ADVANCE JAVASCRIPT NOTES

By: Deepa Chaurasia

https://www.linkedin.com/in/deepa-chaurasia-3704351a8

Table of Contents

How Var, let and const keywords works in Javascript

Earlier, pre-ES6 era, only var keyword was introduced for declaration of variable

With ES6, the let and const keyword introduced.

How to declare Variables in JavaScript

```
// without keywords. It is same as var
   and not allowed in 'strict' mode.

name = 'Jack';

// using var
var price = 100;

// using let
let isPermanent = False;

// using const
const PUBLICATION = 'Jack';
```

Date: / /

We'll disuss
- Scope
- Reassigning New Value
- When you access a variable before declaring it.

Variable Scope in JavaScript

The variable may exist in a block, inside function or outside function.

A block is section of code inside { }

Eg →
```
{
    let name = 'deepa';
}
```

* It has block Scope

A function is bunch of code you want to place logicall together.

It is declared using Function keyword

```
function test() {
    let name = 'deep';
}
```

* It has function scope
* Everything declared outside block and function is global Scope

So there are three types of Scope

- Block Scope
- Function Scope
- Global Scope

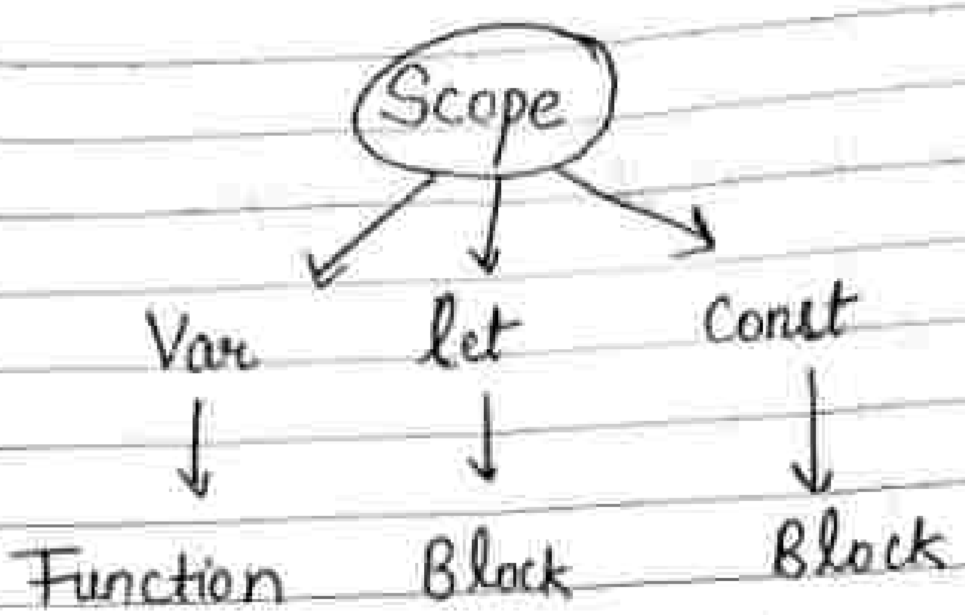
The three keyword var, let and const work around these scopes.

How to Use Javascript Variable in Global Scope

We can use var, let and const to declare global variable.
But it is recommended not to do it.
By doing this, variable are accessible everywhere.

So to restrict scope of variable using var, let and const keywords, here's order of accessiblity in scope starting with lowest:

- var : The functional Scope level
- let : The block Scope level
- const : The block Scope level



How to Reassign a New value to Variable in Javascript

You can reassign var or let variables, but you cannot reassign a new value to const variable.

Const — (Constant) — ~~Always same.~~ ~~Cannot change~~

One Tricky part
When object is declared and assigned value with const, you CAN STILL CHANGE VALUE OF ITS PROPERTIES

But you cannot reassign any object value to same variable

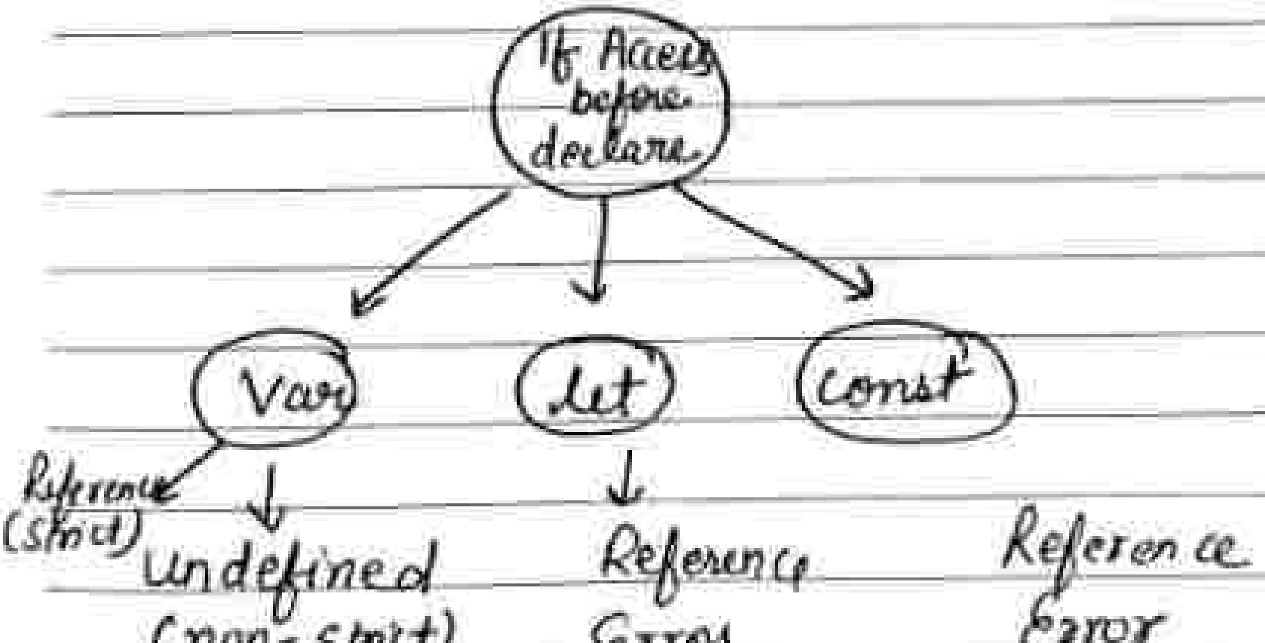
When You Access a Variable before declaring

With var in non-strict mode, the variable will have an undefined value

This means variable declared but not assigned

In strict mode, you will get Reference Error that variable is not declared

With let and const, you will always get Reference Error



* Don't use Var
* Use let or const
* Use const more often.
* Use let, when you need to reassign

Hoisting In JavaScript

Hoisting simply gives higher specificity to javascript declarations. Thus, it makes the computer read and process declarations first before analyzing other code in program

Note → Hosting does not mean Javascript rearranges or move code above one another

```
    Console.log (name) // Uncaught Reference
                                  Error
Eg → let name = 'Deepa';
```

Variables declared with let and const are hoisted but not initialized with a default value.

Accessing let or const before it's ~~dclo~~ declared will give :-

Uncaught Reference Error: cannot access before initialization

Remember the Error message tells variable is initialized somewhere

Variable hoisting with var

When interpreter hoists a variable declared with var, it initialized its value to undefined, unlike let or const.

Eg →
```
Console.log ( name ); // undefined
var name = 'deepa';
console.log ( name ); // 'deepa'
```

Now let's analyze this behaviour:

```
var name ;
console.log (name); //undefined
name = 'deepa';
console.log (name); // deepa
```

Remember, the first console.log (name) outputs undefined becoz name is hoisted, and given a default value (not becoz variable is never declared).

Using undeclared variable will throw Reference Error

```
console.log (name); // Uncaught Reference Error
                    : ~~fo~~ name is not defined
```

Now let's see if we donot declare var what happens

```
console.log (name);   // Uncaught Referenceerro
  name = 'deepa';
```

↓

Assigning a variable that's not declared is valid

Javascript let us access variable before they're declared. This behaviour is an unusual part of javascript and can lead to errors.

Using variable before it's declaration is not desirable

The Temporal Dead Zone

The reason why we get reference Error when we try to access let or const before its declaration is <u>Temperal Dead zone</u>.

The TDZ starts at begining of the variables enclosing sope and ends when it is declared.

Accessing variable in TDZ gives Reference Error.

Eg

```
{   // start of foo's TDZ
      let bar = 'bar'
      console.log(bar); //'bar'

      console.log(foo); // Reference Error
                           becoz we're in TDZ
      let foo = 'foo';
}.  // End of foo's TDZ.
```

type of TDZ for let or const → Reference Error
for var ⇒ undefined

functional Hoisting

Function declarations are hoisted too. Function hoisting allows us to call function before it is declared or defined.

```
foo();              // 'Foo'

Function foo() {
    console.log('foo');
}
```

Note only function declaration are hoisted not function Expressions.

Eg

```
foo(); // Uncaught Type Error:
var foo = function() {  }
```

Uncaught Type Error: foo is not a function

```
bar(); // Uncaught Type Error
let bar = function() {  }
```

Uncaught Reference Error: cannot access 'bar' before initialization

Similarly for const.

For function that is never declared:

```
foo(); // Uncaught Reference Error:
       foo is not defined
```

Closures in JavaScript

```
function sayWord (word) {
   return () => consol.log(word);
}
const sayHello = SayWord ("hello");

SayHello(); // "hello"
```

There's 2 interesting point to notice:-

→ The returned function from sayWord can access the word parameter

→ The returned function maintain the value of word when SayHello is called outside scope of word.

The first point can be explained by Lexical Scope. :

lexical Scope - The returned function can access word before it exists in its outer scope

The second point becoz of Closures

A closure is a function combined with references to variables define outside of it.

Closure maintain the variable references, which allow function to access variables outside of their scope.

They "enclose" the function and variable in its environment

Example of Closures In JavaScript

Callbacks - It is common for a callback to reference a variable declared outside of itself.

eg →
```
function getCars By Make (make) {
    return cars.filter(x => x.make == make)
}
```

make is available in caleback because of lexical Scoping and make is persisted when filter caleed becoz of closure.

Storing state → We can use closures from functions that store states

Let's say a fn which returns an object that can store and change name.

~~fo~~
```
function makePerson (name) {
      let _name = name;

   return {
        setName : (newName) => (_name = newName)
        getName : () => _Name ,
     };
}
 Const me = make Person ("deepa");
 Consoli.log (me.getName()); //"deepa"

 me.setName ("Deepa Chaurasia");
 console.log (me.get Name());
                  // "Deepa Chaurasia");
```

It shows how closure do not freeze values of variables from function's outer scope during creation. Insted they maintain the references throughout closure's lifetime.

Private methods

So In oops concept, ~~we~~ In a class we have private state and expase getter and setter methods public.

We can extend this oops.

```
function makePerson(name) {
    let _name = name;

    function PrivateSetName(newName){
        _name = newName;
    }

    return {
        setName: (newName) => PrivateSetName
                                (newName);
        getName: () => _name,
    };
}
```

PrivateSetName is not directly accessible to consumers and it can access the private state variable _name through closure

Closures make it possible for:

functions to maintain connections with outer variables , even outside scope of the variables

(like LinkedIn maybe :))

There are many uses of closures from creating class like structures that store state and implement private methods to passing callback to event handlers.

Object and it's Method in JavaScript

How to Create objects in JavaScript?

```
const person = {
        name : 'Deepa'
};
```

This is simple and most popular way

* you can also use new keyword

```
const person = new Object();
  person.name = 'Deepa';
```

* you can also use 'new' with user defined constructor function

Eg →
```
function Person (name) {
    this.name = name;
}
```

Now anytime you want Person object

```
const personOne = new Person ('deepa');
```

* Using Object.create() to create new Objects

The Object.create() methods creates a new object, using an existing object as prototype of the newly created Object

It contains 2 parameter:

- First parameter is mandatory that serves prototype of new object to be created

- Second is optional, it contains properties to be added to new object

Eg →
```
const orgObject = { company: 'ABC'};

Const employee = Object.create (orgObject,
                   { name: { value: 'EmpOne'}});

console.log (employee); // { company: 'ABC'}
console.log (employee.name); // 'EmpOne'
```

Date: / /

* Using Object.assign() to create new obj.

The Object.assign () method is used to copy all enumerable own properties value from one or more sources objects to target object.

It will return target object.

Eg → const orgObject = { company: 'ABC' }
const carObject = { carName: 'Ford' }

const employee = Object.assign({ }, orgObject, carObject);

Now you can get employee object that has company and carName as its property

```
console.log ( employee);
// { carName : 'Ford', company :'ABC' }
```

* Using Object.defineProperties ()

This method defines new or modify existing property on object

```
const object1 = {};

Object. define Properties ( object1 , {
        property1 : {
            value : 42,
        },
    });

    Console.log (object1. property1) ; // 42
```

Similarly we also have Object. defineProperty()

* Using Object. Entries()

It returns an array of Object's key value pairs.

The order of array is same as provided by a For ... in loop

```
Const Object1 = {   a : 'something'
                    b: 'nothing'
                };
```

```
for (const [key, value] of Object.entries(object))
  { console.log (`${key}: ${Value}`);
  }

  // "a": something"
     "b : nothing"
```

Object. freeze()

It freezes an object
No longer can be changed

Eg.—
```
const obj = {
        prop: 42
     };
   Object.freeze (obj);
   ~~console.log (obj~~

   obj. prop = 43;
   console.log (obj. prop);

      // output 42
```
It can no longer be change due to freeze

Date: / /

Object. fromEntries()

It transforms a list of key-value pairs into an object

Eg →

```
   Const entries = new Map([
         [ 'foo', 'bar' ],
         [ 'baz', 42]
   ]);

Const obj = Object.fromEntries(entries);
Console.log (obj);

// Object { foo: "bar", baz : 42 }
```

* Object. hasOwn ()

- This method returns true if the specified object has indicated property as its own property.

Eg →
```
const object 1 = {
      prop : 'exists'
  };

Console. log (Object. hasOwn (object 1, 'prop'));
        // true

Console. log (Object. hasOwn (object1, 'prop2'));
        // False
```

Imp

* object. hasOwnProperty ()

It returns boolean value, whether specified object has the property mentioned as its own

Eg →
```
const object1 = { };
object1. property1 = 42;
```

```
console.log (object1.hasOwnProperty('property1'));
            // true
```

* Object.is() (can be use to compare)

It determines whether two values are same.

```
eg → Object.is (25, 25);          // true
     Object.is ('foo', 'bar');    // false

     Object.is (NaN, Number.NaN)
                         // true
     Object.is (NaN, 0/0);        // true
```

* Object.isExtensible

determines whether new properties can be added or not

```
Eg → Const object1 = {};
console.log (Object.isExtensible(object1));
        // true
Object.preventExtensions(object1);
console.log (Object.isExtensible(object1));
        // False
```

Date: / /

*Object. isfrozen()

determines if an object is frozen

If frozen → you cannot make any change

* Object. isSealed( )

determines if object is sealed

Imp

*Object . Seal( )

It seals an object, prevent new properties from being added to it and ~~not~~ making all existing properties non-configurable

Imp

* Object. keys( )

returns an array of all keys present in given object

Imp

* Object. values( )

returns an array of all values present in given object

Callback Functions

A Callback function is function that is performed after another function has completed its execution

It is typically supplied as an input to other function.

Callbacks are critical to understand, as they are used in array methods (such as map(), filter(), and so on), setTimeout, eventlistners (such as click, scroll)

Eg→
```
Function orderPizza (type, name, callback){
    console.log('Pizza ordered..');
    console.log ('Pizza is on preparation');
   setTimeout (function(){
  let msg = `Your ${type} ${name}
            Pizza is ready`;
   callback (msg);
  }, 3000);
}
```

Date: / /

Now Invocation of Order Pizza

```
orderPizza ('veg', 'cheese', function(message)
  {
   console. log(message);
  });
```

· Imp points to Note

→ Javascript fn can accept other fn as arg.

→ passing fn as arguement is powerful programming concept that can be used to notify caller that something happened.
It is also known as callback function.

→ Nesting too many callback fn is not a great idea and it creates Callback hell.

JavaScript Map

The Array. Map() allows you to iterate over array using loop.

This method allows you to iterate and modify its elements using a callback function.

The callback function will then be executed on each of array's element.

For. eg let arr = [2, 3, 4, 5, 6];

Now Imagine you have to multiply each element of array by 3

you can use for loop also like this

```
let arr = [2, 3, 4, 5, 6];
for (let i = 0; i<arr.length; i++){
        arr[i] = arr[i] * 3; }
console.log(arr);

// [6, 9, 12, 15, 18]
```

By using map it will look like this:

```
let arr = [2,3,4, 5,6];

let modified Arr = arr.map (function(el){
            return el*3;
   });
  console.log (modified Arr);
            // [6, 9, 12, 15, 18]
```

How to Use Map over ARRAY of OBJECT

```
let users = [
  {firstName: 'Deepa', lastName: 'chaurasia},
  {firstName: 'Devesh', lastName: 'chaurasia'},
 {first Nam: 'Jyoti', lastName: 'chaurasia},
];
```

You can iterate as follow

```
let userFullnames = users.map(function(el){
    return `${el.firstName} ${el.lastName}`;
  })
     console.log (userFullnames);
// ['Deepa chaurasia', 'Devesh chaurasia', 'Jyoti Chaura
```

The Complete map() method syntax

The syntax of map() as follows

```
arr.map(function(element, index, array{ },
                                   this);
```

The callback function() is called on each array element, and the map() method always passes the current element, the index of current element and whole array object to it.

The this argument will be used inside callback function

By default it's value is undefined.

Eg—
```
let arr = [2, 3, 5, 7]
  arr.map(function(element, index, array){
      console.log(this) // 80
  }, 80);
```

Here you can see this value is 80 which is default value.

Reduce Method In JavaScript

Use it When: you have array of numbers, you want to add them all.

For eg —
```
const nos = [29, 40, 30];
const sum = nos.reduce((total,amount)
        => total+amount);
    sum // 99
```

Filter() and Find() in JS

Fiter () provides new array depending on certain criteria.

Unlike map(), it can alter size of new Array, whereas find() return just a single instance.

Foreg →
```
let users = [
    {firstName : 'Ram' age : 14},
    {firstName : 'shyam' age : 17},
    {first Name : 'Jacob' age : 25}
];
```

You could choose to sort these data by age groups, such as young (1-15) Adult (15-50)

Like this :

```
const young People = users. Filter ((person)=>{
        return person.age <=15;
  });

const adult = users. Filter ((person) => {
      return person. age >= 50});

Console. log (young People);
console.log. (adult);
```

And The Example of Find goes like this

```
const Ram = users. find ((person ) =>
        person. firstName = = = 'Ram');

Console .log (Ram);
```

Unique Value - Set() in JS

```
let animals = [
  {
     name : 'Lion',
     category : 'wild'
  },
  {  name : 'dog'
     Category : 'pet'
  },
   {  name : 'cat'
      category : 'pet'
   }.
]
```

If we loop through map, we will get repeated value.

But we don't want repeated value here.

So we will use Unique value - Set()

For eg -
```
let category = [...new Set(
                animals.map((animal) =>
                    animal.category))];
  console.log (category); // [wild, pet]
```

Date: / /

What is Destructuring in JavaScripts

Destructuring is act of unpacking elements in an array or object.

It not only allow you to unpack but also manipulate and switch elements acc to yourd demand

Destructuring in Arrays

Eg.

Rest operator

```
* Const [ var1, var2, ...Var3] =
                    [ "Deepa", "Jyoti", "Devesh",
                                      "Ram" ]

  Console.log (var1); // 'Deepa'
  Console.log (var2); // 'Jyoti'
  Console.log (var3);   [ 'Devesh', 'Ram' ]
```

Javascript allows you to use rest operator with an destrucling array to assign the rest of regular array to variable

As you have noticed ["Devesh", "Ram"] remaining both get stored in Var3

Eg

```
* const [ , , website ] = ['Google', 'yahoo', 'firefox'];
console.log (website) ; // 'firefox'
```

Here we use ',' to skip variables at destructing array's first and second index positions

How Default value work in an Array Destructing Assignment

Eg

```
const [ firstName = 'Deepa', lastName = 'chaurasia']
        = ["Deepa Chaurasia"];

console.log ( firstName)  // Deepa Chaurasia
console.log (lastName) // chaurasia
```

Here 'Deepa' and 'chaurasia' are default value of 'firstName' and 'lastName' variables.

∴ In our attempt to extract first index value from right side of array, the computer defaulted to "chaurasia" – becoz. Only zeroth index value exists in ["Deepa Chaurasia"]

Date / /

Object destructing In JavaScript

Object destructing is unique technique that allows you to neatly extract an object's value to new variables

```
Const profile = {
      firstName : 'Deepa";
  };
```

Destructing object

```
Const { firstName : firstName } = profile ;
```

This key references the profile object's firstName key

This value represents your new variable

The destructing object's key references its profile objects property name.

And destructing object's value represents your new variable.

Date / /

```
Eg → const profile {
        firstName : 'Deepa',
        lastName : 'Chaurasia',
     };

const { firstName, lastName } = profile;

 console.log (firstName); // 'Deepa'
 console.log (lastName); // 'Chaurasia'
```

How to Use object Destructing to Swap Value

```
let firstName = 'Deepa';
let lastName = 'chaurasia';

({ firstName , lastName } = { firstName : lastName ,
                              lastName : firstName });

console.log (firstName); // 'chaurasia'
console.log (lastName); // 'Deepa'
```

Synchronous Vs Asynchronous

Synchronous : Every statement of code get executed one by one

So basically, a statement has to wait for earlier statement to get executed

Eg -
```
console.log ("I");

console.log("eat");

console.log ("ice-cream");
```

It will point I first,
then eat,
after that ice-cream

Asynchronous: It allows program to be executed immediately without blocking the code. Unlike the Synchronous method It doesn't wait for earlier statement to get executed first.

Each task execute completed Independently.

Eg-

```
Console.log ("I");

    setTimeout (() => {
        console.log ("eat"); }, 2000)

    Console.log ("Ice Cream")
```

It will print

```
"I"
"Ice Cream" (will execute immediately)
"eat" (will print after 2s)
```

Asynchronous Functions

→ It contains <u>async</u> keyword.

How to use in Normal Function declaration

```
async function name ( arg ) {
    }
```

How to use in an arrow function

```
Cons FunctionName = async ( arg ) => {
    }
```

Asynchronous functions always return promises

It doesn't matter what you return. The returned value will always be promise

Eg →

```
const getOne = async () => {
    return 1;
}

const promise = getOne();
console.log (promise).
```

The await keyword

The await keyword lets you wait for promise to resolve. Once promise is resolved it returns the parameter passed into then call.

Eg- ~~con~~

Eg→

```
Const getOne = async _ => {
    return 1; }

getOne (). then (value => {
                console.log (value.) }); //1
```

Now use of await keyword

```
const test = async _ => {
    const one = await getOne ();
      Console.log (one);
}

test ()
```

We can only use await when we have async.

Let's implement the fetch API code using asynct/await :

```
Const FetchData = async () => {
Const qoutes = await Fetch ("http://- -/quotes");
Const response = await qoutes.json();

Console.log(response);

}
   FetchData();
```

We can also handle errors in async/await by using try and Catch.

```
const FetchData = async () => {
try {
   const quotes = await Fetch ("http://-- ");
   const response = await quotes.json();
     console.log (response);
 }
    catch (error) {
         Console.log(error);
    }

};
   FetchData();
```

Date: / /

Promises In Javascript

A promise is a javascript object that allows you to make asynchronous calls. It produces a value when async operation completes successfully or produces an error if it doesn't complete.

You can create promise using constructor

```
let promise = new Promise (Function(resolve, reject)
{
    });
```

↑ Executor function

Executor fn takes 2 arguements :-

→ resolve — indicate successful completion
→ reject — indicates an error

The Promise objects and states

The promise object should be capable of informing consumers when execution has been started, completed or returned with an error

1. State → pending - when execution fn starts

Fulfilled - when promise resolved successfully

rejected — when the promise rejects

2. result →

undefined - Initially when state value is pending

value -

when promise is resolved

Error

when the promise is rejected

A promise that is either resolved or rejected are settled

Handling Promises by Consumer

Three important handler methods

- then()
- catch()
- Finally

These methods helps us create a link between executor and consumer

The .then() Promise Handler

It is used to let consumer know outcome of promise. It accept 2 arguements
- result
- error

Eg-
```
promise.then (
   (result) => {
         console.log (result);
   },
  (error) => {
         console.log(error);
  }
);
```

The catch Promise Handler

To handle Error (rejections) from Promises. It's better syntax to handle Error than handling it with .then()

Eg →
```
promise.catch (function (Error) {
        console.log (Error);
  });
```

The finally () Promise Handler

The finally () handler method performs cleanups like stopping a loader, closing a live connection and so on

Irrespective of whethere promise resolve or rejects, the finally () method will run

Eg - promise. finally (() => {

```
        console.log ("Promise settled");
}).then ((result) => {
                console.log ({result});
});
```

Imp point to note,
the finally () method passes through result or error to the next handler which can call a .then () or .catch() again.

Date: / /

Why need async/await over promise

The purpose of async/await functions is to simplify behaviour of promises synchronously and perform task on group of promise

Async

putting keyword async before a function tells the function to return a promise

Await

It simply makes javascript wait until the promise settles and then go to result

Meanwhile, engine carries other tasks

A promise which will be resolved with value returned by async function or
If rejected, uncaught exception thrown from async

Date: / /

Why async/await ?

1. Error Handling

Using try/catch makes it easy to handle both synchronous and asynchronous erros

Eg ⇒
```
const makeRequest =() => {
 try {
   getJSon ().
    .then (result => {.
      // this may fail.
    const data = JSON. parse. (result)
    console. log(data)
  })

 catch (err) {
      console.log (err)
  }
```

We can make it better by using.
asyn/await

```
Const makeRequest = asyn () => {

  try {
    // this may fail
  const data = JSON.parse (await
                        getJSON())
   console.log (data)
 }
   catch (err) {
    console.log (err);
   }
}
```

2) Concise and Clean

We don't have to write .then, avoided nesting our code.

3) Conditionals

Eg→

```
const makeRequest = () => {

  return getJSON ()
   .then (data => {
     If (data. needs Another Request) {
       return makAnother Request(data)
        .then ( more Data => {
        console. log (moreData)
        return moreData
   })
 }    else {
         console.log(data)
          return data
      }
 })
}
```

It's easy to get lost in all nesting of 6(levels) braces, return statements that are only needed to propagate final result to main promise.

Asyn / Await provides us option to make it more readable

Date: / /

```
const makeRequest = async() => {
  const data = await getJSON()
  If (data.needAnotherResponse){
   const moreData =~~await makeRequest~~
            =await makeAnotherRequest(data)
   console.log (moreData)
     return moreData
}
   else console.log(data)
          return data
   }
}
```

4) Error Stacks

The error stack returned from promise chain gave no clue where the error happened.

However the error stack from async/await points to function that contains error

5) Debugging

A killer advantage when using async/await is that it's much easier to Debug.

Debugging promises has always been Such a pain for 2 reasons

1) You can't set breakpoints in arrow functions that return expressions

2) If you set a breakpoint inside .then block. and use debug shortcuts, debugger won't move to following. then() because it only steps through synchronous code.

With async/await you don't use arrow functions as much, You can easily debug and step through await calls.

Date: / /

Fetch API - How to Make a GET Request and Post Request in JavaScript.

What is the Fetch API?

Fetch() is mechanism that lets you make simple AJAX (Asynchronous Javascript and XML) calls with Javascript.

Asynchronous means that you can use fetch to make a call to an external API without halting execution of other instructions.

That way other functions will continue to run when an API call has not been resolved.

Date: / /

when response (data) is sent back from API, the async tasks resume.

It is imp to note, that fetch is not part of JS, but WWWTAG.

∴ you have to install special module to use in Node js Environment

How to use fetch () in JS

When we talk about APIs, we also need to talk about endpoins.

An endpoint is simply a unique URL you call to interact with other system

Let's assume that we are making a request to an external API to get some data.

For this we'll use simple GET Request.

Simply call fetch() with endpoint URL as argument

fetch ('https://deepachaurasio.tech /posts/1'),

Trying to fetch blogpost from external API

The response body for this endpoint

```
{
   userId : 1,
   id : 1,
   title : 'A post by deepa',
   body : 'Brilliant post',
 };
```

Ultimately you want response body.

But response body contains quite a bit information including status code, header info etc

Remember Fetch API returns a promise you need to nest a then() method to handle resolution.

Date: / /

The data return from API is not usually useable form.

So you'll need to convert data to a form which your javascript can operate.

You can use JSON()method for that

```
Fetch('https://deepachaurasia.tech/posts/1')
 .then ( data => {
  return data.json ();
})
 .then (post => {
  Console.log ( post.title );
 })
```

Note - To make simple GET Request with fetch, you need to pass only URL point as argument.

Date: / /

How to make post Request

For post, you'll need to pass an object of configuration options as a second argument.

The optional object can take a lot of different parameters.

Becoz you're sending POST request, you have to declare that you're using POST method.

You'll also need to pass some data to actuall create new blog post.

Since you're sending JSON data, you have to

Set a header of Content-Type set to application/json.

Finally you'll need body, which will be single string of JSON data.

Date: / /

```
const update = {

title : 'A blog post by deepa',
body : 'Brilliant post',
userId : 1,
};

const options = {
method : 'POST'
headers :
  {
    'Content-Type' : 'application/json',
  },

body : JSON.stringify (update),
};
```

And then API call

Note → To make a post request you'll need to pass along certain other parameters including configuration object.

Date: / /

```
Fetch ('https://jsonplaceholder.typicode.com/
                                posts', options)
 .then (data => {
     If ( !data.ok) {
       throw Error (data.status);
   }
     return data.json();
 }). then (update => {
     Console.log (update);
   };

  }). catch (e => {
      Console. log (e);
   });
```

If your request is successful, you'll get a response body containing blog post object along with new ID.

The response will vary depending on how API is set up.

How to handle Errors in fetch

The fetch() function will automatically thrown an error for network errors but not for HTTP Errors Such as 400 to 5xx responses

The good news is fetch provides a simple response.ok flag that indicates whether the request failed or an HTTP request

Response status code is in successful range.

This is very simple to implement:

```
fetch ("https://type.fit/api/quotes")
  .then ((response) => {
    If ( ! respons.ok)
      throw Error (response.statusText);
  }
    return response.json();
})
```

Date: / /

// Now interesting part

```
.then ((data) ⇒ console.log (data))
.catch((error) → console.log (error))
```

Thankyou
For
Reading

Like,Share and Comment

https://www.linkedin.com/in/deepa-chaurasia-3704351a8