

# Code explanation step-by-step



```

var vipul = 30;
let malik = 40;
function counter () {
    var count = 0;
    var count1 = 20;
    return function incrementCounter () {
        var countersDummy1 = 1000;
        let countersDummy2 = 2000;
        count++;
        console.log(count);
    }
}

var counter1 = counter();
counter1();
counter1();

counter.count1;
counter1.count1;

var counter2 = counter();
counter2();
counter2();
output

```

Let's go step by step carefully 🤟

## Code:

js

```

var vipul = 30;
let malik = 40;
function counter () {
    var count = 0;
    var count1 = 20;
    return function incrementCounter () {
        var countersDummy1 = 1000;
        let countersDummy2 = 2000;
        count++;
        console.log(count);
    }
}

var counter1 = counter();
counter1();
counter1();

counter.count1;

```

```
counter1.count1;
var counter2 = counter();
counter2();
counter2();
```

## Step-by-step explanation

### Step 1: var vipul = 30;

→ Creates a global variable **vipul** with value **30**.

### Step 2: let malik = 40;

→ Creates a block-scoped variable **malik** with value **40**.

No output yet.

---

### Step 3: Function declaration function counter() {...}

This defines a function named **counter**.

It has **two local variables** inside:

- **count** = 0
- **count1** = 20

And it returns another inner function **incrementCounter()**.

---

### Step 4: var counter1 = counter();

- When **counter()** is called:
  - A **new execution context** is created.
  - **count = 0, count1 = 20**
  - It returns the **inner function incrementCounter**.

So now **counter1** refers to that **inner function**, and it **has a closure** over **count** and **count1**.

---

### Step 5: counter1();

When you call it:

- It runs the inner function.
- **count++ → count = 1**
- **console.log(count) → prints 1**

---

### Step 6: counter1(); again

- `count` is still remembered from the closure (was 1 before)
- `count++ → count = 2`
- `console.log(count) → prints 2`

 Output so far:

```
1
2
```

**Step 7:** `counter.count1;`

- Here you are trying to access the property `count1` of the **function object** `counter`.
- But `count1` is a **local variable** inside the function — not attached as a property.

So this returns `undefined`.

**Step 8:** `counter1.count1;`

- `counter1` is also a function (the returned inner function).
- Again, `count1` is not a property on it, it's just in its **closure environment**, not on the function object itself.

So this also returns `undefined`.

**Step 9:** `var counter2 = counter();`

- Creates a new closure entirely.
- New `count = 0`, new `count1 = 20`

**Step 10:** `counter2();`

- `count++ → 1`
- Prints **1**

**Step 11:** `counter2();`

- `count++ → 2`
- Prints **2**

 Additional output:

```
1
2
```



## Final Console Output:

```
1
2
1
2
```

## And expression results:

Expression	Value
counter.count1	undefined
counter1.count1	undefined

## Final Answer

### Output in console:

```
1
2
1
2
```

## Values of expressions:

```
js

counter.count1    // undefined
counter1.count1   // undefined
```



you are saying that count1 is a local variable inside the function — not attached as a property.  
tha how can we make it a property

Excellent question 👏 Vipul!

Let's break this down clearly and practically 👏

## What's happening now

In your code:

```
js

function counter() {
  var count = 0;
  var count1 = 20;
  return function incrementCounter() {
    count++;
```

```

        console.log(count);
    };
}

```

Here, both `count` and `count1` are **local variables** — they live **inside the function's scope**, not attached to the **function object**.

So, when you do this:

```

js

counter.count1

```

You get `undefined`, because `count1` is not stored *on* the function object — it's stored *inside* its execution context.

 **To make it a property of the function, you must explicitly attach it to the function object.**

Here's how 

### ✓ Option 1: Attach property directly to the function

```

js

function counter() {
    var count = 0;

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

    // attach a property directly to the returned function
    incrementCounter.count1 = 20;

    return incrementCounter;
}

var counter1 = counter();

console.log(counter1.count1); // ✓ 20
counter1(); // 1
counter1(); // 2

```

 Now `count1` is an actual **property** on the function object `counter1`.

### ✓ Option 2: Attach property to `this` if called with `new`

If you call the function as a **constructor** using `new`, you can make properties available on the instance using `this`:

```

js

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

```

```

this.count1 = 20;

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

const counter1 = new Counter();

console.log(counter1.count1); // ✓ 20
counter1.increment(); // 1
counter1.increment(); // 2

```

Here, `count1` is an **instance property** instead of a local variable.

## ✓ Option 3: Attach property to the function after defining it

You can even assign it **outside** the function:

```

js

function counter() {
  var count = 0;
  return function() {
    count++;
    console.log(count);
  };
}

// add property later
counter.count1 = 20;

console.log(counter.count1); // ✓ 20

```

But note — this attaches `count1` to the **outer function `counter`**, not to its returned inner function.

## 🧭 Summary

Goal	Where to put it	Example
Make <code>count1</code> a property of returned inner function	Inside function: <code>incrementCounter.count1 = 20;</code>	✓ Best for closures
Make <code>count1</code> a property of instances created via <code>new</code>	Use <code>this.count1 = 20;</code>	✓ Best for OOP style
Make <code>count1</code> a property of the outer function itself	<code>counter.count1 = 20;</code> after defining	✓ Works, but not closure-related

Would you like me to show how `count1` behaves differently in **closure** vs **function property** side-by-side (with console outputs)?



is we do this than it is undefined why:  
`function Counter() {`

```

this.count = 0;
this.count1 = 20;

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

const counter1 = new Counter();

console.log(Counter.count1);
console.log(Counter().count1);

if we are doing ounter

```

Perfect Vipul 🤘

You caught one of the most **important and tricky** concepts in JavaScript — the difference between a **constructor function** and a **normal function call**, and between **function properties** and **instance properties**.

Let's carefully go line by line ⏵

## Your Code

```

js

function Counter() {
  this.count = 0;
  this.count1 = 20;

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

const counter1 = new Counter();

console.log(Counter.count1);
console.log(Counter().count1);

```

### Step 1: When you do const counter1 = new Counter();

- Using `new` creates a **new object** — `counter1`.
- Inside the constructor:
  - `this` points to that **new object**.
- So:

```
js

counter1.count = 0;
counter1.count1 = 20;
counter1.increment = function() {...}
```

Now `counter1` has those properties:

```
js

{
  count: 0,
  count1: 20,
  increment: f()
}
```

## Step 2: `console.log(Counter.count1);`

- `Counter` here is just the **function itself** — not the object created by it.
- You never attached `count1` as a property on the function.
- You attached it to `this` (the instance).

So:

```
js

Counter.count1 // ❌ undefined
```

`count1` exists on `counter1`, not on `Counter`.

## Step 3: `console.log(Counter().count1);`

This line causes a **different issue** 😬

When you call `Counter()` **without** `new`,  
then:

- `this` inside the function does **not** point to a new object.
- In **non-strict mode**, `this` points to the **global object** (`window` in browser, `global` in Node).
- In **strict mode**, `this` becomes **undefined**, and the code will throw an error:

`TypeError: Cannot set property 'count' of undefined`

Even if you're not in strict mode,

`Counter()` returns **nothing** (no `return`), so its value is **undefined**.

So:

```
js

Counter().count1; // ❌ TypeError (because Counter() returns undefined)
```

## ✓ Correct Ways

### 1 Access via instance:

```
js
const counter1 = new Counter();
console.log(counter1.count1); // ✓ 20
```

### 2 If you want to make it property of the function itself:

```
js
Counter.count1 = 20;
console.log(Counter.count1); // ✓ 20
```

## 💡 Summary

Expression	Explanation	Output
<code>Counter.count1</code>	Property not defined on function	<code>undefined</code>
<code>Counter().count1</code>	<code>Counter()</code> returns undefined (no return)	<code>✗ Error</code>
<code>counter1.count1</code>	Instance property created using <code>new</code>	<code>✓ 20</code>

Would you like me to explain *why `new` changes the meaning of `this`* behind the scenes (how JS creates the instance step-by-step)?

That's a beautiful concept to visualize.