

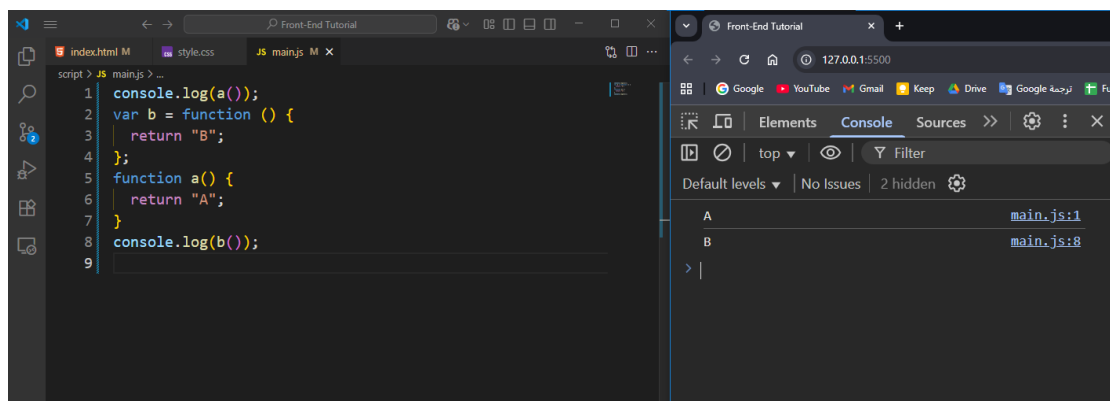
## JavaScript Assignments Day-5

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

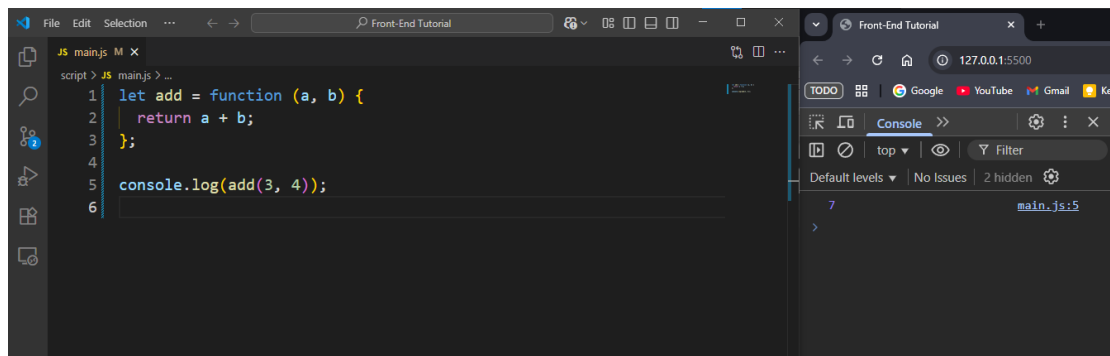
```
console.log(a());  
var b = function(){ return 'B'; };  
function a(){ return 'A'; }  
console.log(b());
```

After verifying, explain (one short line) why a works before definition and b does not.

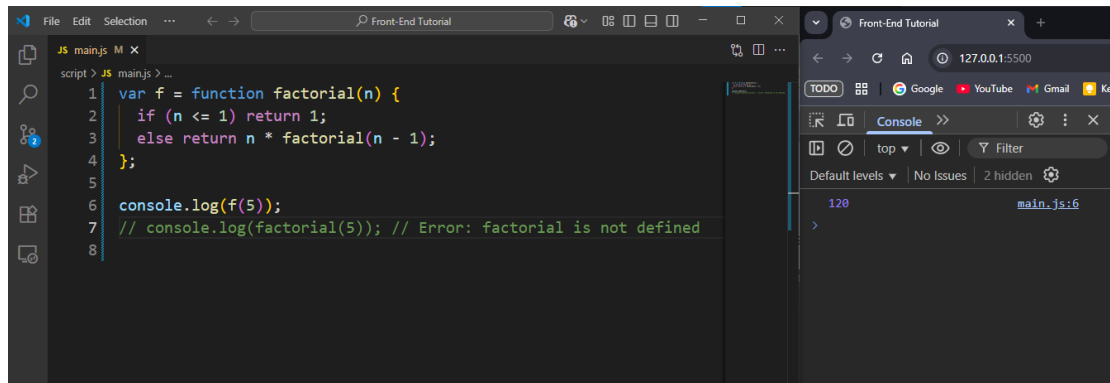
>> Because of Hoisting (The Functions Are Hoisted), The Function Runs Fine.



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).



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.

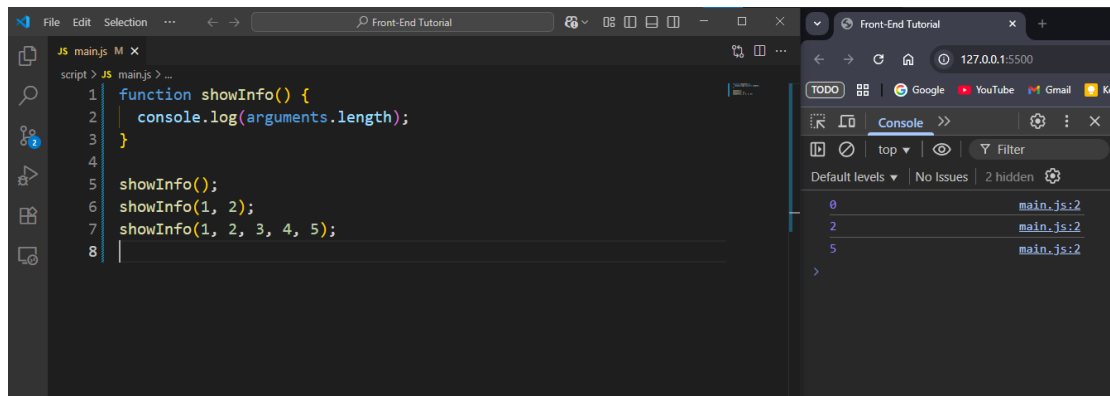


The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines a recursive function 'factorial' using a named function expression assigned to a variable 'f'. The function has a base case for n <= 1 and a recursive case. It logs the result of f(5). A comment indicates that calling factorial(5) directly would result in an error because the internal name 'factorial' is not globally accessible.

```
1 var f = function factorial(n) {  
2   if (n <= 1) return 1;  
3   else return n * factorial(n - 1);  
4 };  
5  
6 console.log(f(5));  
7 // console.log(factorial(5)); // Error: factorial is not defined  
8
```

The browser console shows the output '120' from 'main.js:6'.

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



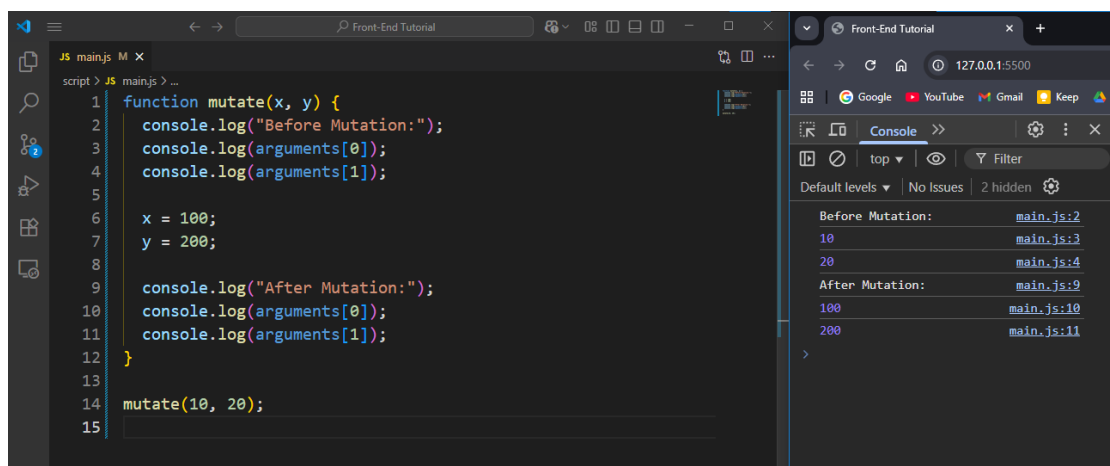
The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines a function 'showInfo' that logs the number of arguments and each argument. It is called with no arguments, two arguments, and five arguments.

```
1 function showInfo() {  
2   console.log(arguments.length);  
3 }  
4  
5 showInfo();  
6 showInfo(1, 2);  
7 showInfo(1, 2, 3, 4, 5);  
8
```

The browser console shows the following output:

- 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.



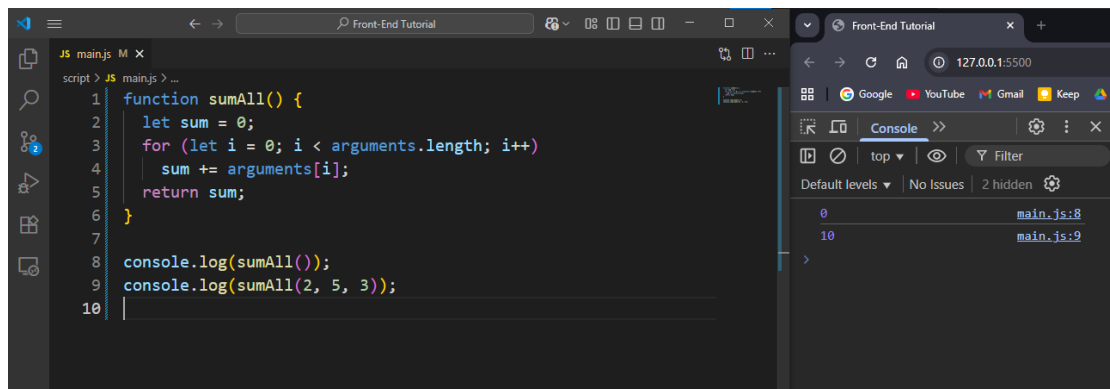
The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines a function 'mutate' that logs the arguments before and after mutation. It changes the values of x and y inside the function. It is called with arguments 10 and 20.

```
1 function mutate(x, y) {  
2   console.log("Before Mutation:");  
3   console.log(arguments[0]);  
4   console.log(arguments[1]);  
5  
6   x = 100;  
7   y = 200;  
8  
9   console.log("After Mutation:");  
10  console.log(arguments[0]);  
11  console.log(arguments[1]);  
12 }  
13  
14 mutate(10, 20);  
15
```

The browser console shows the following output:

- 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)

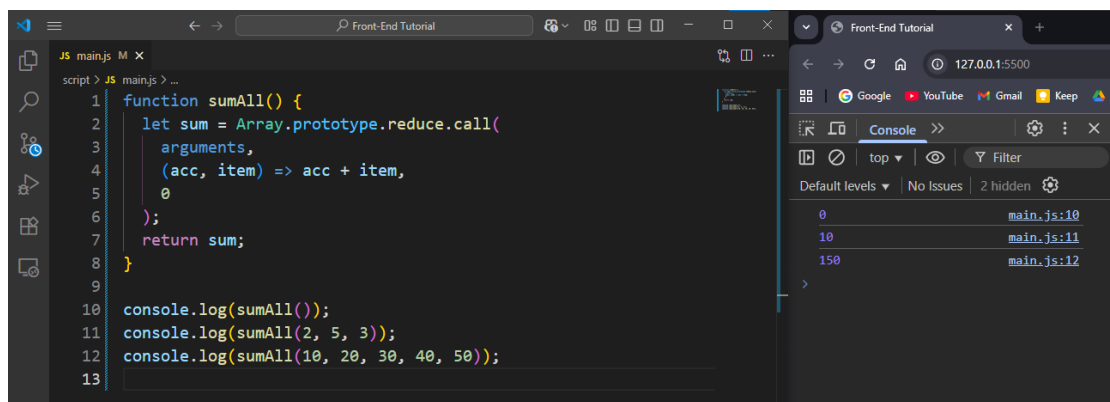
6. Implement sumAll() using only the arguments object (no arrays) to total all numeric arguments. Test sumAll(2,5,3) and sumAll().



The screenshot shows a VS Code editor with a file named 'main.js'. The code defines a function `sumAll()` that initializes a `sum` variable to 0, then iterates through the `arguments` object using a `for` loop to calculate the total. The function is tested with `console.log(sumAll());` and `console.log(sumAll(2, 5, 3));`. The browser console on the right shows the output: 0 for the first call and 10 for the second call.

```
1 function sumAll() {
2   let sum = 0;
3   for (let i = 0; i < arguments.length; i++)
4     sum += arguments[i];
5   return sum;
6 }
7
8 console.log(sumAll());
9 console.log(sumAll(2, 5, 3));
10
```

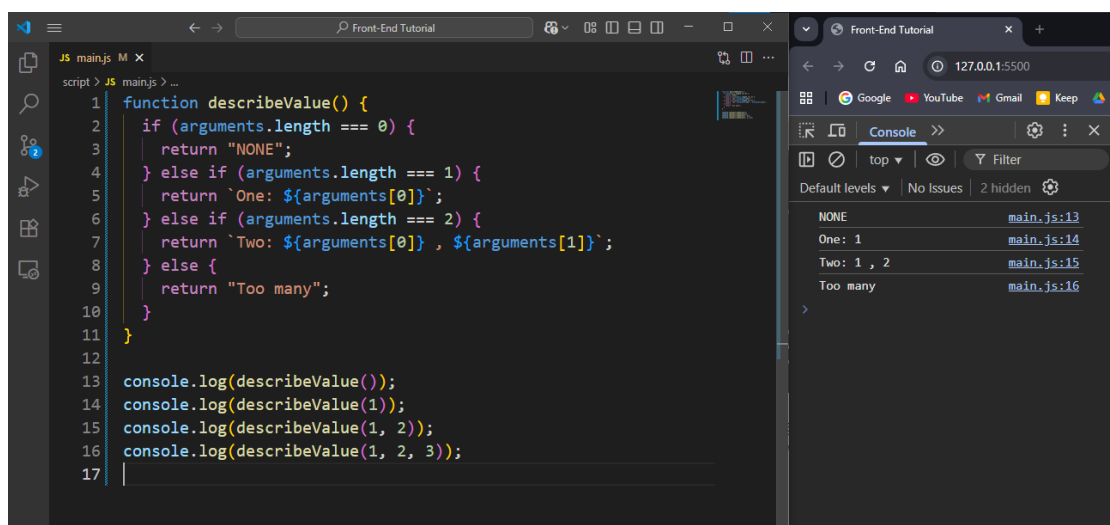
7. Implement sumAll() using only the arguments object but with the Array method reduce.



The screenshot shows a VS Code editor with a file named 'main.js'. The code defines a function `sumAll()` that uses `Array.prototype.reduce.call()` to sum the arguments. The function is tested with `console.log(sumAll());`, `console.log(sumAll(2, 5, 3));`, and `console.log(sumAll(10, 20, 30, 40, 50));`. The browser console on the right shows the output: 0, 10, and 150.

```
1 function sumAll() {
2   let sum = Array.prototype.reduce.call(
3     arguments,
4     (acc, item) => acc + item,
5     0
6   );
7   return sum;
8 }
9
10 console.log(sumAll());
11 console.log(sumAll(2, 5, 3));
12 console.log(sumAll(10, 20, 30, 40, 50));
13
```

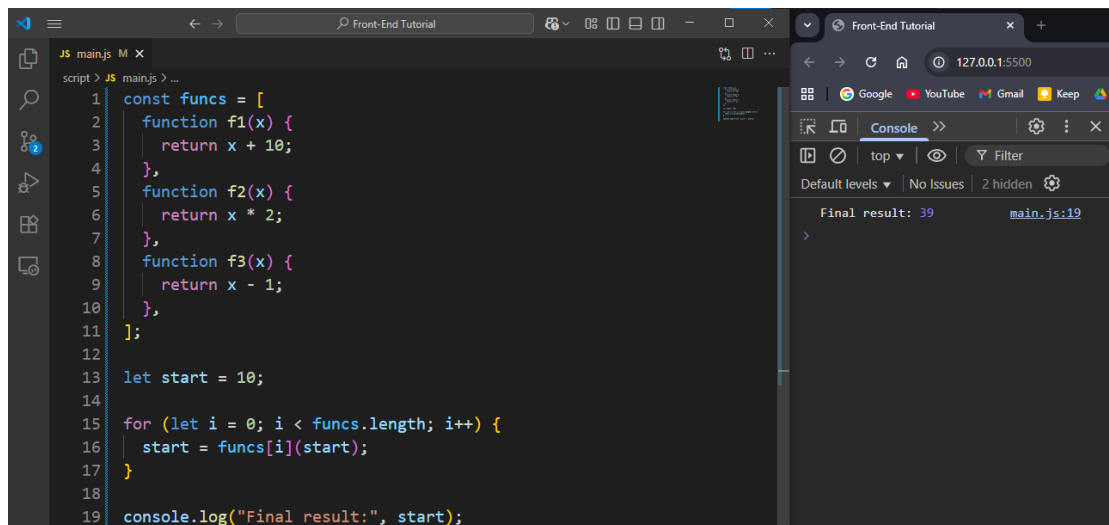
8. Write describeValue that returns different strings based on number of args: 0 -> 'none', 1 -> 'one:'+val, 2 -> 'two:'+a+', '+b else 'too many'.



The screenshot shows a VS Code editor with a file named 'main.js'. The code defines a function `describeValue()` that uses conditional logic to return different strings based on the number of arguments. The function is tested with `console.log(describeValue());`, `console.log(describeValue(1));`, `console.log(describeValue(1, 2));`, and `console.log(describeValue(1, 2, 3));`. The browser console on the right shows the output: NONE, One: 1, Two: 1, 2, and Too many.

```
1 function describeValue() {
2   if (arguments.length === 0) {
3     return "NONE";
4   } else if (arguments.length === 1) {
5     return `One: ${arguments[0]}`;
6   } else if (arguments.length === 2) {
7     return `Two: ${arguments[0]}, ${arguments[1]}`;
8   } else {
9     return "Too many";
10  }
11 }
12
13 console.log(describeValue());
14 console.log(describeValue(1));
15 console.log(describeValue(1, 2));
16 console.log(describeValue(1, 2, 3));
17
```

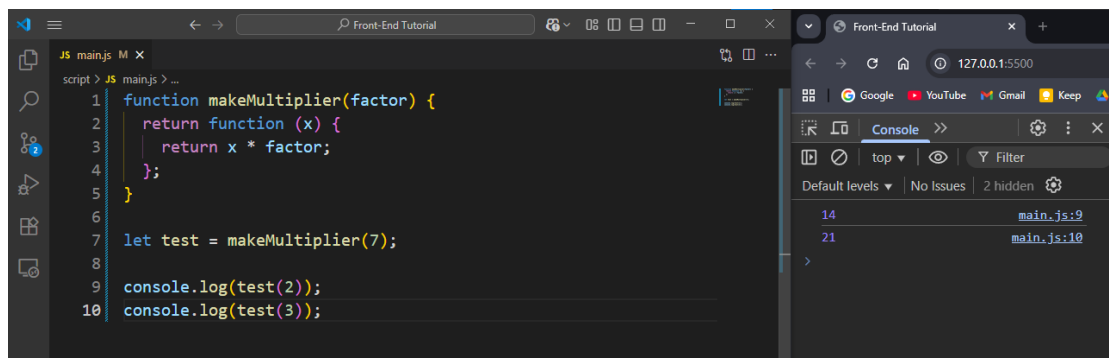
9. Create an array `funcs` of three small anonymous functions that transform a number. Apply them in order to `start = 10` (loop). Log final result.



The screenshot shows a VS Code editor with a JavaScript file named `main.js`. The code defines an array `funcs` containing three functions: `f1(x) { return x + 10; }`, `f2(x) { return x * 2; }`, and `f3(x) { return x - 1; }`. A `start` variable is initialized to 10. A `for` loop iterates over the `funcs` array, applying each function to the `start` value. Finally, `console.log("Final result:", start);` is executed. The browser console on the right shows the output: "Final result: 39" at `main.js:19`.

```
1 const funcs = [
2   function f1(x) {
3     return x + 10;
4   },
5   function f2(x) {
6     return x * 2;
7   },
8   function f3(x) {
9     return x - 1;
10  },
11 ];
12
13 let start = 10;
14
15 for (let i = 0; i < funcs.length; i++) {
16   start = funcs[i](start);
17 }
18
19 console.log("Final result:", start);
```

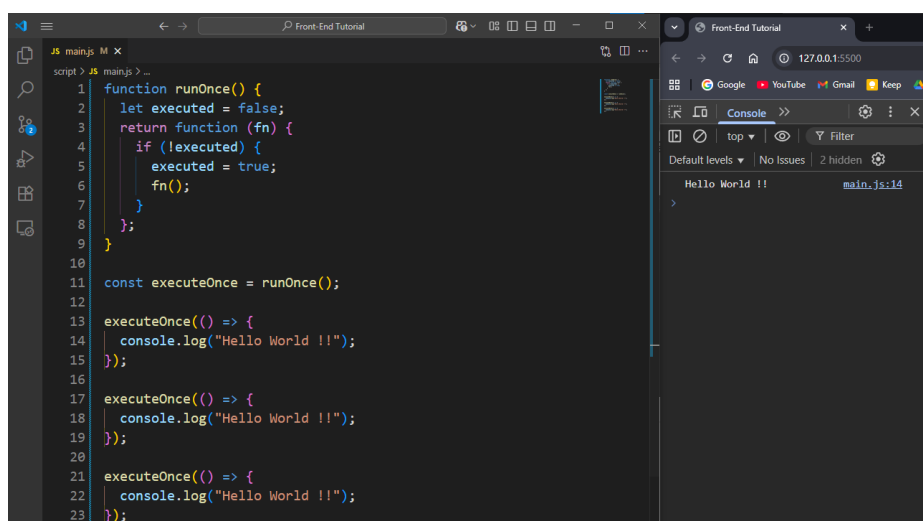
10. Write `makeMultiplier(factor)` returning a function(`n`) that multiplies. Create double and triple; test with 7.



The screenshot shows a VS Code editor with a JavaScript file named `main.js`. The code defines a `makeMultiplier(factor)` function that returns a function `(x) { return x * factor; }`. A `test` variable is created by calling `makeMultiplier(7)`. The code then tests the `test` function with `test(2)` and `test(3)`, logging the results. The browser console on the right shows the output: "14" at `main.js:9` and "21" at `main.js:10`.

```
1 function makeMultiplier(factor) {
2   return function (x) {
3     return x * factor;
4   };
5 }
6
7 let test = makeMultiplier(7);
8
9 console.log(test(2));
10 console.log(test(3));
```

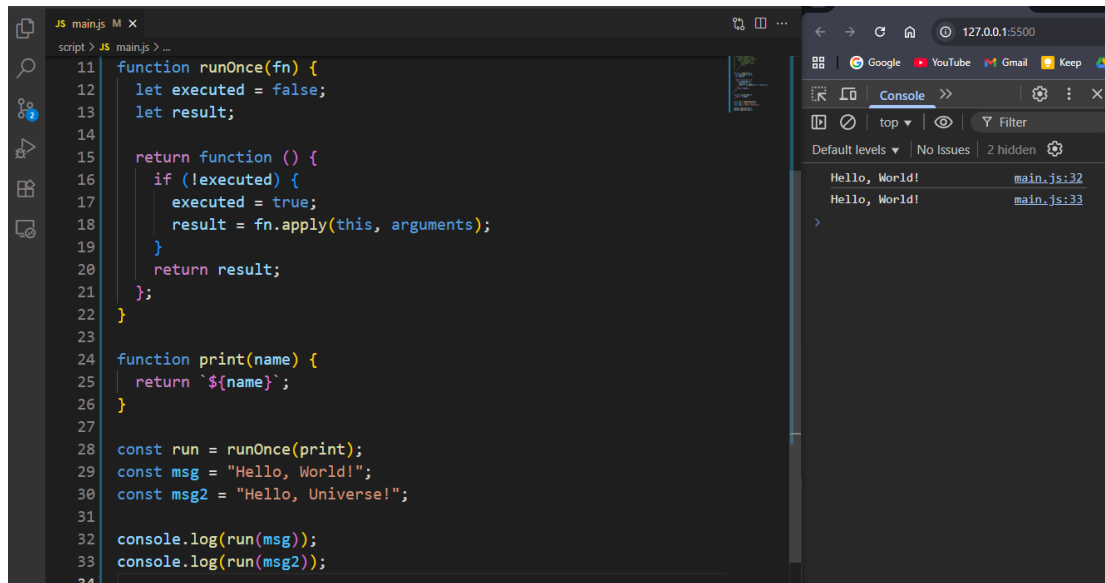
11. Implement `once(fn)` runs `fn` only first time, returns its return value. Test with a function that logs and returns a string.



The screenshot shows a VS Code editor with a JavaScript file named `main.js`. The code defines a `runOnce()` function that returns a function `(fn) { if (!executed) { executed = true; fn(); } }`. A `executeOnce` variable is created by calling `runOnce()`. The code then tests the `executeOnce` function with three different functions, each logging "Hello World !!". The browser console on the right shows the output: "Hello World !!" at `main.js:14`.

```
1 function runOnce() {
2   let executed = false;
3   return function (fn) {
4     if (!executed) {
5       executed = true;
6       fn();
7     }
8   };
9 }
10
11 const executeOnce = runOnce();
12
13 executeOnce(() => {
14   console.log("Hello World !!");
15 });
16
17 executeOnce(() => {
18   console.log("Hello World !!");
19 });
20
21 executeOnce(() => {
22   console.log("Hello World !!");
23 });
```

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.

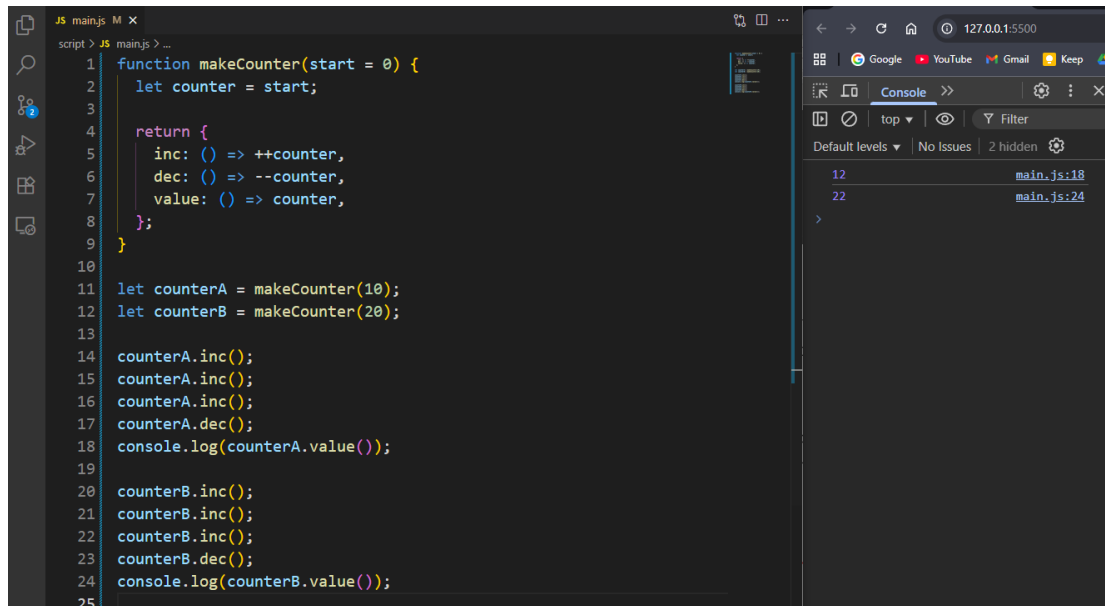


```
11 function runOnce(fn) {
12   let executed = false;
13   let result;
14
15   return function () {
16     if (!executed) {
17       executed = true;
18       result = fn.apply(this, arguments);
19     }
20     return result;
21   };
22 }
23
24 function print(name) {
25   return `${name}`;
26 }
27
28 const run = runOnce(print);
29 const msg = "Hello, World!";
30 const msg2 = "Hello, Universe!";
31
32 console.log(run(msg));
33 console.log(run(msg2));
34
```

Console output:

```
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.

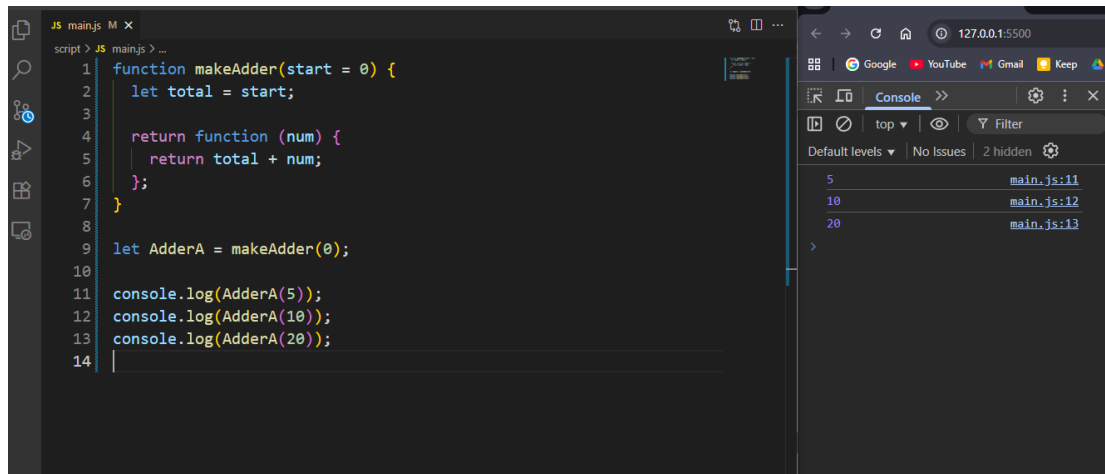


```
1 function makeCounter(start = 0) {
2   let counter = start;
3
4   return {
5     inc: () => ++counter,
6     dec: () => --counter,
7     value: () => counter,
8   };
9 }
10
11 let counterA = makeCounter(10);
12 let counterB = makeCounter(20);
13
14 counterA.inc();
15 counterA.inc();
16 counterA.inc();
17 counterA.dec();
18 console.log(counterA.value());
19
20 counterB.inc();
21 counterB.inc();
22 counterB.inc();
23 counterB.dec();
24 console.log(counterB.value());
25
```

Console output:

```
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.

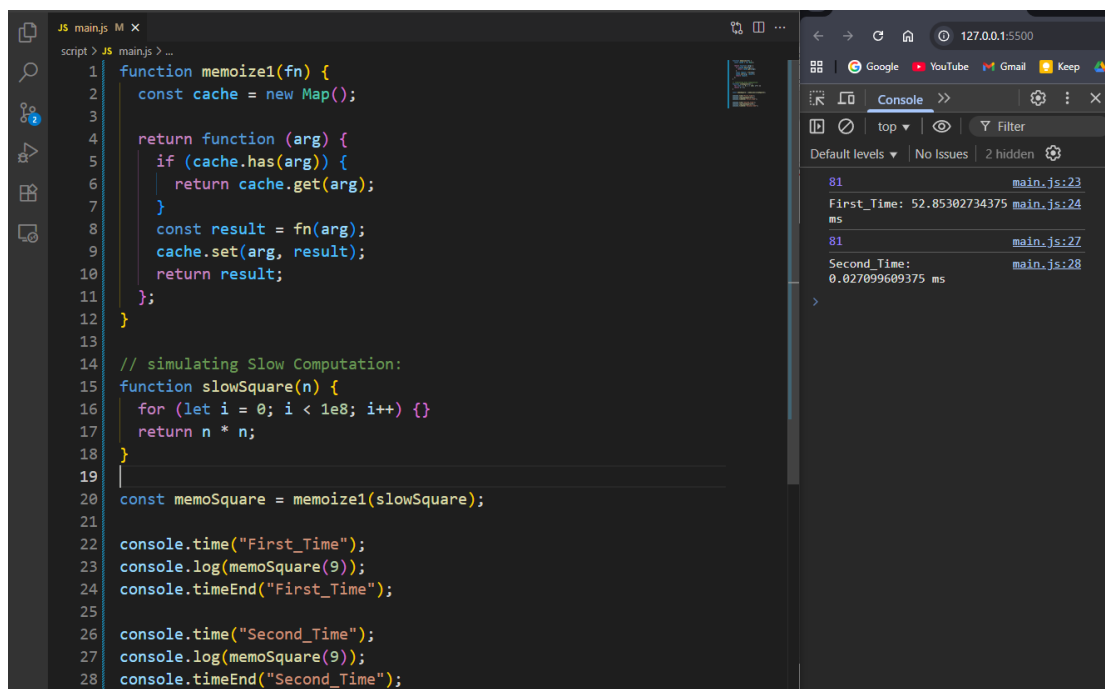


```
1 function makeAdder(start = 0) {
2   let total = start;
3
4   return function (num) {
5     return total + num;
6   };
7 }
8
9 let AdderA = makeAdder(0);
10
11 console.log(AdderA(5));
12 console.log(AdderA(10));
13 console.log(AdderA(20));
14
```

The browser console shows the following output:

Value	File
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).

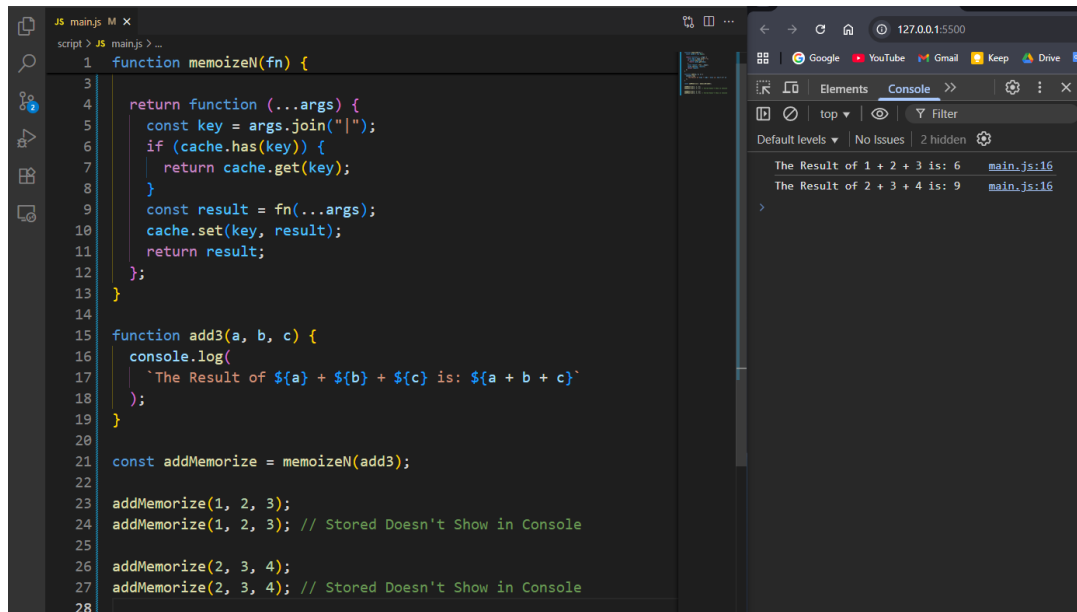


```
1 function memoize1(fn) {
2   const cache = new Map();
3
4   return function (arg) {
5     if (cache.has(arg)) {
6       return cache.get(arg);
7     }
8     const result = fn(arg);
9     cache.set(arg, result);
10    return result;
11  };
12 }
13
14 // simulating Slow Computation:
15 function slowSquare(n) {
16   for (let i = 0; i < 1e8; i++) {}
17   return n * n;
18 }
19
20 const memoSquare = memoize1(slowSquare);
21
22 console.time("First_Time");
23 console.log(memoSquare(9));
24 console.timeEnd("First_Time");
25
26 console.time("Second_Time");
27 console.log(memoSquare(9));
28 console.timeEnd("Second_Time");
```

The browser console shows the following output:

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

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

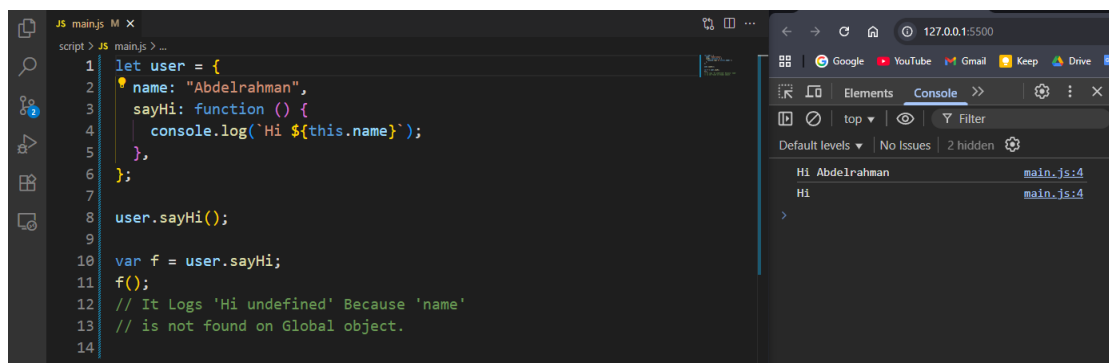


```
1 function memoizeN(fn) {
2
3   return function (...args) {
4     const key = args.join("|");
5     if (cache.has(key)) {
6       return cache.get(key);
7     }
8     const result = fn(...args);
9     cache.set(key, result);
10    return result;
11  };
12 }
13
14
15 function add3(a, b, c) {
16   console.log(
17     `The Result of ${a} + ${b} + ${c} is: ${a + b + c}`
18   );
19 }
20
21 const addMemoize = memoizeN(add3);
22
23 addMemoize(1, 2, 3);
24 addMemoize(1, 2, 3); // Stored Doesn't Show in Console
25
26 addMemoize(2, 3, 4);
27 addMemoize(2, 3, 4); // Stored Doesn't Show in Console
28
```

The browser console shows the following output:

```
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.

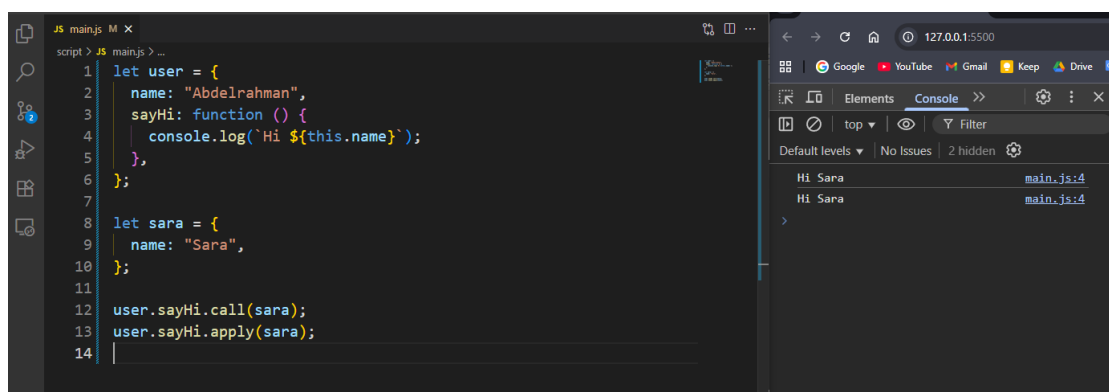


```
1 let user = {
2   name: "Abdelrahman",
3   sayHi: function () {
4     console.log(`Hi ${this.name}`);
5   },
6 };
7
8 user.sayHi();
9
10 var f = user.sayHi;
11 f();
12 // It Logs 'Hi undefined' Because 'name'
13 // is not found on Global object.
14
```

The browser console shows the following output:

```
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.



```
1 let user = {
2   name: "Abdelrahman",
3   sayHi: function () {
4     console.log(`Hi ${this.name}`);
5   },
6 };
7
8 let sara = {
9   name: "Sara",
10 };
11
12 user.sayHi.call(sara);
13 user.sayHi.apply(sara);
14
```

The browser console shows the following output:

```
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','!'.



The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines a 'greeter' object with a 'greet' function that logs a greeting and a sign. A 'person' object with 'name: "Ali"' is created. The 'greeter.greet' function is called with 'person' and 'Hello, !!'. Then, 'greeter.greet.apply' is used with 'person' and an array of arguments ['Hello', '!!']. The browser console on the right shows two log entries: 'Hello, Ali !!' from 'main.js:3'.

```
1 const greeter = {
2   greet: function (greeting, sign) {
3     console.log(`${greeting}, ${this.name} ${sign}`);
4   },
5 };
6
7 const person = { name: "Ali" };
8
9 greeter.greet.call(person, "Hello", "!!");
10 greeter.greet.apply(person, ["Hello", "!!"]);
11
```

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



The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines a 'greeter' object with a 'greet' function that logs a greeting and a sign. A 'person' object with 'name: "Lara"' is created. A 'greetLara' function is created by binding 'greeter.greet' to 'person' with 'Hello' as the sign. 'greetLara()' is called, and then 'greetLara' is re-bound with 'Hi' as the sign and called again. The browser console on the right shows two log entries: 'Hello, Lara' and 'Hi, Lara' from 'main.js:3'.

```
1 const greeter = {
2   greet: function (greeting) {
3     console.log(`${greeting}, ${this.name} `);
4   },
5 };
6
7 const person = { name: "Lara" };
8
9 let greetLara = greeter.greet.bind(person, "Hello");
10 greetLara();
11
12 greetLara = greeter.greet.bind(person, "Hi");
13 greetLara();
14
```

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

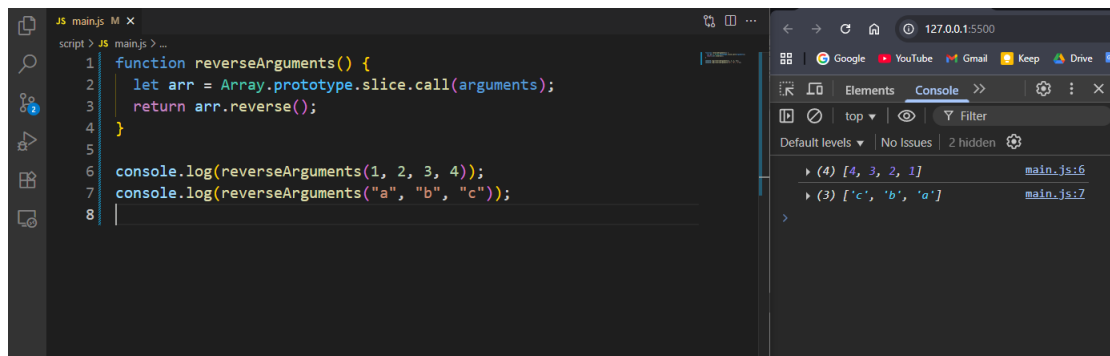


The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines a 'greeter' object with a 'greet' function that logs 'Hello' followed by a sign. A 'person' object with 'name: "Abdelrahman"' is created. 'greeter.greet.call' is used with 'person' and '!!'. The browser console on the right shows one log entry: 'Hello Abdelrahman !!' from 'main.js:3'.

```
1 const greeter = {
2   greet: function (sign) {
3     console.log('Hello ${this.name} ${sign}');
4   },
5 };
6
7 const person = { name: "Abdelrahman" };
8
9 greeter.greet.call(person, "!!");
10
```



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.



The screenshot shows a code editor with the following JavaScript code:

```
1 function reverseArguments() {  
2   let arr = Array.prototype.slice.call(arguments);  
3   return arr.reverse();  
4 }  
5  
6 console.log(reverseArguments(1, 2, 3, 4));  
7 console.log(reverseArguments("a", "b", "c"));  
8
```

The browser console on the right shows the output:

- Line 6: (4) [4, 3, 2, 1] main.js:6
- Line 7: (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.



The screenshot shows a code editor with the following JavaScript code:

```
1 let arr = [5, 2, 11, 7];  
2  
3 console.log(Math.max(...arr));  
4 console.log(Math.max.call(null, ...arr));  
5 console.log(Math.max.apply(null, arr));  
6 console.log("-----".repeat(5));  
7  
8 let mx = arr[0];  
9 for (let i = 1; i < arr.length; i++) {  
10   if (arr[i] > mx) {  
11     mx = arr[i];  
12   }  
13 }  
14 console.log(mx);  
15
```

The browser console on the right shows the output:

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

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



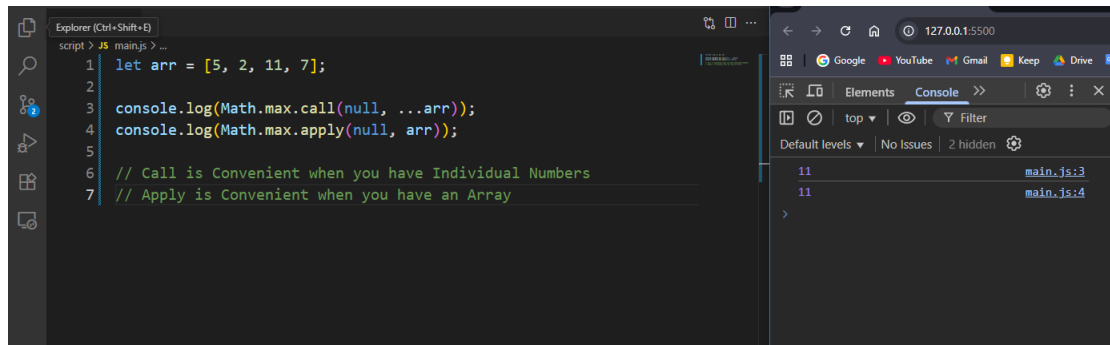
The screenshot shows a code editor with the following JavaScript code:

```
1 let arr = [5, 2, 11, 7];  
2  
3 console.log(Math.max.call(null, ...arr));  
4 console.log(Math.max.apply(null, arr));  
5
```

The browser console on the right shows the output:

- Line 3: 11 main.js:3
- Line 4: 11 main.js:4

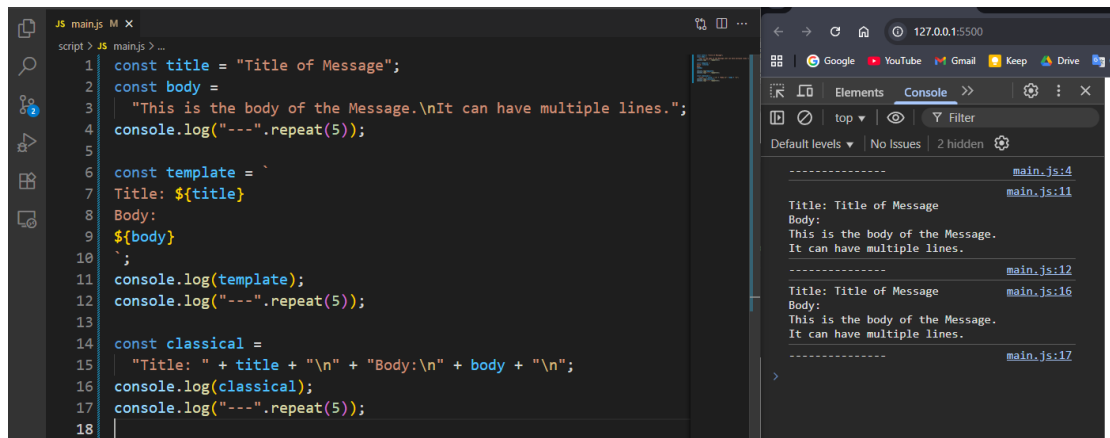
25. Convert string concatenation 'User: '+name+' Age:'+(age+1) into a template literal equivalent.



The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines an array `arr = [5, 2, 11, 7]` and logs the maximum value using `Math.max.call(null, ...arr)` and `Math.max.apply(null, arr)`. The browser console shows the output `11` for both methods.

```
1 let arr = [5, 2, 11, 7];
2
3 console.log(Math.max.call(null, ...arr));
4 console.log(Math.max.apply(null, arr));
5
6 // Call is Convenient when you have Individual Numbers
7 // Apply is Convenient when you have an Array
```

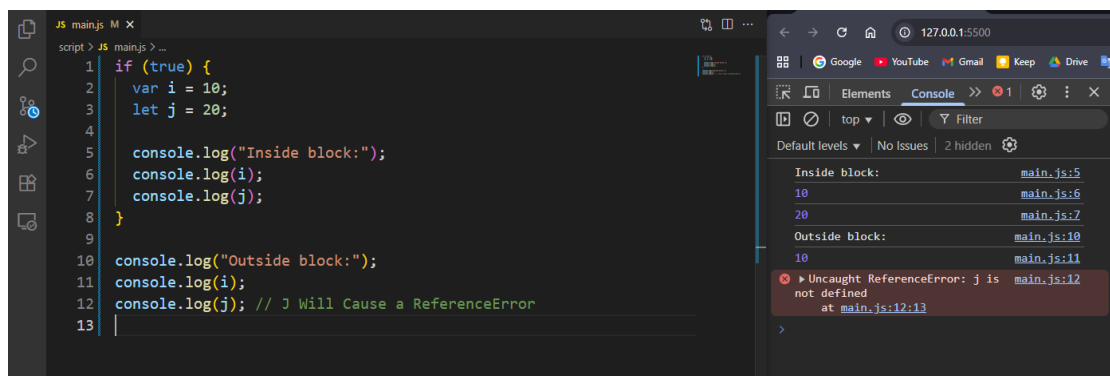
26. Create a multi-line template with variables title and body and log it; show classical \n build version for contrast.



The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines variables `title` and `body`, and logs them using a multi-line template literal and a classical string concatenation method. The browser console shows the output for both methods.

```
1 const title = "Title of Message";
2 const body = "This is the body of the Message.\nIt can have multiple lines.";
3 console.log("----".repeat(5));
4
5
6 const template = `
7 Title: ${title}
8 Body:
9 ${body}
10 `;
11 console.log(template);
12 console.log("----".repeat(5));
13
14 const classical =
15 "Title: " + title + "\n" + "Body:\n" + body + "\n";
16 console.log(classical);
17 console.log("----".repeat(5));
18
```

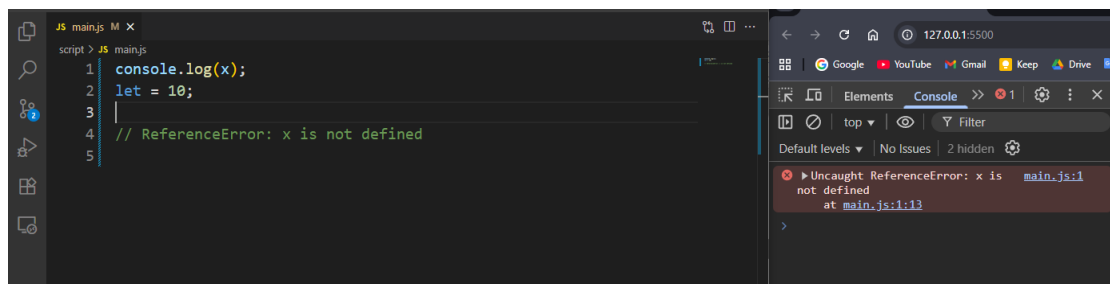
27. Write a block with var i and let j inside if(true) and log both inside and outside. Comment which leaks.



The screenshot shows a VS Code editor with a JavaScript file named 'main.js'. The code defines a block with `var i` and `let j` inside an `if(true)` block. The browser console shows the output for both methods, with a red error message indicating that `j` is not defined outside the block.

```
1 if (true) {
2   var i = 10;
3   let j = 20;
4
5   console.log("Inside block:");
6   console.log(i);
7   console.log(j);
8 }
9
10 console.log("Outside block:");
11 console.log(i);
12 console.log(j); // J Will Cause a ReferenceError
13
```

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



The screenshot shows a VS Code editor with a file named `main.js` containing the following code:

```
1 console.log(x);
2 let x = 5;
3
4 // ReferenceError: x is not defined
5
```

The browser console on the right shows an error: `Uncaught ReferenceError: x is 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).



The screenshot shows a VS Code editor with a file named `main.js` containing the following code:

```
1 const arr = [1, 2, 3];
2
3 arr.push(10);
4 arr.push(20);
5 arr.push(30);
6 console.log(arr);
7
8 // Error: Assignment to constant variable.
9 // arr = [100, 200, 300];
10 // console.log(arr);
11
```

The browser console on the right shows the output of the `console.log(arr)` statement: `(6) [1, 2, 3, 10, 20, 30]` at `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]`.



The screenshot shows a VS Code editor with a file named `main.js` containing the following code:

```
1 let square1 = (x) => {
2   return x * x;
3 };
4
5 let square2 = (x) => x * x;
6
7 let arr = [1, 2, 3];
8 let squares = arr.map((x) => x * x);
9 console.log(squares);
```

The browser console on the right shows the output of the `console.log(squares)` statement: `(3) [1, 4, 9]` at `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).

```

1 const timer = {
2   count: 0,
3
4   startClassicFunction: function () {
5     this.count = 0;
6     let id = setInterval(function () {
7       // this.count++; // Produce NaN Object
8       console.log('Classic Function: ${this.count}');
9     }, 1000);
10    setTimeout(() => clearInterval(id), 3000);
11  },
12
13  startArrowFunction: () => {
14    this.count = 0;
15    let id = setInterval(() => {
16      this.count++;
17      console.log('Arrow Function: ${this.count}');
18    }, 1000);
19    setTimeout(() => clearInterval(id), 3000);
20  },
21 };
22
23 // Run The Code
24 timer.startClassicFunction();
25
26 setTimeout(() => timer.startArrowFunction(), 5000);
27

```

Console output:

```

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.

```

1 // const objectCreator = () => {v: 10};
2 // Need Baranthes To Show That We Need To Return An Object
3
4 const objectCreator = () => ({ v: 10 });
5 console.log(objectCreator());
6

```

Console output:

```

{v: 10}    main.js:5

```

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

```

1 const user = {
2   name: "Abdelrahman",
3
4   // Arrow function here is a bad choice
5   greet1: () => {
6     console.log('Hi, I'm ${this.name}');
7   },
8
9   greet2: function () {
10    console.log('Hi, I'm ${this.name}');
11  },
12 };
13
14 user.greet1();
15 user.greet2();
16

```

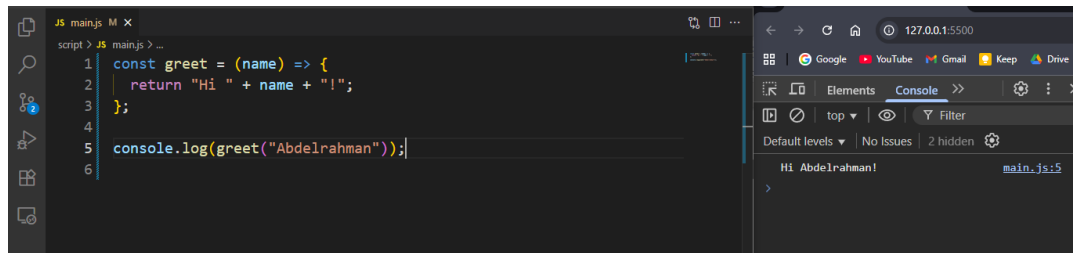
Console output:

```

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 habebay :).

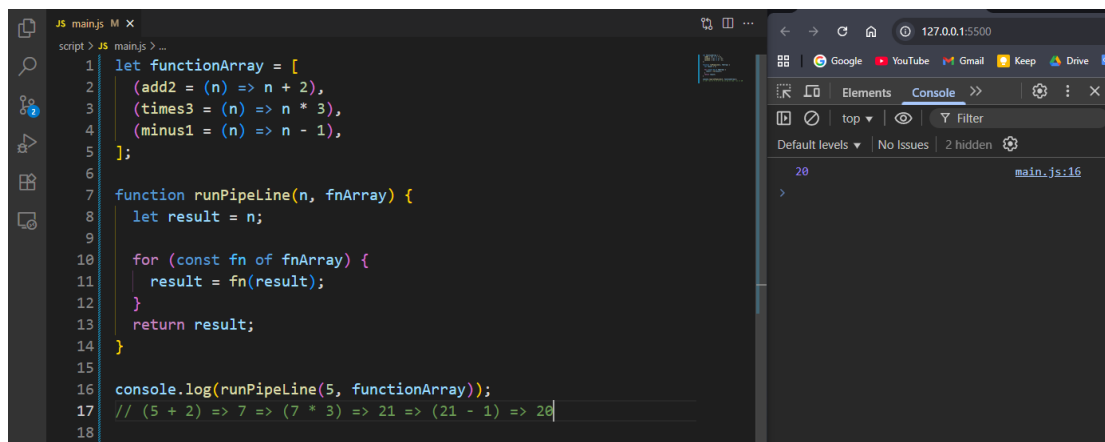


The screenshot shows a VS Code editor with a file named 'main.js'. The code is as follows:

```
1 const greet = (name) => {  
2   return "Hi " + name + "!";  
3 };  
4  
5 console.log(greet("Abdelrahman"));  
6
```

The browser console on the right shows the output: 'Hi Abdelrahman!'.

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]).



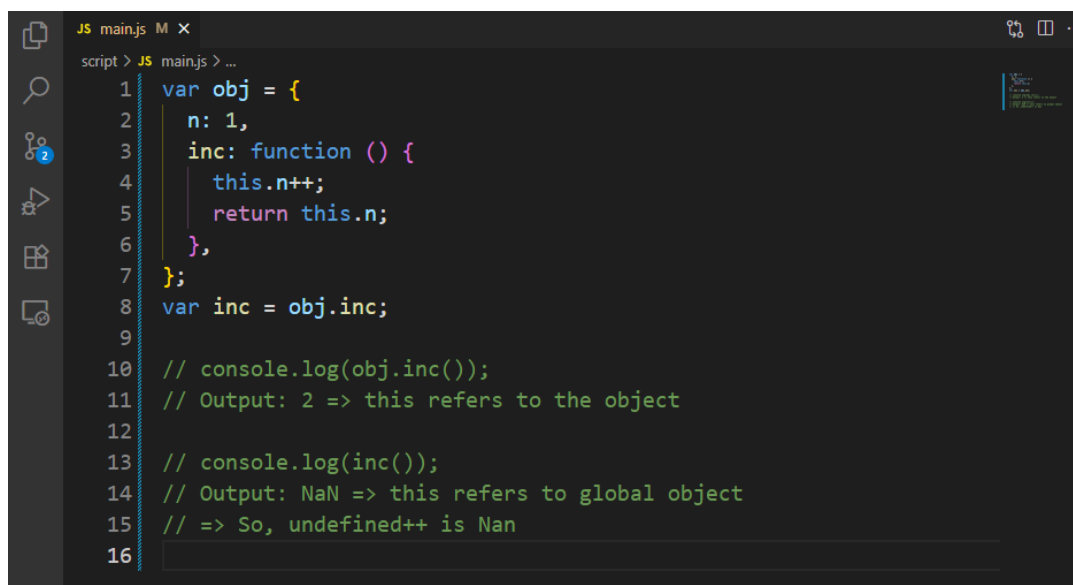
The screenshot shows a VS Code editor with a file named 'main.js'. The code is as follows:

```
1 let functionArray = [  
2   (add2 = (n) => n + 2),  
3   (times3 = (n) => n * 3),  
4   (minus1 = (n) => n - 1),  
5 ];  
6  
7 function runPipeline(n, fnArray) {  
8   let result = n;  
9  
10  for (const fn of fnArray) {  
11    result = fn(result);  
12  }  
13  return result;  
14 }  
15  
16 console.log(runPipeline(5, functionArray));  
17 // (5 + 2) => 7 => (7 * 3) => 21 => (21 - 1) => 20  
18
```

The browser console on the right shows the output: '20'.

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

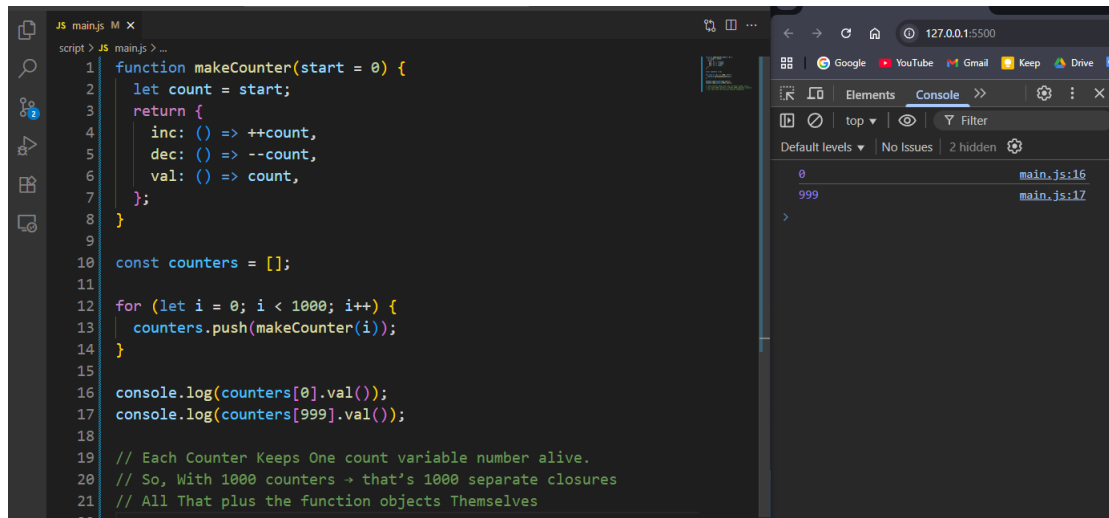
```
var obj = { n: 1, inc: function(){ this.n++; return this.n; } };  
var inc = obj.inc;  
console.log(obj.inc());  
console.log(inc());
```



The screenshot shows a VS Code editor with a file named 'main.js'. The code is as follows:

```
1 var obj = {  
2   n: 1,  
3   inc: function () {  
4     this.n++;  
5     return this.n;  
6   },  
7 };  
8 var inc = obj.inc;  
9  
10 // console.log(obj.inc());  
11 // Output: 2 => this refers to the object  
12  
13 // console.log(inc());  
14 // Output: NaN => this refers to global object  
15 // => So, undefined++ is Nan  
16
```

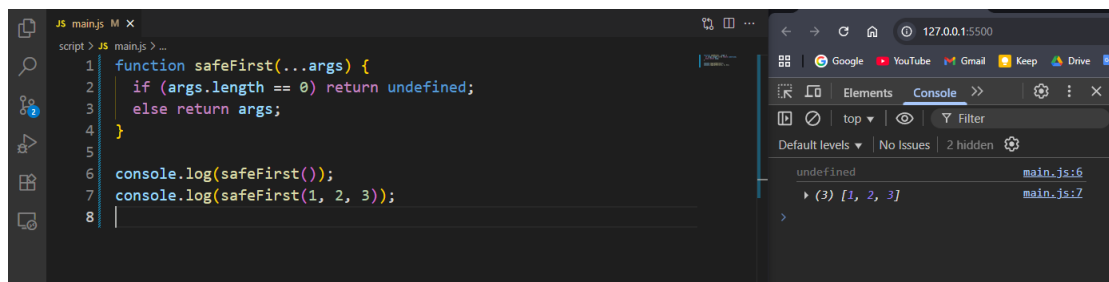
37. Create many counters in a loop (e.g. 1000) and store them in an array. Comment on potential memory considerations of large closure arrays.



```
1 function makeCounter(start = 0) {
2   let count = start;
3   return {
4     inc: () => ++count,
5     dec: () => --count,
6     val: () => count,
7   };
8 }
9
10 const counters = [];
11
12 for (let i = 0; i < 1000; i++) {
13   counters.push(makeCounter(i));
14 }
15
16 console.log(counters[0].val());
17 console.log(counters[999].val());
18
19 // Each Counter Keeps One count variable number alive.
20 // So, With 1000 counters -> that's 1000 separate closures
21 // All That plus the function objects Themselves
22
```

The console output shows the first counter value is 0 (main.js:16) and the last counter value is 999 (main.js:17).

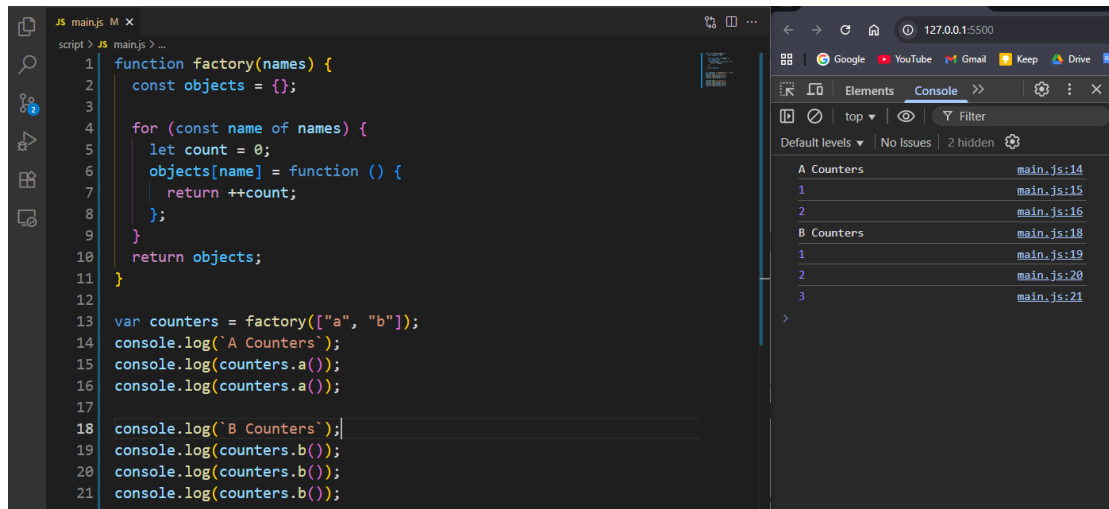
38. Write safeFirst() that returns undefined if called with zero args else return array of the args.



```
1 function safeFirst(...args) {
2   if (args.length == 0) return undefined;
3   else return args;
4 }
5
6 console.log(safeFirst());
7 console.log(safeFirst(1, 2, 3));
8
```

The console output shows undefined (main.js:6) for the first call and (3) [1, 2, 3] (main.js:7) for the second call.

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



```
1 function factory(names) {
2   const objects = {};
3
4   for (const name of names) {
5     let count = 0;
6     objects[name] = function () {
7       return ++count;
8     };
9   }
10  return objects;
11 }
12
13 var counters = factory(["a", "b"]);
14 console.log('A Counters');
15 console.log(counters.a());
16 console.log(counters.a());
17
18 console.log('B Counters');
19 console.log(counters.b());
20 console.log(counters.b());
21 console.log(counters.b());
```

The screenshot shows a code editor with the above JavaScript code. The right-hand pane displays the browser's console output, which is as follows:

Log Entry	Source
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).



&

