Basic JavaScript Interview Questions

**1. What are logical operators in JavaScript?**

Logical operators allow you to compare variables and perform tasks based on the result of the comparison. A technical recruiter or hiring manager typically ask JavaScript interview questions like this one to gauge the candidate’s familiarity with the language and its fundamental features. The candidate should be able to explain each logical operator and their behavior – being able to step through each operand and compute 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 a recent addition to JavaScript to 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. It is important to understand the subtle difference between the two methods 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

}

**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 you can 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

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

JavaScript has multiple ways to indirectly invoke a function. It is important to understand the differences between each one and what their use cases are. 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. An interviewer might ask standard JavaScript interview questions like this one to gauge your knowledge of functions in JavaScript and the various ways a function can be created and used. As the candidate, make sure you 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

**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 using callbacks and events.

By definition, a Promise is a proxy for a value not necessarily known when the promise is created. 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. An interviewer might ask this question to gain an 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

**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”, an interviewer might also be looking 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!'

}

**More Basic JavaScript Interview Questions to Practice**

* 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**

**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 begins 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. The candidate should be able to explain what memoization is and also 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.

When an expensive function is called, the result is immediately stored in a cache. 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. A tech recruiter or hiring manager 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. An interviewer might ask a JS question like this to gauge the candidate’s familiarity with JavaScript arrays and their operators. You, as 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. An interviewer might also be looking to test the candidate’s understanding of the garbage collection mechanism, so make sure to also 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 to get right at interviews for JavaScript jobs. 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. The interviewer 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. A hiring manager or tech recruiter 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 be asked at your upcoming meeting:

* 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**

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 a lot with objects in JavaScript. Understanding how objects work is very important and can help avoid bugs and unexpected behaviors that arise from using objects incorrectly. An interviewer 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. You, as the 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. An interviewer 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. An interviewer might ask this question to gain an insight into your familiarity with how hoisting works in JavaScript and JavaScript’s initialization process. Make sure to explain what the temporal dead zone is. Also, be prepared to spot where a temporal dead zone 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 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 (although you could be asked about any design patterns). An interviewer might ask this question to test your knowledge of design patterns in general and your 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 your script and its execution is important as an app grows and the delay becomes more and more noticeable. An interviewer might ask this type of questions 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 be asked at 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?

**Common Javascript Object methods**

Javascript **Object** is quite special and it doesn’t follow the classical object-oriented concepts used by other languages like Java. JS objects have the prototypical inheritance which is quite different from normal class-based inheritance. We will not explain inheritance as part of this article and will focus on some common methods which are available in JavaScript’s global **Object constructor**. Even though there are many other methods available in Object constructor, we will restrict our discussion to very frequently used ones.

Here is the list of topics we will discuss in this article :

* Shallow copy — Object.assign()
* Deep copy — JSON.parse() & JSON.stringify()
* Object.create()
* Object.entries()
* Object.keys()
* Object.values()
* Object.freeze()

Before we start, lets check out a typical object initialization in JS. We have some more ways to initialize a JS object - but this is the most common way developers use :

let obj = {};

obj.name = “messi”;

obj.year= 2018;

obj.speak = function(){

return “My Name is “+this.name+” and this is year “+this.year;

}

Here you can see the name, year and speak properties. In JS methods are also properties with type function.

In the next section, we will discuss copying an object properly in JS.

**1 Copying an Object**

You can’t use the typical assignment operation to copy an object in JS as that will only lead to creating a reference to the same object.

let newObj = obj;

obj.year = 2019;

console.log(newObj.year)

// 2019

console.log(newObj.speak())

// My Name is messi and this is year 2019

Above example shows that newObj is just a reference to obj and whenever any property changes in either of them — both objects are affected.

**2 Shallow Copy**

You can create a shallow copy i.e. a top level properties copy, using Objects.assign() method

let copyObject = Object.assign({},newObj);

copyObject.name = "ronaldo";

console.log(copyObject.speak());

// My Name is ronaldo and this is year 2019

console.log(newObj.speak());

// My Name is messi and this is year 2019

This example is copying newObj and all its properties to copyObject. You can check out that speak method will only print the new name on copyObject

However, this methods fails when we have nested objects in property values. Those objects are still not copied and work as shared reference in both objects.

Look at this example

let sourceObject = {name:"neymar",country:{name:"brazil"}}

let shallowCopyObj = Object.assign({},sourceObject);

shallowCopyObj.country.name = "India";

console.log(sourceObject);

//{ name: 'neymar', country: { name: 'India' } }

You can check that sourceObject has country property as value object with name property that remains shared between the new shallowCopyObj and sourceObj. So how can we create a deep copy - the answer is not that simple if you are looking for true deep copy. We will give a small workaround in next section but that is only applicable for certain conditions(just search on google as we can have a complete article describing on how to have a deep copy)

**3 Deep Copy**

If you just need to copy only properties which are not functions — there is an efficient method. We are moving away from Object constructor here and using another global Object in JS — JSON

let deepCopyObj = JSON.parse(JSON.stringify(obj));

console.log(deepCopyObj);

//{ name: 'messi', year: 2019 }

You can check in output that we have lost our function property while copying — but this will be true deep copy. Let’s end our discussion on copying values and move to some more useful functions in Object constructor.

**4 Object.create()**

You can also create object with Object.create() function this has additional flexibility that you can choose what will be prototype of your new object.

let createObj = Object.create(obj);

console.log(createObj); //{}

createObj.name = “Pk”;

console.log(createObj.speak());

// My Name is Pk and this is year 2019

In this example obj is the prototype from which createdObj is created. Which means it can use properties of prototype due to inheritance. That’s why we can use speak() method without declaring that in createdObj.

5 Object.entries()

This is a simple function which converts JS objects to an array of arrays. With inner array is pair of key and value of the object. Let’s checkout a self-explanatory example

let person = {name:”Roger”,age:30}

let entries = Object.entries(person);

console.log(entries);

//[ [ 'name', 'Roger' ], [ 'age', 30 ] ]

**6 Object.keys()**

This function picks only keys (or property labels) of objects and returns an array

let keys = Object.keys(person);

console.log(keys);

// [ 'name', 'age' ]

**7 Object.values()**

This function picks only values of objects and returns an array

let values = Object.values(person);

console.log(values);

// [ 'Roger', 30 ]

**8 Object.freeze()**

This function freezes the object for any further changes (key or values). It may not throw any error (unless you are in strict mode) but there will be no effect of value change on your object.

let frozenObject = Object.freeze(person);

frozenObject.name = “Nadal”;

console.log(frozenObject);

//{ name: 'Roger', age: 30 }

**JS string functions**

Some of the important javascript string functions include:

* charAt(x)
* charCodeAt(x)
* concat(v1,v2..)
* fromCharcode(c1,c2)
* indexOf(substr, [start])
* lastIndexOf(substr, [start])
* match(regexp)
* replace(regexp/substr, replacetext)
* search(regexp)
* slice(start, [end])
* split(delimiter, [limit])
* substr(start, [length])
* substring(from, [to])
* toLowerCase()
* toUpperCase()
* includes()
* endsWith()
* repeat()
* valueOf()
* trim()

**1.charAt(x)**

This function will return the character at the x position within the string.

//charAt(x)

var myString = 'jQuery FTW!!!';

console.log(myString.charAt(7));

//output: F

**2.charCodeAt(x)**

This function will return the unicode value of the character at position ‘x’ within the string.

//charAt(position)

var message="jquery4u"

//alerts "q"

alert(message.charAt(1)

**3.concat(v1,v2..)**

This function combines one or more strings(argv1,v2 etc) into existing one.

//concat(v1, v2,..)

var message="Sam"

var final=message.concat(" is a"," hopeless romantic.")

//alerts "Sam is a hopeless romantic."

alert(final)

**4.fromCharcode(c1,c2)**

Function will return a string created by using specified sequence of unicode values(argc1,c2).

//fromCharCode(c1, c2,...)

console.log(String.fromCharCode(97,98,99,120,121,122))

//output: abcxyz

console.log(String.fromCharCode(72,69,76,76,79))

//output: HELLO

**5.indexOf(substr, [start])**

Searches and (if found) returns the index number of the searched character or substring within the string. If not found, -1 is returned. “Start” is an optional argument specifying the position within string to begin the search. Default is 0.

//indexOf(char/substring)

var sentence="Hi, my name is Sam!"

if (sentence.indexOf("Sam")!=-1)

alert("Sam is in there!")

**6.lastIndexOf(substr, [start])**

Searches and (if found) returns the index number of the searched character or substring within the string. Searches the string from end to the beginning. If not found, -1 is returned. “Start” is an optional argument specifying the position within string to begin the search. Default is string.length-1.

//lastIndexOf(substr, [start])

var myString = 'javascript rox';

console.log(myString.lastIndexOf('r'));

//output: 11

**7.match(regexp)**

Executes a search for a match within a string based on a regular expression. It returns an array of information or null if no match is found.

//match(regexp) //select integers only

var intRegex = /[0-9 -()+]+$/;

var myNumber = '999';

var myInt = myNumber.match(intRegex);

console.log(isInt);

//output: 999

var myString = '999 JS Coders';

var myInt = myString.match(intRegex);

console.log(isInt);

//output: null

**8.replace(regexp/substr, replacetext)**

Searches and replaces the regular expression (or sub string) portion (match) with the replaced text instead.

//replace(substr, replacetext)

var myString = '999 JavaScript Coders';

console.log(myString.replace(/JavaScript/i, "jQuery"));

//output: 999 jQuery Coders

//replace(regexp, replacetext)

var myString = '999 JavaScript Coders';

console.log(myString.replace(new RegExp( "999", "gi" ), "The"));

//output: The JavaScript Coders

**9.search(regexp)**

Tests for a match in a string. It returns the index of the match, or -1 if not found.

//search(regexp)

var intRegex = /[0-9 -()+]+$/;

var myNumber = '999';

var isInt = myNumber.search(intRegex);

console.log(isInt);

//output: 0

**10.slice(start, [end])**

This function returns a substring of the string based on the “start” and “end” index arguments, NOT including the “end” index itself. “End” is optional, and if none is specified, the slice includes all characters from “start” to end of the string.

//slice(start, end)

var text="excellent"

text.slice(0,4) //returns "exce"

text.slice(2,4) //returns "ce"

**11.split(delimiter, [limit])**

This will split a string into many according to the specified delimiter, and returns an array containing each element. The optional “limit” is an integer that lets you specify the maximum number of elements to return.

//split(delimiter)

var message="Welcome to jQuery4u"

//word[0] contains "We"

//word[1] contains "lcome to jQuery4u"

var word=message.split("l")

**12.substr(start, [length])**

This function returns the characters in a string beginning at “start” and through the specified number of characters, “length”. “Length” is optional, and if omitted, up to the end of the string is assumed.

//substring(from, to)

var text="excellent"

text.substring(0,4) //returns "exce"

text.substring(2,4) //returns "ce"

**13.substring(from, [to])**

It returns the characters in a string between “from” and “to” indexes, NOT including “to” itself. “To” is optional, and if omitted, up to the end of the string is assumed.

//substring(from, [to])

var myString = 'javascript rox';

myString = myString.substring(0,10);

console.log(myString)

//output: javascript

**14.toLowerCase()**

This will return the string with all of its characters converted to lowercase.

//toLowerCase()

var myString = 'JAVASCRIPT ROX';

myString = myString.toLowerCase();

console.log(myString)

//output: javascript rox

**15.toUpperCase()**

This will return the string with all of its characters converted to uppercase.

//toUpperCase()

var myString = 'javascript rox';

myString = myString.toUpperCase();

console.log(myString)

//output: JAVASCRIPT ROX

**16. includes()**

It is used to check whether a string contains the specified string or characters.

//includes()

var mystring = "Hello, welcome to edureka";

var n = mystring.includes("edureka");

//output: True

**17. endsWith()**

This function checks whether a string ends with specified string or characters.

//endsWith()

var mystr = "List of javascript functions";

var n = mystr.endsWith("functions");

//output: True

**18. repeat()**

This returns a new string with a specified number of copies of an existing string.

//repeat()

var string = "Welcome to Edureka";

string.repeat(2);

//output: Welcome to Edureka Welcome to Edureka

**19. valueOf()**

It is used to return the primitive value of a String object.

//valueOf()

var mystr = "Hello World!";

var res = mystr.valueOf();

//output: Hello World!

**20. trim()**

This function removes whitespace from both ends of a string.

//trim()

var str = " Hello Edureka! ";

alert(str.trim());

**List of JavaScript Array Methods**

JavaScript Array Methods

1. map( )

2. filter( )

3. sort( )

4. forEach( )

5. concat( )

6. every( )

7. some( )

8. includes( )

9. join( )

10. reduce( )

11. find( )

12. findIndex( )

13. indexOf( )

14. fill( )

15. slice( )

16. reverse( )

17. push( )

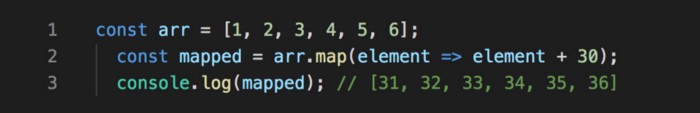
18. pop( )

19. shift( )

20. unshift( )

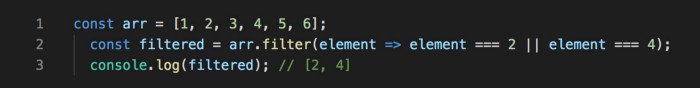
**1. map( )**

This method creates a new array with the results of calling a provided function on every element in this array.



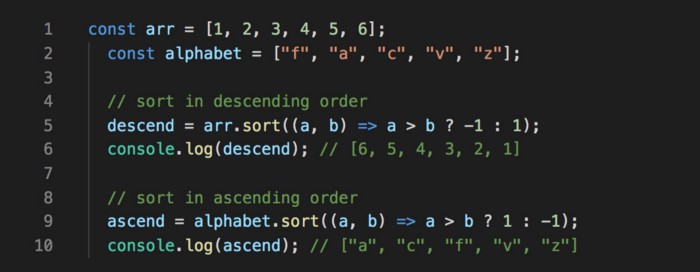
**2. filter( )**

This method creates a new array with only elements that passes the condition inside the provided function.



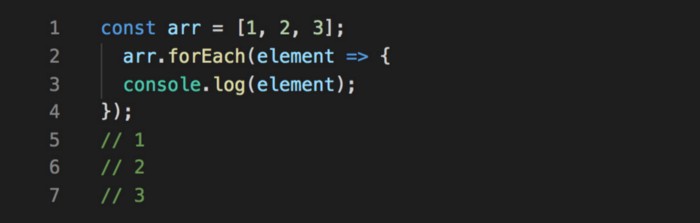
**3. sort( )**

This method is used to arrange/sort array’s elements either in ascending or descending order.



**4. forEach( )**

This method helps to loop over array by executing a provided callback function for each element in an array.



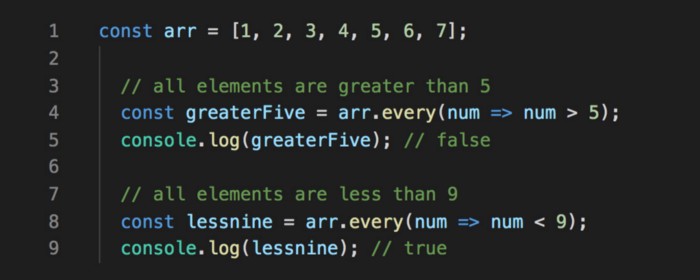
**5. concat( )**

This method is used to merge two or more arrays and returns a new array, without changing the existing arrays.



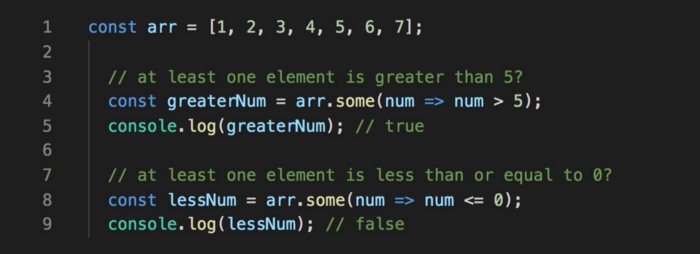
**6. every( )**

This method checks every element in the array that passes the condition, returning true or false as appropriate.



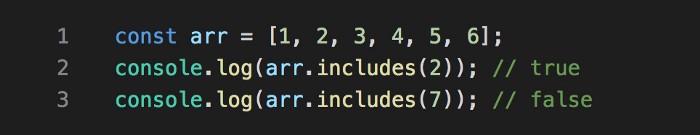
**7. some( )**

This method checks if at least one element in the array that passes the condition, returning true or false as appropriate.



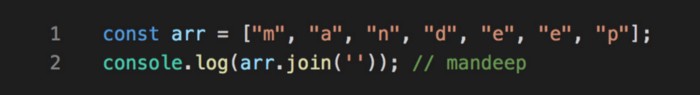
**8. includes( )**

This method checks if an array includes the element that passes the condition, returning true or false as appropriate.



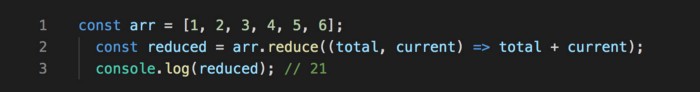
**9. join( )**

This method returns a new string by concatenating all of the array’s elements separated by the specified separator.



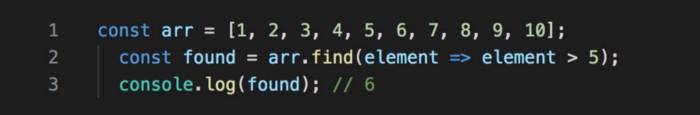
**10. reduce( )**

This method applies a function against an accumulator and each element in the array to reduce it to a single value.



**11. find( )**

This method returns the value of the first element in an array that pass the test in a testing function.



**12. findIndex( )**

This method returns the index of the first element in an array that pass the test in a testing function.



**13. indexOf( )**

This method returns the index of the first occurrence of the specified element in the array, or -1 if it is not found.



**14. fill( )**

This method fills the elements in an array with a static value and returns the modified array.



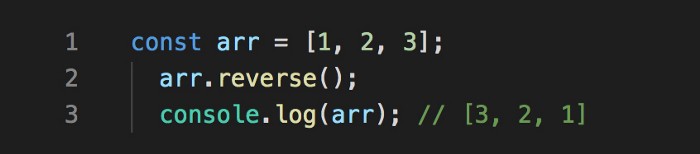
**15. slice( )**

This method returns a new array with specified start to end elements.



**16. reverse( )**

This method reverses an array in place. Element at last index will be first and element at 0 index will be last.



**17. push( )**

This method adds one or more elements to the end of array and returns the new length of the array.



**18. pop( )**

This method removes the last element from the end of array and returns that element.



**19. shift( )**

This method removes the first element from an array and returns that element.



**20. unshift( )**

This method adds one or more elements to the beginning of an array and returns the new length of the array.

