

Day 3 this keyword

This is my day 3 plan
Day 3 – this Keyword

Learn: Global context, function context, object methods, arrow functions, event handlers.

Implement: Example where this changes with call, bind, apply.

Problems: Predict outputs for tricky this usage.

Let me go through every things in details

Perfect 🔥 Let's go through **Day 3 – this Keyword** step by step.

1. What is this ?

- `this` is a special keyword in JavaScript that refers to the **execution context** (the object that is currently calling the function).
- Its value depends **not on where it is defined**, but **how it is called**.

2. Global Context

- In the **browser global scope**, `this` refers to the `window` **object**.
- In **Node.js**, `this` refers to an empty object `{}` in modules (not `global`).

js

```
console.log(this); // In browser → window, in Node.js → {}
```

- In **strict mode**, `this` in global scope is `undefined`.

js

```
"use strict";  
console.log(this); // undefined
```

3. Inside Functions

- **Normal functions:** value of `this` depends on **how the function is called**.

```
js

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

show(); // In browser: window, in strict mode: undefined
```

4. Object Methods

- If a function is called as a method of an object → `this` = that object.

```
js

const obj = {
  name: "Rishav",
  sayName: function() {
    console.log(this.name);
  }
};

obj.sayName(); // "Rishav"
```

- But if you **extract** the method:

```
js

const f = obj.sayName;
f(); // undefined (strict) OR window.name (non-strict)
```

5. Arrow Functions

- Arrow functions **do not have their own** `this`.
- They capture `this` from the surrounding **lexical scope**.

```
js

const obj = {
  name: "Rishav",
  sayName: () => {
```

```
    console.log(this.name);
  }
};
obj.sayName(); // undefined (since `this` is from global scope)
```

To fix it, use normal function:

```
js

const obj2 = {
  name: "Rishav",
  sayName: function() {
    const inner = () => console.log(this.name);
    inner();
  }
};
obj2.sayName(); // "Rishav"
```

6. Event Handlers

- In browser event handlers, `this` refers to the **DOM element** that received the event.

```
html

<button id="btn">Click Me</button>
<script>
document.getElementById("btn").addEventListener("click", function() {
  console.log(this); // <button>
});
</script>
```

- With **arrow functions**, `this` will not point to the element:

```
js

btn.addEventListener("click", () => {
  console.log(this); // window, not button
});
```

7. call, apply, bind

- These methods explicitly set `this`.

```
js

function greet(city) {
  console.log(`Hello ${this.name} from ${city}`);
}

const person = { name: "Rishav" };

greet.call(person, "Delhi"); // Hello Rishav from Delhi
greet.apply(person, ["Mumbai"]); // Hello Rishav from Mumbai

const boundGreet = greet.bind(person, "Bangalore");
boundGreet(); // Hello Rishav from Bangalore
```

8. Tricky `this` Problems

- ◆ Example 1

```
js

var length = 10;
function fn() {
  console.log(this.length);
}

const obj = {
  length: 5,
  method(fn) {
    fn(); // ??
    arguments[0](); // ??
  }
};

obj.method(fn, 1);
```

- `fn();` → global context → logs `10`
- `arguments[0]();` → `arguments` object has length = 2 → logs `2`

♦ Example 2

```
js

const obj = {
  a: 10,
  fn: function() {
    console.log(this.a);
  }
};

setTimeout(obj.fn, 1000); // ??
```

- Loses reference to `obj`, so → `undefined` (strict) or `window.a`

To fix:

```
js

setTimeout(obj.fn.bind(obj), 1000); // 10
```

✓ Practice Checklist for You:

1. Play with `this` in global, object, arrow, and event contexts.
2. Try rewriting normal functions as arrow functions and see how behavior changes.
3. Use `call`, `apply`, `bind` on a few objects.
4. Solve at least 5 tricky `this` puzzles.

Do you want me to **prepare 5–6 tricky `this` puzzles** for you to solve (like the above) so you can test yourself after learning?

what is strict mode and non strict mode

Great question 👍 Let's clear the difference between **strict mode** and **non-strict mode** in JavaScript, since it directly affects things like `this`, variable declarations, and function behavior.

1. What is Strict Mode?

- `"use strict";` is a directive introduced in **ES5**.
- It tells the JavaScript engine to run the code in a **more strict way** (with extra rules and errors).
- Helps catch **common mistakes** and **unsafe actions**.

You enable it at the top of a script or inside a function:

```
js

"use strict";
x = 10; // ❌ Error: x is not defined
```

Without strict mode, this would create a global variable.

2. Main Differences

♦ (a) Value of `this`

- **Non-strict mode:**

If a function is called standalone, `this` defaults to the **global object** (`window` in browsers, `global` in Node.js).

- **Strict mode:**

If a function is called standalone, `this` becomes **undefined**.

```
js

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

// Non-strict
show(); // window (in browser)

// Strict
"use strict";
function showStrict() {
  console.log(this);
}
```

```
}  
showStrict(); // undefined
```

◆ (b) Silent Errors → Real Errors

- Non-strict mode: some mistakes are ignored.
- Strict mode: throws errors.

```
js  
  
// Non-strict  
num = 5; // creates global variable (bad practice)  
  
// Strict  
"use strict";  
num = 5; // ❌ ReferenceError: num is not defined
```


◆ (c) Deleting Variables/Objects

```
js  
  
// Non-strict  
var x = 10;  
delete x; // false (fails silently)  
  
// Strict  
"use strict";  
var y = 20;  
delete y; // ❌ SyntaxError
```

◆ (d) Duplicates Not Allowed

```
js  
  
// Non-strict  
function sum(a, a) {  
  return a + a;  
}
```

```
console.log(sum(2, 3)); // 6


// Strict
"use strict";
function sumStrict(a, a) {} //  SyntaxError
```

◆ (e) `with` Statement

- `with` allows shorthand access to object properties.
- Not allowed in strict mode (because it makes scope confusing).

```
js

// Non-strict
var obj = { a: 1, b: 2 };
with (obj) {
  console.log(a, b); // 1 2
}

// Strict
"use strict";
with (obj) {} //  SyntaxError
```

3. Summary Table

Feature	Non-Strict Mode	Strict Mode
<code>this</code> in standalone function	Global object	<code>undefined</code>
Assigning undeclared variables	Creates global	Error
Deleting variables	Fails silently	Error
Duplicate function parameters	Allowed	Error

Feature	Non-Strict Mode	Strict Mode
<code>with</code> statement	Allowed	Error

✅ Why use strict mode?

- Prevents accidental globals.
- Makes code safer & easier to debug.
- Required for some modern JavaScript features (like ES6 classes and modules are strict by default).

Do you want me to show you **how strict mode changes behavior of `this` with a side-by-side example** (normal function, arrow function, and object method)?

in my case
"use strict";
console.log(this); // undefined

this is also logging global window object

Ah nice catch 🚀 — let's debug this carefully.

When you write:

```
js  
  
"use strict";  
console.log(this);
```

👉 In **strict mode**, `this` at the **top level** depends on whether your code is running as:

1. **Script** (`<script> ... </script>`)

- In browsers, top-level `this` is still the **global object** (`window`), even with `"use strict"`.

```
html
```

```
<script>
  "use strict";
  console.log(this); //  window
</script>
```

This is because strict mode only changes behavior **inside functions**, not the script's top-level scope in browsers.

2. Modules (`<script type="module">` or `.mjs`)

- In ES6 modules, top-level `this` is always `undefined`, regardless of strict mode.

```
html

<script type="module">
  console.log(this); //  undefined
</script>
```

3. Functions

```
js

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

Without strict mode → would log `window`.

Why you're seeing `window`

- If you're writing `"use strict";` at the **top level in a** `<script>` **file**, browsers still give you `window`.
 - If you put the same inside a **function** or use **modules**, you'll see `undefined`.
-

⚡ Quick experiment for you:

```
js

"use strict";
console.log("Top level:", this); // window

function test() {
  "use strict";
  console.log("Inside function:", this);
}
test(); // undefined
```

👉 So the rule is:

- **Top-level in script** → still `window` (even in strict mode).
- **Top-level in module** → `undefined`.
- **Inside functions** → `undefined` in strict mode, global object in non-strict.

Do you want me to make a **side-by-side code snippet** showing `this` in all 4 contexts (script top-level, module top-level, function strict, function non-strict) so you can clearly compare?

why first function logs undefined and second one logs window object

```
var functionExpression = function() {
  'use strict';
  console.log("this inside the function expression",
    this); // undefined
}
functionExpression();
```

```
var arrowFunction = () => {
  'use strict';
  console.log("this inside the arrow function", this);
}
```

Excellent 🙌 this is the **classic difference between normal functions and arrow functions** combined with strict mode. Let's break it step by step.

◆ Case 1: Function Expression

js

```
var functionExpression = function() {  
  'use strict';  
  console.log("this inside the function expression", this);  
}  
functionExpression();
```

- This is a **normal function**.
- In **strict mode**, when you call a normal function without any object (i.e., `functionExpression()`), `this` is set to `undefined` instead of falling back to the global object.

✓ That's why you see:

javascript

```
this inside the function expression undefined
```

◆ Case 2: Arrow Function

js

```
var arrowFunction = () => {  
  'use strict';  
  console.log("this inside the arrow function", this);  
}  
arrowFunction();
```

- **Arrow functions don't have their own `this`**.
- Instead, they **capture `this` from the surrounding lexical scope** at the time they are defined.
- In your case, the arrow function is defined in the **global (script) scope**.

Now, what is `this` in the **global scope**?

- In a normal `<script>` file → `this` is the **global object** (`window`), even if you write `"use strict"` at the top.
- That's what the arrow function inherits.

✓ That's why you see:

```
javascript
```

```
this inside the arrow function Window {...}
```

Side-by-side mental model

Function Type	Where <code>this</code> comes from	What you saw
Normal function	Set dynamically when called → <code>undefined</code> in strict mode	<code>undefined</code>
Arrow function	Inherits <code>this</code> from where it was defined (global scope = <code>window</code>)	<code>window</code>

⚡ Quick experiment for you:

```
js
```

```
"use strict";
```

```
var obj = {
  normal: function() {
    console.log("normal:", this);
  },
  arrow: () => {
    console.log("arrow:", this);
  }
};
```

```
obj.normal(); // normal: obj
```

```
obj.arrow(); // arrow: window (because arrow inherited from global)
```

👉 So the difference comes from:

- **Normal function** → `this` is determined by **how it's called**.
- **Arrow function** → `this` is determined by **where it's defined**.

Do you want me to prepare a **visual diagram** showing how `this` flows differently in normal vs arrow functions?

Explain why this logs nothing

```
const obj = {  
  name: "Rishav",  
  sayName: () => {  
    console.log(this.name);  
  }  
};  
obj.sayName();
```

Great one 🔥 This is one of the most confusing `this` cases in JavaScript. Let's dissect why your code:

```
js  
  
const obj = {  
  name: "Rishav",  
  sayName: () => {  
    console.log(this.name);  
  }  
};  
obj.sayName();
```

logs nothing / undefined.

1. Arrow functions don't have their own `this`

- A normal function: `this` depends on **how it's called**.

- An **arrow function**: `this` is **lexically bound** → it captures `this` from the surrounding scope (where the function is defined).
-

2. Where is `sayName` defined?

- `sayName` is defined **inside the object literal**, but that does **not** mean its `this` = `obj`.
- Instead, the arrow function looks **one scope up** → the scope where `obj` was created.

If you're running this in:

- A **browser script** → top-level scope `this` = `window`.
- If `"use strict"` or **modules** → top-level scope `this` = `undefined`.

So, inside the arrow function:

```
js  
  
this.name // either undefined.name (error) OR window.name (usually empty string)
```

That's why nothing shows up.

3. Fixing it

If you actually want `this` to refer to `obj`, use a **normal function**:

```
js  
  
const obj = {  
  name: "Rishav",  
  sayName: function() {  
    console.log(this.name);  
  }  
};  
  
obj.sayName(); // "Rishav"
```

4. When arrow is useful

Arrows are useful when you want to preserve `this` from an outer function:

js

```
const obj = {  
  name: "Rishav",  
  sayName: function() {  
    const inner = () => {  
      console.log(this.name);  
    };  
    inner();  
  }  
};  
obj.sayName(); // "Rishav"
```

Here, the arrow inherits `this` from the normal method (`obj`).

✓ Rule of thumb:

- Use **normal functions** for object methods (when you need `this` to be the object).
 - Use **arrow functions** for inner callbacks (when you want `this` from the outer function).
-

Do you want me to give you a **set of practice puzzles** where you predict what `this` will log in normal vs arrow functions inside objects?

Lets give me some question to practice that asked in interview

Perfect 🔥 Let's put your **Day 3: `this` keyword** learning into practice.

I'll give you some **interview-style tricky questions** (in increasing difficulty).

Try to **predict the output** before running them.

♦ Q1: Basic object method

js


```
const person = {  
  name: "Rishav",  
  greet: function() {  
    console.log(this.name);  
  }  
};  
person.greet();
```

👉 What will this log?

◆ Q2: Extracted method

```
js  
  
const person = {  
  name: "Rishav",  
  greet: function() {  
    console.log(this.name);  
  }  
};  
  
const greetFn = person.greet;  
greetFn();
```

👉 What will this log?

◆ Q3: Arrow inside object

```
js  
  
const obj = {  
  name: "Rishav",  
  sayName: () => {  
    console.log(this.name);  
  }  
};  
obj.sayName();
```

👉 Why does this not print "Rishav" ?

◆ Q4: Arrow inside normal function

js

```
const obj = {  
  name: "Rishav",  
  sayName: function() {  
    const inner = () => {  
      console.log(this.name);  
    };  
    inner();  
  }  
};  
obj.sayName();
```

👉 What will this log?

◆ Q5: setTimeout with method

js

```
const obj = {  
  value: 42,  
  log: function() {  
    setTimeout(function() {  
      console.log(this.value);  
    }, 1000);  
  }  
};  
obj.log();
```

👉 What will be printed after 1 second?

◆ Q6: Fix with arrow

js

```
const obj = {  
  value: 42,
```

```
log: function() {  
  setTimeout(() => {  
    console.log(this.value);  
  }, 1000);  
}  
};  
obj.log();
```

👉 Now what will be printed?

♦ Q7: call, apply, bind

```
js  
  
function greet(city) {  
  console.log(`Hello ${this.name} from ${city}`);  
}  
  
const user = { name: "Rishav" };  
  
greet.call(user, "Delhi");  
greet.apply(user, ["Mumbai"]);  
  
const boundFn = greet.bind(user, "Bangalore");  
boundFn();
```

👉 What will be the three outputs?

♦ Q8: arguments trap

```
js  
  
var length = 10;  
function fn() {  
  console.log(this.length);  
}  
  
const obj = {  
  length: 5,  
  method(fn) {
```

```
fn();
arguments[0]();
}
};

obj.method(fn, 1);
```

👉 What will be logged? (Hint: one will be `10` , the other `2`)

These are **real interview-style traps** where many candidates get stuck.

Would you like me to **give solutions immediately** or let you **attempt and then check against my answers?**

sure

Alright 🚀 Let's go through the **answers + explanations** for the practice `this` questions one by one.

◆ Q1: Basic object method

```
js

const person = {
  name: "Rishav",
  greet: function() {
    console.log(this.name);
  }
};

person.greet();
```

✅ **Output:**

```
nginx
```

```
Rishav
```

👉 Because `this` refers to the object `person` (method call).

◆ Q2: Extracted method

js

```
const person = {  
  name: "Rishav",  
  greet: function() {  
    console.log(this.name);  
  }  
};  
  
const greetFn = person.greet;  
greetFn();
```

✓ Output:

javascript

undefined

👉 The function is called standalone, so in **strict mode** `this = undefined`, otherwise in non-strict mode it becomes the global object (`window`), which doesn't have `name`.

◆ Q3: Arrow inside object

js

```
const obj = {  
  name: "Rishav",  
  sayName: () => {  
    console.log(this.name);  
  }  
};  
obj.sayName();
```

✓ Output:

javascript

undefined

👉 Arrow functions don't have their own `this`. Here, `this` comes from the outer scope (global), not from `obj`.

◆ Q4: Arrow inside normal function

js

```
const obj = {
  name: "Rishav",
  sayName: function() {
    const inner = () => {
      console.log(this.name);
    };
    inner();
  }
};
obj.sayName();
```

✓ Output:

nginx

Rishav

👉 Arrow function inherits `this` from its enclosing function (`sayName`), which is called on `obj`.

◆ Q5: setTimeout with method

js

```
const obj = {
  value: 42,
  log: function() {
    setTimeout(function() {
      console.log(this.value);
    }, 1000);
  }
}
```

```
};  
obj.log();
```

✓ Output after 1 sec:

```
javascript
```

```
undefined
```

👉 Inside `setTimeout`, the callback is a normal function → `this` = global object (or `undefined` in strict). Not `obj`.

◆ Q6: Fix with arrow

```
js  
  
const obj = {  
  value: 42,  
  log: function() {  
    setTimeout(() => {  
      console.log(this.value);  
    }, 1000);  
  }  
};  
obj.log();
```

✓ Output after 1 sec:

```
42
```

👉 Arrow function inherits `this` from `log`, which belongs to `obj`.

◆ Q7: call, apply, bind

```
js  
  
function greet(city) {  
  console.log(`Hello ${this.name} from ${city}`);  
}
```

```
}

const user = { name: "Rishav" };

greet.call(user, "Delhi");
greet.apply(user, ["Mumbai"]);
const boundFn = greet.bind(user, "Bangalore");
boundFn();
```

✓ Outputs:

csharp

Hello Rishav from Delhi
Hello Rishav from Mumbai
Hello Rishav from Bangalore

◆ Q8: arguments trap

```
js

var length = 10;
function fn() {
  console.log(this.length);
}

const obj = {
  length: 5,
  method(fn) {
    fn();
    arguments[0]();
  }
};

obj.method(fn, 1);
```

✓ Outputs:

10

2

👉 Breakdown:

- `fn();` → standalone call, so `this` = global, `length` = 10 .
- `arguments[0]()`; → here `this` = arguments object, which has `length` = 2 (because 2 arguments were passed).

🔥 These 8 cover **90% of interview traps** with `this` .

Do you want me to now prepare a **super tricky final puzzle** (mixing `this` , arrow, bind, and `setTimeout`) — like the ones FAANG companies ask?

```
explain this
const obj = {
  value: 42,
  log: function() {
    console.log("this value inside first log", this); //42
    setTimeout(function() {
      console.log("this value inside second log", this);
    }, 1000);
  }
};
obj.log();
```

why the log inside the `setTimeOut` logs window and first one logs the object itself

Great debugging question 🔥 Let's carefully step through your code:

js

```
const obj = {
  value: 42,
  log: function() {
    console.log("this value inside first log", this);
```

```
setTimeout(function() {  
  console.log("this value inside second log", this);  
}, 1000);  
}  
};  
obj.log();
```

1. First log

js

```
console.log("this value inside first log", this);
```

- Here, `log` is called as a **method of** `obj` (`obj.log()`).
- In that call, `this` = the **object** `obj` .

✓ So it logs:

bash

```
this value inside first log { value: 42, log: f }
```

2. Inside `setTimeout`

js

```
setTimeout(function() {  
  console.log("this value inside second log", this);  
}, 1000);
```

- The callback is a **normal function**, passed to `setTimeout` .
- When `setTimeout` executes it later, it calls the function **without any object**.
- In that standalone call:
 - In **non-strict mode** → `this` = global object (`window` in browsers).
 - In **strict mode** → `this` = `undefined` .

✓ That's why you see `window` here (or `undefined` in strict).

3. Why are they different?

- **First log** → `this` is tied to the method call (`obj.log()`).
 - **Second log** → the callback loses that context, because `setTimeout` doesn't magically bind `this` to `obj`.
-

4. How to fix it (3 common ways)

◆ Option 1: Arrow function (lexical `this`)

```
js

setTimeout(() => {
  console.log("this inside arrow", this);
}, 1000);
```

👉 Now the arrow inherits `this` from the enclosing `log` method → `obj`.

◆ Option 2: Save reference

```
js

const self = this;
setTimeout(function() {
  console.log("this using saved ref", self);
}, 1000);
```

◆ Option 3: Explicit `bind`

```
js

setTimeout(function() {
  console.log("this using bind", this);
}.bind(this), 1000);
```

✅ Summary:

- The first `this` = `obj` because of `obj.log()`.

- The second `this` = global object because the callback is a standalone normal function.
-

Do you want me to **draw a timeline-style diagram** that shows how `this` flows from `obj.log()` into `setTimeout` and why it gets lost?