EventEmitter

- A programming pattern where an object (emitter) emits events, and other objects (listeners) subscribe to those events.
- It follows the Publish–Subscribe or Observer design pattern.

- Promotes loose coupling (the emitter doesn’t need to know about the listener).
- Makes applications more modulr and scalable.

Example use cases:

- Node.js EventEmitter (handling data, error, close events in streams).
- Custom events in frontend frameworks for UI interactions.

Example of EventEmitter

```
class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(event, listener) {
    if (!this.events[event]) this.events[event] = [];
    this.events[event].push(listener);
  }

  off(event, listener) {
    if (!this.events[event]) return;
    this.events[event] = this.events[event].filter(l => l !== listener);
  }

  emit(event, ...args) {
    if (!this.events[event]) return;
    this.events[event].forEach(listener => listener(...args));
  }
}


const emitter = new EventEmitter();
const greet = (name) => console.log(`Hello, ${name}!`);

emitter.on("greet", greet);
emitter.emit("greet", "Ajay");
emitter.off("greet", greet);
emitter.emit("greet", "Ajay");
```

Memoization

- An optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again.
- Works best with pure functions (same input → same output).

- Saves time by avoiding redundant calculations.
- Improves performance in CPU-heavy tasks.

- Example use cases:
  - Caching recursive function results (e.g., Fibonacci, factorial).
  - Avoiding duplicate API/database queries with the same parameters.
  - Storing results in data-heavy UI apps (charts, filters).

Example of Memoization

```
function memoize(fn) {
  const cache = new Map();
  return function(...args) {
    const key = JSON.stringify(args);
    if (cache.has(key)) {
      return cache.get(key);
    }
    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
}

const slowSquare = (n) => {
  console.log("Computing...");
  return n * n;
};

const fastSquare = memoize(slowSquare);
console.log(fastSquare(5));
console.log(fastSquare(5));
```

Async/Await

- A syntactic sugar over Promises introduced in ES2017.
- async declares a function that returns a Promise.
- await pauses execution inside an async function until the Promise resolves/rejects.

- Makes asynchronous code look synchronous.
- Easier error handling with try/catch.
- Improves readability compared to nested callbacks or .then() chains.

Example use cases:

- Fetching API data in web apps.
- Database queries in Node.js.
- File read/write operations.

Example of async/await

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Data received"), 1000);
  });
}

async function getData() {
  console.log("Fetching...");
  const result = await fetchData();
  console.log(result);
}

getData();

Ans: Data received (after 1 sec)
```

Call, Apply, Bind

- Methods of all JavaScript functions that allow explicit control of this.
- call → invokes immediately, arguments passed one by one.
- apply → invokes immediately, arguments passed as an array.
- bind → returns a new function with this fixed, but doesn't invoke immediately.

- To reuse functions with different objects.
- To explicitly set context in event handling or callbacks.
- To achieve partial application or function borrowing.

Example use cases:

  - Using array methods on array-like objects (arguments).
  - Borrowing methods between objects.
  - Binding this in React class components.

Example of Call, Apply, Bind

```
function greet(city) {
  console.log(`${this.name} from ${city}`);
}
greet.call({ name: "Ajay" }, "Pune");


function greet(city, country) {
  console.log(`${this.name} from ${city}, ${country}`);
}
greet.apply({ name: "Ajay" }, ["Pune", "India"]);


function greet(city) {
  console.log(`${this.name} from ${city}`);
}
const boundGreet = greet.bind({ name: "Ajay" }, "Pune");
boundGreet();
```