Here's a **technical note with simple examples** for each topic in the picture, written in **easy words** and kept **short** for quick learning:

🧬 JavaScript Anatomy

1. **History of JavaScript**
   JS was made in 1995 to make web pages interactive.
   ✅ Example: Clicking a button to show a popup.
2. **Know your browser**
   Browsers run JS using engines like V8 (Chrome) or SpiderMonkey (Firefox).
   ✅ Example: Inspect elements using Chrome DevTools.
3. **The rendering engine**
   Draws HTML and CSS on screen (like a painter).
   ✅ Example: Displays a styled webpage when loaded.
4. **JavaScript at runtime**
   JS runs in the browser when the page loads.
   ✅ Example: alert("Welcome!") runs when page opens.

🧱 JavaScript Foundations

5. **JavaScript engine**
   The part of the browser that runs JS.
   ✅ Example: V8 executes console.log("Hello").
6. **Call stack**
   Keeps track of what function is being run.
   ✅ Example:
   ```
   function one() { two(); }
   function two() { console.log("2"); }
   one(); // Stack: one → two
   ```
7. **Memory heap**
   Where JS stores objects and data.
   ✅ Example:
   ```
   let obj = { name: "Ali" }; // Stored in heap
   ```
8. **Memory leak**
   Happens when data isn't released from memory.
   ✅ Example: Unused DOM elements still referenced in JS.
9. **Stack overflow**
   Too many nested calls crash the stack.
   ✅ Example:
   ```
   function loop() { loop(); }
   loop(); // Stack overflow
   ```
10. **Garbage collection**
    JS clears unused memory automatically.
    ✅ Example: After let x = null, memory is freed.

11. **Synchronous**
    JS runs one task at a time.
    ✅ Example:
    console.log("1");
    console.log("2");
    // Output: 1 2
12. **Callback Queue**
    Stores async tasks waiting to run.
    ✅ Example:
    setTimeout(() => console.log("Hello"), 0);
13. **Event loop**
    Moves tasks from callback queue to call stack.
    ✅ Example: Handles setTimeout, promises, events.
14. **3 Ways to Promise**

- **Callback**:
- setTimeout(() => console.log("Done"), 1000);
- **Promise**:
- fetch(url).then(res => console.log(res));
- **Async/Await**:
- async function load() { await fetch(url); }

🔍 **JavaScript Under the Hood**

15. **Execution Context**
    Each function creates its own context (memory + code).
    ✅ Example:

```
function run() {
  let x = 10;
}
```

16. **Hoisting**
    Declarations are moved to the top.
    ✅ Example:

```
console.log(a); // undefined
var a = 5;
```

17. **Lexical Environment**
    Scope defined where variables live in code.
    ✅ Example: Functions access parent variables.
18. **Scope Chain**
    JS searches variables from inner to outer scopes.
    ✅ Example:

```
let a = 1;
function f() { console.log(a); }
```

19. **This**
    Refers to the object that is calling the function.
    ✅ Example:

```
const obj = {
  name: "Yumna",
  sayHi() { console.log(this.name); }
};
```

20. **Let and Const**
    Block scoped; better than var.
    ✅ Example:

```
let x = 5; const y = 10;
```

21. **Arrow Function**
    Short syntax, no own this.
    ✅ Example:

```
const add = (a, b) => a + b;
```

22. **Call, Apply, Bind**
    Set the value of this manually.
    ✅ Example:

```
function show() { console.log(this.name); }
show.call({ name: "Ali" });
```

🧠 The 2 Pillars: Closures & Prototypes

23. **Closure**
    A function remembers its outer variables.
    ✅ Example:

```
function outer() {
  let x = 1;
  return () => console.log(x);
}
```

24. **Prototypes**
    Used for inheritance in JS.
    ✅ Example:

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function() {
  console.log("Hi " + this.name);
};
```

🧩 Extras

25. **Function Constructor**
    Creates functions from strings (not safe).
    ✅ Example:

```
const sum = new Function('a', 'b', 'return a + b');
```

26. **Prototype vs proto**
- prototype: On constructor function.
- __proto__: On object, links to prototype.

✅ Example:

```
obj.__proto__ === Object.prototype
```

27. **Callback Object**
An object passed as a callback (less common).
✅ Example:
Used in some libraries instead of plain functions.

28. **HOC (Higher Order Function)**
Function that takes or returns another function.
✅ Example:

```
const logger = fn => () => {
  console.log("Start");
  fn();
};
```

Sure, Yumna! Here's a short and simple explanation of **Caching** and **Memoization** with examples:

⚡ Caching

Definition:
Caching is storing data temporarily so it can be reused quickly later instead of recomputing or refetching it.
Example:
When you open a website, images are stored (cached) in the browser so they load faster next time.

✅ **Example in JS (fetch cache):**

```
let cache = {};

function fetchData(url) {
  if (cache[url]) return cache[url]; // return cached data
  let result = fetch(url);
  cache[url] = result;
  return result;
}
```

⚡ Memoization

Definition:
Memoization is a specific type of caching used to store results of expensive function calls based on their inputs.
Example:
If you call a function fib(5), the result is saved. If you call fib(5) again, it returns instantly from memory.
✅ **Example in JS:**

```
function memoize(fn) {
  const cache = {};
  return function (x) {
    if (cache[x]) return cache[x];
    const result = fn(x);
    cache[x] = result;
    return result;
  };
}

const square = memoize(x => x * x);
console.log(square(4)); // 16 (computed)
console.log(square(4)); // 16 (cached)
```

Summary:

- **Caching** = storing data (e.g., API or image files).
- **Memoization** = caching function results to avoid repeated work.

Let me know if you want a diagram or code demo!

Great questions, Yumna! Here are **short, clear, and beginner-friendly explanations** for each topic:

---

1. Why is Client-Side Rendering (CSR) different from Server-Side Rendering (SSR)?

- **CSR**: Browser downloads JS, renders content on the client (user's device).
- **SSR**: Server renders HTML and sends it to the browser.
  ✅ *CSR loads faster after initial load; SSR shows content faster on first load.*

---

2. How to define an API?

- An **API** (Application Programming Interface) lets frontend and backend talk using HTTP methods like GET, POST, etc.

✅ *Example in Express.js:*

```
app.get('/api/users', (req, res) => {
  res.json(users);
});
```

3. Most commonly used APIs & why?

- **REST API**: Simple, uses HTTP.
- **GraphQL**: Flexible, request only needed data.
- **Firebase API**: Real-time database.

  ✅ *Used to send/receive data (login, fetch content, save data).*

4. Difference: require vs export

- require = Node.js CommonJS way to import.
- export = ES6 module syntax.

  ✅ *Use import/export in React or modern JS.*

5. What is a shared function?

- A **function used in multiple components or files** (e.g., a utility).

  ✅ *Placed in a utils.js and reused across your app.*

6. What is "lifting state up"?

- Moving shared state to the **nearest common parent** component.

  ✅ *Used when two children need the same data.*

7. Dummy fetch method

```
fetch('https://api.example.com/data')
  .then(res => res.json())
  .then(data => console.log(data));
```

✅ *fetch sends a request, gets a response, converts it to JSON, then uses it.*

8. REST API vs MVC

- **REST API**: Focuses on routes & HTTP methods (GET, POST).
- **MVC**: Design pattern (Model, View, Controller) for app structure.

  ✅ *REST can be used inside MVC.*

9. How does Google remember credentials?

- Through **cookies, localStorage**, and secure tokens like **OAuth**.

  ✅ *It saves login tokens in the browser, and refreshes them securely.*

10. Brief cache description & resolving issues

- Cache stores data to reuse it.

  ✅ *Problem:* Outdated data shown.

  ✅ *Fix:* Use versioning, cache headers, or no-cache:

```
fetch(url, { cache: "no-cache" });
```

11. How to fix CORS issue?

- CORS blocks unknown origins.
  ✅ *Solution (Node.js):*

```
const cors = require('cors');
app.use(cors());
```

---

12. Controlled vs Dumb components

- **Controlled**: Has internal logic & state (e.g., form).
- **Dumb (Presentational)**: Gets data via props, no logic.
  ✅ *Use dumb components for UI only.*

---

13. Does React follow MVC?

- React itself is mostly the **View** in MVC.
  ✅ *You manage Model (data) and Controller (logic) separately.*

---

14. Random number function in JS

```
Math.random(); // Gives 0 to <1
Math.floor(Math.random() * 10); // 0 to 9
```

✅ *Use for games, IDs, tests, etc.*

---

15. Why clearTimeout in debounce?

- **Debounce** waits before calling a function.
  ✅ *clearTimeout cancels old calls to avoid repeated triggers:*

```
let timer;
function debounce(fn, delay) {
  return function () {
    clearTimeout(timer);
    timer = setTimeout(fn, delay);
  };
}
```

---

16. Heap Management in JS

- JS uses **heap** to store objects.
  ✅ *Garbage Collector removes unused objects automatically.*
  ✅ *Avoid memory leaks (unremoved listeners, etc).*

---

17. Why is bind(this) needed?

- JS functions lose their this when passed around.
  ✅ *bind(this) locks the right context.*

```
class App {
  sayHi() {
    console.log(this);
  }
}
```

```
const app = new App();
const fn = app.sayHi.bind(app);
fn();
```

---

18. Conditional rendering: definition + example

- Show content based on a condition.

```
{isLoggedIn ? <Profile /> : <Login />}
```

✅ *Used to show/hide elements like buttons, text, etc.*

---

19. Arrow function and this

- Arrow functions do **not bind their own this**, they inherit from parent scope.

```
const obj = {
  name: "Yumna",
  arrow: () => console.log(this.name), // undefined
  regular() { console.log(this.name); }
};
```

✅ *Use arrow functions in callbacks to avoid losing this.*