## Promises is faster then async await-> is main ek sath sari request jati hai lekin async main tab tak wait krta hai jab tak response nhi aa jata hai.

## async function fetchNormal(){

## const catergories = await fetch(

## "url"

## ).then((response)=>response.json());

## }

## async function fetchNormal(){

## const catergories = await fetch(

## "url"

## ).then((response)=>response.json());

## }

## async function fetchNormal(){

## const catergories = await fetch(

## "url"

## ).then((response)=>response.json());

## }

## return [categoreis,categoreis,categoreis];

## }

## fetchNormal().then((res)=>console.log(res));

## async function fetchConcurrent(){

## const categories = fetch('url').then((res)=>res.json());

## const categories = fetch('url').then((res)=>res.json());

## const categories = fetch('url').then((res)=>res.json());

## return Promise.all([categories,categories,categories]);

## }

## fetchConcurrent().then((data)=>console.log(data));

## <-Closures in JavaScript->

Closure is defined as an inner function that has access to variables

and parameters of an outer function even after the outer function has returned!!

In simple words, closures are Javascript functions combined with its

lexical environment.

let shield = 50;//parent scope

function runtimeTerror(){

return health + shield;//captures from parent scope.

}

The lexical environment is generally an outer function that provides

its local variables and other information to the inner function.

function outerFunction(){

let outerVar = 'I am outside!';

function innerFunction(){

console.log(outerVar);

//I am outside!

}

return innerFunction;

}

function execute(){

const myInnerFunction = outerFunction();

myInnerFunction();

}

execute();

The key to forming a closure is returning a function from another function.

But why use closures in Javascript?

For data privacy and encapsulation

Encapsulation in javascript is like making the properties of an object private.

The simplest and most elegant way to do that is using closures.

## What are the advantages of an arrow function??

How does it differ from one declared with the function keyword?

->A more concise and compact syntax.()=>{} vs function (){}.

Implicit return if the function is one line. You can leave out the function body and JavaScript will automatically return the value;

Const triple = (num)=>3\*num;

Both do the same job;

Const triple = function(num){

Return 3\* num;

}

Important point-> 1-> A key benefit is how arrow functions handle this.

2-> Previously, using function bound the value of this was based on where the function was called. This forced developers to add hacky fixes to retain the original this value.

3-> On the other hand, an arrow function does not create its own this and instead uses the value from its enclosing scope.

Extra Points->

Arrow functions don’t have access to the arguments object.

Arrow functions don’t change this from bind/call/apply since they don’t define their won value for it.

Arrow functions can’t be used as constructors functions with the new keyword.

## Page Refresh-> Window.location.reload();

Location->Represents url, Reload-> Main function.

Another options is using-> document.location.reload();

<button type=”button” onClick=”window.location.reload();” Refreash for update values.</button>

## Scope Differences

"var" - is function scoped, thus it is only accessible within the function it is defined.

"let" and "const" are block scoped, thus accessible within the block in which it's defined.

### Defining Global Variables

"var" can be used to define global variables.

"let" and "const" cannot define global variables.

## Redeclaration:

Variables declared with "var" can be declared again in the scope within which it's defined.

But it is not possible with the case of "let" and "const".

### Hoisting Difference

When variable is declared with "var" (but not initialized) at the end of the function, It is moved to the top of it's scope by the JavaScript runtime and, therefore, there will be no error by the runtime if that variable is used before being declared.

Variables declared with "let" and "const" are only accessible after their declaration.

# Node has several dedicated logging frameworks you can use in your application. Winston, Bunyan, and Pino are a few examples of such tools. You can test and select a one that suits your application’s needs.

# [JavaScript: Difference between .forEach() and .map()](https://stackoverflow.com/questions/34426458/javascript-difference-between-foreach-and-map)

**forEach()->**

1. Performs given operation on each element of the array
2. Returns undefined

**Map()->**

1. creates a new array with the results of calling a provided function on every element in this array
2. Returns new array with transformed elements, leaving back original array unchanged..

typeof null // "object" (not "null" for legacy reasons)

typeof undefined // "undefined"

null === undefined // false

null == undefined // true

null === null // true

null == null // true

!null // true

isNaN(1 + null) // false

isNaN(1 + undefined) // true

1 === 1 //true

1 === "1" //false

false === 0 //false

0 === “” //false

“” === false //false

null === undefined //false

NaN === null //false

NaN === undefined //false

NaN === NaN //false

1 == 1 //true

1 == "1" //true

false == 0 //true

0 == “” //true

“” == false //true

null == undefined //true

NaN == null //false

NaN == undefined //false

NaN == NaN //fals

## [Literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types#literals)

Literals represent values in JavaScript.

### [Converting strings to numbers](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types#converting_strings_to_numbers)

In the case that a value representing a number is in memory as a string, there are methods for conversion.

* [parseInt()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt)
* [parseFloat()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseFloat)

An alternative method of retrieving a number from a string is with the + (unary plus) operator:

'1.1' + '1.1' // '1.11.1'

(+'1.1') + (+'1.1') // 2.2

// Note: the parentheses are added for clarity, not required.

A variable declared using the var or let statement with no assigned value specified has the value of [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined).

The undefined value converts to NaN when used in numeric context.

When you evaluate a [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null) variable, the null value behaves as 0 in numeric contexts and as false in boolean contexts. For example:

**Adding a constant value**

Specify the constant value in the same statement in which it's declared:

**process.nextTick ( ) in Node JS**

process is the core module of Node JS and we don’t have a need to install this module. You can simply access this module by require method.

const process = require('process')

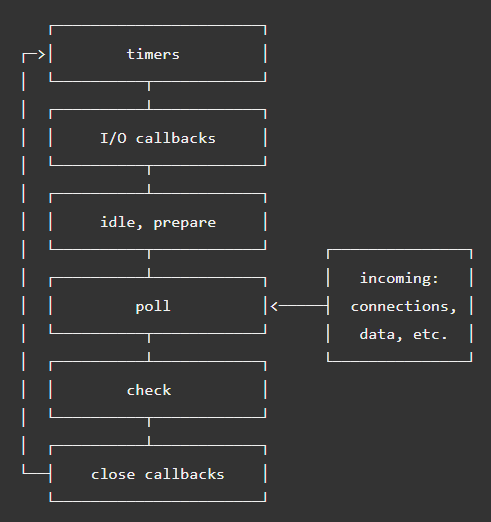
If you don’t want to require process module, you directly use it.

process.env

By the use of an above snippet of code, you access all the environment variables and values of the system in Node JS app. Like process.env, you will access all the methods of process from any of the files in Node JS app without requiring it. process.nextTick ( ) is one of the methods of process module.

What is process.nextTick ( ) ?

When in Node JS, one iteration of the event loop is completed. This is known as a tick. process.nextTick() taking a callback function which is executed after completing the current iteration/tick of the event loop.



In the above diagram, process.nextTick ( ) is not the part of any phase of event loop. Instead, nextTickQueue will process all the callbacks after completing current iteration and before starting the next iteration of event loop.

process.nextTick ( ) schedule a callback function to be executed in next iteration of event loop.

process.nextTick(()=>{

console.log("Executed in next iteration.");

});

console.log('Executed in current iteration.');

The above code of snippet, the second console is printed first because this is a part of the current iteration of the event loop and the first console is a part of a callback function which is associated with process.nextTick() executed in next iteration of event loop.

When we use process.nextTick ( ) ?

To clean up unwanted resources.

To allow users to handle errors.

To try a request to run before starting the next iteration of the event loop.

Allow a callback to run after the call stack and before the next iteration of the event loop.

When a number is added to a string, the number type is always converted to the string type.

**var** x = 3;

**var** y = "3";

x + y // Returns "33"

‘ + ‘ operator when used to add two numbers, outputs a number. The same ‘ + ‘ operator when used to add two strings, outputs the concatenated string.

**var** name = "Vivek";

**var** surname = " Bisht";

name + surname // Returns "Vivek Bisht"

#### Note - 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.

**var** x = 3;

Var y = "3";

x - y //Returns 0 since the variable y (string type) is converted to a number type

Logical operators in javascript, unlike operators in other programming languages, **do not return true or false. They always return one of the operands.**

**OR ( | | ) operator**- If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.

**AND ( && ) operator**- If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

### What is NaN property in JavaScript?

NaN property represents the **“Not-a-Number”**value. It indicates a value that is not a legal number.

**typeof**of NaN will return a **Number**.

To check if a value is NaN, we use the **isNaN()**function,

#### Note- isNaN() function converts the given value to a Number type, and then equates to NaN.

isNaN("Hello") // Returns true

isNaN(345) // Returns false

isNaN('1') // Returns false, since '1' is converted to Number type which results in 0 ( a number)

isNaN(true) // Returns false, since true converted to Number type results in 1 ( a number)

isNaN(false) // Returns false

isNaN(undefined) // Returns true

[**https://arc.dev/developer-blog/javascript-interview-questions/**](https://arc.dev/developer-blog/javascript-interview-questions/)

**https://www.interviewbit.com/javascript-interview-questions/**

Reverse a string

function reverseStr(str) {

var reversestr = '';

for(var i = str.length-1; i >=0; i--) {

reversestr += str[i];

}

return reversestr;

}

Exponent power of a number

// Method 1 : Math.pow(base, exp)

Math.pow(2,3)

//Method 2: By use of for loop

var power = function (base, exp) {

var pow = 1;

for(var i=0; i< exp; i++) {

pow = pow\*base;

}

return pow;

}

//Method 3: Recursive method

var power = function (base, exp) {

if(exp === 0) {

return 1;

} else {

return base\*power(base, --exp)

}

}

[req.params](http://expressjs.com/en/api.html#req.params) contains route parameters (in the path portion of the URL), and [req.query](http://expressjs.com/en/api.html" \l "req.query) contains the URL query parameters (after the ? in the URL).

You can also use [req.param(name)](http://expressjs.com/en/api.html" \l "req.param) to look up a parameter in both places (as well as req.body), but this method is now deprecated.

Route path: /users/:userId/books/:bookId

Request URL: http://localhost:3000/users/34/books/8989

req.params: { "userId": "34", "bookId": "8989" }

app.get('/users/:userId/books/:bookId', (req, res) => {

res.send(req.params)

})

Use the JavaScript function JSON.parse() to convert text into a JavaScript object:

Use the JavaScript function JSON.stringify() to convert it into a string.

**Falsy vs false**

Falsy values are values with an inherent boolean false, the following are falsy values.

**0, '' or "", null, undefined, NaN**

const zero = 0;

const emptyString = "";

if(!zero){

console.log("0 is falsy");

}

if(!emptyString){

console.log("An empty string is falsy")

}

console.log(NaN || 1); // => 1 //value is nothing NaN return a number.

console.log(null || 1); // => 1//null return as object;

console.log(undefined || 1); // => 1// return undefined

**Truthy vs true**

Every value that is not falsy is considered truthy, these include

**strings, arrays, objects, functions**

function somethingIsWrong(){

console.log("Something went horribly wrong")

}

function callback(){

console.log("Hello From Callback");

}

const string = "Hello world!"

const array = [1,2,3];

const object = {};

if(string){

console.log(string) // => "Hello world!"

const functionToCall = callback || somethingIsWrong

functionToCall() // => "Hello From Callback"

console.log(array || "That was not meant to happen")

console.log(object || "This is strange")

}

null >= 0; --> true // bcz null return as obejct

null > 0; --> false

null == 0; --> false

let s = []==0;//true -> bcz array start from 0;

let s = {}==0;//false -> bcz its an object;

0 == false // => true//bcz false value now 0;

0 === false // => false//bcz its also checking type false type is boolean.

[]==[] or []===[] //false, refer different objects in memory

{}=={} or {}==={} //false, refer different objects in memory

**Number + Number -> addition;**

**Boolean + Number -> addition;**

**Boolean + Boolean -> addition;**

**Number + String -> concatenation;**

**String + Boolean -> concatenation;**

**String + String -> concatenation;**

**27.toString() // > Uncaught SyntaxError: Invalid or unexpected token**

**(27).toString(); // -> '27'**

**27..toString(); // -> '27'**

**function a(x) {**

**arguments[0] = "hello";**

**console.log(x);**

**}**

**a(); // > undefined**

**a(0); // > "hello"**

**arguments is an Array-like object that contains the values of the arguments passed to that function. When no arguments are passed, then there's no x to override.**

**{}{}; // -> undefined**

**{}{}{}; // -> undefined**

**{}{}{}{}; // -> undefined**

**{foo: 'bar'}{}; // -> 'bar'**

**{}{foo: 'bar'}; // -> 'bar'**

**{}{foo: 'bar'}{}; // -> 'bar'**

**{a: 'b'}{c:' d'}{}; // -> 'd'**

**{a: 'b', c: 'd'}{}; // > SyntaxError: Unexpected token ':'**

**({}{}); // > SyntaxError: Unexpected token '{'**

**When inspecting each {}, they returns undefined. If you inspect {foo: 'bar'}{}, you will find {foo: 'bar'} is 'bar'.**

**There are two meanings for {}: an object or a block. For example, the {} in () => {} means block. So we need to use () => ({}) to return an object.**

**Let's use {foo: 'bar'} as a block. Write this snippet in your console:**

**if (true) {**

**foo: "bar";**

**} // -> 'bar'**

**Surprisingly, it behaviors the same! You can guess here that {foo: 'bar'}{} is a block.**

**Math.max() less than Math.min()**

**Math.min() > Math.max(); // -> true**

**Math.min() < Math.max(); // -> false**

**Math.min(); // -> Infinity**

**Math.max(); // -> -Infinity**

**Infinity > -Infinity; // -> true**

**Why so? Well, Math.max() is not the same thing as Number.MAX\_VALUE. It does not return the largest possible number.**

**Math.max takes arguments, tries to convert the to numbers, compares each one and then returns the largest remaining. If no arguments are given, the result is −∞. If any value is NaN, the result is NaN.**

**The opposite is happening for Math.min. Math.min returns ∞, if no arguments are given.**

**Chaining assignments on object**

**var foo = { n: 1 };**

**var bar = foo;**

**foo.x = foo = { n: 2 };**

**foo.x; // -> undefined**

**foo; // -> {n: 2}**

**bar; // -> {n: 1, x: {n: 2}}**

**From right to left, {n: 2} is assigned to foo, and the result of this assignment {n: 2} is assigned to foo.x, that's why bar is {n: 1, x: {n: 2}} as bar is a reference to foo. But why foo.x is undefined while bar.x is not ?**

**💡 Explanation:**

**Foo and bar references the same object {n: 1}, and lvalues are resolved before assignations. foo = {n: 2} is creating a new object, and so foo is updated to reference that new object. The trick here is foo in foo.x = ... as a lvalue was resolved beforehand and still reference the old foo = {n: 1} object and update it by adding the x value. After that chain assignments, bar still reference the old foo object, but foo reference the new {n: 2} object, where x is not existing.**

**It's equivalent to:**

**var foo = { n: 1 };**

**var bar = foo;**

**foo = { n: 2 }; // -> {n: 2}**

**bar.x = foo; // -> {n: 1, x: {n: 2}}**

**// bar.x point to the address of the new foo object**

**// it's not equivalent to: bar.x = {n: 2}**

**Tricky return**

**return statement is also tricky. Consider this:**

**(function() {**

**return**

**{**

**b: 10;**

**}**

**})(); // -> undefined**

**return and the returned expression must be in the same line:**

**(function() {**

**return {**

**b: 10**

**};**

**})(); // -> { b: 10 }**

This is because of a concept called Automatic Semicolon Insertion, which automagically inserts semicolons after most newlines. In the first example, there is a semicolon inserted between the return statement and the object literal, so the function returns undefined and the object literal is never evaluated.

**arguments and arrow functions**

Consider the example below:

let f = function() {

return arguments;

};

f("a"); // -> { '0': 'a' }

Now, try do to the same with an arrow function:

let f = () => arguments;

f("a"); // -> Uncaught ReferenceError: arguments is not defined

💡 Explanation:

Arrow functions are a lightweight version of regular functions with a focus on being short and lexical this. At the same time arrow functions do not provide a binding for the arguments object. As a valid alternative use the rest parameters to achieve the same result:

let f = (...args) => args;

f("a");

**Arrow functions can not be a constructor**

Consider the example below:

let f = function() {

this.a = 1;

};

new f(); // -> f { 'a': 1 }

Now, try do to the same with an arrow function:

let f = () => {

this.a = 1;

};

new f(); // -> TypeError: f is not a constructor

💡 Explanation:

Arrow functions cannot be used as constructors and will throw an error when used with new. Because they have a lexical this, and do not have a prototype property, so it would not make much sense.

**Tricky arrow functions**

Consider the example below:

let f = () => 10;

f(); // -> 10

Okay, fine, but what about this:

let f = () => {};

f(); // -> undefined

💡 Explanation:

You might expect {} instead of undefined. This is because the curly braces are part of the syntax of the arrow functions, so f will return undefined. It is however possible to return the {} object directly from an arrow function, by enclosing the return value with brackets.

let f = () => ({});

f(); // -> {}

**Non-coercible objects**

With well-known symbols, there's a way to get rid of type coercion. Take a look:

function nonCoercible(val) {

if (val == null) {

throw TypeError("nonCoercible should not be called with null or undefined");

}

const res = Object(val);

res[Symbol.toPrimitive] = () => {

throw TypeError("Trying to coerce non-coercible object");

};

return res;

}

Now we can use this like this:

// objects

const foo = nonCoercible({ foo: "foo" });

foo \* 10; // -> TypeError: Trying to coerce non-coercible object

foo + "evil"; // -> TypeError: Trying to coerce non-coercible object

// strings

const bar = nonCoercible("bar");

bar + "1"; // -> TypeError: Trying to coerce non-coercible object

bar.toString() + 1; // -> bar1

bar === "bar"; // -> false

bar.toString() === "bar"; // -> true

bar == "bar"; // -> TypeError: Trying to coerce non-coercible object

// numbers

const baz = nonCoercible(1);

baz == 1; // -> TypeError: Trying to coerce non-coercible object

baz === 1; // -> false

baz.valueOf() === 1; // -> true

**Insidious try..catch**

What will this expression return? 2 or 3?

(() => {

try {

return 2;

} finally {

return 3;

}

})();

The answer is 3. Surprised?

**Labels**

Not many programmers know about labels in JavaScript. They are kind of interesting:

foo: {

console.log("first");

break foo;

console.log("second");

}

// > first

// -> undefined

💡 Explanation:

The labeled statement is used with break or continue statements. You can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

In the example above, we identify a label foo. After that console.log('first'); executes and then we interrupt the execution.

**Comparison of three numbers**

1 < 2 < 3; // -> true

3 > 2 > 1; // -> false

Explanation:

Why does this work that way? Well, the problem is in the first part of an expression. Here's how it works:

1 < 2 < 3; // 1 < 2 -> true

true < 3; // true -> 1

1 < 3; // -> true

3 > 2 > 1; // 3 > 2 -> true

true > 1; // true -> 1

1 > 1; // -> false

We can fix this with Greater than or equal operator (>=):

3 > 2 >= 1; // true

**[] and null are objects**

typeof []; // -> 'object'

typeof null; // -> 'object'

// however

null instanceof Object; // false

**NaN is not a number**

Type of NaN is a 'number':

typeof NaN; // -> 'number

**Math with true and false**

true + true; // -> 2

(true + true) \* (true + true) - true; // -> 3

**undefined and Number**

If we don't pass any arguments into the Number constructor, we'll get 0. The value undefined is assigned to formal arguments when there are no actual arguments, so you might expect that Number without arguments takes undefined as a value of its parameter. However, when we pass undefined, we will get NaN.

Number(); // -> 0

Number(undefined); // -> NaN

**Array equality is a monster**

Array equality is a monster in JS, as you can see below:

[] == '' // -> true

[] == 0 // -> true

[''] == '' // -> true

[0] == 0 // -> true

[0] == '' // -> false

[''] == 0 // -> true

[null] == '' // true

[null] == 0 // true

[undefined] == '' // true

[undefined] == 0 // true

[[]] == 0 // true

[[]] == '' // true

[[[[[[]]]]]] == '' // true

[[[[[[]]]]]] == 0 // true

[[[[[[ null ]]]]]] == 0 // true

[[[[[[ null ]]]]]] == '' // true

[[[[[[ undefined ]]]]]] == 0 // true

[[[[[[ undefined ]]]]]] == '' // true

**Trailing commas in array**

You've created an array with 4 empty elements. Despite all, you'll get an array with three elements, because of trailing commas:

let a = [, , ,];

a.length; // -> 3

a.toString(); // -> ',,'

**null is falsy, but not false**

Despite the fact that null is a falsy value, it's not equal to false.

!!null; // -> false

null == false; // -> false

At the same time, other falsy values, like 0 or '' are equal to false.

0 == false; // -> true

"" == false; // -> true

**[] is truthy, but not true**

An array is a truthy value, however, it's not equal to true.

!![] // -> true

[] == true // -> false

**Object.is() and === weird cases**

Object.is() determines if two values have the same value or not. It works similar to the === operator but there are a few weird cases:

Object.is(NaN, NaN); // -> true

NaN === NaN; // -> false

Object.is(-0, 0); // -> false

-0 === 0; // -> true

Object.is(NaN, 0 / 0); // -> true

NaN === 0 / 0; // -> false

In JavaScript lingo, NaN and NaN are the same value but they're not strictly equal. NaN === NaN being false is apparently due to historical reasons so it would probably be better to accept it as it is.

Similarly, -0 and 0 are strictly equal, but they're not the same value.

For more details about NaN === NaN, see the above case.

**NaN is not a NaN**

NaN === NaN; // -> false

The specification strictly defines the logic behind this behavior:

If Type(x) is different from Type(y), return false.

If Type(x) is Number, then

If x is NaN, return false.

If y is NaN, return false.

… … …

**true is false**

!!"false" == !!"true"; // -> true

!!"false" === !!"true"; // -> true

Explanation:

Consider this step-by-step:

// true is 'truthy' and represented by value 1 (number), 'true' in string form is NaN.

true == "true"; // -> false

false == "false"; // -> false

// 'false' is not the empty string, so it's a truthy value

!!"false"; // -> true

!!"true"; // -> true

**baNaNa**

"b" + "a" + +"a" + "a"; // -> 'baNaNa'

This is an old-school joke in JavaScript, but remastered. Here's the original one:

"foo" + +"bar"; // -> 'fooNaN'

**true is not equal ![], but not equal [] too**

Array is not equal true, but not Array is not equal true too; Array is equal false, not Array is equal false too:

true == []; // -> false

true == ![]; // -> false

false == []; // -> true

false == ![]; // -> true

Explanation:

true == []; // -> false

true == ![]; // -> false

// According to the specification

true == []; // -> false

toNumber(true); // -> 1

toNumber([]); // -> 0

1 == 0; // -> false

true == ![]; // -> false

![]; // -> false

true == false; // -> false

false == []; // -> true

false == ![]; // -> true

// According to the specification

false == []; // -> true

toNumber(false); // -> 0

toNumber([]); // -> 0

0 == 0; // -> true

false == ![]; // -> true

![]; // -> false

false == false; // -> true

**[] is equal ![]**

[] == ![]; // -> true

+[] == +![];

0 == +false;

0 == 0;

true;

<https://github.com/denysdovhan/wtfjs>