ADVANCED

→ Asynchronous Programming (Promises, Async/Await)

1. Promises

→ A promise is an object representing the eventual completion or failure of an asynchronous operation.

- States:

a). Pending - Initial state, neither fulfilled nor rejected.
b). Fulfilled - Operation completed successfully.
c). Rejected - Operation failed.

- Methods:

a). then (onFulfilled, onRejected): Attaches callbacks for the resolution and/or rejection of the promise.
b). catch (onRejected): Attaches a callback for only the rejection of the Promise.
c). finally (onFinally): Executes a callback once the Promise is settled, regardless of the outcome.

e.g.-

```
let promise = new Promise ((resolve, reject) => {
    // async operation
    if (success)
    {
        resolve(result);
    }
    else
    {
        reject (error);
    }
});
promise
    .then (result => console.log (result))
    .catch (error => console.log (error));
```

2. Async / Await

→ 'async' and 'await' are syntactic sugar over Promises. They make asynchronous code look and behave more like synchronous code.

- Usage:

a). async function: Declares an asynchronous function. Always returns a Promise.

b). await: Pauses execution until the Promise is settled and returns the result.

e.g:

```
async function fetchData()
{
    try
    {
        let response = await fetch(url);
        let data = await response.json();
        console.log(data);
    }
    catch (error)
    {
        console.log(error);
    }
}
```

→ Closures

A closure is a function that retains access to its lexical scopes, even when the function is executed outside that scope.

- Characteristics:

a). Functions can access variables from their outer scope.
b). Useful for data encapsulation and private variables.

e.g=
```
function outerFunction()
{
    let counter = 0;
    function innerFunction()
    {
        counter++;
        return counter;
    }
    return innerFunction;
}
const inc = outerFunction();
console.log(inc()); // 1
console.log(inc()); // 2
```

→ ES6 Features

1. Arrow Functions

→ • Syntax: Concise way to write function using '⇒'.
• Characteristics:

a). No 'this' binding: 'this' refers to the enclosing context.
b) Cannot be used as constructors.
c). No 'arguments' object.

e.g-
```
const add = (a, b) ⇒ a+b;
console.log(add(2, 3));
```

2. Destructing

→ A syntax for extracting values from arrays or objects into distinct variables.

- **Array Destructing:**

e.g-
```
let [a, b] = [1, 2];
console.log(a, b);  // 1 2
```

- **Object Destructing:**

e.g-
```
let {name, age} = {name: "Alice", age: 25};
console.log(name, age);  // Alice 25.
```

3. **Default Parameters**

→ Allowes function parameters to have default values.

e.g-
```
function greet(name = "Athorva")
{
    console.log(`Hello, ${name}!`);
}
greet();
```

4. **Template literals**

→ A way to include expressions in strings using backticks and `${}`.

e.g-
```
let name = "Athorva";
console.log(`Hello, ${name}!`);
```

5. **Rest and Spread Operators**

→ • Rest (`...`): Collects all remaining elements into an array.

e.g-
```
function sum(...args)
{
    return args.reduce((a,b) => a+b, 0);
}
console.log(sum(1,2,3));  // 6
```

• Spread (`...`): Spreads elements of an array or object.

e.g-
```
let arr = [1,2,3];
let arrCopy = [...arr];  // [1,2,3]
let obj = {a:1, b:2};
let objCopy = {...obj};  // {a:1, b:2}
```