upGrad

#RahoAmbitious

Full Stack Software Development

upGrad

Course: JavaScript and Server-Side Communication

Lecture On: Callbacks and Promises

Instructor: Siddhesh Prabhugaonkar

upGrad

In the previous class, we covered…

- Introduction to ES6
- Introduction to let and const
- Arrow functions of ES6
- map(), filter() and reduce()
- Spread operator & Rest parameters
- Template literals
- Import and Export

Poll 1 (15 Sec)

What would be the output of the following code?

1. ["tea", "ate", "eat"]
2. ["coffee"]
3. ["milk"]
4. ["coffee", "milk"]

```
const words = ['tea', 'coffee', 'milk', 'ate', 'eat'];

const result = words.filter(word => word.length > 3);

console.log(result);
```

Poll 1 (Answer)

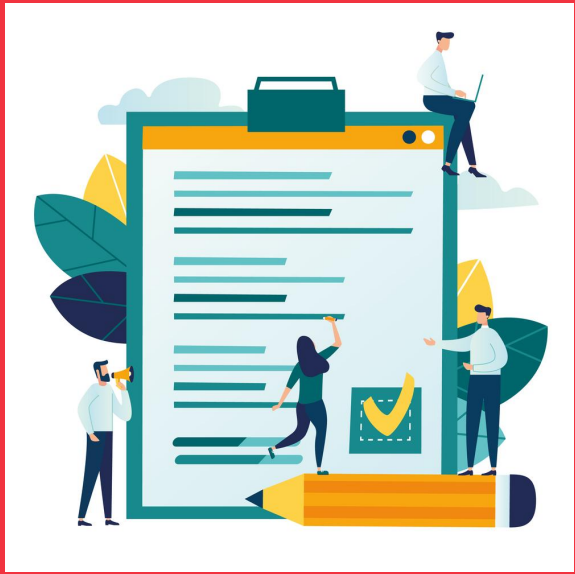
What would be the output of the following code?

```
const words = ['tea', 'coffee', 'milk', 'ate', 'eat'];

const result = words.filter(word => word.length > 3);

console.log(result);
```

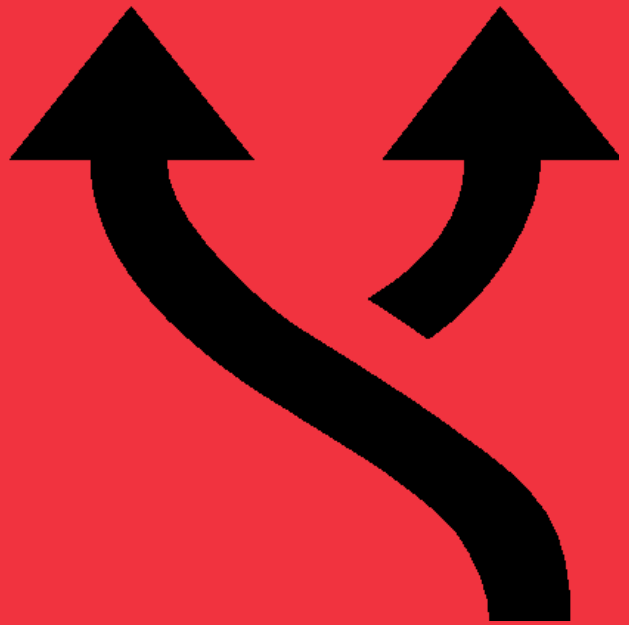
1. ["tea", "ate", "eat"]
2. ["coffee"]
3. ["milk"]
4. **["coffee", "milk"]**

Today's Agenda

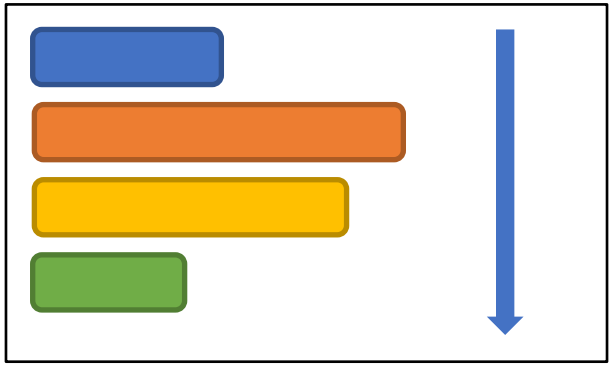
- JavaScript Callbacks
- JavaScript Promises



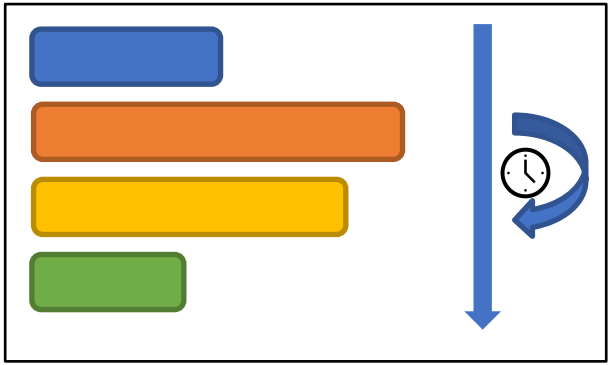
Asynchronous JavaScript



Introduction to Asynchronous JavaScript



Synchronous JavaScript



Asynchronous JavaScript

- JavaScript (JS) is, by default, synchronous in nature. It is a single-threaded programming language, which means that only one thing can happen at a time.
- Suppose you are requesting data from a server. Depending on the network speed and the size of the request, it will take some time to process the request and then send across the response. In such a case, the JS execution will be stopped until the request is resolved.
- So, to prevent the case given above from impacting your JS code and to continue waiting for the request without blocking the main thread of code, you can use the asynchronous JS.

setTimeout()

```
let showAlert = () => {
    alert("Hello World");
}

setTimeout(showAlert, 3000);

alert("Goodbye World");
```

- The setTimeout() method is an example of asynchronous JavaScript.
- In **setTimeout()** function, the first parameter is the function whose code will be executed and the second parameter will be the time in milliseconds.
- In the example given above, although the alert "Goodbye World" is placed after the setTimeout() function, it will be executed first. Then, after 3,000 milliseconds or 3 seconds, the showAlert() function will be executed and "Hello World" will be displayed in an alert box.
- You can see from the output of the above code that even though the second statement where setTimeout method is mentioned is blocked for 3 seconds, the third statement is executed without waiting for the second statement to finish execution. This is an example of asynchronous nature of JavaScript.

Asynchronous JavaScript

upGrad

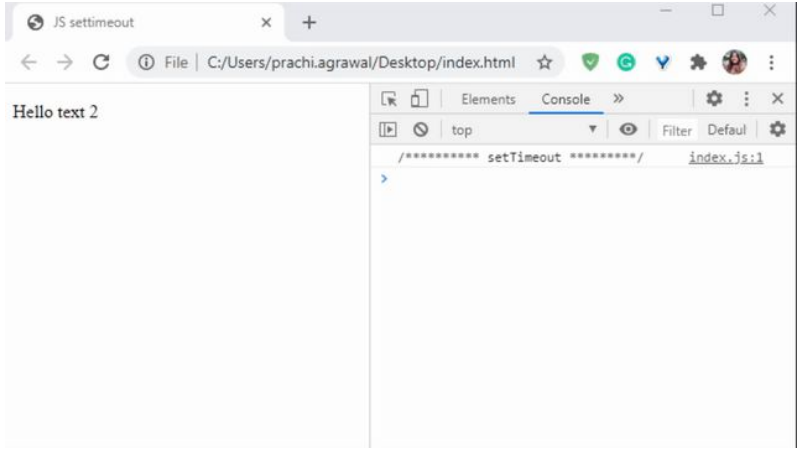
setTimeout()

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
content="width=device-width,
initial-scale=1.0">
    <title>JS settimeout</title>
</head>
<body>
    <p id="text-1"></p>
    <p id="text-2"></p>
    <script src="index.js"></script>
</body>
</html>
```

index.js

```
console.log("/********* setTimeout ********/");
setTimeout(() => {
  document.getElementById("text-1").innerHTML = "Hello text 1";
}, 2000);
document.getElementById("text-2").innerHTML = "Hello text 2";
```



Poll 2 (15 Sec)

Which of the following statements about JavaScript is false?

1. It is a single-threaded programming language.
2. You can write synchronous as well as asynchronous code in JavaScript.
3. In synchronous programming, you can execute tasks in parallel, which, at times, can give the output in an order different from the one in which the code is written.
4. In asynchronous programming, the code that is to be run in the background is sent to the Web APIs.

Poll 2 (Answer)

Which of the following statements about JavaScript is false?

1. It is a single-threaded programming language.
2. You can write synchronous as well as asynchronous code in JavaScript.
3. **In synchronous programming, you can execute tasks in parallel, which, at times, can give the output in an order different from the one in which the code is written.**
4. In asynchronous programming, the code that is to be run in the background is sent to the Web APIs.

Poll 3 (15 Sec)

In what order will the numbers 10–40 be logged to the console when the code given in the adjoining screenshot is executed?

```
(() => {
    console.log(10);
    setTimeout(() => {console.log(20)}, 1000);
    setTimeout(() => {console.log(30)}, 0);
    console.log(40);
})();
```

1. 10
   20
   30
   40
2. 10
   40
   30
   20
3. 10
   30
   40
   20

Poll 3 (15 Sec)

In what order will the numbers 10–40 be logged to the console when the code given in the adjoining screenshot is executed?

```
(() => {
    console.log(10);
    setTimeout(() => {console.log(20)}, 1000);
    setTimeout(() => {console.log(30)}, 0);
    console.log(40);
})();
```

1. 10
   20
   30
   40
2. **10**
   40
   30
   20
3. 10
   30
   40
   20

Hands-On Exercise (5 mins)

upGrad

Write a setTimeout() method in ES6 syntax which is executed after 5 seconds and prints the message "Welcome to upGrad" on the console.

The stub code is provided here.

The solution is provided here.

JavaScript Callbacks

Function Passed as Argument

upGrad

- JavaScript callback functions are a great way to execute a part of code only after the execution of the other part of the code is complete.
- In the example given below, the function add is called with arguments 1 and 2, and the function display.
- The numbers are added first and then the callback function display is called.

```
let add = (num1, num2, callback) => {
     let result = num1 + num2;
     callback(result);
}

let display = answer => alert(`The sum is ${answer}.`);

add(1, 2, display);
```

Callback Functions

```
let concept;

const getConcept = () => {
//get data from database (here we are mocking this functionality by
calling setTimeout)
   setTimeout(() => {
      concept = "Callback";
    }, 2000);
}
const print = () => {
   console.log(`JavaScript ${concept}`);
}

getConcept();
print();
//JavaScript undefined
```

Without using callback function

Callback Functions

```
let concept;

const getConcept = (callback) => {
//get data from database (here we are mocking this functionality by
calling setTimeout)
  setTimeout(() => {
     concept = "Callback";
     callback();
  }, 2000);
}

const print = () => {
   console.log(`JavaScript ${concept}`);
}
getConcept(print);

//JavaScript Callback
```

Using the callback function

Poll 4 (15 Sec)

Which of the following statements about callbacks in JavaScript is true?
(Note: More than one option may be correct.)

1. A callback can be used in synchronous and asynchronous programming in JavaScript.
2. A callback can only be invoked at the end of all the tasks in the callee function.
3. You can use callbacks to mark the completion of a task.
4. Callbacks cannot be written as anonymous functions passed inline as an argument to the caller function.

Poll 4 (Answer)

Which of the following statements about callbacks in JavaScript is true?
(Note: More than one option may be correct.)

1. **A callback can be used in synchronous and asynchronous programming in JavaScript.**
2. A callback can only be invoked at the end of all the tasks in the callee function.
3. **You can use callbacks to mark the completion of a task.**
4. Callbacks cannot be written as anonymous functions passed inline as an argument to the caller function.

Hands-On Exercise (7 mins)

upGrad

Imagine that a user is searching for a book for which the user needs to interact with the database. You can mock the functionality of interacting with the database using the *setTimeout()* method. Assume that the book details are getting created after 1 seconds.

After 1 seconds, you need to print the following message on the screen:

Book id is: 1

The name of the book is: Angels and Demons

The author of Angels and Demons is Dan Brown

Do not forget to use a callback to display the above message to the user.

The stub code is provided here.

The solution is provided here.

upGrad

Callback Hell

- Callback hell occurs when asynchronous JavaScript, which uses callback functions, turns into an unorganised mess called the “Pyramid of Doom”.

- Each function will get an argument, which is another function based on the result of the previous function. The entire premise will remain in a loop.

```
let verifyUser = (username, password, callback) => {
     if(error) {
        callback(error);
     } else {
           getRoles = (username, callback) => {
               if(error) {
                  callback(error)
               } else {
                    logUserAccess = (username, callback) => {
                        callback();
                    }
               }
           }
     }
}
```

Callback Hell

How will the following code behave?

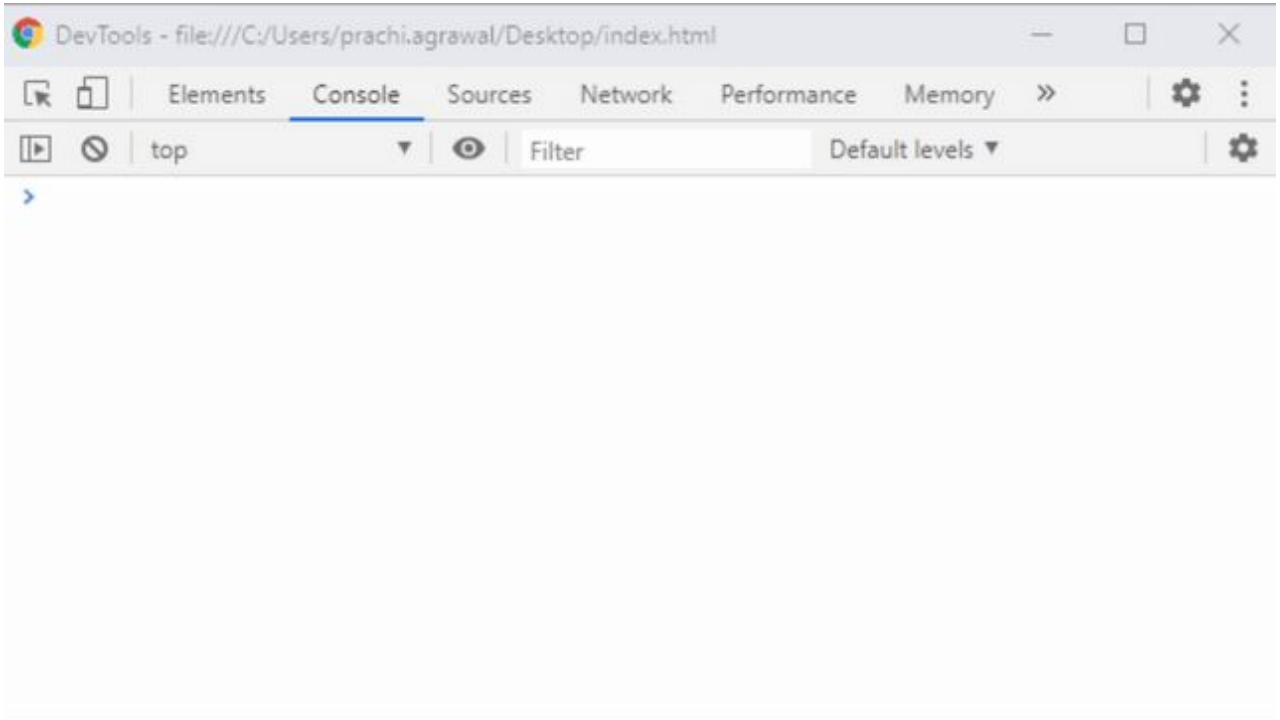
```
const getAddress = () => {
   getContinents(continent => {
     getCountries(continent, country => {
       getStates(country, state => {
         getCities(state, () => {
           done();
         });
       });
     });
   });
};

const done = () => {
   console.log("DONE!!!");
};
getAddress();
```

```
const getContinents = callback => {
   // callback = getCountries()
   setTimeout(() => {
      // code to get all continents
      let continent = "Asia";
      console.log(continent);
      callback(continent);
   }, 3000);
};
const getCountries = (continent,
callback) => {
   // callback = getStates()
   setTimeout(() => {
      // code to get all countries
      let country = "India";
      console.log(country);
      callback(country);
   }, 3000);
};
```

```
const getStates = (country, callback)
=> { // callback = getCities()
    setTimeout(() => {
      // code to get all states
      let state = "Maharashtra";
      console.log(state);
      callback(state);
    }, 3000);
};
const getCities = (state, callback)
=> { // callback = done()
    setTimeout(() => {
       // code to get all cities
       let city = "Mumbai";
       console.log(city);
       callback();
    }, 3000);
};
```

Callback Hell

This is how the code given in the previous slide will behave.



Poll 5 (15 Sec)

What is the output of the code snippet given in the following screenshot?

```
const getDetails = (callback) => {
    setTimeout(() => {
        let details = {
            firstName: "Prachi",
            lastName: "Agrawal"
        }
        callback(details);
    }, 5000);
}

const printDetails = (details) => {
    for (let key in details) {
        console.log(`${key} : ${details[key]}`);
    }
}

getDetails(printDetails);
```

1. Prachi Agrawal
2. firstName: Prachi lastName: Agrawal
3. undefined
4. Error

Poll 5 (Answer)

What is the output of the code snippet given in the following screenshot?

```
const getDetails = (callback) => {
    setTimeout(() => {
        let details = {
            firstName: "Prachi",
            lastName: "Agrawal"
        }
        callback(details);
    }, 5000);
}

const printDetails = (details) => {
    for (let key in details) {
        console.log(`${key} : ${details[key]}`);
    }
}

getDetails(printDetails);
```

1. Prachi Agrawal
2. **firstName: Prachi lastName: Agrawal**
3. undefined
4. Error

upGrad

JavaScript Promises

Introduction to JavaScript Promises

- To understand JavaScript (JS) promises, consider a blockbuster movie, such as Avengers. After its successful release, the fans of the movie asked the producer, Marvel Studios, if there will be a sequel. To resolve the fans' queries, Marvel Studios asked the fans to follow the studio on Twitter. The studio **promised** the fans that it will let them know on Twitter if a sequel is finalised. If it is, then all the followers of Marvel Studios on Twitter will be notified. If, for some reason, the sequel cannot be produced, then the followers will nevertheless be notified.

- This is how **promises** work in JS. There is a **producer code**, which is an analogy to the Marvel Studios in the example given above. Then, there is a **consumer code**, which is an analogy to the fans, and a **promise**, which is an analogy to the Twitter account following a link between the two. The producer code takes its time to produce the code and as per the promise, it will inform the consumer code if and when it is ready.

```
let promise = new Promise((resolve, reject) => {
    if(1 >= 0) {
       resolve("This is working!"); }
    else {
       reject(Error("It failed!"));
    }
})

promise.then(
   (result) => console.log(result),
   (error) => console.log(error)
);
//This is working!
```

Poll 6 (15 Sec)

Which of the following statements about promises is true?

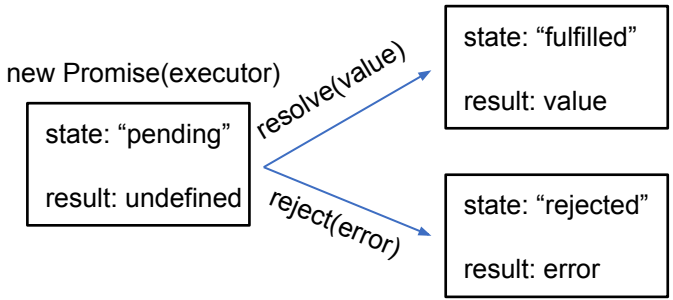
1. A producer code consumes the result produced by the consumer code.
2. A promise provides linkage between the producer and consumer code.
3. A promise contains the consumer code.
4. A producer code is needed to be explicitly called when the promise object is created.

Poll 6 (Answer)

Which of the following statements about promises is true?

1. A producer code consumes the result produced by the consumer code.
2. **A promise provides linkage between the producer and consumer code.**
3. A promise contains the consumer code.
4. A producer code is needed to be explicitly called when the promise object is created.

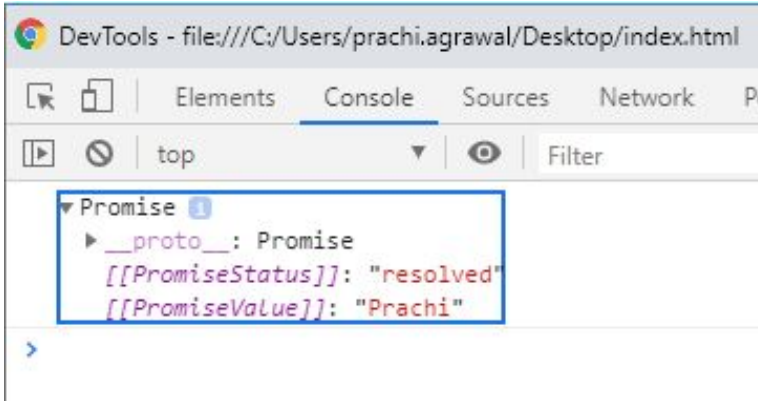
```
let promise = new Promise((resolve, reject) => {
    if(1 >= 0) {
       resolve("This is working!"); }
    else {
       reject(Error("It failed!"));
    }
})

promise.then(
   (result) => console.log(result),
   (error) => console.log(error)
);
//This is working!
```



Now, let's try to invoke the *resolve()* callback in the producer code.

When you invoke the *resolve()* callback in the producer code, the *PromiseStatus* property is changed to *'resolved'* and the *PromiseValue* property is changed to whatever is passed as an argument while resolving the promise.

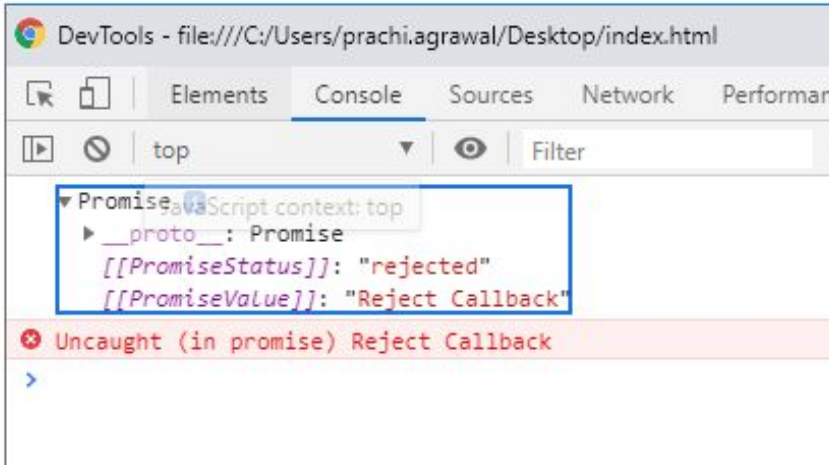
```
// resolving a promise
promiseObj = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Prachi");
    }, 1000);
});
console.log(promiseObj);
```



Now, let's try to invoke the *reject()* callback in the producer code.

When you invoke the *reject()* callback in the producer code, the *PromiseStatus* property is set to *'rejected'* and the *PromiseValue* property is changed to whatever is passed as an argument while rejecting the promise.

```
// rejecting a promise
promiseObj = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject("Reject Callback");
    }, 1000);
});
console.log(promiseObj);
```



Poll 7 (15 Sec)

Which of the following statements regarding the producer code present inside a promise object is true?

1. The result produced by the producer code can result in either success or failure or both.
2. When the result produced by the producer code is successful, the resolve() callback is invoked and when the result is unsuccessful, the reject() callback is invoked.
3. The resolve() and reject() callbacks are the parameters that can be considered as normal variables, which can be used to hold arguments of any data type, and not necessarily a function.

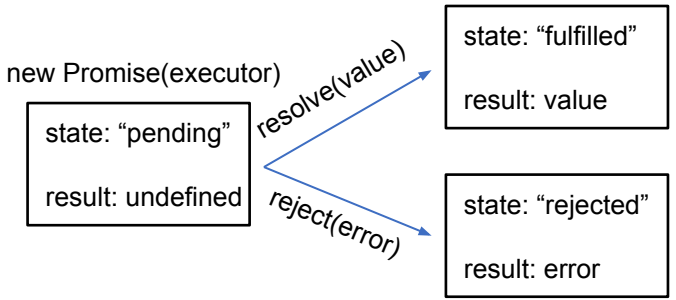
Poll 7 (Answer)

Which of the following statements regarding the producer code present inside a promise object is true?

1. The result produced by the producer code can result in either success or failure or both.
2. **When the result produced by the producer code is successful, the resolve() callback is invoked and when the result is unsuccessful, the reject() callback is invoked.**
3. The resolve() and reject() callbacks are the parameters that can be considered as normal variables, which can be used to hold arguments of any data type, and not necessarily a function.

JavaScript Promises: then, catch and finally

```
let promise = new Promise(function(resolve, reject) {
    //Some code to be executed
    if(1 >= 0) { resolve("This is working!"); }
    else { reject(Error("It failed!")); }
})

promise.then(result => console.log(result)); //This is working!
promise.catch(error => console.log(error));
promise.finally(() => console.log("Code finished execution."));
```



- As shown in the code given above, the promise syntax involves calling the promise with two arguments: **resolve** and **reject**.

- Once the new promise is created, the code starts executing. Then, it gets the result and if it is successful, then resolve is called and a value is passed. If the result is unsuccessful, then reject is called. This is the producer code and the promise variable is the link between the producer and consumer codes.

- The statements **then**, **catch** and **finally** are linked to the consumer code. If the promise works successfully, then the **then** function is called; and if it fails, then the **catch** error is called with the error; and the **finally** statement is called after the then or catch. **finally** will be executed no matter whether the promise is successful or it fails.

Poll 8 (15 Sec)

Why are promises preferred over callbacks?

1. When a lot of callbacks is used, the program can lead to a problem called callback hell.
2. Promises are a cleaner way of running asynchronous tasks.
3. Promises provide a seamless error-catching mechanism.
4. All of the above.

Poll 8 (Answer)

Why are promises preferred over callbacks?

1. When a lot of callbacks is used, the program can lead to a problem called callback hell.
2. Promises are a cleaner way of running asynchronous tasks.
3. Promises provide a seamless error-catching mechanism.
4. **All of the above.**

Promises are always preferred over callbacks because they provide clear code with an error-catching mechanism. Using callbacks, you can get lost in scoping of variables, and also, code becomes complex to read and understand hence leading to callback hell. Using promise, you can write cleaner code.

JavaScript Promises: async and await

```
async function foo() { //The async keyword placed before a function means that the function will return a promise.
   let promise = new Promise(function(resolve, reject) {
      resolve("Done");
   });
   let result = await promise; //The await keyword makes JavaScript wait until the promise gets resolved or rejected and returns its result.
   async(result);
}
```

- The **async** keyword placed before a function means that the function will return a promise.
- The **await** keyword makes JavaScript wait until the promise gets resolved or rejected and returns its result.

upGrad

Doubt Clearance (5 mins)

Hands-On Exercise (7 mins)

upGrad

You are required to get the email ID of the currently logged-in user from the server. Initially, the message should be printed as *'Getting email from server...'* to the console. In order to mock the functionality of getting it from the server, you can use the *setTimeout()* method and assume that the email will be returned as a response after two seconds.

Then, implement this using a promise and in the producer code, **resolve** the promise while sending in your **email** as the string argument.

Now, in the consumer code, use the ***then()*** method and print the email received as an argument to the console.

The output on the console is given as follows:
Getting email from server...
Email = prachi.agrawal@upgrad.com

The stub code is provided here.

The solution is provided here.

Key Takeaways

- JS being single-threaded.
- JavaScript (JS) is, by default, synchronous in nature. However, at instances such as fetching data from the server or waiting for a specified interval of time, JS uses asynchronous function.
- JS callback functions can be called when you want to execute a different function after the current function has been executed. However, if there is a series of callback functions, then it can lead to a problem called *callback hell*.
- JS promises are methods of dealing with asynchronous functions. Once a code is executed, promises can wait for the request. Once the request is received, and if it is successful, then resolve will be called or else reject will be called. Promises are chained using *.then()* for success, *.catch()* for catching errors, and *.finally()* for the statement to be executed at end no matter if the promise is resolved or rejected.

Task to complete after today's session

MCQs
Coding Questions

In the next class, we will discuss...

- Introduction to JSON and how to parse JSON objects
- Introduction to AJAX
- Introduction to HTTP requests and responses
- Differences among GET, POST, PUT and DELETE requests
- HTTP headers
- Debugging the code

upGrad

#RahoAmbitious

Thank You!