Season 2

Topics covered

- Handling Asynchronous Oper...
- Callback Function
  - Callback Function
  - Issues of Callback Function
  - Callback Hell
  - Inversion of Control

- Promises
  - In Promise also we are atta...
  - How Promise is better than...
    - Inversion of control
    - Callback Hell
  - Promise Object
  - Promise State
  - Promise Result
  - Promise with Callback Fun...
  - Immutable Promise Object
  - Solving Callback Hell
- Promise Chaining
- Interview Questions on Pro...
  - What is Promise?
  - Why Promise
- Creating a Promise & Error H...
  - Creating a Promise
  - Error Handling
  - Promise Chaining

- Async await
  - What is async?
  - Difference between normal...
  - How to get an actual value...
  - How to create a promise?
  - How to return a promise fr...
  - Using await with async
  - Handling Promises without...
  - Handling Promises with as...
  - How to handle Promises in...
  - Why do we need async aw...
  - Handling Promise in norma...
- Handling Promise usin... ×
- Scenario 2 - with 2 promises
  - Resolving same Promise...
  - Resolving two different ...
  - Scenario 3 - with change...
- What happens behind the s...
- Real Time example of Asyn...
- Error Handling - Try & Catch
- Interview Tips
- Async/Await vs Promise.th...

Promises
Async/Await

Handling Asynchronous Operations in Javascript

- Callback Functions
- Promises
- Async/Await

Callback Function

Callback Function

Passing a function as callback to another function.
createOrder API will create the order and call proceedToPayment api. So its createOrder API's responsibility to call proceedToPayment API. That way we are handling async operations in Javascript using callback functions

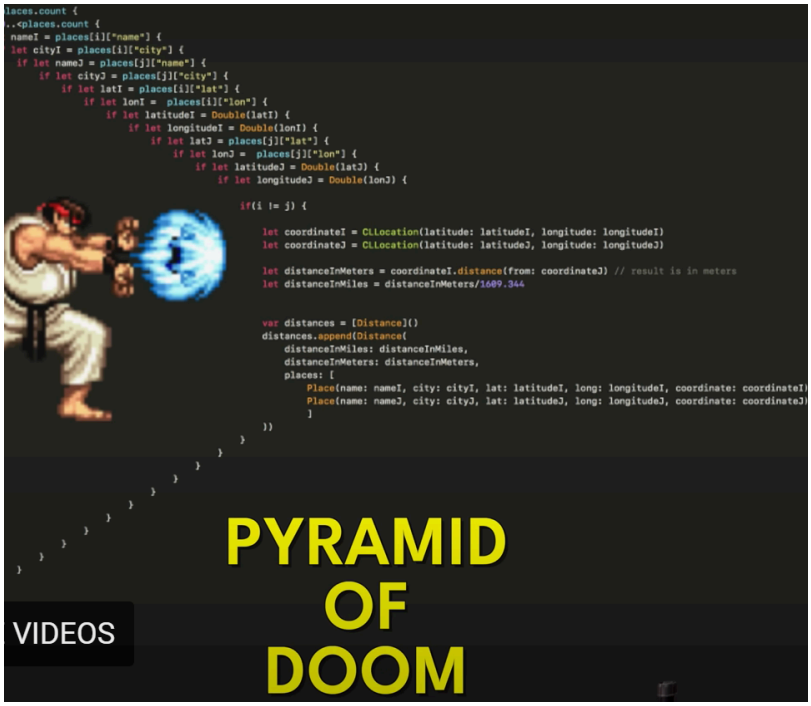
createOrder API will create order and call proceedToPayment. We can proceed to payment only when the order is created. After proceedtoPayment only we can call showSummary. So ProceedToPayment will finish the payment process and call showSummary api. showSummary

api will show the order summary and call updateWallet api. These kind of APIs dependent on another API call is called callback hell. Code starts to grow horizontally instead vertically.
Unreadable and unmaintainable

Issues of Callback Function

- Inversion of Control
- Callback Hell
  Unreadable and unmaitainable

Callback Hell



Inversion of Control

We are blindly trusting createOrder api that it will create the order and will call the proceedToPayment api. We are giving program control to createOrder api. What will happen if createOrder api will have bugs? What will happen if api will never get called? What will happen proceedToPayment will get called twice?

Promises

```
const cart = [ “shoes”, “pants”, “kurta”];

createOrder(cart); // it will return order id

proceedToPayment(orderId);
```

Both createOrder and proceedToPayment APIs are asynchronous function. We don't know how much time these APIs will take and both are dependent on each other and proceedToPayment will get called after orderId is created. We used callback before promises

```
createOrder(cart, function(){
        proceedToPayement(orderId);
});
```

createOrder function will create an order with an orderId and after that it will call proceedToPayement with that orderId. Call back function will lead to Inversion of Control. We

blindly trust createOrder API that it will call or never call the proceedToPayement API. We don't know it called the proceedToPayement or not and proceedToPayement may get called twice or thrice. createOrder may be in other server or cloud or other external world. We are giving the control of code to other API and it's not in our hand to control it. Call back functions are not reliable. Here Promises are coming into picture.

```
const promise = createOrder(cart);
```

We don't know how much time createOrder will take. So we are assuming like it will return an empty object with property data and value undefined (not defined at the moment itself) and then later we are assuming createOrder get called and the property data is filled with value

```
{ data : undefined }
```

So during the execution of the line or calling createOrder API const promise = createOrder(cart); javascript engine will create an empty object with property data and value undefined { data : undefined } and it will move to next line of execution. After some time (how much time the API will take) the property data will be filled with values from API here orderId automatically.

```
{ data : orderDetails }
```

So what will happen to proceedToPayment during this time? We are going to attach the call back function proceedToPayment with our promise object. promise.then()

```
promise.then(function(orderId){
        proceedToPayment (orderId); });
```

Once the data is filled automatically by by createOrderAPI it will call the attached proceedToOrder API automatically.

In Promise also we are attaching call back function. So

How Promise is better than call back function?

Inversion of control

With Call back,

```
const cart = [ "shoes", "pants", "kurta"];
createOrder(cart, function(){
        proceedToPayement(orderId);
});
```

Here passing call back function inside another function. Here we don't have the control of the program in our hand. This is called Inversion of Control. We don't know when createOrder will call proceedToPayment.Here we don't know how ma ny times proceedToPayment will get called by createOrder API.

With Promise,

```
const cart = [ “shoes”, “pants”, “kurta”];
const promise = createOrder(cart);
promise.then(function(orderId){
        proceedToPayment (orderId); });
```

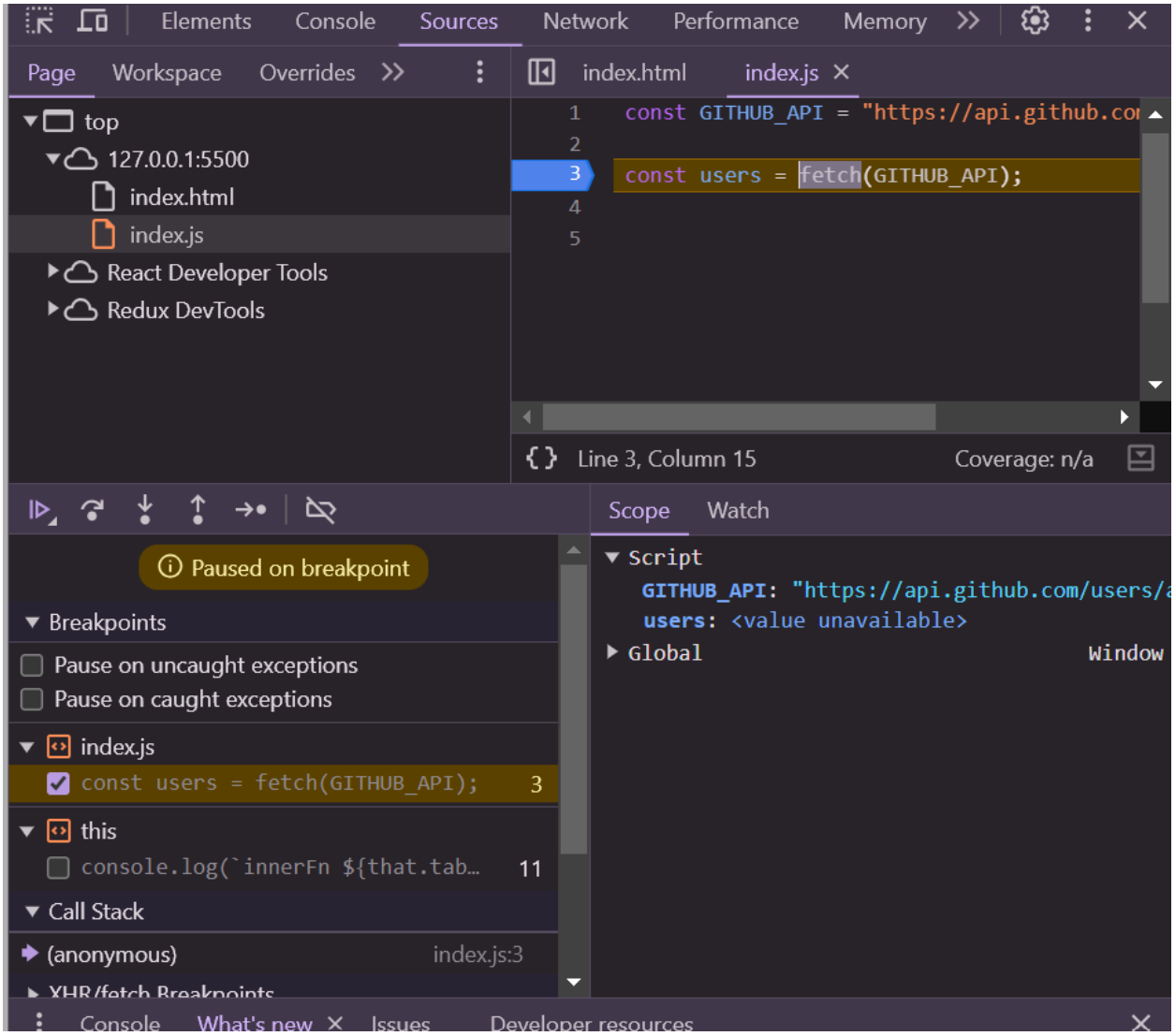
Here we are attaching call back function to the promise object.Here we have the control of the program in our hand. Once the data property is filled with the value from createOrder API, proceedToPayment will get called. So we know when createOrder will call proceedToPayment API.Here we know createOrder API will call proceedToPayment API only once after the data property is filled with the value.

Callback Hell

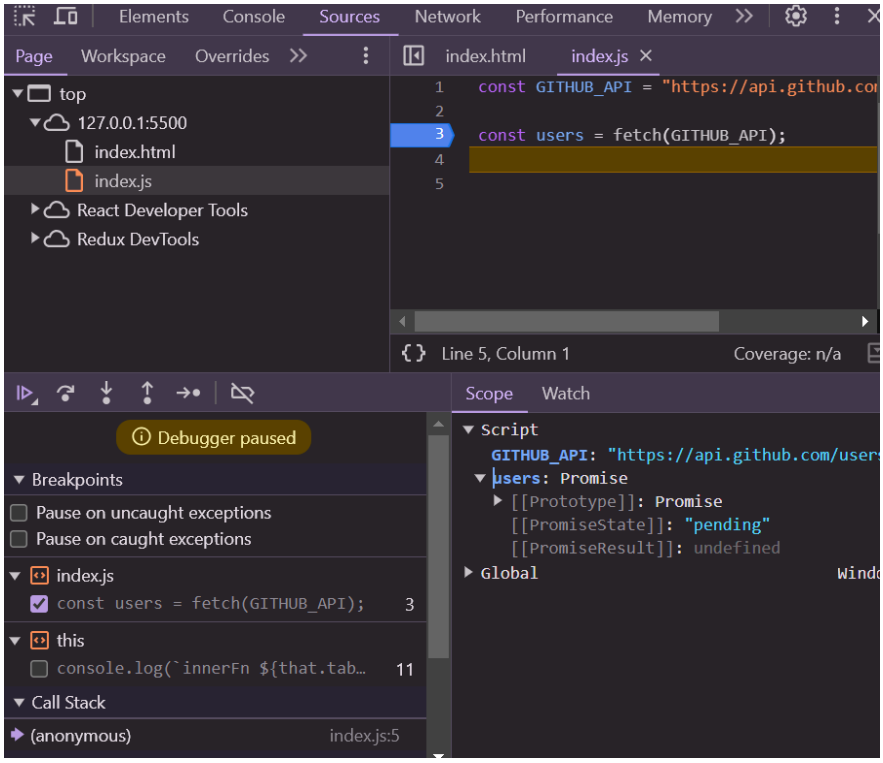
Promise Object

fetch() - fetch function is an API given by browser to make external calls. It returns an object.It is an asynchronous function.

At breakpoint#3, value of users is value unavailable or undefined



After the execution of line# 3,



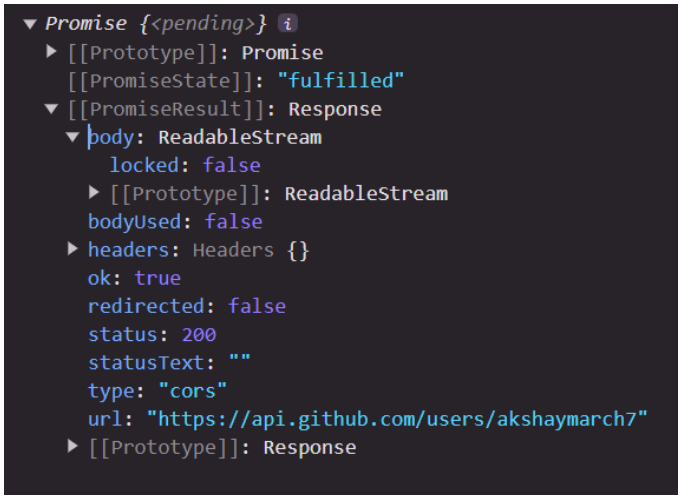
Promise State

What state the promise is. Initially in pending state to fulfilled state.

- Pending
- Fulfilled
- Rejected

Promise Result

Data returns from the fetch function. At line#5 while logging the users object the current state is fulfilled. Data is in PromiseResult: Response -> body: ReadableStream



Promise with Callback Function

index.js index.html

index.js > ...

```
1   const GITHUB_API = "https://api.github.com/users/akshaymarch7";
2
3   const users = fetch(GITHUB_API);
4
5   console.log(users);
6
7   users.then(function(data){
8       console.log(data);
9   })
```

The data from users object will go inside callback function like in line# 7

```
▼ Promise {<pending>} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
  ▼ [[PromiseResult]]: Response
    ▶ body: ReadableStream
      bodyUsed: false
    ▶ headers: Headers {}
      ok: true
      redirected: false
      status: 200
      statusText: ""
      type: "cors"
      url: "https://api.github.com/users/akshaymarch7"
    ▶ [[Prototype]]: Response
▼ Response {type: 'cors', url: 'https://api.github.com/users/akshaymarch7', redirected: false, status: 200, ok: true, …} i
  ▶ body: ReadableStream
    bodyUsed: false
  ▶ headers: Headers {}
    ok: true
    redirected: false
    status: 200
    statusText: ""
    type: "cors"
    url: "https://api.github.com/users/akshaymarch7"
  ▶ [[Prototype]]: Response
>
```

Immutable Promise Object

Promise object is immutable.

Solving Callback Hell

```
createOrder(cart, function(orderId) {
        proceedToPayment(orderId, function(paymentInfo) {
                showOrderSummary(paymentInfo, function () {
                        updateWalletBalance();
});
});
});
```

Promise Chaining

```
createOrder(cart)
.then( function(orderId) {
        return proceedToPayment(orderId);
})
.then(function(paymentInfo) {
        return showOrderSummary(paymentInfo);
})
.then(function(paymentInfo) {
        return updateWalletBalance();
})
```

With arrow function

```
createOrder(cart)
.then(( orderId) => proceedToPayment(orderId))
.then((paymentInfo) => showOrderSummary(paymentInfo))
.then((paymentInfo) => updateWalletBalance())
```

Interview Questions on Promises

What is Promise?

A promise is an object representing the eventual completion or failure of an asynchronous operation. Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function. Promise will be resolved only once. Promise object is an immutable object

Why Promise

Solving Inversion of control in callback function and Callback Hell

Creating a Promise & Error Handling & Promise Chaining

Creating a Promise

```
///
function createOrder(cart) {

    const pr = new Promise(function(resolve, reject){


    });


    return pr;

}
```

Promise( ) is a constructor
Resolve, reject are functions given by javascript

The Promise() constructor is the place for an API call or writing database queries.We can validate the api call whether its success or not based on that we can resolve or reject the promise.

```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId

promise.then(function() {
    proceedToPayment(orderId);
})

/// Producer

function createOrder(cart) {

    const pr = new Promise(function(resolve, reject){
        // createOrder
        // validateCart
        // orderId
        if(!validateCart(cart)) {
            const err = new Error("Cart is not valid");
            reject(err);
        }
        // logic for createOrder
        const orderId = "12345";
        if(orderId) {
            resolve(orderId);
        }
    });

    return pr;
```

Validate cart -> not empty cart, cart items are proper or not

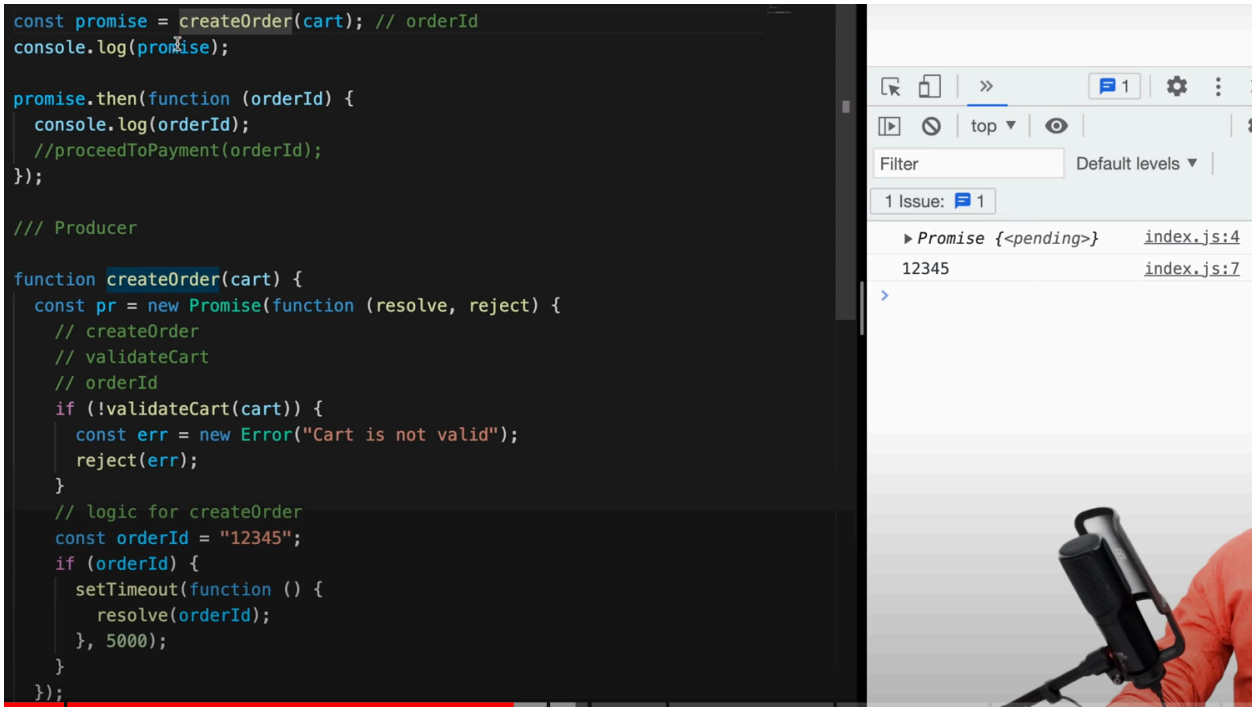
```
function validateCart(cart) {
    return true;
}
```

```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId

promise.then(function (orderId) {
  console.log(orderId);
  //proceedToPayment(orderId);
});

// Producer

function createOrder(cart) {
  const pr = new Promise(function (resolve, reject) {
    // createOrder
    // validateCart
    // orderId
    if (!validateCart(cart)) {
      const err = new Error("Cart is not valid");
      reject(err);
    }
    // logic for createOrder
    const orderId = "12345";
    if (orderId) {
      resolve(orderId);
```

```
1 Issue: 1

  12345                                index.js:6

>
```

```
// logic for createOrder
const orderId = "12345";
if (orderId) {
  setTimeout(function () {
    resolve(orderId);
  }, 5000);
}
```

First the promise was pending then it printed “12345”. This is because createOrder API took 5seconds to execute or resolve. After 5secs the promise got resolved and callback function attached to the promise got executed and printed the order id



Error Handling

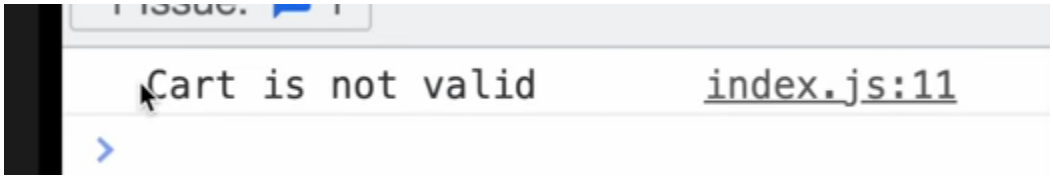
```
function validateCart(cart) {
  return false;
}
```

```
Uncaught (in promise)  index.js:18
Error: Cart is not valid
    at index.js:18:19
    at new Promise (<anonymous>)
    at createOrder (index.js:13:14)
    at index.js:3:17
```

Above one is browser error. As a developer we have to handle the error in code.user will not see this error in the console. So we have handle the error to show it on the page by using .catch(). **.then() is success callback and .catch() is failure callback**

```
promise
  .then(function (orderId) {
    console.log(orderId);
    //proceedToPayment(orderId);
  })
  .catch(function (err) {
    console.log(err.message);
  });

/// Producer

function createOrder(cart) {
  const pr = new Promise(function (resolve, reject) {
    // createOrder
    // validateCart
    // orderId
    if (!validateCart(cart)) {
      const err = new Error("Cart is not valid");
      reject(err);
    }
```

Now we consoled the log in program not a red error in browser



Promise Chaining

If createOrder API executes successfully it will return orderId and got consoled in the first .then() and that orderId will pass as an argument to the proceedToPayment in second .then()

```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
  })
  .then(function() {
    proceedToPayment(orderId);
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

```
function proceedToPayment(orderId) {
    ///
    return new Promise( function(resolve, reject) {
        resolve("Payment Successful");
    })
}
```

Wrong Way

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
  })
  .then(function() {
    proceedToPayment(orderId);
  })
  .then(function(paymentInfo){
      console.log(paymentInfo)
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

Correct Way -> return a promise or result of the promise

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function(orderId) {
    return proceedToPayment(orderId);
  })
  .then(function(paymentInfo){
    console.log(paymentInfo)
})
  .catch(function (err) {
    console.log(err.message);
  });
```

To avoid promise call like below attaching function to promise, we are attaching the call back to the next level of chaining like above

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function(orderId) {
    return proceedToPayment(orderId).then(function(paymentInfo){
        console.log(paymentInfo)
    });
  })

  .catch(function (err) {
    console.log(err.message);
  });
```

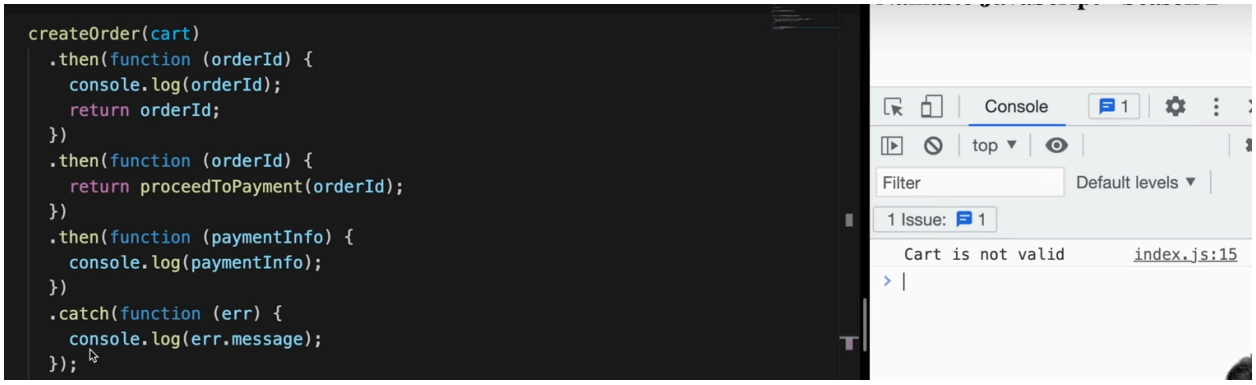
```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function(orderId) {
    return proceedToPayment(orderId);
  })
  .then(function(paymentInfo){
    console.log(paymentInfo)
})
  .catch(function (err) {
    console.log(err.message);
  });
```

```
function validateCart(cart) {
  return true;
}
```

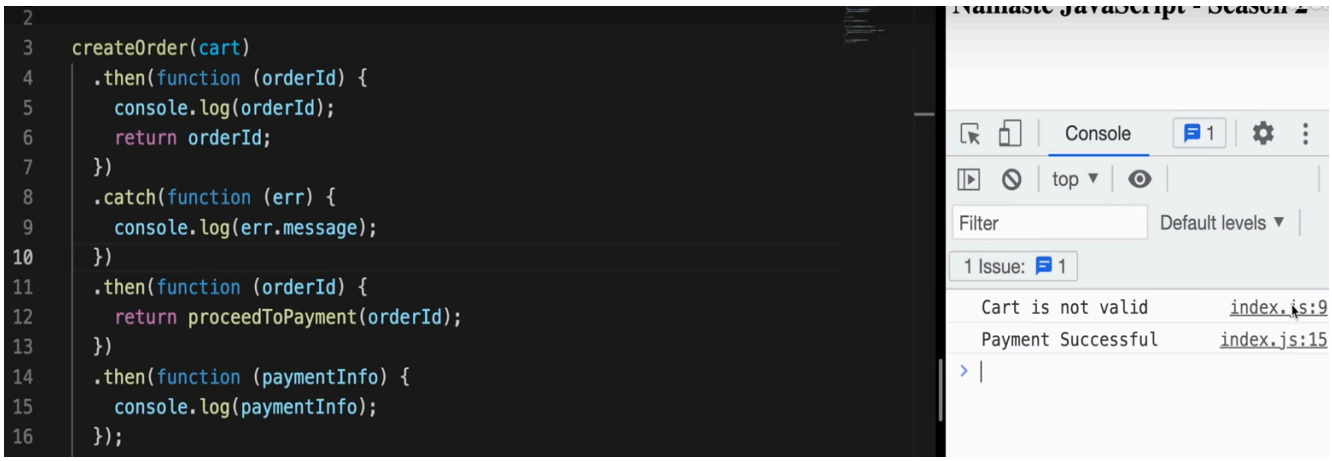
```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

```
top
Filter
Default levels
1 Issue: 1
12345 index.js:5
Payment Successful index.js:12
```

```
function validateCart(cart) {
  return false;
}
false
```

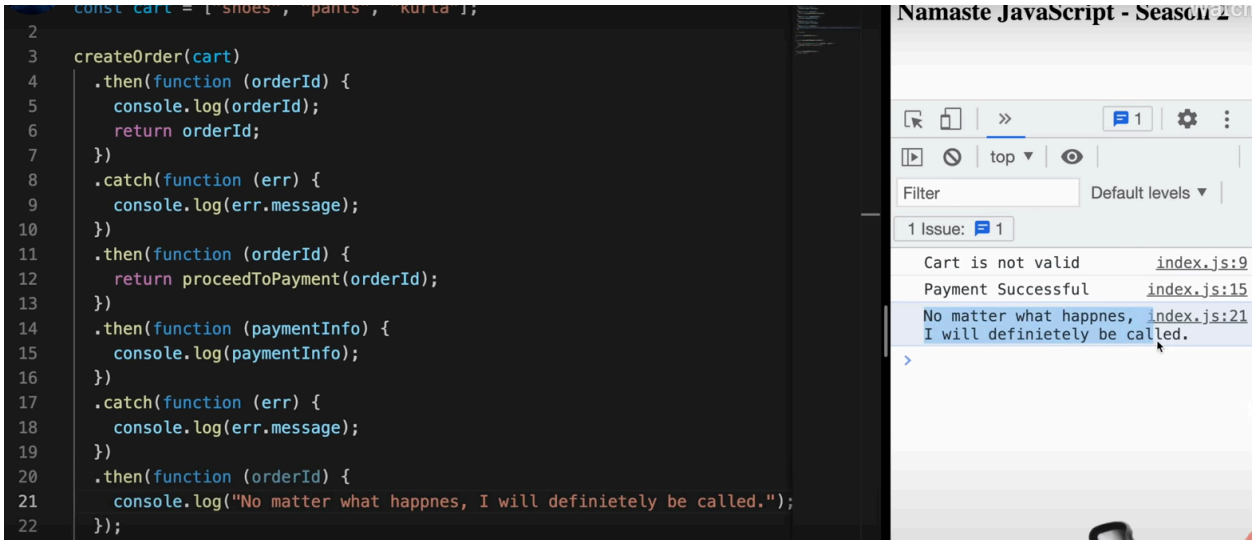


We don't know which promise is failing



.catch() will check only the promises chain above it. In the above scenario proceedToPayment will still run.

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .catch(function (err) {
    console.log(err.message);
  })
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  })
  .catch(function (err) {
    console.log(err.message);
  })
  .then(function (orderId) {
    console.log("No matter what happnes, I will definietely be called.");
  })
  .catch(function (err) {
    console.log(err.message);
```



Async await

- async and await are keywords in JavaScript that are used to work with asynchronous code,

- making it easier to handle asynchronous operations like network requests, file I/O, or timers in a more synchronous-like manner, which can make your code more readable and maintainable.

What is async?

Async is a keyword used before a function to create a async function

Syntax:

```
async function getData(){

}
```

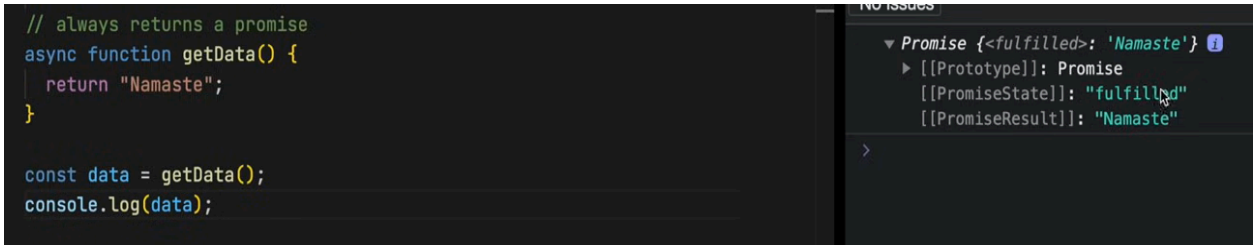
Difference between normal function and async function:

- **async function always returns a promise**.by property of an async function, async function will always return a promise

-If you return a non promise value in async function it will wrap the non promise value inside a promise and return it

- suppose if you return a value which is a non-promise value like string numeric float Boolean for example the string Namaste what this function will do it will basically take this value wrap it inside a promise and it will return it

- **Non-promise value will be converted to promise value like below**



How to get an actual value/data from a promise?

- Using **.then()**

How to create a promise?

```
const p = new Promise((resolve, reject) => {
  resolve("Promise Resolved Value!!");
});
```

How to return a promise from async function?

As we are returning a promise it won't get wrapped again in a promise

```
const p = new Promise((resolve, reject) => {
  resolve("Promise Resolved Value!!");
});

// always returns a promise
async function getData() {
  return p;
}
```

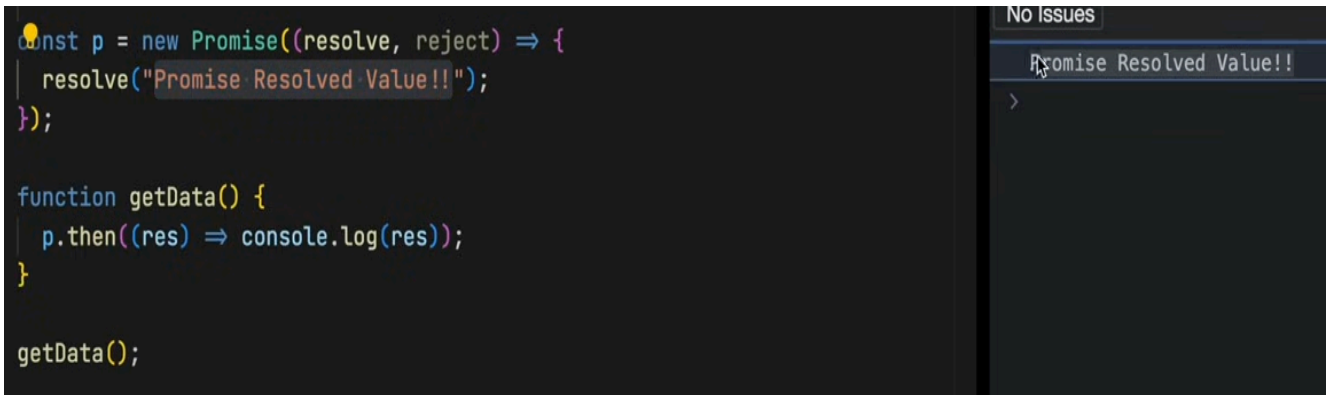
```
const p = new Promise((resolve, reject) => {
  resolve("Promise Resolved Value!!");
});

// always returns a promise
async function getData() {
  return p;
}

const dataPromise = getData();

dataPromise.then((res) => console.log(res));
```

```
Promise Resolved Value!!
>
```

Using await with async

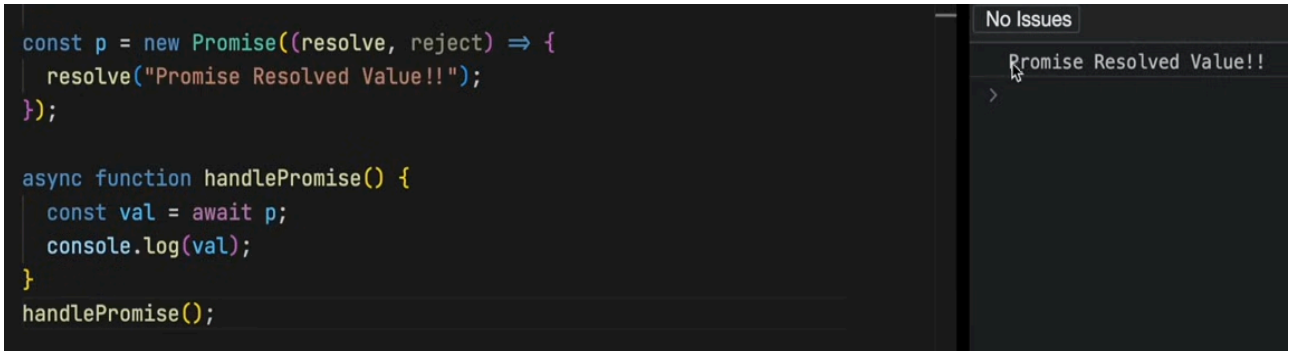
- Async await are used to handle promises

Handling Promises without async await



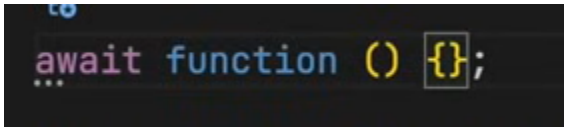
Handling Promises with async await

- use await keyword in front of promise you can get value directly from a promise without using .then()



Await is a keyword that can be used only inside an asynchronous(async) function

If we use await in front of normal function will result in error like below



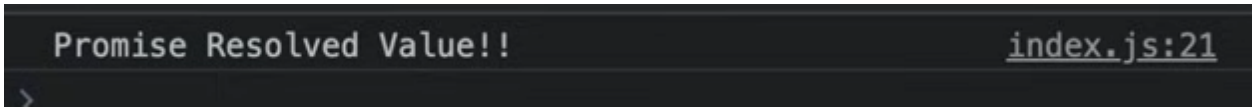
How to handle Promises in normal way and using async await?

Why do we need async await?

Async await, promise are async operations it will happen after some time period. It wont execute immediately.

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise Resolved Value!!");
  }, 10000);
});
```

After 10seconds,



Handling Promise in normal way - using ‘then’

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise Resolved Value!!");
  }, 10000);
});
```

```
function getData() {
  p.then((res) => console.log(res));
  console.log("Namaste JavaScript");
}
```

```
getData();
```

First, Namaste Javascript will get printed and after 10 seconds,
Promise Resolved Value will get printed.
Javascript won't wait for Promise to resolve.It will code ahead and print Namaste Javascript

```
Namaste JavaScript                    index.js:27
Promise Resolved Value!!              index.js:26
>
```

Handling Promise using async await

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise Resolved Value!!");
  }, 10000);
});

// await can only be used inside an async function
async function handlePromise() {
  const val = await p;
  console.log("Namaste JavaScript");
  console.log(val);
}
handlePromise();
```

After 10 seconds only, below will be printed in console

```
Namaste JavaScript                    index.js:21
Promise Resolved Value!!              index.js:22
>
```

Javascript engine will wait for promise to resolve at line# const val = wait p; The whole program will wait until promise has to be resolved. After that only it will goto next line and print 'Namaste Javascript'

Promise	**Async - Await**
Javascript engine will not wait for promise to be resolved. It will go ahead and print the next line and then print the promise value once it got resolved after some time period	Javascript engine will wait for promise to get resolved. It will go the next line only after the Promise got resolved until then whole program will wait.

Scenario 1:

```
11
12  const p = new Promise((resolve, reject) => {
13    setTimeout(() => {
14      resolve("Promise Resolved Value!!");
15    }, 10000);
16  });
17
18  // await can only be used inside an async function
19  async function handlePromise() {
20    console.log("Hello World!!");
21    // JS Engine was waiting for promise to reoslved
22    const val = await p;
23    console.log("Namaste JavaScript");
24    console.log(val);
25  }
26  handlePromise();
27
```

Hello World!! Will get printed immediately
After 10 seconds only it will print 'Namaste JavaScript' and then only Promise Resolved Value will get printed
So order of execution will be like below

```
Hello World!!                    index.js:20
Namaste JavaScript               index.js:23
Promise Resolved Value!!         index.js:24
```

Scenario 2 - with 2 promises

Resolving same Promise twice

```
11
12  const p = new Promise((resolve, reject) => {
13    setTimeout(() => {
14      resolve("Promise Resolved Value!!");
15    }, 10000);
16  });
17
18  // await can only be used inside an async function
19  async function handlePromise() {
20    console.log("Hello World!!");
21    // JS Engine was waiting for promise to reoslved
22    const val = await p;
23    console.log("Namaste JavaScript");
24    console.log(val);
25
26    const val2 = await p;
27    console.log("Namaste JavaScript 2");
28    console.log(val2);
29  }
30  handlePromise();
31
```

It will print 'Hello World!!' first and the program will wait for 10 seconds. After 10 seconds 'Namaste Javascript', 'Promise Resolved Value!!','Namaste Javascript','Promise Resolved Value!!' will get printed

```
Hello World!!                                    index.js:20
Namaste JavaScript                               index.js:23
Promise Resolved Value!!                         index.js:24
Namaste JavaScript 2                             index.js:27
Promise Resolved Value!!                         index.js:28
```

Resolving two different Promises

```
12  const p1 = new Promise((resolve, reject) => {
13    setTimeout(() => {
14      resolve("Promise Resolved Value!!");
15    }, 10000);
16  });
17
18  const p2 = new Promise((resolve, reject) => {
19    setTimeout(() => {
20      resolve("Promise Resolved Value!!");
21    }, 5000);
22  });
23
24
25  // await can only be used inside an async function
26  async function handlePromise() {
27    console.log("Hello World!!");
28    // JS Engine was waiting for promise to reoslved
29    const val = await p1;
30    console.log("Namaste JavaScript");
31    console.log(val);
32
33    const val2 = await p2;
34    console.log("Namaste JavaScript 2");
35    console.log(val2);
36  }
37  handlePromise();
38
```

It will print 'Hello World!!' first and the program will wait for 10 seconds. After 10 seconds everything 'Namaste Javascript', 'Promise Resolved Value!!','Namaste Javascript','Promise Resolved Value!!' will get printed. Promise p2 with timer 5 secs will wait for Promise p1 with timer 10 secs. So after 10 secs both promise got resolved and output will be printed.

Scenario 3 - with change in timer

Hello World! will print immediately. After 5 secs promise p1 will get resolved and print ‘Namaste Javascript’ and ‘Promise Resolved value!!’ and After 10 secs promise p2 will get resolved and print ‘Namaste Javascript 2’ and ‘Promise Resolved value!!’

```
12  const p1 = new Promise((resolve, reject) => {
13    setTimeout(() => {
14      resolve("Promise Resolved Value!!");
15    }, 5000);
16  });
17
18  const p2 = new Promise((resolve, reject) => {
19    setTimeout(() => {
20      resolve("Promise Resolved Value!!");
21    }, 5000);
22  });
23
24
25  // await can only be used inside an async function
26  async function handlePromise() {
27    console.log("Hello World!!");
28    // JS Engine was waiting for promise to reoslved
29    const val = await p1;
30    console.log("Namaste JavaScript");
31    console.log(val);
32
33    const val2 = await p2;
34    console.log("Namaste JavaScript 2");
35    console.log(val2);
36  }
37  handlePromise();
38
```

```
Hello World!!                                   index.js:26
Namaste JavaScript                              index.js:29
Promise Resolved Value!!                        index.js:30
Namaste JavaScript 2                            index.js:33
Promise Resolved Value!!                        index.js:34
```

What happens behind the scenes of Promises Async Await

34:20 - 48:35

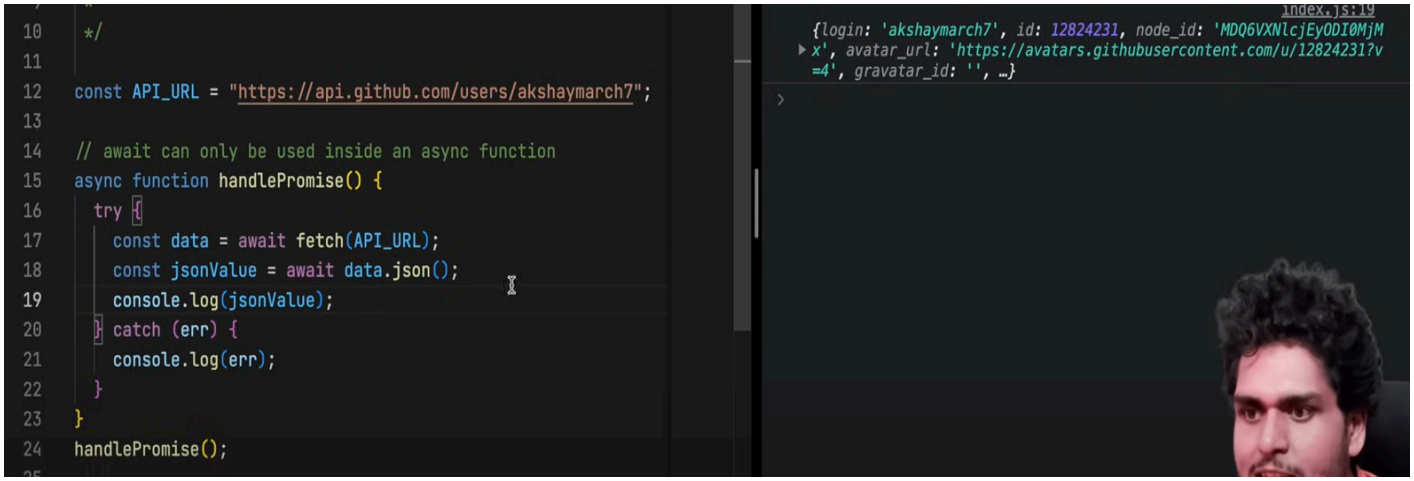
Real Time example of Async/Await Using Fetch

fetch() => fetch is a promise object. When it got resolved it will give you a response object. This response has a body and it is a readablestream. To convert to JSON format we have to do Response.json(). Response.json() is again a promise. When Response.json() got resolved it will give you the result actual json value.

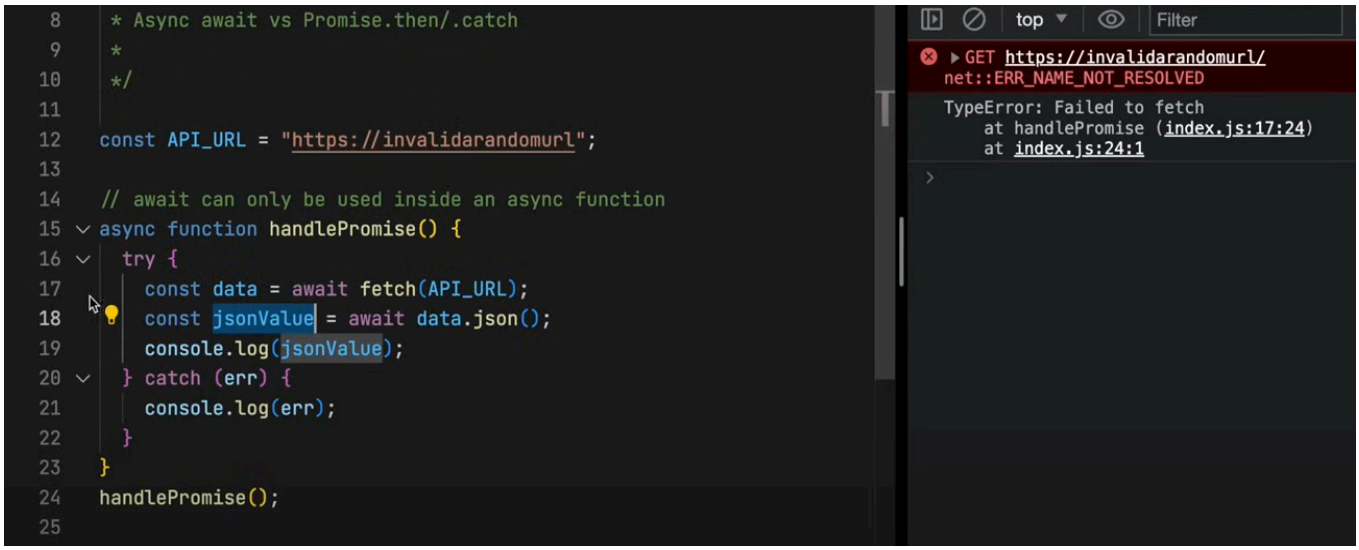
```
const API_URL = "https://api.github.com/users/akshaymarch7";


// await can only be used inside an async function
async function handlePromise() {

  const data = await fetch(API_URL);

  const jsonValue = await data.json()

  fetch().then(res=> res.json()).then(res=> consolelog())



  // fetch() => Response.json() => jsonValue

}
handlePromise();
```

```
const API_URL = "https://api.github.com/users/akshaymarch7";


// await can only be used inside an async function
async function handlePromise() {

  const data = await fetch(API_URL);

  const jsonValue = await data.json();

  console.log(jsonValue)


}
handlePromise();
```

Error Handling - Try & Catch



Invalid url - network call failed - Type error failed to fetch



Async function always return a promise. So we can use .catch() on async function like below

Interview Tips

1. What is async/await?

   Async is a keyword used with function. Explain about async function. Await can be used only with async function to handle promises.

Async/Await vs Promise.then/catch

async/await keywords are syntactic sugar over promise.then catch. Await use promise.then() behind the scene. Async await avoid promise chaining that means avoiding/ no need to pass callback function