📓 `call`, `apply`, and `bind` in JavaScript

> ✅ **Purpose:** Function borrowing — use functions written for one object with a different object, without copying the function.

🧠 `call()` Method

- Invokes a function with an explicit `this` value and arguments provided one by one.
- Commonly used for **function borrowing**.

```
let name1 = {
  firstName: "Sidharth",
  lastName: "Juyal",
  printFullName: function () {
    console.log(this.firstName + " " + this.lastName);
  },
};

let name2 = {
  firstName: "Vex",
  lastName: "Persona",
};

name1.printFullName();              // Sidharth Juyal
name1.printFullName.call(name2);    // Vex Persona
```

🧠 Function Reuse Without Duplicating Logic

- Instead of putting function inside objects, define a generic function and reuse it.

```
const printFullName = function () {
  console.log(this.firstName + " " + this.lastName);
};

let name3 = {
  firstName: "Vegeta",
  lastName: "Saiyan",
};

printFullName.call(name3); // Vegeta Saiyan
```

🔥 Passing Arguments with `call()`

```
const printFullNameAndAge = function (age, state) {
  console.log(this.firstName + " " + this.lastName +
              " is " + age + " years old from " + state);
};
printFullNameAndAge.call(name1, 24, "Uttarakhand");
// Output: Sidharth Juyal is 24 years old from Uttarakhand
```

🧠 `apply()` Method

- Works just like `call()`, but accepts arguments as an **array**.

```
printFullNameAndAge.apply(name2, [24, "Haryana"]);
// Output: Vex Persona is 24 years old from Haryana
```

🧠 `bind()` Method

- Doesn't invoke the function immediately.
- Returns a **copy of the function**, with bound `this` and optional preset arguments.

```
let printVegetaInfo = printFullNameAndAge.bind(name3, 24, "Tokyo");

printVegetaInfo();
// Output: Vegeta Saiyan is 24 years old from Tokyo
```

📝 Summary Table

Method	Invoking Immediately?	Pass Arguments	Returns a Function?
`call`	✅ Yes	Individually	❌ No
`apply`	✅ Yes	As array	❌ No
`bind`	❌ No	Individually	✅ Yes

Polyfill for `.bind()` & Custom `.map()`

🚀 bind() Recap

- `Function.prototype.bind()` creates a **copy of a function** with `this` keyword bound to the object passed.
- It doesn't invoke the function immediately (unlike `call` or `apply`), but **returns a new function** to be called later.

✅ Example:

```
const name1 = {
  firstName: "Sidarth",
  lastname: "Juyal",
};
const printName = function (hometown, state, country) {
  console.log(
    this.firstName +
      " " +
      this.lastname +
      " from " +
      hometown +
      ", " +
      state +
      ", " +
      country
  );
};
const printMyName = printName.bind(name1, "Kotdwar");
printMyName("Uttarakhand", "India");
```

🧠 Polyfill for `.bind()`

❔What is a Polyfill?

A polyfill is a **custom implementation** of a built-in JavaScript method, for learning or for legacy browser support.

💡 Custom `myBind()` implementation:

```
Function.prototype.myBind = function (...args) {
  let obj = this, // Refers to the function on which myBind was called (printName)
      params = args.slice(1); // Extracting params to be pre-set

  return function (...args2) {
    obj.apply(args[0], [...params, ...args2]);
  };
};
```

✅ Using `myBind`:

```
let printMyName2 = printName.myBind(name1, "Kotdwar");
printMyName2("Uttarakhand", "India");
```

> ✔️ `myBind` does:

> - First argument = object to bind (`this`)
> - Rest are **preset parameters**
> - Returned function can be called later with additional args

---

🎯 Pollyfill for `map()` Implementation

- `.map()` is a **Higher Order Function** – it takes a function as input and returns a new array.
- Let's polyfill our own `.myMap()` method on `Array.prototype`

💻 Implementation:

```
Array.prototype.myMap = function (logic) {
  let result = [];
  for (let i = 0; i < this.length; i++) {
    result.push(logic(this[i]));
  }
  return result;
};
```

✅ Usage:

```
let arr = [1, 2, 3];
function double(x) {
  return x * 2;
}
console.log(arr.myMap(double)); // [2, 4, 6]
```

📦 Function Currying in JavaScript

- Function Currying is a **transforming technique** where a function with multiple arguments is converted into a sequence of functions, each taking a single argument.
- **Currying** is a functional programming technique where a function with multiple arguments is **transformed into a sequence of functions**, each taking **a single argument** and returning a new function until all arguments are provided..

🧠 In simpler terms:

Currying breaks down a function that takes `n` **arguments** into `n` **nested functions**, each taking **one argument at a time**.

🧪 Example:

```
function sum(a) {
  return function (b) {
    return function (c) {
      return a + b + c;
    };
  };
}
console.log(sum(1)(2)(3)); // Output: 6
```

🧠 Why Use Currying?

- Reuse functions with preset arguments
- Avoid repeating logic
- Create flexible, partial utilities

✅ Method 1: Using `bind()`

```
let multiply = function (x, y) {
  console.log("Multiply Result: " + x * y);
};
let multiplyByTwo = multiply.bind(this, 2);
multiplyByTwo(5); // Output: 10
let multiplyByThree = multiply.bind(this, 3);
multiplyByThree(5); // Output: 15
```

What's happening?

- `bind(this, 2)` locks the first parameter `x = 2`
- Returns a new function which takes the remaining parameter `y`

✅ Method 2: Using **Closures**

```
let divide = function (x) {
  return function (y) {
    console.log("Divide Result: " + x / y);
  };
};
let divideByFive = divide(25);
divideByFive(5); // Output: 5
```

What's happening?

- `divide()` returns a new function that remembers `x`
- This is the classic form of currying using **lexical scoping**

💡 Real World Use Case

Suppose you're formatting logs:

```
const log = (prefix) => (message) => console.log(`[${prefix}] ${message}`);
const errorLog = log("ERROR");
const infoLog = log("INFO");
errorLog("Something went wrong");
infoLog("User logged in");
```

Debouncing in JavaScript

Debouncing is a programming pattern used to limit the rate at which a function gets executed. It’s commonly used in search inputs, resizing events, or any scenario where a high-frequency event should only trigger a function after a delay.

🔥 Real-World Use Case

Typing in a search bar – we don’t want to call the API on every keystroke. Instead, we wait until the user pauses typing for some milliseconds and then fire the API.

🧠 How Debouncing Works

- Every time the user types (or triggers the event), we reset a timer.
- If another event comes in before the timer ends, the previous one is cancelled.
- Only the last one gets executed after the delay.

💻 Code Breakdown

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Debouncing</title>
</head>
<body>
    <input type="text" onkeyup="betterFunction()" />
    <script src="./Debouncing.js"></script>
</body>
</html>
```

Debouncing.js

```
let counter = 0;
function getData() {
  console.log("Fetching Data...", counter++);
  console.log(this);        // Shows the context passed using .call()
  console.log(arguments);   // Shows the arguments passed to betterFunction
}
const debounceFunction = function (fn, delay) {
  let timer;
  return function () {
    let context = this,
        args = arguments;
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, delay);
  };
};
const betterFunction = debounceFunction(getData, 300);
// Simulating a call with custom 'this' and argument
betterFunction.call({ custom: "object" }, "Sid");
```

🧠 Key Concepts Clarified

✅ 1. `fn.apply(context, args)` vs `getData.apply(...)`

- We must call the function we are debouncing — that's `fn`.
- Using `getData` directly breaks reusability. `fn` is dynamic and allows any function.

✅ 2. `arguments` Inside `getData`

- `getData` must **not** be an arrow function.
- Arrow functions do **not** have their own `this` or `arguments`, so using them throws errors.
- Changing it to a **regular function** gives access to `this` and `arguments` properly.

✅ 3. Context (`this`) Explanation

- When calling `betterFunction.call({ custom: "object" }, "Sid")`, that object becomes `this` inside `getData`.
- So, `this` refers to `{ custom: "object" }` when logged from `getData`.

✅ 4. How the `args` Get Passed

- Any arguments passed to the `debounced function` are captured via `arguments`.
- These are passed on to `fn` via `.apply(context, args)`.

🧪 Output Example

If you call:

```
betterFunction.call({ name: "Vex" }, "Sidharth");
```

You get:

```
Fetching Data... 0
{ name: 'Vex' }
[Arguments] { '0': 'Sidharth' }
```

⚠️ Common Gotchas

- ❌ Don't use arrow functions for handlers that need `this` or `arguments`.
- ❌ Don't hardcode `getData` in your debounce logic — use `fn` parameter.
- ✅ Always test your debounce logic with `.call()` or dynamic arguments to validate its flexibility.

✅ Summary

- Debouncing is a performance optimization strategy.
- Helps delay execution until after a pause.
- Uses closures, timers, and `.apply()` to preserve context and arguments.
- Should be generic and reusable for any function.

🚀 async vs defer in JavaScript

When loading a webpage, two major processes occur:

1. **HTML Parsing** – The browser reads and parses the HTML document top to bottom.
2. **Script Loading** – When the browser encounters a `<script>` tag, it handles:
    - **Fetching**: Getting the script file from the network (I/O bound).
    - **Execution**: Running the JavaScript (CPU bound).

🛑 Without `async` or `defer` (Normal `<script src="...">`)

🧱 **Blocking Behavior**:

- When the parser encounters a script:
    - HTML parsing **stops**.
    - The script is **fetched** from the network.
    - Once fetched, the script is **executed immediately**.

  - Only **after execution**, HTML parsing **resumes**.

```html
<script src="app.js"></script>
```

❌ Cons:

- Blocks rendering.
- Bad for performance.
- Not suitable for large scripts or multiple `<script>` tags.

⚡ With `async`

🔄 Asynchronous Fetch, Immediate Execution

```html
<script src="app.js" async></script>
```

- HTML parsing and script **fetching happen in parallel**.
- As soon as the script is fetched:
  - HTML parsing is **paused**.
  - Script is **executed immediately**.
  - After execution, HTML parsing **resumes**.

⚠️ Important:

- **Execution order is **not guaranteed**.
- Scripts execute **as soon as they're ready**, possibly out of order.
- Best for **independent third-party scripts** (e.g., analytics, ads).

✅ Use `async` when:

- Scripts **don’t depend** on each other or on HTML.
- Execution order **doesn’t matter**.
- Example: Google Analytics.

🕒 With `defer`

⏳ Asynchronous Fetch, Deferred Execution

```html
<script src="app.js" defer></script>
```

- HTML parsing and script fetching happen **in parallel**.
- Script execution is **deferred** until **after the entire HTML is parsed**.

- Scripts are executed **in order of appearance in the HTML**.

✅ Best of Both Worlds:

- Doesn't block HTML parsing.
- Preserves execution order.

✅ Use `defer` when:

- Scripts rely on the DOM being fully available.
- Scripts are interdependent.
- Ideal for most **main application JS files**.

🧠 Comparison Table

Feature	Normal Script	`async`	`defer`
Fetching	Blocking	Asynchronous	Asynchronous
Execution	Immediately (blocking)	Immediately after fetch	After HTML parsing
HTML Parsing	Paused during fetch+exec	Paused during exec	Never paused
Execution Order	Top to bottom	Not guaranteed	Preserved
Use case	Old school scripts	Third-party libs	Main app scripts

---

⚠ Bonus Gotcha: Placement Matters

- `async` and `defer` only work with **external scripts** (i.e., with `src`).
- They have **no effect on inline scripts**.

```html
<!-- ✅ Works -->
<script src="main.js" defer></script>
<!-- ❌ async/ defer ignored -->
<script defer>
  console.log("Ignored");
</script>
```

✅ Summary

- `async`: Load and execute scripts **as soon as possible**, without waiting.
  - Fastest but **order is not guaranteed**.
- `defer`: Load scripts in parallel, **execute after parsing**, **preserves order**.
  - Best for app scripts.
- Regular scripts **block HTML parsing** — avoid them when possible.

🧠 Pro Tips

- Use `defer` for all internal JS files.
- Use `async` only for independent 3rd-party tools.
- Never use both `async` and `defer` on the same script — browser will treat it as `async`.

📌 Event Bubbling vs Capturing in JavaScript

🧠 What Happens When an Event is Triggered?

When a DOM event (e.g., click) occurs, it goes through **three phases**:

1. **Capturing Phase (Trickling Down)** – Event moves from `window → root → target`.
2. **Target Phase** – Event reaches the actual element clicked.
3. **Bubbling Phase (Bubbling Up)** – Event moves back from target to root.

🔁 addEventListener: Third Argument

```
element.addEventListener('click', callback, useCapture);
```

- `useCapture = true` → event is handled in **capturing phase**
- `useCapture = false` (default) → event is handled in **bubbling phase**

🧪 Example Setup

HTML Structure:

```html
<div id="grandparent">
  <div id="parent">
    <div id="child"></div>
  </div>
</div>
```

Each element has a border, so when you click on `#child`, events fire on all 3.

📌 CASES

✅ Case 1: **Pure Bubbling** (default)

```
grandparent.addEventListener('click', ..., false);
parent.addEventListener('click', ..., false);
child.addEventListener('click', ..., false);
```

Click on `child` ➔ Output: `child clicked`, `parent clicked`, `grandparent clicked` → **Reason**: Event bubbles up after reaching target.

✅ Case 2: **Pure Capturing**

```
grandparent.addEventListener('click', ..., true);
parent.addEventListener('click', ..., true);
child.addEventListener('click', ..., true);
```

Click on `child` ➔ Output: `grandparent clicked`, `parent clicked`, `child clicked` → **Reason**: Event captured from top to bottom.

✅ Case 3: **Mixed – Capturing + Bubbling**

```
grandparent.addEventListener('click', ..., true);  // Capturing
parent.addEventListener('click', ..., false);      // Bubbling
child.addEventListener('click', ..., true);        // Capturing
```

Click on `child` ➔ Output: `grandparent clicked`, `child clicked`, `parent clicked` → **Reason**:

- Capturing → grandparent → child
- Bubbling → back up → parent

🚫 stopPropagation()

Stops the event from continuing its journey.

🛑 Case 4: Bubbling + stopPropagation on Target

```
child.addEventListener('click', (e) => {
  console.log('child clicked');
  e.stopPropagation();
}, false);
```

Click on `child` ➔ Output: `child clicked` → **Reason**: Bubbling is prevented from propagating up.

🛑 Case 5: Capturing + stopPropagation on Target

```
child.addEventListener('click', (e) => {
  console.log('child clicked');
  e.stopPropagation();
}, true);
```

Click on `child` ➔ Output: `grandparent clicked`, `parent clicked`, `child clicked` ➡️ **Reason**: Capturing finishes before `stopPropagation` at target.

🛑 Case 6: Capturing + stopPropagation on Grandparent

```
grandparent.addEventListener('click', (e) => {
  console.log('grandparent clicked');
  e.stopPropagation();
}, true);
```

Click on `child` ➔ Output: `grandparent clicked` ➡️ **Reason**: Event is stopped in capturing phase at the top, so it doesn't even reach child or parent.

🔍 Summary Table

Phase	Order (Click on `child`)	Propagation	Control
Capturing	Grandparent → Parent → Child	Top → Bottom	`useCapture: true`
Target	Child		
Bubbling	Child → Parent → Grandparent	Bottom → Top	`useCapture: false`
Stop Flow	`event.stopPropagation()`	Stops either phase at any point	

✅ When to Use What?

Situation	Solution
Prevent parent from reacting	`e.stopPropagation()`
Need control from outer → inner	Use capturing (`true`)
Want outer elements to react later	Use bubbling (`false`)

🔁 Event Delegation in JavaScript

🧠 What is Event Delegation?

Event Delegation is a technique where you **attach a single event listener to a parent element**, instead of adding individual listeners to multiple child elements. The event naturally **bubbles up**, and we use `event.target` to find which child triggered it.

🔎 Why Use Event Delegation?

✅ **Better performance** (especially with many or dynamic child elements) ✅ **Less memory usage** ✅ **Handles dynamically added children automatically** ✅ **Cleaner, more maintainable code**

🧪 Example 1: Click on `<li>` Items

✅ HTML:

```html
<ul id="category">
  <li id="laptops">laptops</li>
  <li id="cameras">cameras</li>
  <li id="shoes">shoes</li>
</ul>
```

✅ JavaScript:

```javascript
document.getElementById("category").addEventListener("click", (e) => {
  if (e.target.tagName === "LI") {
    window.location.href = "/" + e.target.id;
  }
});
```

⚙️ How it works:

1. Listener is added to `<ul id="category">`
2. When any `<li>` inside it is clicked, the event bubbles to the `<ul>`.
3. We check `e.target.tagName === "LI"` to ensure we're acting only on actual `<li>` clicks.

📤 Output:

- Clicking on `laptops` → navigates to `/laptops`

🧪 Example 2: Auto Uppercasing Text Input

✅ HTML:

```html
<form id="category">
  <input type="text" data-uppercase />
  <input type="text" />
  <input type="text" />
</form>
```

✅ JavaScript:

```
document.getElementById("category").addEventListener("keyup", (e) => {
  if (e.target.dataset.uppercase !== undefined) {
    e.target.value = e.target.value.toUpperCase();
  }
});
```

⚙️ How it works:

- Listens for `keyup` on the entire form
- Checks if the input has the attribute `data-uppercase`
- Converts its value to uppercase

✅ Output:

- Only the input with `data-uppercase` will auto-capitalize as the user types.

🧠 `e.target` vs `e.currentTarget`

Property	**Refers To**
`e.target`	The actual element clicked or typed on
`e.currentTarget`	The element the event listener is attached to (`category` in our case)

🛠️ When to Use Event Delegation

Use Case	**Reason**
Dynamic content (e.g., dropdowns, search suggestions)	Children created later still handled
Forms with many fields	Centralized validation/logic
Lists or menus	Fewer listeners, better performance

🔥 Advantages

- Performance gain in **large lists or dynamic UIs**
- Easy to handle **dynamic DOM changes**
- **Reduces redundant code**

🚨 Caution

- You must **filter** `e.target` carefully; events bubble from *deepest nested elements*
- Use `tagName`, `classList`, `matches()`, or `dataset` to check targets
- Some events **don't bubble** (e.g. `focus`, `blur`)

🧪 Interview Tip

> **Q: How would you handle a click on hundreds of dynamic `<li>` items efficiently?** ✅ Use **Event Delegation**: Attach a single listener on the parent (like `<ul>`) and use `e.target` to determine which `<li>` was clicked.

Local Storage & Session Storage

```
// local storage
localStorage.setItem("hello", "world");
localStorage.setItem("hello1", "world1");
localStorage.getItem("hello");
localStorage.removeItem("hello1");
localStorage.clear();
// if you ant to store an object, do JSON.stringify(obj)
// when retrieving, do JSON.parse(obj)

// session storage
sessionStorage.setItem("hello3", "world");
sessionStorage.setItem("hello4", "world1");
sessionStorage.getItem("hello3");
sessionStorage.removeItem("hello4");
sessionStorage.clear();
```

🧠 Prototype vs Prototypal Inheritance in JavaScript

🔁 What is a Prototype?

- In JavaScript, **every object has a hidden internal property `[[Prototype]]`**, which can be accessed via `__proto__` (unofficial) or `Object.getPrototypeOf(obj)` (official).
- A **prototype** is just another object that acts as a **fallback source** for properties and methods when they're not found on the object itself.

🧬 Prototypal Inheritance

- **If a property or method is not found on an object**, JavaScript looks up the `__proto__` chain until it finds it or reaches `null`.

```
let object = {
    name: "Sid",
    city: "Kotdwar",
    getIntro: function() {
        console.log(this.name + " from " + this.city);
    }
```

```js
};
let object2 = {
    name: "Vex"
};
// ❌ Avoid in practice, but used here to demonstrate inheritance
object2.__proto__ = object;
object2.getIntro(); // "Vex from Kotdwar"
```

🔍 Explanation:

- `object2` has no `city` or `getIntro`, so it looks into its prototype (`object`) for those.
- `this` inside `getIntro` still refers to `object2`, because it's the calling object.

⚠️ `__proto__` vs `prototype`

Concept	Belongs To	Description
`__proto__`	Any object	The internal reference to the object's prototype
`prototype`	Constructor Func	The object from which new instances inherit properties

TL;DR:

- `__proto__` is what **an object points to**
- `prototype` is what **a function uses to build that `__proto__`**

🧰 Function Prototype & Custom Methods

You can extend native prototypes like `Function.prototype`, `Array.prototype`, etc.

```js
Function.prototype.myBind = function() {
    console.log("My custom bind");
};
function hello() {}
hello.myBind(); // My custom bind
```

🔄 Prototype Chain Lookup

```js
object2.getIntro();
// Looks like:
- object2.getIntro → ❌
- object2.__proto__.getIntro → ✅
```

If `object2.__proto__ = object`, and `object.__proto__ = Object.prototype`, it continues looking until it hits `null`.

✅ Real-World Use (Cleaner Version)

Instead of `__proto__`, use `Object.create()`:

```
let parent = {
    greet() {
        console.log("Hello from parent");
    }
};
let child = Object.create(parent);
child.name = "Vex";
child.greet(); // Hello from parent
```

❌ Why You Should Avoid `__proto__`

- It's considered legacy and non-performant.
- Can lead to unexpected bugs if misused.
- Use `Object.create`, `Object.setPrototypeOf`, or class-based inheritance instead.

🧠 Interview Snippets

> “Prototype in JavaScript is a mechanism by which objects can inherit properties from other objects.” “Prototypal inheritance lets objects inherit directly from other objects, forming a chain via their `__proto__` reference.” “Every function has a `prototype` property, which is used when creating objects using the `new` keyword. This becomes the new object's `__proto__`.”

Debouncing in JavaScript

Debouncing is a programming pattern used to limit the rate at which a function gets executed. It’s commonly used in search inputs, resizing events, or any scenario where a high-frequency event should only trigger a function after a delay.

🔥 Real-World Use Case

Typing in a search bar – we don’t want to call the API on every keystroke. Instead, we wait until the user pauses typing for some milliseconds and then fire the API.

🧠 How Debouncing Works

- Every time the user types (or triggers the event), we reset a timer.
- If another event comes in before the timer ends, the previous one is cancelled.
- Only the last one gets executed after the delay.

💻 Code Breakdown

index.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Debouncing</title>
</head>
<body>
    <input type="text" onkeyup="betterFunction()" />
    <script src="./Debouncing.js"></script>
</body>
</html>
```

Debouncing.js

```javascript
let counter = 0;
function getData() {
  console.log("Fetching Data...", counter++);
  console.log(this);        // Shows the context passed using .call()
  console.log(arguments);   // Shows the arguments passed to betterFunction
}
const debounceFunction = function (fn, delay) {
  let timer;
  return function () {
    let context = this,
        args = arguments;
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, delay);
  };
};
const betterFunction = debounceFunction(getData, 300);
// Simulating a call with custom 'this' and argument
betterFunction.call({ custom: "object" }, "Sid");
```

🧠 Key Concepts Clarified

✅ 1. `fn.apply(context, args)` vs `getData.apply(...)`

- We must call the function we are debouncing — that's `fn`.
- Using `getData` directly breaks reusability. `fn` is dynamic and allows any function.

✅ 2. `arguments` Inside `getData`

- `getData` must **not** be an arrow function.
- Arrow functions do **not** have their own `this` or `arguments`, so using them throws errors.
- Changing it to a **regular function** gives access to `this` and `arguments` properly.

✅ 3. Context (`this`) Explanation

- When calling `betterFunction.call({ custom: "object" }, "Sid")`, that object becomes `this` inside `getData`.
- So, `this` refers to `{ custom: "object" }` when logged from `getData`.

✅ 4. How the `args` Get Passed

- Any arguments passed to the `debounced function` are captured via `arguments`.
- These are passed on to `fn` via `.apply(context, args)`.

🧪 Output Example

If you call:

```
betterFunction.call({ name: "Vex" }, "Sidharth");
```

You get:

```
Fetching Data... 0
{ name: 'Vex' }
[Arguments] { '0': 'Sidharth' }
```

⚠ Common Gotchas

- ✕ Don't use arrow functions for handlers that need `this` or `arguments`.
- ✕ Don't hardcode `getData` in your debounce logic — use `fn` parameter.
- ✅ Always test your debounce logic with `.call()` or dynamic arguments to validate its flexibility.

✅ Summary

- Debouncing is a performance optimization strategy.
- Helps delay execution until after a pause.

- Uses closures, timers, and `.apply()` to preserve context and arguments.
- Should be generic and reusable for any function.

🔥 Throttling in JavaScript (Flipkart UI Interview Concept)

✅ Definition

Throttling is a technique used to **limit the number of times a function can execute over time**. It ensures that a function is invoked at most **once every specified interval**, regardless of how many times the event is triggered.

⚡ Real-World Use Cases

- `resize` events (e.g., adjusting layout/UI responsively)
- `scroll` events (e.g., infinite scroll logic)
- `mousemove`, `keyup`, or `input` events
- Preventing **button spamming**
- Rate-limiting **API calls** (e.g., search suggestions)

🚫 Problem (Without Throttling)

If you directly bind an **expensive function** to a high-frequency event (e.g., `resize`), it gets called **hundreds of times per second**, resulting in:

- Laggy UI
- Memory leaks
- Performance bottlenecks

✅ Solution (With Throttling)

Throttle limits the function call to once every `n` milliseconds, regardless of how frequently the event is triggered.

🧠 Code Example

```
function expensive() {
  console.log("Expensive function called...");
}

function throttle(func, limit) {
  let flag = true;

  return function () {
    let context = this;
    let args = arguments;
    if (flag) {
      func.apply(context, args);
      flag = false;
      setTimeout(() => {
        flag = true;
      }, limit);
    }
  };
}
// Create a throttled version of expensive()
const betterExpensive = throttle(expensive, 1000);
// Bind it to the resize event
window.addEventListener("resize", betterExpensive);
```

🧪 Explanation of How It Works

- The `flag` is initially `true`.
- On the first call, `flag` allows the function to run and is then set to `false`.
- A `setTimeout` resets the `flag` to `true` after `limit` milliseconds.
- Until `flag` becomes true again, no further execution is allowed.

This creates a **cooldown window** where the function is ignored even if events keep firing.

❗ Common Mistakes

```
// ❌ Incorrect
window.addEventListener("resize", betterExpensive());
```

This executes `betterExpensive()` **immediately** during setup, not on `resize`. ✅ Correct version:

```
window.addEventListener("resize", betterExpensive);
```

🧠 Concept Summary Table

Feature	Throttling	Debouncing
Function executes	At most once every X ms	Only after X ms of inactivity
Use case	Resize, scroll, clicks	Input boxes, search bars
Ideal for	Rate-limiting calls	Avoiding unnecessary repeat calls
Triggers execution	Periodically	After idle period

🔚 Final Thought

Throttling = **controlled frequency** Debouncing = **controlled silence** Use throttling when you want **consistent execution at intervals**, and debouncing when you want **execution only once after inactivity**.