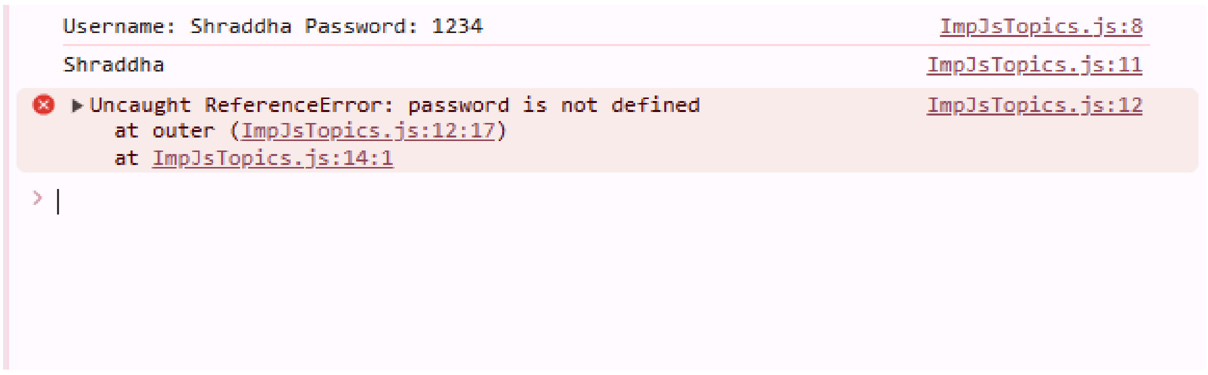
JavaScript Part 2

Lexical Scope

```
// lexical scope - it is the simple term inner
function can access the outer function variables
but outer function cannot access the inner
function variables.

function outer(){
    let username = "Shraddha";
    function inner(){
        let password = "1234";
        console.log("Username: "+username,
"Password: "+password);
    }
    inner();
    console.log(username);
    console.log(password);
}
outer();
```

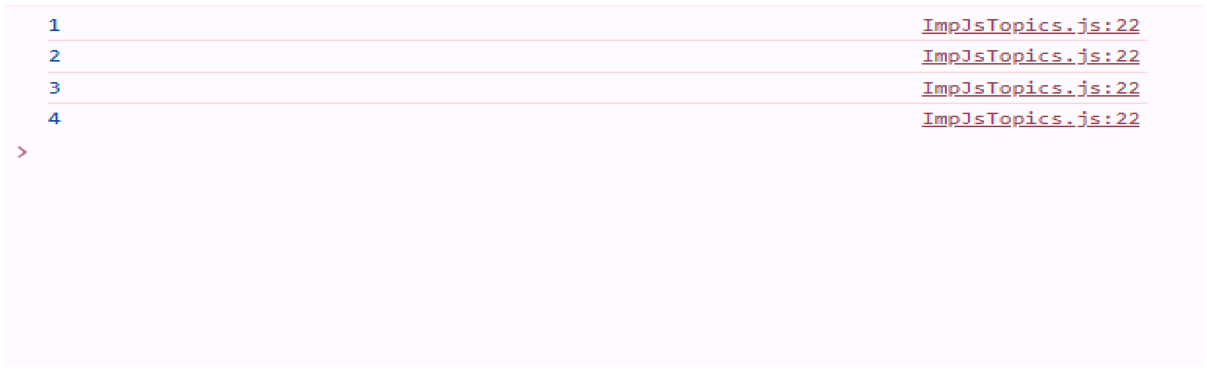


```
Username: Shraddha Password: 1234                     ImpJsTopics.js:8
Shraddha                                              ImpJsTopics.js:11
▶Uncaught ReferenceError: password is not defined     ImpJsTopics.js:12
    at outer (ImpJsTopics.js:12:17)
    at ImpJsTopics.js:14:1
> |
```

This is because the inner function's lexical scope includes the scope of the outer function.

Closure

```
// Closure - function with lexical scope = closure
function outer(){
    let count = 0;
    function inner(){
        count++;
        console.log(count);
    }
    return inner;
}
let counter = outer();
counter();
counter();
counter();
counter();
```



```
1                                              ImpJsTopics.js:22
2                                              ImpJsTopics.js:22
3                                              ImpJsTopics.js:22
4                                              ImpJsTopics.js:22
>
```

Use closure when it is actually needed. Because there is lots of memory uses which can be turned into memory leaks.

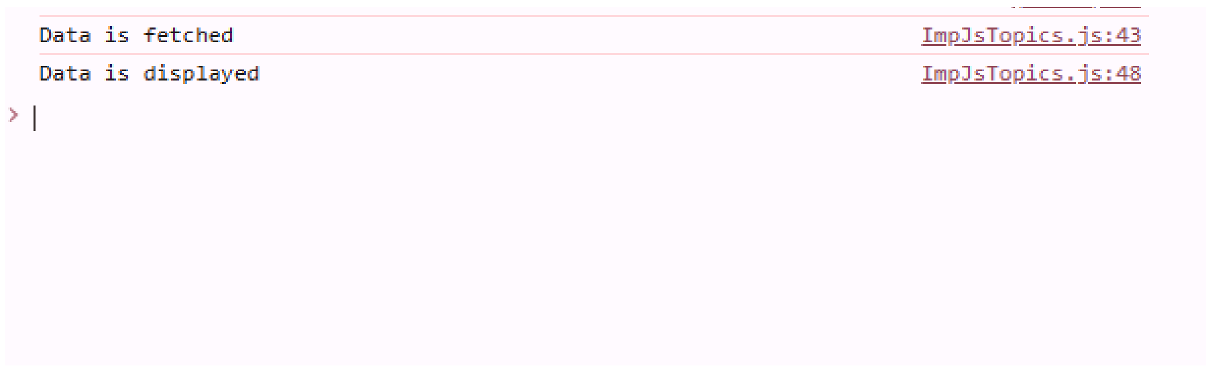
```
// JavaScript is single threded language. It can do one thing at a time. It has a call stack and event loop. everything happens sequentially. line by line execution.
// It is inefficient to wait for the response from the server. So, we use asynchronous programming.
```

```
It is non-blocking. It is used to handle multiple
requests at the same time.
// Callbacks - It is a function that is passed as
an argument to another function and is executed
after the completion of the task.
// Promices - It is an object that represents the
eventual completion or failure of an asynchronous
operation. It is used to handle multiple
asynchronous operations.
// Async/Await - It is a syntactic sugar for
promises. It makes the code more readable and easy
to understand.
// Asynchronous operation = non-blocking behaviour
// Synchronous operation = blocking / sequential
operations.
```

Callbacks

```
// callback
function getData(callback){
    setTimeout(()=>{
        console.log("Data is fetched");
        callback();
    }, 2000);
}
function displayData(){
    console.log("Data is displayed");
}
getData(displayData);
```

```
// after 2000 mlsec, it will display the data.
```



```
Data is fetched                                          ImpJsTopics.js:43
Data is displayed                                        ImpJsTopics.js:48
>
```

Callback : function ke andar dusra function pass karto tyala callback mantat.
Callbacks synchronous kam kartat aani with setTimeout function ni asynchronous ni kam kartat
A callback is a function passed as an argument to another function.
Function is passed to be called when some operation happens.

Problems : Callback hell / Pyramid of doom. (It is complex and not readable)
To solve this problem Promises have come into the picture.

Promises

In JavaScript, a Promise is an object that represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

Promise is object in javascript , which has 3 states pending state, resolve/fulfilled , reject
Promises are readable asynchronous opeartions.

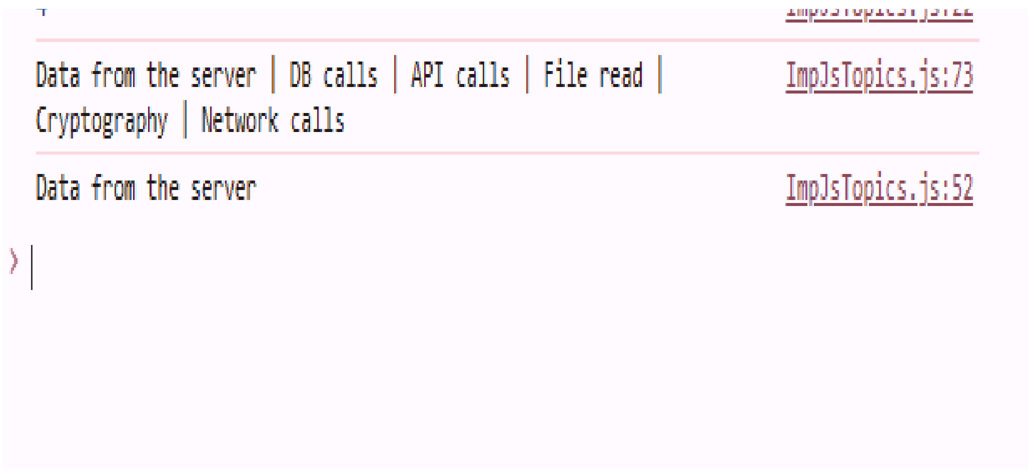
A Promise is in one of these states:

- pending: initial state, neither fulfilled nor rejected.
- fulfilled: meaning that the operation was completed successfully.
- rejected: meaning that the operation failed.

Mostly we consume promises. One part is the creation of promises and then the second one is consuming already created promises.

```
//creation of promise


const myPromise = new Promise((resolve, reject)=>{
    let data = "Data from the server | DB calls |
API calls | File read | Cryptography | Network
calls";
    let error = null;
    if(error){
        reject(error);//calling asynchronous
operation ie calling catch block
    }else{
        resolve(data);// calling then block
    }
})
// consume the promise
myPromise.then((data)=>{
    console.log(data);
})
.catch((error)=>{
    console.error(error);
})
```



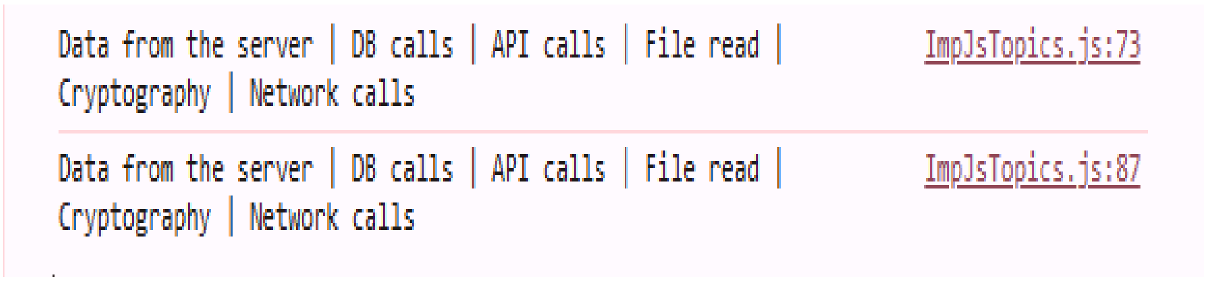
```
Data from the server | DB calls | API calls | File read |        ImpJsTopics.js:73
Cryptography | Network calls
Data from the server                                             ImpJsTopics.js:52
>
```

Async/Await

```
// Async/Await - Instead of using then and catch, we can use async and await. which makes the code more readable and easy to understand.
// async - it is used to define an asynchronous function.
// await - it is used to wait for the promise to be resolved.
// async function always returns a promise.

async function getData(){
    try{
        let data = await myPromise;
        console.log(data);
    }
    catch(error){
        console.error(error);
    }
}

getData();
```



```
Data from the server | DB calls | API calls | File read |          ImpJsTopics.js:73
Cryptography | Network calls
Data from the server | DB calls | API calls | File read |          ImpJsTopics.js:87
Cryptography | Network calls
```

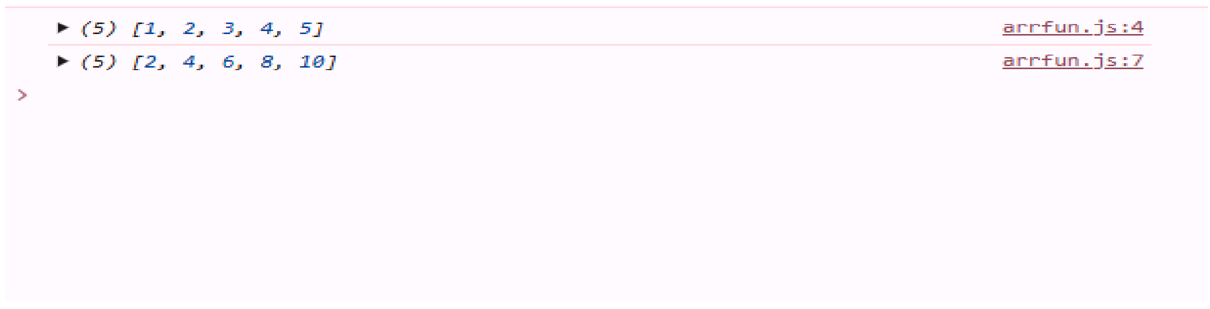
Common Higher order functions

Array Manipulation mostly used Functions

Map

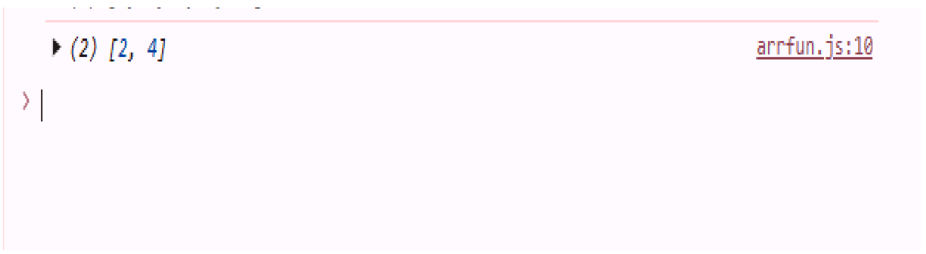
```
// 1.map
const nums=[1,2,3,4,5];
console.log(nums);

const doubnums=nums.map((num) => num*2);
console.log(doubnums);
```



```
▶ (5) [1, 2, 3, 4, 5]                                   arrfun.js:4
▶ (5) [2, 4, 6, 8, 10]                                  arrfun.js:7
>
```

```
// 2.filter
const evennums=nums.filter((num)=>num%2==0);
console.log(evennums);
```

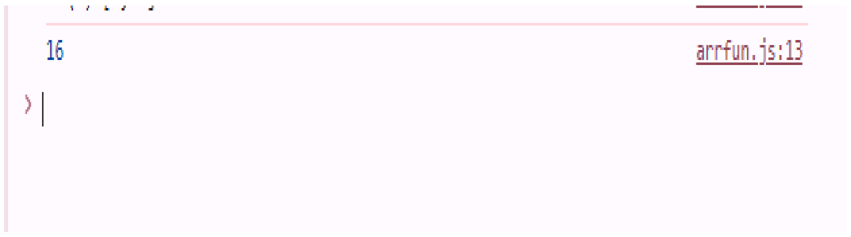


```
▶ (2) [2, 4]                                  arrfun.js:10
>|
```

```
// 3.reduce
const sum=nums.reduce((accumulator,
num)=>accumulator+num, 0);
console.log(sum);
```

```
// 3.reduce
const sum=nums.reduce((accumulator,
num)=>accumulator+num, 1);
console.log(sum);
```

16



```
16                                                  arrfun.js:13
>
```