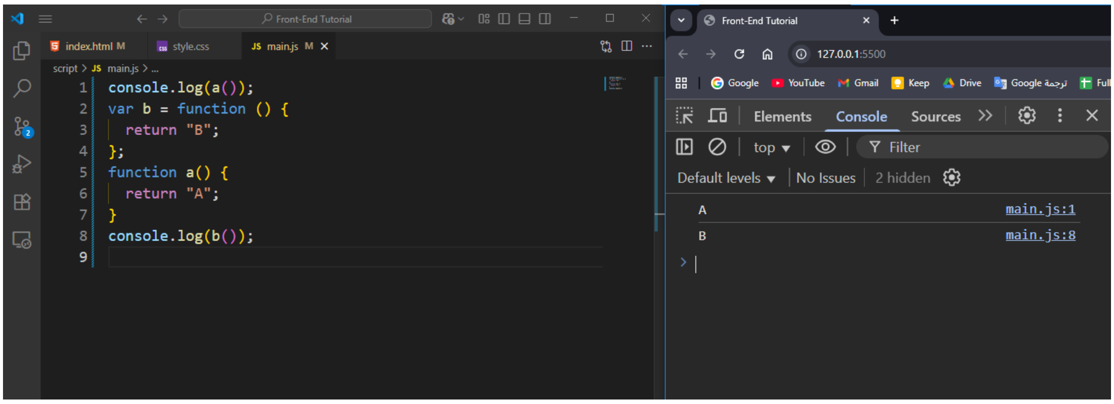
JavaScript Assignments Day-5

1. Predict (in comments) the output order of this code, then run to verify.

```
console.log(a());
var b = function(){ return 'B'; };
function a(){ return 'A'; }
console.log(b());
```

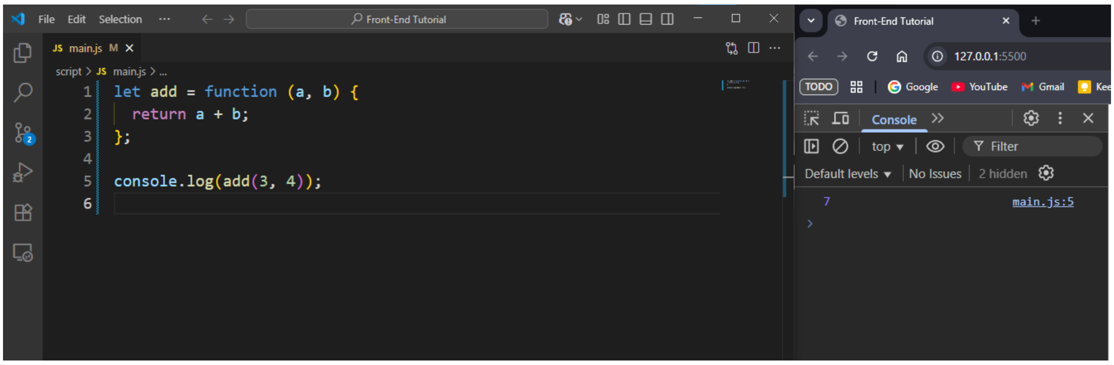
After verifying, explain (one short line) why a works before definition and b does not.
>> Because of Hoisting (The Functions Are Hoisted), The Function Runs Fine.



```
console.log(a());
var b = function () {
  return "B";
};
function a() {
  return "A";
}
console.log(b());
```

```
A                                        main.js:1
B                                        main.js:8
```

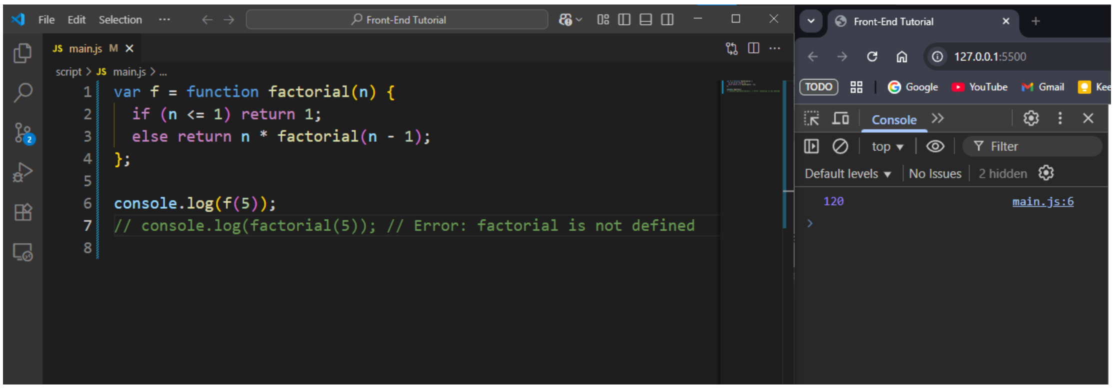
2. Rewrite a function declaration sum(a,b) into a function expression stored in a variable named add and confirm both produce same result for (3,4).



```
let add = function (a, b) {
  return a + b;
};

console.log(add(3, 4));
```

```
7                                        main.js:5
```

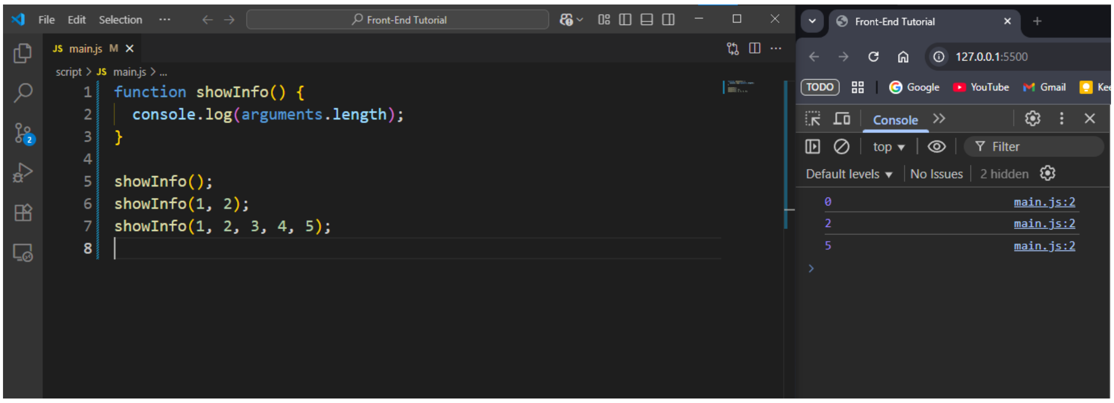
3. Create a named function expression assigned to var factorial. Use the internal name ONLY for recursion. Log factorial(5). Show (comment) that the internal name is not global.



```javascript
var f = function factorial(n) {
  if (n <= 1) return 1;
  else return n * factorial(n - 1);
};

console.log(f(5));
// console.log(factorial(5)); // Error: factorial is not defined
```

```
120                       main.js:6
```

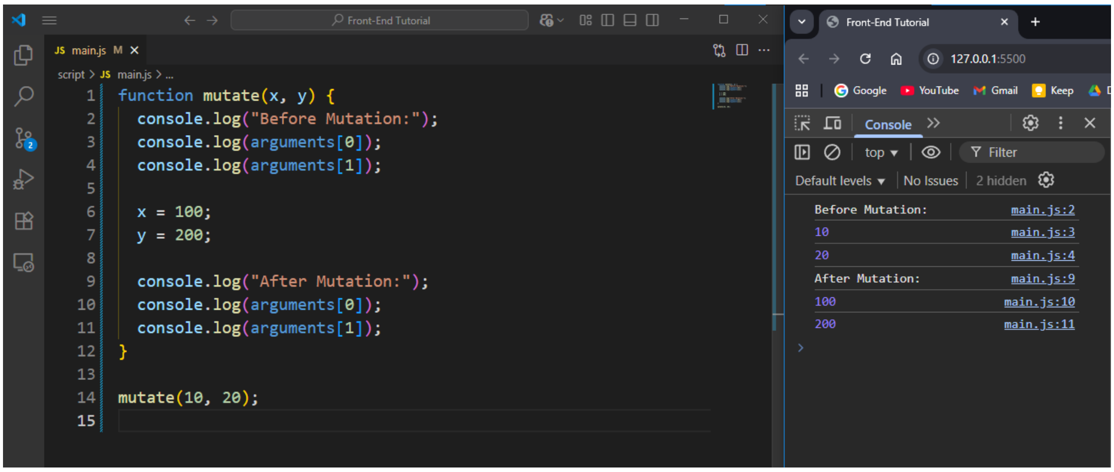
4. Write a function showInfo that logs arguments.length and each argument. Call it with 0, then 2, then 5 arguments.



```javascript
function showInfo() {
  console.log(arguments.length);
}

showInfo();
showInfo(1, 2);
showInfo(1, 2, 3, 4, 5);
```

```
0                         main.js:2
2                         main.js:2
5                         main.js:2
```

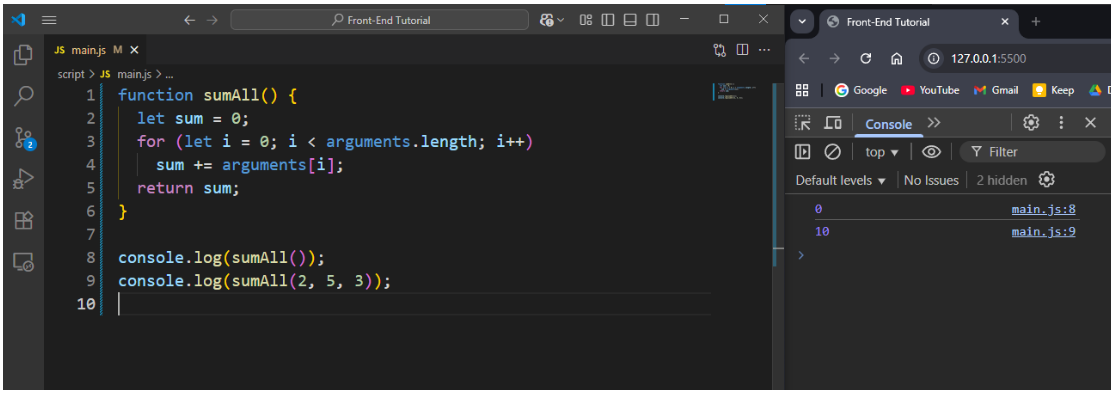
5. Write a function mutate(x,y) that changes x and y inside and shows arguments[0] / arguments[1] before and after. Explain result in a comment.



```javascript
function mutate(x, y) {
  console.log("Before Mutation:");
  console.log(arguments[0]);
  console.log(arguments[1]);

  x = 100;
  y = 200;

  console.log("After Mutation:");
  console.log(arguments[0]);
  console.log(arguments[1]);
}

mutate(10, 20);
```

```
Before Mutation:          main.js:2
10                        main.js:3
20                        main.js:4
After Mutation:           main.js:9
100                       main.js:10
200                       main.js:11
```

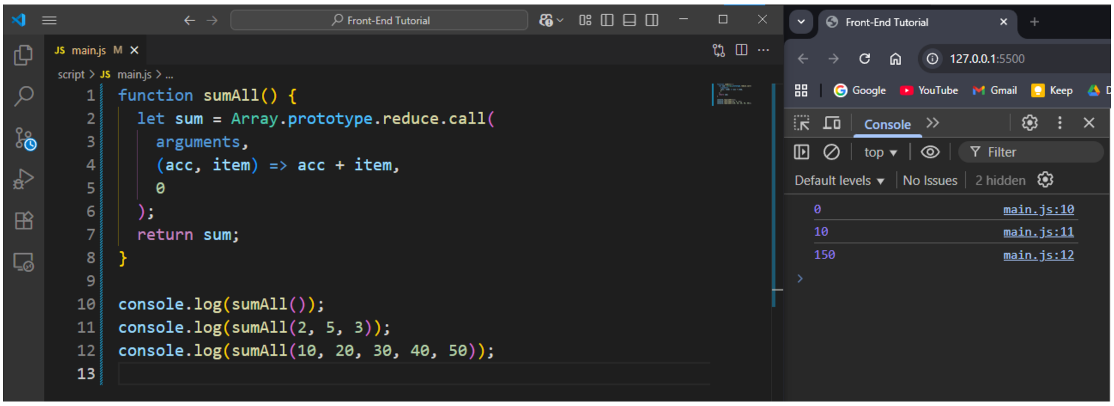
6. Implement sumAll() using only the arguments object (no arrays) to total all numeric arguments. Test sumAll(2,5,3) and sumAll().



```javascript
function sumAll() {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++)
    sum += arguments[i];
  return sum;
}

console.log(sumAll());
console.log(sumAll(2, 5, 3));
```

```
0                                   main.js:8
10                                  main.js:9
```

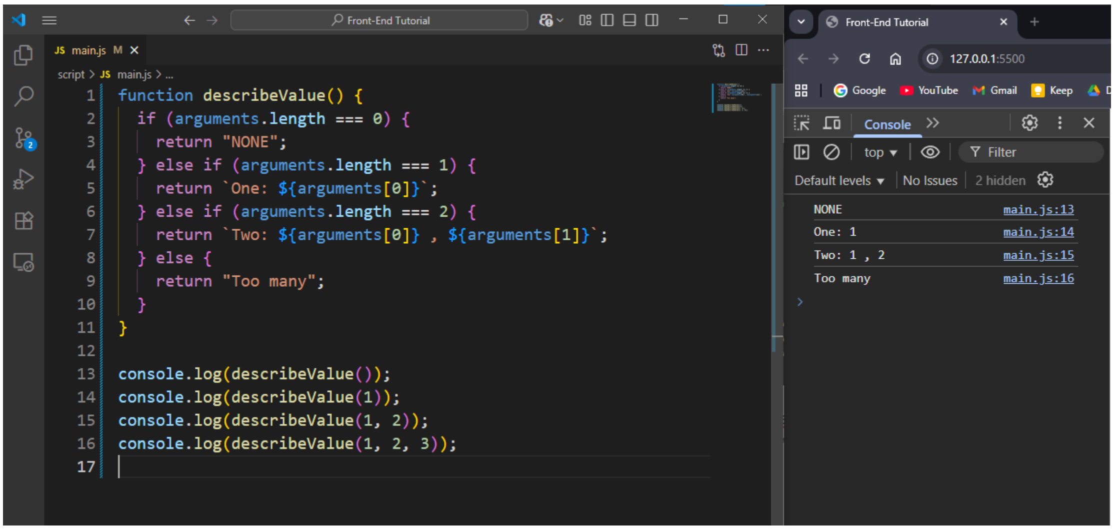
7. Implement sumAll() using only the arguments object but with the Array method reduce.



```javascript
function sumAll() {
  let sum = Array.prototype.reduce.call(
    arguments,
    (acc, item) => acc + item,
    0
  );
  return sum;
}

console.log(sumAll());
console.log(sumAll(2, 5, 3));
console.log(sumAll(10, 20, 30, 40, 50));
```

```
0                                   main.js:10
10                                  main.js:11
150                                 main.js:12
```

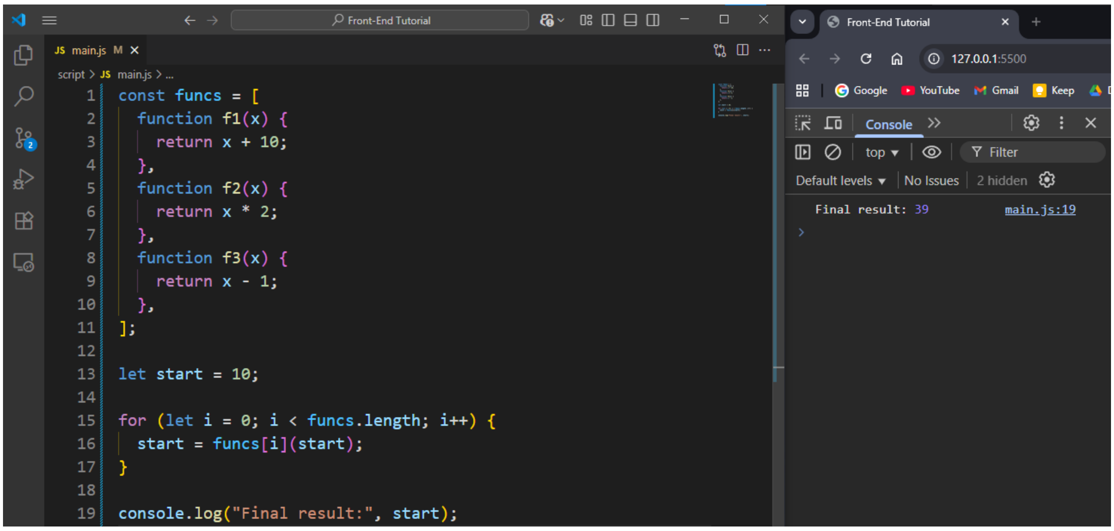
8. Write describeValue that returns different strings based on number of args: 0 -> 'none', 1 -> 'one:'+val, 2 -> 'two:'+a+','+b else 'too many'.



```javascript
function describeValue() {
  if (arguments.length === 0) {
    return "NONE";
  } else if (arguments.length === 1) {
    return `One: ${arguments[0]}`;
  } else if (arguments.length === 2) {
    return `Two: ${arguments[0]} , ${arguments[1]}`;
  } else {
    return "Too many";
  }
}

console.log(describeValue());
console.log(describeValue(1));
console.log(describeValue(1, 2));
console.log(describeValue(1, 2, 3));
```

```
NONE                                main.js:13
One: 1                              main.js:14
Two: 1 , 2                          main.js:15
Too many                            main.js:16
```

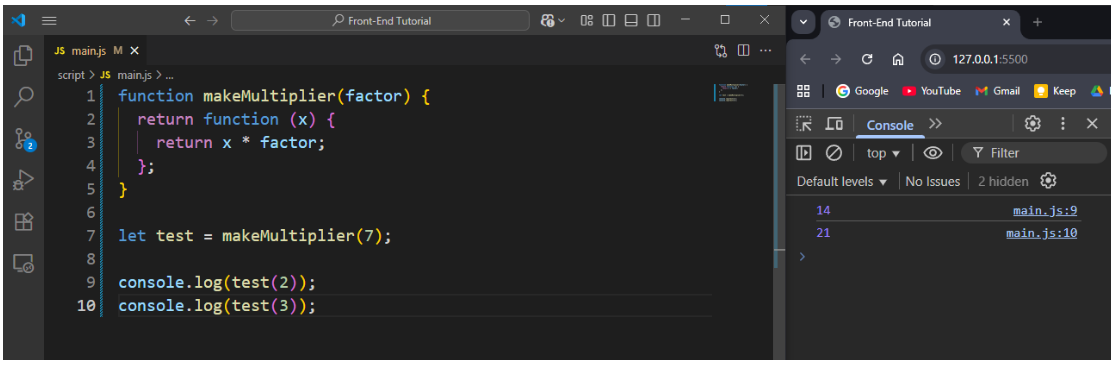
9. Create an array funcs of three small anonymous functions that transform a number. Apply them in order to start = 10 (loop). Log final result.



```
const funcs = [
  function f1(x) {
    return x + 10;
  },
  function f2(x) {
    return x * 2;
  },
  function f3(x) {
    return x - 1;
  },
];

let start = 10;

for (let i = 0; i < funcs.length; i++) {
  start = funcs[i](start);
}

console.log("Final result:", start);
```

```
Final result: 39          main.js:19
```

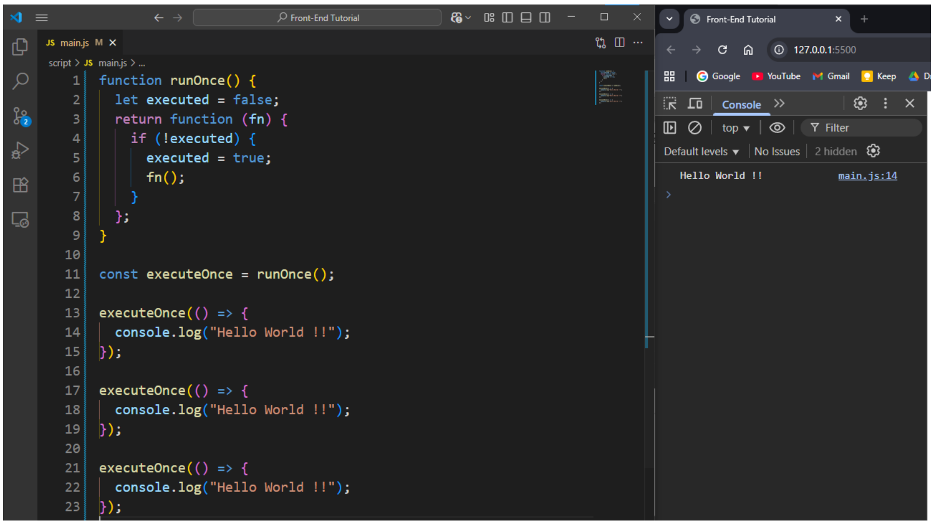
10. Write makeMultiplier(factor) returning a function(n) that multiplies. Create double and triple; test with 7.



```
function makeMultiplier(factor) {
  return function (x) {
    return x * factor;
  };
}

let test = makeMultiplier(7);

console.log(test(2));
console.log(test(3));
```

```
14          main.js:9
21          main.js:10
```

11. Implement once(fn) runs fn only first time, returns its return value. Test with a function that logs and returns a string.

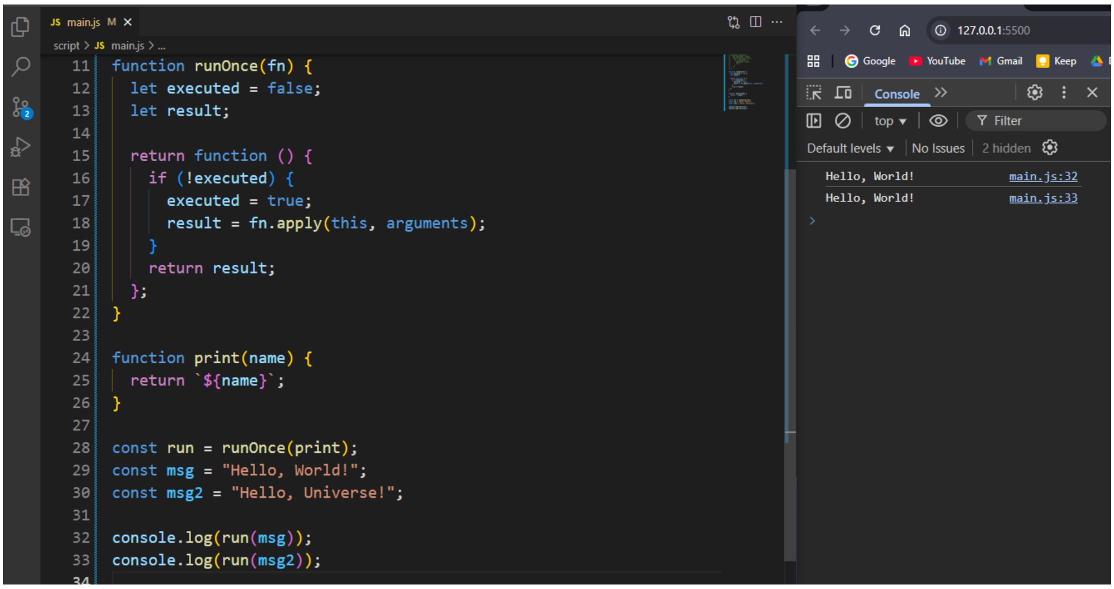


```
function runOnce() {
  let executed = false;
  return function (fn) {
    if (!executed) {
      executed = true;
      fn();
    }
  };
}

const executeOnce = runOnce();

executeOnce(() => {
  console.log("Hello World !!");
});

executeOnce(() => {
  console.log("Hello World !!");
});

executeOnce(() => {
  console.log("Hello World !!");
});
```

```
Hello World !!          main.js:14
```

12. (Bonus) Modify once so subsequent calls return the FIRST result (like a memo of zero-arg function). Keep original version comment above for comparison.

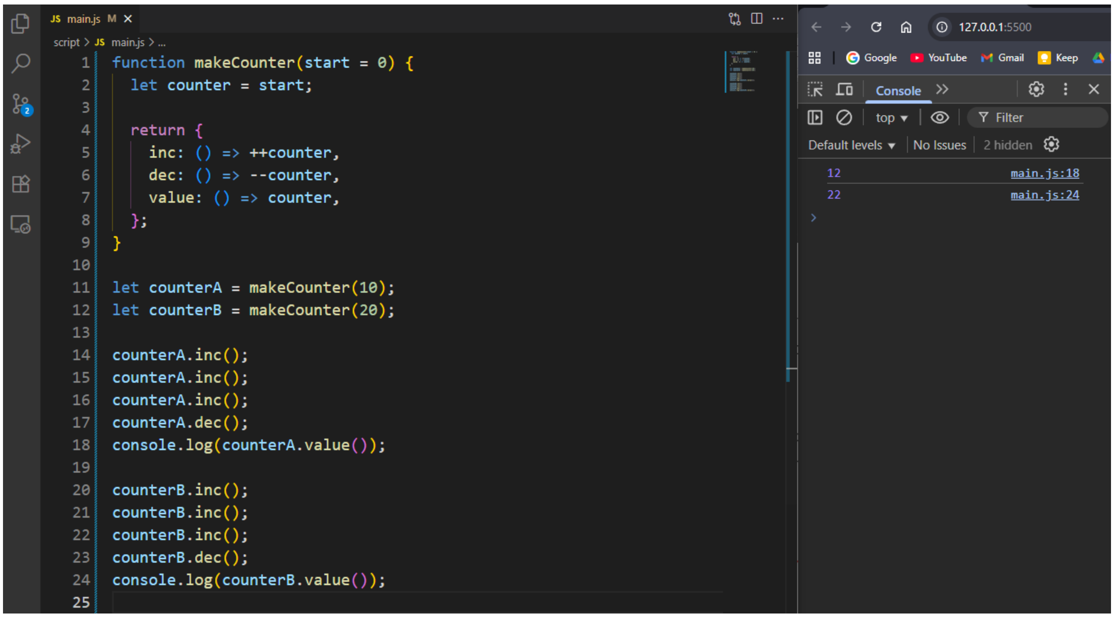
>> Will Return The First Result Only.



```javascript
function runOnce(fn) {
  let executed = false;
  let result;

  return function () {
    if (!executed) {
      executed = true;
      result = fn.apply(this, arguments);
    }
    return result;
  };
}

function print(name) {
  return `${name}`;
}

const run = runOnce(print);
const msg = "Hello, World!";
const msg2 = "Hello, Universe!";

console.log(run(msg));
console.log(run(msg2));
```

```
Hello, World!        main.js:32
Hello, World!        main.js:33
```

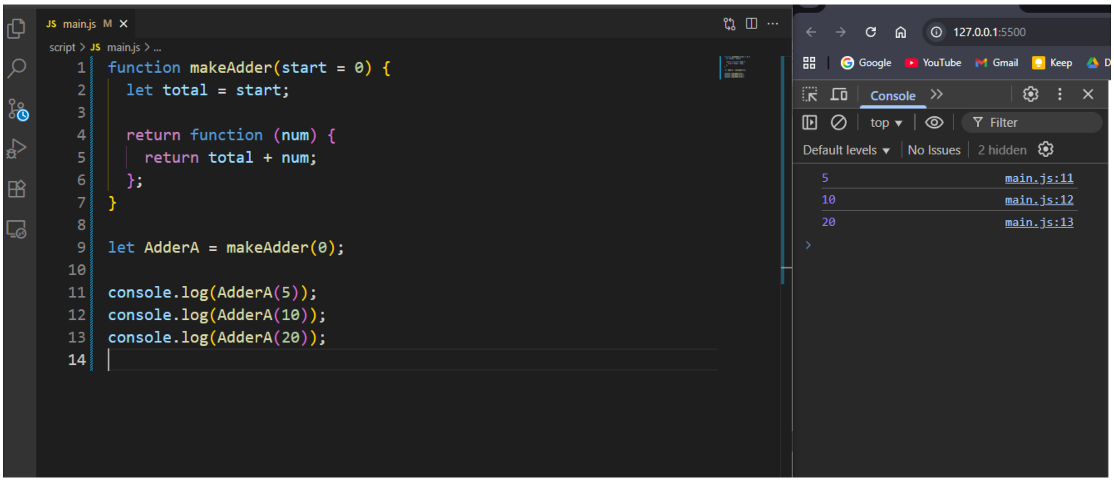
13. (Bonus) Implement makeCounter(start) that returns { inc: fn, dec: fn, value: fn }. State stays private. Demonstrate two independent counters.



```javascript
function makeCounter(start = 0) {
  let counter = start;

  return {
    inc: () => ++counter,
    dec: () => --counter,
    value: () => counter,
  };
}

let counterA = makeCounter(10);
let counterB = makeCounter(20);

counterA.inc();
counterA.inc();
counterA.inc();
counterA.dec();
console.log(counterA.value());

counterB.inc();
counterB.inc();
counterB.inc();
counterB.dec();
console.log(counterB.value());
```

```
12        main.js:18
22        main.js:24
```

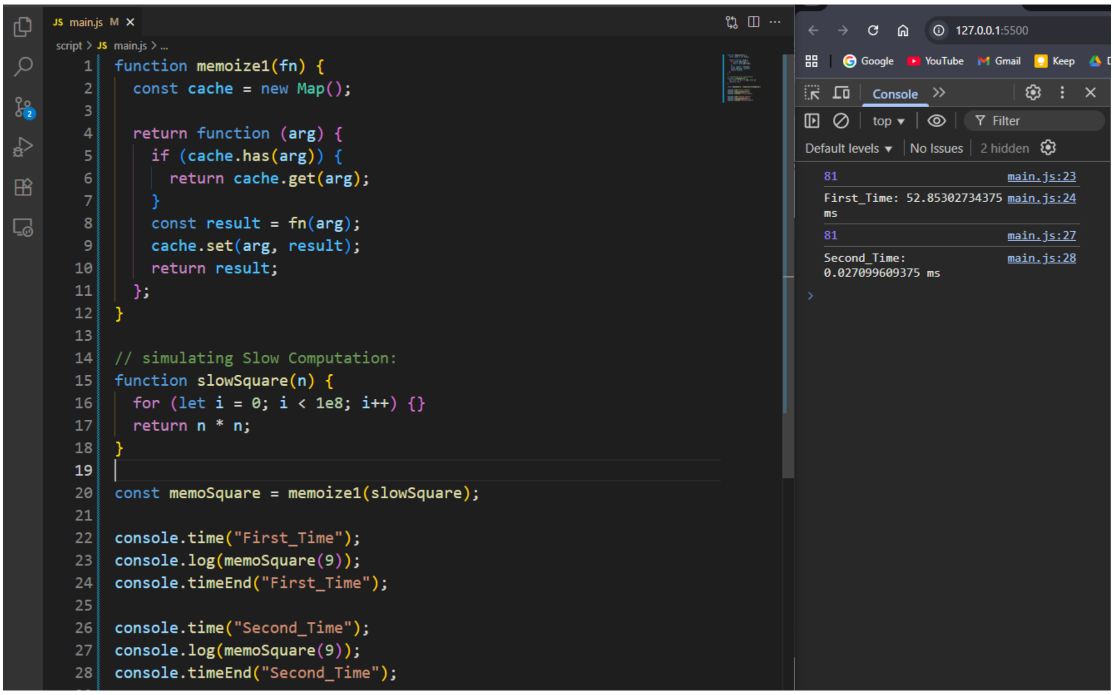
14. makeAdder(start) returns a function that adds its argument to internal total and returns current total each call. Demonstrate separate instances.



```
function makeAdder(start = 0) {
  let total = start;

  return function (num) {
    return total + num;
  };
}

let AdderA = makeAdder(0);

console.log(AdderA(5));
console.log(AdderA(10));
console.log(AdderA(20));
```

```
5                          main.js:11
10                         main.js:12
20                         main.js:13
```

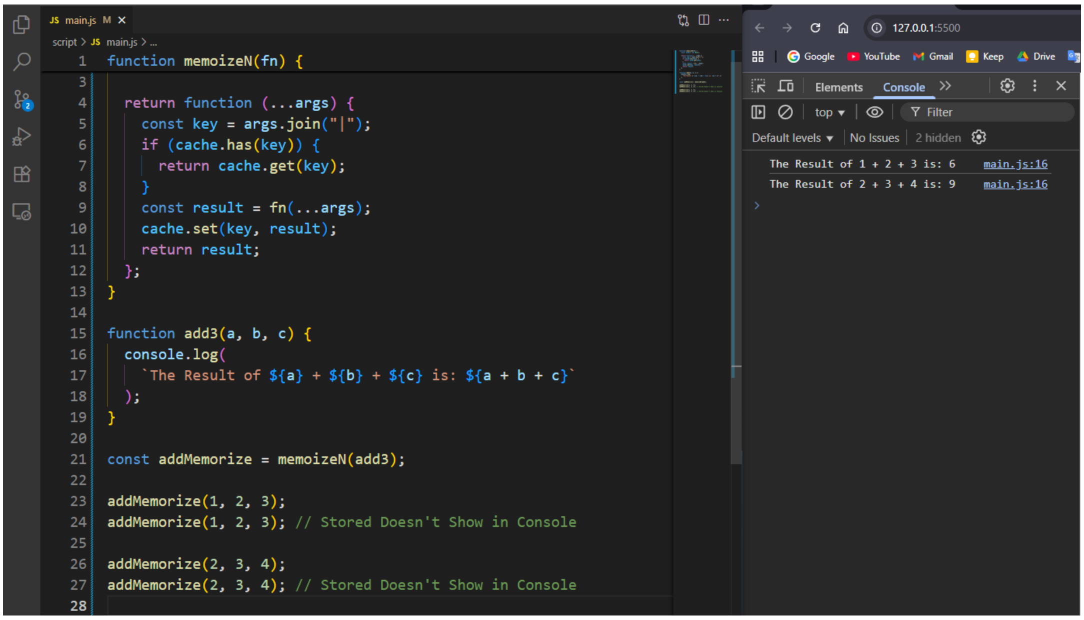
15. Implement memoize1(fn). Show it caches slowSquare(9) twice (timing optional comment).



```
function memoize1(fn) {
  const cache = new Map();

  return function (arg) {
    if (cache.has(arg)) {
      return cache.get(arg);
    }
    const result = fn(arg);
    cache.set(arg, result);
    return result;
  };
}

// simulating Slow Computation:
function slowSquare(n) {
  for (let i = 0; i < 1e8; i++) {}
  return n * n;
}

const memoSquare = memoize1(slowSquare);

console.time("First_Time");
console.log(memoSquare(9));
console.timeEnd("First_Time");

console.time("Second_Time");
console.log(memoSquare(9));
console.timeEnd("Second_Time");
```

```
81                          main.js:23
First_Time: 52.85302734375  main.js:24
ms
81                          main.js:27
Second_Time:                main.js:28
0.027099609375 ms
```

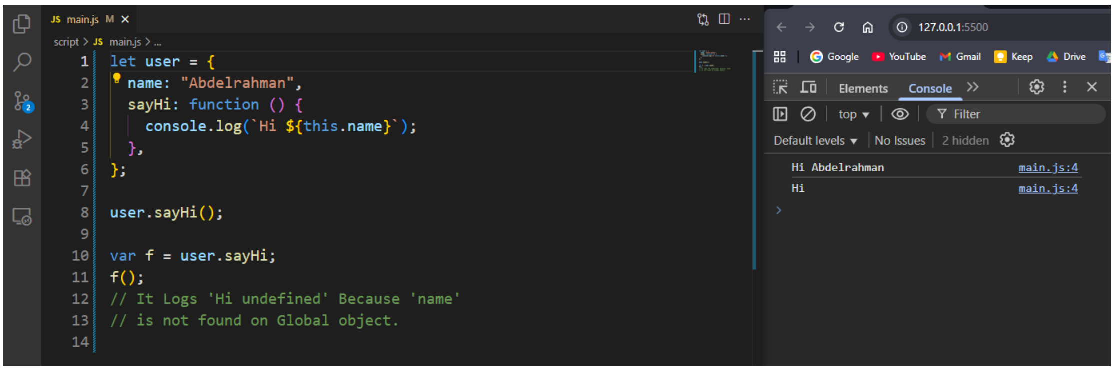
16. (Bonus) Implement memorizeN(fn) that supports any number of primitive args by joining them with '|' as a key. Show with add3(a,b,c).



```javascript
function memoizeN(fn) {

  return function (...args) {
    const key = args.join("|");
    if (cache.has(key)) {
      return cache.get(key);
    }
    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
}

function add3(a, b, c) {
  console.log(
    `The Result of ${a} + ${b} + ${c} is: ${a + b + c}`
  );
}

const addMemorize = memoizeN(add3);

addMemorize(1, 2, 3);
addMemorize(1, 2, 3); // Stored Doesn't Show in Console

addMemorize(2, 3, 4);
addMemorize(2, 3, 4); // Stored Doesn't Show in Console
```

```
The Result of 1 + 2 + 3 is: 6    main.js:16
The Result of 2 + 3 + 4 is: 9    main.js:16
```

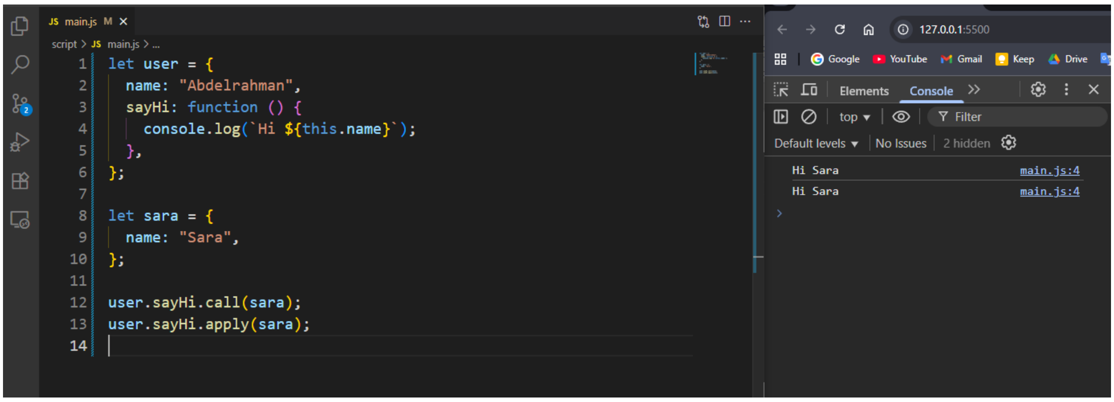
17. Make object user with name and method sayHi logging 'Hi NAME'. Call sayHi, then assign var f = user.sayHi; call f(). Explain (comment) output difference.



```javascript
let user = {
  name: "Abdelrahman",
  sayHi: function () {
    console.log(`Hi ${this.name}`);
  },
};

user.sayHi();

var f = user.sayHi;
f();
// It Logs 'Hi undefined' Because 'name'
// is not found on Global object.
```

```
Hi Abdelrahman    main.js:4
Hi                main.js:4
```

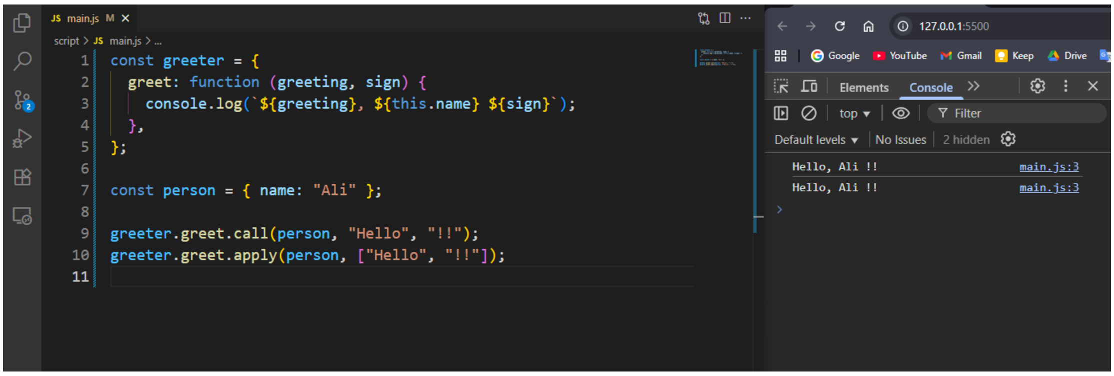
18. Re-use sayHi but call it with another object { name: 'Sara' } using two different ways.



```javascript
let user = {
  name: "Abdelrahman",
  sayHi: function () {
    console.log(`Hi ${this.name}`);
  },
};

let sara = {
  name: "Sara",
};

user.sayHi.call(sara);
user.sayHi.apply(sara);
```

```
Hi Sara    main.js:4
Hi Sara    main.js:4
```

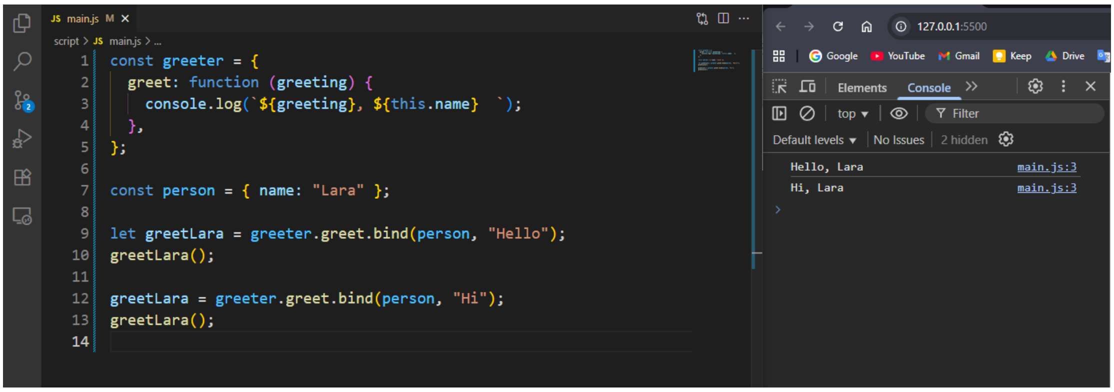
19. Create greeter.greet(greeting,sign). Use apply to invoke it on { name: 'Ali' } with 'Hello','!'.



```javascript
const greeter = {
  greet: function (greeting, sign) {
    console.log(`${greeting}, ${this.name} ${sign}`);
  },
};

const person = { name: "Ali" };

greeter.greet.call(person, "Hello", "!!");
greeter.greet.apply(person, ["Hello", "!!"]);
```

```
Hello, Ali !!                    main.js:3
Hello, Ali !!                    main.js:3
```

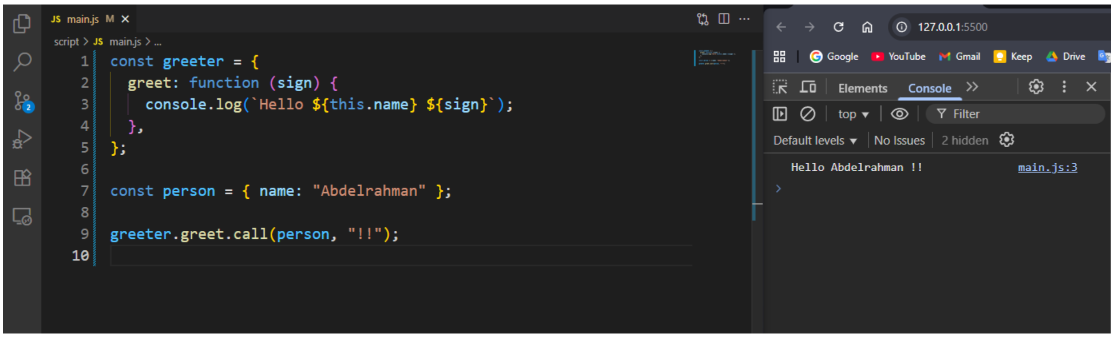
20. Bind greet to { name:'Lara' } as greetLara (no preset greeting). Call with different greetings.



```javascript
const greeter = {
  greet: function (greeting) {
    console.log(`${greeting}, ${this.name}  `);
  },
};

const person = { name: "Lara" };

let greetLara = greeter.greet.bind(person, "Hello");
greetLara();

greetLara = greeter.greet.bind(person, "Hi");
greetLara();
```

```
Hello, Lara                      main.js:3
Hi, Lara                         main.js:3
```

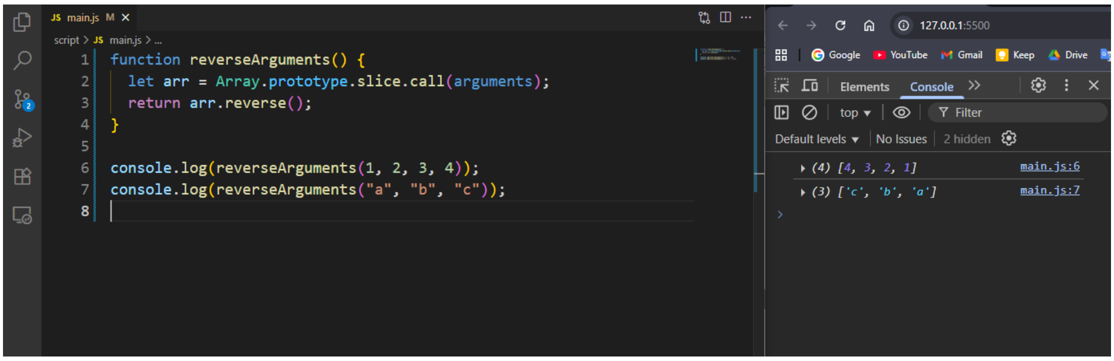
21. Bind greet to produce a sayHello(obj) that always uses greeting 'Hello' but variable sign(!,*,!!,<#).



```javascript
const greeter = {
  greet: function (sign) {
    console.log(`Hello ${this.name} ${sign}`);
  },
};

const person = { name: "Abdelrahman" };

greeter.greet.call(person, "!!");
```

```
Hello Abdelrahman !!             main.js:3
```

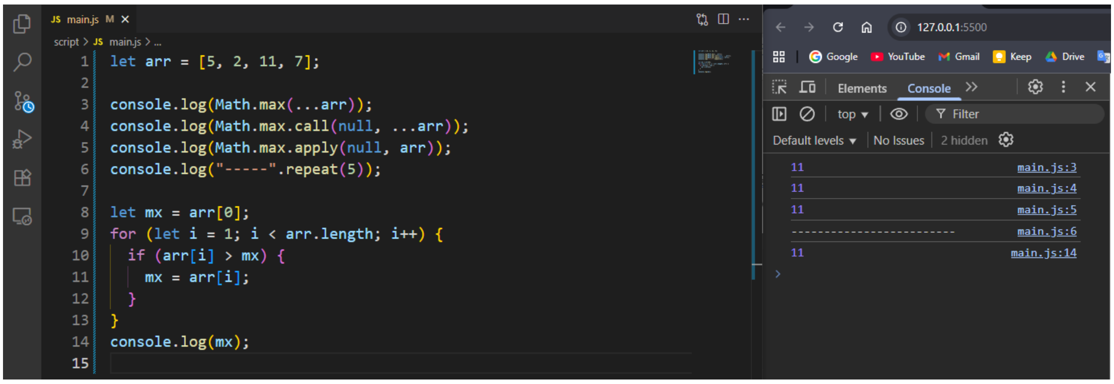
22. Use slice inside a function to convert its arguments(remember it is an array like) to a real array and log reversed copy without mutating original.



```javascript
function reverseArguments() {
  let arr = Array.prototype.slice.call(arguments);
  return arr.reverse();
}

console.log(reverseArguments(1, 2, 3, 4));
console.log(reverseArguments("a", "b", "c"));
```

```
▸ (4) [4, 3, 2, 1]          main.js:6
▸ (3) ['c', 'b', 'a']       main.js:7
```

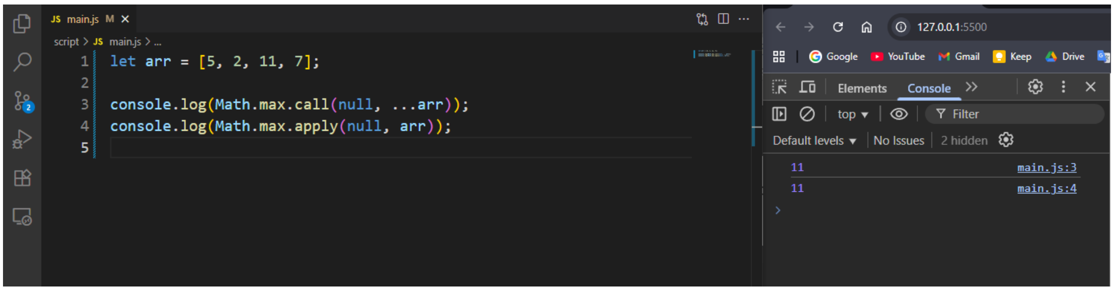
23. Given arr = [5,2,11,7] find max WITHOUT loop using max(). Then show an alternative with a loop.



```javascript
let arr = [5, 2, 11, 7];

console.log(Math.max(...arr));
console.log(Math.max.call(null, ...arr));
console.log(Math.max.apply(null, arr));
console.log("-----".repeat(5));

let mx = arr[0];
for (let i = 1; i < arr.length; i++) {
  if (arr[i] > mx) {
    mx = arr[i];
  }
}
console.log(mx);
```

```
11                               main.js:3
11                               main.js:4
11                               main.js:5
-------------------------        main.js:6
11                               main.js:14
```

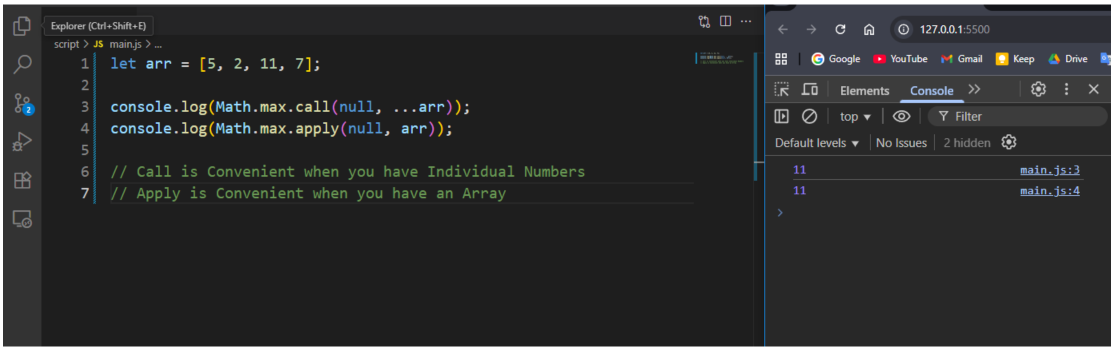
24. Demonstrate calling Math.max with individual numbers using call and explain why apply is better.



```javascript
let arr = [5, 2, 11, 7];

console.log(Math.max.call(null, ...arr));
console.log(Math.max.apply(null, arr));
```

```
11                               main.js:3
11                               main.js:4
```

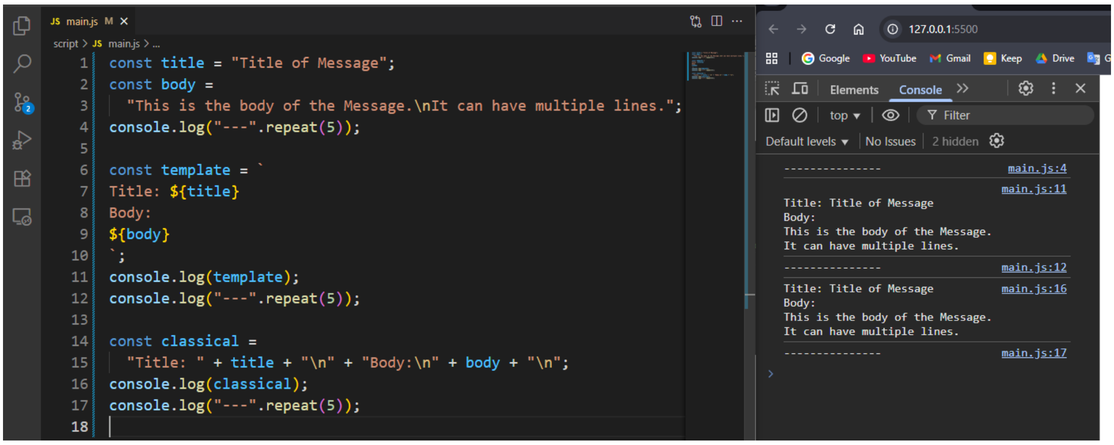
25. Convert string concatenation 'User: '+name+' Age: '+(age+1) into a template literal equivalent.



```javascript
let arr = [5, 2, 11, 7];

console.log(Math.max.call(null, ...arr));
console.log(Math.max.apply(null, arr));

// Call is Convenient when you have Individual Numbers
// Apply is Convenient when you have an Array
```

```
11                                     main.js:3
11                                     main.js:4
>
```

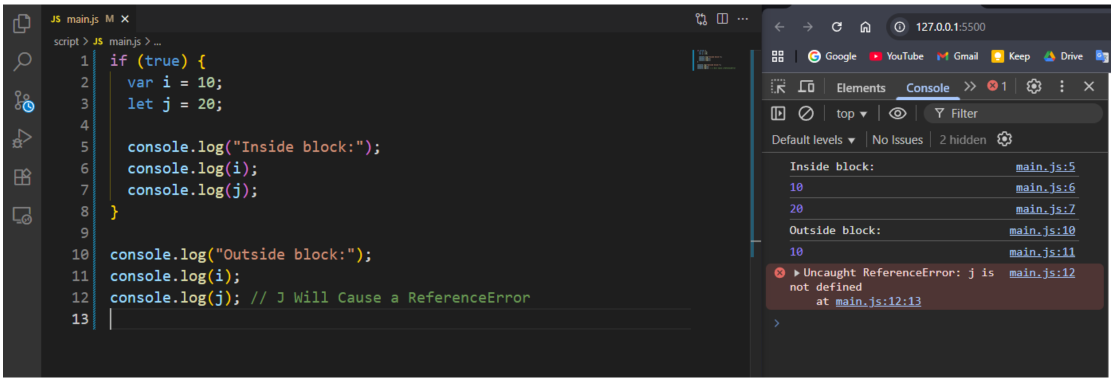
26. Create a multi-line template with variables title and body and log it; show classical \n build version for contrast.



```javascript
const title = "Title of Message";
const body =
  "This is the body of the Message.\nIt can have multiple lines.";
console.log("---".repeat(5));

const template = `
Title: ${title}
Body:
${body}
`;
console.log(template);
console.log("---".repeat(5));

const classical =
  "Title: " + title + "\n" + "Body:\n" + body + "\n";
console.log(classical);
console.log("---".repeat(5));
```

```
---------------                        main.js:4
                                       main.js:11
Title: Title of Message
Body:
This is the body of the Message.
It can have multiple lines.
---------------                        main.js:12
Title: Title of Message                main.js:16
Body:
This is the body of the Message.
It can have multiple lines.
---------------                        main.js:17
>
```

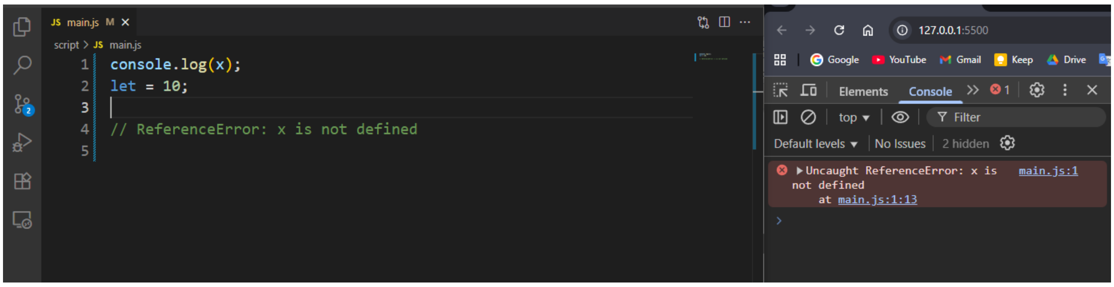
27. Write a block with var i and let j inside if(true) and log both inside and outside. Comment which leaks.



```javascript
if (true) {
  var i = 10;
  let j = 20;

  console.log("Inside block:");
  console.log(i);
  console.log(j);
}

console.log("Outside block:");
console.log(i);
console.log(j); // J Will Cause a ReferenceError
```

```
Inside block:                          main.js:5
10                                     main.js:6
20                                     main.js:7
Outside block:                         main.js:10
10                                     main.js:11
✖ ▸Uncaught ReferenceError: j is       main.js:12
not defined
    at main.js:12:13
>
```

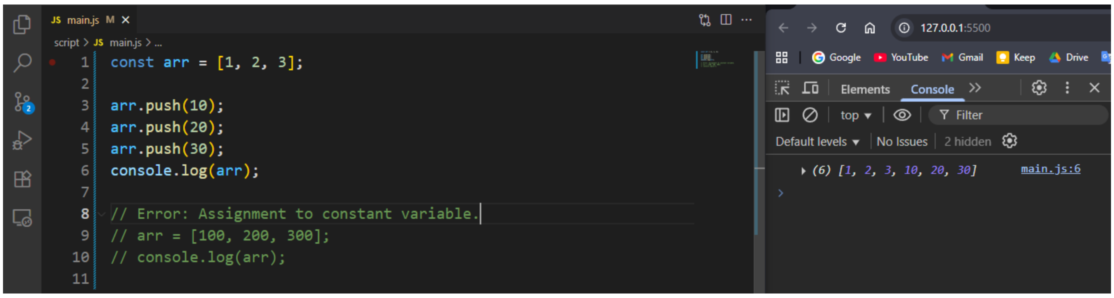
28. Write code that tries to log x before let x = 5;



```
console.log(x);
let = 10;

// ReferenceError: x is not defined
```

```
▶Uncaught ReferenceError: x is    main.js:1
 not defined
    at main.js:1:13
```

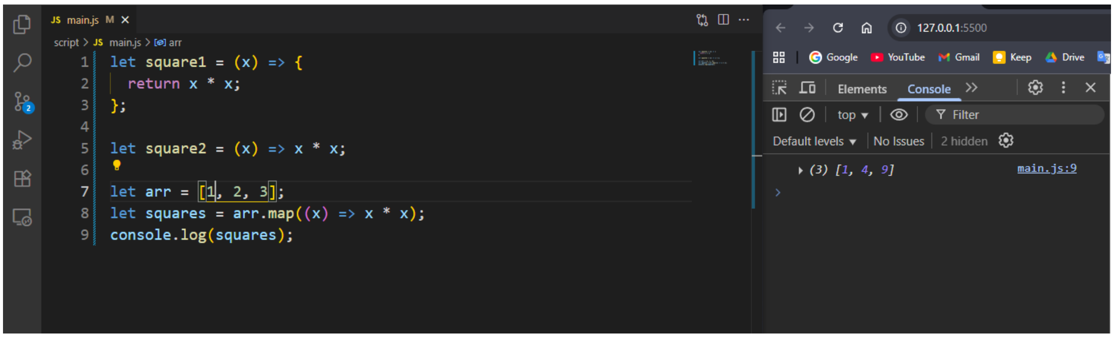
29. Show that pushing to a const array works but reassigning it does not (comment error you would get if attempted—do not actually break execution).



```
const arr = [1, 2, 3];

arr.push(10);
arr.push(20);
arr.push(30);
console.log(arr);

// Error: Assignment to constant variable.
// arr = [100, 200, 300];
// console.log(arr);
```

```
▸ (6) [1, 2, 3, 10, 20, 30]      main.js:6
```

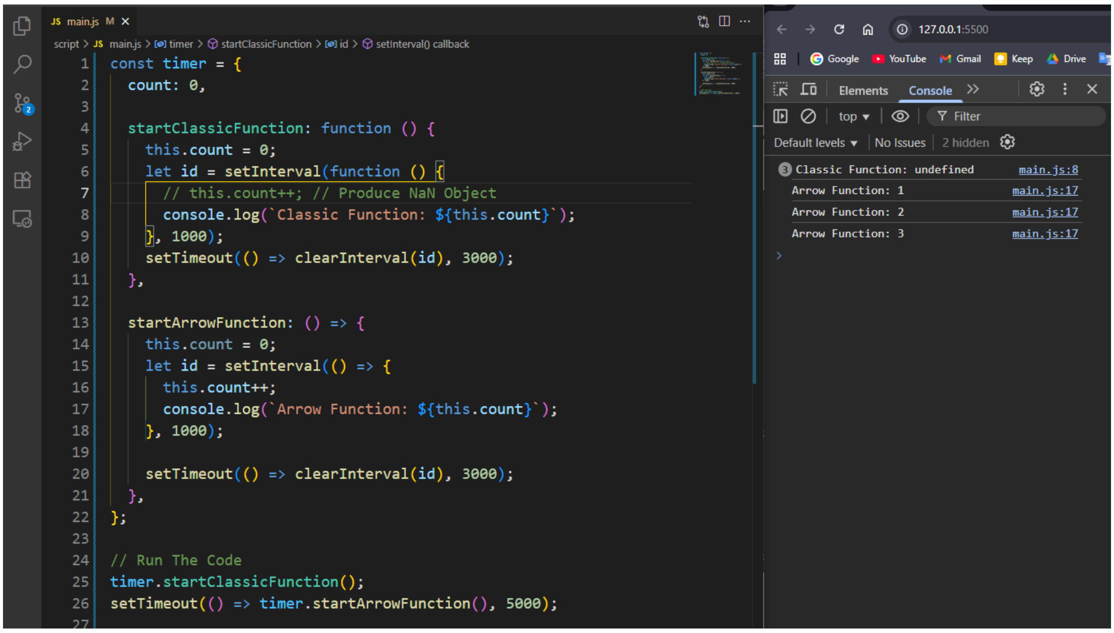
30. Rewrite a normal function square(n) { return n*n; } as arrow in three forms: full body, concise, inline in map over [1,2,3].



```
let square1 = (x) => {
  return x * x;
};

let square2 = (x) => x * x;

let arr = [1, 2, 3];
let squares = arr.map((x) => x * x);
console.log(squares);
```

```
▸ (3) [1, 4, 9]      main.js:9
```

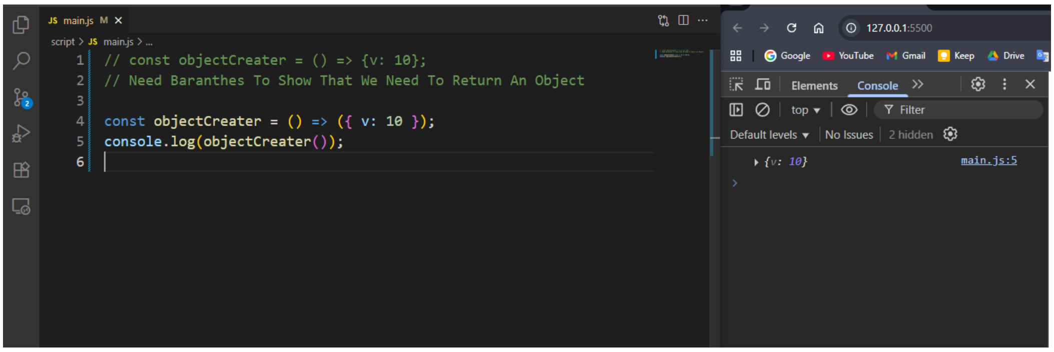
31. Create object timer with count:0 and method startClassic using setInterval(function(){...}) and startArrow using setInterval(()=>{...}). Show difference in how this works (stop after a few increments using clearInterval).



```javascript
const timer = {
  count: 0,

  startClassicFunction: function () {
    this.count = 0;
    let id = setInterval(function () {
      // this.count++; // Produce NaN Object
      console.log(`Classic Function: ${this.count}`);
    }, 1000);
    setTimeout(() => clearInterval(id), 3000);
  },

  startArrowFunction: () => {
    this.count = 0;
    let id = setInterval(() => {
      this.count++;
      console.log(`Arrow Function: ${this.count}`);
    }, 1000);

    setTimeout(() => clearInterval(id), 3000);
  },
};

// Run The Code
timer.startClassicFunction();
setTimeout(() => timer.startArrowFunction(), 5000);
```

```
3 Classic Function: undefined     main.js:8
  Arrow Function: 1               main.js:17
  Arrow Function: 2               main.js:17
  Arrow Function: 3               main.js:17
```

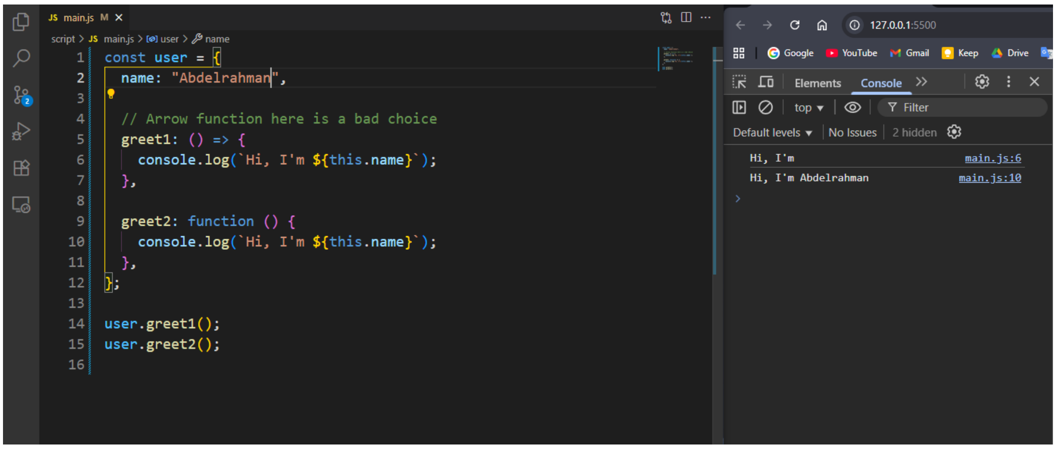
32. Write an arrow function that returns an object {v:10}. Show the need for parentheses.



```javascript
// const objectCreater = () => {v: 10};
// Need Baranthes To Show That We Need To Return An Object

const objectCreater = () => ({ v: 10 });
console.log(objectCreater());
```

```
▸ {v: 10}     main.js:5
```

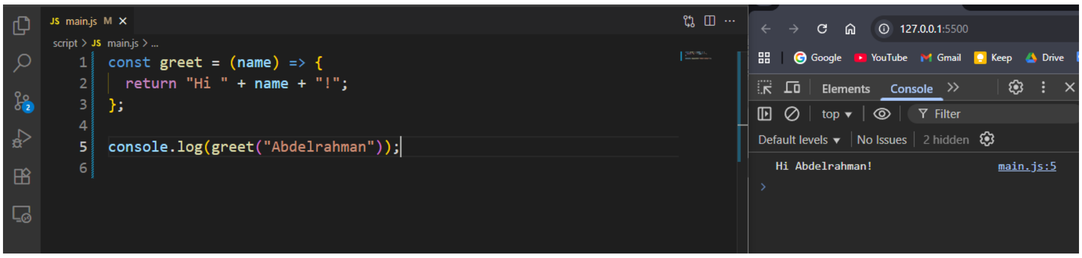
33. Give one example where arrow is a bad choice (e.g., method needing dynamic this).



```javascript
const user = {
  name: "Abdelrahman",

  // Arrow function here is a bad choice
  greet1: () => {
    console.log(`Hi, I'm ${this.name}`);
  },

  greet2: function () {
    console.log(`Hi, I'm ${this.name}`);
  },
};

user.greet1();
user.greet2();
```

```
Hi, I'm                  main.js:6
Hi, I'm Abdelrahman      main.js:10
```

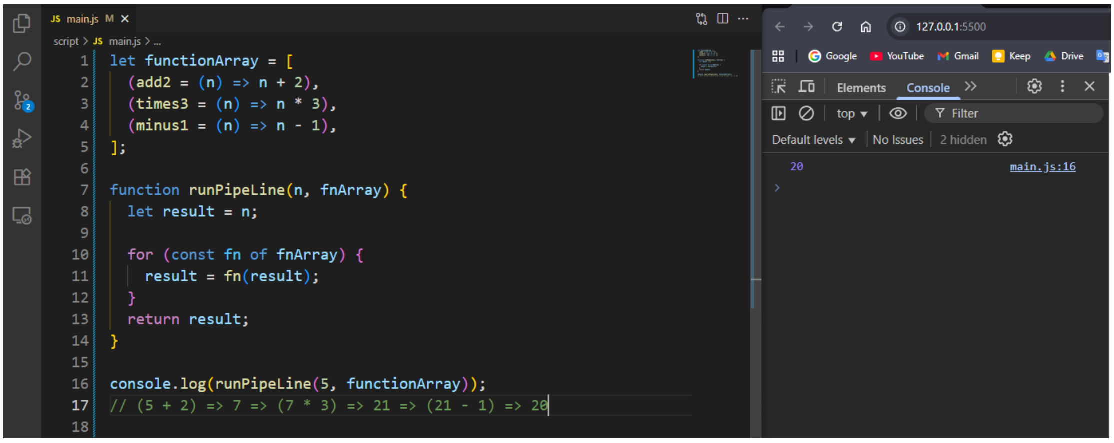
34. Start with function greet(name){ return 'Hi '+name+'!'; } Convert to arrow function using Const not let ya habeby :).



```
const greet = (name) => {
  return "Hi " + name + "!";
};

console.log(greet("Abdelrahman"));
```

```
Hi Abdelrahman!                                        main.js:5
```

35. Build pipeline functions: add2, times3, minus1 (all arrows). Write runPipeline(n, fnsArray) that loops through and applies each. Test runPipeline(5, [add2,times3,minus1]).



```
let functionArray = [
  (add2 = (n) => n + 2),
  (times3 = (n) => n * 3),
  (minus1 = (n) => n - 1),
];

function runPipeLine(n, fnArray) {
  let result = n;

  for (const fn of fnArray) {
    result = fn(result);
  }
  return result;
}

console.log(runPipeLine(5, functionArray));
// (5 + 2) => 7 => (7 * 3) => 21 => (21 - 1) => 20
```

```
20                                                     main.js:16
```

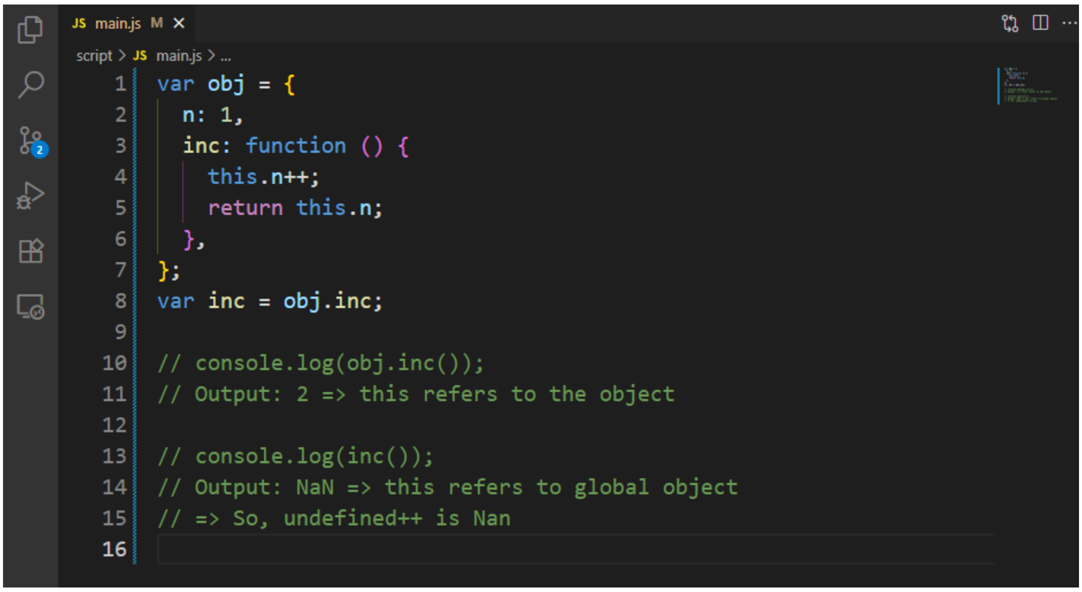
36. (write answers BEFORE running): Explain both lines.

var obj = { n: 1, inc: function(){ this.n++; return this.n; } };

var inc = obj.inc;

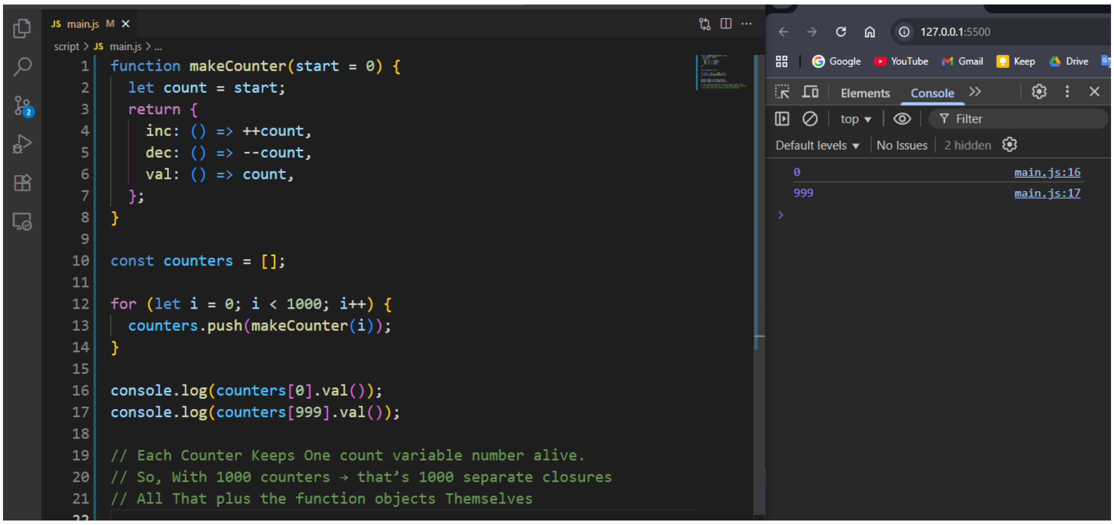
console.log(obj.inc());

console.log(inc());



```
var obj = {
  n: 1,
  inc: function () {
    this.n++;
    return this.n;
  },
};
var inc = obj.inc;

// console.log(obj.inc());
// Output: 2 => this refers to the object

// console.log(inc());
// Output: NaN => this refers to global object
// => So, undefined++ is Nan
```

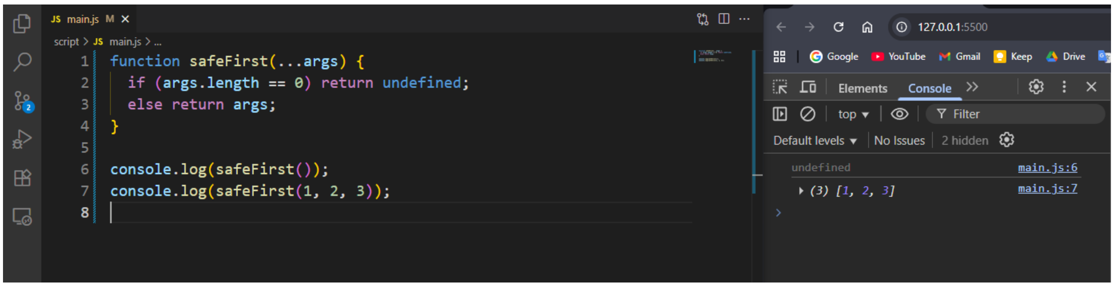
37. Create many counters in a loop (e.g. 1000) and store them in an array. Comment on potential memory considerations of large closure arrays.



```
function makeCounter(start = 0) {
  let count = start;
  return {
    inc: () => ++count,
    dec: () => --count,
    val: () => count,
  };
}

const counters = [];

for (let i = 0; i < 1000; i++) {
  counters.push(makeCounter(i));
}

console.log(counters[0].val());
console.log(counters[999].val());

// Each Counter Keeps One count variable number alive.
// So, With 1000 counters → that’s 1000 separate closures
// All That plus the function objects Themselves
```

```
0                                         main.js:16
999                                       main.js:17
```

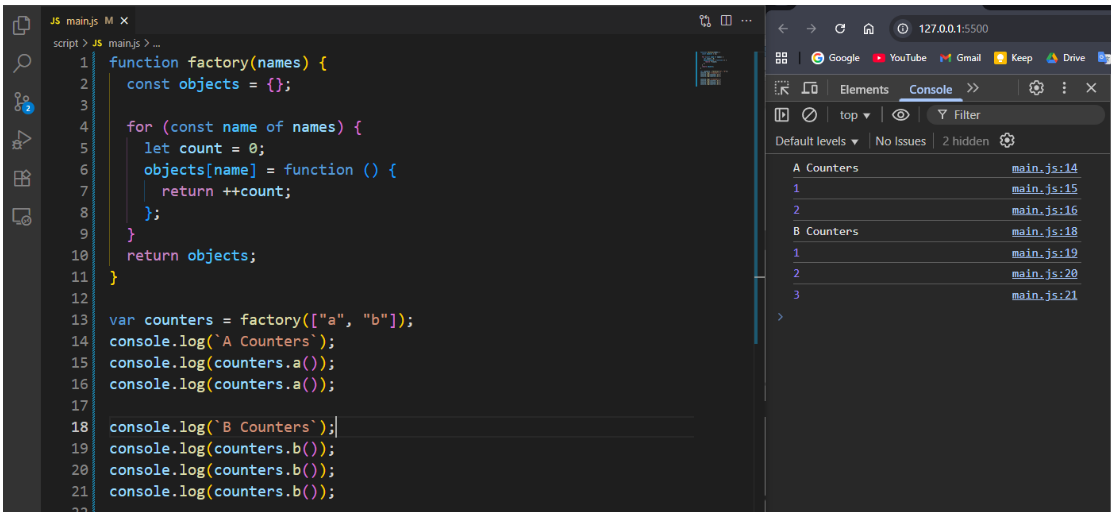
38. Write safeFirst() that returns undefined if called with zero args else return array of the args.



```
function safeFirst(...args) {
  if (args.length == 0) return undefined;
  else return args;
}

console.log(safeFirst());
console.log(safeFirst(1, 2, 3));
```

```
undefined                                 main.js:6
▸ (3) [1, 2, 3]                           main.js:7
```

39. factory(namesArray) returns object with a counter function for each name (all independent). Example: var counters = factory(['a','b']); counters.a(); counters.b();



```javascript
function factory(names) {
  const objects = {};

  for (const name of names) {
    let count = 0;
    objects[name] = function () {
      return ++count;
    };
  }
  return objects;
}

var counters = factory(["a", "b"]);
console.log(`A Counters`);
console.log(counters.a());
console.log(counters.a());

console.log(`B Counters`);
console.log(counters.b());
console.log(counters.b());
console.log(counters.b());
```

```
A Counters                  main.js:14
1                           main.js:15
2                           main.js:16
B Counters                  main.js:18
1                           main.js:19
2                           main.js:20
3                           main.js:21
```

40. Write 2 things that were new or tricky today (comment).

&