[ECMAScript Editions 3](#_Toc1)

[What is DOM? 3](#_Toc2)

[Variables 4](#_Toc3)

[Declaration 4](#_Toc4)

[Scope 4](#_Toc5)

[Hoisting: 4](#_Toc6)

[Operator 5](#_Toc7)

[“ == “ and “ === “ 5](#_Toc8)

[Data Type & Structure 5](#_Toc9)

[Primary Type 5](#_Toc10)

[Object 6](#_Toc11)

[Definition 6](#_Toc12)

[Object() Constructor 6](#_Toc13)

[The Object.create Method 6](#_Toc14)

[The Object.assign() Function 7](#_Toc15)

[Deleting Properties 7](#_Toc16)

[Object and Array De-structuring 7](#_Toc17)

[Array 7](#_Toc18)

[Basic variable assignment 7](#_Toc19)

[Assignment separate from declaration 7](#_Toc20)

[Default values 8](#_Toc21)

[Ignoring some returned values 8](#_Toc22)

[Assigning the rest of an array to a variable 8](#_Toc23)

[Object destructuring 8](#_Toc24)

[Basic assignment Example 1 9](#_Toc25)

[Assignment without declaration - Example 2 9](#_Toc26)

[rest operator Example 3 9](#_Toc27)

[Implicit Type Coercion 10](#_Toc28)

[String coercion 10](#_Toc29)

[Boolean coercion 10](#_Toc30)

[Functions 11](#_Toc31)

[passed by value and passed by reference. 11](#_Toc32)

[Rest Parameters 11](#_Toc33)

[Higher Order Functions in javascript. 12](#_Toc34)

[Immediately Invoked Function 12](#_Toc35)

[Anonymous Function 12](#_Toc36)

[Callback Function 13](#_Toc37)

[Lambda or => Function 13](#_Toc38)

[**Syntax** 13](#_Toc39)

[**OR** 13](#_Toc40)

[**OR** 14](#_Toc41)

[Array.prototype.map() and arrow function 14](#_Toc42)

[Arrow function and “this” 14](#_Toc43)

[Function Expression and Function Declaration 15](#_Toc44)

[Function Hoisting 15](#_Toc45)

[Function Currying 15](#_Toc46)

[Closures in JavaScript 16](#_Toc47)

[Array - https://javascript.info/array-methods 17](#_Toc48)

[Search 17](#_Toc49)

[indexOf/lastIndexOf and includes 17](#_Toc50)

[filter 17](#_Toc51)

[Array.every() 17](#_Toc52)

[Array.some() 17](#_Toc53)

[Travarse 17](#_Toc54)

[Iterate: forEach 17](#_Toc55)

[Array.entries() 18](#_Toc56)

[Array.keys() 18](#_Toc57)

[Transform 18](#_Toc58)

[map 18](#_Toc59)

[sort(fn) 18](#_Toc60)

[reverse 18](#_Toc61)

[reduce/reduceRight 18](#_Toc62)

[Construction 19](#_Toc63)

[Add/remove items 19](#_Toc64)

[Map 19](#_Toc65)

[JavaScript Type Conversion 20](#_Toc66)

[JavaScript Data Types 20](#_Toc67)

[The typeof Operator 20](#_Toc68)

[Example 20](#_Toc69)

[Loops 21](#_Toc70)

[The For/In Loop 21](#_Toc71)

[The For/Of Loop 21](#_Toc72)

[this - https://gist.github.com/zcaceres/2a4ac91f9f42ec0ef9cd0d18e4e71262 21](#_Toc73)

[Implcit binding 21](#_Toc74)

[explicit binding 22](#_Toc75)

[default binding 22](#_Toc76)

[new binding 22](#_Toc77)

[lexical binding 22](#_Toc78)

[Eyeballing This 23](#_Toc79)

[.bind() – The Exception 23](#_Toc80)

# 

# URLs

https://www.interviewbit.com/javascript-interview-questions/#classes

https://www.toptal.com/javascript/interview-questions

* Synthetic event
* Combine behavior of multiple browser into one API
* props
* How to prevent component rerunning in react
* How do Code Splitting happen in react ? --> react loader
* React Refs (reference) - explain with an example, style component
* HOC
* Redux and its 3 principle
* before Redux what other process --> Flux
* JSX
* What is Generics
* what you check first while using function in Generics what benefit --<type safety
* React Reconciliation --> efficiently to update the dom
* How do you search in array of 2M records ? How will be your efficient algorithm ? what could be the complexity on this
* If answer to above is binary search then is there any better option if it is too large like 100M or 200M records
* CSS - if not using bootstrap then what other
* Any animations done ?
* Any GraphQL experience ?

console.log(1 + "2" + "2");

console.log(1 + +"2" + "2");

console.log(1 + -"1" + "2");

console.log(+"1" + "1" + "2");

console.log( "A" - "B" + "2");

console.log( "A" - "B" + 2);

"122"

"32"

"02"

"112"

"NaN2"

NaN

# ECMAScript Editions

| Ver | Official Name | Description |
| --- | --- | --- |
| 1 | ECMAScript 1 (1997) | First Edition. |
| 2 | ECMAScript 2 (1998) | Editorial changes only. |
| 3 | ECMAScript 3 (1999) | Added Regular Expressions. Added try/catch. |
| 4 | ECMAScript 4 | Never released. |
| **ES5** | ECMAScript 5 (2009) - | Added "strict mode". Added JSON support. Added String.trim(). Added Array.isArray(). Added Array Iteration Methods. |

| 5.1 | ECMAScript 5.1 (2011) | Editorial changes. |
| --- | --- | --- |
| **ES6** | ECMAScript 2015 | Added let and const. Added default parameter values. Added Array.find(). Added Array.findIndex(). |

| 7 | ECMAScript 2016 | Added exponential operator (\*\*). Added Array.prototype.includes. |
| --- | --- | --- |
| 8 | ECMAScript 2017 | Added string padding. Added new Object properties. Added Async functions. Added Shared Memory. |
| 9 | ECMAScript 2018 | Added rest / spread properties. Added Asynchronous iteration. Added Promise.finally(). Additions to RegExp |

# What is DOM?

DOM stands for Document Object Model.  
  
DOM is a programming interface for HTML and XML documents.  
  
When the browser tries to render a HTML document, it creates an object based on the HTML document called DOM. Using this DOM, we can manipulate or change various elements inside the HTML document.  
  
Example of how HTML code gets converted to DOM:

# Variables

## Declaration

Var : the var keyword was used to declare a variable in JavaScript. Variables declared using var do not support block level scope. This means if a variable is declared in a loop or if block it can be accessed outside the loop or the if block. This is because the variables declared using the var keyword support hoisting.

let : let and block level safety. If we try to declare a let variable twice within the same block, it will throw an error. Consider the following example −

const : The const declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. Constants are block-scoped, much like variables defined using the let statement. The value of a constant cannot change through re-assignment, and it can't be re-declared.

var vs let at Global Scope

Variables that are declared with the var keyword in the global scope are added to the window/global object. Therefore, they can be accessed using window.variableName.

**var vs let in functional scope**

Variables declared in a functional/local scope using **var**and **let**keywords behave exactly the same, meaning , they cannot be accessed from outside of the scope.

**var vs let in block scope**

Variables declared with **var**keyword do not have block scope. It means a variable declared in block scope **{}**with the **var**keyword is the same as declaring the variable in the global scope.

Variables declared with **let**keyword inside the block scope cannot be accessed from outside of the block.

## Scope

* + Global Scope − A variable with global scope can be accessed from within any part of the JavaScript code.
  + Local Scope − A variable with a local scope can be accessed from within a function where it is declared.
  + The Let and Block Scope - The block scope restricts a variable’s access to the block in which it is declared. The var keyword assigns a function scope to the variable. Unlike the var keyword, the let keyword allows the script to restrict access to the variable to the nearest enclosing block.

## Hoisting:

Variable hoisting allows the use of a variable in a JavaScript program, even before it is declared. Such variables will be initialized to undefined by default. JavaScript runtime will scan for variable declarations and put them to the top of the function or script. Variables declared with var keyword get hoisted to the top. Consider the following example −

<script>

variable company is hoisted to top , var company = undefined

console.log(company); // using variable before declaring

var company = "TutorialsPoint"; // declare and initialized here

console.log(company);

</script>

The output of the above code will be as shown below −

undefined

TutorialsPoint

# Operator

## “ == “ and “ === “

Both are comparison operators. The difference between both the operators is that,“==” is used to compare values whereas, “ === “ is used to compare both value and types.

# Data Type & Structure

## Primary Type

undefined

When a variable is declared but not assigned, it has the value of undefined and it’s type is also undefined.

boolean

It represents a logical entity and can have only two values : true or false. Booleans are generally used for conditional testing.

number

**var** x = 3; *//without decimal* **var** y = 3.6; *//with decimal*

bigint

**var** bigInteger = 234567890123456789012345678901234567890n;

string

**var** str = "Vivek Singh Bisht"; *//using double quotes*

**var** str2 = 'John Doe'; *//using single quotes*

symbol

**var** symbol1 = Symbol('symbol');

null

It represents a non-existent or a invalid value.

**typeof** "John Doe" *// Returns "string"*

**typeof** 3.14 *// Returns "number"*

**typeof** **true** *// Returns "boolean"*

**typeof** 234567890123456789012345678901234567890n *// Returns bigint*

**typeof** **undefined** *// Returns "undefined"*

**typeof** **null** *// Returns "object" (kind of a bug in JavaScript)*

**typeof** Symbol('symbol') *// Returns Symbol*

## Object

### Definition

Like the primitive types, objects have a literal syntax: curly bracesv ({and}). Following is the syntax for defining an object.

var identifier = {

Key1:value, Key2: function () {

//functions

},

Key3: [“content1”,” content2”]

}

### Object() Constructor

JavaScript provides a special constructor function called **Object()** to build the object

var obj\_name = new Object();

obj\_name.property = value;

OR

obj\_name["key"] = value

**Using a Function Constructor**

**function Car() {**

this.make = "Ford"

this.model = "F123"

}

var obj = new Car()

console.log(obj.make)

console.log(obj.model)

### The Object.create Method

Objects can also be created using the Object.create() method. It allows you to create the prototype for the object you want, without having to define a constructor function.

**var roles = {**

type: "Admin", // Default value of properties

displayType : function() {

// Method which will display type of role

console.log(this.type);

}

}

// Create new role type called super\_role

var super\_role = Object.create(roles);

super\_role.displayType(); // Output:Admin

### The Object.assign() Function

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

Following is the syntax for the same.

Object.assign(target, ...sources)

Example − Cloning an Object

"use strict"

var det = { name:"Tom", ID:"E1001" };

var copy = Object.assign({}, det);

console.log(copy);

for (let val in copy) {

console.log(copy[val])

}

### Deleting Properties

// Removes the ‘a’ property

delete myobj.a;

console.log ("a" in myobj) // yields "false"

## Object and Array De-structuring

The term destructuring refers to breaking up the structure of an entity. The destructuring assignment syntax in JavaScript makes it possible to extract data from arrays or objects into distinct variables. The same is illustrated in the following example.

### Array

#### Basic variable assignment

const foo = ['one', 'two', 'three'];

const [red, yellow, green] = foo;

console.log(red); // "one"

console.log(yellow); // "two"

console.log(green); // "three"

#### Assignment separate from declaration

let a, b;

[a, b] = [1, 2];

console.log(a); // 1

console.log(b); // 2

#### Default values

A variable can be assigned a default, in the case that the value unpacked from the array is undefined.

let a, b;

[a=5, b=7] = [1];

console.log(a); // 1

console.log(b); // 7

#### Ignoring some returned values

function f() {

return [1, 2, 3];

}

const [a, , b] = f();

console.log(a); // 1

console.log(b); // 3

const [c] = f();

console.log(c); // 1

#### Assigning the rest of an array to a variable

When destructuring an array, you can unpack and assign the remaining part of it to a variable using the rest pattern:

const [a, ...b] = [1, 2, 3];

console.log(a); // 1

console.log(b); // [2, 3]

Be aware that a SyntaxError will be thrown if a trailing comma is used on the right-hand side of a rest element:

const [a, ...b,] = [1, 2, 3];

// SyntaxError: rest element may not have a trailing comma

// Always consider using rest operator as the last element

### Object destructuring

#### Basic assignment Example 1

When destructuring an object the variable names and the object property names must match.

<script>

let student = {

rollno:20,

name:'Prijin',

cgpa:7.2

}

//destructuring to same property name

let {name,cgpa} = student

console.log(name)

console.log(cgpa)

//destructuring to different name

let {name:student\_name,cgpa:student\_cgpa}=student

console.log(student\_cgpa)

console.log("student\_name",student\_name)

</script>

#### Assignment without declaration - Example 2

If the variable and assignment are in two different steps, then the destructuring object syntax will be surrounded by **()** as shown in the example **({rollno} = student)** −

<script>

let student = {

rollno:20,

name:'Prijin',

cgpa:7.2

}

// destructuring to already declared variable

let rollno;

({rollno} = student)

console.log(rollno)

// assign default values to variables

let product ={ id:1001,price:2000} //discount is not product property

let {id,price,discount=.10} = product

console.log(id)

console.log(price)

console.log(discount)

</script>

#### rest operator Example 3

The below example shows **destructuring** using the **rest operator** and how to destruct nested objects.

<script>

// rest operator with object destructuring

let customers= {

c1:101,

c2:102,

c3:103

}

let {c1,...others} = customers

console.log(c1)

console.log(others)

//nested objects

let emp = {

id:101,

address:{

city:'Mumbai',

pin:1234

}

}

let {address} = emp;

console.log(address)

let {address:{city,pin}} = emp

console.log(city)

</script>

The output of the above code will be as mentioned below −

101

{c2: 102, c3: 103}

{city: "Mumbai", pin: 1234}

Mumbai

## Implicit Type Coercion

### String coercion

When JavaScript sees that the operands of the expression x + y are of different types ( one being a number type and the other being a string type ) , **it converts the number type to the string type** and then performs the operation. Since after conversion, both the variables are of string type, the ‘ + ‘ operator outputs the concatenated string “33” in the first example and “24Hello” in the second example.

Type coercion also takes place when using the ‘ - ‘ operator, but the difference while using ‘ - ‘ operator is that, **a string is converted to a number** and then subtraction takes place.

### Boolean coercion

All values except **0, 0n, -0, “”, null, undefined and NaN**are truthy values.

## object Prototypes

All javascript objects inherit properties from a prototype.

For example,

Date objects inherit properties from the Date prototype

Math objects inherit properties from the Math prototype

Array objects inherit properties from the Array prototype.

On top of the chain is **Object.prototype.**Every prototype inherits properties and methods from the Object.prototype.

**A prototype is a blueprint of an object.**Prototype allows us to use properties and methods on an object even if the properties and methods do not exist on the current object.

# Functions

Functions are the building blocks of readable, maintainable, and reusable code. Functions are defined using the function keyword. Following is the syntax for defining a standard function.

## passed by value and passed by reference.

In JavaScript, primitive data types are passed by value and non-primitive data types are passed by reference.

## Rest Parameters

Rest parameters are similar to variable arguments in Java. Rest parameters doesn’t restrict the number of values that you can pass to a function. However, the values passed must all be of the same type. In other words, rest parameters act as placeholders for multiple arguments of the same type.

function fun1(...params) {

console.log(params.length);

}

fun1();

fun1(5);

fun1(5, 6, 7);

## Higher Order Functions in javascript.

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions.  
  
Higher order functions are a result of functions being first-class citizens in javascript.

**function** higherOrder2() {

**return** **function**() {

**return** "Do something";

}

}

**var** x = higherOrder2();

x() // Returns "Do something"

## Immediately Invoked Function

An Immediately Invoked Function ( known as IIFE and pronounced as IIFY) is a function that runs as soon as it is defined.

(function(){

// Do something;

})();

## Anonymous Function

Functions that are not bound to an identifier (function name) are called as anonymous functions. These functions are dynamically declared at runtime. Anonymous functions can accept inputs and return outputs, just as standard functions do. An anonymous function is usually not accessible after its initial creation.

**Example − Anonymous Function**

var f = function(){ return "hello"}

console.log(f())

The following output is displayed on successful execution of the above code.

hello

**Example − Anonymous Parameterized Function**

var func = function(x,y){ return x\*y };

function product() {

var result;

result = func(10,20);

console.log("The product : "+result)

}

product()

## Callback Function

In javascript, functions are treated as first-class citizens, they can be used as an argument of another function, can be returned by another function and can be used as a property of an object.

**Functions that are used as an argument to another function are called callback functions**

function divideByHalf(sum){

console.log(Math.floor(sum / 2));

}

function multiplyBy2(sum){

console.log(sum \* 2);

}

function operationOnSum(num1,num2,operation){

var sum = num1 + num2;

operation(sum);

}

operationOnSum(3, 3, divideByHalf); // Outputs 3

operationOnSum(5, 5, multiplyBy2); // Outputs 20

## Lambda or => Function

There are 3 parts to a Lambda function −

* **Parameters** − A function may optionally have parameters.
* The **fat arrow notation/lambda notation** (=>): It is also called as the goes to operator.
* **Statements** − Represents the function’s instruction set.

([param1, parma2,…param n] )=>statement;

**Example − Lambda Expression**

var foo = (x)=>10+x

console.log(foo(10))

**Syntax**

//Arrow function that points to a single line of code

()=>some\_expression

**or**

//Arrow function that points to a block of code

()=> { //some statements }`

**or**

//Arrow function with parameters

(param1,param2)=>{//some statement}

### Array.prototype.map() and arrow function

In the following example, an arrow function is passed as a parameter to the **Array.prototype.map() function.** The map() function executes the arrow function for each element in the array. The arrow function in this case, displays each element in the array and its index.

<script>

const names = ['TutorialsPoint','Mohtashim','Bhargavi','Raja']

names.map((element,index)=> {

console.log('inside arrow function')

console.log('index is '+index+' element value is :'+element)

})

</script>

### Arrow function and “this”

Inside an arrow function if we use **this pointer**, it will point to the enclosing lexical scope. This means arrow functions do not create a new **this pointer** instance whenever it is invoked. Arrow functions makes use of its enclosing scope. To understand this, let us see an example.

<script>

//constructor function

function Student(rollno,firstName,lastName) {

this.rollno = rollno;

this.firstName = firstName;

this.lastName = lastName;

this.fullNameUsingAnonymous = function(){

setTimeout(function(){

console.log(this.firstName+ " "+this.lastName)

},2000)

}

this.fullNameUsingArrow = function(){

setTimeout(()=>{

console.log(this.firstName+ " "+this.lastName)

},3000)

}

}

const s1 = new Student(101,'Mohammad','Mohtashim')

s1.fullNameUsingAnonymous();

s1.fullNameUsingArrow();

</script>

When an anonymous function is used with **setTimeout()**, the function gets invoked after 2000 milliseconds. A new instance of “**this”** is created and it shadows the instance of the Student function. So, the value of **this.firstName** and **this.lastName** will be **undefined**. The function doesn't use the lexical scope or the context of current execution. This problem can be solved by using an **arrow function**.

The output of the above code will be as follows −

undefined undefined

Mohammad Mohtashim

## Function Expression and Function Declaration

Function expression and function declaration are not synonymous. Unlike a function expression, a function declaration is bound by the function name.

The fundamental difference between the two is that, function declarations are parsed before their execution. On the other hand, function expressions are parsed only when the script engine encounters it during an execution.

When the JavaScript parser sees a function in the main code flow, it assumes function declaration. When a function comes as a part of a statement, it is a function expression.

## Function Hoisting

Like variables, functions can also be hoisted. Unlike variables, function declarations when hoisted, hoists the function definition rather than just hoisting the function’s name.

The following code snippet, illustrates function hoisting in JavaScript.

hoist\_function();

function hoist\_function() {

console.log("foo");

}

The following output is displayed on successful execution of the above code.

foo

However, function expressions cannot be hoisted. The following code snippet illustrates the same.

hoist\_function(); // TypeError: hoist\_function() is not a function

var hoist\_function() = function() {

console.log("bar");

};

## Function Currying

Briefly, currying is a way of constructing functions that allows partial application of a function’s arguments. What this means is that you can pass all of the arguments a function is expecting and get the result, or pass a subset of those arguments and get a function back that’s waiting for the rest of the arguments. It really is that simple.

function sum(a) {

return (b) => {

return (c) => {

return a + b + c

}

}

}console.log(sum(1)(2)(3)) // 6

const sum1 = sum(1);

const sum2 = sum1(2);

const result = sum2(3);

console.log(result); // 6

## Closures in JavaScript

Closures is an ability of a function to remember the variables and functions that are declared in its outer scope.

function makeFunc() {

var name = 'Mozilla';

function displayName() {

alert(name);

}

return displayName;

}

var myFunc = makeFunc();

myFunc();

The reason is that functions in JavaScript form closures. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created. In this case, myFunc is a reference to the instance of the function displayName that is created when makeFunc is run. The instance of displayName maintains a reference to its lexical environment, within which the variable name exists. For this reason, when myFunc is invoked, the variable name remains available for use, and "Mozilla" is passed to alert.

## generator functions?

**They can be stopped midway and then continue from where it had stopped.**

Generator functions are declared with the **function\***keyword instead of the normal **function**keyword

**function**\* genFunc(){

yield 3;

yield 4;

}

genFunc(); *// Returns Object [Generator] {}*

The generator object consists of a method called **next()**, this method when called, executes the code until the nearest **yield**statement, and returns the yield value.

For example if we run the next() method on the above code:

genFunc().next(); *// Returns {value: 3, done:false}*

# Array - https://javascript.info/array-methods

## Search

### [**indexOf/lastIndexOf and includes**](https://javascript.info/array-methods#indexof-lastindexof-and-includes)

* arr.indexOf(item, from) – looks for item starting from index from, and returns the index where it was found, otherwise -1.
* arr.lastIndexOf(item, from) – same, but looks for from right to left.
* arr.includes(item, from) – looks for item starting from index from, returns true if found.

[**find and findIndex**](https://javascript.info/array-methods#find-and-findindex)

Imagine we have an array of objects. How do we find an object with the specific condition?

Here the [arr.find(fn)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find) method comes in handy.

* let result = arr.find(function(item, index, array) {

// if true is returned, item is returned and iteration is stopped

// for falsy scenario returns undefined

});

### [**filter**](https://javascript.info/array-methods#filter)

The find method looks for a single (first) element that makes the function return true.

If there may be many, we can use [arr.filter(fn)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter).

The syntax is similar to find, but filter returns an array of all matching elements:

let results = arr.filter(function(item, index, array) {

// if true item is pushed to results and the iteration continues

// returns empty array if nothing found

});

### Array.every()

The **every()** method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

### Array.some()

The **some()** method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

## Travarse

### [**Iterate: forEach**](https://javascript.info/array-methods#iterate-foreach)

The [arr.forEach](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) method allows to run a function for every element of the array.

The syntax:

arr.forEach(function(item, index, array) {

// ... do something with item

});

1. When we need to iterate over an array – we can use forEach, for or for..of.
2. When we need to iterate and return the data for each element – we can use map.

### Array.entries()

The **entries()** method returns a new **Array Iterator** object that contains the key/value pairs for each index in the array.

#### Array.keys()

The **keys()** method returns a new **Array Iterator** object that contains the keys for each index in the array.

## Transform

### [**map**](https://javascript.info/array-methods#map)

The [arr.map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) method is one of the most useful and often used.

It calls the function for each element of the array and returns the array of results.

The syntax is:

let result = arr.map(function(item, index, array) {

// returns the new value instead of item

});

### [**sort(fn)**](https://javascript.info/array-methods#sort-fn)

The call to [arr.sort()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) sorts the array *in place*, changing its element order.

### [**reverse**](https://javascript.info/array-methods#reverse)

The method [arr.reverse](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse) reverses the order of elements in arr.

### [**reduce/reduceRight**](https://javascript.info/array-methods#reduce-reduceright)

When we need to iterate over an array – we can use forEach, for or for..of.

When we need to iterate and return the data for each element – we can use map.

The methods [arr.reduce](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) and [arr.reduceRight](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) also belong to that breed, but are a little bit more intricate. They are used to calculate a single value based on the array.

The syntax is:

let value = arr.reduce(function(accumulator, item, index, array) {

// ...

}, [initial]);

The function is applied to all array elements one after another and “carries on” its result to the next call.

Arguments:

* accumulator – is the result of the previous function call, equals initial the first time (if initial is provided).
* item – is the current array item.
* index – is its position.
* array – is the array.

## Construction

### [**Add/remove items**](https://javascript.info/array-methods#add-remove-items)

We already know methods that add and remove items from the beginning or the end:

* arr.push(...items) – adds items to the end,
* arr.pop() – extracts an item from the end,
* arr.shift() – extracts an item from the beginning,
* arr.unshift(...items) – adds items to the beginning.

# Map

|  | Map | Object |
| --- | --- | --- |
| Accidental Keys | A Map does not contain any keys by default. It only contains what is explicitly put into it. | An Object has a prototype, so it contains default keys that could collide with your own keys if you're not careful.  **Note:** As of ES5, this can be bypassed by using [Object.create(null)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create), but this is seldom done. |
| Key Types | A Map's keys can be any value (including functions, objects, or any primitive). | The keys of an Object must be either a [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String) or a [Symbol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol). |
| Key Order | The keys in Map are ordered. Thus, when iterating over it, a Map object returns keys in order of insertion. | The keys of an Object are not ordered.  **Note:** Since ECMAScript 2015, objects *do* preserve creation order for string and Symbol keys. In JavaScript engines that comply with the ECMAScript 2015 spec, iterating over an object with only string keys will yield the keys in order of insertion. |
| Size | The number of items in a Map is easily retrieved from its [size](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/size) property. | The number of items in an Object must be determined manually. |
| Iteration | A Map is an [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable), so it can be directly iterated. | Iterating over an Object requires obtaining its keys in some fashion and iterating over them. |
| Performance | Performs better in scenarios involving frequent additions and removals of key-value pairs. | Not optimized for frequent additions and removals of key-value pairs. |

# JavaScript Type Conversion

## JavaScript Data Types

In JavaScript there are 5 different data types that can contain values:

* string
* number
* boolean
* object
* function

There are 6 types of objects:

* Object
* Date
* Array
* String
* Number
* Boolean

And 2 data types that cannot contain values:

* null
* undefined

## The typeof Operator

You can use the typeof operator to find the data type of a JavaScript variable.

### Example

typeof "John"                 // Returns "string"   
typeof 3.14                   // Returns "number"  
typeof NaN                    // Returns "number"  
typeof false                  // Returns "boolean"  
typeof [1,2,3,4]              // Returns "object"  
typeof {name:'John', age:34}  // Returns "object"  
typeof new Date()             // Returns "object"  
typeof function () {}         // Returns "function"  
typeof myCar                  // Returns "undefined" \*  
typeof null                   // Returns "object"

Please observe:

* The data type of NaN is number
* The data type of an array is object
* The data type of a date is object
* The data type of null is object
* The data type of an undefined variable is **undefined** \*
* The data type of a variable that has not been assigned a value is also **undefined** \*

You cannot use typeof to determine if a JavaScript object is an array (or a date).

# Loops

## The For/In Loop

The JavaScript for/in statement loops through the properties of an object:

## The For/Of Loop

The JavaScript for/of statement loops through the values of an iterable objects

for/of lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.

The for/of loop has the following syntax:

# this - https://gist.github.com/zcaceres/2a4ac91f9f42ec0ef9cd0d18e4e71262

## Implcit binding

Implicit binding occurs when dot notation is used to invoke a function.

For example:

var MyObject = function (){

this.name = 'MyObjectName';

this.myProperty = 'property';

};

MyObject.prototype.doStuff = function (action) {

console.log(this.name + ' is ' + action + '!');

}

var obj = new MyObject();

obj.doStuff('awesome'); // prints 'MyObjectName is awesome!'

In implicit binding, **whatever is to the left of the dot** becomes the context for this in the function.

## explicit binding

For .call() we pass in the this we'd like to use, along with parameters.

myFunc.call(thisContext, param1, param2, ... );

For example, using the object declared above...

var runner = { name: 'John', myFavoriteActivity: 'running' };

MyObject.prototype.doStuff.call(runner, runner.myFavoriteActivity); // prints 'John is running!';

Since we have .call, we must ignore what appears before the dot in our function call. We are using MyObject's method and calling it on another this context: runner.

.apply() is almost the same, except we must pass in an array of parameters after our this context.

myFunc.apply(thisContext, [param1, param2, ...]);

Note: .call() can take a series of parameters as well, but they're just separated by commas.

## default binding

Default binding refers to how this is the global context whenever a function is invoked without any of these other rules. If *we aren't using a dot* and *we aren't using call(), apply(), or bind()*, our this will be our global object.

Your global context depends on where you're working. If you're in the browser, this will be the window. When programming in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode), the global context is undefined.

For example while in Chrome: function printMe = function () { console.log(this); } printMe() // prints your 'Window Object' if in the browser!

## new binding

## lexical binding

Lets try the shortest possible definition:

**Lexical Scoping** defines how variable names are resolved in nested functions: **inner functions contain the scope of parent functions even if the parent function has returned**.

That is all there is to it!

### Eyeballing This

You'll notice that none of these rules require too much work.

1. Is there a dot? Look to the left. That's this.
2. Do you see .call() or .apply()? What's passed in before the first comma? Thats this.
3. Does the function stand alone when it's invoked? Then what's your global context? That's this.

These three rules-of-thumb point to the most important rule of all: this refers to a function's *callsite* (where it is invoked).

#### .bind() – The Exception

Unfortunately, .bind() complicates matters a bit.

When called on a function, .bind() sets a this context and returns a **new function** of the same name with a bound this context.

For example:

var sayMyName = function () {

console.log('My name is ' + this.name);

};

var jake = {

name: 'Jake'

}

var sayMyName = sayMyName.bind(jake);

sayMyName(); // 'My name is Jake'

Now, each time we invoke sayMyName, we will get the context of 'jake', because this has been bound to it.

Since bind creates a persistent this context, we can't eyeball it. We have to go back and find where this is bound.

**\*\* Callbacks and this \*\***

Callbacks seem to introduce another layer of confusion to this, but they don't need to.

var MyObject = function (){

this.name = 'MyObjectName';

this.myProperty = 'property';

};

MyObject.prototype.doStuff = function (action) {

console.log(this.name + ' is ' + action + '!');

}

var obj = new MyObject();

setTimeout(obj.doStuff, 1000, 'awesome');

^ Here's our callback!

If we run the above example, we'll see that this.name isn't defined. We only get ' is awesome!' in our console.

Why? obj.doStuff has a dot! Isn't it implicitly bound?

When obj.doStuff is passed as the callback to setTimeout, *we don't invoke it*. So even though a function with a dot suggests implicit binding, this is not the function's *callsite*.

Instead, the callsite will be in setTimeout, when obj.doStuff is invoked after the delay. In this different context, this has no .name property.

We can fix our function by using .bind(), which we discussed above.

var MyObject = function (){

this.name = 'MyObjectName';

this.myProperty = 'property';

};

MyObject.prototype.doStuff = function (action) {

console.log(this.name + ' is ' + action + '!');

}

var obj = new MyObject();

setTimeout(obj.doStuff.bind(obj), 1000, 'awesome'); // prints 'MyObjectName is awesome!' after a 1 second delay.

In this case, we are binding our callback to the this context of MyObject. When the callback is invoked later, we have carried the proper this context with it.

**24. What is the use of promises in javascript?**

**Promises are used to handle asynchronous operations in javascript.**Before promises, callbacks were used to handle asynchronous operations. But due to limited functionality of callback, using multiple callbacks to handle asynchronous code can lead to unmanageable code.  
  
Promise object has four states -

* Pending - Initial state of promise. This state represents that the promise has neither been fulfilled nor been rejected, it is in the pending state.
* Fulfilled - This state represents that the promise has been fulfilled, meaning the async operation is completed.
* Rejected - This state represents that the promise has been rejected for some reason, meaning the async operation has failed.
* Settled - This state represents that the promise has been either rejected or fulfilled.

A promise is created using the **Promise**constructor which takes in a callback function with two parameters, **resolve**and **reject**respectively.



**resolve**is a function that will be called, when the async operation has been successfully completed.  
  
**reject**is a function that will be called, when the async operation fails or if some error occurs.  
  
Example of a promise:  
  
**Promises are used to handle asynchronous operations like server requests, for the ease of understanding, we are using an operation to calculate the sum of three elements.**In the function below, we are returning a promise inside a function:

**function** sumOfThreeElements(...elements){

**return** **new** Promise((resolve,reject)=>{

**if**(elements.length > 3 ){

reject("Only three elements or less are allowed");

}

**else**{

**let** sum = 0;

**let** i = 0;

**while**(i < elements.length){

sum += elements[i];

i++;

}

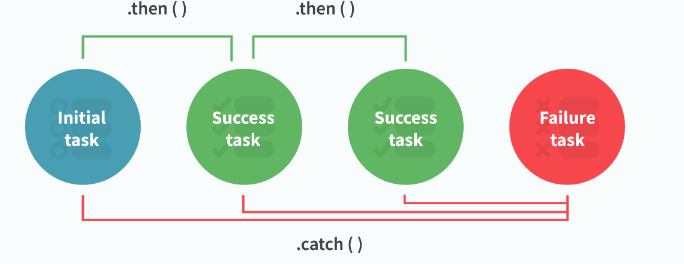
resolve("Sum has been calculated: "+sum);

}

})

}

In the code above, we are calculating the sum of three elements, if the length of elements array is more than 3, promise is rejected, else the promise is resolved and the sum is returned.  
  
We can consume any promise by attaching then() and catch() methods to the consumer.



**then()**method is used to access the result when the promise is fulfilled.  
  
**catch()**method is used to access the result/error when the promise is rejected.  
  
In the code below, we are consuming the promise:

sumOfThreeElements(4, 5, 6)

.then(result=> console.log(result))

.**catch**(error=> console.log(error));

*// In the code above, the promise is fulfilled so the then() method gets executed*

sumOfThreeElements(7, 0, 33, 41)

.then(result => console.log(result))

.**catch**(error=> console.log(error));

*// In the code above, the promise is rejected hence the catch() method gets executed*