WEEK-8 NOTES

▼ OOPs in JS

What is OOP in JavaScript?

JavaScript is **prototype-based**, not class-based like Java or C++.

Even ES6 `class` syntax is just **syntactic sugar over prototypes**.

OOP in JS helps you:

- Model real-world entities
- Reuse code
- Organize large applications
- Achieve abstraction, encapsulation, inheritance, and polymorphism

Constructor Functions

A **constructor function** is a normal function used with `new` to create objects.

Rules

- Function name starts with **capital letter**
- Called using `new`
- `this` refers to the newly created object

Example

```
functionPerson(name, age) {
this.name = name;
this.age = age;
```

```
}

const p1 =new Person("Nisharg",21);
const p2 =new Person("Amit",22);

console.log(p1.name);// Nisharg
```

Prototype & Prototypal Inheritance

Every JavaScript object has a **hidden property** called `[[Prototype]]`

Accessible via:

- `__proto__` (not recommended)
- `Object.getPrototypeOf(obj)`

Bad Practice

```
functionPerson(name) {
this.name = name;
this.sayHello =function () {
console.log("Hello");
 };
}
```

Correct Way (Using Prototype)

```
functionPerson(name) {
this.name = name;
}

Person.prototype.sayHello =function () {
console.log("Hello, my name is " +this.name);
};

const p1 =newPerson("Nisharg");
const p2 =newPerson("Amit");
```

```
p1.sayHello();
```

ES6 Classes

```
class Person {
constructor(name, age) {
this.name = name;
this.age = age;
  }

greet() {
console.log("Hi, I'm " +this.name);
  }
}

const p =new Person("Nisharg",21);
p.greet();
```

extends

`Used for inheritance`

super

`Calls parent constructor or methods`

Example

```
class Animal {
constructor(name) {
this.name = name;
  }

speak() {
console.log("Animal speaks");
  }
}
```

```
class Dog extends Animal {
constructor(name, breed) {
super(name);// must be called first
this.breed = breed;
 }

speak() {
console.log("Dog barks");
 }
}

const d =new Dog("Buddy","Labrador");
d.speak();
```

Static Methods

- Belongs to **class**
- Not accessible via object
- Used for utility/helper functions

Example

```
classMathUtil {
staticadd(a, b) {
return a + b;
 }
}
console.log(MathUtil.add(5,3));// 8
```

Encapsulation

- Hiding internal data
- Exposing only required methods

Using Closures (Old way)

```
function BankAccount(balance) {
let _balance = balance;

this.getBalance = function () {
return _balance;
  };
}

const acc = new BankAccount(5000);
console.log(acc._balance);// undefined
```

Using Private Fields (Modern)

```
class BankAccount {
  #balance;

constructor(balance) {
this.#balance = balance;
  }

getBalance() {
return this.#balance;
  }
}

const acc =new BankAccount(10000);
acc.getBalance();// 10000
```

Polymorphis

`Same method name → different behavior`

Method Overriding

```
class Shape {
area() {
return 0;
```

```
  }
}

class Rectangle extends Shape {
area() {
return 10 *5;
  }
}

class Circle extends Shape {
area() {
return 3.14 *5 *5;
  }
}

const shapes = [newRectangle(),newCircle()];
shapes.forEach(s =>console.log(s.area()));
```

JS does **NOT support method overloading** like Java Only **last defined method** is considered.

▼ Callback Functions

A function passed as an argument to another function and executed later.

Example

```
function greet(name, callback) {
console.log("Hello " + name);
callback();
}

greet("Nisharg",function () {
console.log("Callback executed");
});
```

Async Callback Example

```
setTimeout(function () {
console.log("Executed later");
},2000);
```

Callback Hell (Problem)

Nested callbacks make code:

- Hard to read
- Hard to debug
- Hard to maintain

Example (Callback Hell)

```
setTimeout(() => {
console.log("Step 1");

setTimeout(() => {
console.log("Step 2");

setTimeout(() => {
console.log("Step 3");
   },1000);

 },1000);

},1000);
```

This is called **Pyramid of Doom**

▼ Promises

A promise represents a value that will be:

- **Fulfilled**
- **Rejected**
- **Pending**

Promise States

```
pending → fulfilled
pending → rejected
```

Creating a Promise

```
const promise =new Promise((resolve, reject) ⇒ {
let success =true;

if (success) {
resolve("Done");
 }else {
reject("Error");
 }
});
```

then() and catch()

Example

```
promise
 .then(result ⇒ {
console.log(result);
 })
 .catch(error ⇒ {
console.log(error);
 });
```

Promise Chaining

```
fetchData()
 .then(data ⇒ processData(data))
 .then(result ⇒ saveResult(result))
```

```
  .then(() => console.log("Done"))
  .catch(err => console.log(err));
```

Each `then()` returns a new promise

Promise.all()

```
Promise.all([p1, p2, p3])
  .then(results =>console.log(results))
  .catch(err =>console.log(err));
```

Behavior:

- All must succeed
- If **any fails → reject immediately**

Use case: Multiple API calls that all are required

Promise.allSettled()

```
Promise.allSettled([p1, p2])
  .then(results =>console.log(results));
```

Waits for **all promises**

Returns status of each

```
[
  {status:"fulfilled",value: ... },
  {status:"rejected",reason: ... }
]
```

Promise.race()

```
Promise.race([p1, p2])
  .then(res =>console.log(res))
  .catch(err =>console.log(err));
```

First settled promise wins

Can be fulfilled or rejected

Promise.any()

```
Promise.any([p1, p2])
  .then(res =>console.log(res))
  .catch(err =>console.log(err));
```

First **fulfilled** promise wins

Rejects only if **all fail**

Returns `AggregateError`

▼ Async / Await

What is `async` ?

- Makes a function return a promise

What is `await` ?

- Pauses execution **inside async function only**

Example

```
async function demo() {
return"Hello";
}

demo().then(val => console.log(val));
```

Using `await`

```
function delay() {
return new Promise(resolve => {
setTimeout(() =>resolve("Done"),2000);
 });
}
```

```
async function run() {
const result =await delay();
console.log(result);
}

run();
```

Error Handling in Async Code

Using try/catch

```
async function test() {
try {
let res =await Promise.reject("Failed");
 }catch (err) {
console.log(err);
 }
}

test();
```

Comparison between async/await and promise

Example (GET)

```
<script>
fetch("https://jsonplaceholder.typicode.com/users")
   .then(res ⇒ res.json())
   .then(data ⇒console.log(data))
   .catch(err ⇒console.log(err));
</script>
```

Using async/await

```
async function getUsers() {
try {
const res =await fetch("https://jsonplaceholder.typicode.com/users");
```

```
const data =await res.json();
console.log(data);
 }catch (err) {
console.log(err);
 }
}

getUsers();
```

▼ ES6 MODULES

- Each file is its own module
- Variables are **module-scoped**
- `import` / `export` are **static**
- Modules run in **strict mode**
- Browser requires `type="module"`

Named Exports

Exporting

```
// math.js
export const add = (a, b) => a + b;
export const sub = (a, b) => a - b;

//or

const add = (a, b) => a + b;
const sub = (a, b) => a - b;

export { add, sub };
```

Importing

```
import { add, sub } from "./math.js";

console.log(add(2,3));
```

Default Export

```
// logger.js
exportdefaultfunctionlog(message) {
console.log(message);
}
```

Import default (ANY NAME ALLOWED)

```
import log from"./logger.js";
log("Hello");
```

`Default export = one per file`

Import everything

```
import * as math from"./math.js";

math.add(2,3);
```

Rename imports (Alias)

```
import { add as sum }from"./math.js";
sum(2,3);
```

▼ String

`Strings in JavaScript are immutable → methods return new strings`

BASIC PROPERTIES & ACCESS

Method / Property	Description	Example
`length`	Returns length of string	`"abc".length → 3`
`charAt(i)`	Character at index	`"abc".charAt(1) → b`
`charCodeAt(i)`	Unicode value	`"A".charCodeAt(0) → 65`
`at(i)`	Character (supports -ve index)	`"abc".at(-1) → c`
`[i]`	Bracket access	`"abc"[1] → b`

CASE CONVERSION

Method	Description	Example
`toUpperCase()`	Convert to uppercase	`"js".toUpperCase()`
`toLowerCase()`	Convert to lowercase	`"JS".toLowerCase()`

SEARCHING METHODS

Method	Description	Returns
`indexOf()`	First occurrence index	`number / -1`
`lastIndexOf()`	Last occurrence index	`number / -1`
`includes()`	Check substring	`true / false`
`startsWith()`	Starts with value	`true / false`
`endsWith()`	Ends with value	`true / false`
`search()`	Regex search	`index / -1`
`match()`	Regex match	`array / null`
`matchAll()`	All regex matches	`iterator`

EXTRACTING STRING PARTS

Method	Description	Note
`slice(start,end)`	Extract part	Supports negative
`substring(start,end)`	Extract part	No negative

MODIFY

Method	Description	Example
`replace()`	Replace first match	`"a a".replace("a","b")`
`replaceAll()`	Replace all	`"a a".replaceAll("a","b")`
`concat()`	Join strings	`"a".concat("b")`
`repeat()`	Repeat string	`"ha".repeat(3)`
`padStart()`	Pad at start	`"5".padStart(3,"0")`
`padEnd()`	Pad at end	`"5".padEnd(3,"0")`

TRIMMING SPACES

Method	Description
`trim()`	Remove both sides
`trimStart()`	Remove left
`trimEnd()`	Remove right

SPLIT & JOIN

Method	Description
`split()`	Convert string → array
`join()`	Convert array → string

COMPARISON & CONVERSION

Method	Description
`localeCompare()`	Compare strings
`toString()`	Convert to string
`valueOf()`	Primitive value

▼ Math

CONSTANTS

Property	Description	Value
Math.PI	Value of π	3.14159
Math.E	Euler’s number	2.718
Math.SQRT2	√2	1.414
Math.SQRT1_2	√½	0.707
Math.LN2	ln(2)	0.693
Math.LN10	ln(10)	2.302
Math.LOG2E	$\log_2(e)$	1.442
Math.LOG10E	$\log_{10}(e)$	0.434

ROUNDING METHODS

Method	Description	Example
Math.round(x)	Nearest integer	round(4.6) → 5
Math.floor(x)	Downwards	floor(4.9) → 4
Math.ceil(x)	Upwards	ceil(4.1) → 5
Math.trunc(x)	Remove decimal	trunc(4.9) → 4

POWER & ROOT

Method	Description	Example
Math.pow(x,y)	x^y	pow(2,3) → 8
Math.sqrt(x)	Square root	sqrt(25) → 5
Math.cbrt(x)	Cube root	cbrt(27) → 3

ABSOLUTE & SIGN

Method	Description
Math.abs(x)	Absolute value
Math.sign(x)	-1, 0, or 1

MIN / MAX

Method	Description
`Math.min()`	Smallest value
`Math.max()`	Largest value

RANDOM NUMBERS

Method	Description
`Math.random()`	Random number (0–<1)

LOGARITHMS

Method	Description
`Math.log(x)`	Natural log
`Math.log10(x)`	Base-10 log
`Math.log2(x)`	Base-2 log

TRIGONOMETRY (Radians)

Method	Description
`Math.sin(x)`	Sine
`Math.cos(x)`	Cosine
`Math.tan(x)`	Tangent
`Math.asin(x)`	Inverse sine
`Math.acos(x)`	Inverse cosine
`Math.atan(x)`	Inverse tangent

▼ Date/Time Operations

CREATING DATE OBJECTS

Syntax	Description
`new Date()`	Current date & time
`new Date(ms)`	From timestamp
`new Date(dateString)`	From string
`new Date(y,m,d)`	Year, month, day

Syntax	Description
`new Date(y,m,d,h,min,s)`	Full date

GET METHODS

Method	Returns
`getFullYear()`	Year (YYYY)
`getMonth()`	Month (0–11)
`getDate()`	Day of month
`getDay()`	Day of week (0–6)
`getHours()`	Hours
`getMinutes()`	Minutes
`getSeconds()`	Seconds
`getMilliseconds()`	Milliseconds
`getTime()`	Timestamp (ms)

SET METHODS

Method	Purpose
`setFullYear(y)`	Set year
`setMonth(m)`	Set month
`setDate(d)`	Set day
`setHours(h)`	Set hour
`setMinutes(m)`	Set minutes
`setSeconds(s)`	Set seconds
`setMilliseconds(ms)`	Set milliseconds
`setTime(ms)`	Set timestamp

UTC METHODS (TIME ZONE SAFE)

Method	Description
`getUTCFullYear()`	UTC year
`getUTCMonth()`	UTC month
`getUTCHours()`	UTC hours

Method	Description
`setUTCDate()`	Set UTC date

FORMATTING DATE & TIME

Method	Output
`toString()`	Full string
`toDateString()`	Date only
`toTimeString()`	Time only
`toISOString()`	ISO format
`toLocaleString()`	Locale date & time
`toLocaleDateString()`	Locale date
`toLocaleTimeString()`	Locale time

TIMESTAMP & NOW

Method	Description
`Date.now()`	Current timestamp
`date.getTime()`	Timestamp of date

DATE COMPARISON & DIFFERENCE

Operation	Description
`date1 > date2`	Compare dates
`date2 - date1`	Difference (ms)

DATE PARSING

Method	Description
`Date.parse()`	Parse date string