🔑 `this` Keyword in JavaScript – Full Guide

The `this` keyword in JavaScript refers to an **object**, but the object it refers to depends on the **context of execution**.

> The `this` keyword is a special identifier that refers to the execution context of a function. Its value is not static—it is dynamically set depending on how and where a function is called.

📌 Table of Contents

Sr. No	Context	Description
1️⃣	Global Space	Refers to global object (`window` in browsers, `global` in Node.js)
2️⃣	Function (Non-Strict vs Strict)	`this` varies based on mode
3️⃣	Method Inside Object	Refers to the object itself
4️⃣	Arrow Function	Inherits `this` from lexical scope
5️⃣	Nested Arrow Function	Same as arrow — inherits from closest non-arrow function
6️⃣	call, apply, bind	Manually set `this`
7️⃣	DOM Event Handler	Refers to the HTML element
8️⃣	Class Constructor	Refers to the instance being created

🌍 `this` in Global Scope

- In **browsers**, `this` refers to the `window` object.
- In **Node.js**, `this` in the global module scope refers to `{}` (an empty object) because Node wraps each module in a closure.

```
console.log(this); // Browser: window, Node.js: {}
```

☑️ In a browser, this refers to `window` object. ☑️ In Node.js, it refers to an empty object `{}` in modules.

🔁 `this` Inside a Function

🚫 Non-strict Mode:

> If no context is explicitly provided, `this` defaults to the global object:

```
function show() {
  console.log(this);
```

```
}
show(); // 👉 window (global object)
```

☑ Strict Mode:

```
"use strict";
function show() {
  console.log(this);
}
show(); // 👉 undefined ❗
```

> In `'use strict'`, JavaScript does **not** fall back to the global object; `this` becomes `undefined`. 💡 `this substitution`: In **non-strict mode**, if `this` is `undefined` or `null`, it's substituted with the global object. 🧠 **According to MDN**: In strict mode, `this` is preserved as whatever value it was when entering the execution context.

🧠 `this` Depends on How a Function is Called

```
function sayHi() {
  console.log(this);
}
sayHi();       // undefined in strict mode, window in non-strict
window.sayHi(); // window attached with window
```

🧱 `this` Inside an Object's Method

> When a function is called as a **method of an object**, `this` refers to that **object itself**.

```
const obj = {
  a: 42,
  show: function () {
    console.log(this);    // 👉 obj
    console.log(this.a);  // 👉 42
  }
};

obj.show();
```

> 🧠 In methods, `this` refers to the object *before the dot*. 📚 **MDN Analogy**: The value of `this` is determined by the object **before the dot** used to call the function.

🔄 Difference: Function vs Method

Type	Description
Function	Independent, not tied to any object
Method	Function inside an object, called via object

🤝 `call`, `apply`, and `bind` Methods

Used to **borrow or override `this` context**:

```
const user1 = {
  firstName: "Akshay",
  lastName: "Saini",
  fullName: function () {
    console.log(this.firstName + " " + this.lastName);
  },
};

const user2 = {
  firstName: "Sachin",
  lastName: "Tendulkar"
};

// 📞 call
user1.fullName.call(user2);  // Sachin Tendulkar

// 📝 apply (array of args)
function greet(hometown) {
  console.log(this.firstName + " from " + hometown);
}
greet.apply(user2, ["Mumbai"]);

// 📌 bind (returns a new function)
const greetUser2 = greet.bind(user2, "Pune");
greetUser2();  // Sachin from Pune
```

🤝 6️⃣ Using `call()`, `apply()`, `bind()`

These methods allow you to **explicitly set the `this` context** of a function.

📞 `call()`: calls a function with a given `this` and arguments:

```
user1.fullName.call(user2);
```

📝 `apply()`: same as `call` but takes arguments as an array:

```
greet.apply(user2, ["Mumbai"]);
```

📌 `bind()`: returns a **new function** with `this` permanently bound:

```
const greetUser2 = greet.bind(user2, "Pune");
greetUser2(); // Sachin from Pune
```

> 📚 **MDN Note**: `bind()` is ideal for event handlers or callback scenarios where you want to **lock in** a particular `this`.

♉ `this` in Arrow Functions

Arrow functions do **not have their own** `this` — they **inherit from their lexical (outer) scope**.

```
const obj = {
  value: 100,
  arrowFunc: () => {
    console.log(this.value); // ❌ undefined (this = window/global)
  },
};
obj.arrowFunc();
```

> 📚 **MDN Analogy**: The value of `this` is determined by the object **before the dot** used to call the function. When a function is called as a **method of an object**, `this` refers to that **object itself**.

🔄 `this` in Nested Arrow Functions

> Arrow functions **don't have their own** `this`. Instead, they **inherit** `this` from their **lexical environment**—the scope in which they were defined. Arrow functions nested within regular functions still inherit `this` from their enclosing context.

```
const obj = {
  value: 100,
  method: function () {
    const arrow = () => {
      console.log(this.value); // ☑ 100
    };
    arrow();
  },
};
obj.method();
```

☑ In this case, `this` refers to `obj` because `arrow` inherits `this` from the regular function.

🖱️ `this` in DOM Elements

> In DOM event handlers, `this` refers to the **element** that received the event. 📄 **MDN Insight**: Event handlers receive the DOM element they're bound to as `this`. Arrow functions should be avoided if `this` is needed.

```
<button onclick="console.log(this)">Click Me</button>
```

☑ Here, `this` refers to the **DOM element** that received the event.

🏗️ `this` in Class Constructor

> In a class, the constructor and methods have their own context—`this` refers to the **instance** created by `new`. 📄 **MDN Clarification**: `this` inside class constructors refers to the instance of the class being constructed.

```
class Person {
  constructor(name) {
    this.name = name;
  }

  sayHello() {
    console.log(`Hi, I'm ${this.name}`);
  }
}

const p = new Person("Darshan");
p.sayHello(); // Hi, I'm Darshan
```

☑ In class constructors, `this` refers to the **newly created object** (i.e., the instance).

🧠 Summary Table

Context	`this` Value
Global (non-strict)	Global object (`window`)
Global (strict)	`undefined`
Function (non-strict)	Global object
Function (strict)	`undefined`
Object Method	The object itself

Context	`this` Value
Arrow Function	Inherits from enclosing scope
call/apply/bind	Explicitly defined by the caller
Event Handler	DOM element triggering the event
Class Constructor	Newly created object (instance)

Precedence of `this`

Precedence	Context	Takes Effect
1	`bind()`	Highest
2	`call()`, `apply()`	Moderate
3	Object Method	Normal
4	Global/Function	Lowest

Final Notes

- `this` is **not a variable** — it's a **keyword**.
- Arrow functions are great for **lexical scoping** but **not for dynamic binding**.
- `this` behaves differently in **Node.js vs Browser**, and in **strict vs non-strict mode**.
- Understanding how `this` works is crucial for mastering **object-oriented JS, DOM manipulation, and async programming**.

JavaScript Runtime Environment

JavaScript **doesn't run on its own** — it runs **inside a JavaScript engine** provided by a **runtime environment** such as:

- **Browser (e.g., Chrome's V8 engine)**
- **Node.js (server-side JS runtime using V8)**

In Browsers:

- The **global object** is `window`.
- It acts as the **global namespace** for all variables, functions, and objects.

```
console.log(window); // Outputs the global window object in browser
```

☑ Example:

```
var a = 10;
console.log(window.a); // 10
```

💻 In Node.js:

- The **global object** is `global`, **not** `window`.
- Node uses its own module system, so variables are **not automatically added** to the global scope like in browsers.

```
console.log(global); // Outputs global object in Node.js
```

☑ Example:

```
var a = 10;
console.log(global.a); // undefined (NOT added to global scope)
```

🔍 Summary Comparison Table:

Feature	Browser	Node.js
Runtime	V8 inside Chrome/Firefox etc.	V8 inside Node.js
Global Object	`window`	`global`
Global `this`	`this === window` is `true`	`this === global` is `false` at module level
Global Scope	`var` adds to `window`	`var` stays inside module scope
Modules	Script files (global)	CommonJS (`require`) or ESM (`import`)

📌 Key Takeaway:

> **JavaScript behaves differently depending on the environment.** Browser exposes a `window` object as global, while **Node.js uses** `global` and encapsulates modules for better isolation and security.

🧠 Additional Concepts from MDN

1. `this` is not lexically scoped in regular functions

> In traditional functions, `this` is **not determined by where the function is defined**, but rather **how it's called**.

2. `this` in setTimeout or setInterval

```
setTimeout(function () {
  console.log(this); // 👉 window (in browsers)
}, 1000);
```

> Use `bind(this)` or arrow functions if you want to preserve context.

🌐 Environment-Based Differences

🌐 In Browsers:

```
var a = 10;
console.log(window.a);     // 10
console.log(this === window); // true
```

💻 In Node.js:

```
var a = 10;
console.log(global.a);     // undefined
console.log(this);         // {}
console.log(this === global); // false
```

> **MDN-like Summary**: In Node.js, the top-level `this` is not global, but **module-scoped**.

🧩 Final Key Takeaways

- `this` is **dynamic** in regular functions, **static** in arrow functions.
- Use `bind` to **lock** `this`, or arrow functions to **inherit** it.
- DOM elements get `this` when used with regular function event handlers.
- Node.js and browsers behave differently due to their respective **execution contexts**.
- **Understanding `this` is foundational** for writing reusable and bug-free JavaScript functions.

Regular Function `This` Vs Arrow Function `This`

In JavaScript, `this` is a special keyword that refers to the **object that is executing the current function**.

- In **regular functions**, `this` is determined by **how** the function is called.
- In **arrow functions**, `this` is **lexically bound** — it means they inherit `this` from the **surrounding (enclosing) code** where the function was defined.

💡 Arrow Function and `this` Behavior

◇ Regular Function vs Arrow Function – `this` Example:

```
const person = {
  name: 'Darshan',
  regularFunction: function () {
    console.log("Regular:", this.name);
  },
  arrowFunction: () => {
    console.log("Arrow:", this.name);
  }
};

person.regularFunction(); // Regular: Darshan
person.arrowFunction();   // Arrow: undefined (or global object's name, in some
environments)
```

🔍 Explanation:

- `regularFunction`: Here, `this` refers to the `person` object because it was called with `person.regularFunction()`.
- `arrowFunction`: Arrow functions **do not** bind their own `this`. Instead, they use the `this` from the **lexical context** — the place where the arrow function was defined. In this case, it was defined inside the outer script scope, not inside the `person` object, so `this` doesn't point to `person`.

🤯 What is **Lexical Context**?

> The **lexical context** is the environment where the function was **written** in code — not where it was called.

Arrow functions don't define their own `this`; they "close over" or **inherit `this` from their enclosing lexical scope**.

Example with setTimeout:

```
function Timer() {
  this.seconds = 0;

  setInterval(function () {
    this.seconds++;
    console.log("Regular:", this.seconds);
  }, 1000);
}

new Timer(); // NaN or undefined because `this` is NOT the Timer instance
```

🔁 Fixed using an **arrow function**:

```
function Timer() {
  this.seconds = 0;

  setInterval(() => {
    this.seconds++;
    console.log("Arrow:", this.seconds);
  }, 1000);
}

new Timer(); // Works correctly: logs 1, 2, 3...
```

Why it works:

- The arrow function inside `setInterval` inherits `this` from the `Timer` function's context — which **does** refer to the new `Timer` instance.

☑ Summary

Feature	Regular Function	Arrow Function
Own `this` binding?	☑ Yes	❌ No (inherits from lexical context)
`this` depends on	**How** function is called	**Where** function is defined
Used in	Methods, constructors, etc.	Callbacks, short functions, inside classes

---