WEB TECHNOLOGIES

Callback and Promises

Prof. Pavan A C
Department of
Computer Science and Engineering

Acknowledgement:
Teaching Assistants(Harini B and Chandana MS)

Callbacks
Introduction

- A callback is a function passed as an argument to another function
- This technique allows a function to call another function
- They will be called asynchronously based on timer or other events
- These function references are called **Callback functions**
- As seen in setInterval, setTimeout and addEventListener, a function accepts a function reference as an argument.

- Example:
  - div.addEventListener(“keypress”, function(){ … });

Callbacks

Introduction

```
function greet(name, callback) {
    console.log("Hello, " + name);
    callback();
}
function sayBye() {
    console.log("Goodbye!");
}
greet("Ajay", sayBye);
```

In this example, the sayBye function is passed as an argument to the greet function and is then invoked from within greet. This demonstrates how functions in JavaScript can be passed as arguments to other functions—also known as **callbacks**. The greet function can accept **any function** as a callback, and it will execute that function after greeting the user.

Callbacks
Introduction

Synchronous Calls

- Code executes one statement at a time, blocking further execution until the current statement finishes.

```
console.log("One");
console.log("Two");
console.log("Three");
// Output: One
//        Two
//        Three
```

Asynchronous Calls

- Some tasks (like timers or network requests) don’t complete right away. Asynchronous calls let your code schedule work to happen later, without blocking.

Example:

```
console.log("One");
setTimeout(function() {
  console.log("Two");
}, 1000);
console.log("Three");
```

```
// Output:
    One
    Three
// (waits 1 second...)
   Two
```

Callback Hell

Callback hell (also called the "pyramid of doom") occurs when callbacks are nested within other callbacks many levels deep. This makes code hard to read, understand, and maintain.

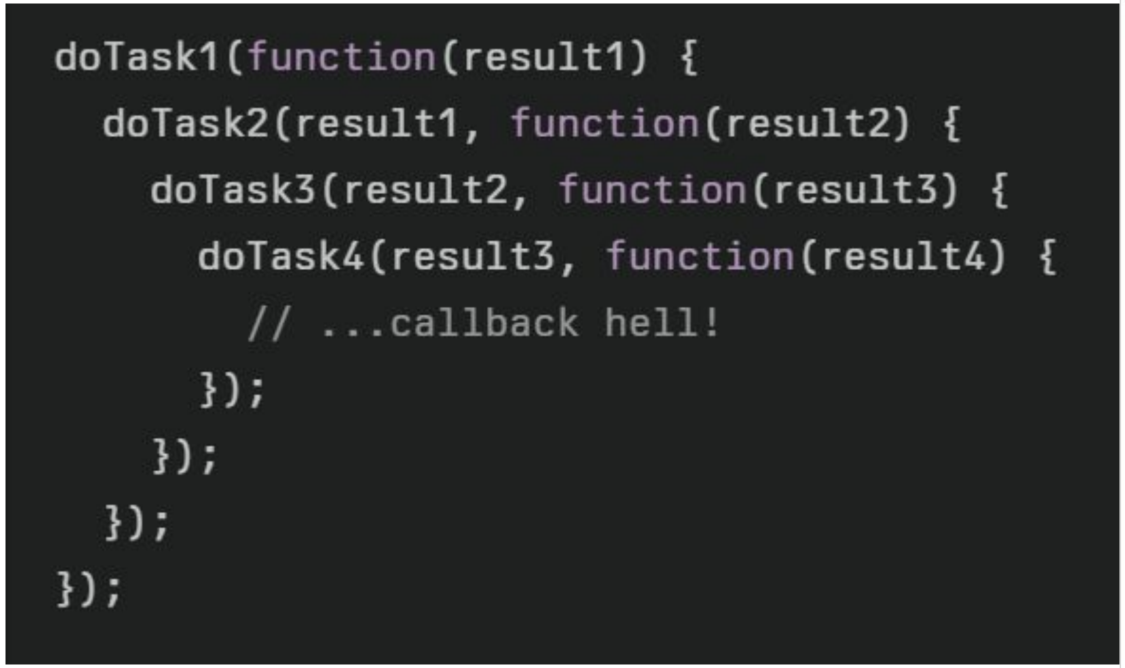
Why does it happen?

- Asynchronous tasks are often dependent on the results of previous tasks.
- If you write each dependent task inside the previous callback, you end up with multiple levels of nesting.
- This nesting makes code:
  a. Difficult to follow.
  b. Harder to debug.

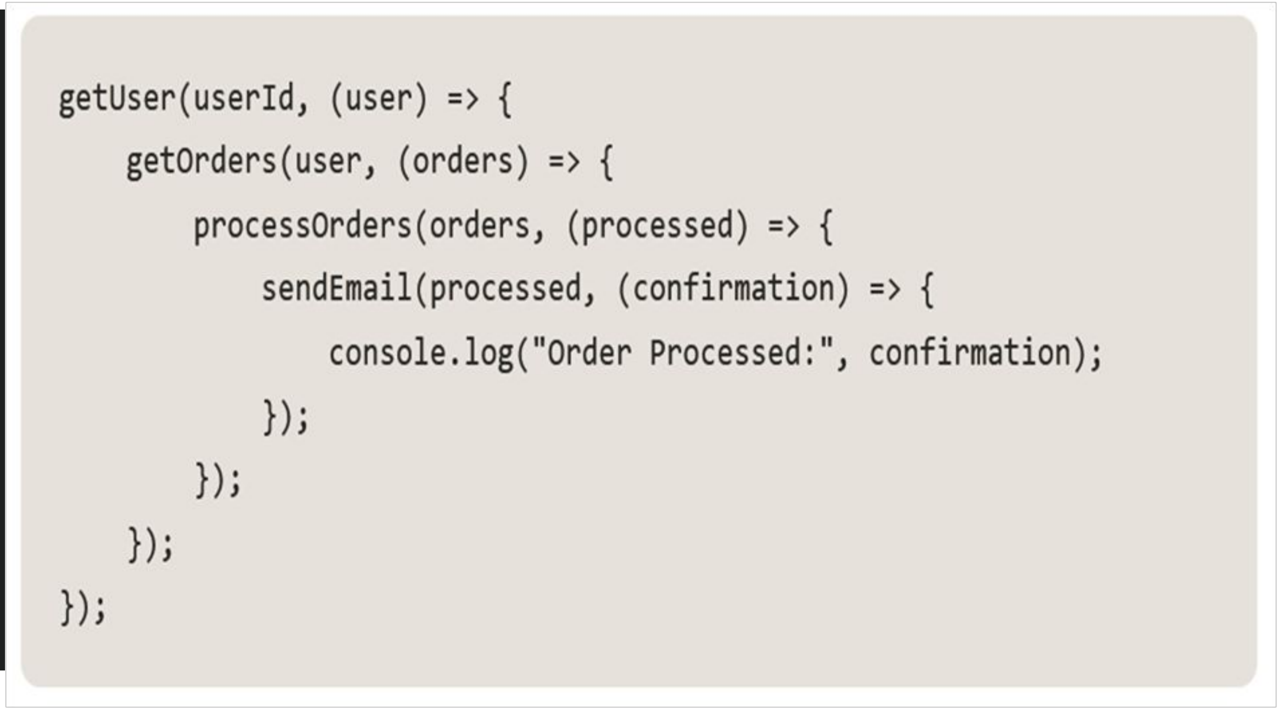
Promises

Introduction

Callback Hell



```
doTask1(function(result1) {
  doTask2(result1, function(result2) {
    doTask3(result2, function(result3) {
      doTask4(result3, function(result4) {
        // ...callback hell!
      });
    });
  });
});
```



```
getUser(userId, (user) => {
    getOrders(user, (orders) => {
        processOrders(orders, (processed) => {
            sendEmail(processed, (confirmation) => {
                console.log("Order Processed:", confirmation);
            });
        });
    });
});
```

Promises

Introduction

I Promise a Result!"

"Producing code" is code that can take some time

"Consuming code" is code that must wait for the result

A Promise is a JavaScript object that links producing code and consuming code

A JavaScript Promise object contains both the producing code and calls to the consuming code:

Promises

A Promise is a special JavaScript object used to handle asynchronous operations. It represents a value that may be available now, later, or never (in case of an error). Promises allow more readable async code compared to deeply nested callbacks (callback hell).

A Promise is a placeholder for the result of an asynchronous task.

- A Promise is in one of three states:
  - Pending: The operation is ongoing.
  - Fulfilled: The operation completed successfully.
  - Rejected: The operation failed.

Promises

Introduction

- Promise is a JavaScript object with a value that may not be available at the moment when the code line executes.

- These values are resolved at some point in the future.

While a Promise object is "pending" (working), the result is undefined.
When a Promise object is "fulfilled", the result is a value.
When a Promise object is "rejected", the result is an error object.
You cannot access the Promise properties state and result.
You must use a Promise method to handle promises.

Promises

Introduction

- A promise is used to handle the asynchronous result of an operation.
- With Promises, we can defer execution of a code block until an async request is completed.
- The Promise object is created using the new keyword and contains the promise; this is an executor function which has a **resolve** and a **reject** callback
- Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

```
const promise = new Promise(function(resolve, reject) {
    // promise description
});
```

Promises

Introduction: Syntax for Callback and promises.

```
const myPromise = new Promise(function(resolve, reject) {
  // Simulate async operation
  setTimeout(function() {
    let success = true; // Try changing to false to see rejection handling
    if (success) {
      resolve("Operation succeeded!");
    } else {
      reject("Operation failed.");
    }
  }, 1000);
});

// Using the Promise
myPromise
  .then(function(result) {
    console.log(result); // Runs if resolved: "Operation succeeded!"
  })
  .catch(function(error) {
    console.log(error); // Runs if rejected: "Operation failed."
  });
```

Promises

Introduction

```
// Promise constructor
let promise = new Promise(function(resolve, reject) {
     const x = "apple";
     const y = "apple"

if (x === y) {
        resolve();
     } else {
        reject();
     }
});

// Consuming the Promise
promise
   .then(function () {
      console.log('Successful');
   })
   .catch(function () {
      console.log('Some error has occured');
   });
```

- Promise constructor takes only one argument which is a callback function (and that callback function is also referred as an anonymous function too).
- Callback function takes two arguments, resolve and reject. Perform operations inside the callback function and if everything went well then call resolve.

Promises

Example

```
var weather;
const date = new Promise(
      function(resolve, reject) {
        weather = true; //usually a API call
        if (weather) {
                const dateDetails = {
                        name: 'Cubana Restaurant',
                        location: '55th Street',
                        table: 5
                };
                resolve(dateDetails)
        } else {
                reject(new Error('Bad weather'))
        }
       }
);
```

```
date
.then(function(done) {
        // This runs when resolve() is called
        console.log('We are going on a date!')
        console.log(done)
        })
.catch(function(error) {
        // This runs when reject() is called
        console.log(error.message)
        })
```

Callback and Promises
Comparison

- Callbacks and Promises are not the same
- Callbacks are function passed to another function as a reference
- Chaining of Callbacks can be clumsy and lead to **Callback Hell**

- Promises use Callbacks and more elegant than Callbacks
- Chaining of Promises is supported

Callback and Promises
MCQ

1)Complete the code so that sayBye is called only after greet prints the greeting:

```
function greet(name, callback) {
  console.log("Hello, " + name);
  // _______
}
function sayBye() {
  console.log("Goodbye!");
}
greet("Ajay", sayBye);
```

A) sayBye();

B) sayBye;

C) callback();

D) callback;

Callback and Promises

MCQ

2) What will be the output of the following code?

```
console.log("First");
setTimeout(() => { console.log("Second"); }, 50);
console.log("Third");
```

A) First Second Third
B) Second First Third
C) First Third Second
D) Third First Second

3) Which of the following statements about Promises is correct?

A) A fulfilled promise can never become pending again
B) A rejected promise will execute the code in .then()
C) A promise can only be in two states: fulfilled or rejected
D) The only way to handle errors in promises is with try...catch blocks

Callback and Promises

MCQ

Answer 1 : C) callback();

Answer 2: C) First Third Second

Answer 3: A) A fulfilled promise can never become pending again

PES UNIVERSITY
CELEBRATING 50 YEARS

THANK YOU

Prof. Pavan A C
Department of Computer Science and Engineering