This keyword in constructor

function Counter () {
  var count = 0; // PRIVATE variable (not accessible from outside or via **this**)
  console.log("this-->>CounterFunction-->>", this);

}

var counter1 = new Counter();

here this refers to

Excellent question, Vipul 👋

Let's break it down carefully and see **what `this` refers to** in this exact code:

```
js

function Counter () {
    var count = 0;  // private variable
    console.log("this-->>CounterFunction-->>", this);
}

var counter1  = new Counter();
```

✅ Step-by-step explanation

1. **When you call a function with `new` keyword**, like:

   ```
   js

   var counter1 = new Counter();
   ```

   then **a brand new empty object** is automatically created by JavaScript.
2. That new object is automatically linked to the function's **prototype** (so it can access methods defined on **`Counter.prototype`**).
3. Inside the constructor function (**`Counter`**), the keyword **`this`** refers to that **newly created object**.
4. Finally, if the function doesn't return anything explicitly, **the new object (`this`) is returned automatically**.

🔍 So, in your case:

Inside the **`Counter`** function:

```js
console.log("this-->>CounterFunction-->>", this);
```

👉 **`this`** refers to **the newly created instance object** — the same object that gets stored in **`counter1`**.

🧠 Therefore:

When you run the code:

```js
var counter1  = new Counter();
```

You'll see something like:

```bash
this-->>CounterFunction-->> Counter {}
```

and after execution:

```js
console.log(counter1); // Counter {}
```

⚠️ If you had called it without `new`:

```js
Counter();
```

Then, in **non-strict mode**,
👉 **`this`** would refer to the **global object** (**`window`** in browser or **`global`** in Node.js).

But in **strict mode**,
👉 **`this`** would be **`undefined`**, and you'd get a **`TypeError`** if you tried to use **`this`** to set properties.

✅ Final Answer:

> In your code, since you called it with **`new Counter()`**,
> **`this`** **refers to the newly created empty object (the instance of Counter).**

Would you like me to show how this same behavior changes if we remove the **`new`** keyword or use a class instead?

and insdie incrementCounter:

function Counter () {
var count = 0; // PRIVATE variable (not accessible from outside or via **this**)
console.log("this-->>CounterFunction-->>", this);
this.incrementCounter = function () {
console.log(this);
}

}

var counter1 = new Counter();

this refres to inside incrementCounter

Excellent follow-up, Vipul 👋

Let's unpack this **step-by-step** — because the meaning of **`this` inside `incrementCounter`** depends on **how it is called**, not where it is defined.

✅ Code:

```
js

function Counter () {
    var count = 0;
    console.log("this-->>CounterFunction-->>", this);

    this.incrementCounter = function () {
        console.log("this-->>incrementCounter-->>", this);
    }
}

var counter1 = new Counter();
```

Now let's see what happens.

🧩 Step 1 — Inside `Counter`

When you do:

```
js

var counter1 = new Counter();
```

- **`new`** creates a **new empty object**.
- Inside the function, **`this`** refers to **that new object**.

- You then assign a method **`incrementCounter`** to that object.

So now:

```
js

counter1 = {
  incrementCounter: function() { console.log(this); }
}
```

🧩 Step 2 — Inside `incrementCounter`

Now, when you call:

```
js

counter1.incrementCounter();
```

👉 **`this`** inside **`incrementCounter`** refers to **the object that called it**, i.e. **`counter1`**.

So the output will be:

```
bash

this-->>CounterFunction-->> Counter {}
this-->>incrementCounter-->> Counter {}
```

🧠 So in simple terms:

- Inside **`Counter()`** constructor — **`this`** → **the new object being created**
- Inside **`incrementCounter()`** — **`this`** → **the object through which the method was called**

⚠️ Important Twist:

If you detached the method like this:

```
js

var ref = counter1.incrementCounter;
ref(); // call without object
```

Then:

- In **non-strict mode**, **`this`** → **global object (window)**
- In **strict mode**, **`this`** → **undefined**

Because now it's **not called via `counter1`**, so there's no object context.

✅ Final Answer:

Where	What `this` refers to
Inside **`Counter()`**	The **new object** created by **`new`**
Inside **`incrementCounter()`** when called via **`counter1.incrementCounter()`**	The **same object (`counter1`)**
Inside **`incrementCounter()`** when called directly (like **`ref()`**)	**`undefined`** (in strict mode) or **`window`** (in non-strict mode)

Would you like me to show a visual diagram of how the object and **`this`** are linked in memory when you use **`new Counter()`** and call **`incrementCounter()`**?