Difference in counter functions

```
function Counter () {
   var count = 0;
   this.incrementCounter = function () {
      count++;
      console.log(count);
   }

   this.decrementCounter = function () {
      count--;
      console.log(count);
   }
}


var counter1  = new Counter();
var counter2 = new Counter();
counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();

counter2.incrementCounter();
counter2.incrementCounter();
counter2.decrementCounter();
```

if we write the above one like this than what is the difference between them explain it tol me like a new bie.

```
function Counter () {
   this.count = 0;
   this.incrementCounter = function () {
      this.count++;
      console.log(this.count);
   }

   this.decrementCounter = function () {
      this.count--;
      console.log(this.count);
   }
}
```

```
// console.log("1->>", Counter)


var counter1  = new Counter();
var counter2 = new Counter();
counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();

counter2.incrementCounter();
counter2.incrementCounter();
counter2.decrementCounter();
```

Perfect question, Vipul 👏 — this is a very important difference in **how variables are stored and protected in JavaScript**.
Let's go step by step like a **newbie** 👇

🧩 Version 1 (using `var count = 0;`)

```js
function Counter() {
  var count = 0;

  this.incrementCounter = function () {
    count++;
    console.log(count);
  };

  this.decrementCounter = function () {
    count--;
    console.log(count);
  };
}
```

🧠 Explanation:

- Here, `count` is declared using `var` **inside** the function.
- That means `count` is a **local variable**, not attached to the object created by `new Counter()`.
- The only way to access or change `count` is through the methods (`incrementCounter`, `decrementCounter`).
- This happens because those methods **form a closure** — they remember the `count` variable from their surrounding function even after it finishes running.

✅ Result:

Each time you do `new Counter()`, a **new private `count`** is created in memory for that object.
You **cannot access** it from outside:

```js
console.log(counter1.count); // ❌ undefined
```

So it's like:

- **`count`** is hidden (private).
- Only functions inside can touch it.

🧩 Version 2 (using `this.count = 0;`)

```js
function Counter() {
  this.count = 0;

  this.incrementCounter = function () {
    this.count++;
    console.log(this.count);
  };

  this.decrementCounter = function () {
    this.count--;
    console.log(this.count);
  };
}
```

🧠 Explanation:

- Here, **`this.count`** means the variable is **attached directly to the object** created by **`new Counter()`**.
- So now **`count`** is **public**, not private.
- You can easily access or change it from outside:

  ```js
  console.log(counter1.count); // ✅ shows value
  counter1.count = 100; // ✅ can directly change it
  ```

✅ Result:

Each time you do **`new Counter()`**, a **new object** is created with its own **`count`** property.

But unlike version 1, it's **not protected** — anyone can modify it from outside.

⚖️ In Simple Words

Feature	Version 1 (`var count`)	Version 2 (`this.count`)
Where stored	Inside closure (private memory)	As object property
Accessible outside?	❌ No	✅ Yes
Protected from outside modification?	✅ Yes	❌ No
Each new object gets separate memory?	✅ Yes	✅ Yes
Uses closure?	✅ Yes	❌ No

🧠 Example difference in action

```js
// Version 1
var c1 = new Counter();
c1.incrementCounter(); // 1
c1.incrementCounter(); // 2
console.log(c1.count); // ❌ undefined

// Version 2
var c2 = new Counter();
c2.incrementCounter(); // 1
c2.incrementCounter(); // 2
console.log(c2.count); // ✅ 2
c2.count = 100; // 😬 manually changed!
c2.incrementCounter(); // 101
```

💡 Easy way to remember

> **If you use `var`, it's private.**
> **If you use `this`, it's public.**

Would you like me to show a **diagram** that visually explains how memory and closures differ between the two versions (private vs public count)?

on emore this if we write it using this.count and we write var count than is there also thing that oen is attached to the prototype and another nor

Excellent question again, Vipul 🔥 — you're connecting the dots perfectly now!

Let's break this **clearly and simply** 👇

🧩 Case 1 — Using `var count`

```js
function Counter() {
  var count = 0;

  this.incrementCounter = function () {
    count++;
    console.log(count);
  };
}
```

✅ Explanation:

- **`var count`** lives **inside the constructor function's local scope**.
- It's **not** attached to the object (**`this`**) or the prototype.
- Each time you do **`new Counter()`**, a **new private variable `count`** is created in that object's memory.

So:

- Not in prototype
- Not in object
- Only in **closure memory** (hidden private area)

That's why you can't access it like:

```js
console.log(counter1.count); // ❌ undefined
```

🧩 Case 2 — Using `this.count`

```js
function Counter() {
  this.count = 0;

  this.incrementCounter = function () {
    this.count++;
    console.log(this.count);
  };
}
```

✅ Explanation:

- **`this.count`** becomes a **property of the object** created by **`new Counter()`**.
- It's **not** on the prototype — it's directly on the instance (object).

You can check this:

```js
console.log(counter1.hasOwnProperty('count')); // ✅ true
console.log(Counter.prototype.hasOwnProperty('count')); // ❌ false
```

🧩 Case 3 — When something goes to the prototype

You have to **explicitly** add methods or properties to the prototype, like this:

```js
function Counter() {
  this.count = 0; // object property
}

Counter.prototype.incrementCounter = function () {
  this.count++;
  console.log(this.count);
};
```

✅ Here:

- `count` → lives on the **object**
- `incrementCounter` → lives on the **prototype**

So:

- Every new object gets its own `count`
- But all objects **share one single function** `incrementCounter` (saves memory)

⚖️ Summary Table

Case	Variable type	Where stored	Shared via prototype?	Accessible outside?
`var count`	Local (private)	Closure memory	❌ No	❌ No
`this.count`	Instance property	Object itself	❌ No	✅ Yes
`Counter.prototype.method`	Prototype method	Prototype	✅ Yes	✅ Yes (through prototype chain)

💡 Easy way to remember

- `var` → private (closure)
- `this` → instance (object itself)
- `prototype` → shared (common for all objects)

Would you like me to show a **diagram** (like bubbles showing closure, object, and prototype layers) to visualize where each one lives in memory?

yes

Perfect 👍 Vipul — let's visualize this step by step with **clear diagrams** so you can "see" what happens in memory.

🧩 Case 1 — Using `var count` (Private via Closure)

```js
function Counter() {
  var count = 0;

  this.incrementCounter = function () {
    count++;
    console.log(count);
```

```
  };
}
```

🧠 What happens in memory

```csharp
[Function Execution Context of Counter]
     │
     ├── var count = 0    ← private (only inside Counter)
     ├── this.incrementCounter = function() {...}  ← saved on object
     │
     ↓
[ Object created by new Counter() ]
     ├── incrementCounter()  → has closure over "count"
     └── (no 'count' property visible)
```

💬 Example:

```js
var c1 = new Counter();
c1.incrementCounter(); // 1
console.log(c1.count); // ❌ undefined
```

✅ **`count`** is **hidden inside closure memory**,
only accessible by the function that created it.

🧩 Case 2 — Using `this.count` (Public property on object)

```js
function Counter() {
  this.count = 0;

  this.incrementCounter = function () {
    this.count++;
    console.log(this.count);
  };
}
```

🧠 What happens in memory

```csharp
[ Object created by new Counter() ]
     ├── count = 0                ← directly on object
     ├── incrementCounter() {...} ← also directly on object
     ↓
[ Prototype Chain ]
     └── Counter.prototype (empty)
```

💬 Example:

```js
var c1 = new Counter();
c1.incrementCounter(); // 1
console.log(c1.count); // ✅ 1
c1.count = 100;        // 😬 can change from outside
```

✅ `count` is **public** — visible and editable.

🧩 Case 3 — Using Prototype for methods (Memory-efficient way)

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

Counter.prototype.incrementCounter = function () {
  this.count++;
  console.log(this.count);
};
```

🧠 What happens in memory

```csharp
[ Object created by new Counter() ]
   ├── count = 0          ← on object
   ↓
[ Counter.prototype ]
   ├── incrementCounter()  ← shared by all objects
```

If you create two objects:

```js
var c1 = new Counter();
var c2 = new Counter();
```

👉 Both share the **same function**:

```js
console.log(c1.incrementCounter === c2.incrementCounter); // ✅ true
```

✅ Saves memory — one function shared
✅ Each object has its own `count`

🔍 Quick Visual Summary

Case	Variable	Stored In	Access	Shared?	Protected?
`var count`	closure	inside function (private memory)	❌	❌	✅ yes

Case	Variable	Stored In	Access	Shared?	Protected?
`this.count`	object property	object instance	✅	❌	❌
`Counter.prototype.method`	prototype method	shared prototype	✅	✅	❌

💡 Super simple analogy

Concept	Like in real life
`var count`	your **password** — only you know
`this.count`	your **name tag** — anyone can read/change
`prototype.method`	a **common rule** everyone follows (shared for all)

Would you like me to create a **visual diagram image** (bubbles/arrows showing object, prototype, and closure memory) so you can save it for revision?

1. means when we add using this it won't get added to the prototype but to object itself .
2. can we add count to the prototype also and if we add it to prototype measn it is the common one which all the objects shares measn if we increases it from one object than it will be increased /decreased for all.
3. what is the difference between when we add something to this adn rpototype and why we need to add to prototype if we can get it using this than why prototype.

explain the above in easy and to a new bie person way

Perfect Vipul 👏👏
You've just hit the **most important concept in JavaScript's OOP** — the difference between **`this`** and **`prototype`**.
Let's go one by one in very **simple beginner language** 👇

✅ 1️⃣ When we use `this` inside a constructor

```js
function Counter() {
  this.count = 0;
  this.increment = function () {
    this.count++;
    console.log(this.count);
  };
}
```

When you use **`this`**, everything you attach to it goes **directly to that particular object**, not to the prototype.

So if you do this:

```js
let c1 = new Counter();
let c2 = new Counter();
```

Then memory looks like this:

```css
c1 → { count: 0, increment: function() {...} }
c2 → { count: 0, increment: function() {...} }
```

🟢 Each object has its **own copy** of **`count`** and its **own copy** of **`increment()`**.

So changing one does **not** affect the other:

```js
c1.count++; // c1 = 1
console.log(c2.count); // 0 → not affected
```

✅ Means:
`this` → property belongs only to that object.

✅ 2️⃣ Can we add `count` to the prototype?

Yes, we can.

```js
function Counter() {}

Counter.prototype.count = 0;

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

Now if you do this:

```js
let c1 = new Counter();
let c2 = new Counter();
```

👉 Both objects will **share the same prototype**.

So memory looks like this:

```javascript
Counter.prototype → { count: 0, increment: function() {...} }

c1 → {}
c2 → {}
```

They both **refer to the same count** through prototype:

```js
c1.increment(); // count = 1
c2.increment(); // count = 2 (same shared one!)
```

✅ Means:

> If you change `count` from one object, it changes for all — because it's the **same shared value** from the prototype.

✅ 3️⃣ Why do we need the prototype at all?

Excellent question 💡

Let's understand this like a real-world example 👇

🧍 Using `this`

If every person has their **own separate notebook**,
and you write the same rule (like "Don't be late") in each notebook —
you're **copying the same thing again and again**.

🧠 That's what happens when you use `this` —
each object gets its **own copy** of that function in memory.

👨‍👩‍👧‍👦 Using `prototype`

Now imagine instead, you write the rule once on a **classroom board**.
Everyone can just look at it — they don't each need a copy.

🧠 That's the prototype —
a **shared space** that all objects can use,
so you **save memory** and keep code DRY (Don't Repeat Yourself).

⚖️ Comparison

Feature	`this`	`prototype`
Where stored	Inside the object	On the prototype (shared)
Shared?	❌ No, separate for each	✅ Yes, shared for all
Memory efficient?	❌ No	✅ Yes
Can access directly?	✅ Yes (`obj.property`)	✅ Yes (via lookup)
Common use	Data/properties that are unique to object	Functions or constants shared by all

---

💡 Rule of Thumb

🧩 Use **`this`** for **object-specific data** (like **`count`**, **`name`**, etc.)
🧩 Use **`prototype`** for **shared methods** (like **`increment`**, **`display`**, etc.)

🧠 Example combining both

```js
function Counter() {
  this.count = 0; // unique for each object
}

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

let c1 = new Counter();
let c2 = new Counter();

c1.increment(); // 1
c2.increment(); // 1 (separate count values)
```

Here:

- **`count`** is **different** for each object (stored with **`this`**)
- **`increment()`** is **shared** (stored in prototype)

✅ This is the **best practice** used in real-world code.

Would you like me to make a **diagram image** (showing "object has its own data" but "all share same function from prototype")? It'll make this super clear visually.

function Counter() {}

Counter.prototype.count = 0;

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

let c1 = new Counter();
let c2 = new Counter();

1. in this if it is attached to prototype than how you are accessing increment using
c1.increment() but increment is attached to the prototype not to the objec itself.
2. also if we added one more count in the function using this in the same than what happen
3.. also if we attaches coun to the prototype tha we increases it by two way
 count++ and this.count++ than which count increases which count

Fantastic Vipul 👏👏
You've just stepped into **prototype lookup and shadowing**, which is one of the most important and most confusing JavaScript concepts — but I'll make it crystal clear with easy words and examples 💡

Let's go step-by-step on your 3 questions 👇

🧩 Code Reminder

```js
function Counter() {}

Counter.prototype.count = 0;

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

let c1 = new Counter();
let c2 = new Counter();
```

✅ 1️⃣ How can `c1.increment()` work if `increment` is attached to the prototype?

💬 Very good question.

When you call:

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

👉 JavaScript follows something called the **prototype chain lookup**.

Here's what happens step-by-step:

1. JS first looks inside `c1` itself —
   "Does c1 have its own `increment` property?"
   ❌ No, it doesn't.
2. Then JS looks at `Counter.prototype` (the parent) —
   ✅ Finds `increment` there.
3. JS executes that function,
   but **with `this` referring to `c1`** (the object that called it).

That's why `this.count++` means
👉 increment the `count` of `c1`.

So even though the function lives in the prototype,
it **behaves as if it belongs to the object that called it**.

🧠 Simple Analogy

> Imagine your school has a common library (prototype).
> You don't own the library, but you can **use** its books.
> When you read a book, **you** are reading — not the library.

Same way, when **`c1.increment()`** runs,
the method comes from the prototype,
but it **acts on `c1`**.

✅ 2️⃣ What if we add one more `count` inside constructor using `this`?

```js
function Counter() {
  this.count = 10; // each object starts with its own count
}

Counter.prototype.count = 0;

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

let c1 = new Counter();
let c2 = new Counter();
```

Now each object (**`c1`**, **`c2`**) has **its own `count` (10)**.
Prototype also has a **`count`** (0), but objects have their **own copy**.

When we do:

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

👉 JS looks for **`count`** on **`c1`** first —
Finds **`this.count = 10`** there → increases to 11.
It **never touches the prototype count** anymore.

✅ So:

- Each object now has its **own `count`** (independent)
- Prototype's **`count`** remains unchanged

💬 Example output

```js
c1.increment(); // 11
c2.increment(); // 11
```

```
console.log(Counter.prototype.count); // 0 → still same
```

✅ Because once you define **`this.count`**,
it **shadows** (hides) the **`count`** in the prototype.

✅ 3️⃣ If `count` is on prototype, what's the difference between `count++` and `this.count++`?

Let's take this code again:

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

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

⚡ Case A — Using `this.count++`

- JS looks for **`count`** on the **object (`c1`)**.
- If not found, it looks in the **prototype**.
- It finds **`count = 0`** in prototype.
- But then it creates a **new `count` on c1 itself**,
  and increments that.

Let's test it 👇

```js
let c1 = new Counter();
let c2 = new Counter();

c1.increment(); // 1
console.log(c1.count); // 1 (own property now)
console.log(c2.count); // 0 (still prototype count)
console.log(Counter.prototype.count); // 0
```

✅ Explanation:

- After first call, c1 got its **own count = 1**
- Prototype's count was not touched
- c2 still sees prototype's count (0)

⚡ Case B — Using `count++` (without `this`)

If you just write:

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

💥 This will throw an **error**, because
`count` is not defined anywhere inside the function.
There's no global `count`, and no local `count`.

👉 `count` must always be referenced with either:

- `this.count` (object's or prototype's property), or
- `Counter.prototype.count` (explicitly)

✅ So the difference is:

Expression	What happens
`this.count++`	Looks on the object first → if not found, uses prototype's → then creates a local copy on object
`Counter.prototype.count++`	Directly modifies the prototype's shared count for all objects
`count++`	❌ Error (count is undefined)

💬 Example showing both

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

let c1 = new Counter();
let c2 = new Counter();

Counter.prototype.count++; // increases shared count
console.log(Counter.prototype.count); // 1
console.log(c1.count); // 1 (since it reads from prototype)
console.log(c2.count); // 1

c1.increment(); // uses this.count++ → makes c1's own count = 2
console.log(c1.count); // 2 (own property now)
console.log(c2.count); // 1 (still prototype count)
console.log(Counter.prototype.count); // 1
```

🎯 Final Summary

Concept	Description	Affects whom?
`this.count`	Belongs to individual object	Only that object
`Counter.prototype.count`	Shared by all objects	All objects see same value
`this.count++`	Creates/updates **`count`** on the object	Only that object
`Counter.prototype.count++`	Updates shared prototype value	Affects all objects

Concept	Description	Affects whom?
`count++` (no `this`)	❌ Error (undefined variable)	-

💡 Simple Analogy

> 🧍 `this.count` = your **personal wallet** 💰
> 👨‍👩‍👧 `Counter.prototype.count` = a **family wallet** 🏦 (shared by all)
> If you take money from your own wallet → affects only you.
> If you take from the family wallet → affects everyone.

Would you like me to create an **illustrated image** (showing prototype, object, and how lookup + shadowing works for `this.count` and `prototype.count`)? It'll be super helpful to revise.

for this:
function Counter() {}

Counter.prototype.count = 0;

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

let c1 = new Counter();
let c2 = new Counter();

at point 3 you saud that
JS executes that function,
but with this referring to c1 (the object that called it).

this referring to c1 means here?

Excellent follow-up Vipul 🔥 — and yes, this "`this` refers to c1" part is **super important** to truly understand object behavior in JavaScript.

Let's go slowly, in **beginner-friendly steps** 👇

🧩 Code again

```js
function Counter() {}

Counter.prototype.count = 0;

Counter.prototype.increment = function () {
```

```
    this.count++;
    console.log(this.count);
};

let c1 = new Counter();
let c2 = new Counter();

c1.increment();
```

🧠 Step-by-step what happens

When you call:

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

Step 1️⃣

JS checks:

> Does **`c1`** have its own property **`increment`**?

- ❌ No, **`c1`** doesn't have **`increment`**.
- ✅ But its prototype (**`Counter.prototype`**) does.

So JS **finds** the function there.

Step 2️⃣

JS says:

> "Okay, I found the function inside the prototype,
> but who is calling it?"

👉 The **caller** is **`c1`**.

So JS automatically sets:

```js
this = c1
```

for that function call.

Step 3️⃣

Now the line inside the function runs:

```js
this.count++;
```

Since **`this`** is now **`c1`**,
it means:

```js
c1.count++;
```

Step 4️⃣

JS checks if **`c1`** has its own **`count`**:

- If not, it looks in the prototype (finds **`count = 0`**).
- Then creates a new property **`count`** on **`c1`** itself,
  and sets it to **`1`**.

So after this call, memory looks like this:

```css
Counter.prototype → { count: 0, increment: function() {...} }

c1 → { count: 1 }
c2 → {}
```

✅ So **`this`** → refers to **the object before the dot** when you call a function.

💬 In simple words

> In **`c1.increment()`**,
> JS runs the function found in the prototype,
> but treats **`c1`** as **the current object** (so **`this = c1`**).

🧠 Analogy (easy to remember)

Think of the prototype like a **shared remote control** that lives in the living room.
You don't *own* the remote, but when *you* press the button, the TV in *your* room changes channel.

So:

- The **remote** = function on prototype
- The **person pressing it** = object (like **`c1`**)
- The **TV** = data (like **`count`**)

Who pressed it decides which TV (**`count`**) changes —
that's why **`this`** becomes the object that called it.

🧩 Quick proof

Try this 👇

```js
c1.increment(); // 1
console.log(this); // in browser, prints Window (global)
```

and

```js
const test = c1.increment;
test(); // undefined or NaN (because now "this" is not c1)
```

See?
If you remove the `.c1.` part, `this` no longer points to c1.
It depends **on how the function is called**, not where it's defined.

✅ In short

Situation	What this refers to
`c1.increment()`	`c1`
`c2.increment()`	`c2`
`Counter.prototype.increment()`	`Counter.prototype`
`test = c1.increment; test();`	`undefined` (in strict mode) or `window` (non-strict mode)

Would you like me to show a **small diagram** showing this call chain — how `c1.increment()` → finds function in prototype → executes with `this = c1`?

> explain this part with example and in easy way confusion in this...
> line by line explanation jut like to a newbie

Nice — let's go **line-by-line** and make the whole `this = c1` thing obvious with small code examples and plain language. I'll treat the snippet you had and walk through what happens when you run `c1.increment()` — like teaching a beginner.

Code we'll explain

```js
function Counter() {}

Counter.prototype.count = 0;
```

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

let c1 = new Counter();
let c2 = new Counter();
```

Line-by-line (very beginner-friendly)

1. **`function Counter() {}`**
   - You create a **constructor function** named **`Counter`**.
   - Think: this is a blueprint for making Counter objects.
   - At this point it does nothing special — it just exists.
2. **`Counter.prototype.count = 0;`**
   - You add a property **`count`** to the **prototype** of **`Counter`**.
   - Prototype is a shared object that all instances (objects made by **`new Counter()`**) use as a fallback for properties and methods.
   - Right now the *shared* **`count`** value is **`0`**. No individual object has its own **`count`** yet.
3.

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

- You add a function **`increment`** to the prototype.
- That function uses **`this.count++`**. Important: **`this`** is not decided now — it is decided **when the function is called.**
- **`this.count++`** means: "look for **`count`** on the object referred to by **`this`**; if not found, look up the prototype; then increment that value — and if incrementing via **`this`** creates an own property, create it on the object."

4. **`let c1 = new Counter();`**
   - You create a new object **`c1`**.
   - **`c1`** has an internal link to **`Counter.prototype`**.
   - **`c1`** does **not** have its own **`count`** property (yet), but it can see **`Counter.prototype.count`** (which is **`0`**).
5. **`let c2 = new Counter();`**
   - Same as above, but a separate object **`c2`**. It also points to the same **`Counter.prototype`**.

Now the important call: `c1.increment()`

The code you run:

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

Step A — Find the function to run

- JavaScript looks on **`c1`** for **`increment`** → not found.
- Then it looks at **`c1`**'s prototype → finds **`Counter.prototype.increment`**.
- So the function to execute is the one stored on the prototype.

Step B — Decide what `this` should be

- When JS calls a function as **`object.method()`**, **`this` becomes `object`** — the thing before the dot.
- Here the call is **`c1.increment()`**, so **`this = c1`** inside the function.
  - That's what people mean by "**`this`** refers to **`c1`**".

Step C — Run the function body with `this = c1`

Inside **`increment`**:

```js
this.count++;          // becomes c1.count++
console.log(this.count); // prints c1.count
```

- JS checks **`c1.count`** — not found, so it looks at prototype and sees **`Counter.prototype.count === 0`**.
- **`this.count++`** will create an **own property `count`** on **`c1`** and set it to **`1`** (it increments the inherited 0 and stores result on c1).
- **`console.log(this.count)`** prints **`1`**.

After the call:

- **`c1`** now has its own **`count: 1`**.
- **`Counter.prototype.count`** is still **`0`**.
- **`c2`** still sees **`Counter.prototype.count`** as **`0`** (no change).

Short demo showing outputs (step-by-step)

```js
console.log(c1.count); // 0  (reads from prototype)
c1.increment();        // prints 1
console.log(c1.count); // 1  (own property now)
console.log(c2.count); // 0  (still prototype)
c2.increment();        // prints 1
```

```
console.log(c2.count); // 1
console.log(Counter.prototype.count); // 0 (unchanged)
```

Extra examples to clarify different `this` behaviors

1. **Calling via another object**

   ```js
   c2.increment(); // this becomes c2 inside the function
   ```

   Result: increments `c2`'s count, not `c1`'s.

2. **Calling the function without an object**

   ```js
   const f = c1.increment;
   f(); // this is NOT c1 now (in strict mode: undefined; otherwise: global object)
   ```

   - Because you called `f()` directly, there is no object before the dot, so `this` is not set to `c1`. That's why calling methods detached from their object can break `this`.

3. **Forcing `this` with `call`**

   ```js
   c1.increment.call(c2); // runs increment but sets this = c2
   ```

   - `this` becomes `c2` inside the function.

Key takeaways (newbie-friendly)

- **`this` is set when the function is called**, not when it is defined.
- When you call `obj.method()`, inside the method `this` is `obj`.
  So `c1.increment()` → `this` is `c1`.
- Methods can live on the prototype (shared), but when used with `obj.method()` they act **as if they belong to the object** because `this` points to that object.
- If the property being incremented is found only on the prototype, `this.count++` will create an **own property on the object** (shadowing the prototype value).
- If you extract the function (like `const f = c1.increment`) and call it as `f()`, `this` is no longer `c1` — so the code behaves differently.

If you want, I can next:

- show a tiny visual diagram (object → prototype → value), or

- give a live clipboard snippet you can paste and run to observe each step. Which one would help you more?

const f = c1.increment;
f(); // this is NOT c1 now (in strict mode: undefined; otherwise: global object)

Because you called f() directly, there is no object before the dot, so this is not set to c1. That's why calling methods detached from their object can break this.

but in thi s case e are storing the reference that it is called. y c1 or we are storing the function increment reference that's why it is undefined

Excellent question 👏 — and you've caught the **exact subtle point** that many people miss!
Let's unpack it very slowly and clearly like a story.

The code again:

```js
const f = c1.increment;
f(); // ?
```

You're asking:

> "We are storing the reference of the function that is called by c1, right? So shouldn't it remember that it belongs to c1?"

That sounds logical — but **JavaScript doesn't store "who called it" with the function**.
Let's see why 👇

Step 1: What happens on this line

```js
const f = c1.increment;
```

- **`c1.increment`** means "get the value of the **`increment`** property from c1."
- What is that value?
  → It's just a **reference to the function** stored in **`Counter.prototype.increment`**.

👉 JavaScript does **not** store any information about "who" it came from (like **`c1`**).
It simply copies the **function itself**, not the *object-method relationship*.

Think of it like this analogy:

You took a screwdriver (the function) out of a toolbox (c1).
The screwdriver itself doesn't "remember" which toolbox it came from.
You can use it anywhere — but it no longer knows it belonged to **`c1`**.

Step 2: When you call `f()`

```js
f(); // called directly
```

Here's the key rule in JS:

- When you call a function as **`object.method()`**, **`this`** becomes **`object`**.
- When you call just **`functionName()`**, **`this`** depends on the *call site*:
  - In **strict mode**: **`this`** is **`undefined`**
  - In **non-strict mode**: **`this`** is the global object (**`window`** in browser, **`global`** in Node)

Since you wrote **`f()`** (no object before the dot), JavaScript sets:

```js
this = undefined  // in strict mode
```

That's why inside the function, **`this.count`** → error (since **`this`** is undefined).

Step 3: Compare with `c1.increment()`

When you do:

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

JavaScript does this under the hood:

1. Find the function → **`Counter.prototype.increment`**
2. Call it **with `this = c1`**

So it behaves like:

```js
Counter.prototype.increment.call(c1);
```

Step 4: Visual summary

Code	How function is called	Value of this inside
`c1.increment()`	As a method on `c1`	`this = c1`
`const f = c1.increment;` `f();`	As a plain function	`this = undefined` (strict) or `window` (non-strict)
`f.call(c1)`	Forced `this` using `call`	`this = c1`

Step 5: Example to see it in action

Try this in console 👇

```js
"use strict";

function Counter() {}
Counter.prototype.count = 0;
Counter.prototype.increment = function () {
  console.log("this is:", this);
  this.count++;
  console.log(this.count);
};

let c1 = new Counter();
const f = c1.increment;

c1.increment(); // ✅ this = c1
f();            // ❌ this = undefined
f.call(c1);     // ✅ this = c1 (forced)
```

✅ In short (to copy in your VS Code notes):

> When we do `const f = c1.increment;`, we're only storing the **function reference**, not the connection between the function and `c1`.
> So when `f()` is called directly, there's no object before the dot — hence `this` is **undefined** (in strict mode).
> `this` is always decided **at the time of function call**, not at the time of variable assignment.

Would you like me to show a **simple visual diagram** (like boxes for `c1`, `prototype`, and how `this` changes for each call type)?
That'll make the difference between `c1.increment()` and `f()` crystal clear.

and in non strict mode whne window object in that case what happens

Fantastic follow-up, Vipul 👏 — this is exactly the curiosity that builds deep JavaScript understanding.
Let's go step by step again — clearly and visually — to see what happens in **non-strict mode** when `this` becomes the **window** object.

🧩 Code (non-strict mode)

```js
function Counter() {}

Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  console.log("this is:", this);
  this.count++;
  console.log(this.count);
};

let c1 = new Counter();
const f = c1.increment;

f(); // called directly (non-strict mode)
```

🧠 Step-by-step explanation

🔸 Step 1: What happens at `const f = c1.increment;`

- JS looks at `c1.increment`
- Finds it in `Counter.prototype`
- Assigns **that function** to `f`

So now `f` is **just a plain function variable**.
There's no link to `c1` anymore.

🔸 Step 2: What happens at `f();`

- JS sees it's called **directly** (no object before the dot).
- Since we're **not in strict mode**, JS defaults:

  ```js
  this = window;
  ```

So inside the function,

```js
this.count++;
```

actually means:

```js
window.count++;
```

🔄 Step 3: What does that do?

If your browser had no **`window.count`** before,
JS first creates it:

```js
window.count = 0;
```

Then increments it:

```js
window.count = 1;
```

So, after calling **`f()`**,
`window.count` becomes **`1`**.

🧾 Output in console

```js
this is: Window { ... }   // logs the global window object
1                         // logs window.count (1)
```

✅ And if you check:

```js
console.log(window.count); // 1
console.log(c1.count);     // undefined
```

You'll see **`window.count`** changed, not **`c1.count`**.

💡 Simple analogy

Think of **strict mode** like "no auto-assumptions mode."
If you forget to say *who* is calling, JS won't guess — it gives **`this = undefined`**.

But in **non-strict mode**, JS is "too helpful":
it assumes the global object (**`window`** in browsers, **`global`** in Node.js) is the caller.

🧠 Summary Table

Call Type	Mode	Value of `this`	Affected `count`
`c1.increment()`	Any	**`c1`**	**`c1.count`**
`f()`	Strict	**`undefined`**	❌ Error (can't read property 'count' of undefined)
`f()`	Non-strict	**`window`**	**`window.count`**

Call Type	Mode	Value of `this`	Affected `count`
`f.call(c1)`	Any	`c1`	`c1.count`

✅ To remember in your notes:

> In non-strict mode, if a function is called directly (not as an object method),
> JavaScript automatically sets `this` to the **global object (`window`)**.
> So `this.count++` modifies a global variable instead of your object's property — which can cause unexpected bugs.

Would you like me to show a small **browser console demo** you can copy-paste to see how `window.count` vs `c1.count` behave live?

like this also write both case in one for notes
In non-strict mode, if a function is called directly (not as an object method),
JavaScript automatically sets this to the global object (window).
So this.count++ modifies a global variable instead of your object's property — which can cause unexpected bugs.

Perfect 👍 Vipul!
Here's a **clean, ready-to-copy** version for your VS Code notes — showing **both strict and non-strict mode cases** side by side 👇

🔍 `this` Behavior — Strict Mode vs Non-Strict Mode

```js
function Counter() {}

Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  console.log("this is:", this);
  this.count++;
  console.log("count:", this.count);
};

let c1 = new Counter();
const f = c1.increment;

// ---------- Case 1: Strict Mode ----------
"use strict";
f();
// this = undefined
// ❌ Error: Cannot read properties of undefined (reading 'count')
// Because in strict mode, JS does NOT default to any object
// When a function is called directly (not as an object method),
// 'this' becomes undefined.

// ---------- Case 2: Non-Strict Mode ----------
// Remove or comment out "use strict"
f();
// this = window (global object)
```

```
// ✅ Works, but modifies window.count instead of c1.count
// this.count++ actually becomes window.count++
// Can create accidental global variables → BAD PRACTICE
```

🧠 Summary Notes (for quick recall)

Mode	Function Call Type	this value	What actually changes	Safe?
Strict	`c1.increment()`	`c1`	`c1.count`	✅
Strict	`f()`	`undefined`	❌ Error	⚠️
Non-Strict	`c1.increment()`	`c1`	`c1.count`	✅
Non-Strict	`f()`	`window`	`window.count`	❌ (global pollution)

🧩 In simple words

> In **strict mode**, JavaScript plays safe — if you forget to call through an object, **`this`** is **`undefined`**.
> In **non-strict mode**, JavaScript assumes you meant the **global object (`window`)**, and silently adds or changes global variables — which is risky.

Would you like me to also include a short **diagram** (how **`this`** points to **`c1`** vs **`window`** vs **`undefined`**) below this for your notes? It'll make the concept even more visual.

function Counter() {}

Counter.prototype.count = 0;

Counter.prototype.increment = function () {
 console.log("this is:", this);
 this.count++;
 console.log(this.count);
};

let c1 = new Counter();
const f = c1.increment;

console.log("newFunction-1->", f)
console.log("newFunction-2->", f())

f();

in my vs code i have coded and ran it on browser giving out put:

newFunction-1-> ƒ () {

console.log("this is:", this);
 this.count++;
 console.log(this.count);
}
this is: Window {window: Window, self: Window, document: document, name: '', location: Location, …}
NaN
 newFunction-2-> undefined
 this is: Window {window: Window, self: Window, document: document, name: '', location: Location, …}
 NaN

why undefined and nan while this is pointing to window object

Excellent observation, Vipul 👏 — this is a **perfect** beginner-level confusion that shows you're actually understanding how JS executes line by line.

Let's break your example **step by step** (super clearly) 👇

🧩 Your code

```js
function Counter() {}

Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  console.log("this is:", this);
  this.count++;
  console.log(this.count);
};

let c1 = new Counter();
const f = c1.increment;

console.log("newFunction-1->", f);
console.log("newFunction-2->", f());

f();
```

and your browser output:

```kotlin
newFunction-1-> ƒ () { console.log("this is:", this); this.count++; console.log(this.count); }
this is: Window { ... }
NaN
newFunction-2-> undefined
this is: Window { ... }
NaN
```

⚙️ Let's understand line by line

🔸 Line 1:

```js
console.log("newFunction-1->", f);
```

✅ You are **printing the function itself**, not calling it.
So it prints the actual **function definition**:

```kotlin
ƒ () {
  console.log("this is:", this);
  this.count++;
  console.log(this.count);
}
```

✅ `f` is just a reference to the `increment` function, nothing more.

🔹 Line 2:

```js
console.log("newFunction-2->", f());
```

⚠️ This is **actually calling** the function `f()` — not just printing it.

So the steps here are:

1️⃣ Call `f()`
Since there is **no object before the dot**, `this = window` (because non-strict mode).

2️⃣ Inside the function:

```js
console.log("this is:", this); // prints "this is: Window { ... }"
this.count++;                  // means window.count++
```

But wait — what is `window.count` right now?
It doesn't exist yet.

So JS tries:

```javascript
undefined + 1 = NaN
```

That's why it prints:

```r
NaN
```

3️⃣ The function doesn't `return` anything, so by default it returns `undefined`.

Hence the `console.log` line prints:

```javascript
newFunction-2-> undefined
```

✅ Because that's the **return value** of **`f()`**.

 Line 3:

```js
f();
```

This again calls the same function.
Now **`window.count`** already exists (but has the value **`NaN`**),
and doing **`NaN + 1`** again still gives **`NaN`**.

So again:

```kotlin
this is: Window { ... }
NaN
```

💡 Why exactly NaN?

Because:

1. **`this = window`**
2. **`window.count`** doesn't exist → **`undefined`**
3. **`undefined + 1`** → **`NaN`**
4. JS saves **`window.count = NaN`**

So **`window.count`** becomes **`NaN`**,
and stays **`NaN`** even after further increments.

🧠 Why `undefined` after `f()`?

Because your **`increment`** function doesn't return anything explicitly.
All functions in JS **return `undefined` by default** if no **`return`** is written.

✅ In short (for your notes)

```js
console.log(f);     // prints the function itself
console.log(f());   // calls it → executes → returns undefined

// Inside f():
// "this" → window (in non-strict mode)
```

```
// window.count starts as undefined → undefined + 1 = NaN
// So prints: "NaN"
```

🧾 Output summary table

Line	What happens	Output
`console.log(f)`	Prints the function itself	`ƒ () { ... }`
`console.log(f())`	Calls the function, returns undefined	`undefined`
Inside `f()`	`this = window`, so modifies `window.count`	`NaN` printed

If you now add:

```js
window.count = 0;
f(); // this time prints 1 ✅
```

Then it will correctly print:

```kotlin
this is: Window { ... }
1
```

Would you like me to show how this same code behaves **differently in strict mode**, side-by-side? (You'll see why `undefined` happens instead of `window` then.)

but above at
Step 2: What happens at f();

you said and step 3:
If your browser had no window.count before,
JS first creates it
window.count = 0;
then increment it
window.count = 1;

💯 Excellent catch, Vipul — this shows you're paying **very close attention**, which is exactly how you master JS.

You're absolutely right to ask:

> "Earlier you said that when JS finds no `window.count`, it first creates it with 0 and then increments — but now it's showing `NaN`. So which one is correct?"

Let's clear this **confusion properly**, step by step. 👇

🧩 The truth (line by line)

When you do this in **non-strict mode**:

```js
function Counter() {}

Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  console.log("this is:", this);
  this.count++;
  console.log(this.count);
};

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

□ Step 1: `this = window`

Because we called **`f()` directly**, **`this`** becomes **`window`** (in non-strict mode).

So this line:

```js
this.count++;
```

actually means:

```js
window.count++;
```

□ Step 2: Does `window.count` exist yet?

Here's the key:

- When you **read** a property that doesn't exist, like **`window.count`**, JS gives you **`undefined`**.
- Then when you do **`undefined + 1`**, JS gives you → **NaN**.

✅ **JavaScript does *not* automatically create `window.count = 0`** before incrementing.
It simply takes **`undefined`**, adds 1, and the result is **`NaN`**.

□ Step 3: What happens next

After doing **`undefined + 1`**,
JS assigns the result (**`NaN`**) back to **`window.count`**.

So now:

```js
window.count = NaN;
```

And next time you call **`f()`**, it does:

```js
NaN + 1 = NaN
```

So it stays **`NaN`**.

✅ Corrected explanation (for your notes)

Earlier simplification "JS first creates it with 0" was to give you an *intuitive* sense that JS adds the property to **`window`**.
But the **accurate technical behavior** is:

1️⃣ JS checks if **`window.count`** exists.
2️⃣ If not, it returns **`undefined`**.
3️⃣ Then it computes **`undefined + 1`** → **`NaN`**.
4️⃣ Then assigns **`NaN`** back to **`window.count`**.

✅ So **`window.count`** **does get created** —
but with the value **NaN**, not 0.

🧠 Quick proof

Try this in browser console:

```js
delete window.count; // make sure it doesn't exist
console.log(window.count); // undefined
window.count++;
console.log(window.count); // NaN
```

See?
It doesn't start at 0 — it becomes NaN after the first **`++`**.

🧩 So the two truths together:

Step	What happens	Result
You call **`f()`**	**`this = window`**	✅
JS looks for **`window.count`**	Not found → **`undefined`**	⚠️
`undefined + 1`	**`NaN`**	❌

Step	What happens	Result
JS assigns back	**`window.count = NaN`**	✅ property created but invalid value

🧾 Correct short note to keep in VS Code:

```js
// In non-strict mode:
this = window;
this.count++  // means window.count++
If window.count doesn't exist:
  JS reads it as undefined,
  does undefined + 1 = NaN,
  and assigns window.count = NaN.
```

Would you like me to show how this behaves if you **initialize window.count first**, and then call **`f()`** — so you can see the difference (NaN → 1 → 2)?

YES

Perfect, Vipul 🙌 — let's go step-by-step again, but this time we'll **set up `window.count` first** to see exactly how behavior changes.

We'll work in **non-strict mode** (so **`this = window`** when calling **`f()`** directly).

🧩 Full Code Example

```js
function Counter() {}

Counter.prototype.count = 0;

Counter.prototype.increment = function () {
  console.log("this is:", this);
  this.count++;
  console.log("count is now:", this.count);
};

let c1 = new Counter();
const f = c1.increment;

// 🟢 Step 1: initialize window.count manually
window.count = 0;
console.log("Initially:", window.count);

// 🟢 Step 2: call f() multiple times
f(); // call 1
f(); // call 2
f(); // call 3

// 🟢 Step 3: check both counts
console.log("Final window.count:", window.count);
console.log("Final c1.count:", c1.count);
```

🧠 Step-by-step Explanation

Step 1️⃣

```js
window.count = 0;
```

Now the global object has a **real property `count`** starting at 0.

```javascript
window = {
  count: 0,
  ...
}
```

Step 2️⃣

```js
f();
```

Since you're calling **`f()`** directly →
👉 **`this = window`** (in non-strict mode).

Inside the function:

```js
this.count++; // means window.count++
```

So after each call:

```pgsql
Call 1 → window.count = 1
Call 2 → window.count = 2
Call 3 → window.count = 3
```

✅ No more **`NaN`** because the property already existed with a numeric value (0).

Step 3️⃣

```js
console.log("Final window.count:", window.count);
console.log("Final c1.count:", c1.count);
```

Output:

```javascript
Final window.count: 3
Final c1.count: undefined
```

👉 Because all increments happened on the **global object (`window`)**, not on `c1`.

🧾 Output Summary

```kotlin
Initially: 0
this is: Window { ... }
count is now: 1
this is: Window { ... }
count is now: 2
this is: Window { ... }
count is now: 3
Final window.count: 3
Final c1.count: undefined
```

✅ For your Notes (clear version to copy)

```js
// Non-strict mode example: `this` becomes window when function called directly.

function Counter() {}
Counter.prototype.count = 0;
Counter.prototype.increment = function () {
  console.log("this is:", this);
  this.count++;
  console.log("count is now:", this.count);
};

let c1 = new Counter();
const f = c1.increment;

window.count = 0; // initialize first!

f(); // window.count → 1
f(); // window.count → 2
f(); // window.count → 3

console.log("window.count =", window.count); // 3
console.log("c1.count =", c1.count);         // undefined
```

🧠 Quick takeaway:

> If **`window.count`** (or **`this.count`** when **`this = window`**) already exists and is numeric, the **`++`** operator increases it correctly.
>
> If it doesn't exist, JS reads it as **`undefined`**, and **`undefined + 1 = NaN`**.

Would you like me to also show the **strict mode version** side-by-side with this (so you can visually compare how both behave and why one gives **`undefined`** error)?

when we call like this:
function Counter() {}
Counter.prototype.count = 0;

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

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

in this case it attaches coun to c1 object and initalize it with 0 and than increases one.
but in case of when it points to window object it is nan and it is attaching undefined first than nan but in this why different

Excellent question, Vipul 👏 — you've noticed a **very subtle but super important difference** between how **`this.count`** behaves when **`this`** is an *object* (**`c1`**) vs when **`this`** is the *global object (**`window`**)*.

Let's go **step by step** 👇

✅ Case 1: When called as `c1.increment()`

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

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

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

Step-by-step:

1. **`c1`** is created using **`new Counter()`**.
2. **`Counter.prototype.count = 0;`** → means **all Counter objects share** a property **`count`** on the prototype (not directly inside each object yet).
3. When we call **`c1.increment()`**,
   - **`this`** → refers to **c1**.
4. JS looks for **`c1.count`**.
   - It doesn't find it **on c1**,
   - So it looks **up the prototype chain** and finds **`count = 0`** on **`Counter.prototype`**.
5. Then it does:

```js
this.count++;  // means c1.count = c1.count + 1
```

So it **copies that value (0)**, adds 1, and **creates a new `count` property directly on c1** (not prototype now).

6. ✅ Final result:
   - **`c1.count = 1`**
   - **`Counter.prototype.count = 0`** (unchanged)

❌ Case 2: When called like this:

```js
const f = c1.increment;
f();
```

Step-by-step:

1. **`f`** is just a **reference** to the function **`increment`**.
2. When you call **`f()`** directly (without any object before the dot),
   - In **non-strict mode**, **`this`** becomes the **global object** → **`window`**.
   - So inside that function, **`this`** → **`window`**.
3. JS tries to do:

   ```js
   this.count++;  // means window.count++
   ```

4. But initially, there is **no `window.count` variable**.
5. So JS tries to **read** it first:
   `window.count` → **`undefined`**
6. Now it tries to add **`1`**:
   `undefined + 1` → results in **NaN**.
7. Then JS sets:
   `window.count = NaN`
8. That's why:
   - First time → **`NaN`**
   - Next time → **`NaN + 1`** → still **`NaN`**

🧠 So Why the Difference?

Case	`this` refers to	`count` initially	First increment result	Reason
`c1.increment()`	`c1`	found on prototype (`0`)	`1`	value 0 copied and incremented
`f()`	`window`	`undefined`	`NaN`	no existing count property, undefined + 1 = NaN

✅ Summary Note (for your VS Code file)

```js
// When called as c1.increment(), this -> c1
// JS finds count=0 on prototype, copies it, creates c1.count=1

// When called as f(), this -> window (in non-strict mode)
// window.count is undefined, so undefined + 1 = NaN
// then window.count = NaN
```

Would you like me to show you a quick **console log version** that proves step-by-step that **`window.count`** starts as undefined and becomes NaN?