

Code analysis explanation



```
function Counter() {}
Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  // this.count++;
  count = 10
  console.log(this.count);
  console.log(count)
};

let c1 = new Counter();

c1.increment()
```

Let's go step by step 📌

Code:

```
js

function Counter() {}
Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  // this.count++;
  count = 10;
  console.log(this.count);
  console.log(count);
};

let c1 = new Counter();
c1.increment();
```

Step 1: Prototype property

```
js

Counter.prototype.count = 0;
```

This means **every instance of Counter** (like `c1`) will have access to the property `count` through the prototype chain.

So:

```
js

c1.count // → 0 (inherited from prototype)
```

⚙️ Step 2: Method increment

```
js
Counter.prototype.increment = function () {
  count = 10;
  console.log(this.count);
  console.log(count);
};
```

Inside this function:

- `count = 10`; — here, **you are not using** `var`, `let`, or `const`, so `count` becomes an **implicit global variable** (in non-strict mode).
- `this.count` refers to the `count` property of the object `c1` (inherited from the prototype).

⚙️ Step 3: Execution

When you call:

```
js
c1.increment();
```

Here's what happens step by step:

1. `count = 10` → creates (or modifies) a **global variable** named `count`.
2. `this.count` → since `c1` doesn't have its own `count` property, JS looks up the prototype chain and finds `Counter.prototype.count = 0`.
3. So `console.log(this.count)` → prints **0**.
4. Then `console.log(count)` → prints **10** (the global one).

✅ Output:

```
0
10
```

⚠️ Important Note:

If you were in **strict mode** (`'use strict'`; at the top), the line `count = 10` would throw a **ReferenceError**, because assigning to an undeclared variable is not allowed in strict mode.

Would you like me to show what happens if we uncomment `this.count++` and remove `count = 10`? That version behaves very differently.



this 10 will become part of global window object

✅ Exactly right — perfect observation! 🧠

Let's break that down clearly 📝

🧠 In your code:

```
js

Counter.prototype.increment = function () {
  count = 10; // <-- no var, let, or const
  console.log(this.count);
  console.log(count);
};
```

Here you **assigned a value to count without declaring it** using var, let, or const.

❓ What happens (in non-strict mode)

JavaScript automatically treats such assignments as **creating a new global variable**.
In a browser environment, **global variables become properties of the window object**.

So effectively:

```
js

count = 10;
```

is equivalent to:

```
js

window.count = 10;
```

That means you can now access it from anywhere as:

```
js

console.log(window.count); // → 10
```

⚠️ If you enable strict mode:

If you add **'use strict';** at the top of your script or function:

```
js

'use strict';
```

```
count = 10; // ❌ ReferenceError
```

You'll get:

```
vbnet
```

```
ReferenceError: count is not defined
```

because JavaScript will no longer allow creation of global variables accidentally.

Quick Summary:

Statement	Scope	Becomes property of window?	Allowed in strict mode?
var count = 10	Function/global (depending on where defined)	✅ Yes (in browsers)	✅ Yes
let count = 10	Block scope	❌ No	✅ Yes
count = 10 (no declaration)	Global (implicit)	✅ Yes (in non-strict mode)	❌ No

So yes — ✅ your statement is **100% correct**:

That **10** becomes part of the global **window** object in non-strict mode.

Would you like me to show a small browser console demo that proves it?



and if we sue var with that count than

Excellent question, Vipul 🙌 — let's see what changes when you use **var**!

Modified code:

```
js

function Counter() {}
Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  var count = 10;
  console.log(this.count);
  console.log(count);
};

let c1 = new Counter();
c1.increment();
```

Step-by-step explanation:

1. `var count = 10;`

- Now `count` becomes a **local variable** inside the `increment()` function.
- It is **not** attached to **this**, **not** global, and **not** accessible outside this function.

2. `this.count`

- Still refers to the `count` property on the object `c1` (which it inherits from `Counter.prototype`).
- That value is still `0`.

3. `console.log(this.count);` → prints `0`

`console.log(count);` → prints `10`

✓ Output:

```
0
10
```

⊘ But what's different now?

Let's compare the three cases:

Declaration	Scope	Attached to window?	Accessible outside function?
<code>count = 10</code>	Global (implicit)	✓ Yes (<code>window.count</code>)	✓ Yes
<code>var count = 10</code>	Function scope	✗ No	✗ No
<code>let count = 10</code>	Block scope	✗ No	✗ No

🧩 Summary:

- `count = 10` → becomes **global** (→ `window.count = 10` in browser).
- `var count = 10` → becomes **local to the function** (not global).
- `this.count = 10` → creates/updates a **property on the object instance** (`c1` in this case).

Would you like me to show what happens if we use `this.count = 10` instead — and how it affects the instance vs prototype?