x = !false; // x is equal to true

let y = !('Hello'); // y is equal to false ('Hello' is truthy)

2. What is the nullish coalescing operator in JavaScript?

Nullish coalescing is an addition to JavaScript that helps provide a nicer and more concise syntax for getting the first “defined” value. The candidate should be able to explain what nullish coalescing is at a high-level and how to use the operator when asked this JS interview question.

Nullish coalescing is a JavaScript logical operator represented by two question marks (??). Nullish coalescing is an operator that returns the first “defined” value. “defined” here refers to an expression whose value is neither null nor undefined.

Let’s look at how the operator works.

a ?? b

The output of the code above is as follows:

if a is defined, the value of a is returned

if a isn’t defined, the value of b is returned

Let’s look at a few examples of this operator when given a different set of arguments.

let undefinedUser;

console.log(undefinedUser ?? 'Anonymous'); // will print 'Anonymous'

let definedUser = 'Ryan';

console.log(definedUser ?? 'Anonymouse') // will print 'Ryan'

3. What is the difference between == and === operators?

JavaScript has two ways to test equality. Understanding the subtle difference between the two methods is important to prevent misusing each method. The candidate should be able to explain the differences and demonstrate a basic understanding of each method’s usage.

Both double-equals (==) and triple-equals (===) are comparison operators meant to compare the equality of two values.

Double-equals only compares the value of the element. Double-equals does type coercion, where the type of the variables is converted to a common type before checking their values. In short, the double-equals operator will return true for any two variables with the same value regardless of their type.

Triple-equals on the other hand, check for strict equality – both the value and type of the element being compared. Both the value and type of being compared has to match to satisfy the triple-equal operator

let x = 1; // number 1

let y = '1'; // string 1

if (x == y) {

// true! Both x and y's values are equal

}

if (x === y) {

// false! Although both x and y has the same value, x is a number where as y is a string

}

Read More: Common Interview Questions for Software Developer Jobs (Non-Technical)

4. What is a spread operator?

The spread operator is a feature from ES6 to help unpack an element. Candidates being asked this interview question on JavaScript should be able to demonstrate an understanding of how the spread operator expands an element – being able to come up with the output of a spread operator.

Spread operator allows iterables such as arrays, objects, and strings to be expanded into single arguments. The spread operator is denoted by three dots (...) followed by the variable to be expanded.

Let’s look at an example where we combine two arrays using the spread operator. Below we have a male and female array containing a few strings each. The combined array combines the expanded male and female array, creating a single array with the contents from both male and female.

const male = ['Mike', 'Alex', 'Bob'];

const female = ['Sam', 'Maggie'];

const combined = [...male, ...female];

console.log(combined); // will print ['Mike', 'Alex', 'Bob', 'Sam', 'Maggie']

5. Explain loops in JavaScript.

We often require repeat actions. Loops are a way to execute the same code multiple times. The candidate should be able to explain how to use loops in JavaScript. An ideal answer should include the pros and cons of each looping method and its respective applications.

There are two main ways to create loops in JavaScript – while and for. Both methods consist of a condition to stop the loop and a “loop body”, the code that will be executed multiple times.

while loop

while loops are typically used when the “loop body” needs to be repeated an unknown number of times until the condition is met.

The code snippet below shows a simple while loop that prints the value of i on every iteration and stops when i is equal to 3.

let i = 0;

while (i < 3) {

console.log(i); // will print 0, 1, and 2

i++;

}

for loop

A for loop, on the other hand, is better suited for executing the “loop body” a fixed number of times.

The same loop in the previous code snippet can be re-written using a for loop this way:

for (let i = 0; i < 3; i++) {

console.log(i); // will print 0, 1, and 2

}

6. Explain the “this” keyword.

The this keyword is widely used in JavaScript applications. It behaves differently compared to other languages such as Java and Python. The candidate should have a thorough understanding of how the this keyword works and how it relates to its context.

The this keyword behaves differently depending on the caller’s context. Let’s look at a few contexts and what the this keyword refers to in each one

Global context

Global context refers to anything outside of any function – global object. this refers to the window object in web browsers and global object in Node.js applications.

If you assign a property to the this object in a web browser, JavaScript will add that property to the window object.

// in web browsers

this.name = 'Adam';

console.log(window.name) // will print 'Adam' in your console

Function context

Functions can be invoked in four different ways.

Function invocation

Method invocation

Constructor invocation

Indirect invocation

Each of the invocations results in a different this behavior.

Function invocation

Depending on whether you are using “strict mode” or not, the this keyword refers to different values.

By default, the this keyword refers to the window or global object depending on where you are running the application.

// in web browsers

function callMe() {

if (this === window) {

// true!

}

}

In “strict mode”, JavaScript sets the this keyword to undefined.

"use strict"

function callMe() {

if (this === window) {

// false!

}

if (this === undefined) {

// true!

}

}

Method invocation

When you call a method of an object (getName in the example below), the this keyword is set to the object that owns the method (user in the example below).

let user = {

name: 'Bob',

getName: function() {

return this.name;

}

}

console.log(user.getName()); // will print 'Bob' in your console

Constructor invocation

Constructor invocation is when the new keyword is used to create a new instance of a function object.

The new User('Bob') is a constructor invocation of the User function where a new instance of the User function is created. The this keyword in this case refers to the newly created object.

function User(name) {

this.name = name;

}

User.prototype.getName = function() {

return this.name;

}

let user = new User('Bob');

console.log(user.getName()); // will print 'Bob' in your console

Indirect invocation

Indirect invocation is when the callee of the function uses the call or apply keyword to call a function. Both these methods allow passing in the this value (bob and adam in the example below) as a parameter.

function sayHello(greeting) {

console.log(`${gretting} ${this.name}`);

}

let bob = {

name: 'Bob'

};

let adam = {

name: 'Adam'

};

sayHello.call(bob, "Hello"); // will print 'Hello Bob' in your console

sayHello.call(adam, "Hi"); // will print 'Hi Adam in your console

The apply keyword is identical to the call keyword above. Instead of accepting a single value as the second parameter, the apply keyword expects an array of values.

sayHello.call(bob, ["Hello"]); // will print 'Hello Bob' in your console

sayHello.call(adam, ["Hi"]); // will print 'Hi Adam in your console

Read More: 31 Questions to Ask at an Interview for Software Development Jobs

7. What are the differences between call, apply, and bind?

JavaScript has multiple ways to indirectly invoke a function. Your candidate needs to understand the differences between each and their use cases. You, as the candidate, should be able to explain not only their differences conceptually but also their use case and the reason behind it.

call, apply, and bind are different methods to tie a function to an object and call the function within the specified context.

call

The call method invokes the function with the specified context – the function is called as if it’s part of the object.

The function sayHello in the example below references this.name which is part of the user object (out of the scope of the sayHello function). We can use the call function and pass in the user object as the first argument to tie the sayHello function and the user object momentarily, giving it access to the this.name property.

let user = { name: 'Bill' };

function sayHello(greeting){

console.log(`${greeting} ${this.name}`)

}

sayHello('Hello'); // will print 'Hello'

sayHello.call(user, 'Hello'); // will print 'Hello Bill'

apply

The apply method is identical to the call method with the difference being in how each method accepts their arguments. The call method accepts an argument list, whereas the apply method accepts an array of arguments.

Using the same example as above, we can convert the call method to apply by wrapping the function’s arguments (excluding the context – user) in an array before passing it to apply method.

let user = { name: 'Bill' };

function sayHello(greeting){

console.log(`${greeting} ${this.name}`)

}

sayHello.apply(user, ['Hello']); // will print 'Hello Bill'

bind

Unlike call and apply, the bind method doesn’t execute the function immediately. Instead, it returns a function that is tied to the object that can be executed later.

Let’s update the example again to use the bind method. We’ll first bind the sayHello function to the user object and assign it to a new variable (helloBill). We can then invoke that function calling it as you would a regular function.

let user = { name: 'Bill' };

function sayHello(greeting){

console.log(`${greeting} ${this.name}`)

}

let helloBill = sayHello.bind(user);

helloBill('Hello'); // will print 'Hello Bill'

8. What are anonymous functions in JavaScript?

Anonymous functions serve numerous purposes in JavaScript. You might ask standard JavaScript interview questions like this one to gauge your candidates’ knowledge of functions in JavaScript and the various ways a function can be created and used. The candidate should be able to explain the difference between anonymous functions and other types of functions and what they are commonly used for

An anonymous function is a function that does not have any name associated with it. We usually use the function keyword followed by the function’s name to define a function in JavaScript. Anonymous functions omit the function name, making it not accessible after its creation.

An anonymous function can only be accessed by a variable. The anonymous nature of the function makes it great for passing in functions as arguments to other functions (higher-order functions) and functions that are invoked immediately upon initialization.

The following snippet is an example of an anonymous function that is assigned to a variable (sayHello). The function can then be called by calling the variable name (sayHello) with the required arguments.

let sayHello = function (name) {

console.log(`Hello ${name}`);

};

sayHello('Chris'); // will print 'Hello Chris'

9. What is hoisting in JavaScript?

Hoisting allows functions to be used safely before they are declared. This question will test the candidate’s familiarity with the JavaScript language and how classes, functions, and variables are interpreted by JavaScript. A basic understanding of hoisting can prevent unexpected errors caused by an incorrect order of declaration, initialization, and reference of a property. You may get other JavaScript hoisting interview questions, so study up!

Hoisting refers to the process where the interpreter moves the declaration of classes, functions, or variables to the top of their scope, before their execution.

Hoisting allows developers to reference variables, functions, and classes before they are declared. Without hoisting, the order of the example below would need to be reversed, with the function declaration first, followed by the caller.

sayHello("Sam");

function sayHello(name) {

console.log(`Hello ${name}`);

}

However, JavaScript only hoists its declarations, not their initializations. Referencing a variable before its initialization would return the variable’s default value (undefined for variables declared using the var keyword).

console.log(name); // will print 'undefined' from hoisted var declaration below

var name; // declaration

name = 'Mike'; // initialization

console.log(name); // will print 'Mike' after the previous line (initialization) is executed

Read More: 8 Common Interview Mistakes Remote Software Developers Make

10. What is a callback function in JavaScript?

JavaScript runs code sequentially from the top-down. However, sometimes, we need code to run after something has happened (i.e. asynchronous operations). Callback functions are a way to make sure a function runs only after the set of operations is completed. A candidate should be able to explain both how callback functions work and how it relates to asynchronous programming.

A callback function is a function passed into another function as an argument. The callback function is then invoked inside the callee to complete an action.

The example below shows how the callback function is passed into and executed by another function. The last line (greetPerson(sayHello)) passes the sayHello function to the greetPerson function. greetPerson then executes the sayHello function by calling the callback variable, passing in the name value returned by the prompt function.

function sayHello(name) {

console.log('Hello ' + name);

}

function greetPerson(callback) {

let name = prompt('Name'); // displays a prompt to the user to submit a name

callback(name);

}

greetPerson(sayHello);

11. What are Promises in JavaScript?

Promises are an effective way to handle asynchronous operations in JavaScript. A candidate should be able to demonstrate a high-level understanding of Promises and how they handle asynchronous operations. An ideal answer would include the tradeoffs of using Promises and how they compare to callbacks and events.

A Promise is a proxy for a value not necessarily known when the promise is created. A promise is a way to handle asynchronous operations in JavaScript. You can think of Promises as an alternative to callbacks and events.

Promises are ideal for handling multiple asynchronous operations, providing a better flow of control definition and error handling.

Let’s look at an example of a Promise that waits for a setTimeout to complete:

let timeoutPromise = new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Promise completed!'); // resolves the promise after 1 second

}, 1000);

});

timeoutPromise.then((result) => {

console.log(result); // will print 'Promise completed!' after 1 second (when the Promise completes)

});

12. What are the different states of a Promise?

Understanding the different states of a promise is important when dealing with promises to avoid unwanted side effects. You might ask this question to gain insight into the candidate’s familiarity with promises beyond the high-level concept.

Because of the asynchronous nature of Promises, a Promise has four states:

Pending – Promise’s initial state, waiting for the operation to complete

Fulfilled – Promise’s operation was completed successfully

Rejected – Promise’s operation failed

Settled – Promise is either fulfilled or rejected

Read More: 8 Behavioral Interview Questions Asked by Top Tech Companies

13. What is Promise chaining?

Promise chaining is a common requirement when working with multiple Promises that depend on each other. A candidate should ideally be able to explain both what promise chaining is and how it is done in JavaScript.

One of the benefits of Promises is its chaining ability. A Promise can be chained using the then and catch functions. The then function will be called when the Promise completes successfully (fulfilled) whereas the catch function will be called when the Promise failed (rejected).

Each then and catch block can contain more Promises and be further chained, providing you with granular control over how each asynchronous operation should be executed.

Let’s look at an example of chaining two Promises that waits for one second between each execution.

new Promise((resolve) => {

setTimeout(() => resolve(1), 1000);

}).then((result) => {

console.log(result); // will print '1' after 1 second

return new Promise((resolve) => {

setTimeout(() => {

resolve(2) // modify the value being resolved

}, 1000)

})

}).then((result) => {

console.log(result); // will print '2' after another 1 second

return result;

})

The first Promise in the code snippet above waits for one second before returning a result of 1. The code then goes to the then block where it executes the second Promise, waiting for another second before returning a result of 2.

14. What is Promise.all?

JavaScript interview questions like this one might be asked as a follow-up to the Promise chaining question. JavaScript provides several utility functions that help with chaining Promises – Promise.all being one of them. A candidate should be able to describe the function of this type of Promise and also how it alters the flow of the asynchronous functions.

Promise.all is a type of Promise that accepts an array of Promises and waits for each Promise to resolve. Promise.all resolves once each of the Promise inputs resolves, emitting an array of results in the then block. A rejected Promise in the input will cause the Promise.all Promise to also get rejected.

The example below shows how the Promise.all function is used to execute two Promises – Promise1 and Promise2, with a then block to capture the results of each Promise and a catch block to handle any errors.

Promise.all([Promise1, Promise2]).then(

([result1, result2]) => {

// result1 contains the result of Promise1

// result2 contains the result of Promise2

}).catch((error) => {

// when either Promise1 or Promise2 gets rejected

});

15. Explain async/await in JavaScript.

Async and await are special syntax to work with Promises. In addition to the “what”, as an interviewer, you might also want to look for practical examples of async and await usages and how it differs from the Promise then syntax.

The async keyword is placed before a function to denote that the function is asynchronous and can be used as a Promise.

The await keyword, on the other hand, tells JavaScript to wait for the async operation to complete before proceeding to the next task in the function. The await keyword can only be used in an async function.

Line 6 in the code snippet below pauses the function execution as it waits for the promise to resolve. Once the promise resolves, it will continue the execution, assigning the result of the promise to the result variable.

async function f() {

let promise = new Promise((resolve) => {

setTimeout(() => resolve('Promise resolved!'), 1000)

});

let result = await promise; // waits for 1 second, until the promise resolves

console.log(result); // will print 'Promise resolved!'

}

Read More: 10+ Tips for Preparing for a Remote Software Developer Zoom Interview

Check out our entire set of software development interview questions to help you hire the best developers you possibly can.

JavaScript Interview Questions

Machine Learning Interview Questions

MongoDB Interview Questions

TypeScript Interview Questions

Selenium Interview Questions

Spring Interview Questions

Data Engineer Interview Questions

React Interview Questions

Data Analyst Interview Questions

Vue Interview Questions

SQL Interview Questions

DevOps Interview Questions

Engineering Manager Interview Questions

Java Interview Questions

PHP Interview Questions

Ruby on Rails Interview Questions

Angular Interview Questions

Android Interview Questions

Data Warehouse Interview Questions

If you’re a developer, familiarize yourself with the non-technical interview questions commonly asked in the first round by HR recruiters and the questions to ask your interviewer!

Arc is the radically different remote job search platform for developers where companies apply to you. We’ll feature you to great global startups and tech companies hiring remotely so you can land a great remote job in 14 days. We make it easier than ever for software developers and engineers to find great remote jobs. Sign up today and get started.

More Basic JavaScript Interview Questions

Before we wrap this section up and get to the intermediate questions, here are a few other JavaScript basic interview questions you might want to ask your candidate during an interview:

What are self-invoking functions?

What is the purpose of the race method in a JavaScript Promise?

What is a pure function?

What are break and continue statements?

What is variable shadowing in JavaScript?

What is an event loop?

What is an event flow?

How do you sort elements in an array in JavaScript?

What is a debugger statement?

What is a short circuit condition?

Intermediate JavaScript Interview Questions

The following questions should test the candidate’s intermediate knowledge of JavaScript and some of its widely used features.

1. What are rest parameters?

The rest parameter is a JavaScript feature to provide a way to represent variadic functions in JavaScript. The candidate should be able to demonstrate an understanding of how the rest operator is used in a function and how its contents can be accessed.

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array. The rest operator puts the contents of the variable after the rest operator into an array (rest parameter can only be used as the last parameter of a function).

Rest operator is represented by three dots (...) followed by the variable name. The variable can then be used to access the array containing the contents of the function’s arguments.

The example below shows a function that accepts two parameters – greeting and name (with a rest operator). The rest operator on the name argument tells JavaScript to add any arguments from the second argument forward into an array called name.

function sayGreeting(greeting, ...name) {

console.log(greeting + ' ' + name);

}

sayGretting('Hello', 'Mike', 'Greg', 'Tom');

// will print "Hello ['Mike', 'Greg', 'Tom']"

2. What is memoization in JavaScript?

Optimization of processes becomes a necessity as applications grow and begin to perform heavier tasks. Memoization is an optimization technique that helps speed up expensive function calls using cached results. Understanding optimization techniques is important to keep your code fast and efficient. Your candidate should be able to explain memoization and how it relates to code optimization in general.

Memoization is an optimization technique that speeds up your code by storing the results of expensive function calls and reusing the stored result when the same input occurs again.

An expensive function refers to functions that consume a lot of resources (time and memory) during their execution due to heavy computation.

The result is immediately stored in a cache when an expensive function is called. When the same function is called again with the same parameters, it skips the computation required and returns the cached value instead. This process significantly reduces the time and memory your application consumes as the function doesn’t have to redo any calculations or computations that it has done previously.

3. What is currying in JavaScript?

Currying is an advanced technique of working with functions based on a concept from lambda calculus. You might ask intermediate JavaScript interview questions similar to this one to get an insight into the candidate’s level of understanding of functions in JavaScript. The candidate should be able to explain the concept of currying and how a function is decomposed and transformed following this concept.

Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c). In other words, currying a function means the function takes one argument at a time and returns a new function expecting the next argument. Instead of taking all arguments at the same time, currying decomposes the function into a sequence of functions with a single argument.

Let’s look at an example of two functions that accepts three arguments and returns their sum. The first function () is a regular function, whereas the second function () is the curried version.

// regular version

const add = (a, b, c)=>{

return a + b + c

}

console.log(add(1, 2, 3)) // will print 6

// curried version

const addCurry =(a) => {

return (b)=>{

return (c)=>{

return a+b+c

}

}

}

console.log(addCurry(1)(2)(3)) // will print 6

4. How do you empty a JavaScript array?

Arrays are widely used in JavaScript, making understanding their behavior and possible operations crucial when working with JavaScript. You might ask a JS question like this to gauge the candidate’s familiarity with JavaScript arrays and their operators. The candidate should be ready to explain a couple of different approaches and how they work at a high level.

There are various ways to empty an array in JavaScript. Below are a few common ways to do it.

Set the target array to an empty array.

targetArray = [];

Set the length of the target array to 0.

targetArray.length = 0;

Use the splice method to update the target array’s contents.

targetArray.splice(0, targetArray.length);

5. What is a WeakMap in JavaScript?

A WeakMap is a map where the keys are weak – values are garbage collected when there are no more references to the key/value. The candidate should be able to explain what a WeakMap is along with their use case. You might also be looking to test the candidate’s understanding of the garbage collection mechanism, so make sure they explain how WeakMap relates to garbage collection in JavaScript.

By definition, a WeakMap is a collection of key/value pairs whose keys must be objects, with values of any arbitrary JavaScript type, and which do not create strong references to its keys.

A WeakMap provides a way to extend objects externally without interfering with JavaScript’s garbage collection mechanism. Once an object used as a key is collected, the object’s corresponding values in any WeakMap become candidates for garbage collection as well.

A WeakMap is especially useful when mapping keys to information about the key is valuable only if the key has not been garbage collected.

6. What is typecasting in JavaScript?

Converting between different data types is common in programming in general, and this is important for your candidate to get right. These could be when receiving a value from an external API, a user input, a third-party library, etc. A candidate should be able to demonstrate a basic understanding of what typecasting is and how to utilize the typecasts provided by JavaScript to convert between various data types.

Typecasting or coercion means to change the data type of a value to another data type. For example, a conversion from a string to an integer or vice versa.

Coercion can either be implicit or explicit. Implicit coercion is when the type conversion is automatic, whereas explicit coercion is when a developer explicitly writes code to convert the type of a value. The latter is also known as typecasting.

There are three typecasts provided by JavaScript:

Boolean(value) – Casts the input value to a boolean value

Number(value) – Casts the input value to a float or integer value

String(value) – Casts the input value to a string

7. What are the various types of errors in JavaScript?

Every developer is bound to run into errors when programming in any language. JavaScript is no different. Debugging and resolving these errors are usually part of every developer’s day. Identifying the different error types can significantly help the developer narrow down the problem. The candidate should be able to identify and differentiate between the errors described below when asked this important JS interview question.

There are three types of errors in JavaScript:

Syntax errors – Errors that occur at interpretation time such as during the loading of a web page due to incorrect syntax in the code.

Runtime errors – Errors that occur during the runtime of the program after it is interpreted by the compiler. Calling functions that don’t exist is a common cause of this type of error.

Logical Errors – Errors caused by the code’s logic itself. They are syntactically correct and don’t necessarily cause runtime errors. To think of this in terms of a sentence in the English language – the vocabulary and grammar of the sentence are correct, however, the meaning of the sentence is incorrect.

8. What is event bubbling in JavaScript?

Dealing with events is unavoidable, especially when working with the web. Event bubbling is an important concept that is commonly used either directly or via a framework or library. The candidate should be able to demonstrate a high-level understanding of what event bubbling is and how events work in JavaScript.

Event bubbling is a way of event propagation in the HTML DOM API, where events are handled from starting from the innermost element propagating outwards to its parent elements.

Let’s look at an example where an event occurs in a nested element where both elements have a registered event handler for the triggered event. Event bubbling causes the event to be captured and handled by the innermost element first. It is then propagated to the outer elements.

9. What is event capturing in JavaScript?

Similar to the previous question on event bubbling, event capturing is the opposite of event bubbling. You might ask this question as a follow-up to previous tough JavaScript interview questions to gauge the candidate’s understanding of events in JavaScript.

Event capturing is another way of event propagation in the HTML DOM API. Unlike event bubbling, event capturing propagates an event from the outermost element to the target element.

Bubbling is disabled by default on event listeners. To use event capturing, we will need to enable bubbling by passing true to the third argument of the addEventListener function.

targetElement.addEventListener(event, handler, true)

10. What are the different ways an HTML element can be accessed in JavaScript?

If you are working with HTML with JavaScript, you will often find the need to access HTML elements from JavaScript. Although frameworks and libraries have abstractions that ease this process, a fundamental understanding of this is important. As a hiring manager or tech recruiter, you might ask this JS interview question to get an insight into the candidate’s level of understanding of accessing HTML elements beyond what the various frameworks and libraries offer.

There are four ways to access an HTML element in JavaScript:

getElementById('idName') – Returns an element with the specified idName

getElementByClass('className') – Returns all the elements containing the specified className

getElementsByTagName('tagName') – Returns all the elements that contains the specified tagName

querySelector('selectorName') – Returns the first element that contains the CSS style selector sepecified

More Intermediate JavaScript Interview Questions to Practice

Here are a few other intermediate JavaScript interview questions you might want to ask your candidates:

How do you escape characters in JavaScript?

What are JavaScript Cookies?

What are the difference between escape() and unescape() functions?

What is a microtask in JavaScript?

What are higher-order functions in JavaScript?

What are the different types of native errors in JavaScript?

What is the difference between attributes and properties in JavaScript?

What are the main differences between a forEach loop and a map loop?

How do you compare two objects in JavaScript?

How do you remove duplicates in a JavaScript array?

Advanced JavaScript Interview Questions

The following set of advanced JavaScript interview questions should test the candidate’s advanced knowledge of JavaScript and some of its more advanced concepts and features.

1. What is a closure in JavaScript?

Closures in JavaScript lets you associate data with a function that operates on that data. This has close ties to the object-oriented programming concept of objects allowing you to associate its properties with one or more methods. The candidate should not only explain what a closure is but also be able to provide examples and use cases where closures are useful.

A closure is an inner function that has access to the variables in the enclosing/outer function’s scope. The closure has access to variables from three scopes:

within its own scope

within the enclosing function’s scope

global variables

Closures are particularly useful on the web because of the web’s event-based nature. A common pattern in front-end JavaScript is as follows: you define a behavior, then attach it to an event that is triggered by user input. An example of this is a click event handler.

Below is an example of how a closure is used to change the document body’s background color when the targetElement is clicked. The changeBackground function returns an inner function that sets the document body’s style.

function changeBackground(color) {

return function() {

document.body.style.backgroundColor = color;

};

}

document.getElementById('targetElement').onclick = changeBackground('red');

2. What does the instanceof operator do?

Since we don’t explicitly define a type when we declare a variable, we sometimes need to know what the type of the variable is before performing any operation with it. The instanceof operator provides an easy way to perform a check on the variable’s type at run time. The candidate should be able to explain what is the instanceof operator and also its applications and usage.

The instanceof operator checks whether the prototype property of a constructor appears anywhere in the prototype chain of an object. In other words, the instanceof operator checks if the object is an instance of a class or not at run time.

The example below shows how the instanceof operator is used to test the type of the variable users.

const users = ['Mike', 'Bob', 'Will'];

console.log(users instanceof Array); // will print "true"

console.log(users instanceof Object); // will print "false"

console.log(users instanceof Number); // will print "false"

console.log(users instanceof String); // will print "false"

3. How do you create a shallow copy of an object?

Cloning an object in JavaScript is a common task especially when you are working with objects. The candidate should be able to demonstrate a couple of ways to create a shallow copy and what characteristics the shallow copy has as it relates to the original object. A follow-up JS question an interviewer might ask is how to create a deep copy of an object.

Deep copying means that the value of the new variable is disconnected from the original variable while a shallow copy means that some values are still connected to the original variable.

First of all, a deep copy is a copy of an object completely disconnected from the original variable. A shallow copy on the other hand is a copy of the original variable where some values are still connected to the original variable.

There are two main ways to create a shallow copy of an object in JavaScript:

1. Using the spread operator

const person = { name: 'Mike', email: 'mike@email.com' };

const clonedPerson = { ...person };

console.log(clonedPerson); // will print ( name: 'Mike', email: 'mike@email.com' )

2. Using `Object.assign()

const person = { name: 'Mike', email: 'mike@email.com' };

const clonedPerson = Object.assign({}, person);

console.log(clonedPerson); // will print ( name: 'Mike', email: 'mike@email.com' )

4. What is the difference between Object.freeze() and const?

Developers work with a lot of objects in JavaScript. Understanding how objects work is very important and can help avoid bugs and unexpected behaviors that arise from using objects incorrectly. You might ask a question like this to gauge the candidate’s understanding on JavaScript objects and its mutability-related behavior. The candidate should be able to explain the key difference between Object.freeze() and const along with their respective applications.

Both Object.freeze() and const relates to the immutability of the object. However, they each address different aspects of the object’s immutability.

const creates immutable bindings. A variable declared with the const keyword can’t be assigned a new value.

const person = {

name: 'Mike'

};

// assigning a new value to the variable person will result in an error: "person" is read-only

person = {

name: 'Bob'

};

// updating the properties inside the person variable works

person.name = 'Bob';

console.log(person); // will print "{ name: 'Bob' }"

Object.freeze(), on the other hand, makes the contents of the object immutable. You can’t modify the properties in the object.

let person = {

name: 'Mike',

};

person.name = 'Bob'; // works, as object is mutable

Object.freeze(person);

person.name = 'Will' // TypeError: Cannot assign to read-only property of 'name' of object

5. What is Strict mode in JavaScript?

Strict mode is a feature of JavaScript ES5 to enforce stricter rules. The strict mode would cause code errors that would have been ignored or failed silently to generate errors. It is often good practice to use strict mode, though every use case is different. Your candidate should be able to explain the differences between using JavaScript’s strict mode and not.

Strict mode is a mode in JavaScript to enforce stricter parsing and error handling on your JavaScript code.

The main benefit of using strict mode is catching errors early and making debugging your code easier. Common errors such as assigning a value to an undeclared variable would throw an error in strict mode alerting you that there is something wrong in your code.

You need to add the string “use strict” above the file to enable strict mode for that file.

6. What is the difference between local storage and session storage?

The web storage API contains two great tools to save key/value pairs – local storage and session storage. These are often used as an alternative to cookies. You might ask this difficult JavaScript interview question to get a better understanding of the candidate’s familiarity with client-side storage.

Local storage is a read-only property of the window interface to access a storage object. The stored data is saved indefinitely across browser sessions. Data stored in local storage is only cleared when removed explicitly through the browser’s settings or programmatically by the application.

Session storage is similar to local storage with the key difference being the data stored’s expiration time. Data stored in session storage gets cleared when the page session ends. A page session lasts as long as the tab or browser is open and persists between page reloads and restores.

7. What is the Temporal Dead Zone in JavaScript?

Temporal dead zone is a concept closely tied to hoisting. You might ask this question to gain an insight into your candidates’ familiarity with how hoisting works in JavaScript and JavaScript’s initialization process. Make sure your web developer know what temporal dead zone is and how to look for a temporal dead zone’s starts and ends.

In ES6, variables declared using let and const are hoisted similar to var, class and function. However, there is a period between the variable’s declaration and when it enters scope where the variable can’t be accessed. This period is called the Temporal dead zone (TDZ).

Below is an example where the variable name is declared using the let keyword but is assigned a value later in the code. The temporal dead zone is the period before the name variable is declared. Attempting to access the variable while in the temporal dead zone will throw a reference error.

// TDZ starts here (at the beginning of this block’s local scope)

let name; // name TDZ ends here

console.log(name); // will print 'undefined' because name's TDZ does not exist here

name = 'Bob'; // name’s TDZ does not exist here

console.log(name); // will print 'Bob' because name’s TDZ does not exist here

8. What is a generator in JavaScript?

Generators when combined with Promises are a powerful tool for asynchronous programming as they help avoid problems associated with callbacks such as inversion of control and callback hell. The candidate should have a high-level understanding of what a generator is, how generator functions work in JavaScript, and their use cases.

Generators are functions that can be exited and re-entered at a later time. These type of functions saves and persists their context and variable-bindings across re-entrances.

A generator function is defined by a function\* (keyword function followed by an asterisk (\*)) declaration.

When a generator function is initially called, it returns a type of iterator called a generator. The value is then consumed by calling the generator’s next method. The generator function continues its execution until it encounters the yield keyword.

There is no limit to the number of times a generator function can be called, however, each generator can only be iterated once.

Below is an example of a simple generator function that increments the current index and returns the incremented value.

function\* generateUserId() {

let index = 0;

while (true)

yield index++;

}

let generate = generateUserId();

console.log(generate.next().value); // 0

console.log(generate.next().value); // 1

console.log(generate.next().value); // 2

9. What is the Prototype Design Pattern?

This is a different type of question compared to the others you’ve seen in this section. This question is more conceptual, touching on design patterns, instead of discussing the features of JavaScript and how to perform a certain task. There are various design patterns used in software engineering, however since JavaScript is a prototypal language, a question about Prototype Design Pattern might be more relevant.

As the hiring manager, you might ask this question to test your candidates’ general knowledge of design patterns, their familiarity with the prototype design pattern, and how it could be used in the context of JavaScript.\_

The Prototype Design Pattern, also known as Properties Pattern is a creational design pattern based on prototypal inheritance. When an object is created, it acts as a prototype for other objects. You can think of the prototype object as a blueprint for other objects the constructor creates – the properties defined in the prototype object will also be present in the cloned object it creates.

The prototype model is mainly used for creating objects in performance-intensive situations. The prototype pattern helps eliminate the overhead of initializing an object.

Common applications of the prototype pattern are when you have a system independent of how its contents are created or when creating objects from a database whose values are copied to the newly created object.

10. What role do deferred scripts play in JavaScript?

This is an optimization question related to how JavaScript code is loaded. An understanding of how to optimize the loading of the script and its execution is important as an app grows and the delay becomes more and more noticeable. You may ask this type of question to test the candidate’s knowledge of the browser’s page load process and how familiar the candidate is with optimizing this process.

When a page loads, the browser starts to parse the HTML code. By default, when the browser runs into a script during the parsing process, it pauses processing the HTML code and starts executing the script. The browser then resumes parsing the HTML code once the script is done executing.

A slow server or a bulky script will cause a delay in the page load. Deferred scripts delay the script’s execution until the document has been parsed. This delay in the script’s execution results in a reduction in the load time of the webpage.

More JavaScript Advanced Interview Questions to Practice

Before we wrap this article up, here are a few other JavaScript advanced interview questions you might ask your candidates in your upcoming interview.

How does JavaScript garbage collector work?

What is a proper tail call?

What is the difference between shallow and deep copy?

How do you flatten a multi-dimensional array?

What is the purpose of queueMicrotask?

What is the difference between shim and polyfill?

What is the use of preventDefault method?

What is a proxy object?

What are JavaScript accessors?

-What are the differences between mutable and immutable objects?

Conclusion

Whether you’re a hiring manager looking for the ideal candidate or a developer preparing for an interview, we hope these JavaScript interview questions help you through the process.

Keep in mind that technical skills and knowledge are only one part of the hiring process. Past experience and soft skills are equally important to make sure you land (or find the right candidate for) the job.

Keep in mind that many JavaScript interview questions are open-ended and don’t have one correct answer. When you’re interviewing a potential candidate, make sure to focus on evaluating not only their technical expertise but also on their thought process and problem-solving skills.