Node js CallBack and Promises

#Node JS Notes

Callbacks

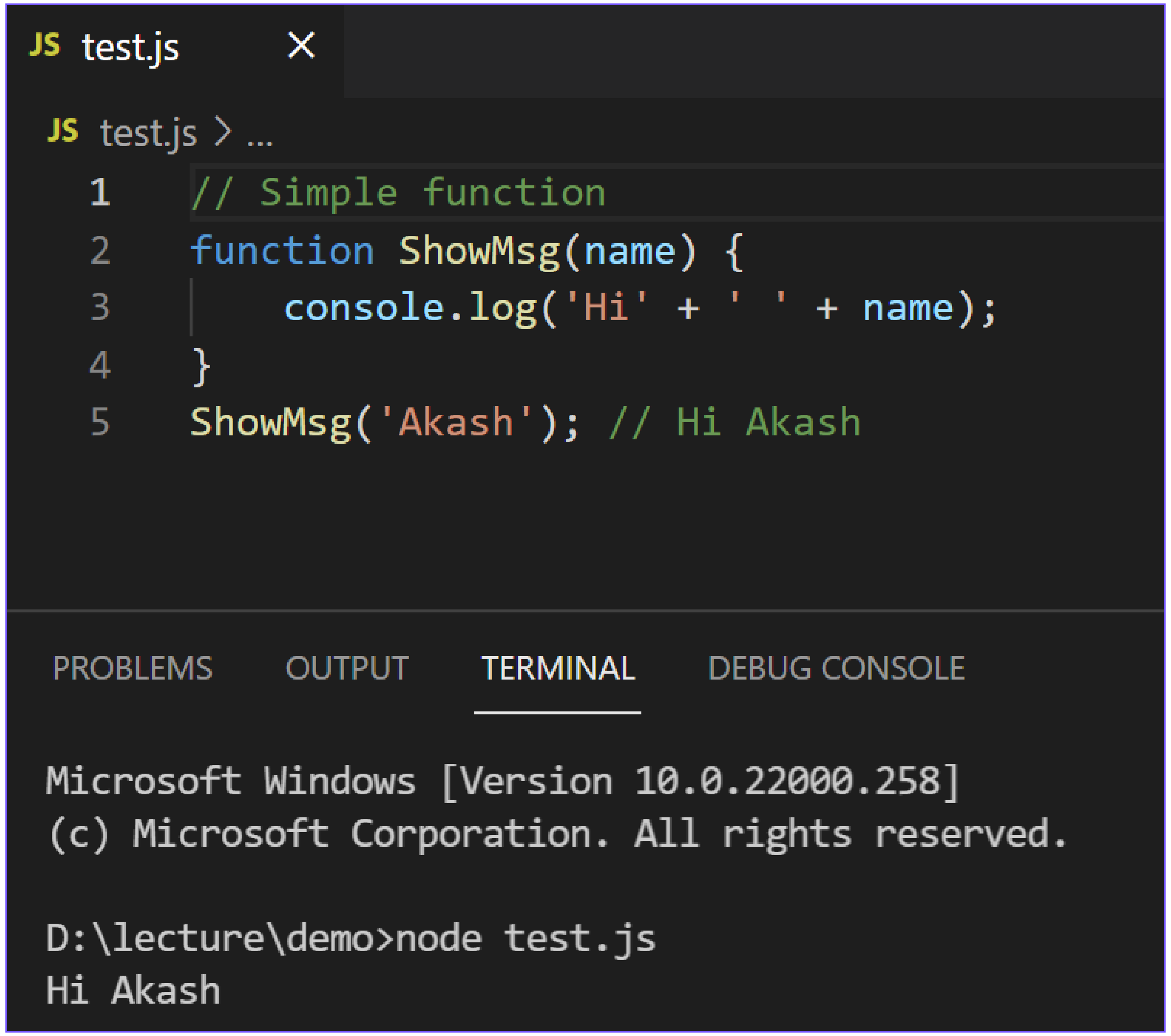
What are callbacks

- 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 test or binary file from the server,
- events,
- the DOM in the browser
- or working with web APIs to fetch data.

Simple Function

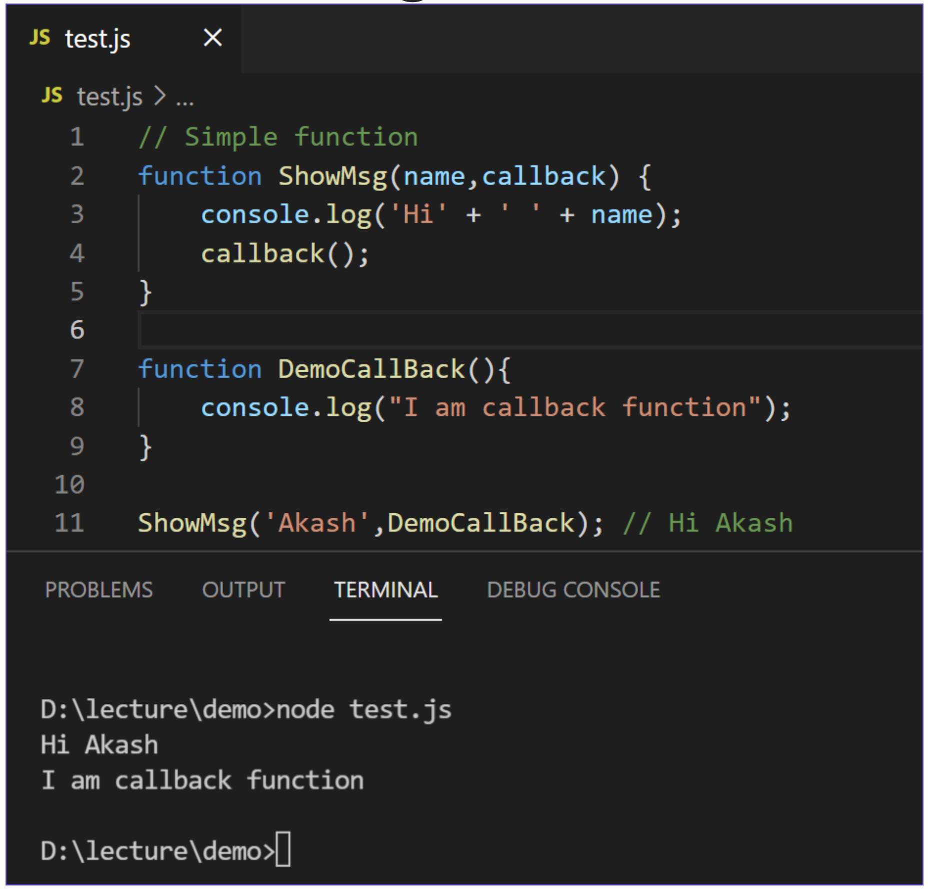


```javascript
// Simple function
function ShowMsg(name) {
    console.log('Hi' + ' ' + name);
}
ShowMsg('Akash'); // Hi Akash
```

```
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.



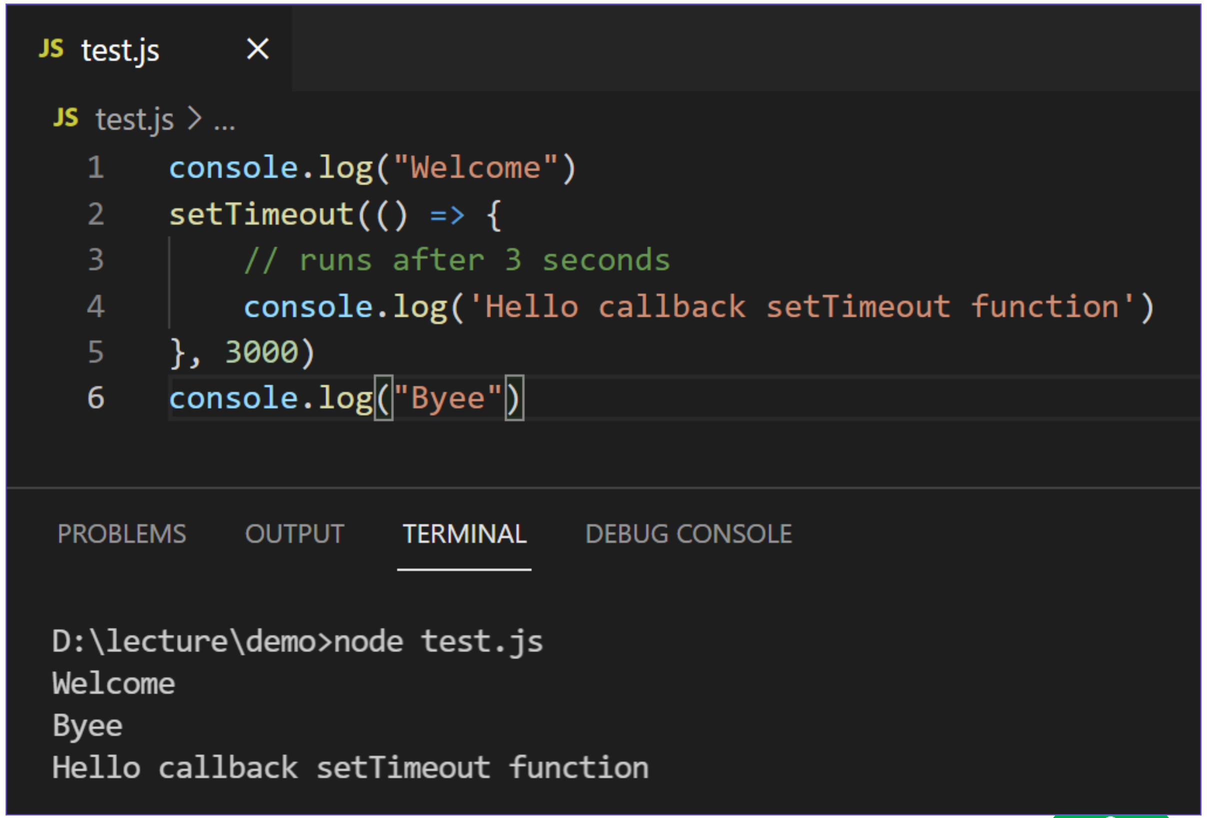
```javascript
// Simple function
function ShowMsg(name,callback) {
    console.log('Hi' + ' ' + name);
    callback();
}

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

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

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

D:\lecture\demo>
```

```javascript
// 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.

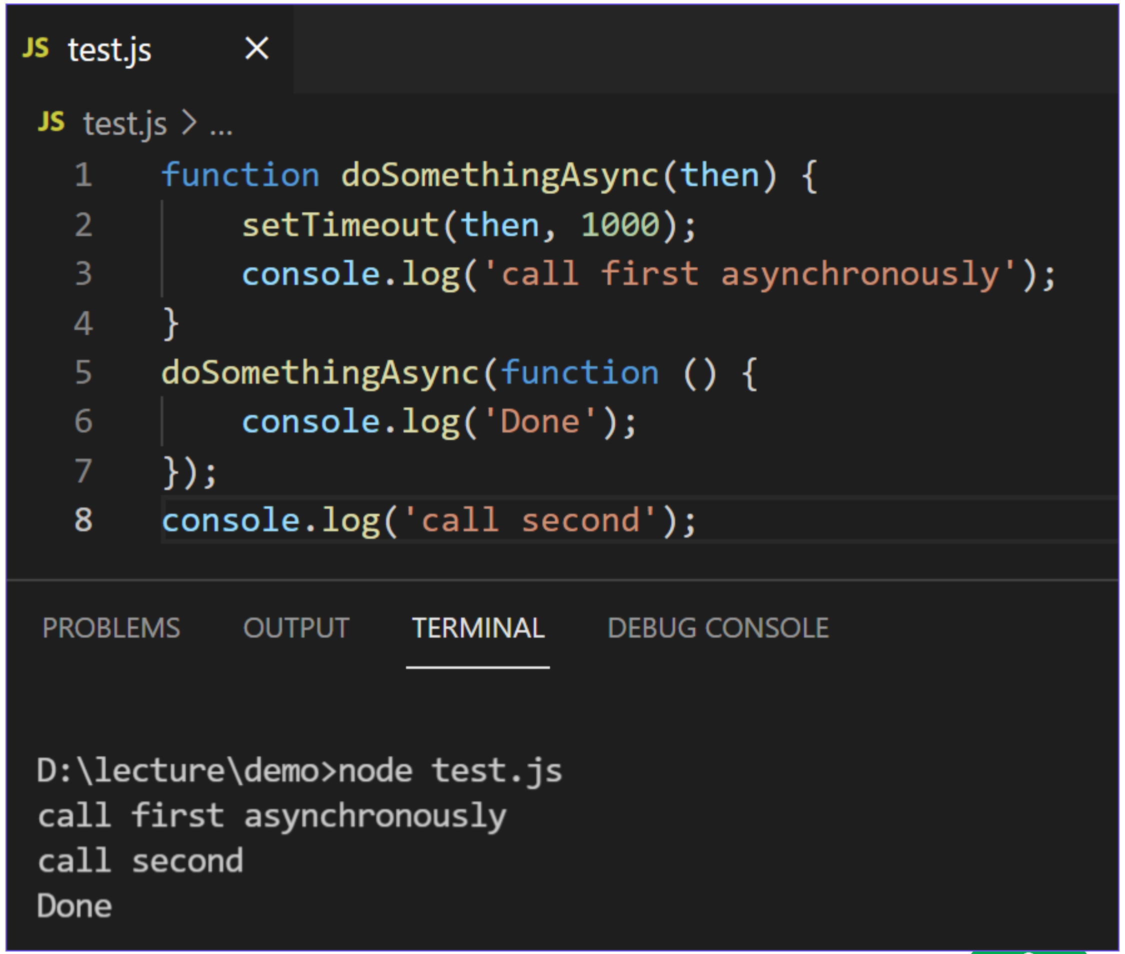


```javascript
console.log("Welcome")
setTimeout(() => {
    // runs after 3 seconds
    console.log('Hello callback setTimeout function')
}, 3000)
console.log("Byee")
```

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

JavaScript Callback Asynchronous

- Callbacks can also be used to execute code asynchronously.

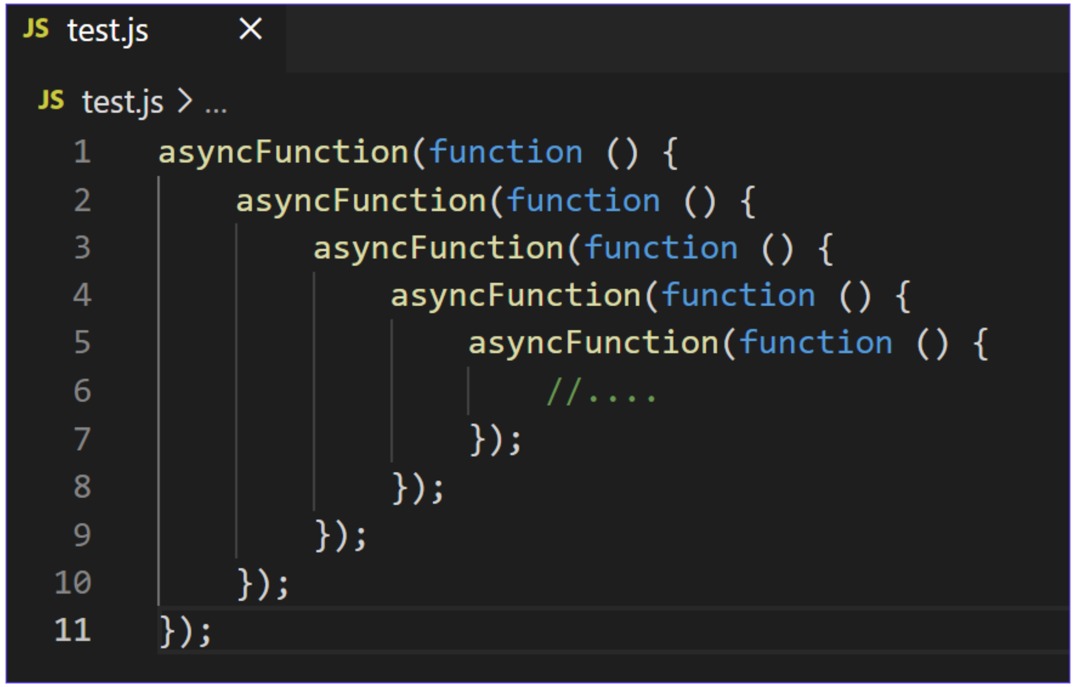


```javascript
function doSomethingAsync(then) {
    setTimeout(then, 1000);
    console.log('call first asynchronously');
}
doSomethingAsync(function () {
    console.log('Done');
});
console.log('call second');
```

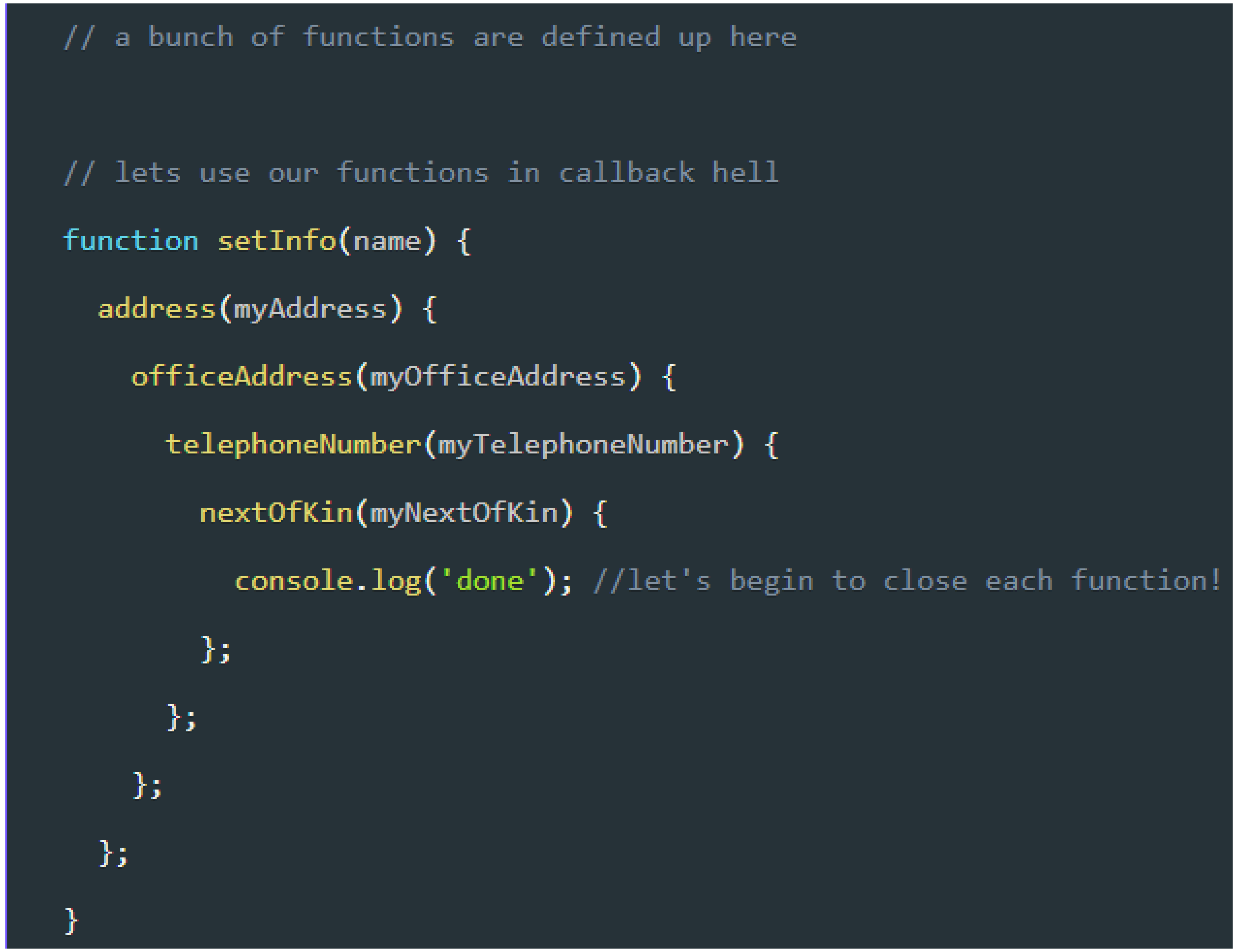
```
D:\lecture\demo>node test.js
call first asynchronously
call second
Done
```

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.**



```javascript
asyncFunction(function () {
    asyncFunction(function () {
        asyncFunction(function () {
            asyncFunction(function () {
                asyncFunction(function () {
                    //....
                });
            });
        });
    });
});
```



```
// 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

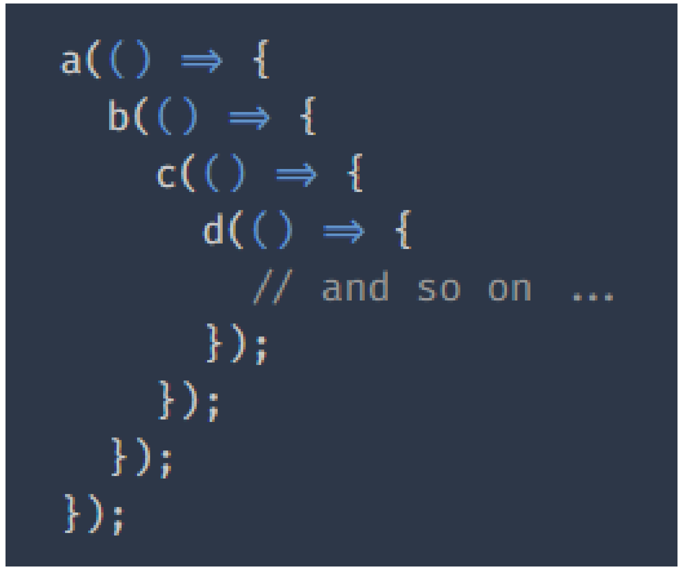
Promises

- 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.

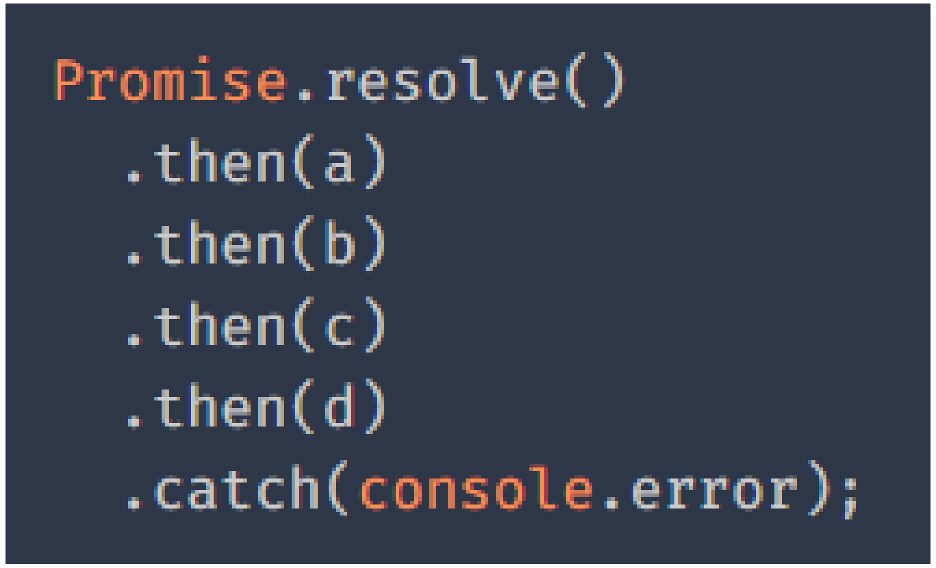
  - Reference : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

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

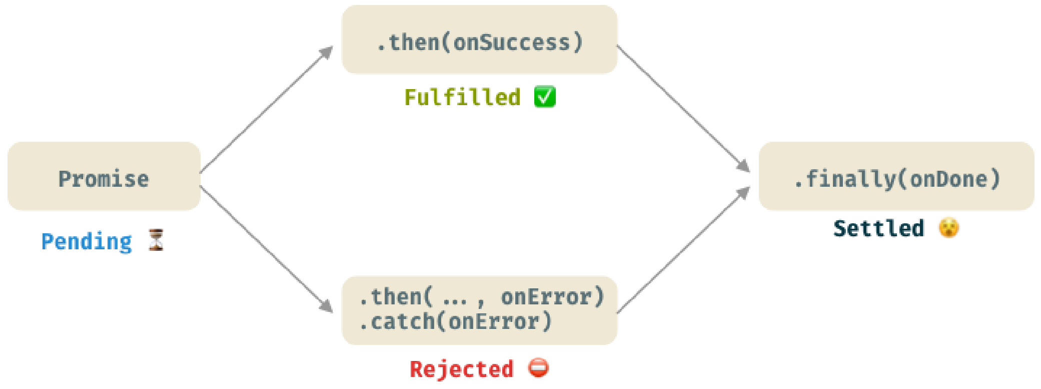
- We can convert callbacks into promises using the Promise constructor.
- The Promise constructor takes a callback with two arguments resolve and reject.
  - **Resolve**: is a callback that should be invoked when the async operation is completed.
  - **Reject**: is a callback function to be invoked when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

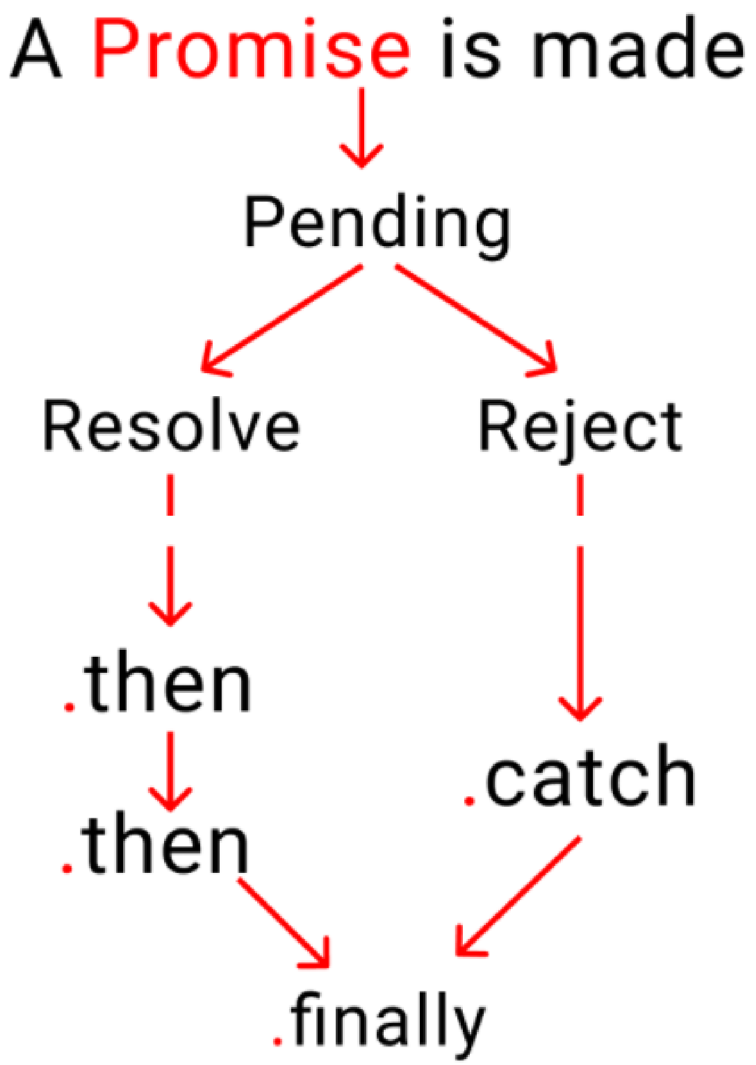
promises just callbacks?

- 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.




A Promise is made
Pending
Resolve
Reject
.then
.then
.catch
.finally

JavaScript Promise Methods

Method	Description
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'));
```

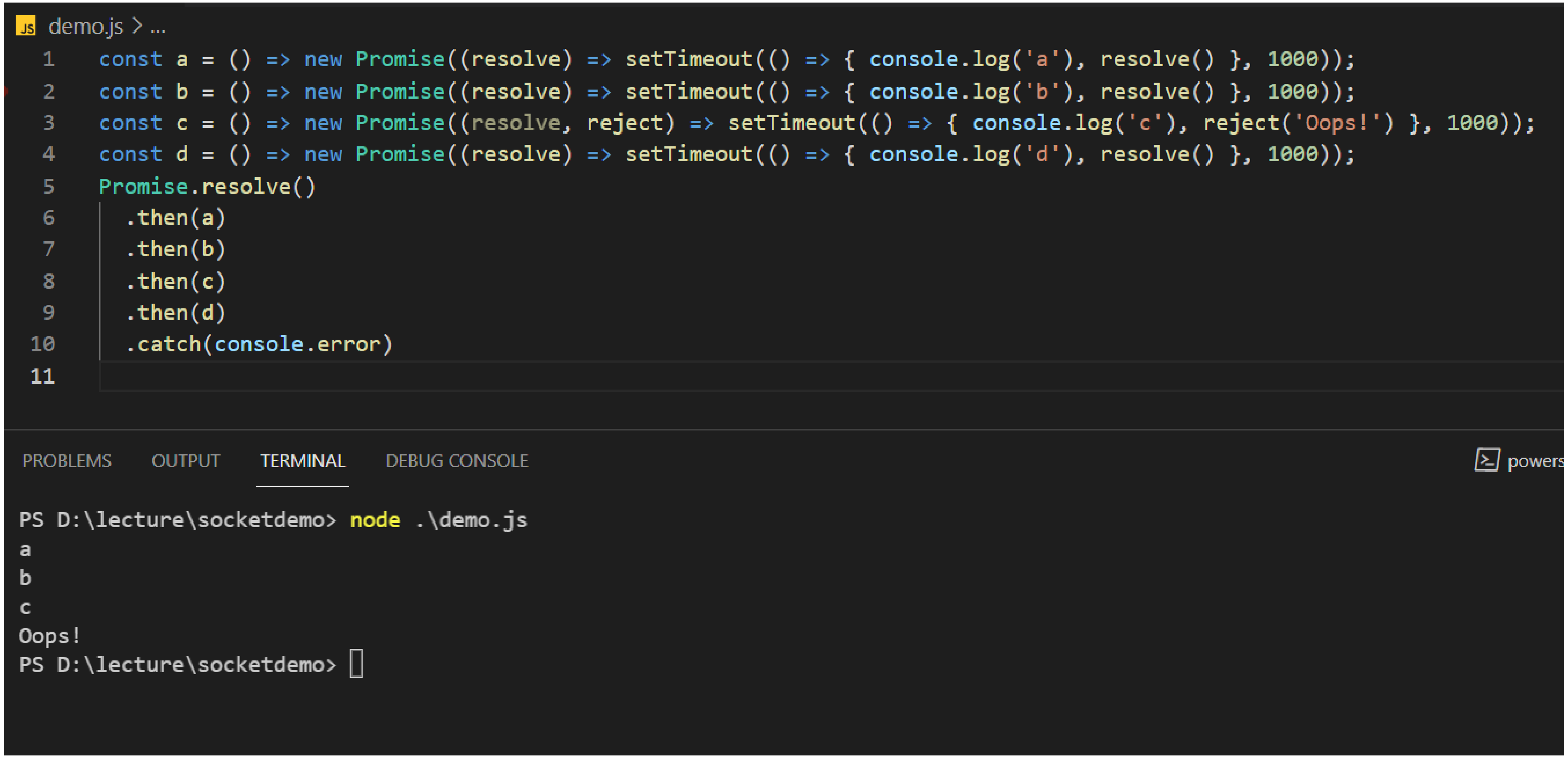
- Promise.resolve immediately resolves the promise as successful. So all the following then are called. The output would be

```
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)
```


demo.js > ...

```javascript
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)
```

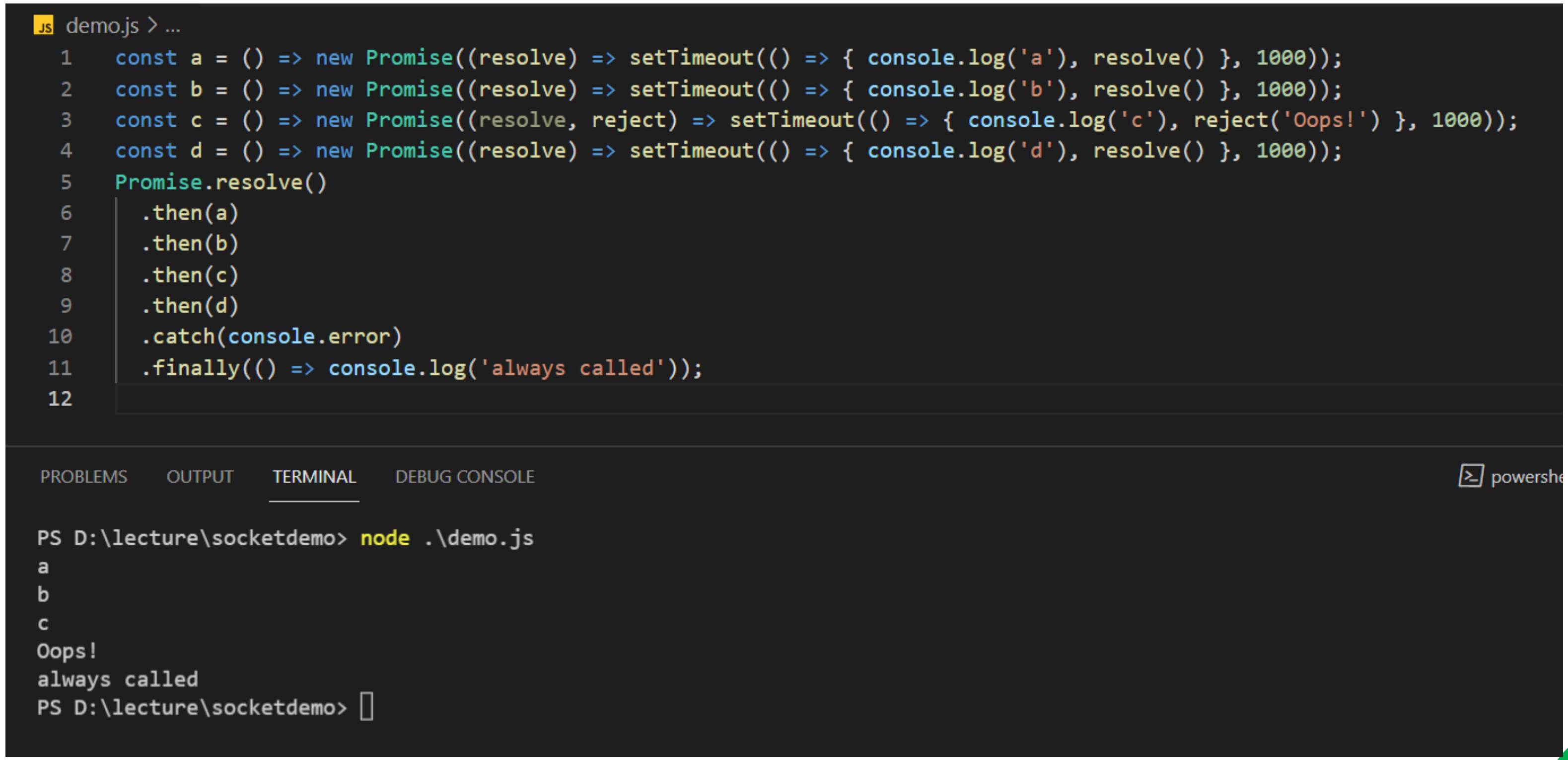
```
PS D:\lecture\socketdemo> node .\demo.js
a
b
c
Oops!
PS D:\lecture\socketdemo>
```

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 > ...

```
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)
  .finally(() => console.log('always called'));

```

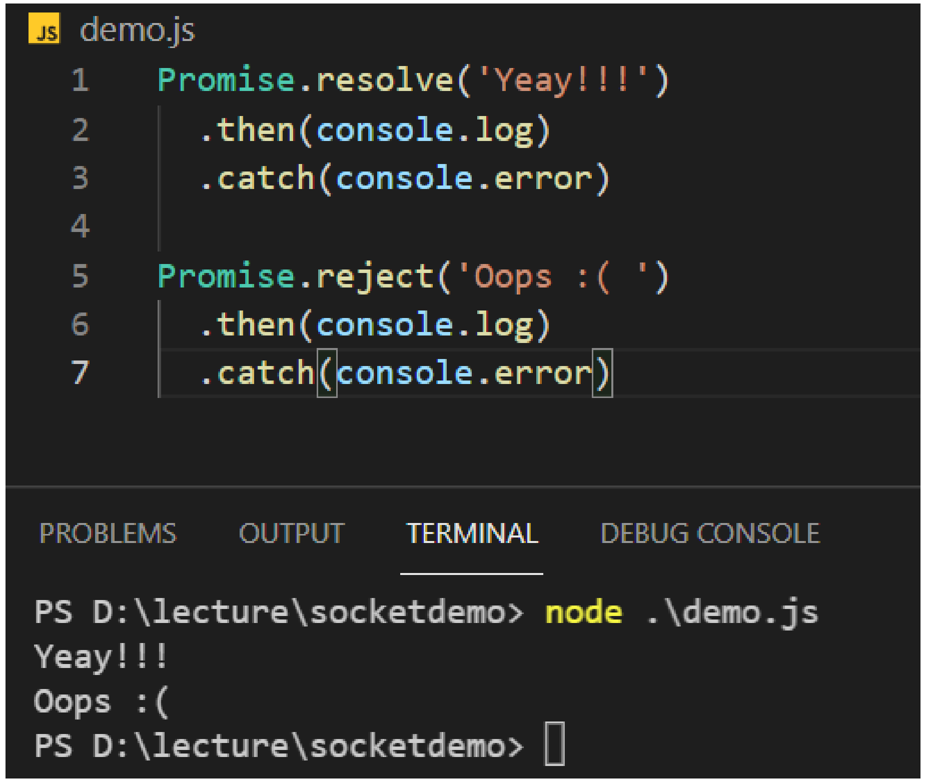
```
PS D:\lecture\socketdemo> node .\demo.js
a
b
c
Oops!
always called
PS D:\lecture\socketdemo> 
```

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.



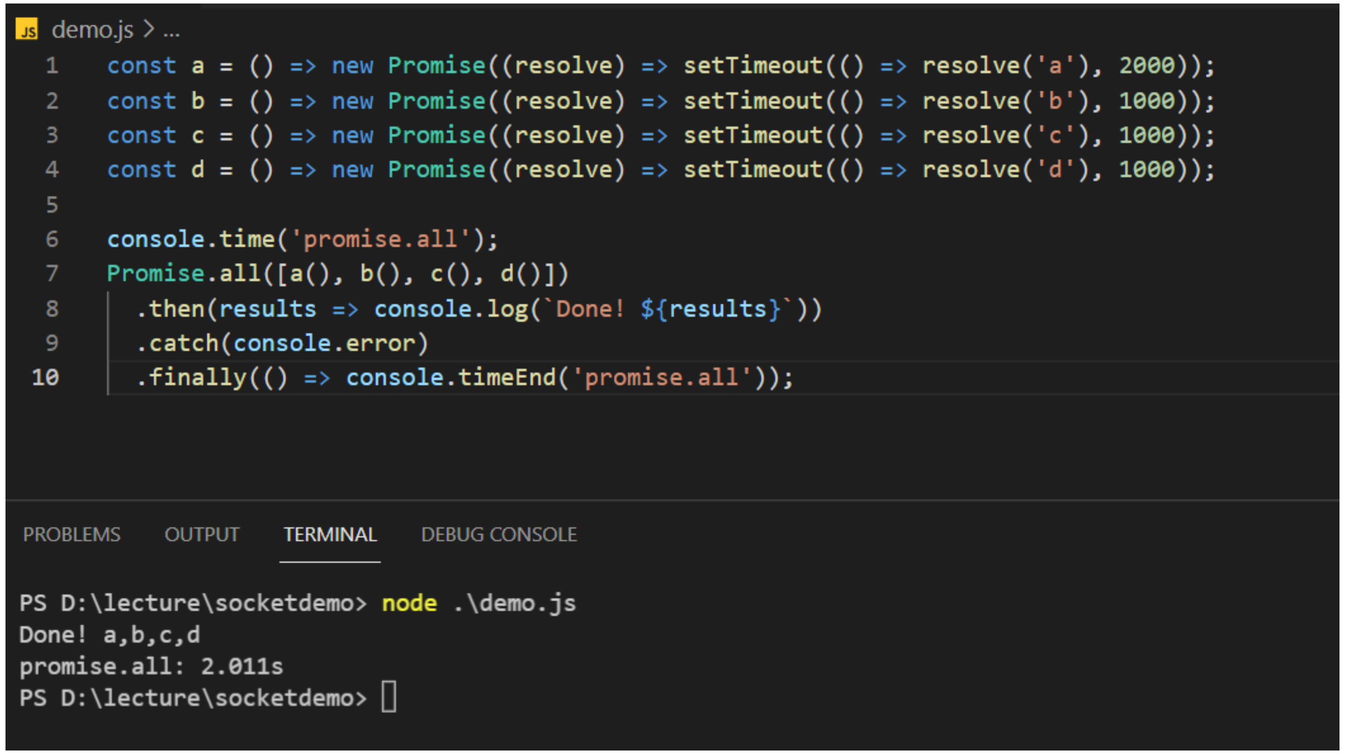
demo.js

```
Promise.resolve('Yeay!!!')
  .then(console.log)
  .catch(console.error)

Promise.reject('Oops :( ')
  .then(console.log)
  .catch(console.error)
```

```
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 > ...

```javascript
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'));
```

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

Example

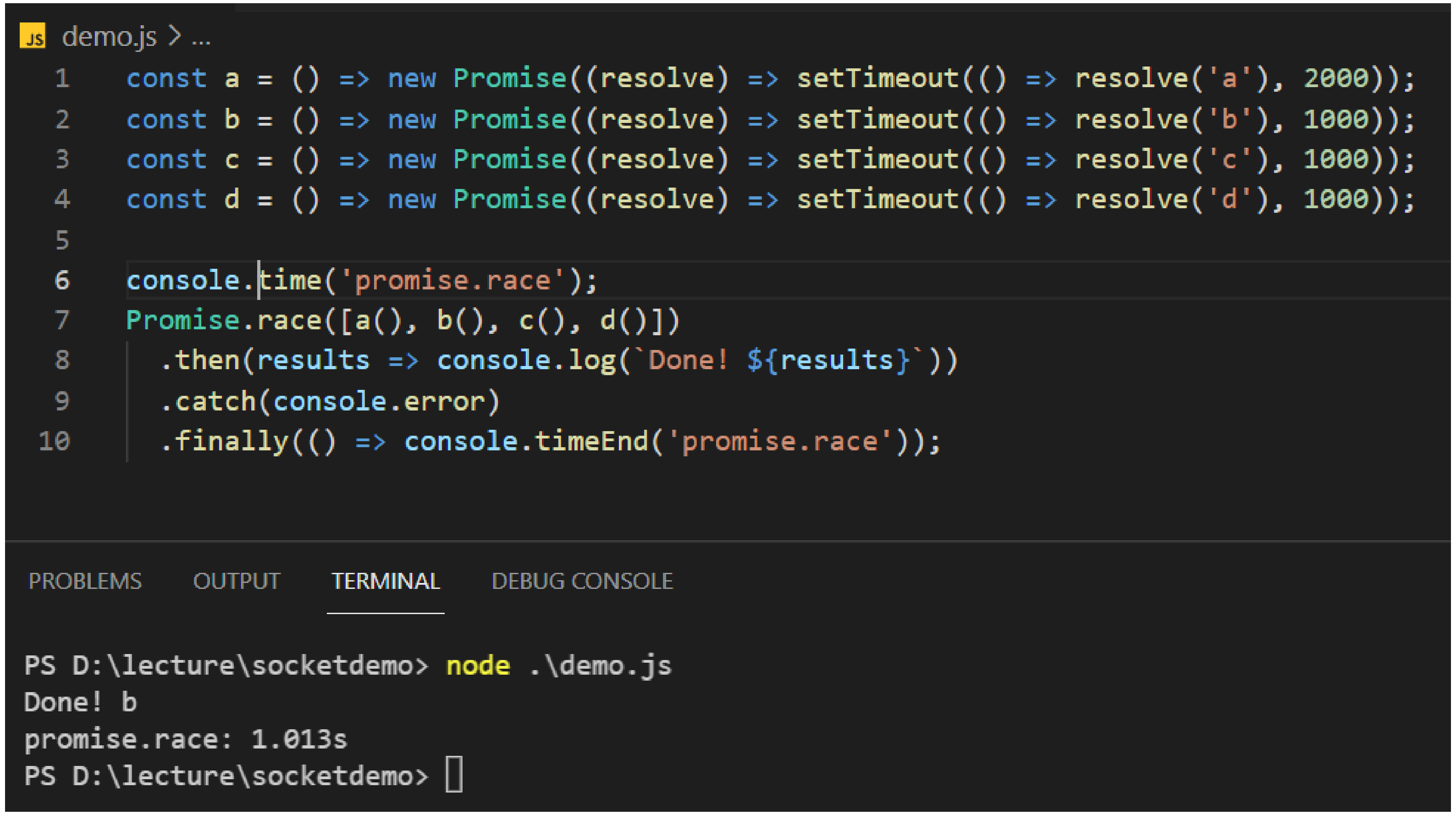
```
const a = () => new Promise((resolve) => setTimeout(() => resolve('a'), 2000));
const b = () => new Promise((resolve) => setTimeout(() => resolve('b'), 1000));
const c = () => new Promise((resolve) => setTimeout(() => resolve('c'), 1000));
const d = () => new Promise((resolve) => setTimeout(() =>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



demo.js > ...

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

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

Async / Await

What is Async/Await?

- It is the newest way to write asynchronous code in JavaScript. Before we used callbacks and promises. Async/await actually builds on top of promises.
- It is non-blocking (just like callbacks and promises).
- **Async/Await** is created to simplify the process of working with and writing chained promises.
- An async function returns the Promise. If the function throws an error, the Promise will be automatically rejected, and if a function returns the value, that means the Promise will be resolved.

Syntax of Async Function

- We need to add the **async** keyword before a **function.**

```
// Normal Function
function add(a,b){
  return a + b;
}
// Async Function
async function add(a,b){
  return a + b;
}
```

Async functions

- To create an async function all we need to do is add the async keyword before the function definition, like this:

```
async function asyncFunc() {
  return "Hey!";
}
```

- The one thing you need to know about async functions is that; they always returns a promise.

Await

- The await keyword can only be used within an async block, otherwise it'll throw a syntax error. This means you cannot use await in the top level of our code, basically, don't use it by itself.

- When do we use it?
  - If we have an asynchronous function inside of an async block.
  - So let's say we need to fetch some data from our server and then use that data within our async block.
  - We will use **await to pause the function execution** and resume after the data comes in. For example;

- Await is simply a more elegant way to write a promise within an async function. It improves readability immensely and hence the reason we use it.

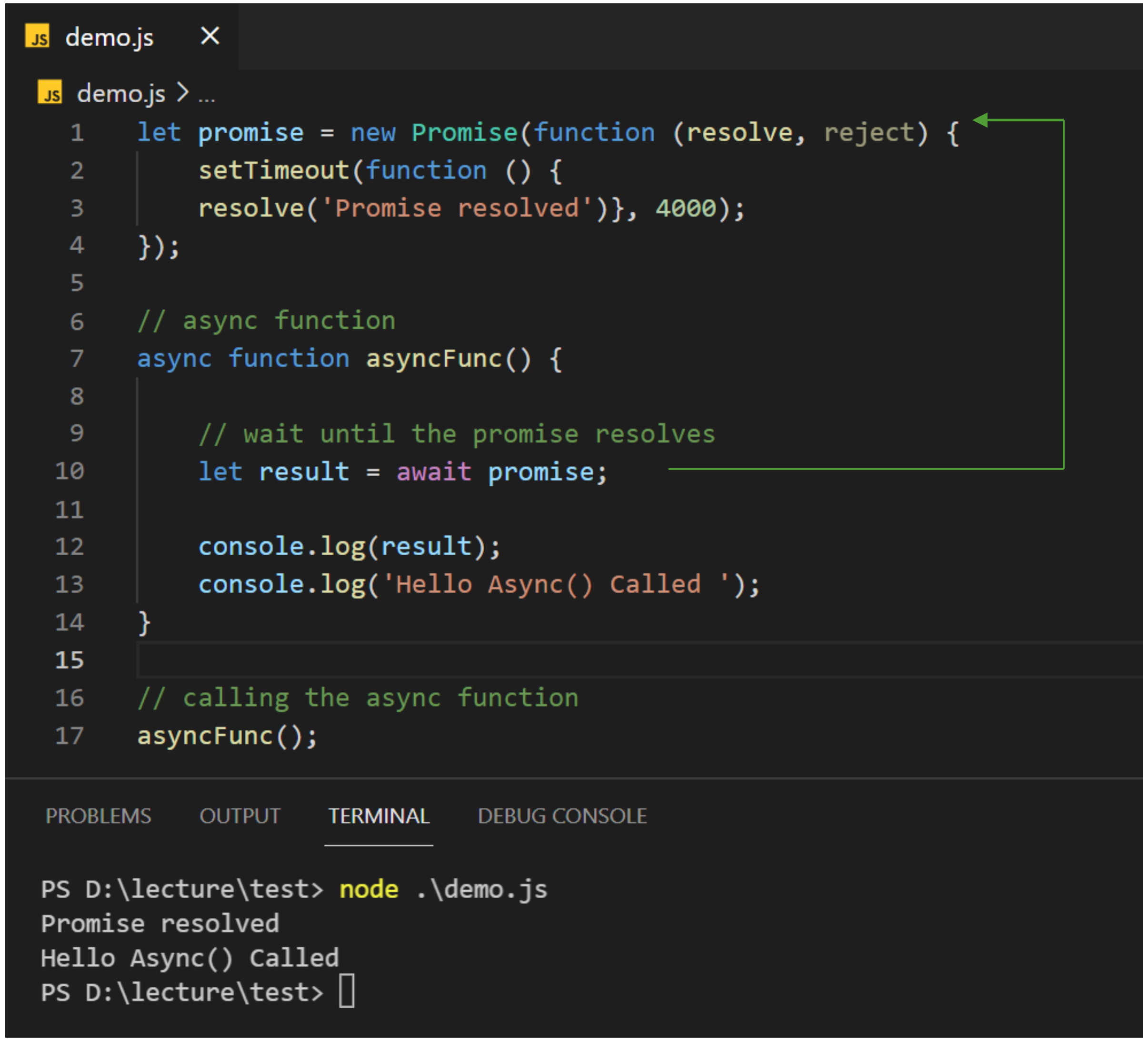
Example :

```
async function asyncFunc() {
  // fetch data from a url endpoint
  const data = await axios.get("/some_url_endpoint");
  return data;
}
```

Promise Based :

```
async function asyncFunc() {
  let data;
  // fetch data from a url endpoint

  axios.get("/some_url_endpoint")
    .then((result) => {
      data = result
    });
  return data;
}
```

Example



```javascript
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {

    // wait until the promise resolves
    let result = await promise;

    console.log(result);
    console.log('Hello Async() Called ');
}

// calling the async function
asyncFunc();
```

```
PS D:\lecture\test> node .\demo.js
Promise resolved
Hello Async() Called
PS D:\lecture\test> 
```

Example

```
let promise = new Promise(function (resolve, reject) {
   setTimeout(function () {
   resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {

   // wait until the promise resolves
   let result = await promise;

   console.log(result);
   console.log('Hello Async() Called ');
}

// calling the async function
asyncFunc();
```

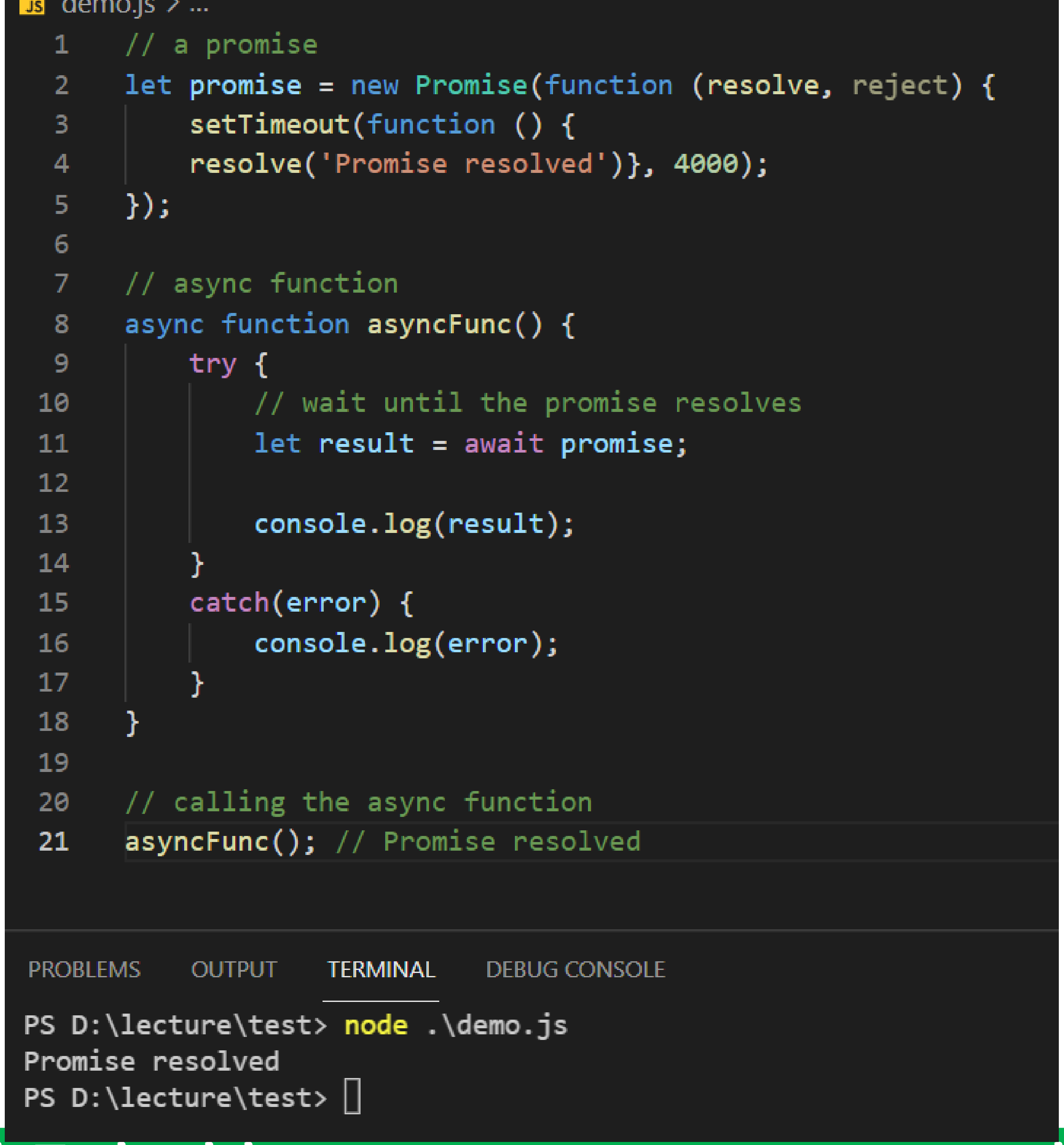
- Let's assume we have a couple of asynchronous functions within our async block. Instead of chaining promises we could do this, which is much cleaner:

```
async function asyncFunc() {
  // fetch data from a url endpoint
  const response = await axios.get("/some_url_endpoint");
  const data = await response.json();

  return data;
}
```

Error handling

- The most common way to handle errors when using async-await, good old try-catch. All you need to do is encapsulate your code in a try block and handle any errors that occur in a catch.

```
async function asyncFunc() {
  try {
    // fetch data from a url endpoint
    const data = await axios.get("/some_url_endpoint");

    return data;
  } catch(error) {
    console.log("error", error);
    // appropriately handle the error
  }
}
```



```javascript
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {
    try {
        // wait until the promise resolves
        let result = await promise;

        console.log(result);
    }
    catch(error) {
        console.log(error);
    }
}

// calling the async function
asyncFunc(); // Promise resolved
```

```
PS D:\lecture\test> node .\demo.js
Promise resolved
PS D:\lecture\test> 
```

Async on class methods

Class methods can be async

```
class Example {
  async asyncMethod() {
    const data = await axios.get("/some_url_endpoint");
    return data
  }
}

const exampleClass = new Example();
  exampleClass.asyncMethod().then(//do whatever you want with the result)
```

Await - Promise.all

- If we have multiple promises we could use Promise.all with await.

```
async function asyncFunc() {
  const response = await Promise.all([
    axios.get("/some_url_endpoint"),
    axios.get("/some_url_endpoint")
  ]);
  ...
}
```

Get Exclusive Video Tutorials

www.aptutorials.com

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

Connect With Me

Akash Padhiyar
#AkashSir

www.akashsir.com
www.akashtechnolabs.com
www.akashpadhiyar.com
www.aptutorials.com

# Social Info

Akash.padhiyar
Akashpadhiyar
Akash_padhiyar
+91 99786-21654

#Akashpadhiyar
#aptutorials

Get More Details

www.akashsir.com

If You Liked It !

Rating Us Now

Just Dial

https://www.justdial.com/Ahmedabad/Akash-Technolabs-Navrangpura-Bus-Stop-Navrangpura/079PXX79-XX79-170615221520-S5C4_BZDET

Sulekha

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