JavaScript and TypeScript Fundamentals

Course Code: CSC 4182 Course Title: Advanced Programming In Web Technologies

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**2**	**Week No:**	**01**	**Semester:**	
Lecturer:	*Sazzad Hossain; sazzad@aiub.edu*				

Lecture Outline

- ✓ NestJS
- ✓ Node.js Vs Express.js Vs NestJS
- ✓ TypeScript(TS)?
- ✓ Declare Variables in JavaScript/TypeScript
- ✓ Data Types in TypeScript
- ✓ TypeScript Decorators
- ✓ TypeScript Generics
- ✓ TypeScript Async/Await and Promises

JavaScript ES6

- JavaScript was invented by Brendan Eich in 1995, and became an **ECMA** standard in 1997.
- **ECMAScript** is the official name of the language.
- ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
- Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).
- ECMAScript 2015 was the **second major** revision to JavaScript.
- ECMAScript 2015 is also known as **ES6 and ECMAScript 6**.
- This chapter describes the most important features of ES6.

JavaScript ES6 Features

Following are some of the ES6 features will be required in this course

- Variable Scope
- Class
- Arrow Functions
- Variables (let, const, var)
- Array Methods like .map()
- Destructuring
- Modules
- Ternary Operator
- Async/Await and Promises

Declare Variables in JavaScript

Variables can be declared using the **let**, **const**, or **var** keywords.

The **let** keyword is used to declare variables that **can be reassigned** with a new value.
is used to declare variables that are **block-scoped**. Block scope refers to the **portion of code within curly braces {}** (e.g., inside a function, loop, or conditional statement).

Example:

```
let age: number = 25;
let message: string = 'Hello, TypeScript!';
let isActive: boolean = true;
```

Declare Variables in JavaScript

The **const** keyword is used to declare variables that have a **constant** value and **cannot be reassigned**.

Example:

```
const PI: number = 3.14159;
const fullName: string = 'John Doe';
const isActive: boolean = true;
```

The **var** keyword is the legacy way of declaring variables in JavaScript and TypeScript. Variables declared with var have **function** or **global scope** and are hoisted.

Example:

```
var age: number = 25;
var message: string = 'Hello, TypeScript!';
var isActive: boolean = true;
```

Classes

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

The super() method refers to the parent class.

By calling the super() method in the constructor method, we call the parent's constructor method and get access to the parent's properties and methods.

Classes

```
class Car {
  constructor(name) {
    this.brand = name;
  }
  present() {
    return 'I have a ' + this.brand;
  }
}
class Model extends Car {
  constructor(name, mod) {
    super(name);
    this.model = mod;
  }
  show() {
      return this.present() + ', it is a ' + this.model
  }
}
const mycar = new Model("Ford", "Mustang");
mycar.show();
```

Arrow Functions

- Arrow functions are always unnamed. If the arrow function needs to call itself, use a named function expression instead.
- Arrow functions can have either a concise body or the usual block body.
- Arrow functions don't have their own bindings to ‘this’ keyword, arguments, or super, and should not be used as methods.
- Arrow functions cannot be used as constructors.
- Calling them with new throws a TypeError.

Arrow Functions

```
hello = () => "Hello World!";

// An empty arrow function returns undefined
const empty = () => {};

(() => "foobar")();
// Returns "foobar"

const simple = (a) => (a > 15 ? 15 : a);
simple(16); // 15
simple(10); // 10

const max = (a, b) => (a > b ? a : b);
```

Array Methods

The .**map**() method allows you to run a function on each item in the array, returning a new array as the result.

map() can be used to generate lists.

```
const myArray = ['apple', 'banana', 'orange'];
const myList = myArray.map((item) => <p>{item}</p>)
```

Destructuring Objects

- The **destructuring assignment** syntax is a JavaScript expression that makes it possible to **unpack** values from arrays, or properties from objects, into distinct variables.

Example;

https://www.w3schools.com/REACT/tryit.asp?filename=tryreact_es6_destructuring_object2

Modules

JavaScript modules allow you to break up your code into separate files.

ES Modules rely on the `import` and `export` statements.

Named Exports

You can create named exports two ways. In-line individually, or all at once at the bottom.

Default Exports

You can only have **one default** export in a file.

https://www.w3schools.com/REACT/react_es6_modules.asp

IMPORT

Import named exports from the file person.js:

```
import { name, age } from "./person.js";
```

Import a default export from the file message.js:

```
import message from "./message.js";
```

Ternary Operator

The ternary operator is a simplified conditional operator like `if` / `else`.
Syntax:
`condition ? <expression if true> : <expression if false>`

https://www.w3schools.com/REACT/tryit.asp?filename=tryreact_es6_ternary2

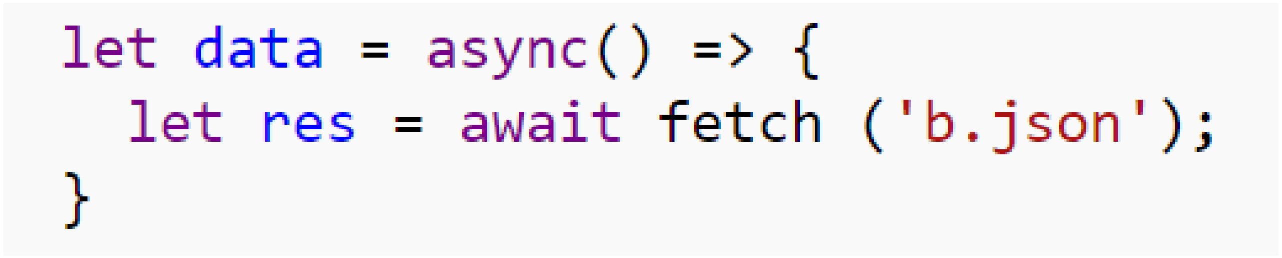
Spread Operator

The JavaScript **spread operator (...)** allows us to quickly copy all or part of an existing array or object into another array or object.

Details are here;
https://www.w3schools.com/REACT/react_es6_spread.asp

Async and Await

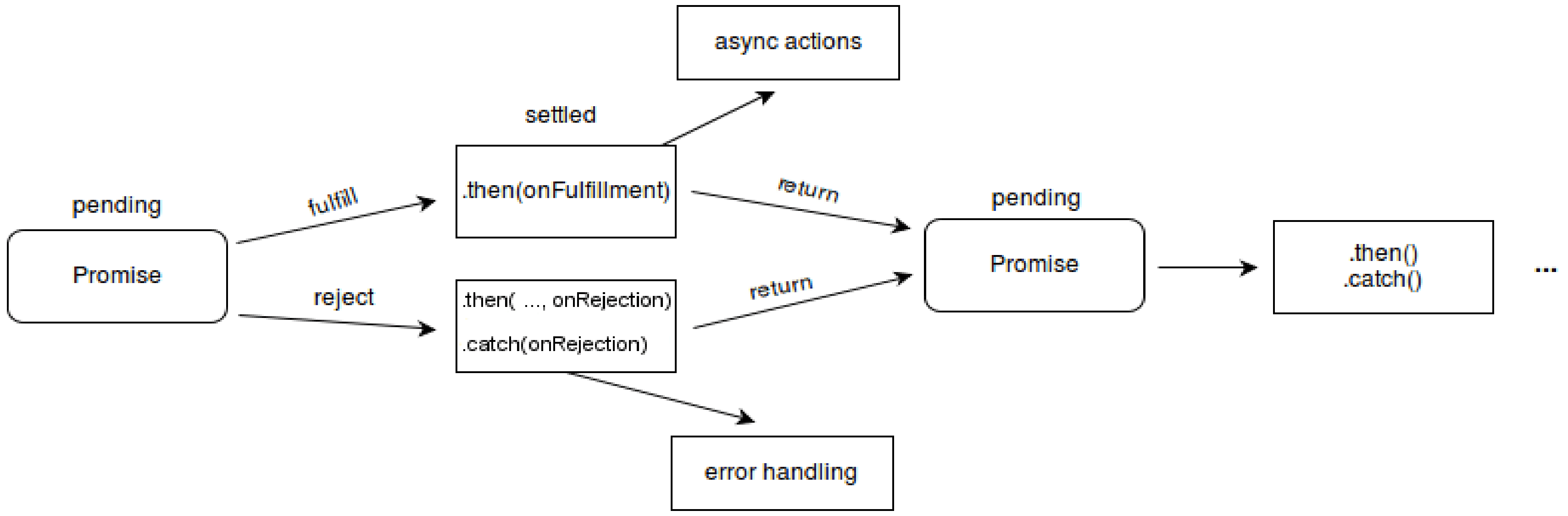
Async/Await is a syntactic feature introduced in **ECMAScript** 2017 that allows you to write asynchronous code in a more synchronous and readable manner.
The **async function** declaration creates a **binding** of a new async function to a given name. The **await** keyword **is permitted within the function body, enabling asynchronous**, **promise-based behavior to be written** in a cleaner style and avoiding the need to explicitly configure promise chains.



```
let data = async() => {
  let res = await fetch ('b.json');
}
```

Async/Await and Promise

Promises are a core feature of JavaScript that provide an abstraction for handling **asynchronous** operations.
A Promise is an **object representing** the eventual completion or failure of an asynchronous operation, and it can be in one of three states: **pending**, **fulfilled**, or **rejected**.
It has methods like **.then()** and **.catch()**



Async/Await and Promise

```
async getUser(id: string): Promise<User> {
  try {
    // Retrieve the user asynchronously
    const user = await this.userService.getUserById(id);
    // Retrieve the posts asynchronously
    const posts = await this.postService.getPostsByUserId(user.id);
    // Assign the retrieved posts to the user object
    user.posts = posts;
    return user;
  } catch (error) {
    // If any error occurs during the asynchronous operations,
throw a NotFoundException
    throw new NotFoundException('User not found');
  }
}
```

By using **await**, the execution of the method is **paused** until the **Promises** returned by the service methods are resolved.

TypeScript(TS)?

TypeScript is a **strongly typed** programming language that builds on JavaScript

- TypeScript adds additional syntax to JavaScript to support a tighter integration with editor. **Catch errors** early in editor.
- TypeScript code converts to JavaScript, which **runs anywhere JavaScript runs**; In a browser or Node.js/Server.
- TypeScript understands JavaScript and uses **type inference to give you great tooling** without additional code.

History of TypeScript

- Developed by Microsoft, was first announced by Anders Hejlsberg in October 2012.
- It was introduced as a **superset** of JavaScript that adds optional **static typing, class-based object-oriented programming,** and improved tooling support to JavaScript development.
- Gained significant popularity in the JavaScript community due to its ability to **catch errors at compile-time**, provide **better tooling** support, and enable developers to build more **maintainable** and **scalable** applications.

Application of TypeScript

- **Web Development:** TypeScript is commonly used for building web applications, both on the client-side and server-side. Popular web frameworks like **Angular, React,** and **Vue.js** have built-in support for TypeScript.
- **Node.js Development:** Node.js frameworks and libraries, such as NestJS and TypeORM, are built using TypeScript.
- **Mobile App Development:** Frameworks like React Native and NativeScript to build cross-platform mobile apps.
- **Testing and Automation:** TypeScript is often used for writing tests and automation scripts.

Data Types in TypeScript

- **Boolean**: Represents the logical values true or false.

```
let isCompleted: boolean = true;
```

- **Number**: Represents numeric values, both integer and floating-point.

```
let age: number = 25;
let price: number = 9.99;
```

- **String**: Represents textual data enclosed in single or double quotes.

```
let message: string = 'Hello, World!';
```

- **Any**: Represents a dynamic or flexible type that allows values of any type.

```
let data: any = 'Hello, World!';
data = 42;
data = true;
```

Data Types in TypeScript

- **Array:** Represents a collection of values of a particular type.

```
let numbers: number[] = [1, 2, 3, 4, 5];
let fruits: Array<string> = ['apple', 'banana', 'orange'];
```

- **Void:** Represents the absence of a value. Typically used as the return type for functions that do not return a value.

```
function sayHello(): void {
  console.log('Hello!');}
```

- **Object**: Represents any non-primitive type, such as arrays, functions, or objects.

```
let user: object = {
  name: 'John Doe',
  age: 25,
};
```

TypeScript Decorators

- TypeScript **decorators** provide a way to add metadata and modify the behavior of classes, properties, methods, or parameters at design time.
- Decorators are expressions prefixed with the **@** symbol and are applied to the target they are decorating.
- To enhance or extend the functionality of existing code without modifying its original implementation.
- NestJS heavily utilizes decorators to define routes, controllers, services, and other application components.

TypeScript Decorators

```
import { Controller, Get } from
'@nestjs/common';

@Controller('users')// Control Decorator
export class UsersController {
  @Get()// Get Decorator
  getUsers(): string {
    return 'Get all users';
  }
}
```

TypeScript Generics

Generics allow you to create **reusable components** that can work with different types.

```
class Box<T> {
  private item: T;

  addItem(item: T): void {
    this.item = item;
  }
  getItem(): T {
    return this.item;
  }
}
const stringBox = new Box<string>();
stringBox.addItem('Hello, World!');
console.log(stringBox.getItem()); // Output: Hello, World!
```

References

1. W3Schools Online Web Tutorials, URL: http://www.w3schools.com
2. Node.js, URL: https://nodejs.org/en/
3. Next.js, URL: https://nextjs.org/
4. TypeScript URL: https://www.typescriptlang.org/
5. MDN Web Docs URL: https://developer.mozilla.org/

Thank You!