CSM
breaking thought barriers

Angular Training

Session -8

Outlines

- Modules
- Promises

Modules

- ES6 modules allow you to structure JavaScript code in a modular fashion.
- Modules provide a standardized way for defining and importing/exporting reusable pieces of code within a JavaScript application.
- By default, ES6 modules encapsulate their code. This means that values (variables, functions, classes, etc.) defined in a module are not accessible from outside of the module by default.
- This prevents naming conflicts and promotes better code structure.
- Modules can export values (variables, functions, classes, etc.,) using the export keyword.
- An ES6 module is a JavaScript file that executes in strict mode only. It means that any variables or functions declared in the module won't be added automatically to the global scope.

Modules

Messages.js

```
export let message = 'ES6 Modules';
```

app.js

```
import { message } from './message.js'
const h1 = document.createElement('h1');
h1.textContent = message
document.body.appendChild(h1)
```

```
import {message, setMessage } from './greeting.js';
```

Import an entire module as an object

```
import * as cal from './cal.js';
```

Aliasing

```
function add(a,b)
{
  return a + b;
}
export { add as sum };
```

```
import { sum } from './math.js';
```

```
import {sum as total} from './math.js';
```

export

- The export keyword exports values from a module so that you can use them in other modules.

- There are two types of exports:

  1. Named exports
  2. Default exports

- A module can have **multiple named exports** but **only one default export**.

- Exporting variables

```
let count = 1;
export { count };
```

```
let count = 1;
const MIN = 0, MAX = 10;
export { MIN, MAX, count };
```

export

- Exporting functions

```
function increase() {
  // ..
}
export { increase };
```

```
export function increase() {
  // ...
}
```

- Exporting classes

```
class Counter {
  constructor() {
    this.count = 1;
  }
  increase() {
    this.count++;
  }
  get current() {
    return this.count;
  }
}

export { Counter };
```

- **Default exports**
  - A module can have one default export. To export a value using a default export, you use the default export keyword.

```
let message = 'Hi';
export { default as message };
```

```
export default let message = 'Hi';
```

  - When importing a default export, you don't need to place the variable inside curly braces:

```
import message from 'module.js';
```

  - Note that if the message was exported using a named export, you would place it inside the curly braces:

```
import { message} from 'module.js';
```

Re-exporting a binding

It’s possible to export bindings that you have imported. This is called re-exporting.

```
import { sum } from './math.js';
export { sum };
```

Default exports

A module can have one and only one default export. The default export is easier to import. The default for a module can be a variable, a function, or a class.

```
export default function(arr) {
// sorting here
}
```

Promises

Why JavaScript promises ?

```
function getUsers() {
  return [
    { username: 'john', email: 'john@test.com' },
    { username: ‘ jane ', email: 'jane@test.com' },
  ];
}

function findUser(username) {
  const users = getUsers();
  const user = users.find((user) => user.username === username);
  return user;
}

console.log(findUser('john'));
```

```
function getUsers() {
  let users = [];
  setTimeout(() => {
    users = [
      { username: 'john', email: 'john@test.com' },
      { username: 'jane', email: 'jane@test.com' },
    ];
  }, 1000);
  return users;
}
function findUser(username) {
  const users = getUsers(); // A
  const user = users.find((user) => user.username === username); // B
  return user;
}
console.log(findUser('john'));
```

Challenge

- The challenge is how to access the users returned from the getUsers() function after one second. One classical approach is to use the callback.

Using callbacks to deal with an asynchronous operation

```
function getUsers(callback) {
  setTimeout(() => {
    callback([
      { username: 'john', email: 'john@test.com' },
      { username: 'jane', email: 'jane@test.com' },
    ]);
  }, 1000);
}
function findUser(username, callback) {
  getUsers((users) => {
    const user = users.find((user) => user.username === username);
    callback(user);
  });
}
findUser('john', console.log);
```

- The callback approach works very well. However, it makes the code more difficult to follow. Also, it adds complexity to the functions with callback arguments.

- If the number of functions grows, you may end up with the **callback hell problem**. To resolve this, JavaScript comes up with the concept of promises.

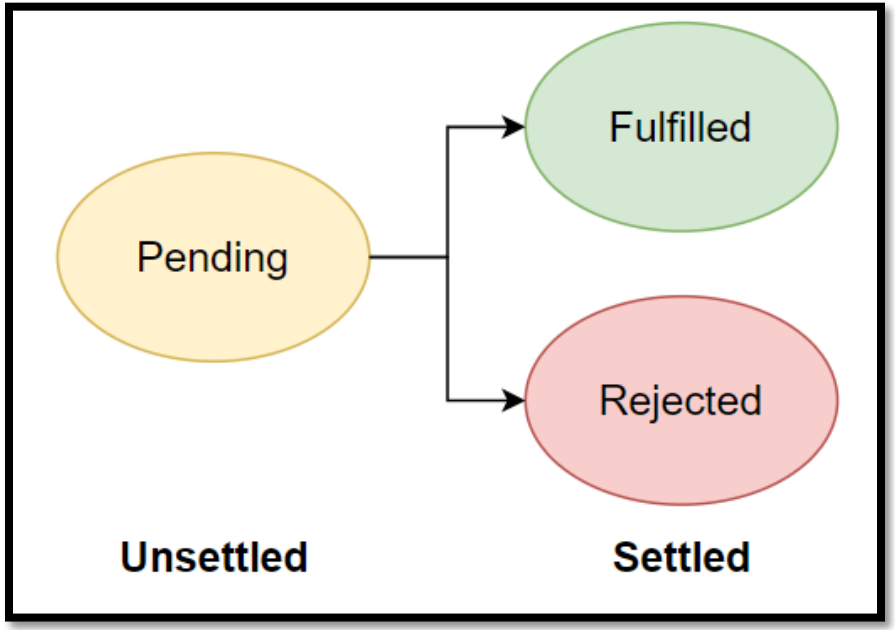
Promises

- By definition, a promise is an object that encapsulates the result of an asynchronous operation.
- A promise object has a state that can be one of the following:

❑ Pending

❑ Fulfilled with a **value**

❑ Rejected for a **reason**




```
const promise = fetch("books.json");
```

Creating a promise

- To create a promise object, you use the Promise() constructor:

```
const promise = new Promise((resolve, reject) => {
  // contain an operation
  // ...

  // return the state
  if (success) {
    resolve(value);
  } else {
    reject(error);
  }
});
```

- The promise constructor accepts a callback function that typically performs an asynchronous operation. This function is often referred to as an executor.

- The executor accepts two callback functions with the name **resolve** and **reject**.

- If the asynchronous operation completes successfully, the executor will call the resolve() function to change the state of the promise from pending to fulfilled with a value.

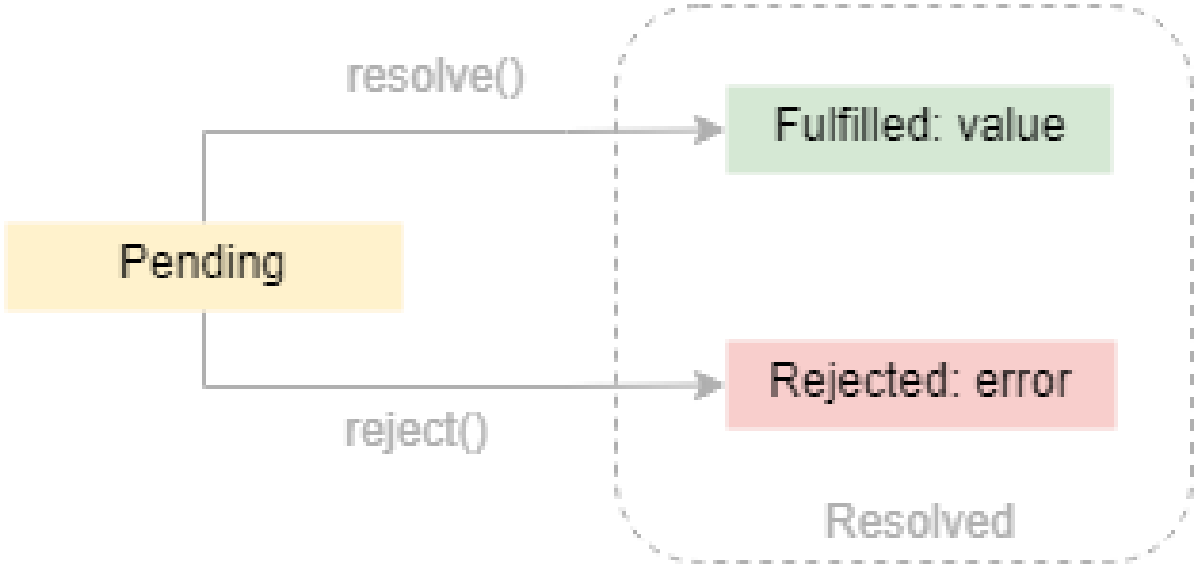
- In case of an error, the executor will call the reject() function to change the state of the promise from pending to rejected with the error reason.

Once a promise reaches either fulfilled or rejected state, it stays in that state and can’t go to another state.

```
function getUsers() {
  return new Promise((resolve, reject) => {
}); setTimeout(() => {
      resolve([
        { username: 'john', email: 'john@test.com' },
        { username: 'jane', email: 'jane@test.com' },
      ]);
    }, 1000);

}
function onFulfilled(users) {
  console.log(users);
}
const promise = getUsers();
promise.then(onFulfilled);
```

```
function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve([
        { username: 'john', email: 'john@test.com' },
        { username: 'jane', email: 'jane@test.com' },
      ]);
    }, 1000);
  });
}

const promise = getUsers();

promise.then((users) => {
  console.log(users);
});
```

- Once a new Promise object is created, its state is pending. If a promise reaches fulfilled or rejected state, it is resolved.



Consuming a Promise: then, catch, finally (**handlers)**

1) The then() method

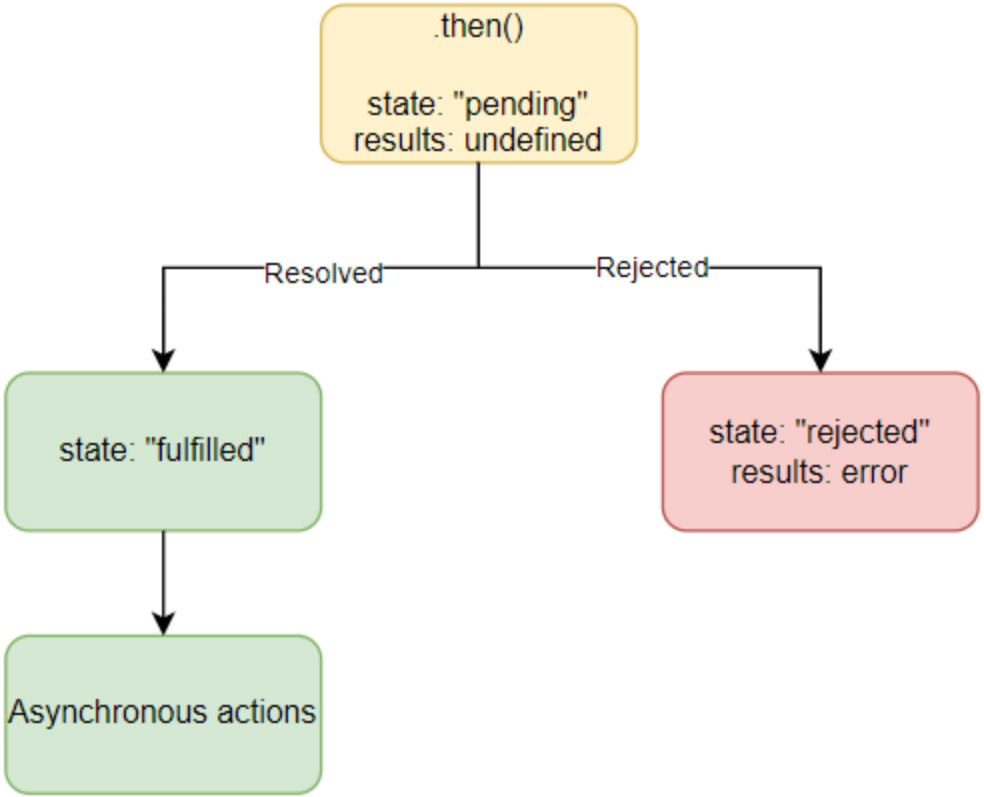
- To get the value of a promise when it's fulfilled, you call the then() method of the promise object.

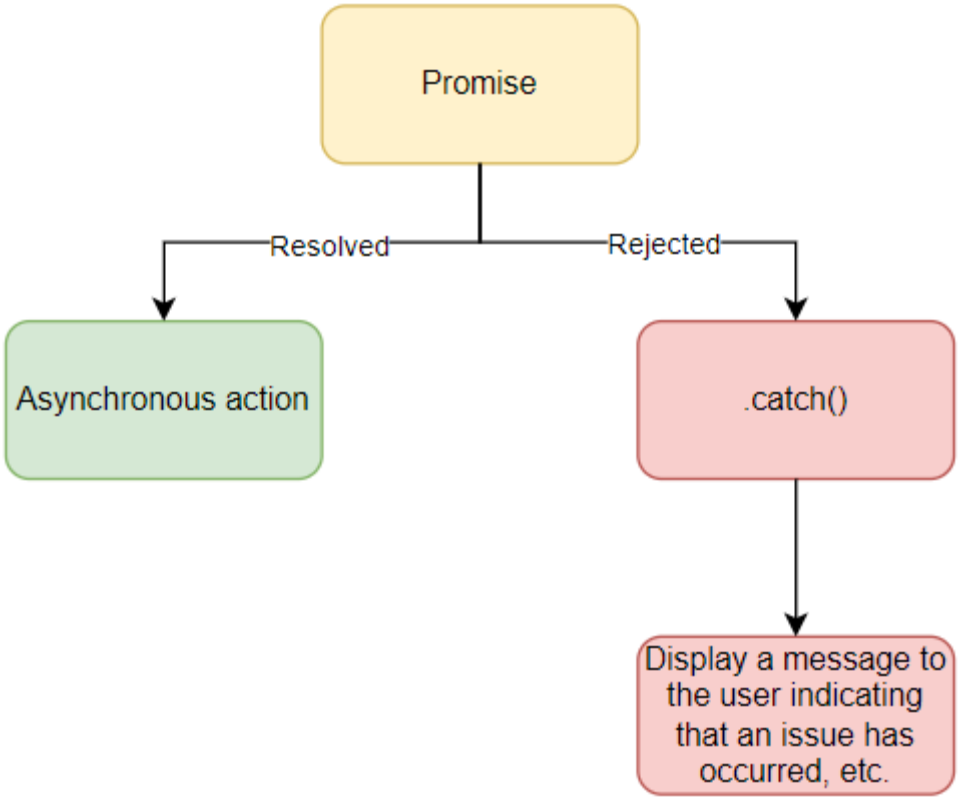
- Syntax :

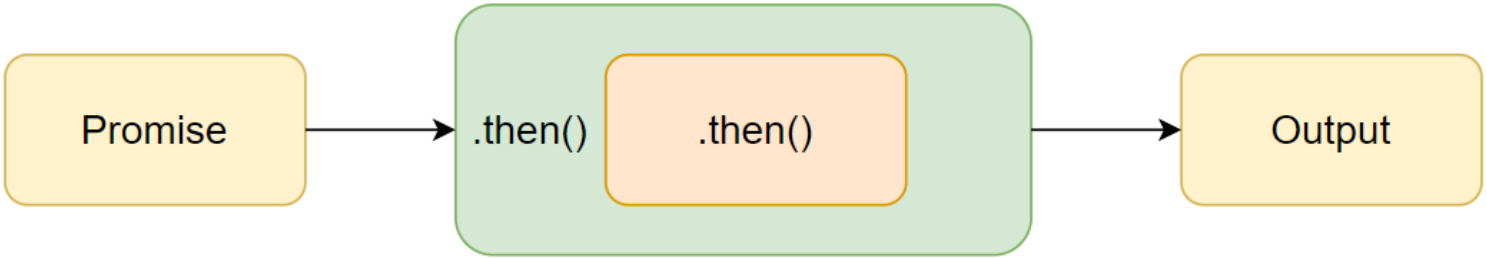
```
promise.then([onFulfilled,onRejected]);
```

- The then() method calls the onFulfilled() with a value, if the promise is fulfilled or the onRejected() with an error if the promise is rejected.

Note that both onFulfilled and onRejected arguments are optional.


.then()
state: "pending"
results: undefined
Resolved
Rejected
state: "fulfilled"
state: "rejected"
results: error
Asynchronous actions


Promise
Resolved
Rejected
Asynchronous action
.catch()
Display a message to the user indicating that an issue has occurred, etc.


Promise
.then()
.then()
Output

```
let success = true;
function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (success) {
        resolve([
          { username: 'john', email: 'john@test.com' },
          { username: 'jane', email: 'jane@test.com' },
        ]);
      } else {
        reject('Failed to the user list');
      }
    }, 1000);
  });
}
function onFulfilled(users) {
  console.log(users);
}
function onRejected(error) {
  console.log(error);
}
const promise = getUsers();
promise.then(onFulfilled, onRejected);
```

2) The catch() method

```
promise.catch(onRejected);
```

```
let success = false;
function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (success) {
        resolve([
          { username: 'john', email: 'john@test.com' },
          { username: 'jane', email: 'jane@test.com' },
        ]);
      } else {
        reject('Failed to the user list');
      }
    }, 1000);
  });
}
const promise = getUsers();
promise.catch((error) => {
  console.log(error);
});
```

3) The finally() method

```
const render = () => {
  //...
};

getUsers()
  .then((users) => {
    console.log(users);
  })
  .catch((error) => {
    console.log(error);
  })
  .finally(() => {
    render();
  });
```

A practical JavaScript Promise example

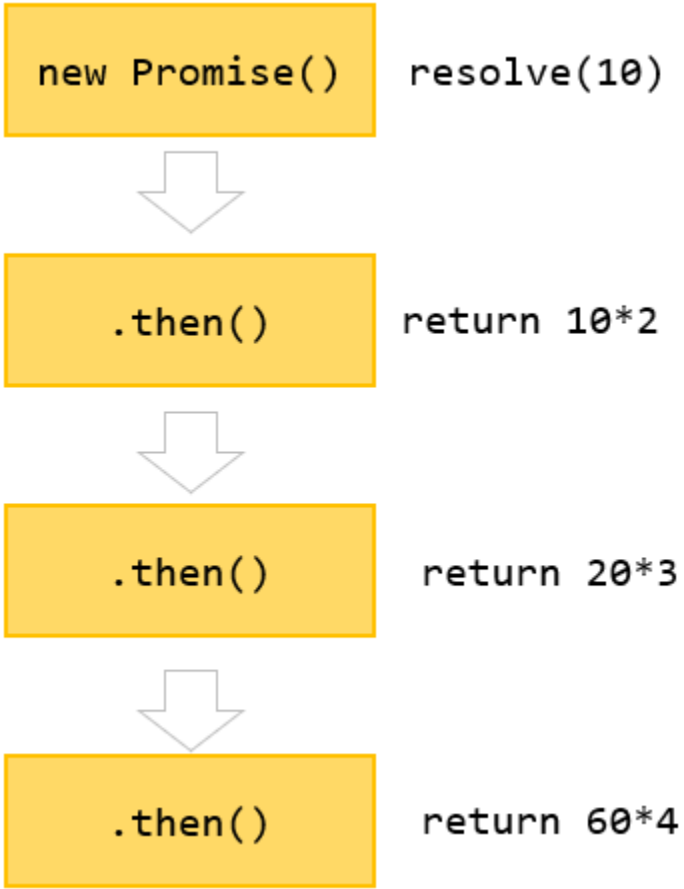
```
fetch( url )
  .then(function() {
    // handle the response
  })
  .catch(function() {
    // handle the error
  });
```

API : **https://jsonplaceholder.typicode.com/users**

Promise Chaining

Sometimes, you want to execute two or more related asynchronous operations, where the next operation starts with the result from the previous step.

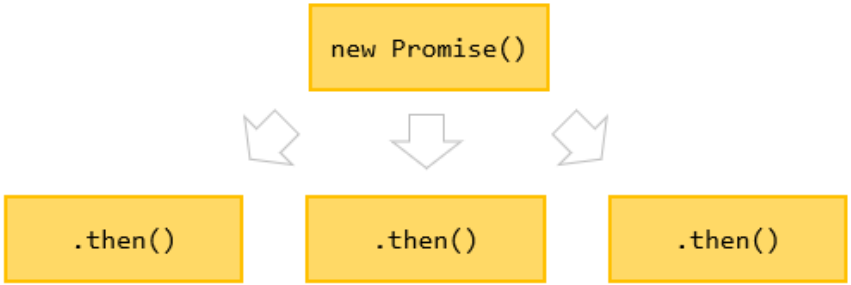
```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});

p.then((result) => {
    console.log(result); // 10
    return result * 2;
}).then((result) => {
    console.log(result); // 20
    return result * 3;
}).then((result) => {
    console.log(result); // 60
    return result * 4;
});
```



Multiple handlers for a promise

When you call the then() method multiple times on a promise, it is not the promise chaining.

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});
p.then((result) => {
    console.log(result); // 10
    return result * 2;
})
p.then((result) => {
    console.log(result); // 10
    return result * 3;
})
```



async/await

- An **async** keyword turns a regular function into an asynchronous function.
- An asynchronous function is a function that runs in its own timing—without waiting for others to finish their execution first.
- By default, an async function returns a resolved or rejected promise object.

```
async function myMomsPromise() {
  return 'I get a book';
}
console.log(myMomsPromise());
```

async/await

- An await keyword instructs a function to wait for a promise to be settled before continuing its execution.

- The await keyword works only inside an async function in regular JavaScript code.

- We can use zero or more await expressions in an async function.

```
async function showMomsPromise() {
  const myMomsPromise = new Promise(function (resolve,
reject) {
    setTimeout(resolve, 5000, "I get a book");
  });
  console.log(await myMomsPromise);
}
showMomsPromise();
```

Revisiting JS

1. **let,const**
2. **Object Destructuring,Object Literals**
3. **Arrow Functions**
4. **Rest,Spread Operator**
5. **Template Literal**
6. **Promises**
7. **Classes**
8. **Modules**
9. **Promises**

Assignment -1

Implement a basic task management system using local storage.
Create a JavaScript program that provides the following functionalities:

Add a Task: Allow the user to add a new task with a title and description.
View Tasks: Display a list of all tasks currently stored in local storage.
Update a Task: Allow the user to update the title or description of an existing task.
Delete a Task: Allow the user to delete a task.

Use the local storage to store and retrieve the tasks.

Requirements:

- The tasks should have at least a title and description.
- Display an intuitive user interface for interacting with the task management system.
- Utilize local storage to persist the tasks between page reloads.

Assignment -2

Implement a basic task management system that communicates with a Web API for CRUD operations.
The API has the following endpoints:

- **GET /tasks:** Retrieve a list of all tasks.
- **POST /tasks:** Add a new task.
- **PUT /tasks/:id:** Update an existing task by ID.
- **DELETE /tasks/:id**: Delete a task by ID.

Create a JavaScript program that provides the following functionalities:

1. **Fetch Tasks:** Retrieve a list of all tasks from the Web API and display them.
2. **Add a Task:** Allow the user to add a new task with a title and description via the Web API.
3. **Update a Task:** Allow the user to update the title or description of an existing task via the Web API.
4. **Delete a Task:** Allow the user to delete a task via the Web API.

Use the provided API endpoints to perform these operations.

Requirements:

- Display an intuitive user interface for interacting with the task management system.
- Handle errors gracefully and provide feedback to the user.
- Ensure that the code is modular, readable, and well-commented.