-
```
function x(){
  var i=1;
  setTimeout(function(){
    console.log(i);
  },3000);
  console.log("Heyy");
}
x();
```

this function forms a closure, remembers its reference to i.

setTimeout – it takes the call back function and stores it into someplace and attaches the timer to it and then javascript proceeds.

Once the timer expires, it takes the function, puts it on the call stack and it runs it.

①
```
function x(){
  for(var i=1; i<=5; i++){
    setTimeout(function(){
      console.log(i);
    }, i*1000);
  }
  console.log("Nomast");
}
x();
```

>> Nomaste
6 6 6 6 6

(i) When the loop runs for the first time it keeps a copy of the function, attaches the timer also remembers the reference of i.

(ii) All the 5 copies of the function, points to the same reference of i.

(iii) When the timer expires, i value becomes 6. So it prints 6 five times as all five copy of functions are pointing to exact same spot in memory of i.

② Correction with let

```
function x(){
  for(let i=1; i<=5; i++){
    setTimeout(function(){
      console.log(i);
    }, i*1000
    );
  }
  console.log("Namaste");
}
x();
```

(i) var is function-scoped (one shared variable i across all iterations).

(ii) let is block-scoped (a new i is created for each iteration of the loop).

(iii) Each setTimeout callback closes over its own copy of i.

(iv) So, when the timer runs later, it still remembers the value of i at the time that iteration happened.

(v) After the loop ends, console.log("Namaste") executes immediately.

(vi) Then the event loop executes each timer callback at their delays, printing 1,2,3,4,5.

2. Block Scope (like `let` and `const`)

- A variable declared with `let` or `const` is **block-scoped**.
- That means: it is only accessible **inside the nearest `{ }` block** where it is defined.
- It **respects block boundaries**.

Example: Block scope with `let`

js Copy code

```
function test() {
    if (true) {
        let y = 20;
    }
    console.log(y); // ❌ ReferenceError: y is not defined
}
test();
```

Here, `y` is not accessible outside the `if { }` block.

1. Function Scope (like `var`)

- A variable declared with `var` is **function-scoped**.
- That means: once declared inside a function, it is accessible **everywhere inside that function**, no matter which block (`if`, `for`, etc.) it's in.
- It **ignores block boundaries** (like `{ }`).

Example: Function scope with `var`

js Copy code

```
function test() {
    if (true) {
        var x = 10;
    }
    console.log(x); // ✅ works, prints 10
}
test();
```

Even though `x` was declared inside the `if { }` block, it is accessible outside, because `var` is **scoped to the whole function**, not the block.

③ Correction with var

```
function x(){
    for(var i=1; i<=5; i++){
        function close(x){
            setTimeout(function(){
                console.log(x)
                }; x*1000;
            }
        close(i);
    }
    console.log("Nomasti");
}
x();
```

(i) x() is called → execution enters the function.

(ii) The for loop runs with var i (function scoped).
- But now, instead of using i directly in setTimeout, we pass i as an argument to close(i).

(iii) Iteration 1 (i=1)
- close(1) is called
- Inside close, a new parameter x is created with value 1.
- setTimeout schedules a callback after 1000ms, and that callback closes over x=1.
- Now even if the loop continues, this callback remembers x=1.

..... so on upto iteration 5.

(iv) After the loop finishes, console.log("Nomasti") runs immediately.

(v) Then as timer expires,
After 1s → callback from iteration 1 runs, prints 1.
After 2s → " " " 2 ", " 2
⋮ upto 5.