



# Node js CallBack and Promises

#Node JS Notes

# Callbacks



# What are callbacks



- In JavaScript, a callback is a function passed into another function as an argument to be executed later.
- Callbacks are generally used to continue the execution after completing an asynchronous operation - such are referred to as the **asynchronous callbacks**.



# When to use callback functions in JavaScript?

- when working with the file system (downloading or uploading),
- Sending the network request to get some resources such as text or binary file from the server,
- events,
- the DOM in the browser
- or working with web APIs to fetch data.



# Simple Function

```
JS test.js ×  
JS test.js > ...  
1 // Simple function  
2 function ShowMsg(name) {  
3     console.log('Hi' + ' ' + name);  
4 }  
5 ShowMsg('Akash'); // Hi Akash  
  
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE  
  
Microsoft Windows [Version 10.0.22000.258]  
(c) Microsoft Corporation. All rights reserved.  
  
D:\lecture\demo>node test.js  
Hi Akash
```



# JavaScript Callback Synchronous

- In JavaScript, you can also pass a function as an argument to a function. This function that is passed as an argument inside of another function is called a callback function.

```
JS test.js x
JS test.js > ...
1 // Simple function
2 function ShowMsg(name,callback) {
3     console.log('Hi' + ' ' + name);
4     callback();
5 }
6
7 function DemoCallBack(){
8     console.log("I am callback function");
9 }
10
11 ShowMsg('Akash',DemoCallBack); // Hi Akash

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

D:\lecture\demo>node test.js
Hi Akash
I am callback function

D:\lecture\demo>
```

```
// Simple function
function ShowMsg(name,callback) {
    console.log('Hi' + ' ' + name);
    callback();
}

function DemoCallBack(){
    console.log("I am callback function");
}

ShowMsg('Akash',DemoCallBack); // Hi Akash
```



# CallBack using SetTimeout

- **setTimeout()** is a JavaScript asynchronous function that executes a code block or evaluates an expression through a callback function after a delay set in milliseconds.

```
JS test.js  X
JS test.js > ...
1  console.log("Welcome")
2  setTimeout(() => {
3      // runs after 3 seconds
4      console.log('Hello callback setTimeout function')
5  }, 3000)
6  console.log("Byee")
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
D:\lecture\demo>node test.js
Welcome
Byee
Hello callback setTimeout function
```



# JavaScript Callback Asynchronous

- Callbacks can also be used to execute code asynchronously.

```
JS test.js ×  
JS test.js > ...  
1 function doSomethingAsync(then) {  
2     setTimeout(then, 1000);  
3     console.log('call first asynchronously');  
4 }  
5 doSomethingAsync(function () {  
6     console.log('Done');  
7 });  
8 console.log('call second');
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
D:\lecture\demo>node test.js  
call first asynchronously  
call second  
Done
```





# Callback hell

- Multiple functions can be created independently and used as callback functions. These create multi-level functions. When this function tree created becomes too large, the code becomes incomprehensible sometimes and is not easily refactored. This is known as **callback hell**.

```
JS test.js  ×
JS test.js > ...
1  asyncFunction(function () {
2    asyncFunction(function () {
3      asyncFunction(function () {
4        asyncFunction(function () {
5          asyncFunction(function () {
6            //....
7          });
8        });
9      });
10    });
11  });
```



```
// a bunch of functions are defined up here

// lets use our functions in callback hell
function setInfo(name) {
  address(myAddress) {
    officeAddress(myOfficeAddress) {
      telephoneNumber(myTelephoneNumber) {
        nextOfKin(myNextOfKin) {
          console.log('done'); //let's begin to close each function!
        };
      };
    };
  };
}
```



# How to avoid Callback Hell/ Pyramid of doom

- to avoid callback hell or the pyramid of doom we can use multiple techniques which are as follows:
- By using **promises**
- By using **async/await** functions.



# Promises



# Promises



- A promise is an object that allows you to handle asynchronous operations. It's an alternative to plain old callbacks.
- Promises have many advantages over callbacks. To name a few:
  - Make the async code easier to read.
  - Provide combined error handling.
  - Better control flow. You can have async actions execute in parallel or series.
- Reference : [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using\\_promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises)



# Callback VS Promises

- The promise object exposes the methods **.then** and **.catch**.
- We are going to explore these methods later.

```
a() => {  
  b() => {  
    c() => {  
      d() => {  
        // and so on ...  
      };  
    };  
  };  
};
```

```
Promise.resolve()  
  .then(a)  
  .then(b)  
  .then(c)  
  .then(d)  
  .catch(console.error);
```



# callback to promises

- We can convert callbacks into promises using the Promise constructor.
- The Promise constructor takes a callback with two arguments resolve and reject.
  - **Resolve**: is a callback that should be invoked when the async operation is completed.
  - **Reject**: is a callback function to be invoked when an error occurs.



# promises just callbacks?

- Promises are not “just” callbacks, but they do use asynchronous callbacks on the .then and .catch methods.
- Promises are an abstraction on top of callbacks that allows you to chain multiple async operations and handle errors more elegantly.





# Promise states

- There are four states in which the promises can be:

- **Pending:**

- initial state. Async operation is still in process.

- **Fulfilled:**

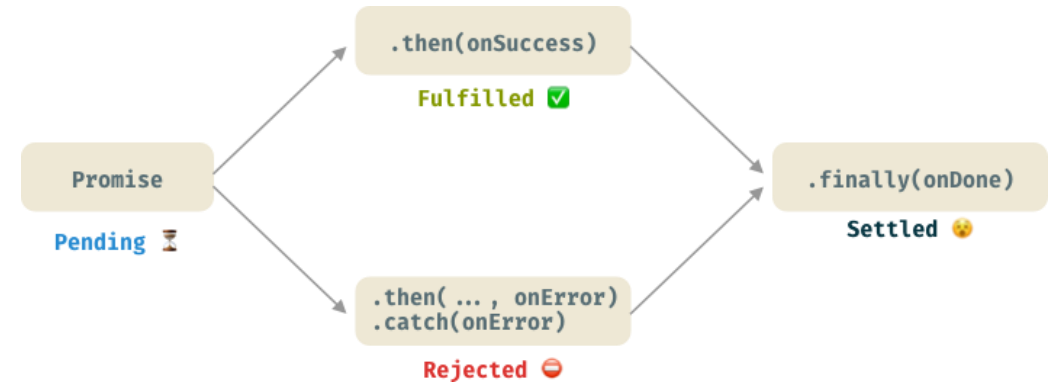
- the operation was successful. It invokes .then callback. E.g., .then(onSuccess).

- **Rejected:**

- the operation failed. It invokes the .catch or .then 's second argument (if any). E.g., .catch(onError) or .then(..., onError)

- **Settled:**

- it's the promise final state. The promise is dead. Nothing else can be resolved or rejected anymore. The .finally method is invoked.



# JavaScript Promise Methods

<u>Method</u>	<u>Description</u>
all(iterable)	Waits for all promises to be resolved or any one to be rejected
allSettled(iterable)	Waits until all promises are either resolved or rejected
any(iterable)	Returns the promise value as soon as any one of the promises is fulfilled
race(iterable)	Wait until any of the promises is resolved or rejected
reject(reason)	Returns a new Promise object that is rejected for the given reason
resolve(value)	Returns a new Promise object that is resolved with the given value
catch()	Appends the rejection handler callback
then()	Appends the resolved handler callback
finally()	Appends a handler to the promise



# Promise instance methods

- The Promise API exposes three main methods:
  - then
  - catch
  - finally
- Let's explore each one and provide examples.



# Promise then

- The then method allows you to get notified when the asynchronous operation is done, either succeeded or failed.
- It takes two arguments, one for the successful execution and the other one if an error happens

**`promise.then(onSuccess, onError);`**

- You can also use catch to handle errors:

**`promise.then(onSuccess).catch(onError);`**



# Promise chaining

- then returns a new promise so you can chain multiple promises together

```
Promise.resolve()  
  .then(() => console.log('then#1'))  
  .then(() => console.log('then#2'))  
  .then(() => console.log('then#3'));
```

- Promise.resolve immediately resolves the promise as successful. So all the following then are called. The output would be

```
then#1  
then#2  
then#3
```



# Promise catch

- Promise **.catch** the method takes a function as an argument that handles errors if they occur.
- If everything goes well, the catch method is never called.

```
Promise.resolve()  
  .then(a)  
  .then(b)  
  .then(c)  
  .then(d)  
  .catch(console.error)
```



JS demo.js > ...

```
1  const a = () => new Promise((resolve) => setTimeout(() => { console.log('a'), resolve() }, 1000));
2  const b = () => new Promise((resolve) => setTimeout(() => { console.log('b'), resolve() }, 1000));
3  const c = () => new Promise((resolve, reject) => setTimeout(() => { console.log('c'), reject('Oops!') }, 1000));
4  const d = () => new Promise((resolve) => setTimeout(() => { console.log('d'), resolve() }, 1000));
5  Promise.resolve()
6    .then(a)
7    .then(b)
8    .then(c)
9    .then(d)
10   .catch(console.error)
11
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

powershell

PS D:\lecture\socketdemo> node .\demo.js

a

b

c

Oops!

PS D:\lecture\socketdemo>



# Example

```
const a = () => new Promise((resolve) => setTimeout(() => { console.log('a'), resolve() }, 1000));
const b = () => new Promise((resolve) => setTimeout(() => { console.log('b'), resolve() }, 1000));
const c = () => new Promise((resolve, reject) => setTimeout(() => { console.log('c'), reject('Oops!') }, 1000));
const d = () => new Promise((resolve) => setTimeout(() => { console.log('d'), resolve() }, 1000));
Promise.resolve()
  .then(a)
  .then(b)
  .then(c)
  .then(d)
  .catch(console.error)
```





# Promise finally

- The **finally** method is called only when the **promise is settled**.
- You can use a **.then** after the **.catch**, in case you want a piece of code to execute always, even after a failure.

```
demo.js > ...
1  const a = () => new Promise((resolve) => setTimeout(() => { console.log('a'), resolve() }, 1000));
2  const b = () => new Promise((resolve) => setTimeout(() => { console.log('b'), resolve() }, 1000));
3  const c = () => new Promise((resolve, reject) => setTimeout(() => { console.log('c'), reject('Oops!') }, 1000));
4  const d = () => new Promise((resolve) => setTimeout(() => { console.log('d'), resolve() }, 1000));
5  Promise.resolve()
6    .then(a)
7    .then(b)
8    .then(c)
9    .then(d)
10   .catch(console.error)
11   .finally(() => console.log('always called'));
12
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS D:\lecture\socketdemo> node .\demo.js
a
b
c
Oops!
always called
PS D:\lecture\socketdemo>
```



# Promise class Methods

- There are four static methods that you can use directly from the Promise object.
  - Promise.all
  - Promise.reject
  - Promise.resolve
  - Promise.race



# Promise.resolve and Promise.reject

- These two are helper functions that **resolve** or **reject** immediately.
- You can pass a reason that will be passed on the next `.then`.

```
JS demo.js
1  Promise.resolve('Yeay!!!')
2    .then(console.log)
3    .catch(console.error)
4
5  Promise.reject('Oops :( ')
6    .then(console.log)
7    .catch(console.error)
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
PS D:\lecture\socketdemo> node .\demo.js
Yeay!!!
Oops :(
PS D:\lecture\socketdemo> 
```



# Executing promises in Parallel with Promise.all

- Usually, promises are executed in series, one after another, but you can use them in parallel as well.
- Promise.all accepts an array of promises.

```
demo.js > ...
1  const a = () => new Promise((resolve) => setTimeout(() => resolve('a'), 2000));
2  const b = () => new Promise((resolve) => setTimeout(() => resolve('b'), 1000));
3  const c = () => new Promise((resolve) => setTimeout(() => resolve('c'), 1000));
4  const d = () => new Promise((resolve) => setTimeout(() => resolve('d'), 1000));
5
6  console.time('promise.all');
7  Promise.all([a(), b(), c(), d()])
8    .then(results => console.log(`Done! ${results}`))
9    .catch(console.error)
10   .finally(() => console.timeEnd('promise.all'));
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

```
PS D:\lecture\socketdemo> node .\demo.js
Done! a,b,c,d
promise.all: 2.011s
PS D:\lecture\socketdemo> 
```



# Example

```
const a = () => new Promise((resolve) => setTimeout(() => resolve('a'), 2000));  
const b = () => new Promise((resolve) => setTimeout(() => resolve('b'), 1000));  
const c = () => new Promise((resolve) => setTimeout(() => resolve('c'), 1000));  
const d = () => new Promise((resolve) => setTimeout(() => resolve('d'), 1000));
```

```
console.time('promise.all');  
Promise.all([a(), b(), c(), d()])  
  .then(results => console.log(`Done! ${results}`))  
  .catch(console.error)  
  .finally(() => console.timeEnd('promise.all'));
```



# Promise race

- The `Promise.race(iterable)` takes a collection of promises and resolves as soon as the first promise settles.



# Example

- Output : It's b! With Promise.race only the fastest gets to be part of the result.

```
JS demo.js > ...
1  const a = () => new Promise((resolve) => setTimeout(() => resolve('a'), 2000));
2  const b = () => new Promise((resolve) => setTimeout(() => resolve('b'), 1000));
3  const c = () => new Promise((resolve) => setTimeout(() => resolve('c'), 1000));
4  const d = () => new Promise((resolve) => setTimeout(() => resolve('d'), 1000));
5
6  console.time('promise.race');
7  Promise.race([a(), b(), c(), d()])
8    .then(results => console.log(`Done! ${results}`))
9    .catch(console.error)
10   .finally(() => console.timeEnd('promise.race'));

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

PS D:\lecture\socketdemo> node .\demo.js
Done! b
promise.race: 1.013s
PS D:\lecture\socketdemo> 
```

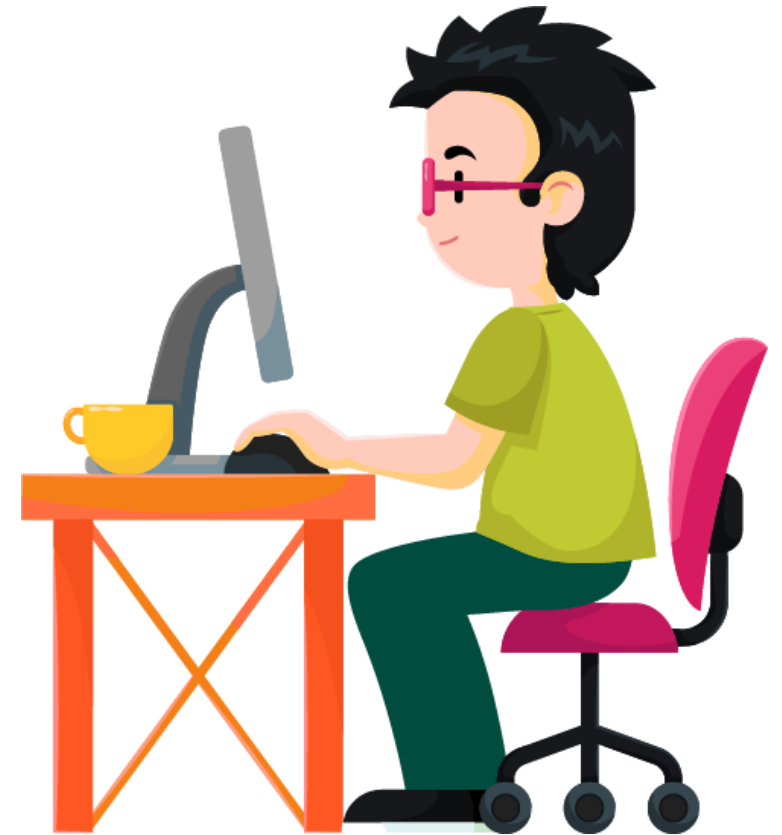


# Get Exclusive Video Tutorials



[www.apptutorials.com](http://www.apptutorials.com)

<https://www.youtube.com/user/Akashtips>





# Connect With Me



Akash Padhiyar  
#AkashSir

[www.akashsir.com](http://www.akashsir.com)  
[www.akashtechlabs.com](http://www.akashtechlabs.com)  
[www.akashpadhiyar.com](http://www.akashpadhiyar.com)  
[www.apptutorials.com](http://www.apptutorials.com)

## # Social Info



Akash.padhiyar



Akashpadhiyar



Akash\_padhiyar



+91 99786-21654



#Akashpadhiyar  
#apptutorials



Get More Details

[www.akashsir.com](http://www.akashsir.com)



# If You Liked It !

## Rating Us Now



**Just Dial**

[https://www.justdial.com/Ahmedabad/Akash-Technolabs-Navrangpura-Bus-Stop-Navrangpura/O79PXX79-XX79-170615221520-S5C4\\_BZDET](https://www.justdial.com/Ahmedabad/Akash-Technolabs-Navrangpura-Bus-Stop-Navrangpura/O79PXX79-XX79-170615221520-S5C4_BZDET)



**Sulekha**

<https://www.sulekha.com/akash-technolabs-navrangpura-ahmedabad-contact-address/ahmedabad>

