Module 9 [Advanced JS]

__/__/____

① Asynchronous JS & Event loop :

JS is a single-threaded (runs one thing at a time), but can handle asynchronous tasks like network requests, timers or I/O without blocking. This is possible because of event loop.

Event loop Flow :

(A) Callstack - where JS execute code line by line

(B) Web/Browser APIs - async oper like setTimeOut, fetch.

(C) Callback Queue / Task Queue - completed async callbacks wait here.

(D) Event loop - moves callbacks from queue to call stack when stack is empty.

→

```
console.log ("Start");
setTimeOut (() => {
    console.log ("TimeOut");
}, 0);
console.log ("End");
```

// Start
// End
// TimeOut

↳ even with 0ms, setTimeOut runs last because it goes to the callback queue.

__/__/____

② Closure :

A closure is a function that remembers its outer space/scope even after that scope has finished execution.

useful
- Data encapsulation
- Private variables
- Functional programming

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    return count;
  };
}

const counter = outer();
console.log(counter()); // 1
console.log(counter()); // 2
```

inner "closes over" count.

③ Promises & Promise Chaining :

Promises represent a future value from an async operation, states : pending, Fulfilled, rejected

```
let promise = new Promise((resolve, reject) => {
  setTime(() => resolve("done!"), 1000);
});

promise
```

__/__/____

```
.then (result => {
    console.log (result);        // Done!
    return "Next step";
})
.then (step => console.log (step));  // Next step
```

→ Chaining allows sequential async operations without 'callback hell.'

④ this & Binding :

this refers to the context of function execution. It changes depending on how the function is called.

rules :

(A) Global / Function call - this is window (or undefined in strict mode).
(B) Method call - this is the object.
(C) Constructor (new) - this is new object.
(D) Explicit Binding - call, apply, bind

```
const obj = {
    name : "JS", greet() { console.log (this.name); }
};
obj.greet();          // JS
const fnc = obj.greet;
fnc();                // undefined

const bound = fn.bind (obj);
bound();              // JS
```

__/__/____

⑤ Async/Await & Promises.all :

- Async/Await simplifies promises.

```
async function fetchData () {
  try {
    const res = await fetch ("https://api.example.
                              com/data");
    const data = await res.json();
    console.log (data);
  } catch (err) {
      console.error (err);
  }
}

fetchData();
```

- Promises.all runs multiple promises in parallel

```
const p1 = Promise.resolve (1);
const p2 = Promise.resolve (2);
Promise.all ([p1, p2]).then (results => console.log
            (results));          // [1, 2]
```

⑥ Iterators & Generators :

objects with next() method returning { value, done }

Functions that can cause execution with yield :

```
Function * gen () {
  yield 1;
```

__/__/____

```
yield 2;  yield 3;  }
const g = gen();
console.log(g.next()); // { value: 1, done: false }
console.log(g.next()); // { value: 2, done: false }
```

Useful for lazy evaluation & async workflows.

⑦ ES6 Modules (Browser & Modern JS):

- export / import

```
// math.js
export function add(a,b) { return a+b; }

// main.js
import {add} from "./math.js";
console.log(add(2,3));
```

⑧ CommonJS (Node.js)

- module.exports / require

```
// math.js
module.exports.add = (a,b) => a+b;

// main.js
const main = require('./math');
console.log(math.add(2,3));
```