**Asynchronous JavaScript**

**Promise**: promise is an object that may produce a single value sometime in the future either a resolved value, or a reason that it’s not resolved ( happens in the back ground )

**Chaining**

promise

.then(result => result + "!!!!")

.then(result2=> console.log(result2));

Three states

const promise = new Promise((resolve,reject)=>{

if(true)

resolve('promise worked');

else

reject('promise error');

})

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

**Fulfilled**

**Rejected**

**pending**

**Catch error**

promise

.then(result => { **throw error;**

result + "!!!!"

})

.then(result2=> console.log(result2)) .**catch(()=>console.log('error'));**

***P***romise.all

const promise = new Promise((resolve,reject)=>{

if(true)

resolve('promise worked');

else

reject('promise error');

})

const promise2 = new Promise((resolve,reject)=>{

**setTimeout(resolve,100,'HI');**

})

const promise3 = new Promise((resolve,reject)=>{

setTimeout(resolve,1000,'comming');

})

const promise4 = new Promise((resolve,reject)=>{

setTimeout(resolve,5000,'bye');

})

**Promise.all([promise,promise2,promise3,promise4])**

**.then(values => console.log(values))**

**Realtime request**

const urls = [

'https://jsonplaceholder.typicode.com/posts',

'https://jsonplaceholder.typicode.com/users',

'https://jsonplaceholder.typicode.com/todos',

]

**Promise.all(urls.map(url=>{**

**return fetch(url).then(resp => resp.json())**

**})).**then(results =>{

console.log(results[0])

console.log(results[1])

console.log(results[2])

}).catch(()=> console.log('error'));

**Async Await:** make async code look synchronous

You can use **await keyword in front of anything that returns a promise**

**async function fetchUsers() {** // async keyword and codes within function

**const resp = await fetch('https://jsonplaceholder.typicode.com/users')**

**const data = await resp.json();** // can pass response into variables

console.log(data); // can add await infront of anything that returns promise

**}**

**Using promise**

Promise.all(urls.map(url=>{

return fetch(url).then(resp => resp.json())

})).then(array =>{

console.log('users',array[0])

console.log('posts',array[1])

console.log('albums',array[2])

}).catch('oops')

**Using async await**

const getData = async function(){

try{

const [users,posts,albums]=await Promise.all(urls.map(url=>{

return fetch(url).then(resp => resp.json())

}))

console.log('users',users)

console.log('posts',posts)

console.log('albums',albums)

} catch(err){

console.log('oops',err);

}

}

const urls = [

'https://jsonplaceholder.typicode.com/posts',

'https://jsonplaceholder.typicode.com/users',

'https://jsonplaceholder.typicode.com/todos',

]

**ES9 (2018)**

**Object spread operator**

const animals = {

tiger:23,

lion: 5,

monkey: 2,

bird:40,

}

function objectSpread(p1,p2,p3){

console.log(p1)

console.log(p2)

console.log(p3)

}

const {tiger, lion. . . rest} = animals;

**objectSpread(tiger,lion,rest);**

**Array spread operator // ES6**

const array = [1,2,3,4,5];

function sum(a,b,c,d,e){

return a+b+c+d+e;

}

sum(...array);

**Object spread operator**

const animals = {

tiger:23,

lion: 5,

monkey: 2,

}

**const {tiger, . . . rest} = animals;**

**ES9 (2018) Async**

finally in promise

finally allows you to run code even if then and catch works or not. It doesnot take parameter

.then(stmt)

.catch(stmt)

.**finally(()=> console.log(‘extra info’));**

for await of /iterate over multiple await promises

const getdata=async function(){ //urls 🡪 refer promise example above

**const arrayOfPromises = urls.map(url=>fetch(url));**

**for await(let request of arrayOfPromises){**

const data = await request.json();

console.log(data);

} }

**Job Queue:**

**//callback queue or task queue**

// asynchronous ()- handled by webapi

setTimeout(()=>{console.log('1', 'is the loneliest number')}, 0)

setTimeout(()=>{console.log('2', 'can be as bad as one')}, 10)

**//Job Queue - microtask queue // higher priority**

//asnychronous ()- handled by JS

// so it needs a seperate queue

// event loop checks job queue first then the callback queue

Promise.resolve('hi').then((data)=> console.log('2', data))

**Parallel sequence and race:**

call all the types of promises

sequence().then(console.log)

parallel().then(console.log)

race().then(console.log)

// then is executed only after getting the response. Which ever gets first response executes first

here ->race->parallel ->sequence

**create multiple promises**

const promisify = (item, delay) =>

new Promise((resolve) =>

setTimeout(() => resolve(item), delay));

const a = () => promisify('a', 100);

const b = () => promisify('b', 5000);

const c = () => promisify('c', 3000);

// sequence promise in execution

async function sequence() {

const output1 = await a();

const output2 = await b();

const output3 = await c();

return `sequence is done ${output1} ${output2} ${output3}`

}

// parallel execution of promise

async function parallel() {

const promises = [a(), b(), c()];

const [output1, output2, output3] = await Promise.all(promises);

return `prallel is done: ${output1} ${output2} ${output3}`

}

// race – which returns first and rest omitted

async function race() {

const promises = [a(), b(), c()];

const output1 = await Promise.race(promises);

return `race is done: ${output1}`;

}

**Threads, Concurrency and Parallelism:**

Tasks are executed in threads outside js -Browser creates one thread per tab where js has memory heap ,callstack etc

Browser has – **web wokers** – work in the background like node.js having worker threads for multiple jobs

Var worker = new Worker(‘worker.js) // Worker is a window obj having access to js thread worker.js

Web worker is a js program running on different thread along the main thread . web worker communicate through messages

Fetch() -> browser takes care of

Concurrency : single core cpu - > thread 1 then thread2 (when thread 1 is not being executed like wise) – in js – nodejs

Concurrentcy + parallelism : multicore cpu -> execute threads side by side -> not build in js -> but node allows (very advanced)