JavaScript Classes and Function

What is a Class?

A **class** defines a *blueprint* for creating objects with properties and methods.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}!`);
  }
}
```

Usage:

```
const p = new Person("Alice");
p.greet(); // Hello, Alice!
```

Constructor and Parameters

The constructor() runs automatically when new is used.

The constructor can accept parameters to set up the object.

```
class Student {
  constructor(name, grade) {
    this.name = name;
    this.grade = grade;
  }
  info() {
    console.log(`${this.name} is in grade ${this.grade}`);
  }
}

const s = new Student("Ken", 10);
s.info();
```

Handling Errors Inside a Class

Classes can throw or handle errors in the constructor.

```
class SafeCalculator {
  constructor(value) {
    if (typeof value !== "number") {
      throw new Error("Value must be a number!");
    }
    this.value = value;
  }
}

try {
  const calc = new SafeCalculator("abc");
} catch (err) {
  console.error("❌ Error:", err.message);
}
```

Example: Database with Error Callback

Here's a class that opens a database and optionally reports errors through a callback.

```
class DBExample {
  constructor(path, onError) {
    console.log(`Opening database at: ${path}`);

    const error = Math.random() > 0.5 ? new Error('DB connection failed!') : null;

    if (error) {
      if (typeof onError === 'function') {
        onError(error);
      } else {
        throw error;
      }
    } else {
      console.log('Database opened successfully.');
    }
  }
}

const db = new DBExample('.', (err) => {
  console.error('Failed to open DB:', err.message);
  process.exit(1);
});
```

Key Ideas

Concept	Description
`constructor()`	Initializes a new instance when creating an object with `new`.
`throw new Error()`	Signals that something went wrong by creating and throwing an error.
Callback function	An optional function passed to handle errors or events dynamically.
`try { ... } catch (err) { ... }`	Captures and handles errors thrown inside the `try` block.

JavaScript Functions

A **function** is a reusable block of code designed to perform a specific task.

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}

greet("Alice"); // Hello, Alice!
```

Functions help you avoid repetition and organize logic clearly.

Function Declaration (Normal Function)

```
function add(a, b) {
  return a + b;
}

console.log(add(2, 3)); // 5
```

- Hoisted (can be called before it’s defined)
- Has its own context
- Commonly used for general logic or reusability

Function Expression (not Hoisted)

You can also assign a function to a variable.

```
const multiply = function (a, b) {
  return a * b;
};

console.log(multiply(3, 4)); // 12
```

This is not hoisted — must be defined before use.

Arrow (Lambda) Function

Introduced in ES6 — shorter syntax for writing functions.

```
const divide = (a, b) => a / b;
console.log(divide(10, 2)); // 5
```

Called a lambda or arrow function because of the => arrow.

Example: Callback Function with Arrow

```
const numbers = [1, 2, 3, 4];

numbers.forEach((n) => console.log(n * 2)); // 2, 4, 6, 8
```

Simplifies anonymous function usage (no need for function (n) { ... }).

Higher-Order Functions (HOF)

A **Higher-Order Function** is a function that:

1. Takes another function as an argument, or
2. Returns a function as its result.

In short: functions that work with other functions!

HOF Example 1 — Function as Argument

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}

function repeatAction(action, times) {
  for (let i = 0; i < times; i++) {
    action(`Run #${i + 1}`);
  }
}
repeatAction(greet, 3);
```

Output:

```
Hello, Run #1!
Hello, Run #2!
Hello, Run #3!
```

HOF Example 2 — Function Returning Function

```
function makeMultiplier(factor) {
  return function (n) {
    return n * factor;
  };
}

const double = makeMultiplier(2);
console.log(double(5)); // 10
```

Here, makeMultiplier returns a new function.

Common HOFs in JavaScript

These are the most frequently used HOFs.

Function	Purpose	Example
`map()`	Transforms each element	`[1, 2, 3].map(x => x * 2)` → `[2, 4, 6]`
`filter()`	Keeps elements that match a condition	`[1, 2, 3].filter(x => x > 1)` → `[2, 3]`
`reduce()`	Combines all values into one	`[1, 2, 3].reduce((a, b) => a + b, 0)` → `6`
`forEach()`	Runs a function for each element	`[1, 2, 3].forEach(x => console.log(x))`

Feature	forEach()	map()
Purpose	Executes a function for each element (side effects)	Transforms each element and creates a new array
Returns	`undefined`	A **new array** with transformed values
Modifies Original Array	❌ No	❌ No
Chainable	❌ No	✅ Yes
Use Case	Logging, updating external data, debugging	Creating a new array from existing data
Example	`[1,2,3].forEach(x => console.log(x))`	`[1,2,3].map(x => x * 2)` → `[2,4,6]`

map example 1

Let's say we need to transform `existingConversation`

```
message = {
  content = "Hello",
  aiMessage = false
}
aiMessage = {
  content = "What's up",
  aiMessage = true
}
existingConversation = {
  messages: [message, aiMessage]
}
```

to a JSON object:

```
{
  content: content,
  role: string
}

`content` is the content in the message or aiMessage, and role is "assistant" for aiMessage and "user" for normal message.
```

The `...` is the spread operator to insert each element into previousConversationMessages one by one.

Without the spread operator, we will the array inside array.

```
const a = [1, 2, 3];
b.push(a);
// b becomes: [ [1,2,3] ]   ← array inside array

const a = [1, 2, 3];
b.push(...a);
// b becomes: [1, 2, 3]    ← elements added individually
```

map example 2

We can transform a JSON object to a different format using the map operator.

```
conversation = {
  messages: [
    { id:1, content: 'Hello', aiMessage: false },
    { id:2, content: "What's up", aiMessage: true }
  ]
}
```

In this example, the index in `(m, index)` automatically generates an index value from 0.

```
converted = conversation.messages.map((m, index) => ({
  key: m.id,
  content: m.content,
  aiMessage: m.aiMessage,
  animate: (index === conversation.messages.length - 1 && m.aiMessage)
}))
```