Lecture 22: This Keyword in javascript

strict mode vs non strict mode

The First Principle: Turning Silent Errors into Loud Errors

Early versions of JavaScript were designed to be extremely forgiving. The philosophy was, "Don't stop the code from running, even if something looks wrong. Just make a guess and keep going." This is **Non-Strict Mode**, often called "sloppy mode."

The problem with this forgiveness is that it hides potential bugs. Your code might run, but it might be doing something completely different from what you intended. These are called **silent errors**.

Strict Mode was introduced in ES5 (2009) as a way to opt-in to a less forgiving, more secure, and more sensible version of the language.

> The First Principle of Strict Mode: Its purpose is to change "sloppy" JavaScript that has silent errors or "bad syntax" into code that throws immediate, obvious errors. It helps you write better, safer code by forcing you to be more explicit.

The Analogy: The Lenient Parent vs. The Strict Teacher

- **Non-Strict Mode (The Lenient Parent):** You make a mess in your room (write bad code). The lenient parent just shrugs and cleans it up for you (the code runs, but maybe not how you expect). You might not even realize you made a mess.
- **Strict Mode (The Strict Teacher):** You make a mess on your homework (write bad code). The strict teacher immediately gives you an "F" and tells you exactly what you did wrong. It's harsh, but you learn from your mistake and don't make it again.

How to Use It

You enable strict mode by putting the following string at the very top of a file or a function:

```
'use strict';
```

```
// File-level strict mode (Recommended) 'use-strict'; function myStrictFunction() { // This whole function runs in strict mode } // ------------------------------------ function mySloppyFunction() { // This function is in non-strict mode function myInnerStrictFunction() { 'use strict'; // This inner function is in strict mode } }
```

The Core Differences in Detail

Let's look at the most important changes strict mode introduces.

1. Prevents Accidental Global Variables

This is the biggest and most important feature. In non-strict mode, if you assign a value to a variable that hasn't been declared, JavaScript "helpfully" creates that variable on the global object (`window`). This is a huge source of bugs.

- **Non-Strict Mode (Sloppy):**

  ```
  function createMistake() { // Forgot to use `let`, `const`, or `var` mistake = "I am now a global variable!"; // No error! } createMistake(); console.log(window.mistake); // "I am now a global variable!"
  ```

- **Strict Mode (Safe):**

  ```
  'use strict'; function avoidMistake() { mistake = "This will not work"; // Loud error! } // avoidMistake(); // Throws ReferenceError: mistake is not defined
  ```

2. Changes the Behavior of `this`

As we discussed, in a simple function call, `this` behaves differently.

- **Non-Strict Mode (Sloppy):** `this` defaults to the global object (`window`).

```
function logThis() { console.log(this); } logThis(); // Logs the `window` object
```

- **Strict Mode (Safe):** `this` is `undefined`. This prevents you from accidentally modifying the global object.

```
'use strict'; function logThis() { console.log(this); } logThis(); // Logs `undefined`
```

3. Prohibits Duplicate Parameter Names

This is just bad practice, and strict mode forbids it.

- **Non-Strict Mode (Sloppy):** The last parameter wins, but there is no error.

```
function sloppy(param1, param1) { console.log(param1); // The value of the second `param1` is used } sloppy(10, 20); // Logs 20
```

- **Strict Mode (Safe):** This is a syntax error. The code won't even run.

```
'use strict'; // function strict(param1, param1) { ... } // Throws SyntaxError: Duplicate parameter name not allowed in this context
```

4. Makes `eval()` Safer

The `eval()` function is powerful and dangerous because it can execute arbitrary strings as code. Strict mode puts a cage around it.

- **Non-Strict Mode (Sloppy):** `eval()` can create new variables in the surrounding scope.

```
eval("var x = 2;"); console.log(x); // 2
```

- **Strict Mode (Safe):** Variables created in an `eval()` call stay inside that `eval()` 's scope.

```
'use strict'; eval("var x = 2;"); // console.log(x); // Throws ReferenceError: x is not defined
```

Summary Table

Feature / Behavior	Non-Strict Mode (Sloppy) 🚨	Strict Mode ✅
Undeclared Variables	Creates a global variable	Throws a `ReferenceError`
`this` **in Simple Calls**	Points to the global object (`window`)	Is `undefined`
Duplicate Parameters	Allowed, last one wins	Throws a `SyntaxError`
`eval()` **Scope**	Can create variables in surrounding scope	Variables stay inside `eval()` scope
Deleting a Variable	`delete myVar` silently fails	Throws a `SyntaxError`
Octal Literals	`010` is 8	Throws a `SyntaxError`

Is This Still Relevant Today?

Yes, more than ever! The good news is that you often get strict mode for free.

- **JavaScript Modules (** `import` **/** `export` **) are always in strict mode.**
- **Code inside a** `class` **body is always in strict mode.**

Because modern JavaScript development is almost entirely based on modules, most of the code you write is already running in strict mode by default.

Recommendation: Always write in strict mode. Put `'use strict';` at the top of any standalone script files that aren't modules to ensure you are writing safe, modern, and robust code.

Imagine I write a sentence on a piece of paper: **"You are wearing a red shirt."**

Who is "you"? You can't know by just looking at the paper. The meaning of "you" depends entirely on **who I am speaking to when I say the sentence.**

- If I say it to Alice, "you" is Alice.
- If I say it to Bob, "you" is Bob.
- If I shout it into an empty room, "you" is... nobody? Or the room itself? (This is where it gets interesting).

The sentence (`function`) is the same. The context of the call (`how I say it`) determines the meaning of "you" (`this`).

The First Principle: The Call-Site

This leads us to the single most important principle for understanding `this`:

> The value of this is not determined by where a function is written. It is determined by HOW that function is CALLED.

That's it. This is the bedrock. To know what `this` is, you don't look at the function's code; you find the line where the function is being invoked (the **call-site**) and look at how it's being called.

Let's use this one principle to explore every scenario.

Part 1: The Four Ways to Call a Function

Every use of `this` in a regular function falls into one of these four invocation patterns. For each one, we will ask: "How was it called?"

Rule 1: The Method Call (The most intuitive)

When a function is called *on* an object, using a dot.

- **How it's called:** `object.myFunction()`
- **Intuition:** The object is the one "in charge" of the call.
- `this` **is:** The object to the left of the dot.

```
const person = { name: "Alice", speak: function() { // Call-site: person.s
peak() // The object to the left is 'person'. // So, `this` is `person`. c
onsole.log(`My name is ${this.name}`); } }; person.speak(); // My name is
Alice
```

Rule 2: The Simple Call (The most confusing)

When a function is called by itself, with no object on the left.

- **How it's called:** `myFunction()`
- **Intuition:** No one is in charge of the call. It's just... called. This is the **default** case.
- `this` **is:** Dependent on the "strictness" of the environment.

Case 2a: Non-Strict Mode (The "Sloppy" Old Way)

- **The Rule:** If nobody is in charge, the ultimate boss, the **global object**, takes over.
- **In a Browser:** The global object is `window`.
- **In Node.js:** The global object is `global`.

```
// Running in a browser, in non-strict mode function whoAmI() { // Call-si
te: whoAmI() // No object on the left. Non-strict mode. // `this` defaults
to the global object. console.log(this === window); // true } whoAmI();
```

Case 2b: Strict Mode (The Modern, Safe Way)

- **The Rule:** If nobody is in charge, then nobody is in charge.
- `this` **is:** `undefined`. This is much safer because it prevents you from accidentally modifying the global object. Your code will throw an error immediately if you try, which is a good thing!

```
'use strict'; function whoAmI() { // Call-site: whoAmI() // No object on t
he left. Strict mode is on. // `this` is undefined. console.log(this === u
ndefined); // true // console.log(this.name); // This would throw a clear
error! } whoAmI();
```

Rule 3: The Explicit Call (The "Bossy" Way)

When you want to *force* a specific value for `this`, no matter how the function is called.

- **How it's called:** `myFunction.call(...)` or `myFunction.apply(...)`
- **Intuition:** You are explicitly telling the function, "Run now, and when you do, `this` must be *this specific object I'm giving you*."

- `this` **is:** The first argument you pass to `.call()` or `.apply()`.

```
function speak() { console.log(`My name is ${this.name}`); } const person1
= { name: "Alice" }; const person2 = { name: "Bob" }; // We are forcing `t
his` to be person1 speak.call(person1); // My name is Alice // Now we forc
e `this` to be person2 speak.call(person2); // My name is Bob
```

Rule 4: The Constructor Call (The "Factory" Way)

When you call a function using the `new` keyword.

- **How it's called:** `new MyFunction()`
- **Intuition:** The `new` keyword is a special instruction to build a brand new object.
- `this` **is:** The brand new, empty object that was just created. The constructor's job is to populate `this`.

```
class Person { constructor(name) { // Call-site: new Person(...) // `new`
creates an empty object and makes `this` point to it. this.name = name; //
Populating the new object } } const person1 = new Person("Alice"); // `thi
s` inside the constructor was the object that became person1 console.log(p
erson1); // { name: 'Alice' }
```

Part 2: The Big Exception - Arrow Functions

Arrow functions (`=>`) are special. They were designed to solve common frustrations with `this`. They break the first principle on purpose.

> Arrow functions do not have their own this. They inherit this from their parent scope, just like any other variable.

They don't care how they are called. They only care about where they are *written*.

The Intuition: An arrow function is like a loyal assistant. If you ask the assistant "Who is your boss?", they don't say "Me!". They point to *their* boss (the `this` of the function they were written inside).

```
const user = { name: "Alice", hobbies: ['reading', 'coding'], printHobbies: function() { // For this regular function, `this` is `user` (Rule 1). this.hobbies.forEach(hobby => { // This is an arrow function. It doesn't have its own `this`. // It inherits `this` from its parent, `printHobbies`. // So, `this` here is also `user`. console.log(`${this.name} likes ${hobby}`); }); } }; user.printHobbies(); // Alice likes reading // Alice likes coding
```

Part 3: `this` Outside of Any Function (The Global Context)

What is `this` when it's not inside any function at all?

- **Intuition:** If you're not in any specific room (`function`), you must be in the main building (`global scope`). So, `this` refers to the "main building" object.
- **In a Browser (in a regular `<script>` tag):** The "main building" is the `window` object.

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

- **In Node.js:** This is the tricky one. **Every file in Node.js is its own module.** Node wraps your file's code in a function. Because of how Node calls that function, `this` at the top level is a reference to `module.exports` , which is an empty object `{}` by default, **NOT** the `global` object.

  ```
  // In a Node.js file console.log(this === global); // false console.log(this); // {} console.log(this === module.exports); // true
  ```

Final Intuitive Summary

1. **Is the function called with a dot? (`person.speak()`)**
   - Yes? `this` is the object to the left of the dot.
2. **Is it an arrow function? (`=>`)**
   - Yes? `this` is whatever `this` was in the parent scope. Stop here.

3. **Is the `new` keyword used? (`new Person()`)**
    - Yes? `this` is the brand new object being created.
4. **Is `.call()` or `.apply()` used? (`speak.call(person)`)**
    - Yes? `this` is the object passed as the first argument.
5. **None of the above? (It's a simple call like `speak()`)**
    - Are you in `'use strict'` mode? `this` is `undefined`.
    - Are you in "sloppy" mode? `this` is the global object (`window` or `global`).

Arrow function in more Detail

Of course. This is one of the most important modern features of JavaScript, and understanding its relationship with `this` is key to writing clean, bug-free code.

Let's dissect it from the ground up.

Part 1: The "Why" - The Classic Problem Arrow Functions Were Born to Solve

Before arrow functions, a very common and frustrating bug would appear in JavaScript. To understand the genius of arrow functions, you must first feel the pain of the problem they solve.

Let's build a simple `Stopwatch` object.

```
// The Old, Painful Way function Stopwatch() { this.seconds = 0; this.star
t = function() { // We want to increment `this.seconds` every second. setI
nterval(function() { // Uh oh. We have a problem here. // What is `this` i
nside THIS function? console.log(this.seconds); this.seconds++; }, 1000);
}; } const myWatch = new Stopwatch(); // myWatch.start();
```

When you run `myWatch.start()`, what happens? `this.seconds` will be `undefined`, and the output will be `NaN` (Not a Number) every second.

Why did this fail?
Let's apply our first principle: **The value of `this` is determined by how the function is called.**

1. When we call `myWatch.start()`, the `this` inside the `start` method is correctly set to `myWatch` (Rule 1: Method Call).
2. But the function we pass to `setInterval` is a **regular function**. The `setInterval` mechanism calls it like a **simple function call** (Rule 2).
3. In a simple function call, `this` is either the global object (sloppy mode) or, more importantly, `undefined` (strict mode).
4. Therefore, inside our callback, we have lost the context of `myWatch`. We are trying to do `undefined.seconds++`, which results in an error or `NaN`.

The Old, Clumsy Workarounds

For years, developers fought this problem with a few common patterns:

Workaround 1: `that = this` (The Closure Trick)
Create a variable to capture the correct `this` before you lose it.

```
function Stopwatch() { this.seconds = 0; const that = this; // Capture the
correct `this` this.start = function() { setInterval(function() { // Use t
he captured variable instead of the wrong `this` console.log(that.second
s); that.seconds++; }, 1000); }; }
```

Workaround 2: `.bind(this)` (The Explicit Way)
Explicitly create a new function where `this` is permanently locked to the correct value.

```
function Stopwatch() { this.seconds = 0; this.start = function() { setInte
rval(function() { console.log(this.seconds); this.seconds++; }.bind(thi
s)); // Bind the function to the correct `this` }; }
```

Both of these work, but they are verbose and add extra noise to the code. We are fighting against the language's own rules.

Part 2: The Solution - The Arrow Function's First Principle

The designers of ES6 saw this common problem and created a new kind of function with one special, game-changing rule.

> An arrow function does not have its own this. It lexically inherits this from its parent scope.

Let's break that down.

- **"Does not have its own `this`"**: It completely ignores the four rules of `this` (method call, simple call, etc.). They do not apply to it.
- **"Lexically inherits"**: This is the key. It doesn't care how it's *called*. It only cares where it is physically *written* in the code. It looks outside of itself to the surrounding code block and uses whatever `this` is in there.

The Analogy: Transparent Glass
Think of a regular function as a **one-way mirror**. You can't see the `this` of the room outside. It creates its own reflection, its own `this`.

An arrow function is like a sheet of **perfectly transparent glass**. It doesn't have its own reflection. It just lets you see the `this` of the room it's in.

The Modern, Clean Solution

Let's rewrite our `Stopwatch` using an arrow function.

```
function Stopwatch() { this.seconds = 0; this.start = function() { // The `this` in this scope correctly points to the stopwatch instance. setInterval(() => { // This is an arrow function. It has no `this` of its own. // It inherits `this` from its parent, the `start` function. // Therefore, `this` here is the stopwatch instance. It just works. console.log(this.seconds); this.seconds++; }, 1000); }; } const myWatch = new Stopwatch(); myWatch.start(); // This works perfectly!
```

The code is clean, intuitive, and does exactly what we want without any workarounds. The context is preserved automatically.

Part 3: The Consequences - When NOT to Use Arrow Functions

Because arrow functions *always* inherit `this`, they are the wrong tool when you explicitly **want** your function to have its own `this` based on how it's called.

1. Object Methods (The most common mistake)

If you define a method on an object using an arrow function, it will inherit `this` from the global scope, which is not what you want.

```
const person = { name: "Alice", // WRONG WAY - using an arrow function for a method sayHi: () => { // This `this` is inherited from the global scope (where the object is defined). // In Node, it would be `module.exports` ({}). In a browser, it's `window`. console.log(`Hi, my name is ${this.name}`); }, // CORRECT WAY - using a regular function speak: function() { // This `this` is determined by the call-site (`person.speak()`), so `this` is `person`. console.log(`Hi, my name is ${this.name}`); } }; person.sayHi(); // Hi, my name is undefined person.speak(); // Hi, my name is Alice
```

2. Event Listeners in HTML

When you add an event listener, you often want `this` to refer to the element that was clicked. A regular function does this automatically. An arrow function does not.

```
const button = document.getElementById('myButton'); // CORRECT WAY button.addEventListener('click', function() { console.log(this); // `this` is the button element itself. this.textContent = 'Clicked!'; }); // WRONG WAY (if you need to reference the button) button.addEventListener('click', () => { // `this` is inherited from the global scope (`window`). It is NOT the button. console.log(this); // `this` is `window`. // this.textContent = 'Clicked!'; // This would be window.textContent, which is an error. });
```

Final Intuitive Checklist

Here is a simple way to decide which function type to use:

1. **Am I writing a callback inside a method?** (e.g., inside `.forEach`, `.map`, `setTimeout`, `.then`)
   - **YES** -> Almost certainly **use an arrow function** to preserve the `this` context.
2. **Am I writing a method directly on an object or a class prototype?**
   - **YES** -> **Use a regular function** (or the method syntax in a class) so that it gets its own `this` from the object.
3. **Am I writing an event listener where I need to access the element that triggered it?**