Week 8 - Javascript

Created by Dakshil Gorasiya

OOPS

JavaScript uses **prototype-based inheritance**.

ES6 `class` syntax is only **syntactic sugar** over this system — it does **not** introduce true classical inheritance.

Working of inheritance

Every object has an internal `[[Prototype]]` link.

When a property or method is accessed:

- JS looks on the object itself
- If not found, it looks on its prototype
- This continues up the prototype chain

Methods are **shared via prototypes**, not copied to each object.

ES6 `class` and `extends`

`class` and `extends` provide a cleaner way to create prototype chains.

What happens internally:

- Parent methods are stored on `Base.prototype`
- `extends` links `Derived.prototype` to `Base.prototype`
- Objects of `Derived` can access parent methods through the prototype chain

`super()`:

- Calls the parent constructor
- Required before using `this` in a derived class
- Internally equivalent to calling `Base.call(this, ...)`

Relation to ES5 (Behind the Scenes)

ES6 inheritance is equivalent to ES5 prototype code:

- Constructor functions create objects
- Parent constructor is invoked using `call`
- Prototype linking is done using `Object.create`
- Constructor reference is restored manually

ES6 just **automates and hides** these steps.

Key Points

- JavaScript is **not class-based**, it is **prototype-based**
- ES6 classes do not change how inheritance works
- Methods live on prototypes, not on instances

To create a class

```
class class_name {}
```

To create constructor

```
constructor(a, b) {
   this.a = a;
   this.b = b;
}
```

To create private field

```
#privateField;
```

To create property

```
get name() {
  return this.#name;
}
set name(newName) {
```

```
  this.#name = newName;
}
```

To create static property

```
static staticProperty = "value";
```

To create static method

```
static staticMethod(){
   // function body
}
```

ES6

`import` / `export`

ES6 modules allow JavaScript code to be **split into files** and reused safely.

Each file is treated as its **own module scope** (no global leakage).

Export

Exports define what a file makes available to other files.

Named export

- Multiple values can be exported from one file
- Import name must match the exported name

```
export { Calculator };
```

Default export

- Only one default export per file
- Import name can be anything

```
export default Calculator;
```

Import

Named import

```
import { Calculator } from "./Calculator.js";
```

- Uses curly braces
- Name must be exact (case-sensitive)

Default import

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

- No curly braces
- Name can be changed freely

Rules & Behavior

- `import` and `export` work only in **module context**
- Imports are **read-only bindings**
- Code is executed **once**, even if imported multiple times
- Modules run in **strict mode by default**

File & Environment

- File extension (`.js`) is required in browser imports
- In HTML:

```
<script type="module" src="app.js"></script>
```

- In Node.js:
  - Use `"type": "module"` in `package.json`

When to Use Named vs Default Export

Named export:

- Multiple exports
- Clear API surface

- Better for large modules

Default export:

- Single main value
- Simpler imports
- Common for utilities or classes

`async` / `await`

`async` and `await` provide a **cleaner way to work with Promises**.

They make asynchronous code look and behave like synchronous code, without blocking the thread.

Async

- Declared using the `async` keyword
- Always returns a **Promise**
- A returned value is automatically wrapped in `Promise.resolve()`
- A thrown error becomes `Promise.reject()`

```
async function getData() {
  return "hello";
}
```

Await

- `await` pauses execution **inside an async function**
- Waits for a Promise to settle
- Returns the resolved value
- If the Promise rejects, it throws an error

```
const result = await somePromise;
```

`await` cannot be used outside an `async` function.

Converting `.then()` to `async / await`

Promise-based code:

```
fetch(url).then(res => res.json()).then(data => console.log(data));
```

Async / await version:

```
const response = await fetch(url);
const data = await response.json();
console.log(data);
```

Same behavior, better readability.

Error Handling with `try...catch`

- Replaces `.catch()`
- Handles rejected Promises like synchronous errors

```
try {
  const res = await fetch(url);
  const data = await res.json();
} catch (err) {
  console.error(err);
}
```

Execution Behavior

- `await` pauses only the **async function**, not the whole program
- JavaScript event loop continues running
- Non-blocking by design

Array Methods

These are **high-order array methods** used to process arrays in a clean and functional way.

`forEach` – Perform Action on Each Element

- Used for **side effects** (logging, DOM updates, API calls)
- Does **not return** a new array

- Return value is always `undefined`

```
arr.forEach(num ⇒ console.log(num));
```

Use when you want to **do something**, not transform data.

`map` – Transform Array

- Returns a **new array**
- Each element is transformed
- Output array length is same as input

```
const squared = arr.map(n ⇒ n * n);
```

Use when you want to **change each value**.

`filter` – Select Elements

- Returns a **new array**
- Keeps elements that satisfy a condition
- Output length may be smaller

```
const evens = arr.filter(n ⇒ n % 2 === 0);
```

Use when you want to **remove unwanted values**.

`reduce` – Combine into Single Value

- Returns **one value** (number, object, array, etc.)
- Uses an accumulator
- Can replace many loops

```
const sum = arr.reduce((acc, n) ⇒ acc + n, 0);
```

Use when you want to **aggregate or compute a result**.

Other note

- These methods iterate **from left to right**

- Callback receives `(currentValue, index, array)`
- Original array remains unchanged

Date, math, string

String

In JavaScript, strings represent **text data** and are **immutable** (any change creates a new string).

Ways to Create Strings

Most of the time, strings are created as **primitives**.

```
let s1 = "Hello";
let s2 = String("Hello");
```

Using `new String()` creates a **String object**, which is rarely needed.

```
let s3 = new String("Hello"); // object, not primitive
```

Best practice: **always use primitive strings**.

Primitive vs String Object

- Primitive string → `typeof === "string"`
- String object → `typeof === "object"`
- JavaScript automatically **boxes** primitives when methods are called
- After method execution, it converts back to primitive

Method, Property, static methods

Method / Property	Purpose / Behavior
`String.fromCharCode(...codes)`	Create string from UTF-16 codes

Method / Property	Purpose / Behavior
`String.fromCodePoint(...codes)`	Create string from Unicode code points
`String.raw(\`string\`)`	Returns raw string (ignores escapes)
`at(index)`	Get character (supports negative index)
`charAt(index)`	Get character at index
`charCodeAt(index)`	UTF-16 code of character
`codePointAt(index)`	Unicode code point
`includes(searchString, position>)`	Check if substring exists
`indexOf(searchValue, fromIndex?)`	First index of substring
`lastIndexOf(searchValue, fromIndex?)`	Last index of substring
`startsWith(searchString, position?)`	Check prefix
`endsWith(searchString, position?)`	Check suffix
`search(regex)`	Regex-based search
`slice(start, end?)`	Extract substring (supports negative index)
`substring(start, end?)`	Extract substring (no negative index)
`concat(...string)`	Join strings
`replace(pattern, replacement)`	Replace first match
`replaceAll(pattern, replacement)`	Replace all matches
`repeat(count)`	Repeat string
`trim()`	Remove whitespace (both ends)
`trimStart()`	Remove leading whitespace
`trimEnd()`	Remove trailing whitespace
`padStart(targetLength, padString?)`	Pad at start
`padEnd(targetLength, padString?)`	Pad at end
`toLowerCase()`	Convert to lowercase
`toUpperCase()`	Convert to uppercase
`localeCompare(compareString, locales?, options?)`	Locale-based comparison
`match(regex)`	Match using regex

Method / Property	Purpose / Behavior
matchAll(regex)	Get all regex matches
normalize(form?)	Unicode normalization
isWellFormed()	Check Unicode validity
split(separator, limit?)	Convert string to array
toString()	Convert to string
valueOf()	Get primitive value
length	Number of UTF-16 units

Math

Methods

Method	Description
abs(number)	Absolute value
acos(number)	Arc cosine (radians)
acosh(number)	Inverse hyperbolic cosine
asin(number)	Arc sine
asinh(number)	Inverse hyperbolic sine
atan(number)	Arc tangent
atan2(y, x)	Angle between x-axis and point
atanh(number)	Inverse hyperbolic tangent
sin(number)	Sine (radians)
sinh(number)	Hyperbolic sine
cos(number)	Cosine (radians)
cosh(number)	Hyperbolic cosine
tan(number)	Tangent (radians)
tanh(number)	Hyperbolic tangent
cbrt(number)	Cube root
ceil(number)	Round up
floor(number)	Round down
round(number)	Round to nearest integer
trunc(number)	Remove decimal part

Method	Description
`exp(number)`	e raised to power
`expm1(number)`	e^x − 1
`log(number)`	Natural logarithm
`log1p(number)`	ln(1 + x)
`log2(number)`	Base-2 logarithm
`log10(number)`	Base-10 logarithm
`hypot(...numbers)`	Square root of sum of squares
`imul(a, b)`	32-bit integer multiplication
`max(...numbers)`	Maximum value
`min(...numbers)`	Minimum value
`pow(base, exponent)`	Power calculation
`random()`	Random number [0, 1)
`sign(number)`	Sign of number
`sqrt(number)`	Square root

Static property

Property	Description
`PI`	Value of π
`E`	Euler's number
`LN2`	Natural log of 2
`LN10`	Natural log of 10
`LOG2E`	Base-2 log of e
`LOG10E`	Base-10 log of e
`SQRT1_2`	Square root of 1/2
`SQRT2`	Square root of 2

- `Math` cannot be instantiated
- Trigonometric methods use **radians**
- `random()` never returns 1

Date

JavaScript represents dates using a **single number**:

the count of **milliseconds since January 1, 1970, 00:00:00 UTC** (Unix Epoch).

This internal value is always stored in **UTC**, never in local time.

How Date Storage & Conversion Works

- Internally: **UTC timestamp (milliseconds)**
- Display:
  - Local time → when using `toString()`, `getHours()`, etc.
  - UTC time → when using `toUTCString()`, `getUTCHours()`, etc.
- JavaScript automatically converts between **UTC ↔ Local timezone** when needed

Example :

- Stored value: `1700000000000`
- Displayed differently depending on:
  - Your system timezone
  - Method used (local vs UTC)

Date Constructors

Constructor	Description
`new Date()`	Current date & time (from system clock)
`new Date(dateString)`	Parse date string
`new Date(timestamp)`	From milliseconds since epoch
`new Date(dateObject)`	Clone an existing date
`new Date(year, monthIndex)`	Month is **0-based**
`new Date(year, monthIndex, day, hours?, minutes?, seconds?, ms?)`	Detailed local date

Static Methods (Date)

Method	Description
`Date.now()`	Current timestamp (ms since epoch)
`Date.parse(dateString)`	Convert date string → timestamp
`Date.UTC(year, monthIndex, day?, hours?, minutes?, seconds?, ms?)`	UTC timestamp

Get Methods (Local & UTC)

Method	Description
`getDate()` / `getUTCDate()`	Day of month (1–31)
`getDay()` / `getUTCDay()`	Day of week (0–6, Sunday = 0)
`getFullYear()` / `getUTCFullYear()`	4-digit year
`getMonth()` / `getUTCMonth()`	Month (0–11)
`getHours()` / `getUTCHours()`	Hours
`getMinutes()` / `getUTCMinutes()`	Minutes
`getSeconds()` / `getUTCSeconds()`	Seconds
`getMilliseconds()` / `getUTCMilliseconds()`	Milliseconds
`getTime()`	Timestamp in ms
`getTimezoneOffset()`	Difference from UTC (minutes)

Set Methods (Local & UTC)

Method	Description
`setDate(number)` / `setUTCDate(number)`	Set day of month
`setFullYear(number)` / `setUTCFullYear(number)`	Set year
`setMonth(number)` / `setUTCMonth(number)`	Set month (0–11)
`setHours(number)` / `setUTCHours(number)`	Set hours
`setMinutes(number)` / `setUTCMinutes(number)`	Set minutes
`setSeconds(number)` / `setUTCSeconds(number)`	Set seconds
`setMilliseconds(number)` / `setUTCMilliseconds(number)`	Set milliseconds
`setTime(timestamp)`	Set from timestamp

Conversion & Formatting Methods

Method	Description	Example Output
`toString()`	Full local date-time string	`Tue Nov 21 2023 03:43:20 GMT+0530 (India Standard Time)`
`toDateString()`	Local date only	`Tue Nov 21 2023`
`toTimeString()`	Local time only	`03:43:20 GMT+0530 (India Standard Time)`
`toISOString()`	ISO 8601 UTC format	`2023-11-20T22:13:20.000Z`
`toUTCString()`	UTC readable string	`Mon, 20 Nov 2023 22:13:20 GMT`
`toJSON()`	JSON-safe UTC format	`2023-11-20T22:13:20.000Z`
`toLocaleString()`	Locale date & time	`11/21/2023, 3:43:20 AM`
`toLocaleDateString()`	Locale date only	`11/21/2023`
`toLocaleTimeString()`	Locale time only	`3:43:20 AM`
`valueOf()`	Primitive timestamp (ms)	`1700518400000`