Function:

A JavaScript function is a block of code designed to perform a particular task. A JavaScript function is executed when "something" invokes it (calls it). A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses (). Function names can contain letters, digits, underscores, and dollar signs (same rules as variables). The parentheses may include parameter names separated by commas: (parameter1, parameter2, ...) The code to be executed, by the function, is placed inside curly brackets: {}

Function parameters are listed inside the parentheses () in the function definition. Function arguments are the values received by the function when it is invoked. Inside the function, the arguments (the parameters) behave as local variables.

Function Invocation:

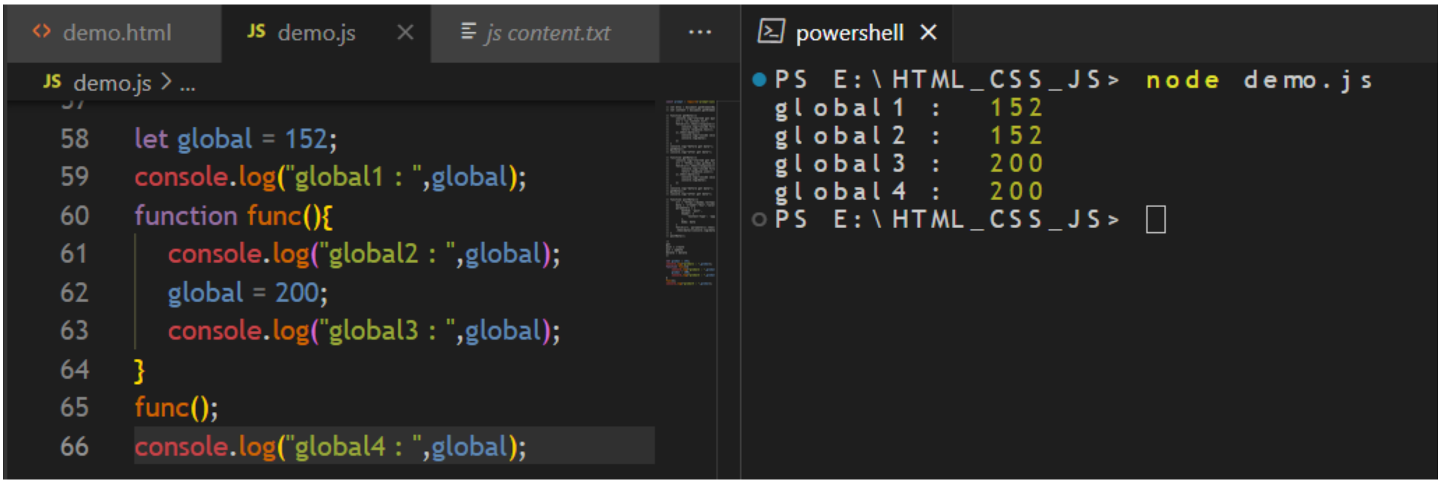
The code inside the function will execute when "something" invokes (calls) the function: When an event occurs (when a user clicks a button) When it is invoked (called) from JavaScript code Automatically (self-invoked).

Function Return:

When JavaScript reaches a return statement, the function will stop executing. If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement. Functions often compute a return value. The return value is "returned" back to the "caller":

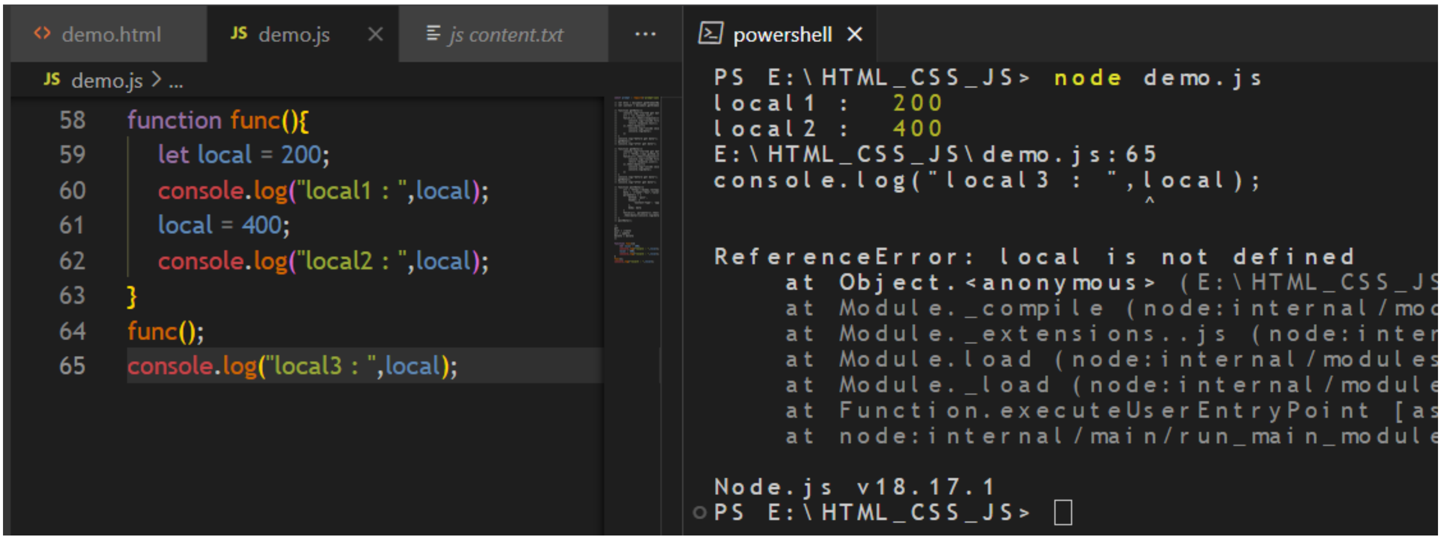
Local Variable & Global Variable:

Variables declared within a JavaScript function, become LOCAL to the function. Local variables can only be accessed from within the function. Since local variables are only recognized inside their functions, variables with the same name can be used in different functions. Local variables are created when a function starts, and deleted when the function is completed. A JavaScript global variable is declared outside the function or declared with window object. It can be accessed from any function.



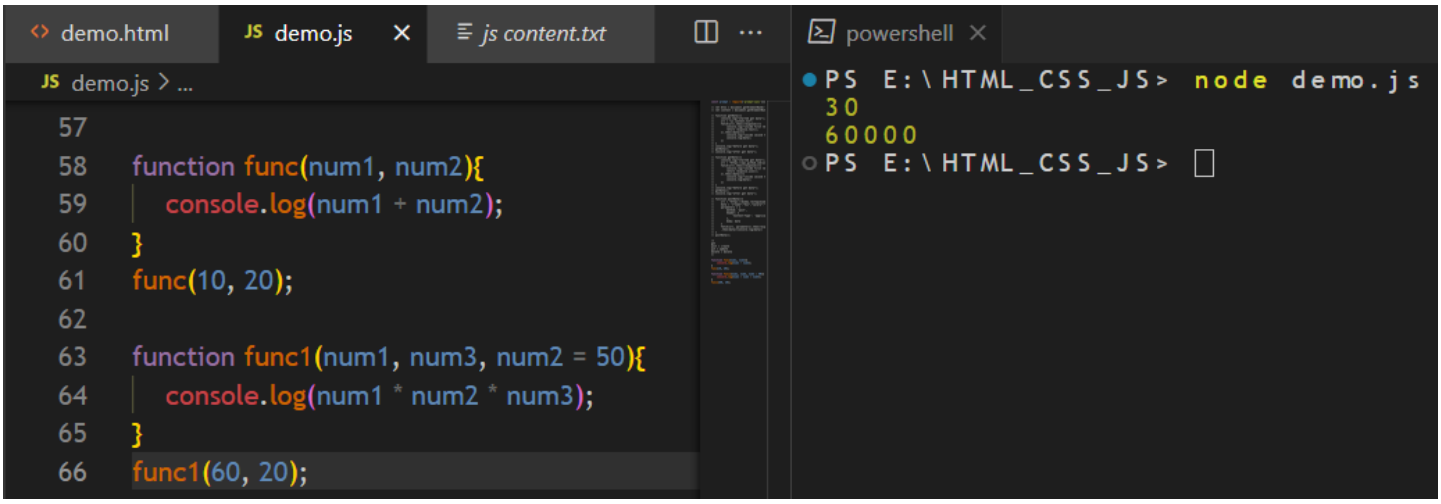
```javascript
let global = 152;
console.log("global1 : ",global);
function func(){
    console.log("global2 : ",global);
    global = 200;
    console.log("global3 : ",global);
}
func();
console.log("global4 : ",global);
```

```
PS E:\HTML_CSS_JS> node demo.js
global1 :  152
global2 :  152
global3 :  200
global4 :  200
PS E:\HTML_CSS_JS>
```



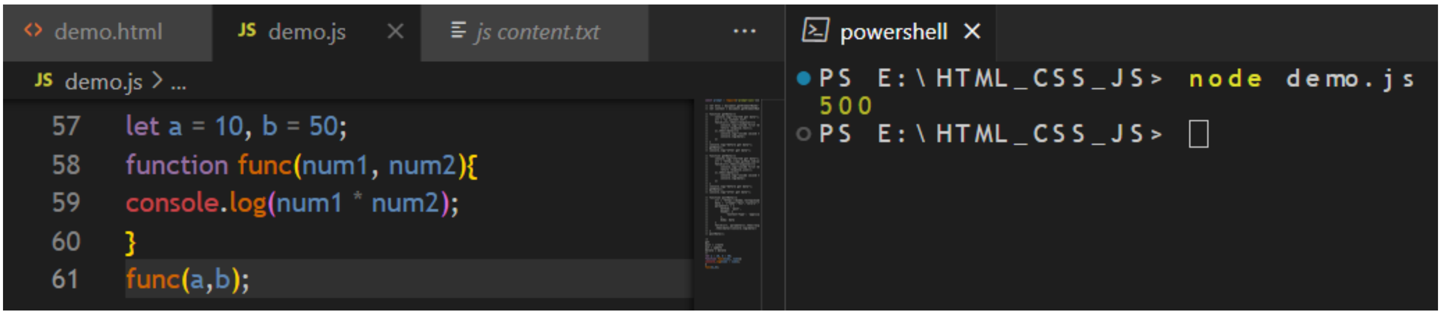
```js
function func(){
    let local = 200;
    console.log("local1 : ",local);
    local = 400;
    console.log("local2 : ",local);
}
func();
console.log("local3 : ",local);
```

```
PS E:\HTML_CSS_JS> node demo.js
local1 :  200
local2 :  400
E:\HTML_CSS_JS\demo.js:65
console.log("local3 : ",local);
                         ^

ReferenceError: local is not defined
    at Object.<anonymous> (E:\HTML_CSS_JS
    at Module._compile (node:internal/mo
    at Module._extensions..js (node:inter
    at Module.load (node:internal/modules
    at Module._load (node:internal/module
    at Function.executeUserEntryPoint [as
    at node:internal/main/run_main_module

Node.js v18.17.1
PS E:\HTML_CSS_JS>
```



```js

function func(num1, num2){
    console.log(num1 + num2);
}
func(10, 20);

function func1(num1, num3, num2 = 50){
    console.log(num1 * num2 * num3);
}
func1(60, 20);
```

```
PS E:\HTML_CSS_JS> node demo.js
30
60000
PS E:\HTML_CSS_JS>
```



```js
let a = 10, b = 50;
function func(num1, num2){
console.log(num1 * num2);
}
func(a,b);
```

```
PS E:\HTML_CSS_JS> node demo.js
500
PS E:\HTML_CSS_JS>
```

Block Scope:

Before ES6 (2015), JavaScript had only Global Scope and Function Scope. ES6 introduced two important new JavaScript keywords: let and const. These two keywords provide Block Scope in JavaScript. Variables declared inside a { } block cannot be accessed from outside the block: Variables declared with the var keyword can NOT have block scope. Variables declared inside a { } block can be accessed from outside the block.

Local Scope:

Variables declared within a JavaScript function, become LOCAL to the function. Local variables have Function Scope: They can only be accessed from within the function. Since local variables are only recognized inside their functions, variables with the same name can be used in different functions. Local variables are created when a function starts, and deleted when the function is completed.

Function Scope:

JavaScript has function scope: Each function creates a new scope. Variables defined inside a function are not accessible (visible) from outside the function. Variables declared with var, let and const are quite similar when declared inside a function. They all have Function Scope:

Global Scope:

A variable declared outside a function, becomes GLOBAL. A global variable has Global Scope: All scripts and functions on a web page can access it. Variables declared Globally (outside any function) have Global Scope. Global variables can be accessed from anywhere in a JavaScript program. Variables declared with var, let and const are quite similar when declared outside a block. They all have Global Scope:

JavaScript Variables:

In JavaScript, objects and functions are also variables. Scope determines the accessibility of variables, objects, and functions from different parts of the code.

Automatically Global:

If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable. This code example will declare a global variable carName, even if the value is assigned inside a function.

Strict Mode:

All modern browsers support running JavaScript in "Strict Mode". You will learn more about how to use strict mode in a later chapter of this tutorial. In "Strict Mode", undeclared variables are not automatically global.

Global Variables In HTML:

With JavaScript, the global scope is the JavaScript environment. In HTML, the global scope is the window object. Global variables defined with the var keyword belong to the window object: Global variables defined with the let keyword do not belong to the window object:

[Do NOT create global variables unless you intend to. Your global variables (or functions) can overwrite window variables (or functions). Any function, including the window object, can overwrite your global variables and functions.]

Lifetime of JS Variable:

The lifetime of a JavaScript variable starts when it is declared. Function (local) variables are deleted when the function is completed. In a web browser, global variables are deleted when you close the browser window (or tab). Function arguments (parameters) work as local variables inside functions.

Difference Between Let, Const, Var:

1. Scope:

`var`: Function-scoped; variables declared with `var` are function-scoped or globally scoped if declared outside a function. `let` and `const`: Block-scoped; variables declared with `let` and `const` are only accessible within the block they are defined in, such as a function, loop, or conditional statement.

2. Hoisting:

`var` declarations are hoisted to the top of their function or global context, which means you can use a `var` variable before it's declared. `let` and `const` are hoisted as well but not initialized, so you'll get a ReferenceError if you try to use them before declaration.

3. Reassignment:

`var` and `let` variables can be reassigned after declaration. `const` variables cannot be reassigned; they are read-only after declaration.

4. Temporal Dead Zone (TDZ):

`let` and `const` have a TDZ, a period where they are declared but not yet initialized. Accessing them during the TDZ results in a ReferenceError.

5. Global Object Property:

Variables declared with `var` at the global level become properties of the global object (e.g., `window` in a browser). This can lead to unexpected behavior. Variables declared with `let` and `const` at the global level do not become properties of the global object.

6. Redeclaration:

`var` allows variable redeclaration within the same scope. `let` and `const` do not allow redeclaration in the same scope.

7. Usage in Loops:

When using `var` in a loop, there can be unintended issues due to variable hoisting and scope. `let` and `const` are generally safer to use in loops as they have block scope.

8. Functional Scoping:

Variables declared with `var` are function-scoped, which means they are available throughout the entire function. `let` and `const` have block scope, which limits their availability to the block they are declared in.

9. Default Value:

Variables declared with `var` are initialized with `undefined` by default if not assigned a value. `let` and `const` variables are not automatically initialized, and using them before assignment results in a ReferenceError.

10. Use Case:

`var` is rarely used in modern JavaScript due to its issues with scoping and hoisting. It's mostly used in older codebases. `let` is used for variables that need to be reassigned or have block-level scope. `const` is used for variables that should not be reassigned and have block-level scope. It's preferred for constants and values that shouldn't change.

In modern JavaScript, it's generally recommended to use `let` and `const` over `var` to improve code predictability and maintainability. The choice between `let` and `const` depends on whether a variable should be reassigned or not.

Difference Between this.variable and Window.variable:

1. Scope:

`this.variable_name` typically refers to a variable within the context of an object or a function, depending on the current execution context. `window.variable_name` refers to a variable in the global scope and is attached to the `window` object in a browser environment.

2. Context:

`this.variable_name` is context-dependent and can change based on the context in which it is used. It often refers to an object or instance. `window.variable_name` is always in the global context and refers to a global variable.

3. Implicit Object Binding:

`this.variable_name` is often used with object methods to access properties and values within the object. It's bound to the object the method is called on. `window.variable_name` is a direct reference to a global variable in the browser's global object.

4. Function Execution:

Within a function, `this.variable_name` refers to a variable within the object or instance the function was called on. `window.variable_name` is available globally within the function but might not necessarily be defined there.

5. Variable Declaration:

`this.variable_name` assumes that `variable_name` is already declared as a property of the object or function's context. `window.variable_name` directly accesses a global variable, and it doesn't need to be declared with `var`, `let`, or `const`.

6. Global Namespace:

`this.variable_name` doesn't add variables to the global namespace; they are limited to the object's scope or function's context. `window.variable_name` adds variables to the global namespace, which can lead to naming conflicts and pollution.

7. Use in Functions:

`this.variable_name` is often used within object methods to access or modify properties of the object. `window.variable_name` can be accessed and modified within functions but is generally avoided to prevent global namespace pollution.

8. Strict Mode:

In strict mode, using `this.variable_name` without a defined object context may result in an error. `window.variable_name` is still accessible in strict mode, but it's considered good practice to avoid it.

9. Dynamic Binding:

The value of `this.variable_name` can change dynamically depending on how a function is called, making it useful for methods in object-oriented programming. `window.variable_name` remains a constant reference to the global scope and doesn't change based on function calls.

10. Global Object:

`this` within a global context, outside of any function or object, usually refers to the `window` object in a browser environment. `window.variable_name` is a direct reference to a property of the `window` object.

In summary, `this.variable_name` is typically used to access object properties or function context variables, while `window.variable_name` directly accesses global variables in the browser's global scope. Care should be taken when using global variables to avoid naming conflicts and unintended side effects.

Difference Between this & window Keyword:

1. Scope:

`this`: The value of `this` depends on the context in which it's used. It can vary within different parts of your code, like inside functions, object methods, or event handlers. It usually refers to the object that is currently executing the code.

`window`: The `window` object is the global object in a web browser's environment. It's always accessible globally and contains properties and methods related to the browser's window, such as the `document`, `location`, and `console` objects.

2. Context:

`this`: The value of `this` is dynamic and depends on how a function or method is called. It is often used to access or modify properties of the object or context in which it's invoked.

`window`: `window` is a fixed reference to the global object. It represents the global context in a browser and doesn't change based on function calls or object methods.

3. Object Association:

\`this\`: It is often used within methods of objects to access properties and values of that object. In this context, \`this\` refers to the object itself.

\`window\`: \`window\` is not associated with an object but is a top-level object containing all global variables and functions.

4. Global Object:

\`this\`: In the global context (outside of any function or object), \`this\` typically refers to the \`window\` object in a browser environment. \`window\`: It is always a reference to the global object.

5. Use in Functions:

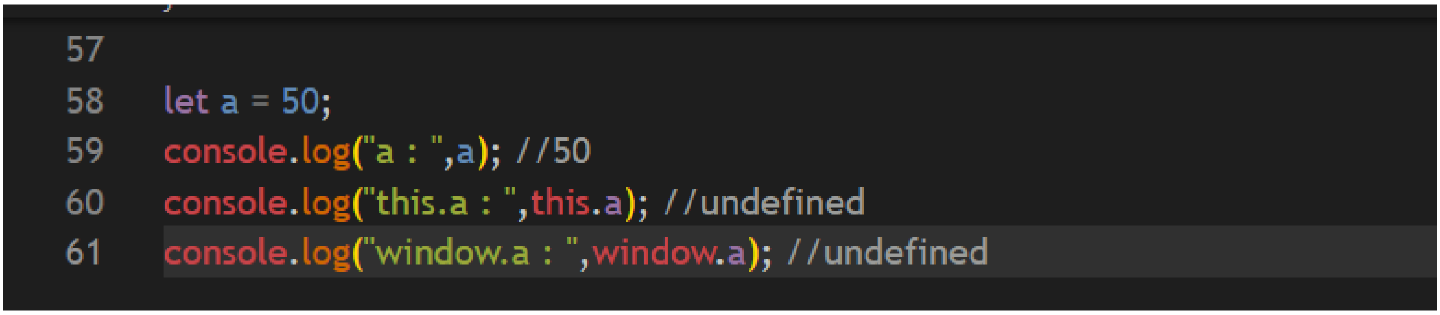
\`this\`: Used within functions and object methods to access the context in which the function is executed. It can be particularly useful in object-oriented programming for method chaining.

\`window\`: Can be accessed and modified within functions, but it's not a recommended practice to modify global variables directly to avoid naming conflicts and pollution of the global namespace.

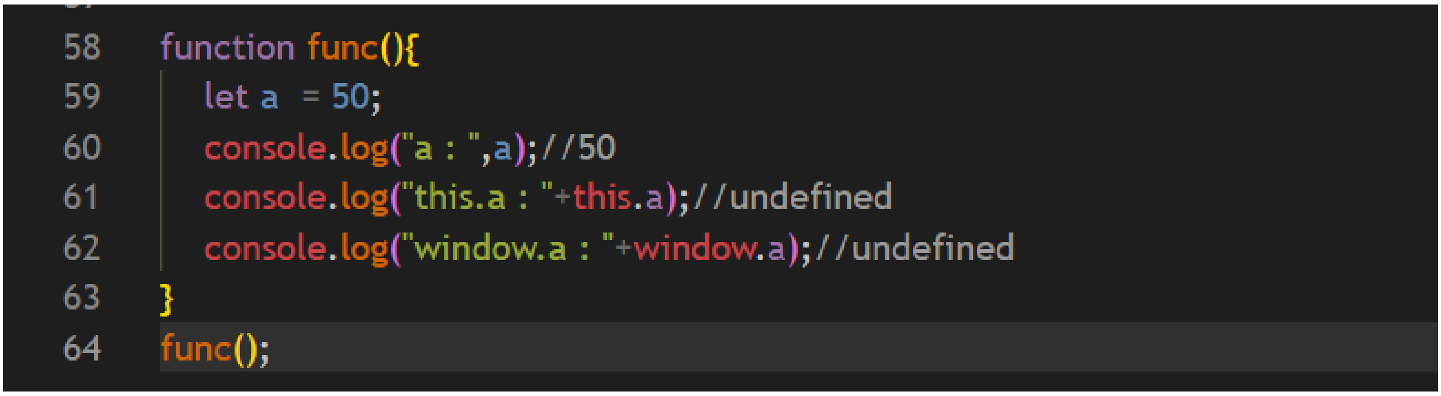
6. Strict Mode:

In strict mode, \`this\` behaves differently and is often \`undefined\` in the global context. This is to prevent potential issues and encourage better coding practices. \`window\` is still accessible in strict mode.

In summary, \`this\` is a dynamic reference to the current execution context, making it useful for working with objects and methods, while \`window\` is a fixed reference to the global object, primarily used for browser-related operations and accessing global variables and functions.

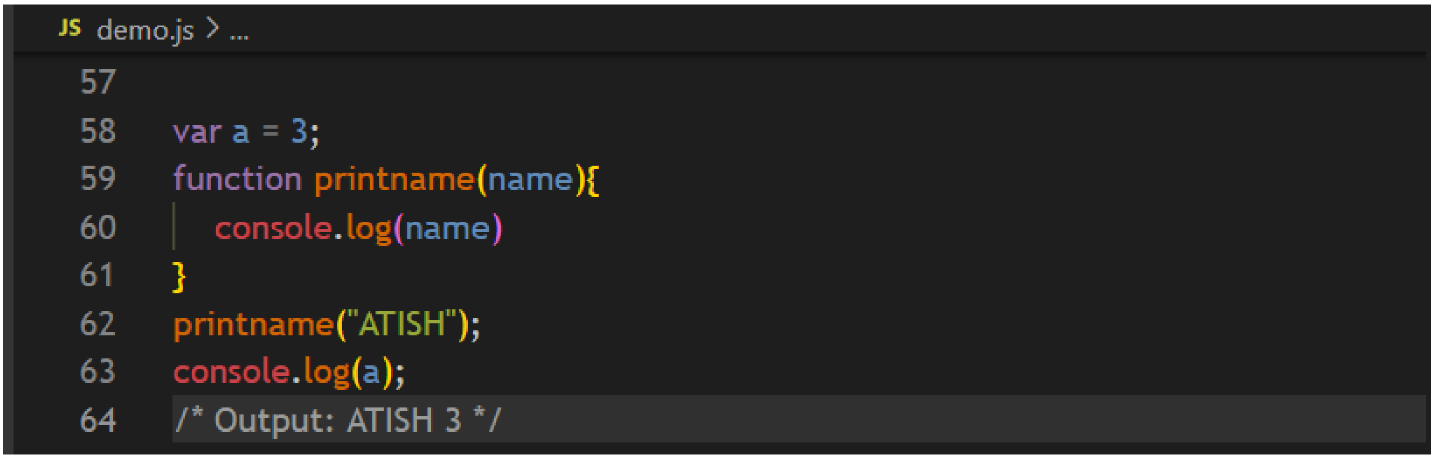


```

let a = 50;
console.log("a : ",a); //50
console.log("this.a : ",this.a); //undefined
console.log("window.a : ",window.a); //undefined
```

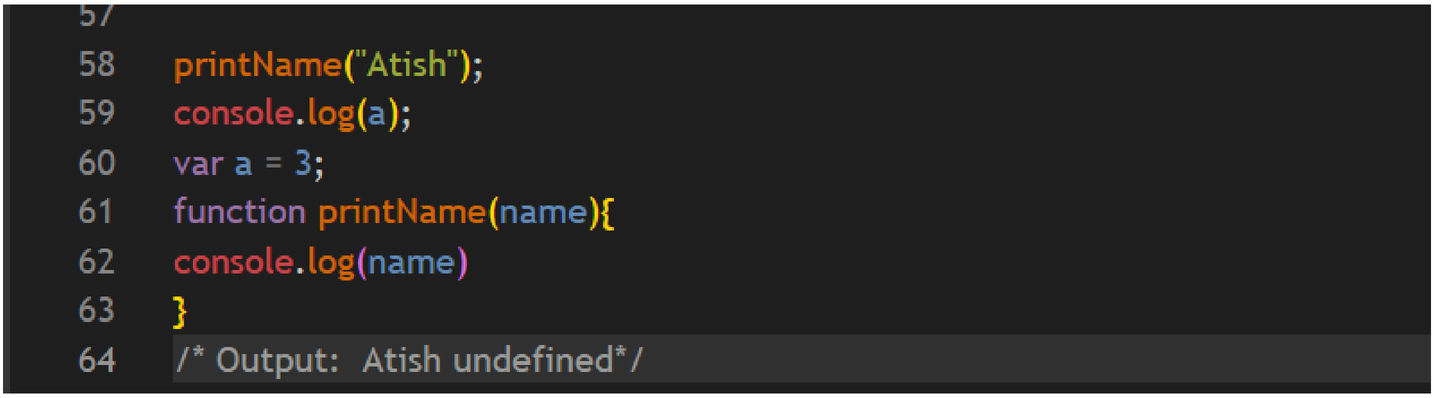


```
function func(){
    let a  = 50;
    console.log("a : ",a);//50
    console.log("this.a : "+this.a);//undefined
    console.log("window.a : "+window.a);//undefined
}
func();
```

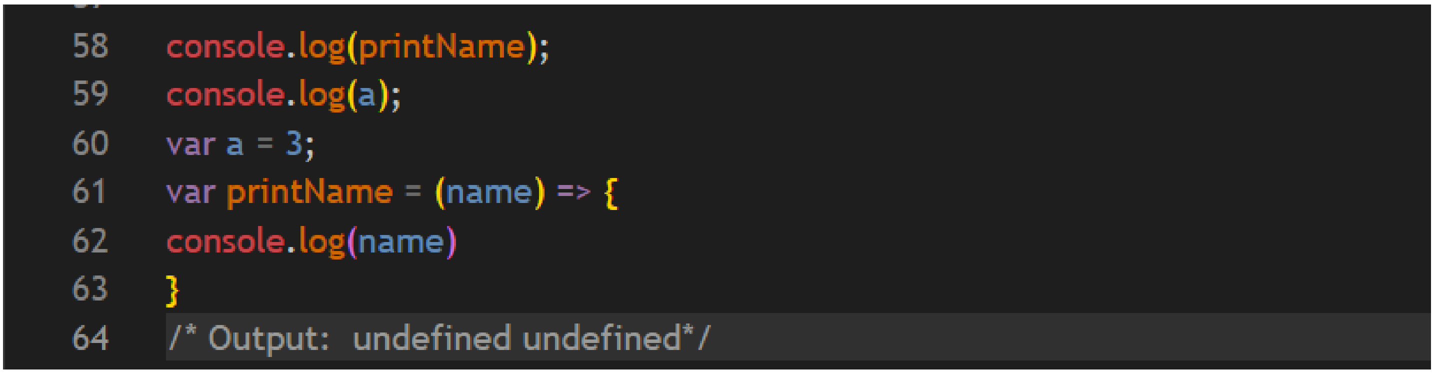
Sharpener MCQ:



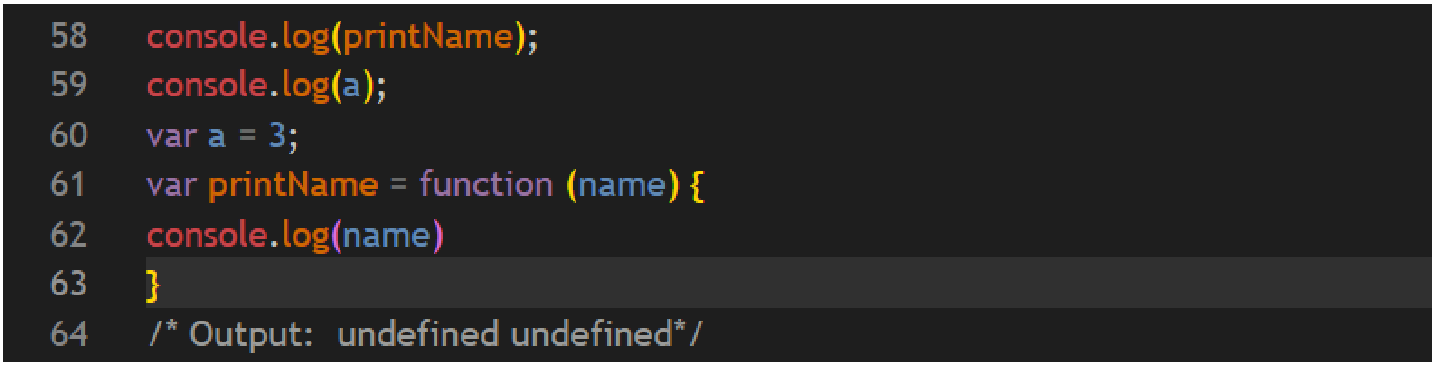
```
var a = 3;
function printname(name){
    console.log(name)
}
printname("ATISH");
console.log(a);
/* Output: ATISH 3 */
```



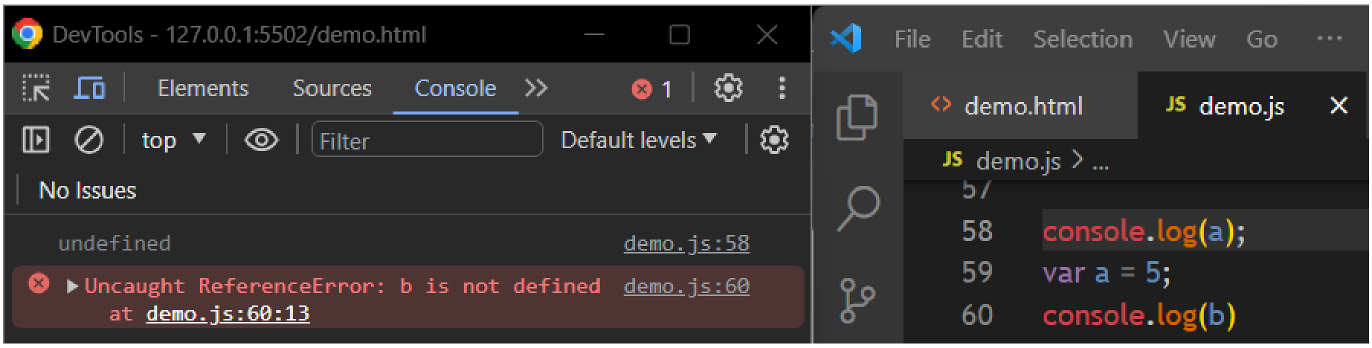
```
printName("Atish");
console.log(a);
var a = 3;
function printName(name){
console.log(name)
}
/* Output:  Atish undefined*/
```



```
console.log(printName);
console.log(a);
var a = 3;
var printName = (name) => {
console.log(name)
}
/* Output:  undefined undefined*/
```

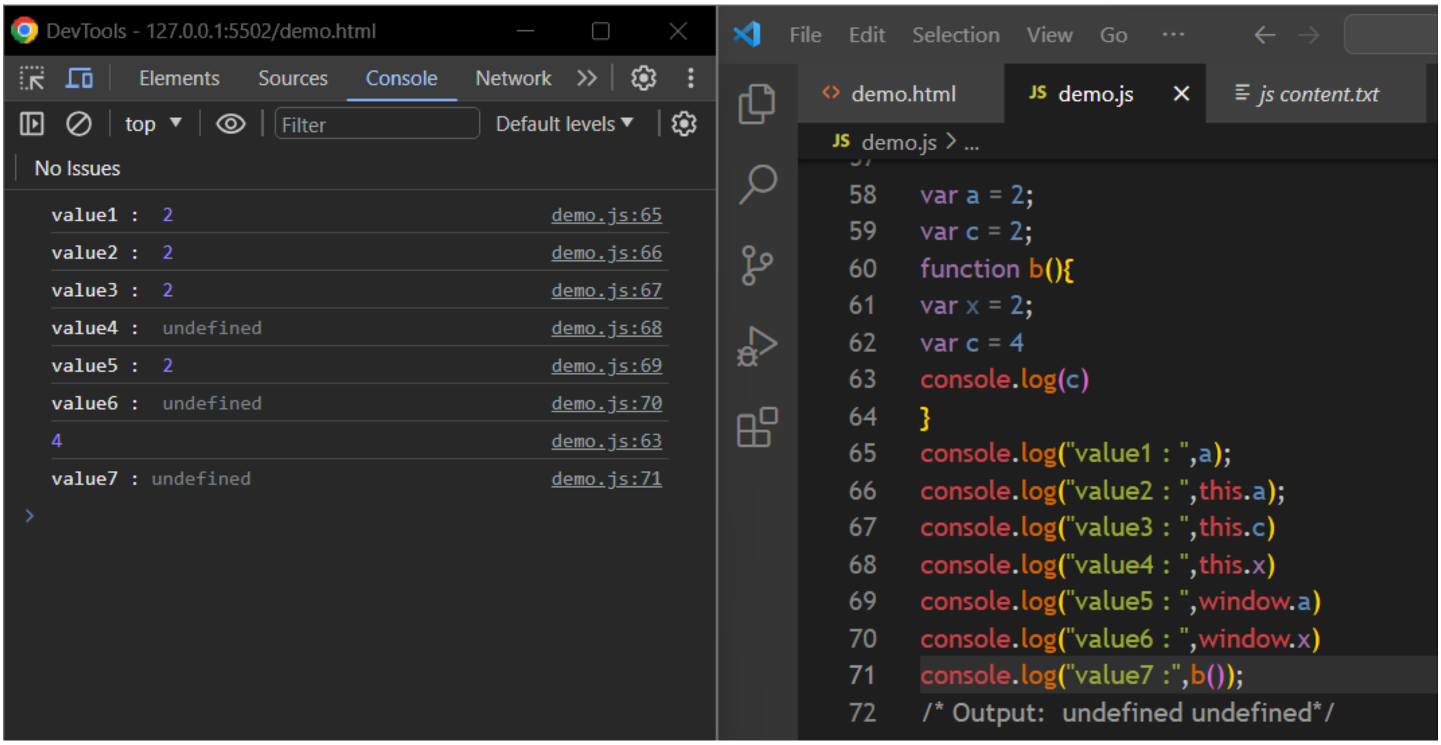


```
console.log(printName);
console.log(a);
var a = 3;
var printName = function (name) {
console.log(name)
}
/* Output:  undefined undefined*/
```



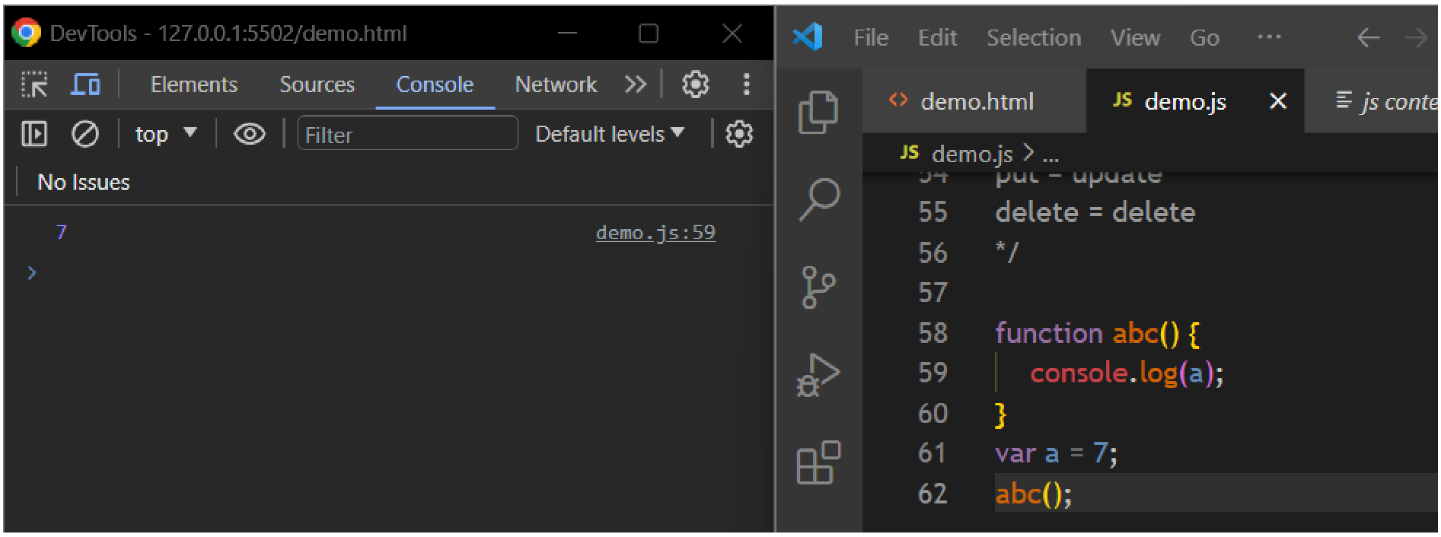
```
undefined                                          demo.js:58
Uncaught ReferenceError: b is not defined          demo.js:60
    at demo.js:60:13
```

```
console.log(a);
var a = 5;
console.log(b)
```



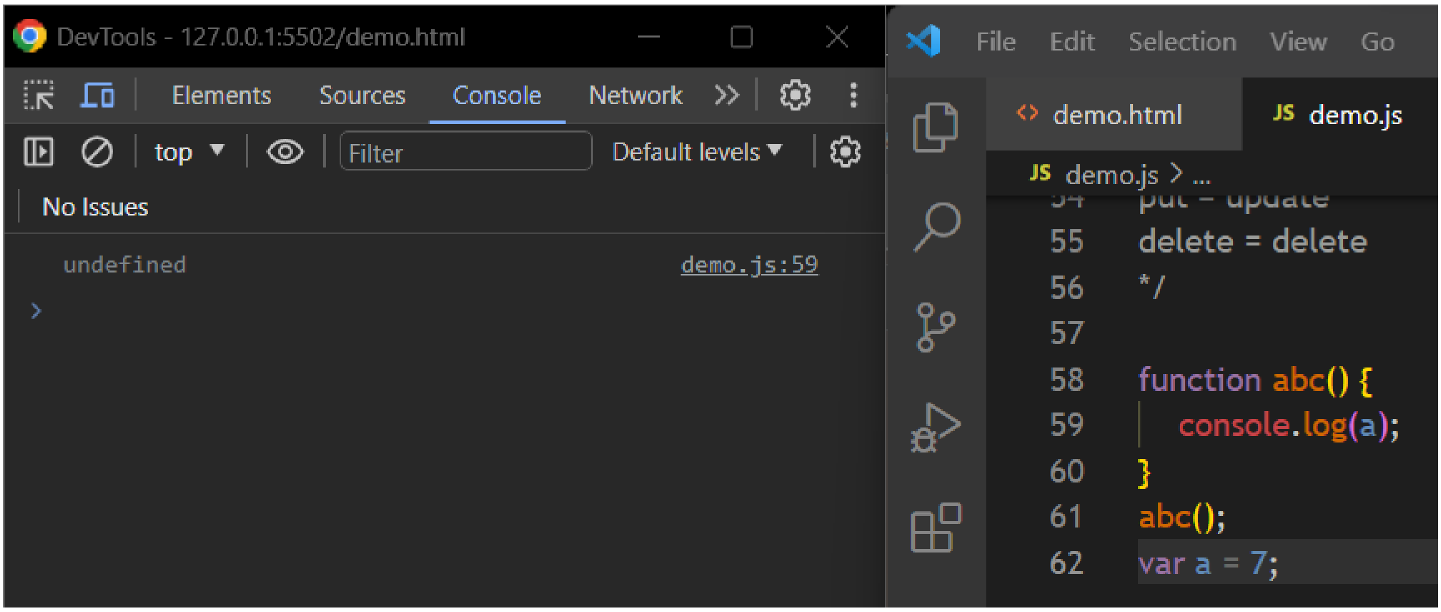
```
value1 :  2                                  demo.js:65
value2 :  2                                  demo.js:66
value3 :  2                                  demo.js:67
value4 :  undefined                          demo.js:68
value5 :  2                                  demo.js:69
value6 :  undefined                          demo.js:70
4                                            demo.js:63
value7 : undefined                           demo.js:71
>
```

```
58  var a = 2;
59  var c = 2;
60  function b(){
61  var x = 2;
62  var c = 4
63  console.log(c)
64  }
65  console.log("value1 : ",a);
66  console.log("value2 : ",this.a);
67  console.log("value3 : ",this.c)
68  console.log("value4 : ",this.x)
69  console.log("value5 : ",window.a)
70  console.log("value6 : ",window.x)
71  console.log("value7 :",b());
72  /* Output:  undefined undefined*/
```



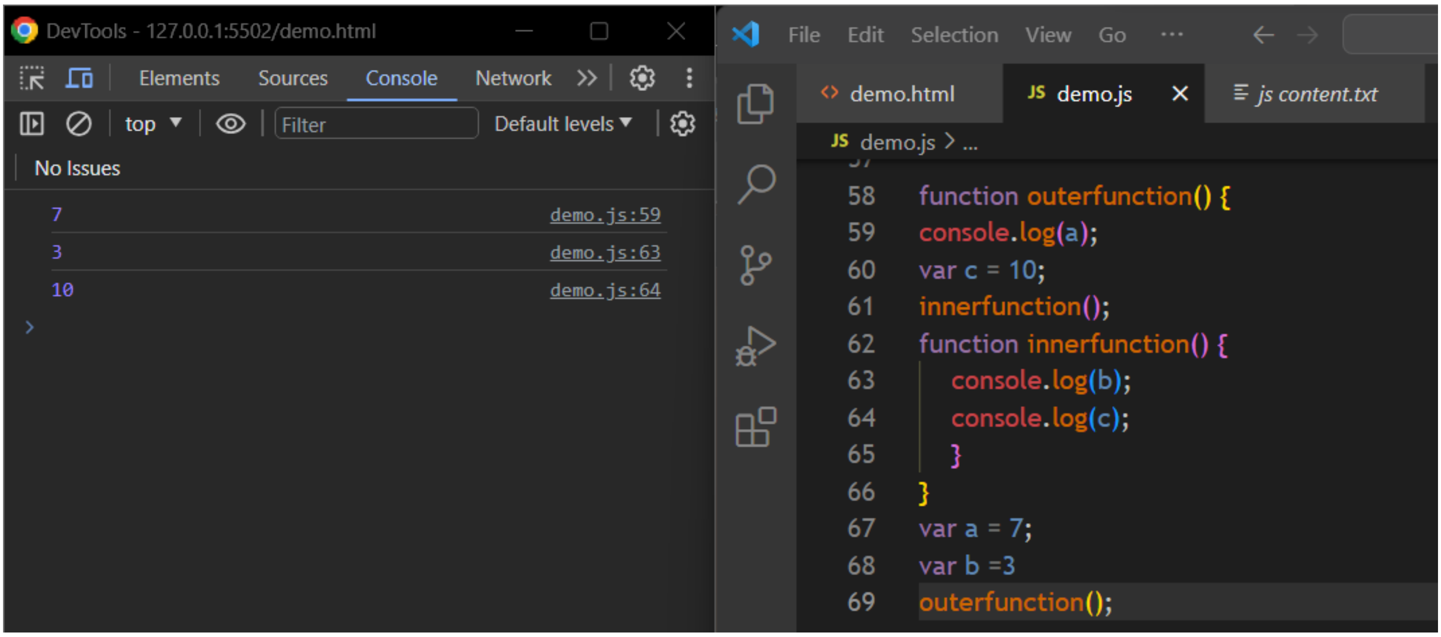
```
7                                            demo.js:59
>
```

```
54  put = update
55  delete = delete
56  */
57
58  function abc() {
59      console.log(a);
60  }
61  var a = 7;
62  abc();
```



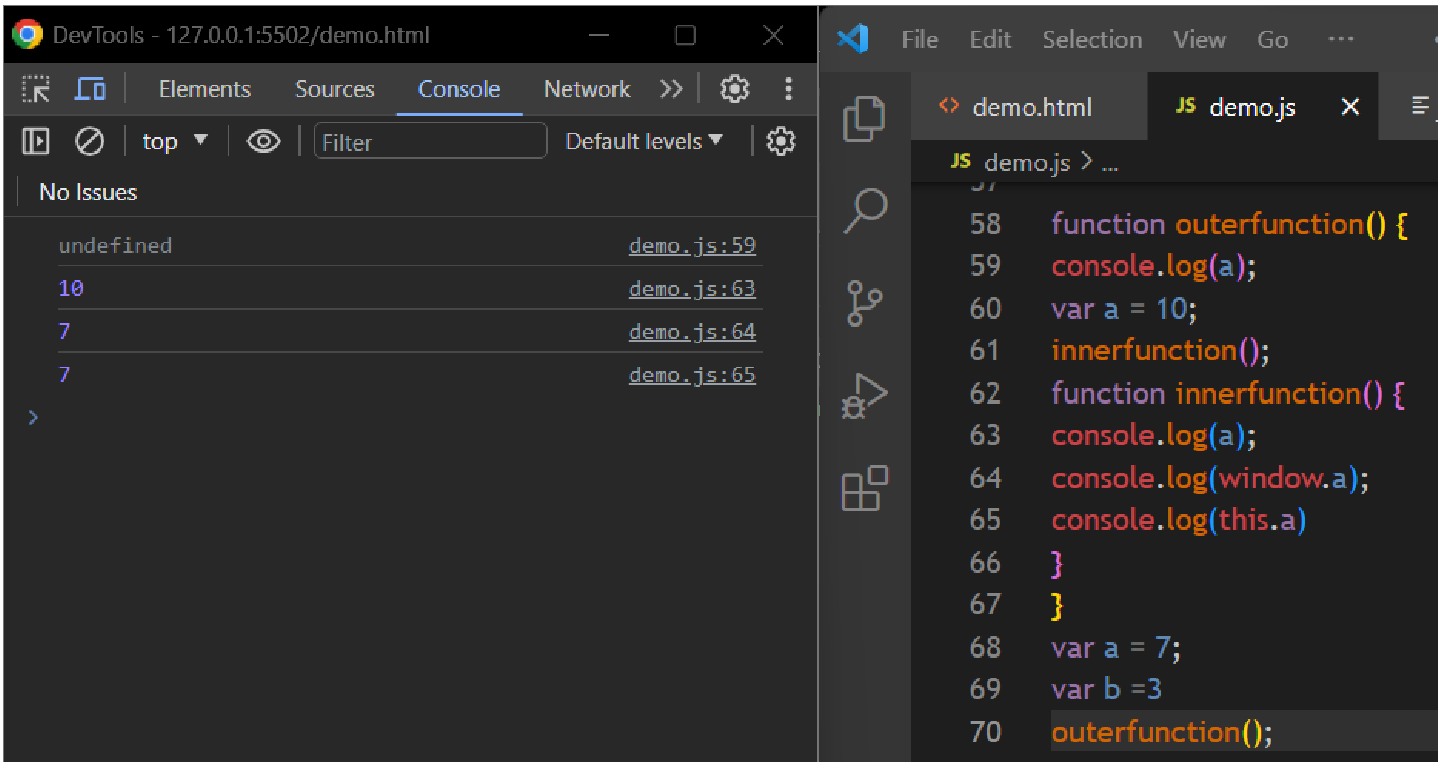
```
undefined                                    demo.js:59
>
```

```
54  put = update
55  delete = delete
56  */
57
58  function abc() {
59      console.log(a);
60  }
61  abc();
62  var a = 7;
```



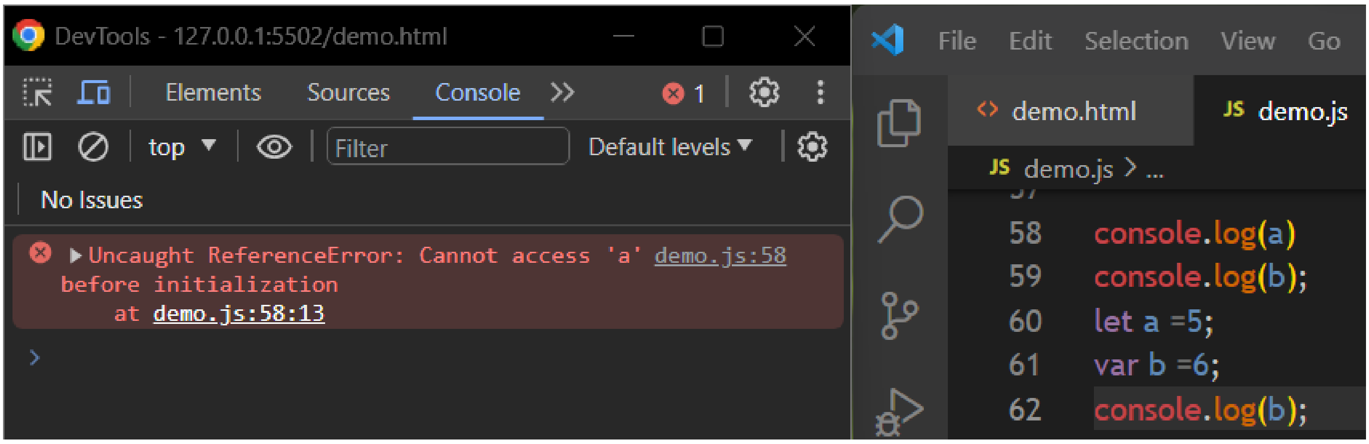
```
7                                          demo.js:59
3                                          demo.js:63
10                                         demo.js:64
```

```
function outerfunction() {
console.log(a);
var c = 10;
innerfunction();
function innerfunction() {
    console.log(b);
    console.log(c);
    }
}
var a = 7;
var b =3
outerfunction();
```



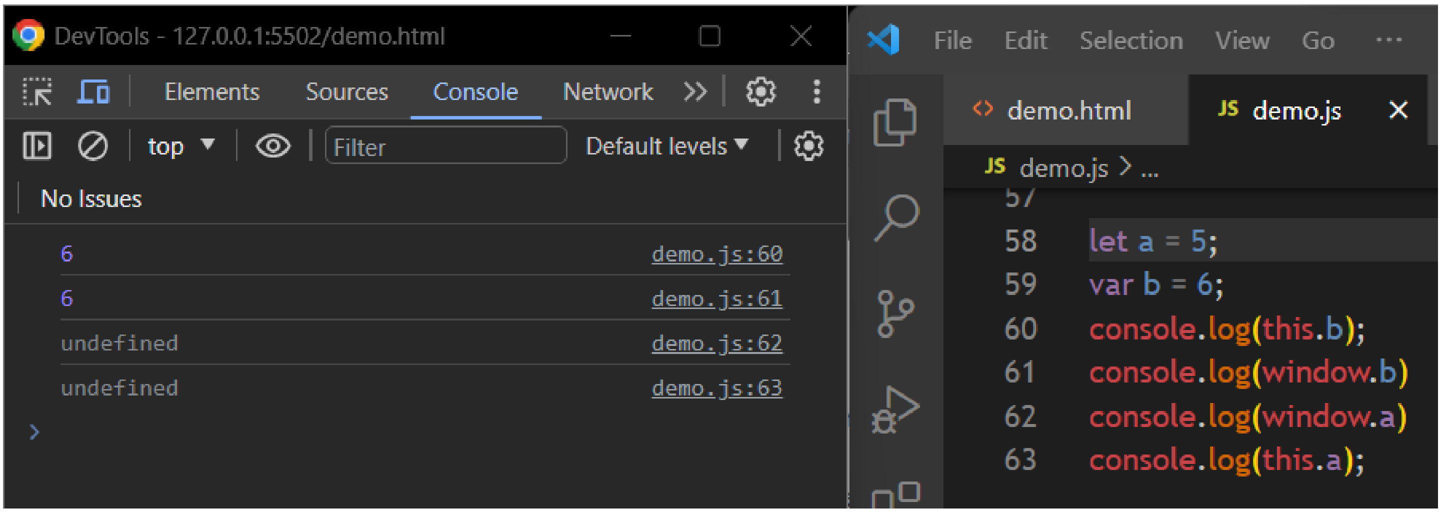
```
undefined                                  demo.js:59
10                                         demo.js:63
7                                          demo.js:64
7                                          demo.js:65
```

```
function outerfunction() {
console.log(a);
var a = 10;
innerfunction();
function innerfunction() {
console.log(a);
console.log(window.a);
console.log(this.a)
}
}
var a = 7;
var b =3
outerfunction();
```



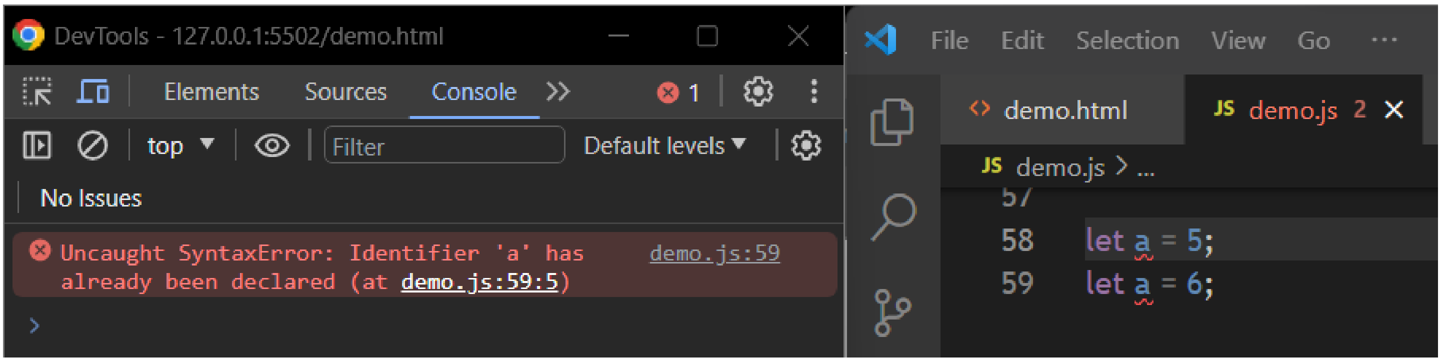
```
▶Uncaught ReferenceError: Cannot access 'a' demo.js:58
before initialization
    at demo.js:58:13
```

```
console.log(a)
console.log(b);
let a =5;
var b =6;
console.log(b);
```



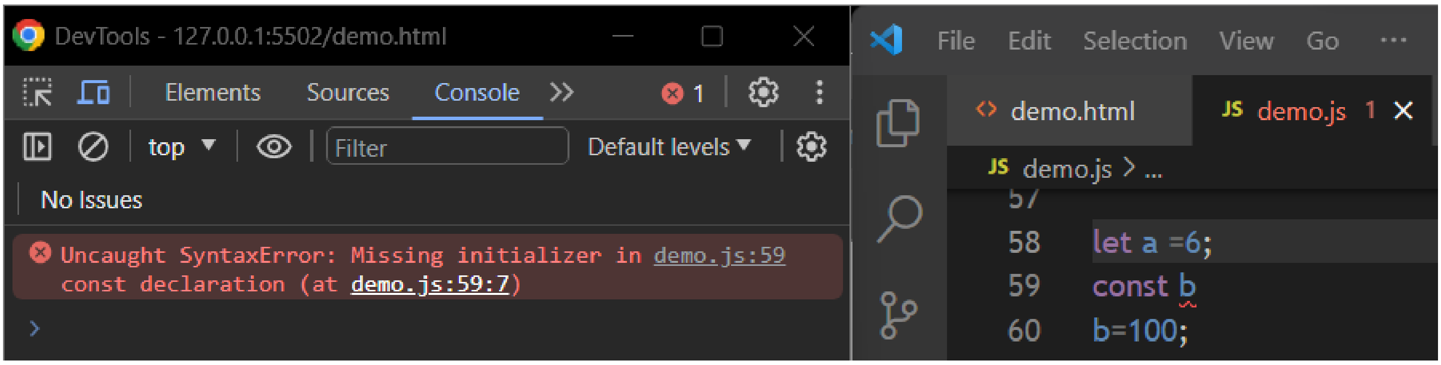
```
6                                              demo.js:60
6                                              demo.js:61
undefined                                      demo.js:62
undefined                                      demo.js:63
```

```
let a = 5;
var b = 6;
console.log(this.b);
console.log(window.b)
console.log(window.a)
console.log(this.a);
```



```
Uncaught SyntaxError: Identifier 'a' has      demo.js:59
already been declared (at demo.js:59:5)
```

```
let a = 5;
let a = 6;
```

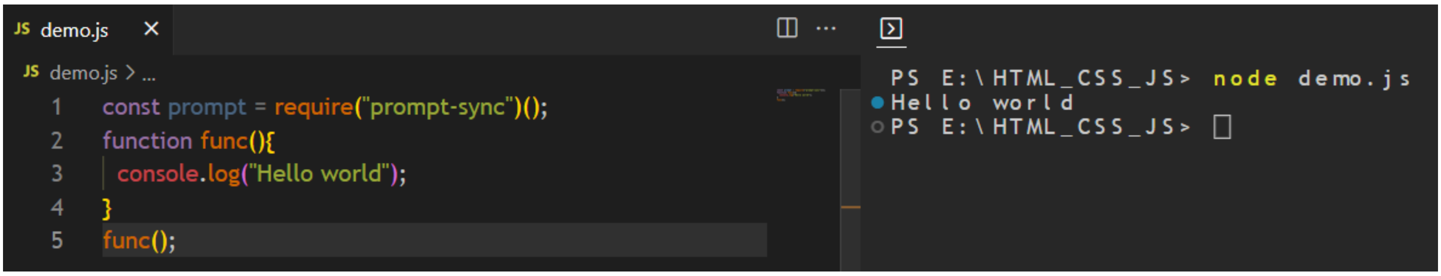


```
Uncaught SyntaxError: Missing initializer in demo.js:59
const declaration (at demo.js:59:7)
```

```
let a =6;
const b
b=100;
```

Types Of Traditional Function Variants:

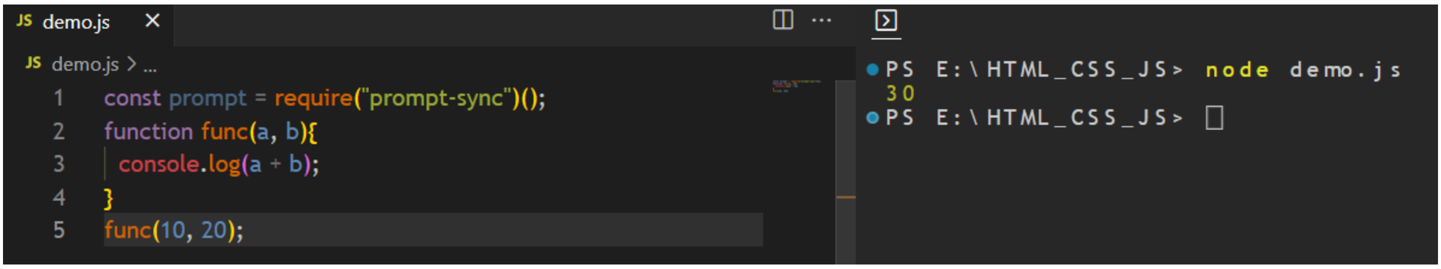
Take Nothing Return Nothing:



```
const prompt = require("prompt-sync")();
function func(){
    console.log("Hello world");
}
func();
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello world
PS E:\HTML_CSS_JS>
```

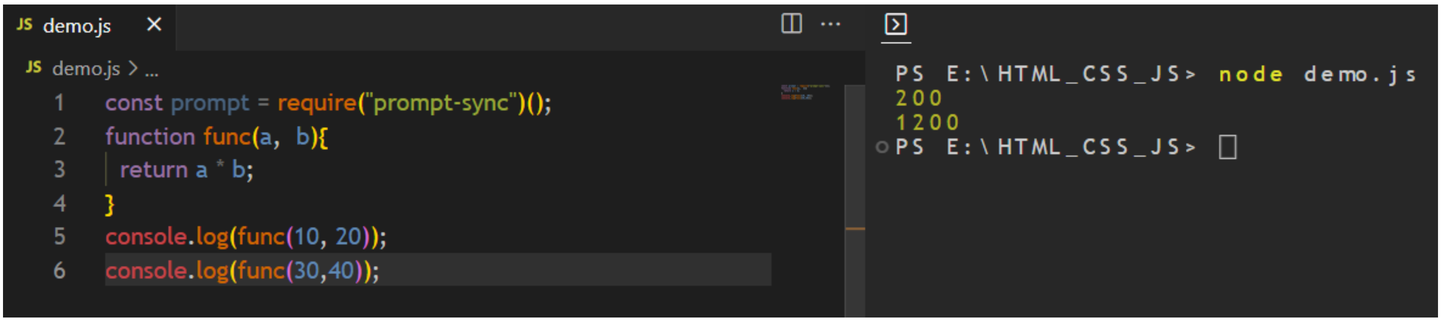
Take Something Return Nothing:



```
const prompt = require("prompt-sync")();
function func(a, b){
    console.log(a + b);
}
func(10, 20);
```

```
PS E:\HTML_CSS_JS> node demo.js
30
PS E:\HTML_CSS_JS>
```

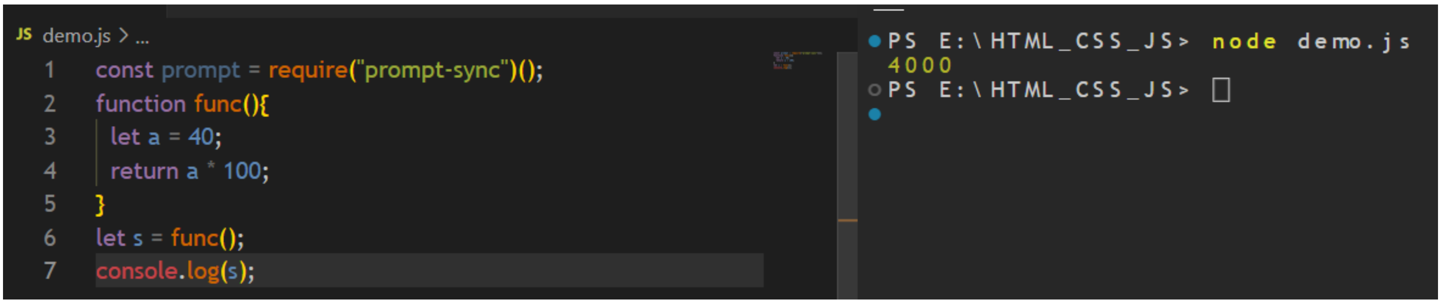
Take Something Return Something:



```
const prompt = require("prompt-sync")();
function func(a,  b){
  return a * b;
}
console.log(func(10, 20));
console.log(func(30,40));
```

```
PS E:\HTML_CSS_JS> node demo.js
200
1200
PS E:\HTML_CSS_JS>
```

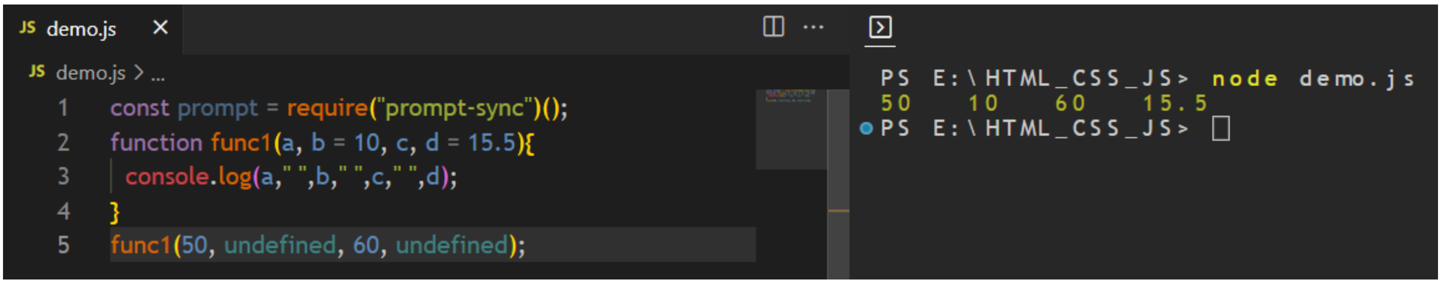
Take Nothing Return Something:



```
const prompt = require("prompt-sync")();
function func(){
  let a = 40;
  return a * 100;
}
let s = func();
console.log(s);
```

```
PS E:\HTML_CSS_JS> node demo.js
4000
PS E:\HTML_CSS_JS>
```

Sample Experiment On Function:



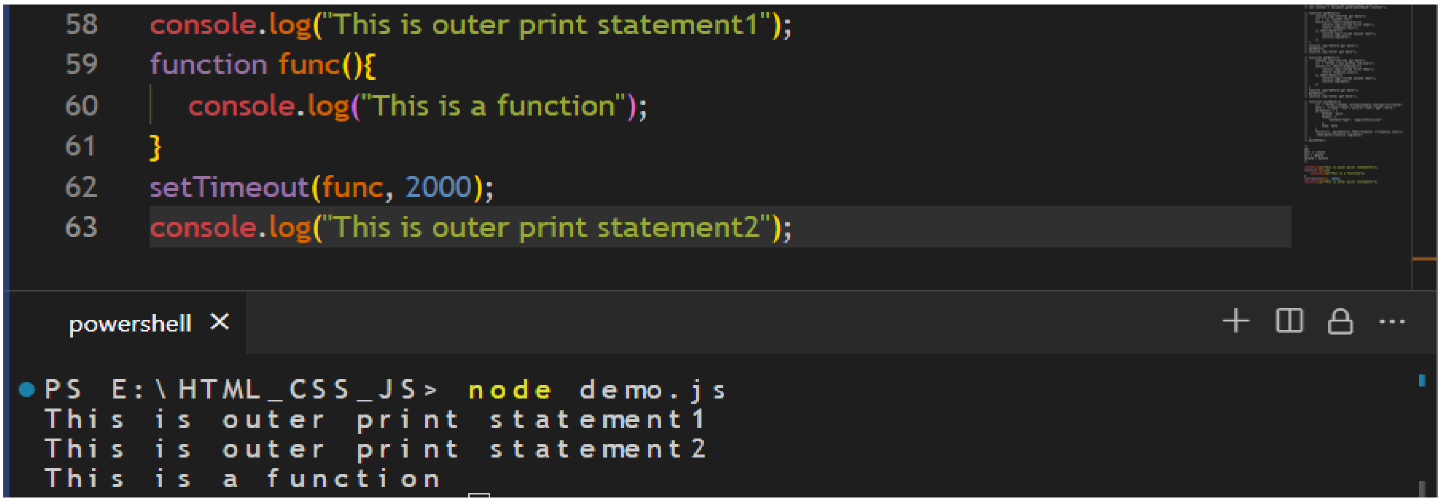
```
const prompt = require("prompt-sync")();
function func1(a, b = 10, c, d = 15.5){
  console.log(a," ",b," ",c," ",d);
}
func1(50, undefined, 60, undefined);
```

```
PS E:\HTML_CSS_JS> node demo.js
50    10    60    15.5
PS E:\HTML_CSS_JS>
```

SetTimeout Function:

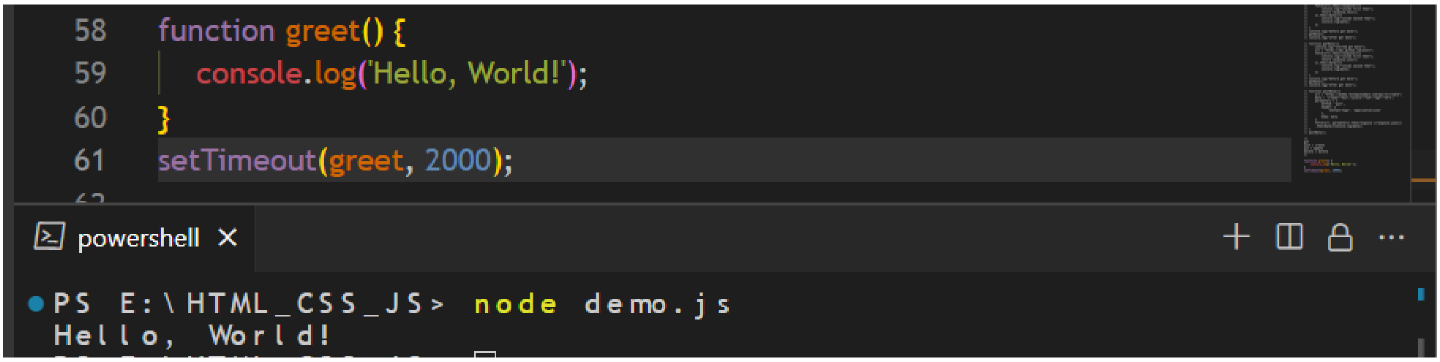
setTimeout is a JavaScript function used for scheduling the execution of a function (or the evaluation of an expression) after a specified delay in milliseconds. It allows you to create delays in your code, which can be useful for various purposes like animation, event handling, or implementing timeouts. Syntax: **setTimeout(function, delay);**

function: The function or code you want to execute after the specified delay. delay: The time, in milliseconds, that you want to wait before executing the function.



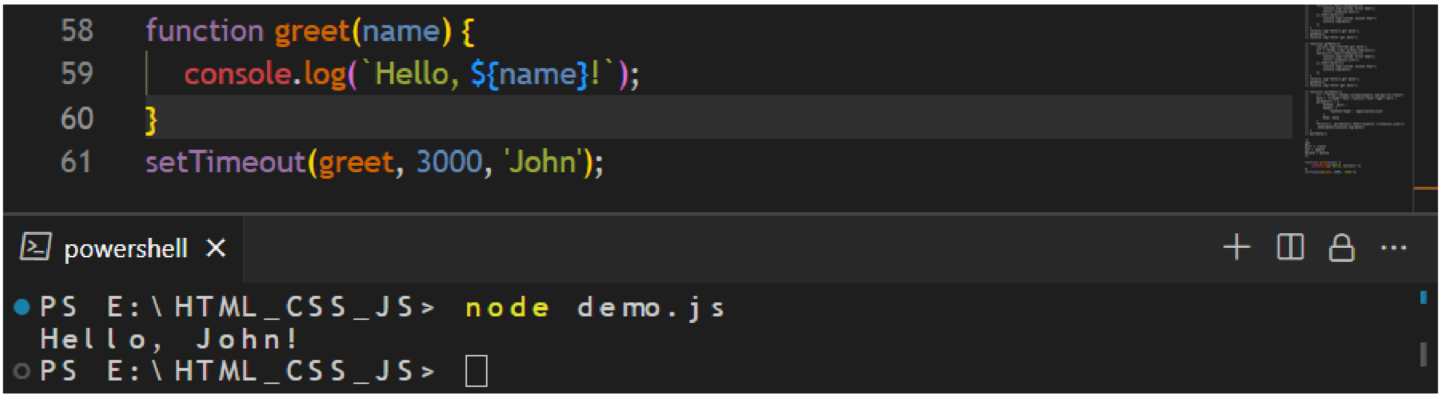
```
console.log("This is outer print statement1");
function func(){
    console.log("This is a function");
}
setTimeout(func, 2000);
console.log("This is outer print statement2");
```

```
PS E:\HTML_CSS_JS> node demo.js
This is outer print statement1
This is outer print statement2
This is a function
```



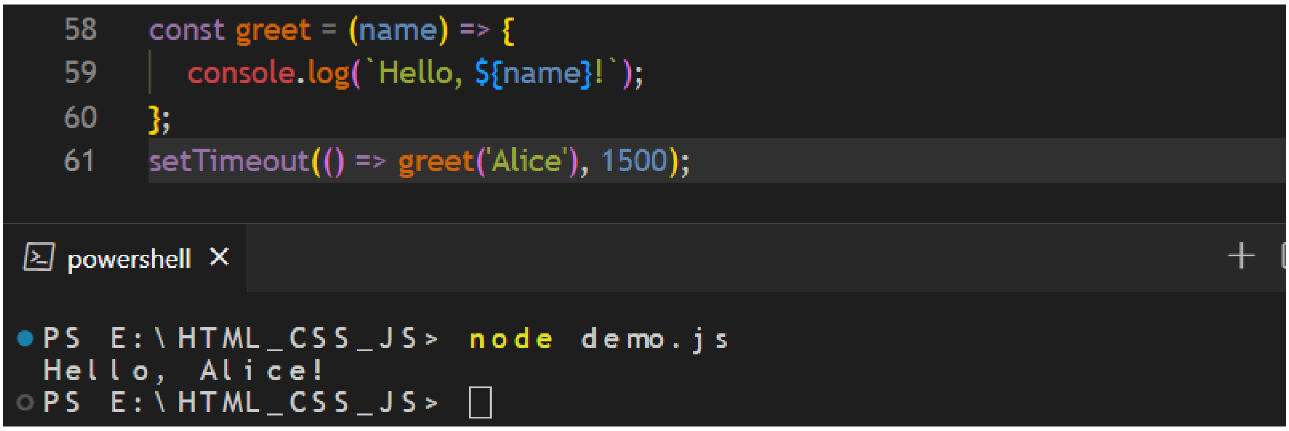
```javascript
function greet() {
  console.log('Hello, World!');
}
setTimeout(greet, 2000);
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello, World!
```



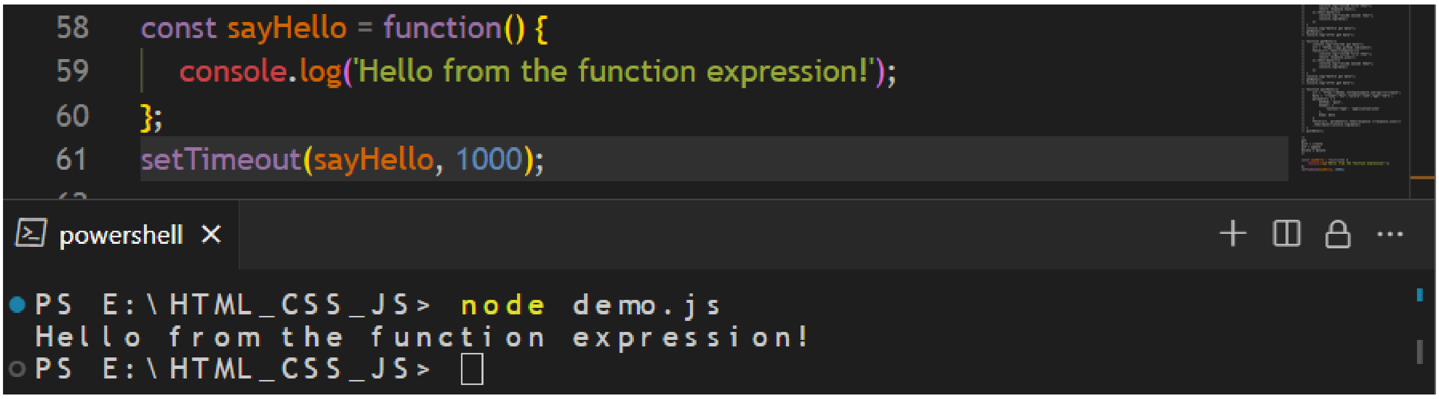
```javascript
function greet(name) {
  console.log(`Hello, ${name}!`);
}
setTimeout(greet, 3000, 'John');
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello, John!
PS E:\HTML_CSS_JS>
```



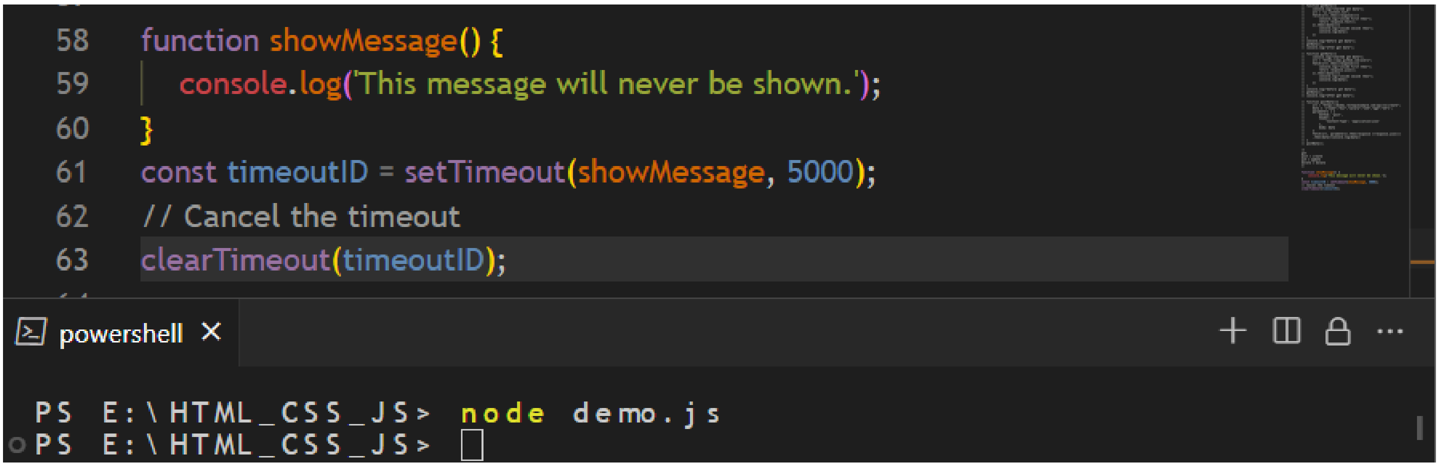
```javascript
const greet = (name) => {
  console.log(`Hello, ${name}!`);
};
setTimeout(() => greet('Alice'), 1500);
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello, Alice!
PS E:\HTML_CSS_JS>
```



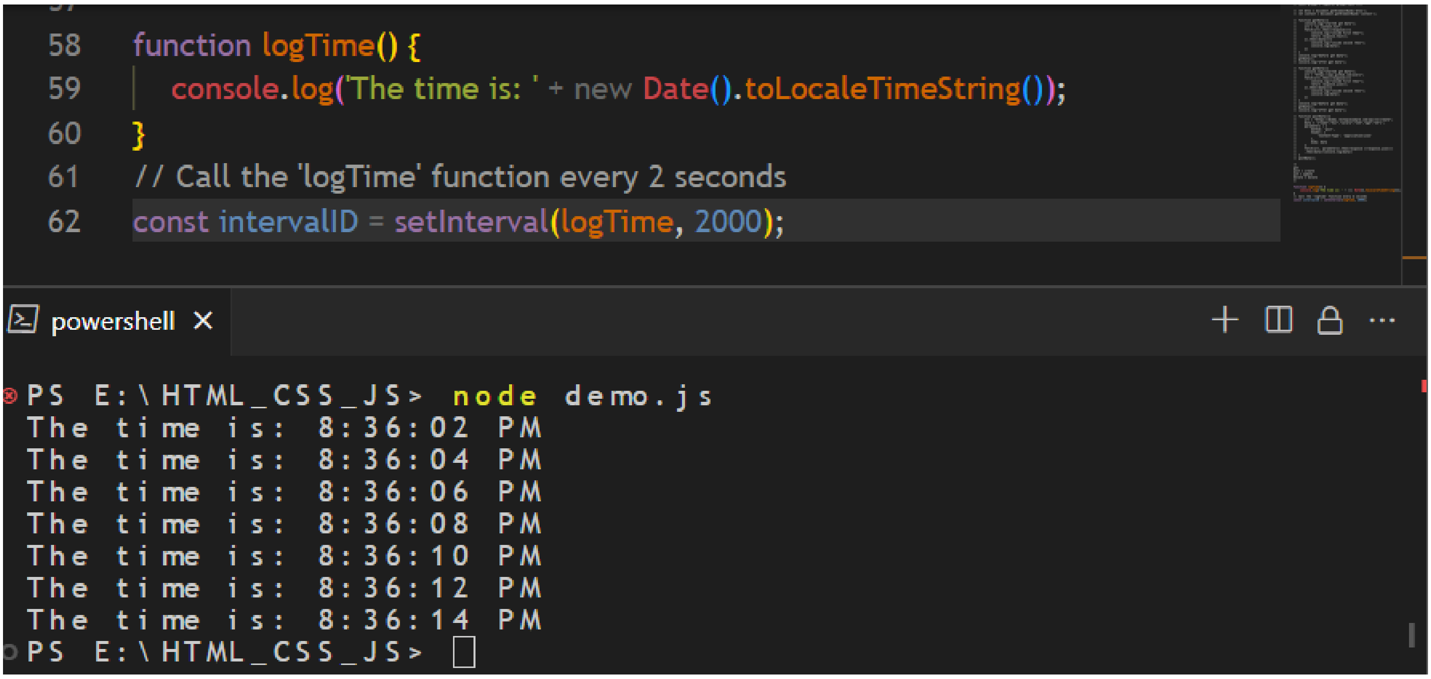
```javascript
const sayHello = function() {
  console.log('Hello from the function expression!');
};
setTimeout(sayHello, 1000);
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello from the function expression!
PS E:\HTML_CSS_JS>
```



```javascript
function showMessage() {
    console.log('This message will never be shown.');
}
const timeoutID = setTimeout(showMessage, 5000);
// Cancel the timeout
clearTimeout(timeoutID);
```

```
PS E:\HTML_CSS_JS> node demo.js
PS E:\HTML_CSS_JS> 
```

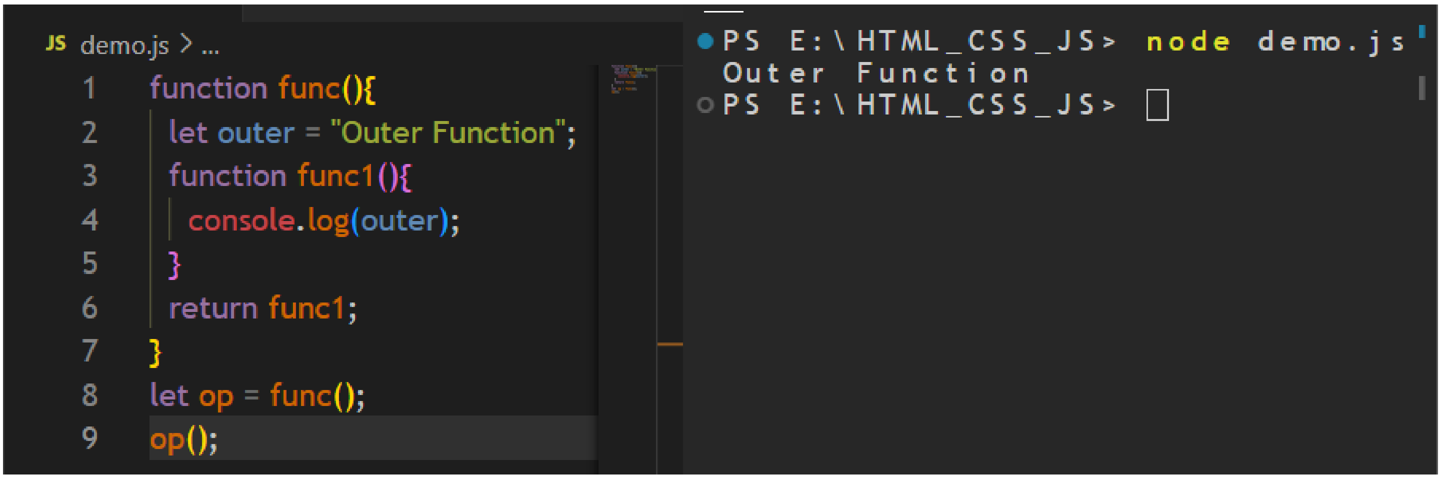


```javascript
function logTime() {
    console.log('The time is: ' + new Date().toLocaleTimeString());
}
// Call the 'logTime' function every 2 seconds
const intervalID = setInterval(logTime, 2000);
```

```
PS E:\HTML_CSS_JS> node demo.js
The time is: 8:36:02 PM
The time is: 8:36:04 PM
The time is: 8:36:06 PM
The time is: 8:36:08 PM
The time is: 8:36:10 PM
The time is: 8:36:12 PM
The time is: 8:36:14 PM
PS E:\HTML_CSS_JS> 
```

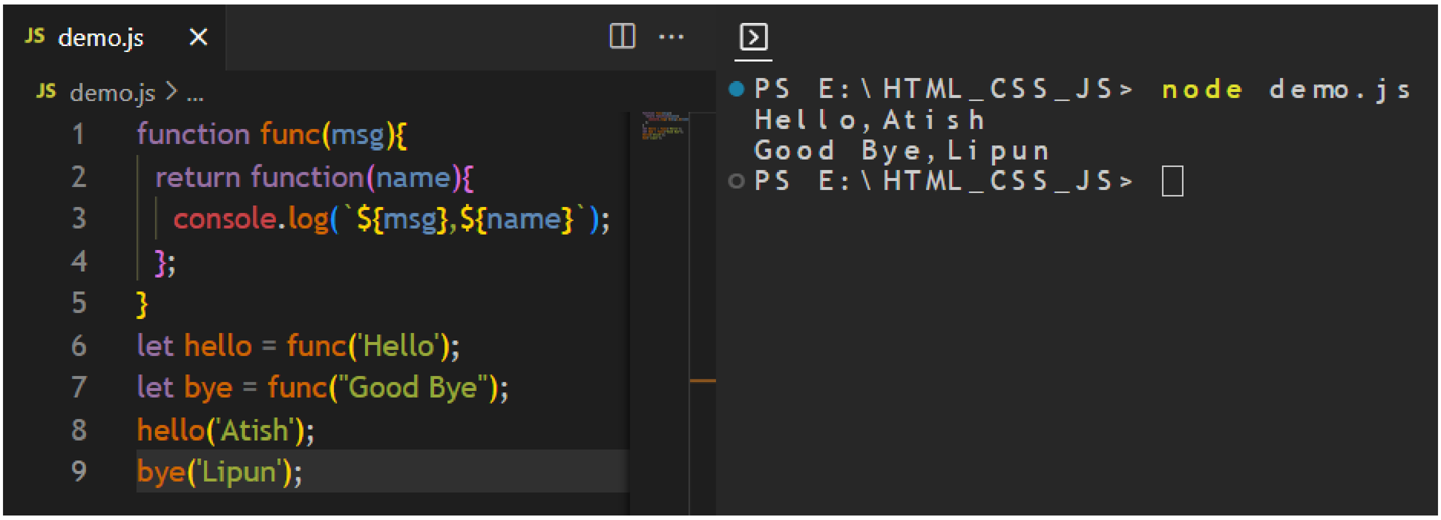
Closure In JavaScript:

In JavaScript, a closure is a fundamental and powerful concept. It occurs when a function "closes over" its surrounding lexical scope, retaining access to the variables, parameters, and functions within that scope even after the outer function has finished executing. Closures are essential for maintaining data encapsulation, creating private variables, and enabling more flexible and modular code.



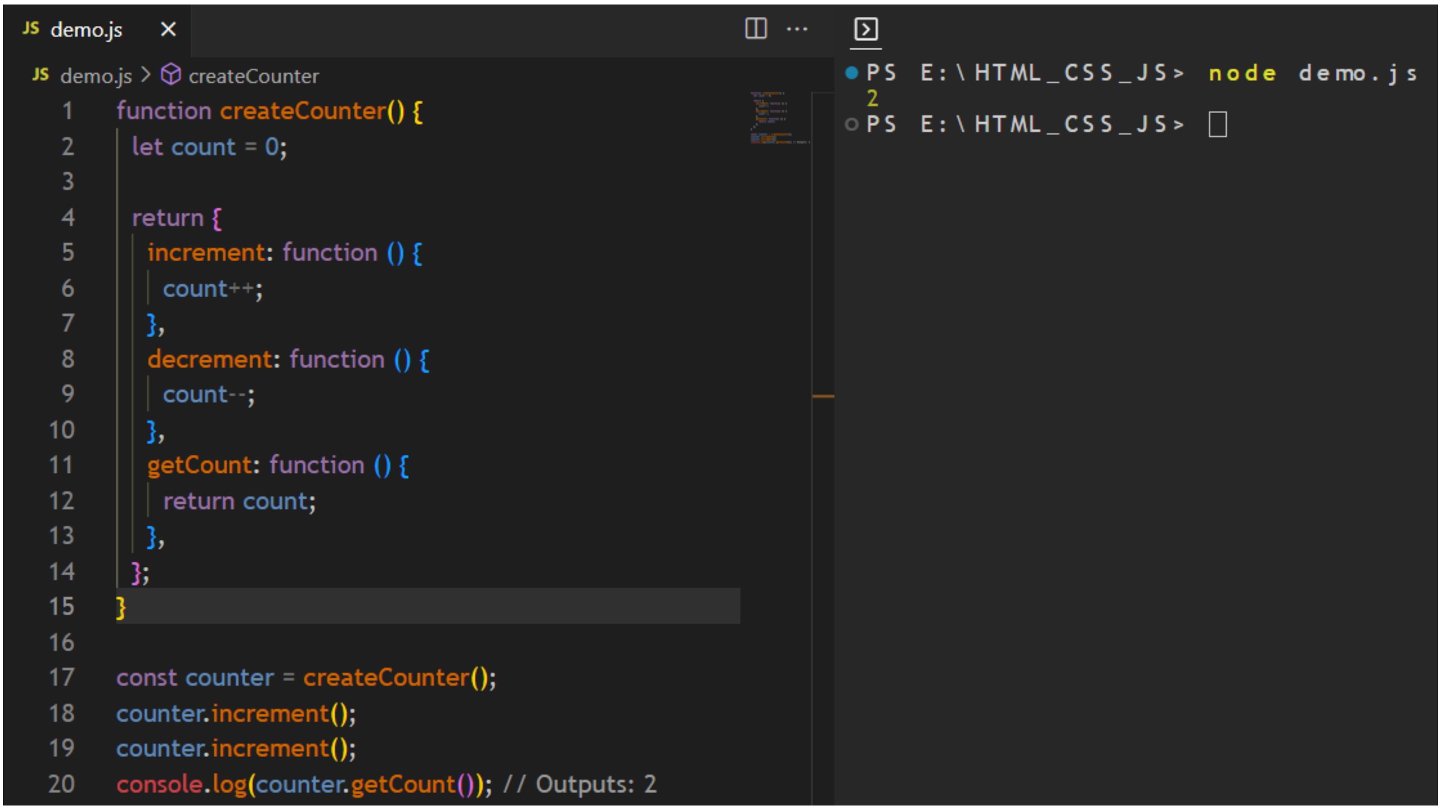
```javascript
function func(){
    let outer = "Outer Function";
    function func1(){
        console.log(outer);
    }
    return func1;
}
let op = func();
op();
```

```
PS E:\HTML_CSS_JS> node demo.js
Outer Function
PS E:\HTML_CSS_JS> 
```



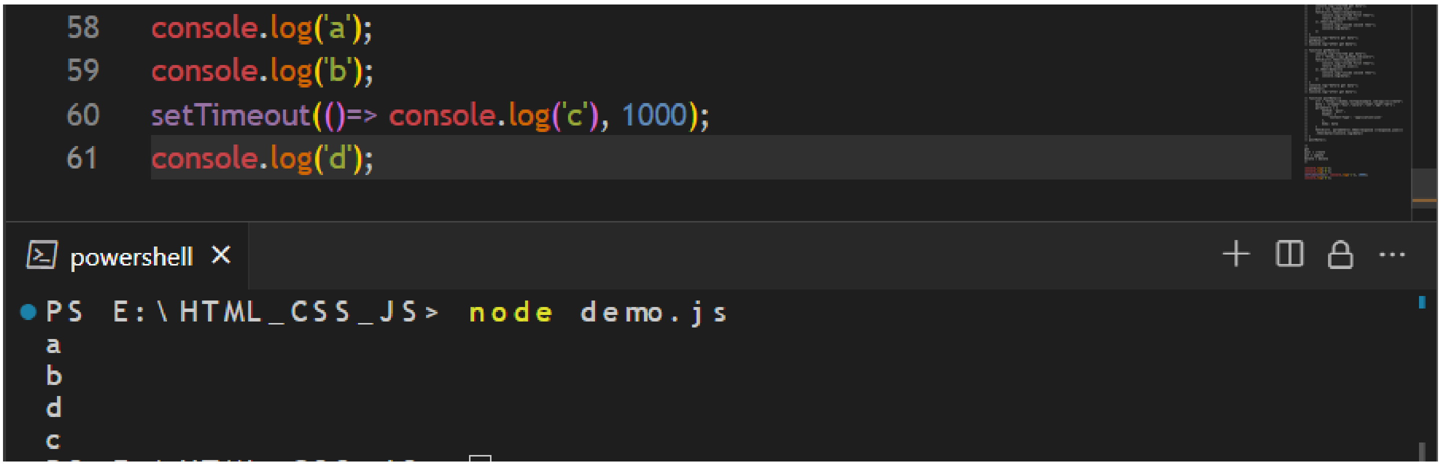
```javascript
function func(msg){
  return function(name){
    console.log(`${msg},${name}`);
  };
}
let hello = func('Hello');
let bye = func("Good Bye");
hello('Atish');
bye('Lipun');
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello,Atish
Good Bye,Lipun
PS E:\HTML_CSS_JS>
```



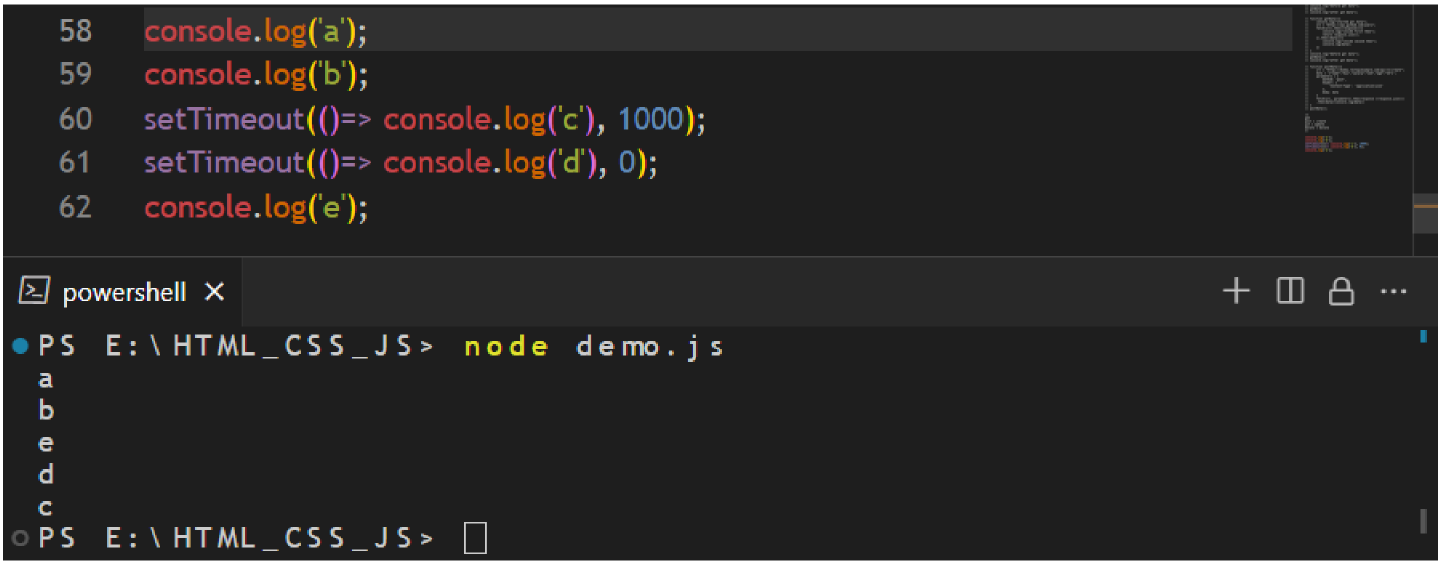
```javascript
function createCounter() {
  let count = 0;

  return {
    increment: function () {
      count++;
    },
    decrement: function () {
      count--;
    },
    getCount: function () {
      return count;
    },
  };
}

const counter = createCounter();
counter.increment();
counter.increment();
console.log(counter.getCount()); // Outputs: 2
```

```
PS E:\HTML_CSS_JS> node demo.js
2
PS E:\HTML_CSS_JS>
```



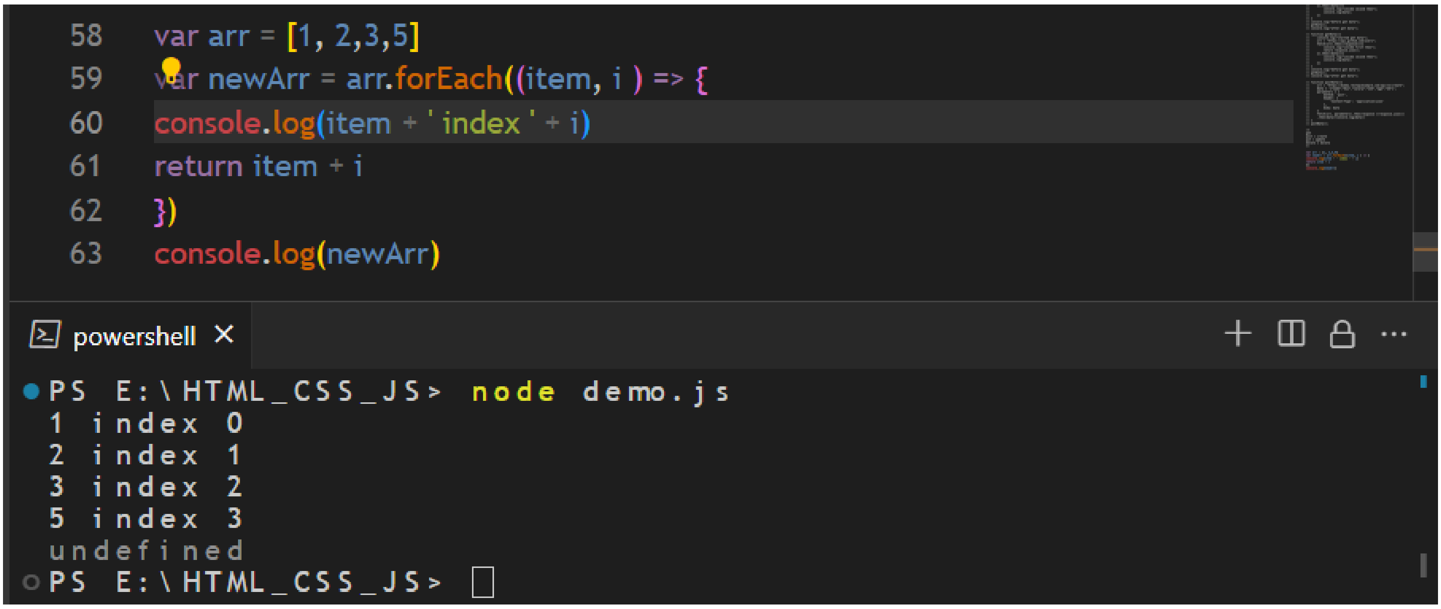
```javascript
console.log('a');
console.log('b');
setTimeout(()=> console.log('c'), 1000);
console.log('d');
```

```
PS E:\HTML_CSS_JS> node demo.js
a
b
d
c
```



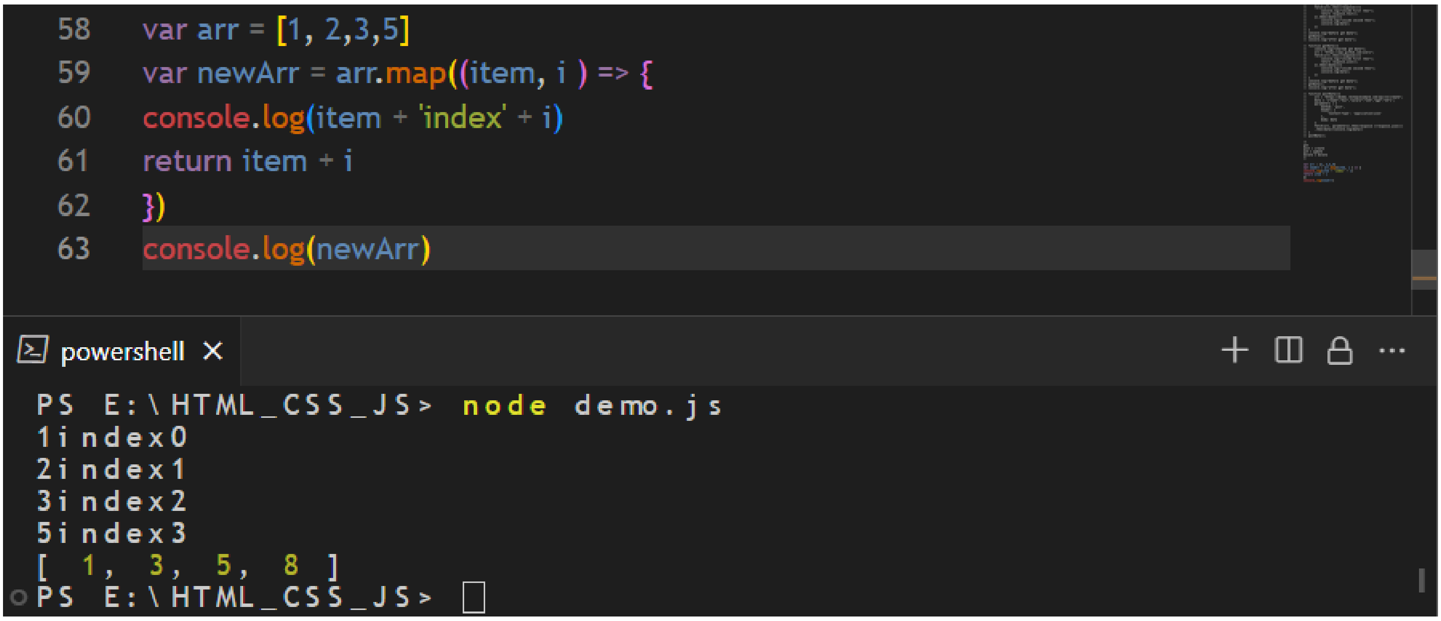
```javascript
console.log('a');
console.log('b');
setTimeout(()=> console.log('c'), 1000);
setTimeout(()=> console.log('d'), 0);
console.log('e');
```

```
PS E:\HTML_CSS_JS> node demo.js
a
b
e
d
c
PS E:\HTML_CSS_JS>
```



```javascript
var arr = [1, 2,3,5]
var newArr = arr.forEach((item, i ) => {
console.log(item + ' index ' + i)
return item + i
})
console.log(newArr)
```

```
PS E:\HTML_CSS_JS> node demo.js
1 index 0
2 index 1
3 index 2
5 index 3
undefined
PS E:\HTML_CSS_JS>
```

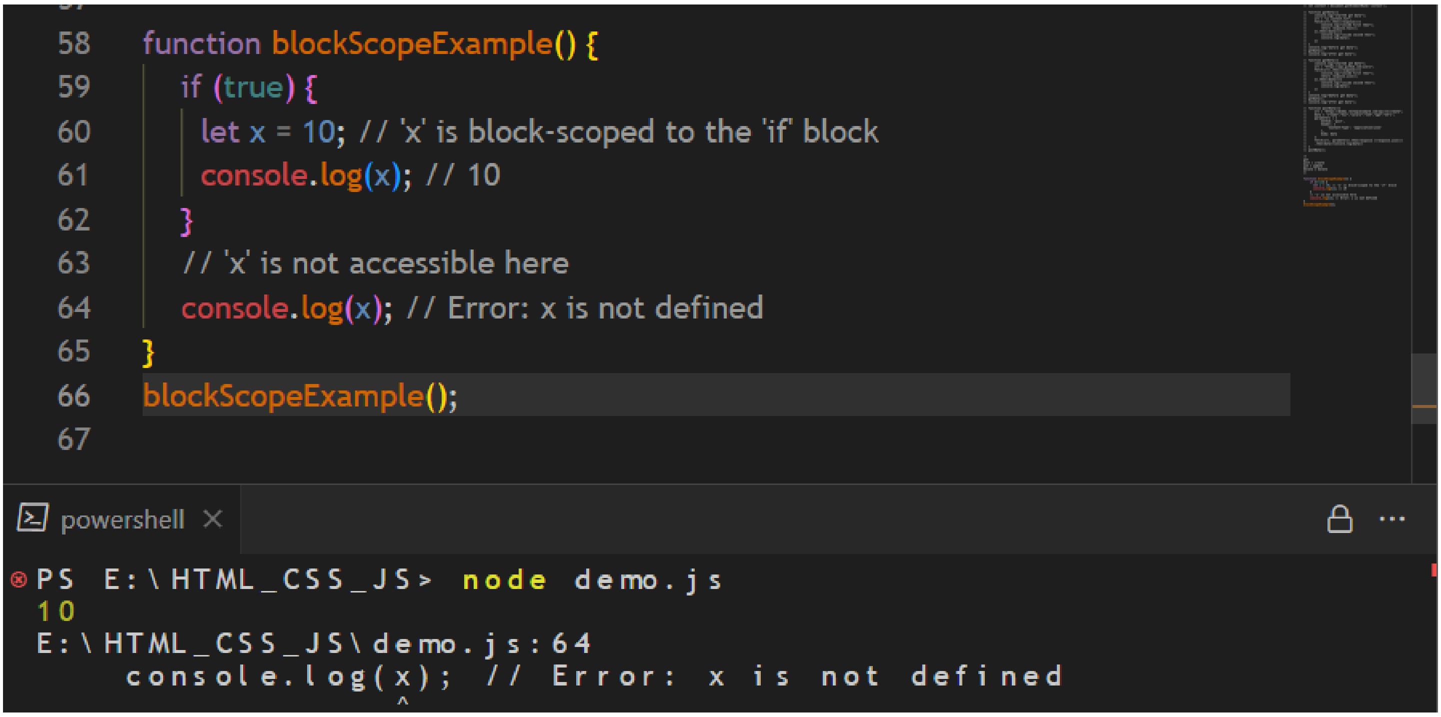


```javascript
var arr = [1, 2,3,5]
var newArr = arr.map((item, i ) => {
console.log(item + 'index' + i)
return item + i
})
console.log(newArr)
```

```
PS E:\HTML_CSS_JS> node demo.js
1index0
2index1
3index2
5index3
[ 1, 3, 5, 8 ]
PS E:\HTML_CSS_JS>
```

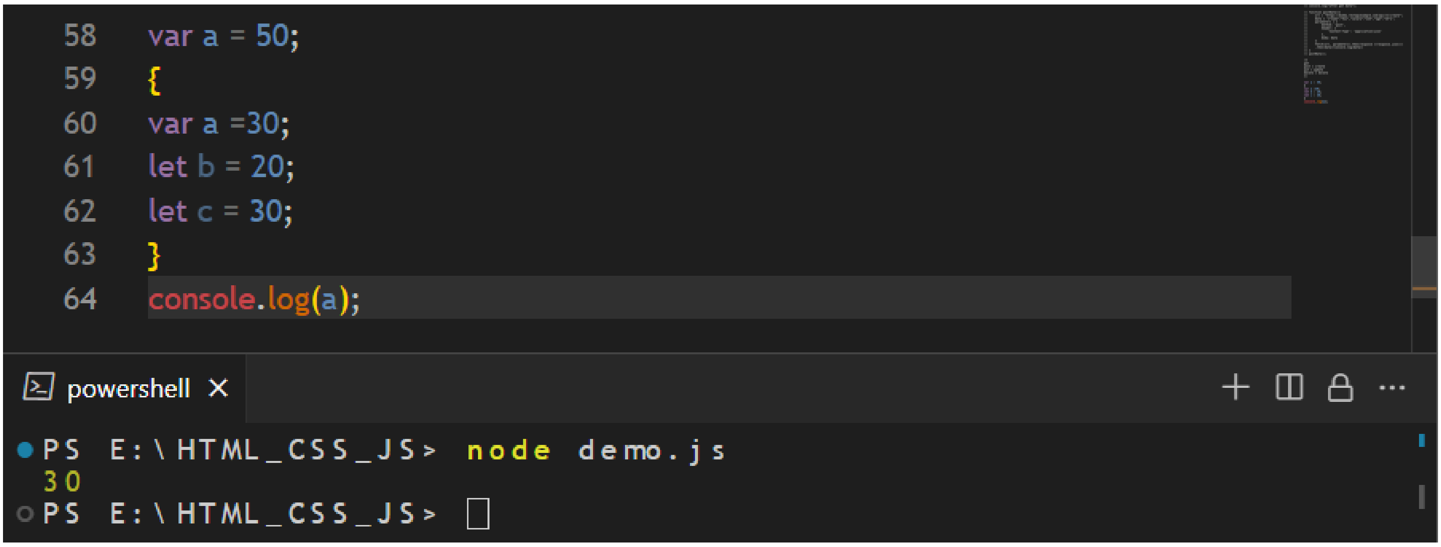
Block Scope:

Block scope refers to the scope of a variable or function that is confined within a specific block of code. A block in JavaScript is defined by a set of curly braces {} and can be found within functions, loops, conditional statements, or anywhere else that creates a new lexical scope. Understanding block scope is essential for managing variable lifetimes and avoiding unintended variable collisions. Here are some examples and explanations of block scope in JavaScript



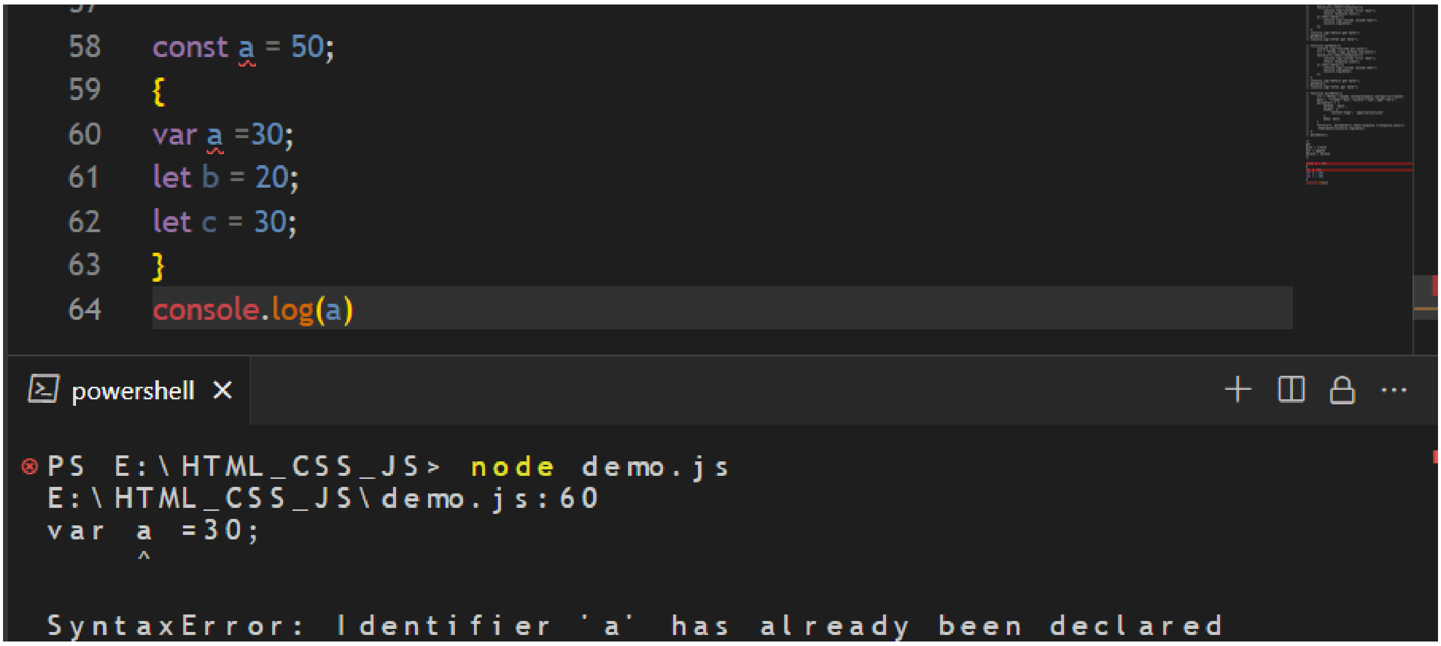
```javascript
function blockScopeExample() {
  if (true) {
    let x = 10; // 'x' is block-scoped to the 'if' block
    console.log(x); // 10
  }
  // 'x' is not accessible here
  console.log(x); // Error: x is not defined
}
blockScopeExample();
```

```
PS E:\HTML_CSS_JS> node demo.js
10
E:\HTML_CSS_JS\demo.js:64
    console.log(x); // Error: x is not defined
                ^
```



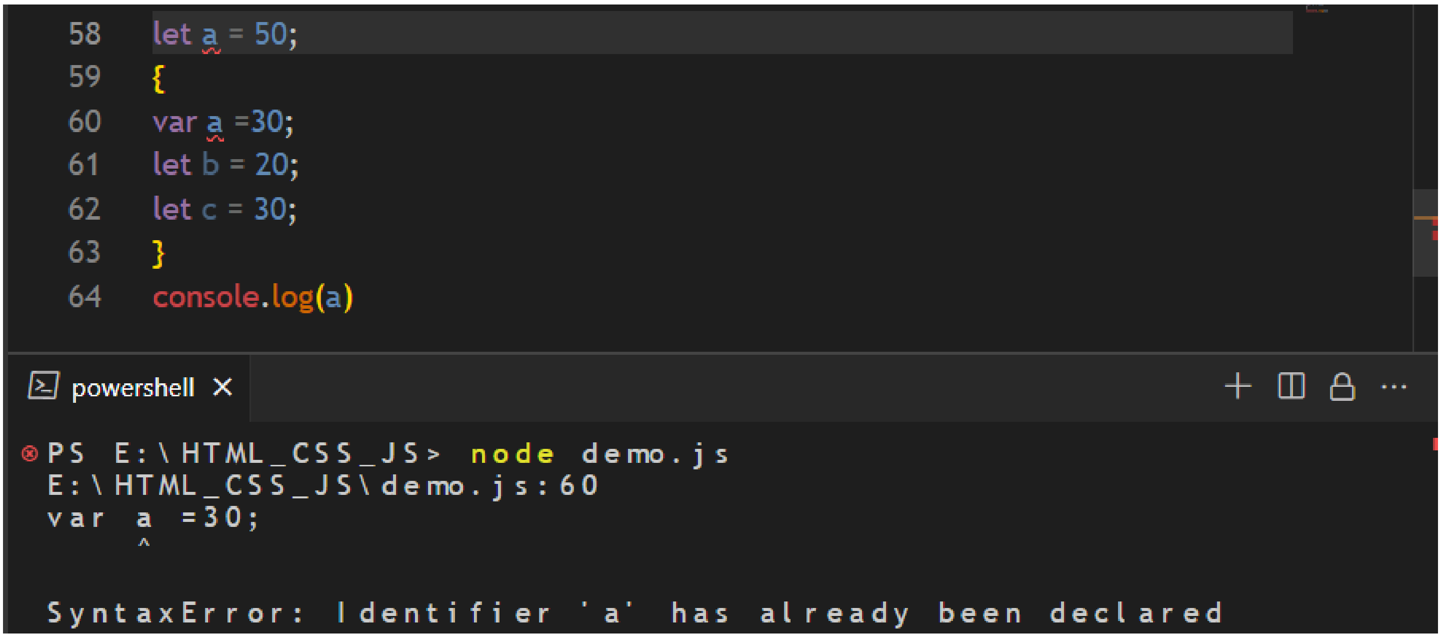
```javascript
var a = 50;
{
var a =30;
let b = 20;
let c = 30;
}
console.log(a);
```

```
PS E:\HTML_CSS_JS> node demo.js
30
PS E:\HTML_CSS_JS>
```



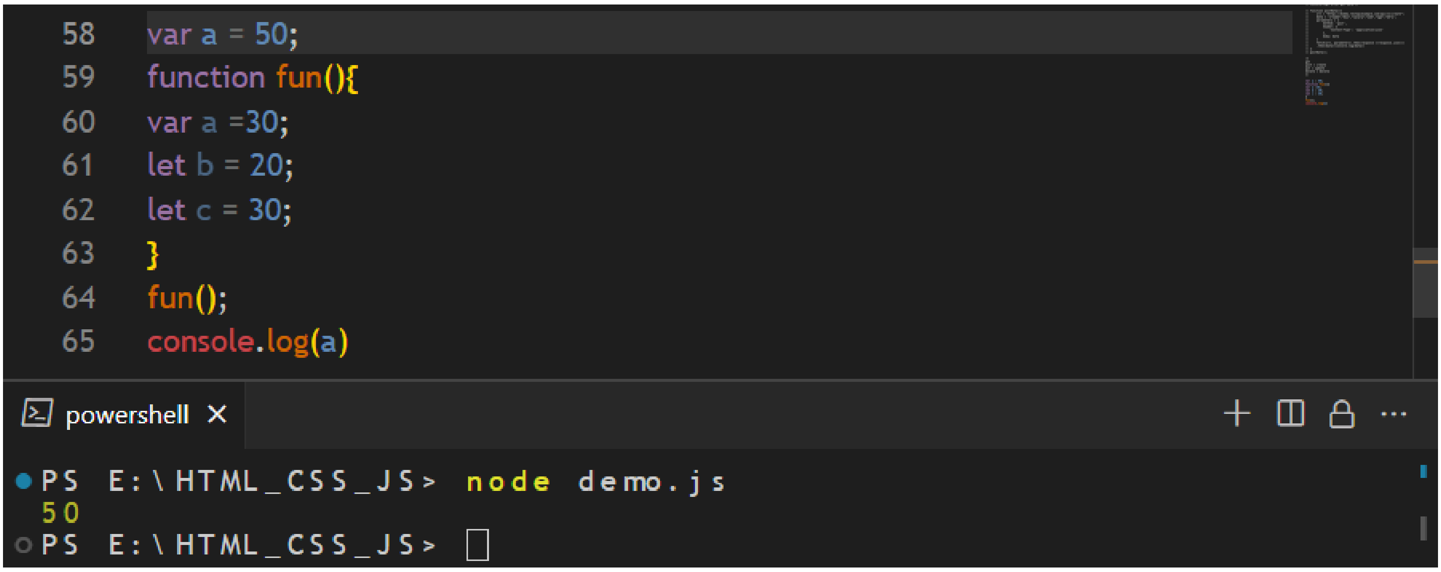
```
const a = 50;
{
var a =30;
let b = 20;
let c = 30;
}
console.log(a)
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:60
var a =30;
    ^

SyntaxError: Identifier 'a' has already been declared
```



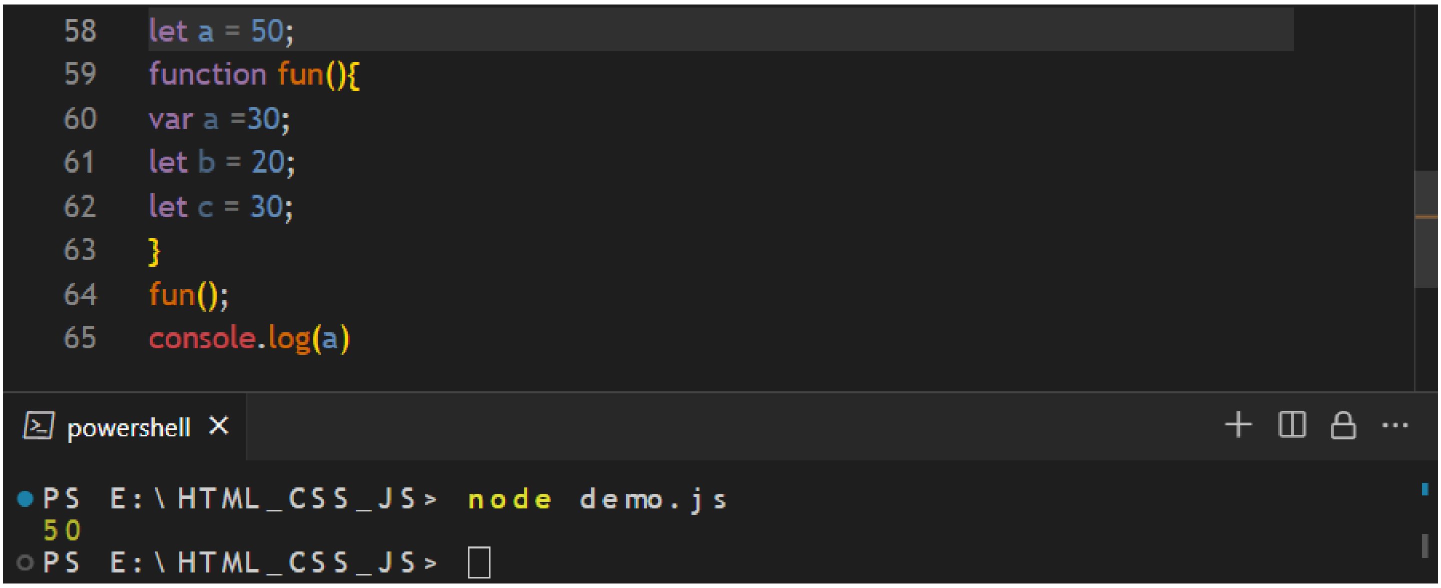
```
let a = 50;
{
var a =30;
let b = 20;
let c = 30;
}
console.log(a)
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:60
var a =30;
    ^

SyntaxError: Identifier 'a' has already been declared
```



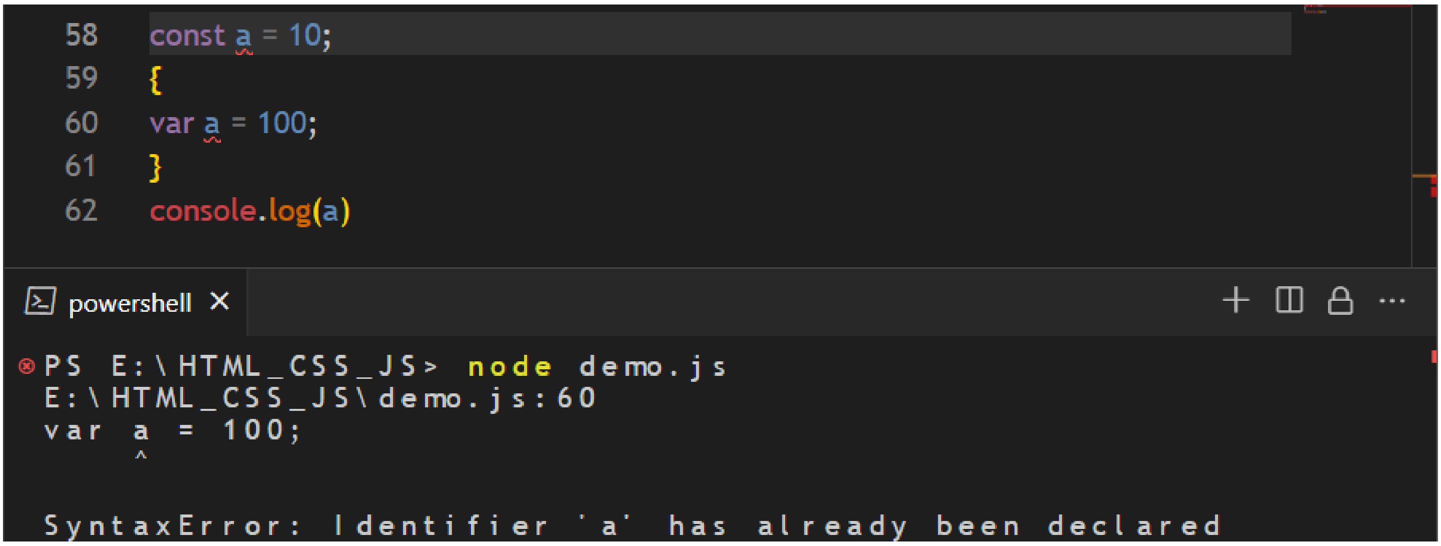
```
var a = 50;
function fun(){
var a =30;
let b = 20;
let c = 30;
}
fun();
console.log(a)
```

```
PS E:\HTML_CSS_JS> node demo.js
50
PS E:\HTML_CSS_JS> 
```



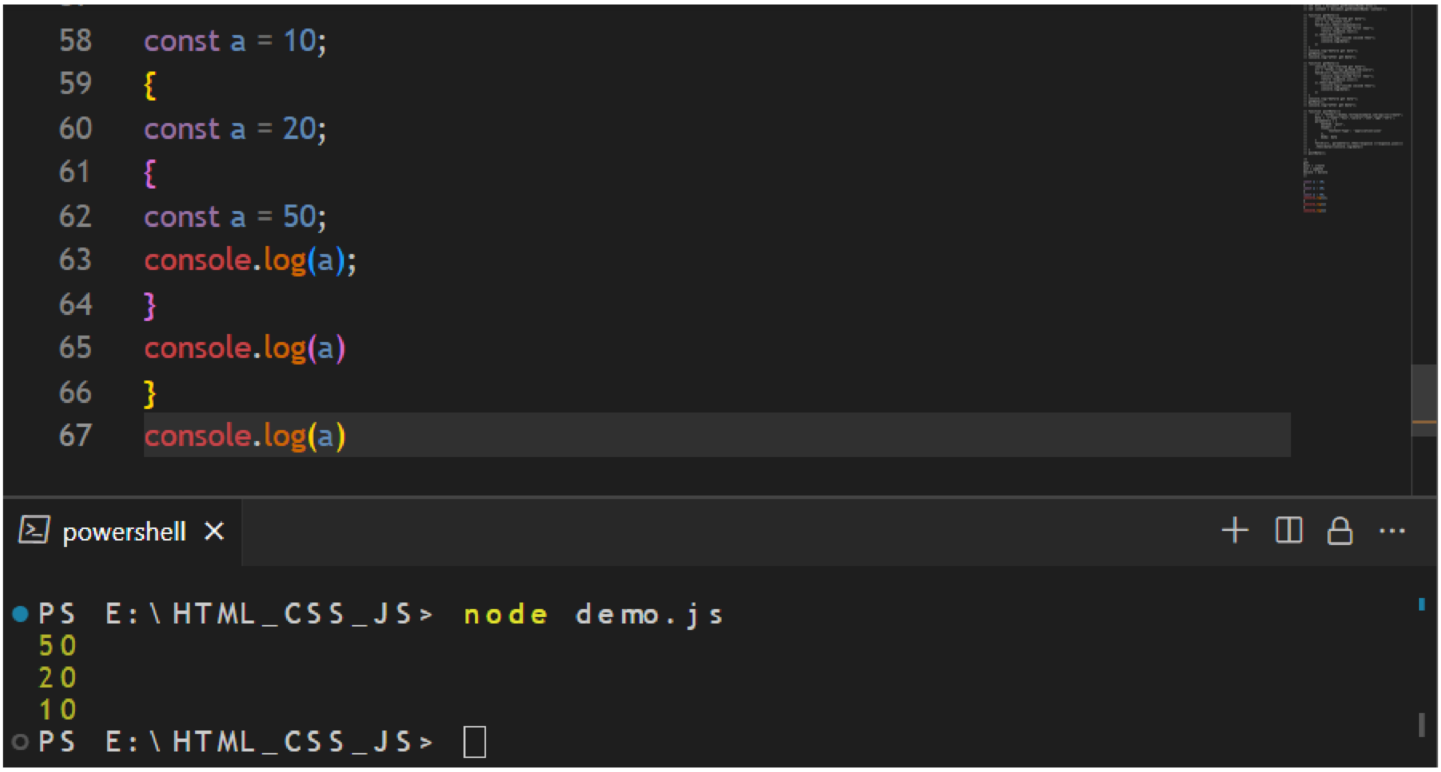
```javascript
let a = 50;
function fun(){
var a =30;
let b = 20;
let c = 30;
}
fun();
console.log(a)
```

```
PS E:\HTML_CSS_JS> node demo.js
50
PS E:\HTML_CSS_JS> 
```



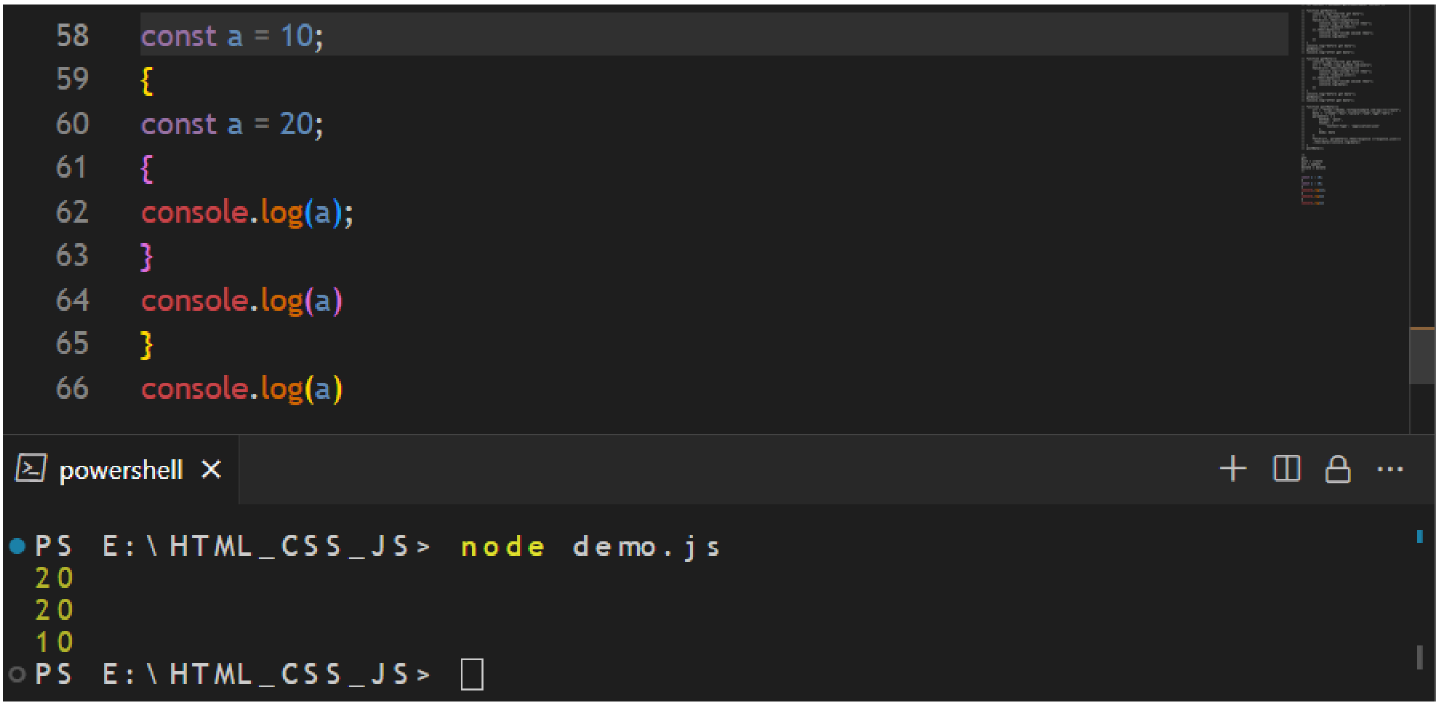
```javascript
const a = 10;
{
var a = 100;
}
console.log(a)
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:60
var a = 100;
    ^

SyntaxError: Identifier 'a' has already been declared
```



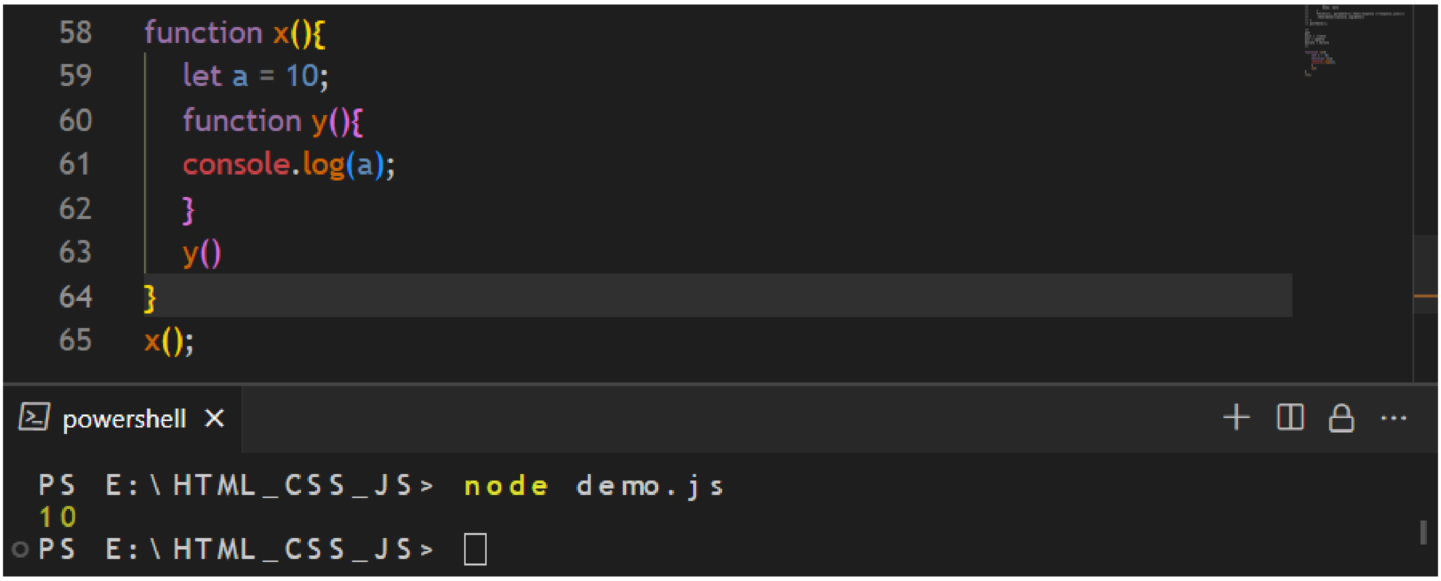
```javascript
const a = 10;
{
const a = 20;
{
const a = 50;
console.log(a);
}
console.log(a)
}
console.log(a)
```

```
PS E:\HTML_CSS_JS> node demo.js
50
20
10
PS E:\HTML_CSS_JS> 
```



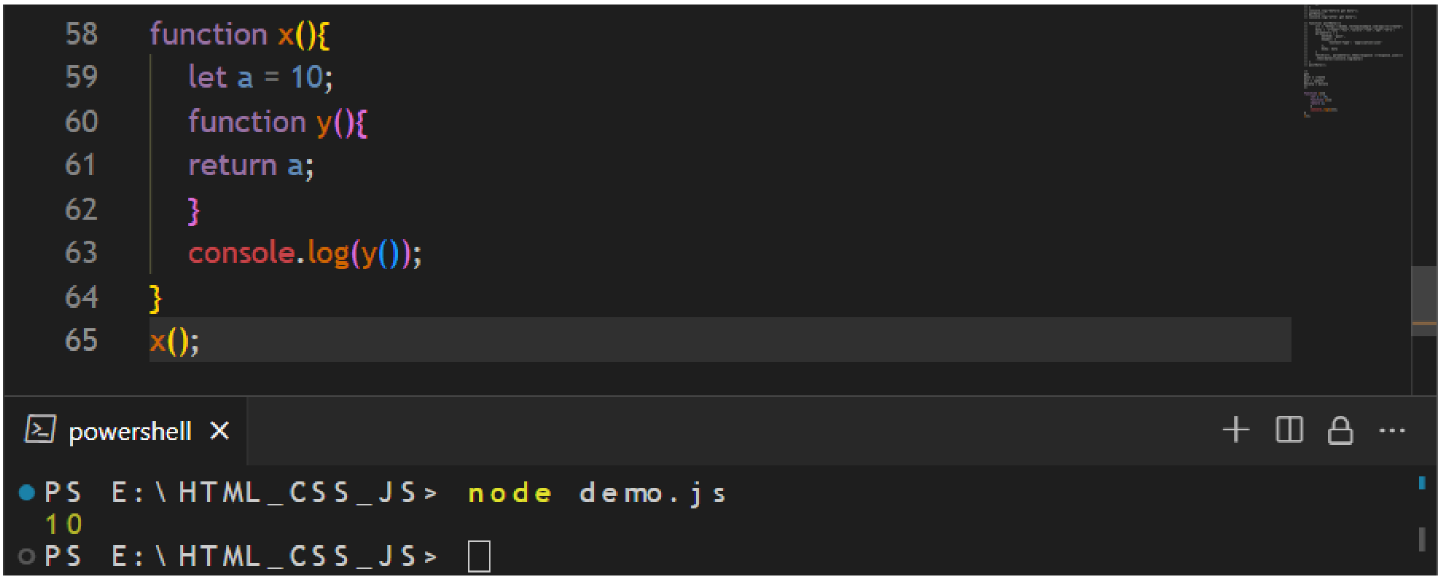
```
const a = 10;
{
const a = 20;
{
console.log(a);
}
console.log(a)
}
console.log(a)
```

```
PS E:\HTML_CSS_JS> node demo.js
20
20
10
PS E:\HTML_CSS_JS> 
```



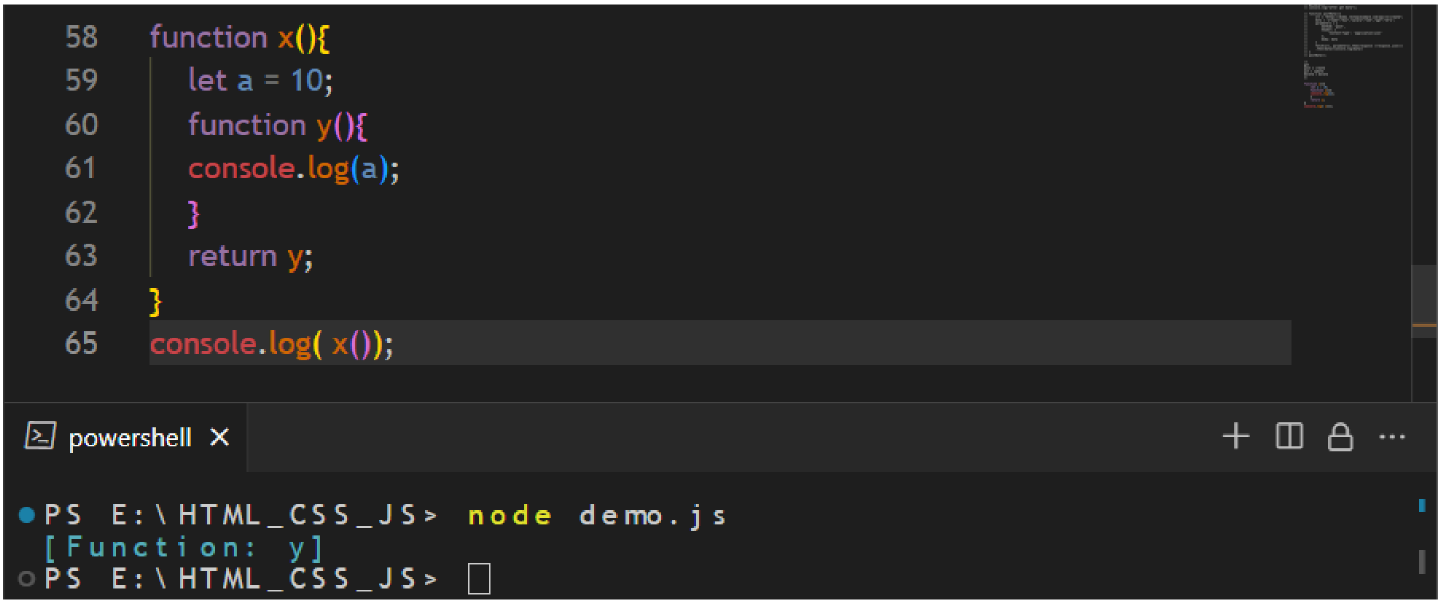
```
function x(){
    let a = 10;
    function y(){
    console.log(a);
    }
    y()
}
x();
```

```
PS E:\HTML_CSS_JS> node demo.js
10
PS E:\HTML_CSS_JS> 
```



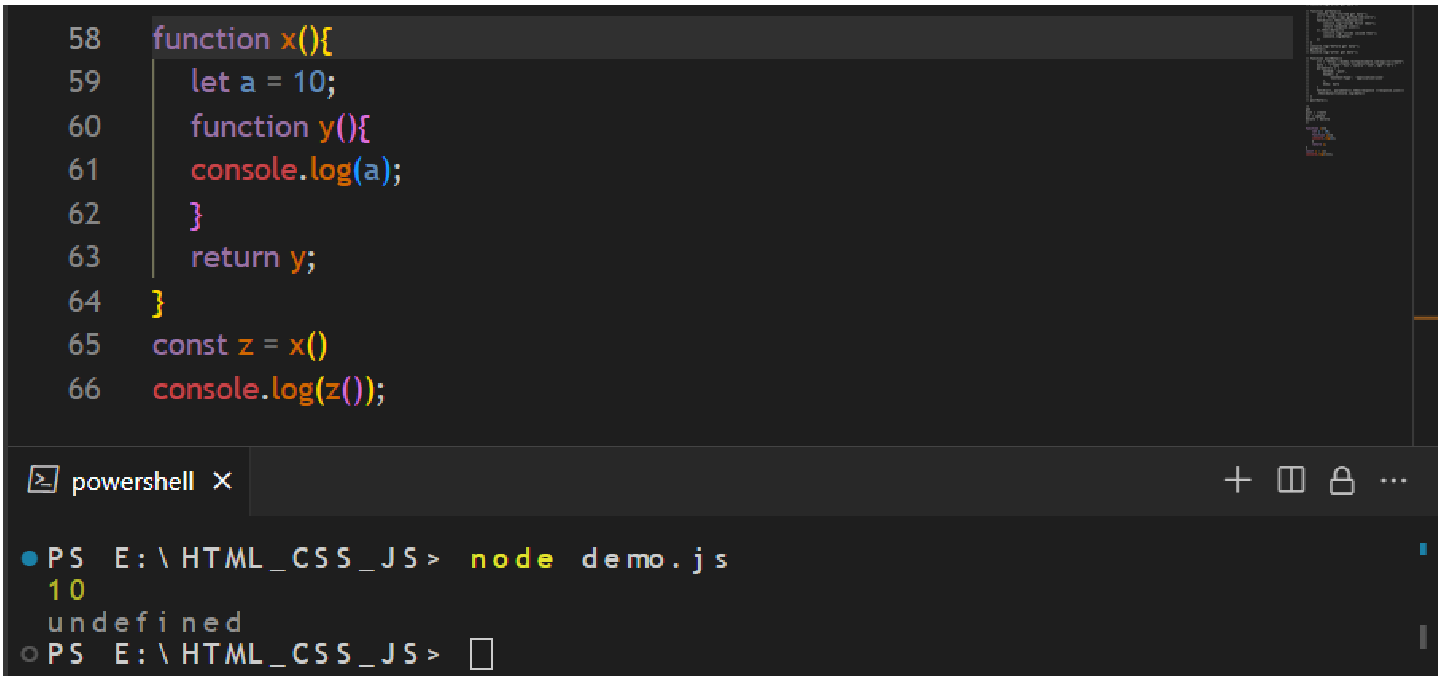
```
function x(){
    let a = 10;
    function y(){
    return a;
    }
    console.log(y());
}
x();
```

```
PS E:\HTML_CSS_JS> node demo.js
10
PS E:\HTML_CSS_JS> 
```



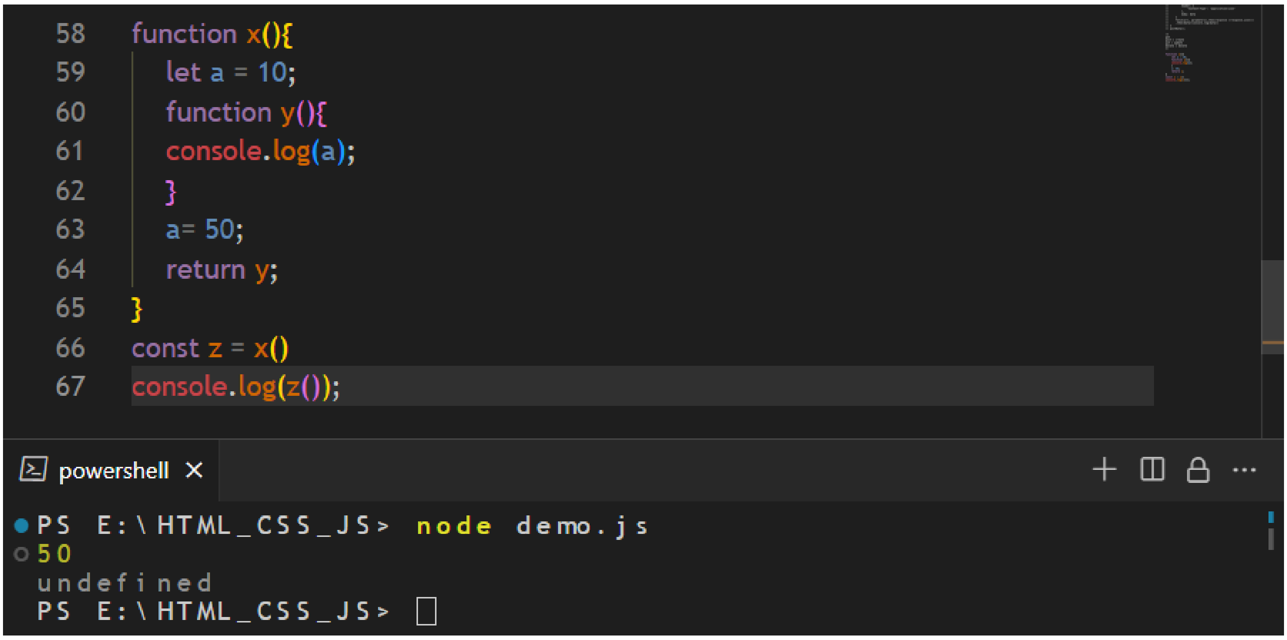
```javascript
function x(){
    let a = 10;
    function y(){
    console.log(a);
    }
    return y;
}
console.log( x());
```

```
PS E:\HTML_CSS_JS> node demo.js
[Function: y]
PS E:\HTML_CSS_JS> 
```



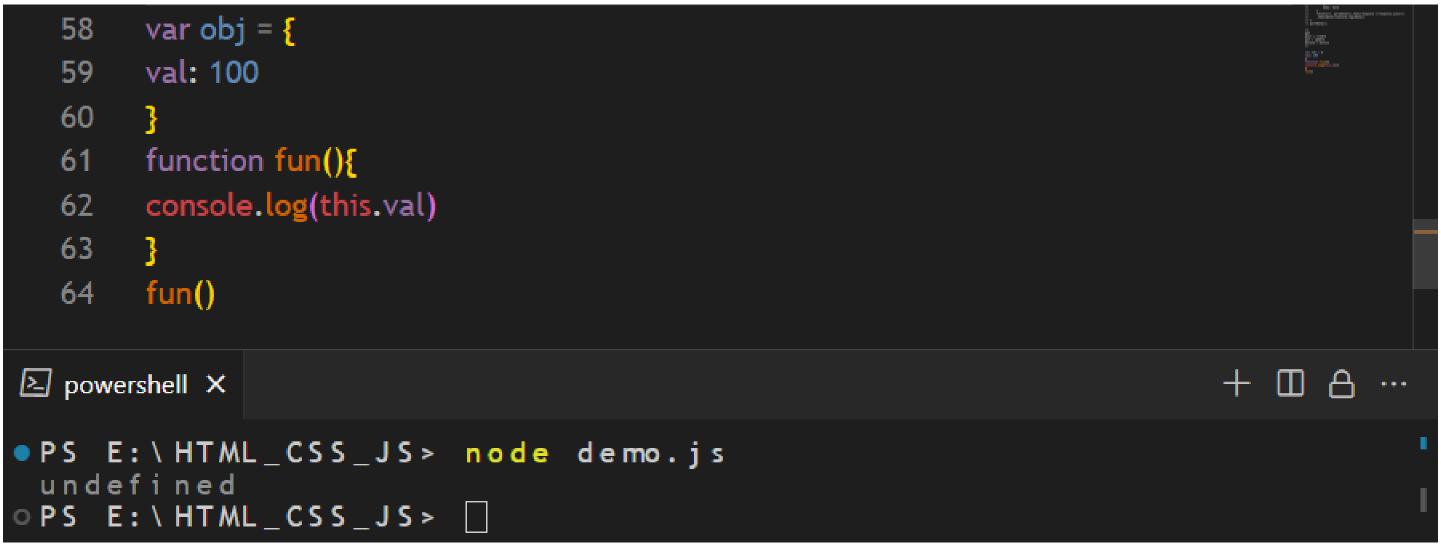
```javascript
function x(){
    let a = 10;
    function y(){
    console.log(a);
    }
    return y;
}
const z = x()
console.log(z());
```

```
PS E:\HTML_CSS_JS> node demo.js
10
undefined
PS E:\HTML_CSS_JS> 
```



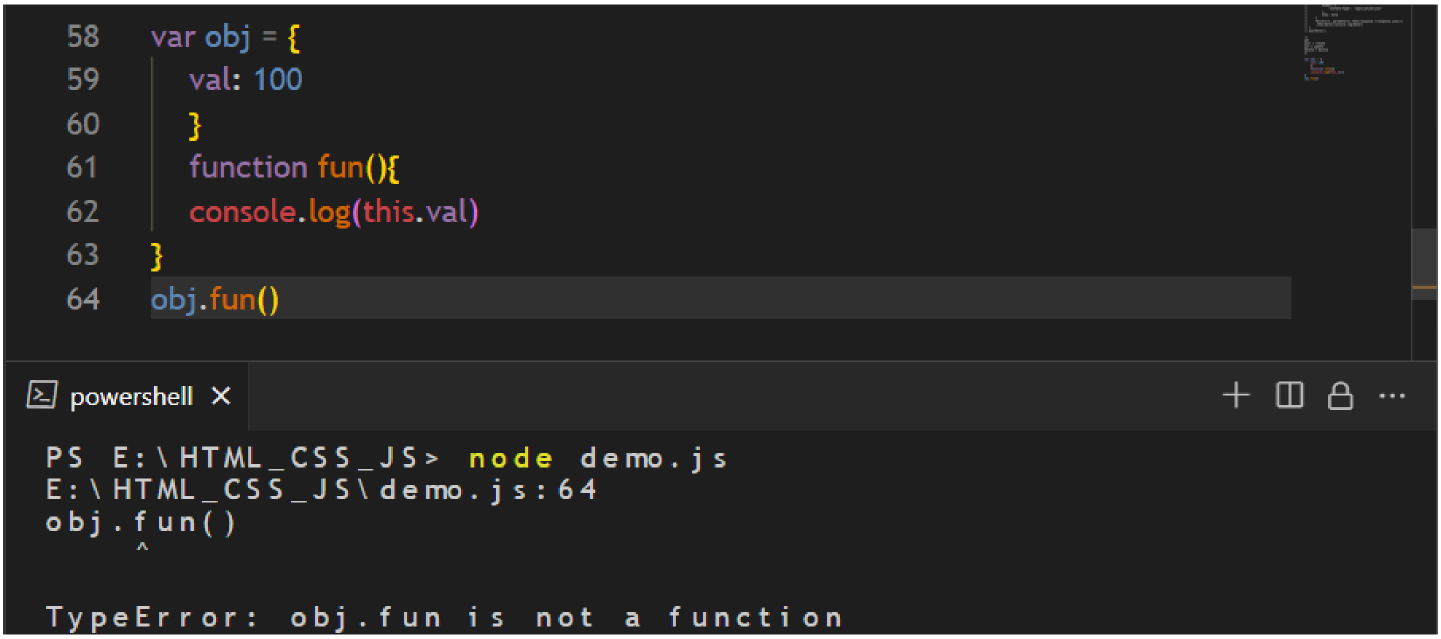
```javascript
function x(){
    let a = 10;
    function y(){
    console.log(a);
    }
    a= 50;
    return y;
}
const z = x()
console.log(z());
```

```
PS E:\HTML_CSS_JS> node demo.js
50
undefined
PS E:\HTML_CSS_JS> 
```



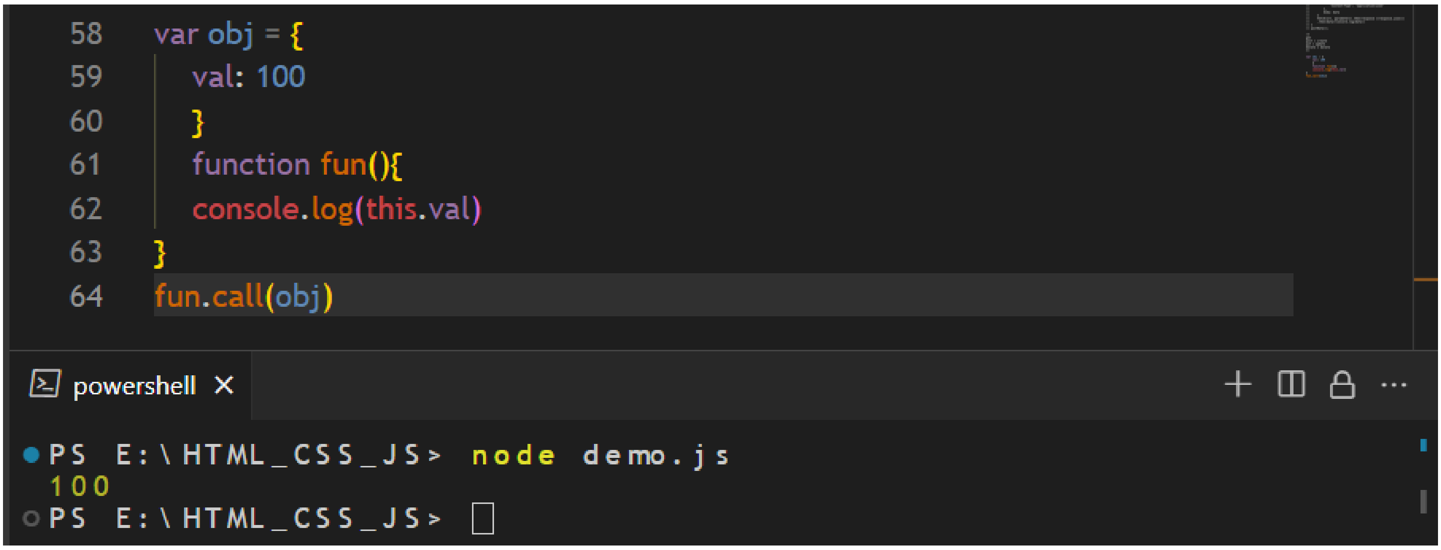
```javascript
var obj = {
val: 100
}
function fun(){
console.log(this.val)
}
fun()
```

```
PS E:\HTML_CSS_JS> node demo.js
undefined
PS E:\HTML_CSS_JS>
```



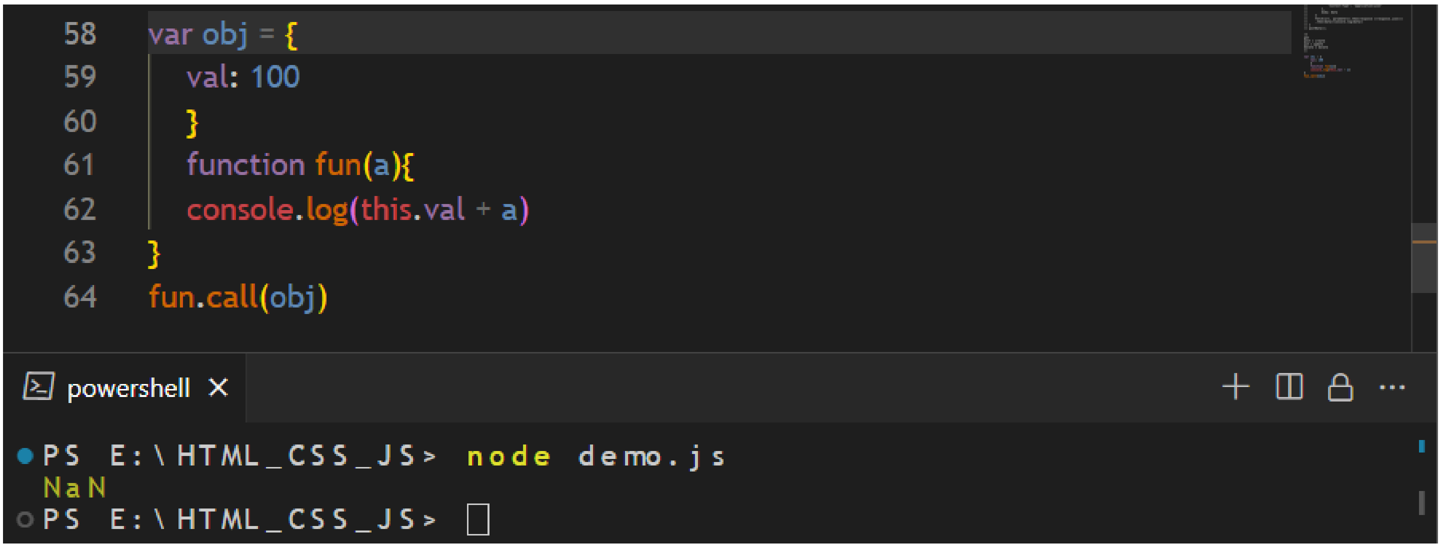
```javascript
var obj = {
    val: 100
    }
    function fun(){
    console.log(this.val)
}
obj.fun()
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:64
obj.fun()
    ^

TypeError: obj.fun is not a function
```



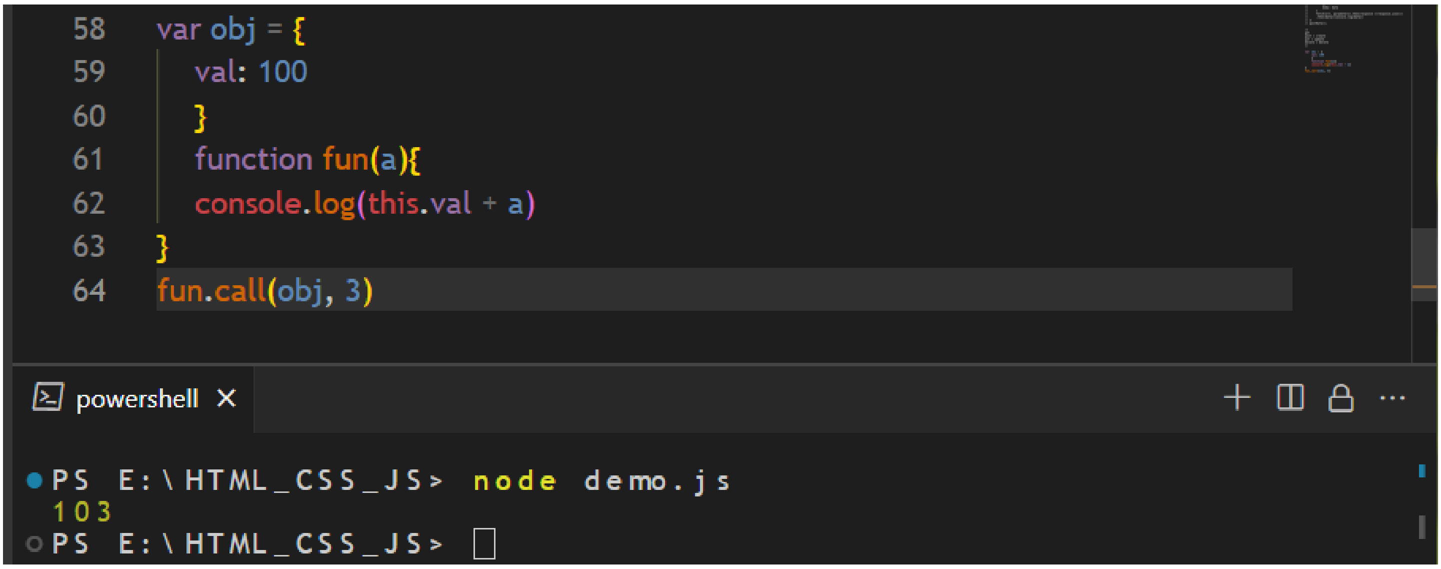
```javascript
var obj = {
    val: 100
    }
    function fun(){
    console.log(this.val)
}
fun.call(obj)
```

```
PS E:\HTML_CSS_JS> node demo.js
100
PS E:\HTML_CSS_JS>
```



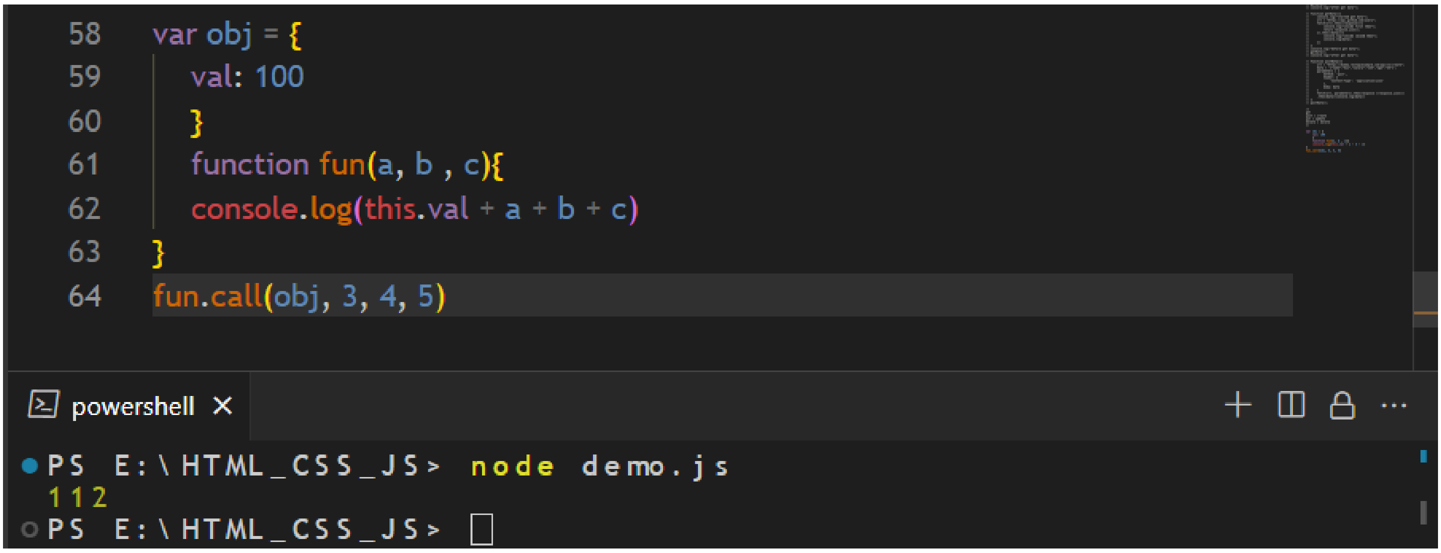
```javascript
var obj = {
    val: 100
    }
    function fun(a){
    console.log(this.val + a)
}
fun.call(obj)
```

```
PS E:\HTML_CSS_JS> node demo.js
NaN
PS E:\HTML_CSS_JS> 
```



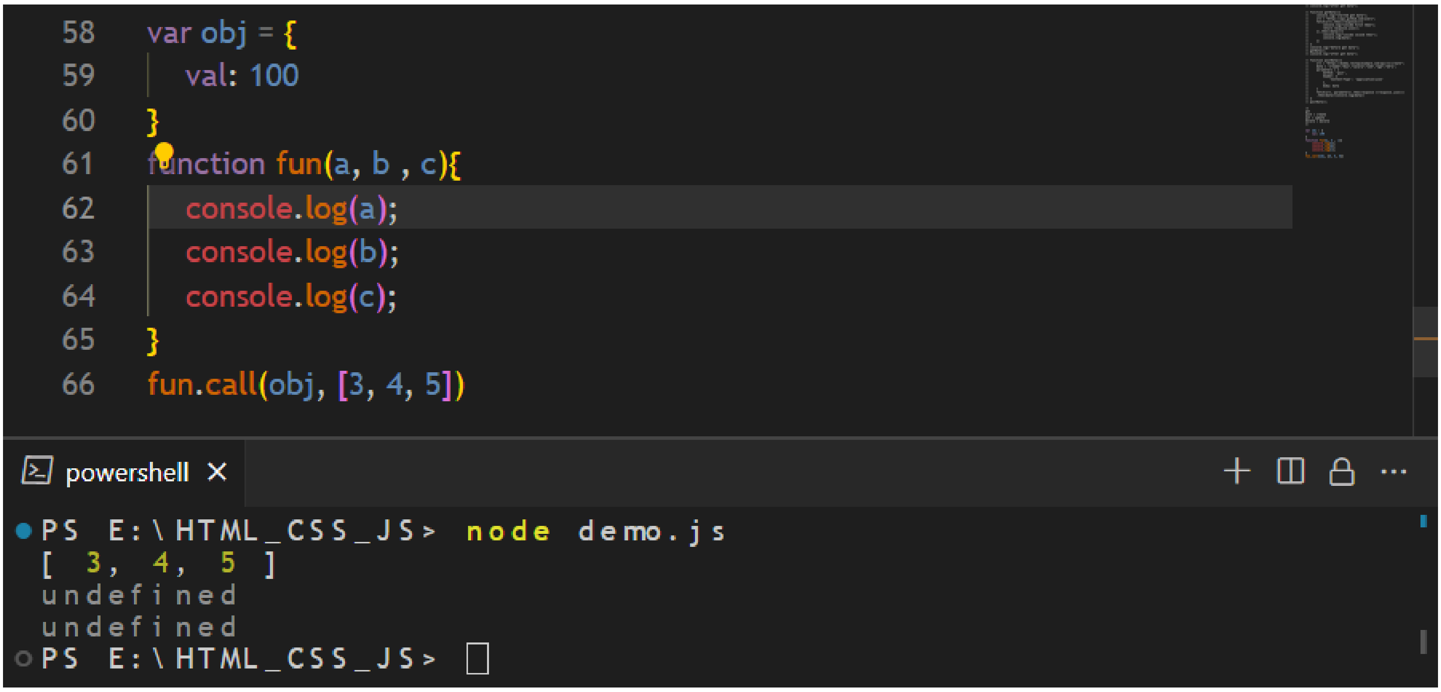
```javascript
var obj = {
    val: 100
    }
    function fun(a){
    console.log(this.val + a)
}
fun.call(obj, 3)
```

```
PS E:\HTML_CSS_JS> node demo.js
103
PS E:\HTML_CSS_JS> 
```



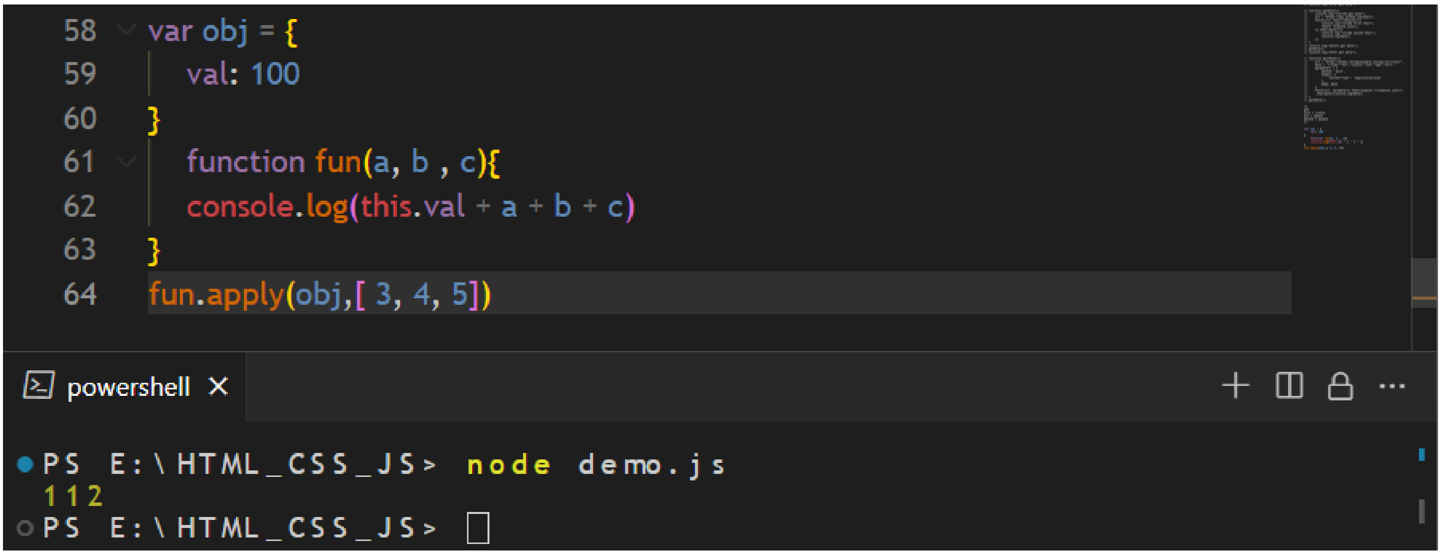
```javascript
var obj = {
    val: 100
    }
    function fun(a, b , c){
    console.log(this.val + a + b + c)
}
fun.call(obj, 3, 4, 5)
```

```
PS E:\HTML_CSS_JS> node demo.js
112
PS E:\HTML_CSS_JS> 
```



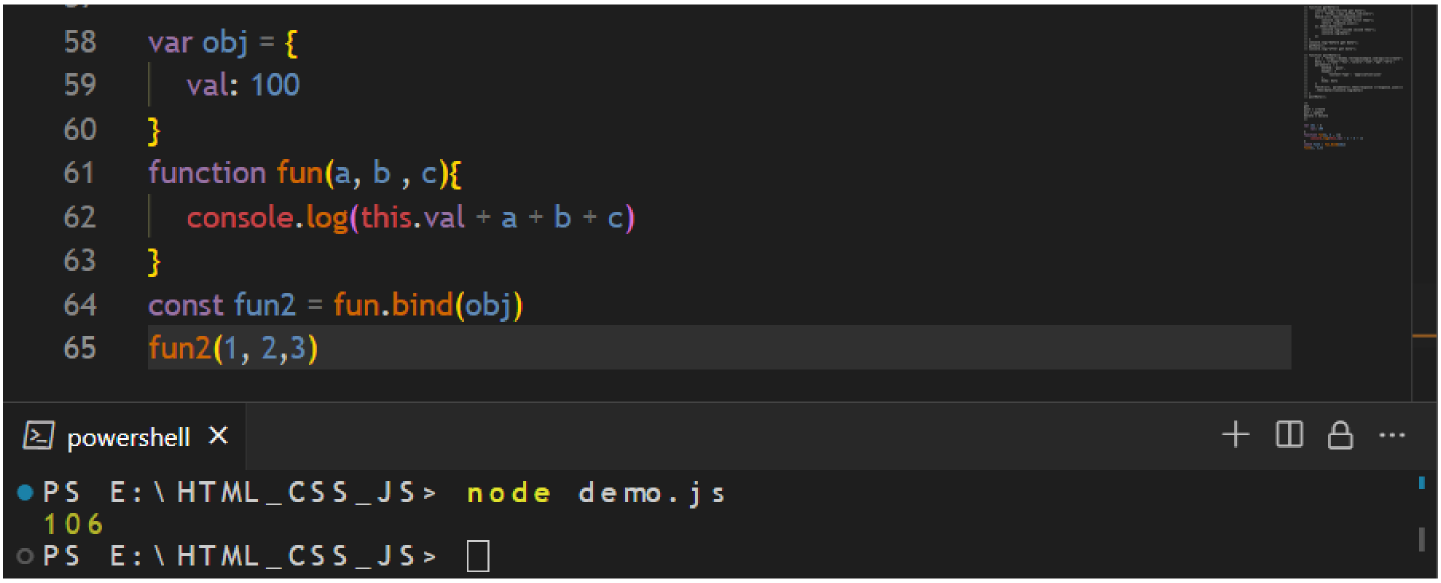
```javascript
var obj = {
    val: 100
}
function fun(a, b , c){
    console.log(a);
    console.log(b);
    console.log(c);
}
fun.call(obj, [3, 4, 5])
```

```
PS E:\HTML_CSS_JS> node demo.js
[ 3, 4, 5 ]
undefined
undefined
PS E:\HTML_CSS_JS>
```



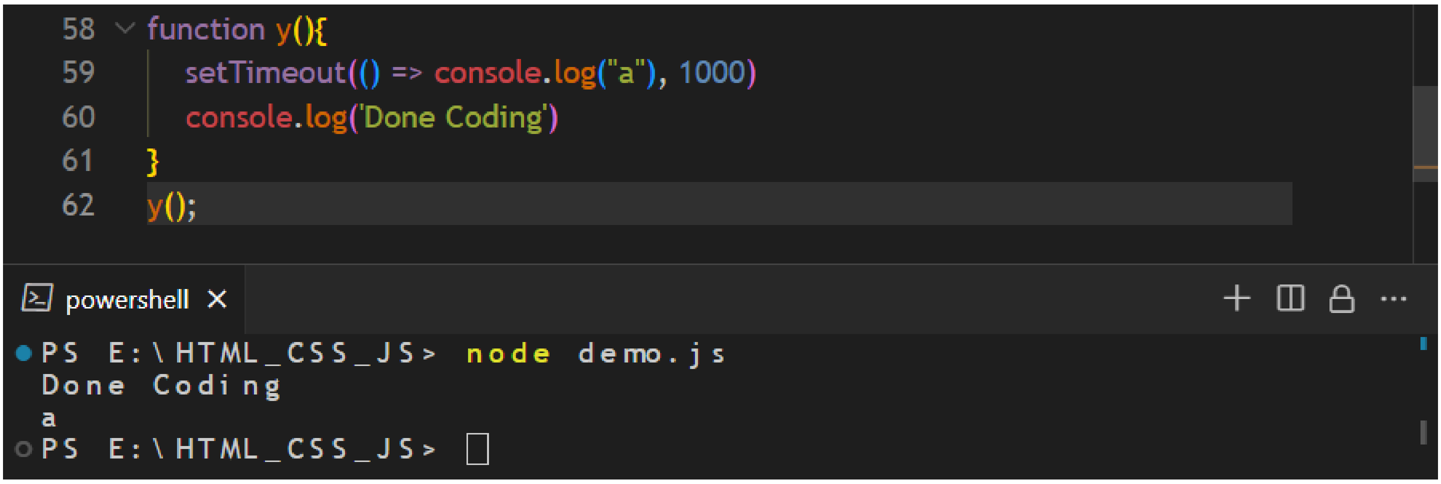
```javascript
var obj = {
    val: 100
}
    function fun(a, b , c){
    console.log(this.val + a + b + c)
}
fun.apply(obj,[ 3, 4, 5])
```

```
PS E:\HTML_CSS_JS> node demo.js
112
PS E:\HTML_CSS_JS>
```



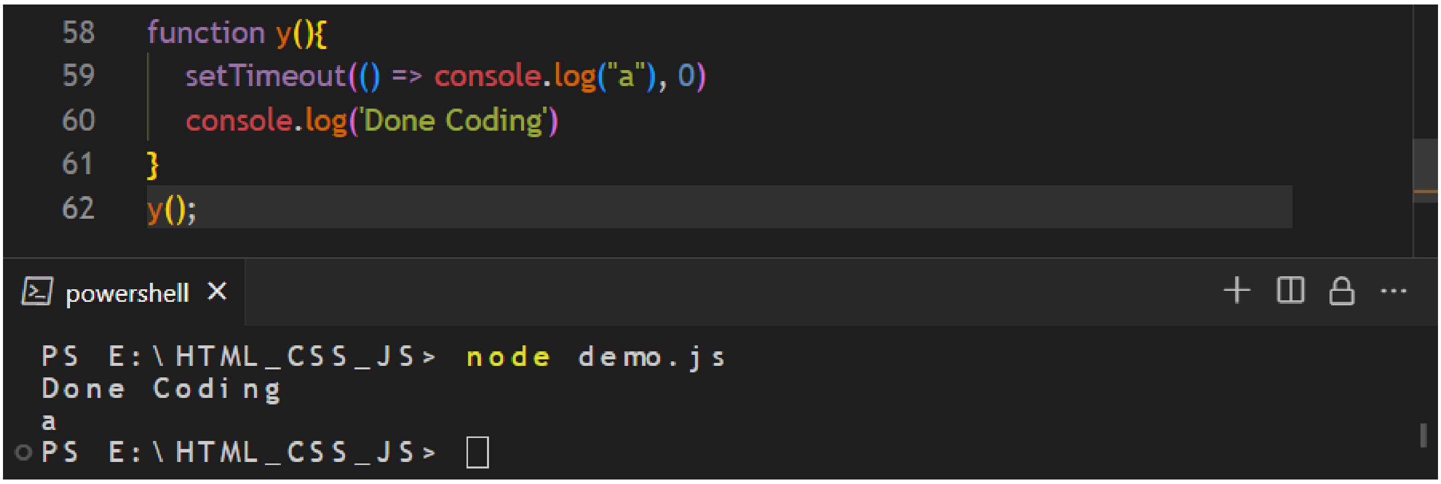
```javascript
var obj = {
    val: 100
}
function fun(a, b , c){
    console.log(this.val + a + b + c)
}
const fun2 = fun.bind(obj)
fun2(1, 2,3)
```

```
PS E:\HTML_CSS_JS> node demo.js
106
PS E:\HTML_CSS_JS>
```



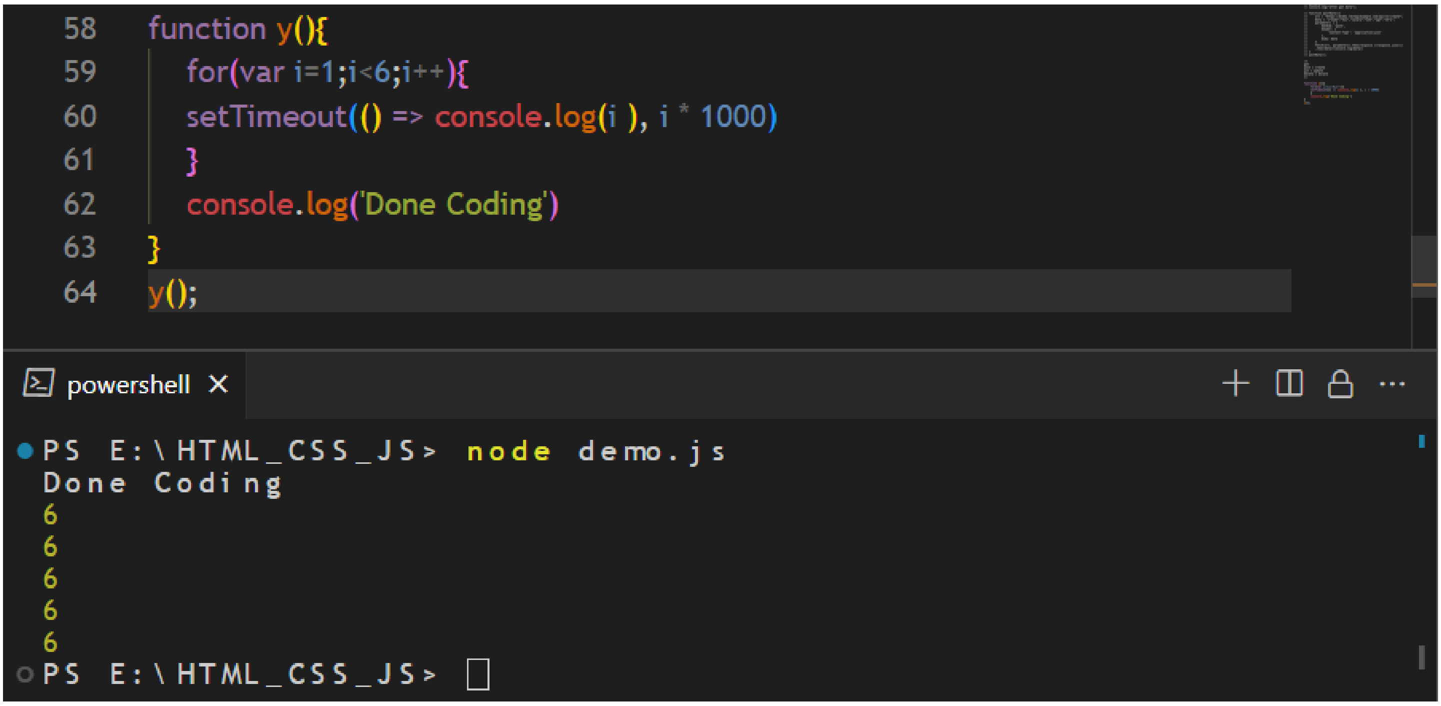
```javascript
function y(){
    setTimeout(() => console.log("a"), 1000)
    console.log('Done Coding')
}
y();
```

```
PS E:\HTML_CSS_JS> node demo.js
Done Coding
a
PS E:\HTML_CSS_JS>
```



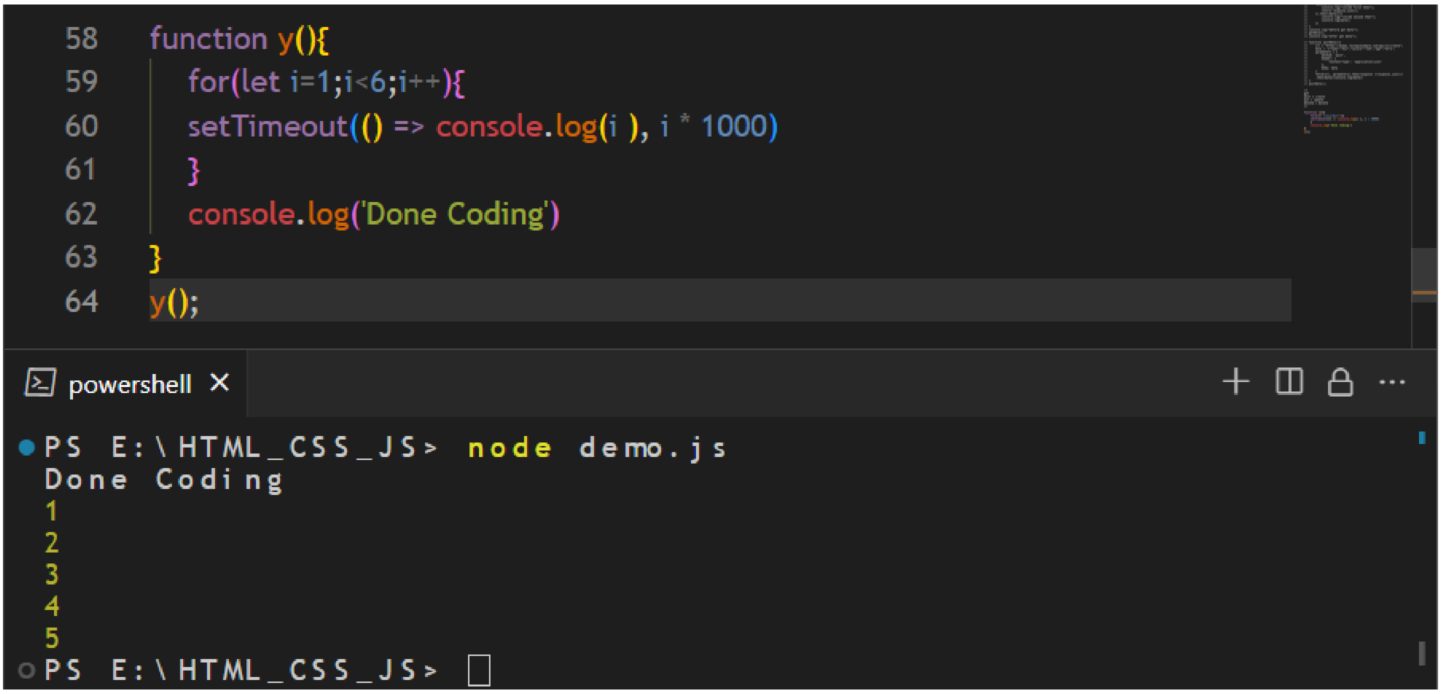
```javascript
function y(){
    setTimeout(() => console.log("a"), 0)
    console.log('Done Coding')
}
y();
```

```
PS E:\HTML_CSS_JS> node demo.js
Done Coding
a
PS E:\HTML_CSS_JS>
```



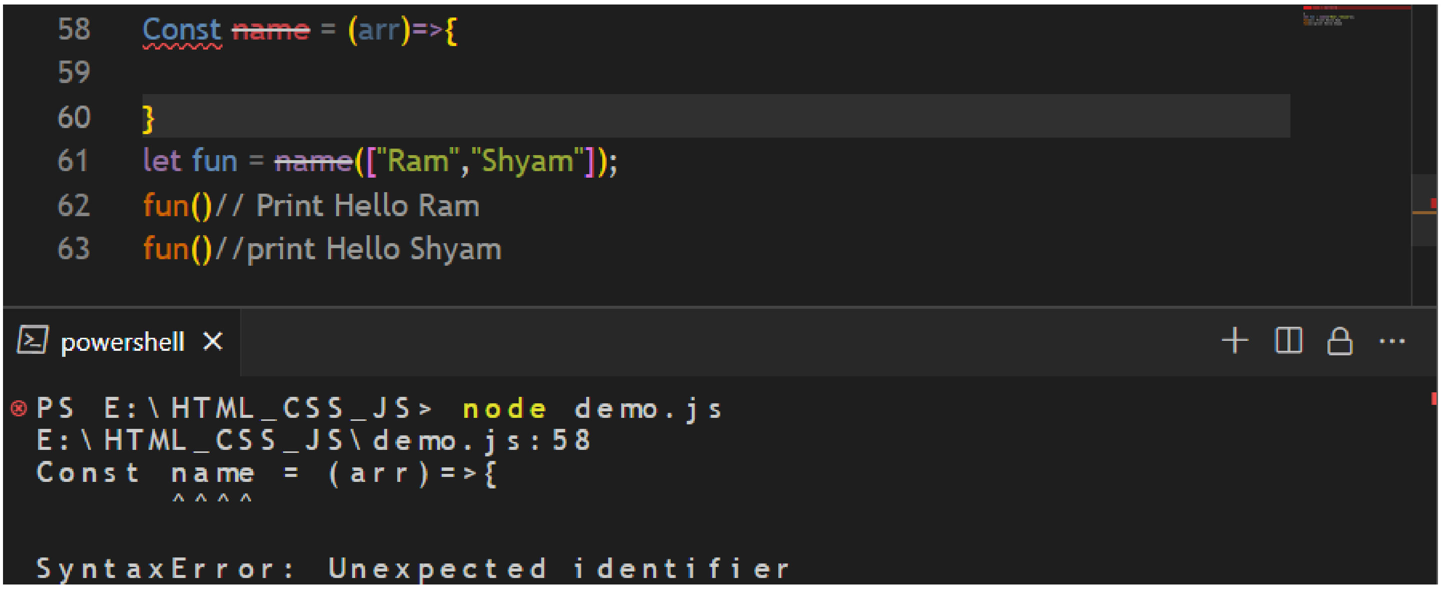
```javascript
function y(){
    for(var i=1;i<6;i++){
    setTimeout(() => console.log(i ), i * 1000)
    }
    console.log('Done Coding')
}
y();
```

```
PS E:\HTML_CSS_JS> node demo.js
Done Coding
6
6
6
6
6
PS E:\HTML_CSS_JS>
```



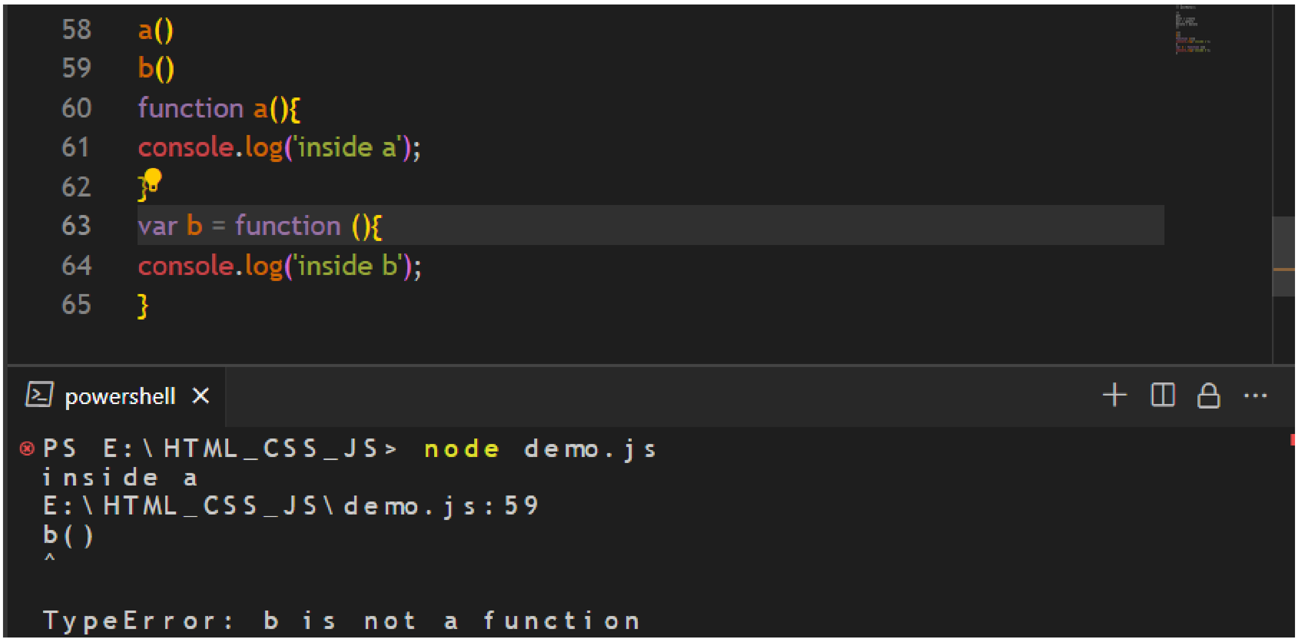
```javascript
function y(){
    for(let i=1;i<6;i++){
    setTimeout(() => console.log(i ), i * 1000)
    }
    console.log('Done Coding')
}
y();
```

```
PS E:\HTML_CSS_JS> node demo.js
Done Coding
1
2
3
4
5
PS E:\HTML_CSS_JS>
```



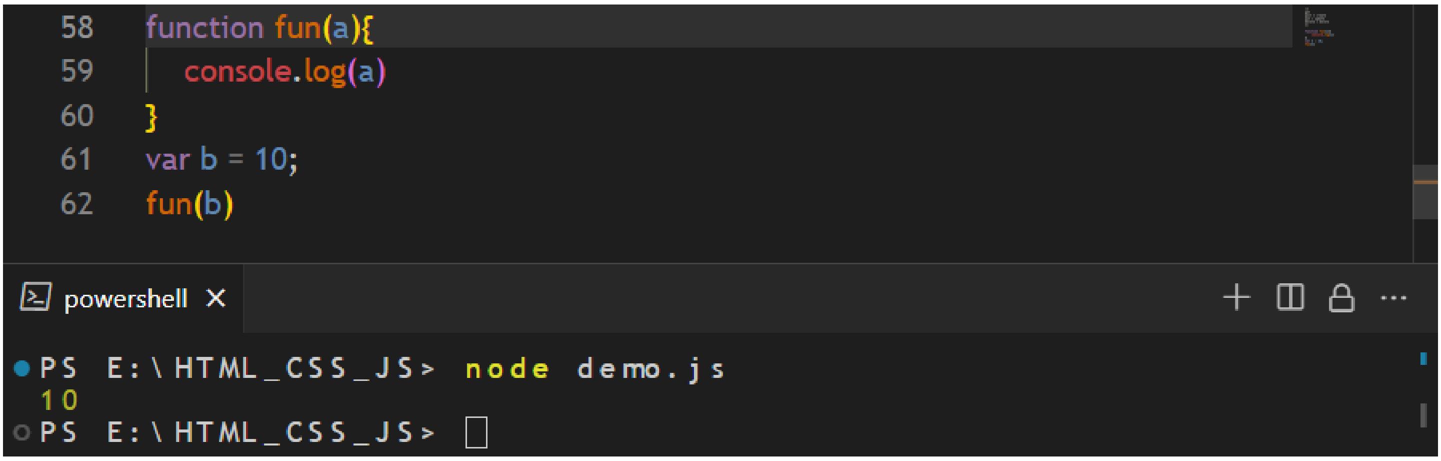
```javascript
Const name = (arr)=>{

}
let fun = name(["Ram","Shyam"]);
fun()// Print Hello Ram
fun()//print Hello Shyam
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:58
Const name = (arr)=>{
      ^^^^

SyntaxError: Unexpected identifier
```



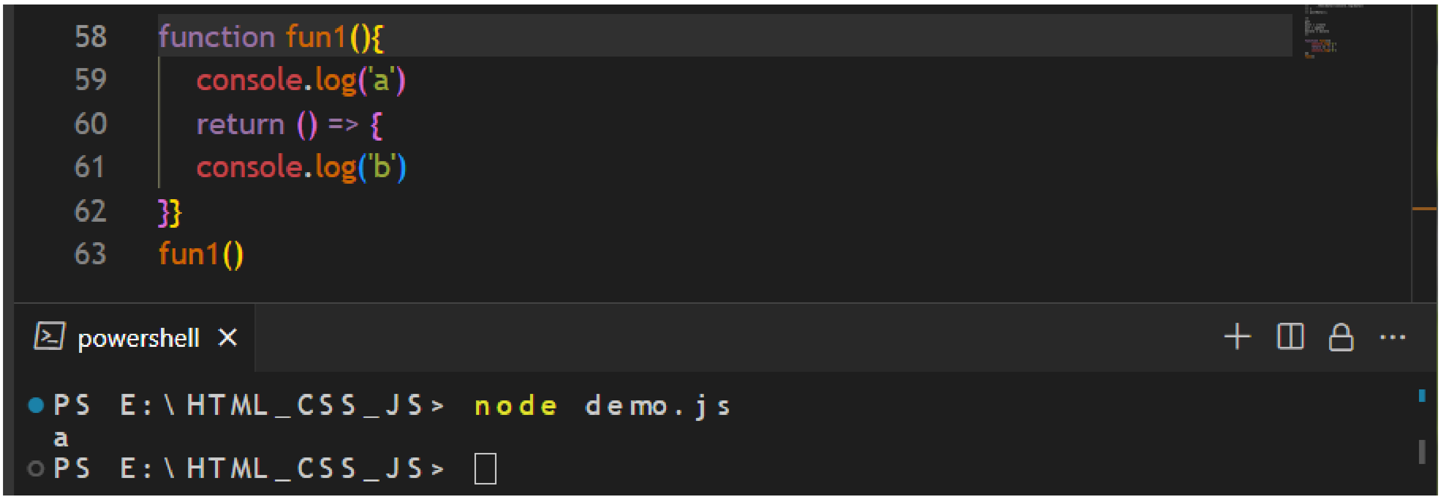
```javascript
a()
b()
function a(){
console.log('inside a');
}
var b = function (){
console.log('inside b');
}
```

```
PS E:\HTML_CSS_JS> node demo.js
inside a
E:\HTML_CSS_JS\demo.js:59
b()
^

TypeError: b is not a function
```



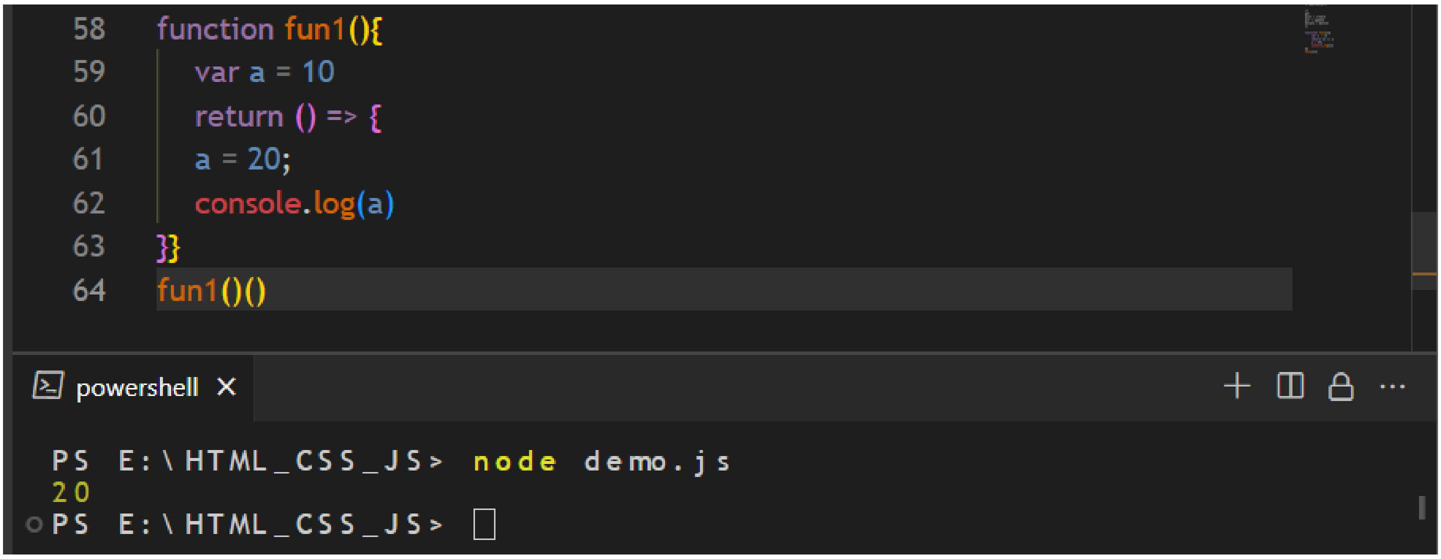
```javascript
function fun(a){
  console.log(a)
}
var b = 10;
fun(b)
```

```
PS E:\HTML_CSS_JS> node demo.js
10
PS E:\HTML_CSS_JS> 
```



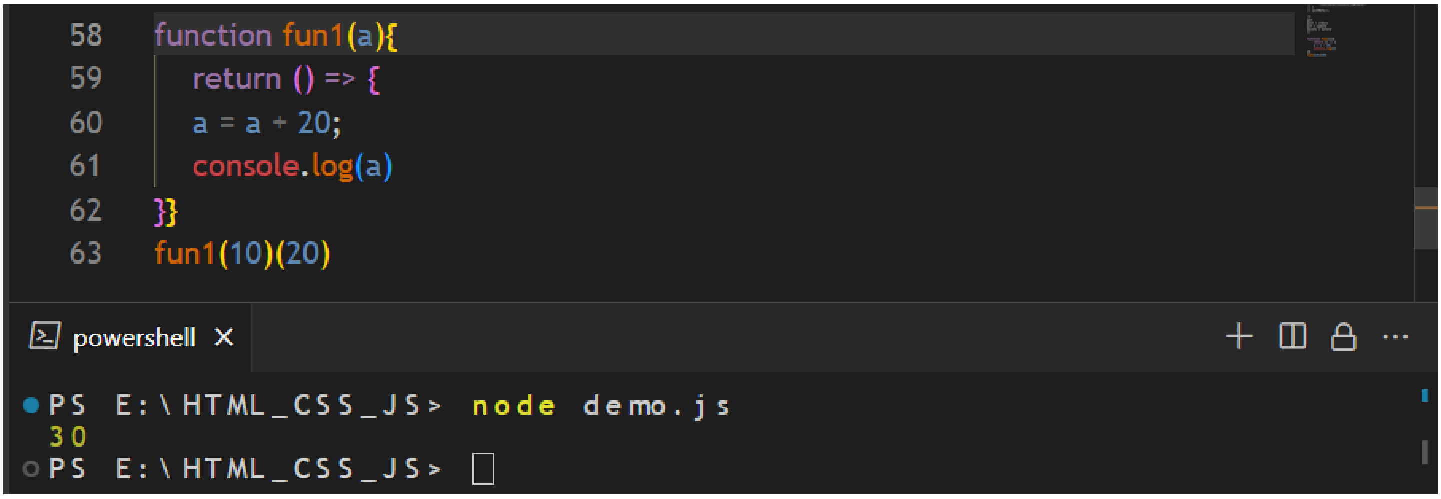
```javascript
function fun1(){
  console.log('a')
  return () => {
  console.log('b')
}}
fun1()
```

```
PS E:\HTML_CSS_JS> node demo.js
a
PS E:\HTML_CSS_JS> 
```



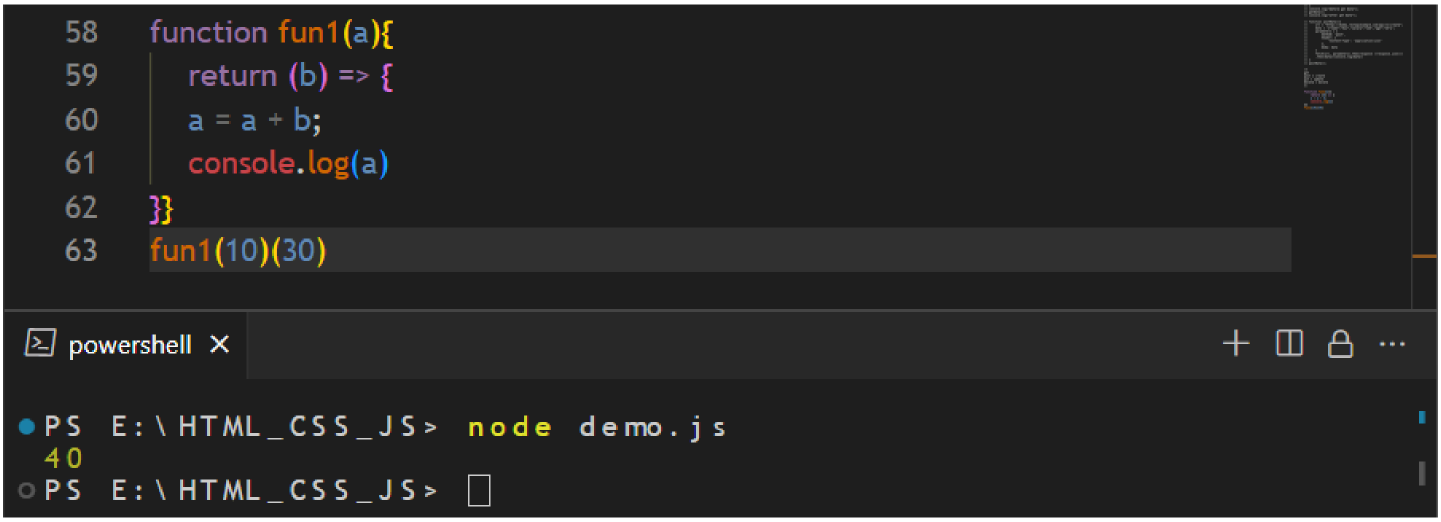
```javascript
function fun1(){
  var a = 10
  return () => {
  a = 20;
  console.log(a)
}}
fun1()()
```

```
PS E:\HTML_CSS_JS> node demo.js
20
PS E:\HTML_CSS_JS> 
```



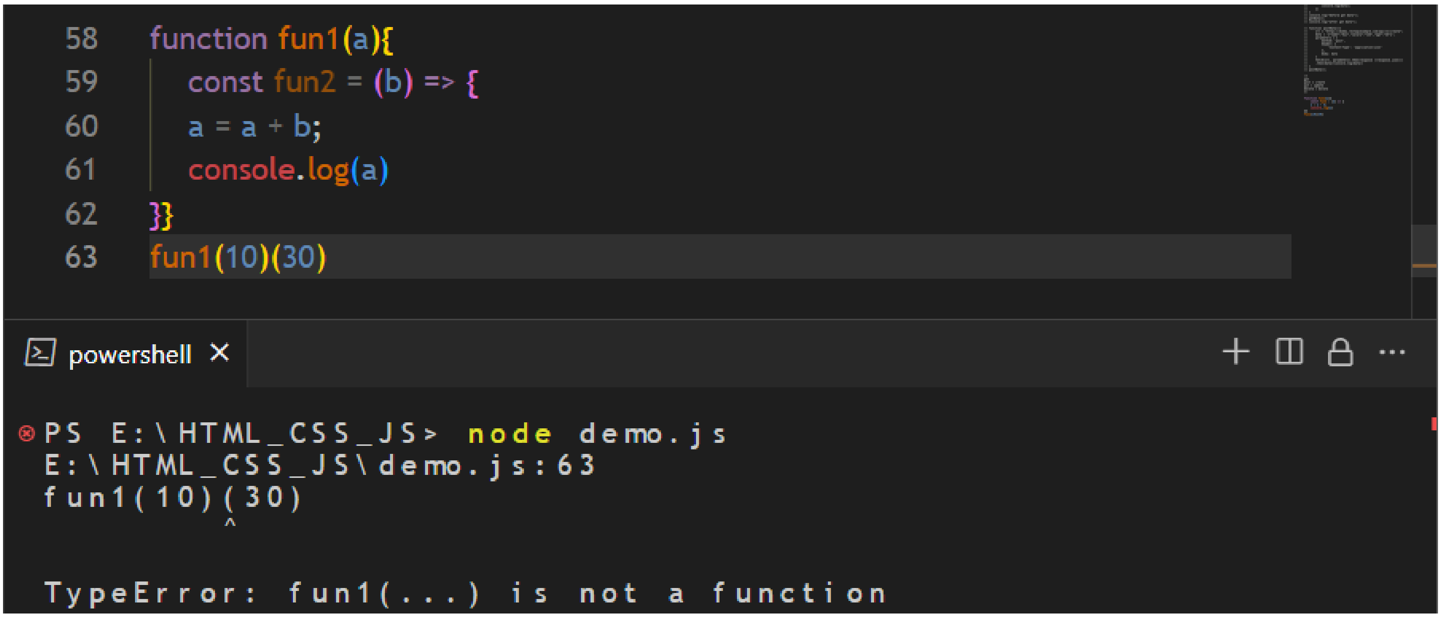
```javascript
function fun1(a){
    return () => {
    a = a + 20;
    console.log(a)
}}
fun1(10)(20)
```

```
PS E:\HTML_CSS_JS> node demo.js
30
PS E:\HTML_CSS_JS> 
```



```javascript
function fun1(a){
    return (b) => {
    a = a + b;
    console.log(a)
}}
fun1(10)(30)
```

```
PS E:\HTML_CSS_JS> node demo.js
40
PS E:\HTML_CSS_JS> 
```



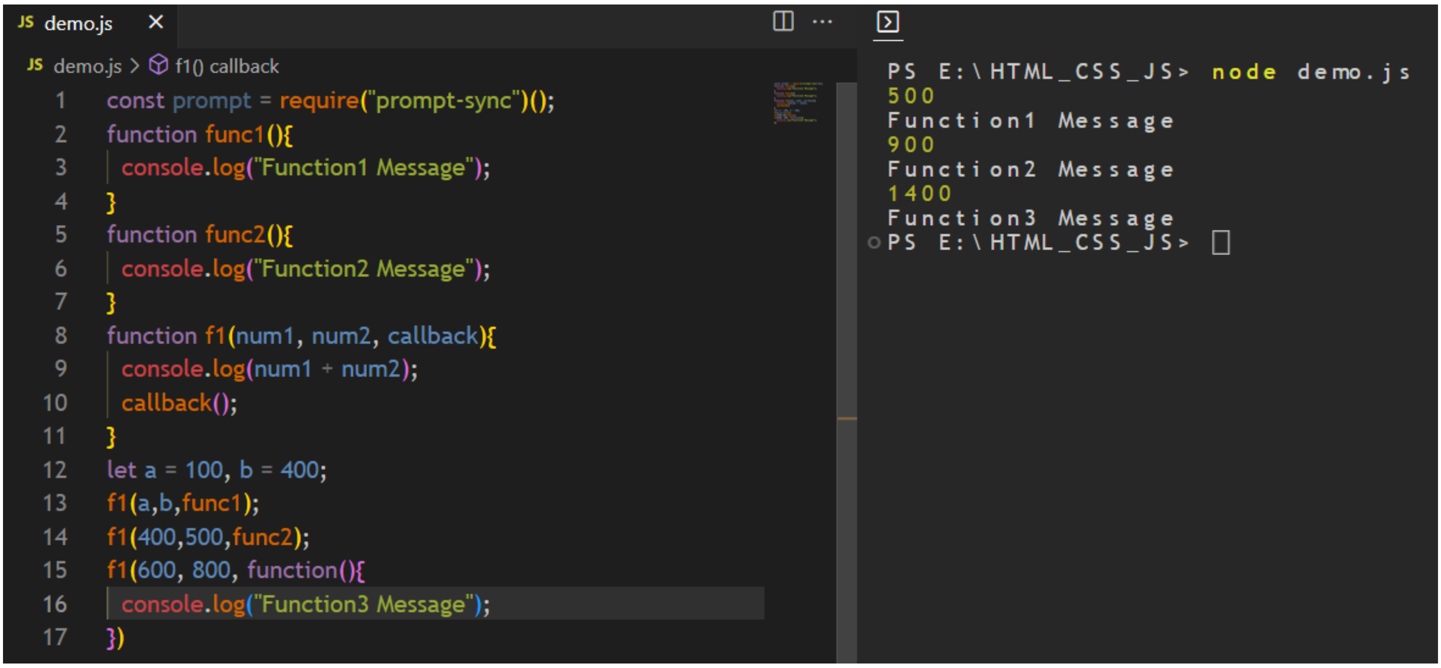
```javascript
function fun1(a){
    const fun2 = (b) => {
    a = a + b;
    console.log(a)
}}
fun1(10)(30)
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:63
fun1(10)(30)
        ^

TypeError: fun1(...) is not a function
```

Callback Function:

A callback is a function passed as an argument to another function. This technique allows a function to call another function. A callback function can run after another function has finished.

A callback function is a concept in programming where a function is passed as an argument to another function and is executed at a later time or under certain conditions. Callback functions are used in various programming languages, but they are particularly prevalent in JavaScript due to its asynchronous nature.

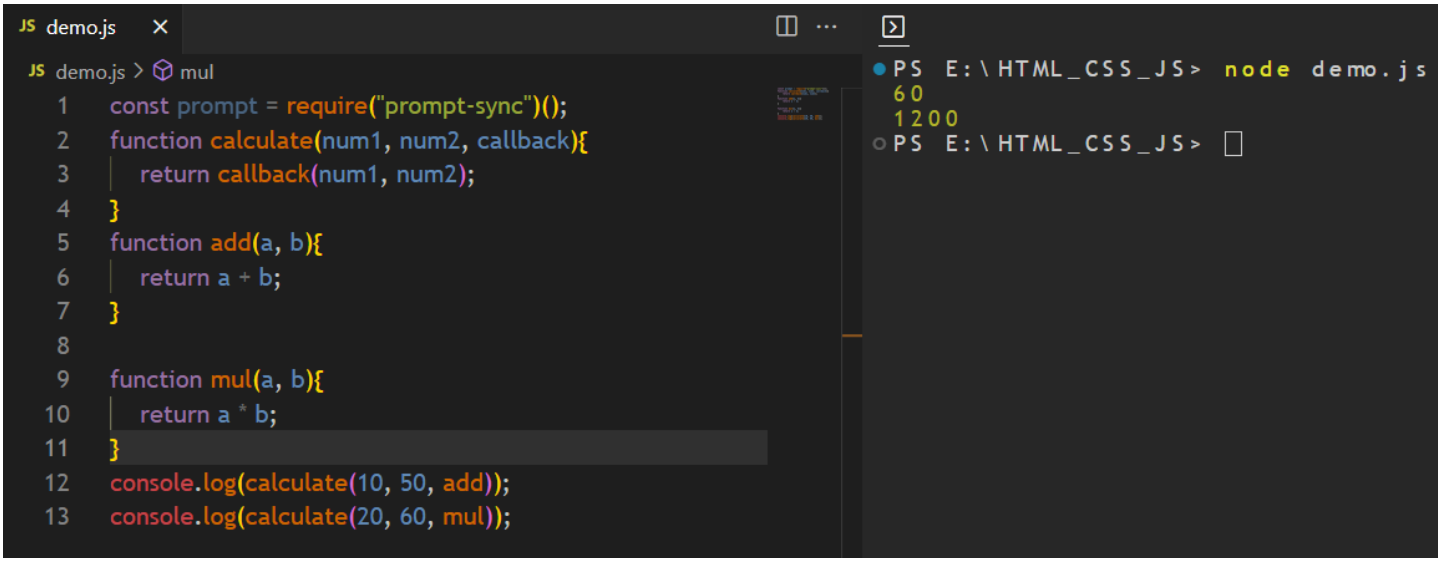


```javascript
const prompt = require("prompt-sync")();
function func1(){
    console.log("Function1 Message");
}
function func2(){
    console.log("Function2 Message");
}
function f1(num1, num2, callback){
    console.log(num1 + num2);
    callback();
}
let a = 100, b = 400;
f1(a,b,func1);
f1(400,500,func2);
f1(600, 800, function(){
    console.log("Function3 Message");
})
```

```
PS E:\HTML_CSS_JS> node demo.js
500
Function1 Message
900
Function2 Message
1400
Function3 Message
PS E:\HTML_CSS_JS>
```

Passing Function As Argument:

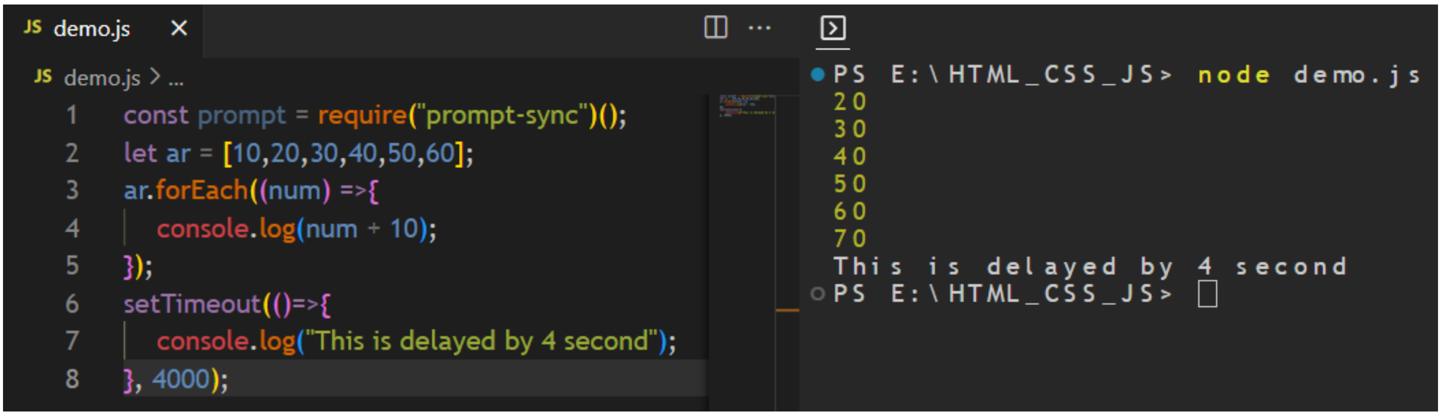
In many programming languages, functions are treated as first-class citizens, which means they can be assigned to variables, passed as arguments to other functions, and returned from functions just like any other data type. Callback functions take advantage of this feature by allowing you to pass a function as an argument to another function.



```javascript
const prompt = require("prompt-sync")();
function calculate(num1, num2, callback){
    return callback(num1, num2);
}
function add(a, b){
    return a + b;
}

function mul(a, b){
    return a * b;
}
console.log(calculate(10, 50, add));
console.log(calculate(20, 60, mul));
```

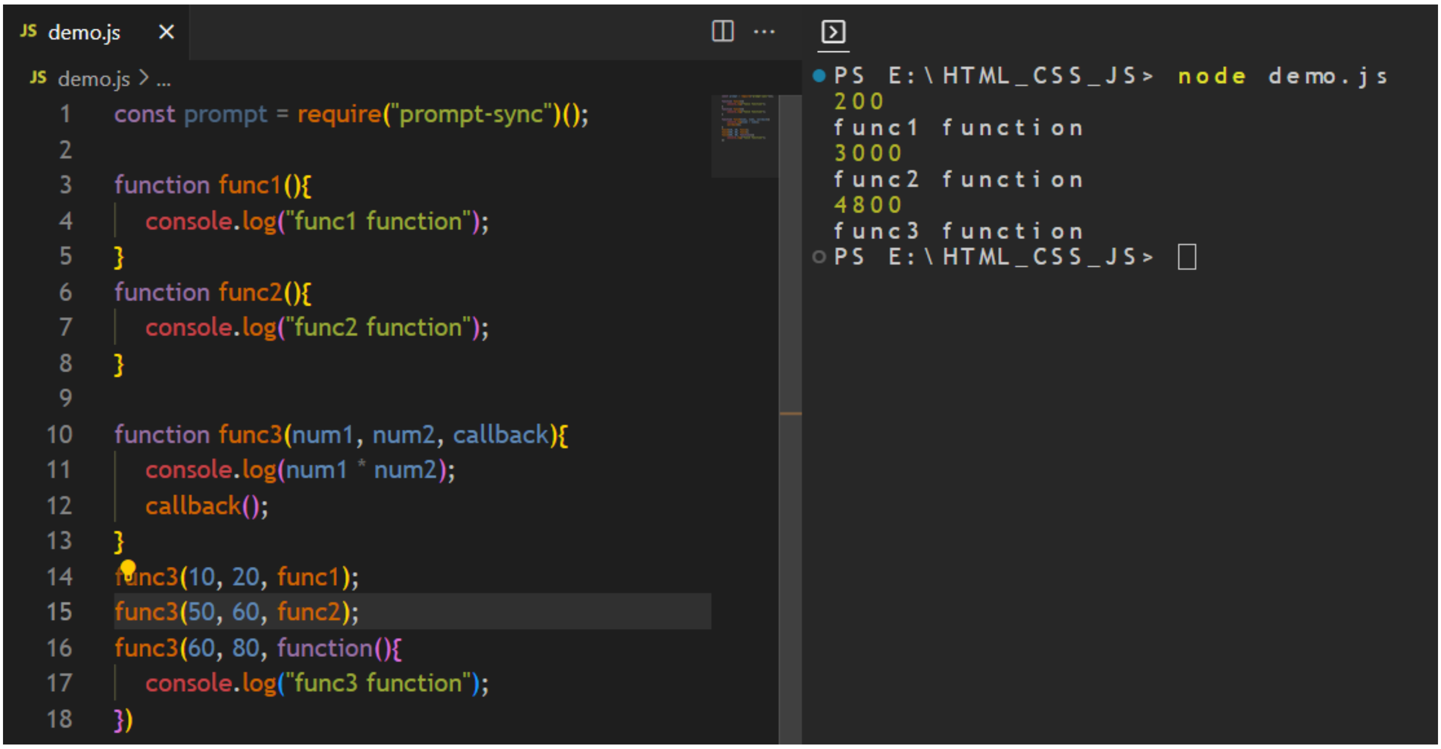
```
PS E:\HTML_CSS_JS> node demo.js
60
1200
PS E:\HTML_CSS_JS>
```

These are executed later, typically after an asynchronous operation like reading a file or making an API request has completed. Asynchronous callbacks are essential for non-blocking code execution. we have an example of both synchronous and asynchronous callback functions. The synchronous forEach method of an array uses a callback function, while the asynchronous setTimeout function uses a callback for delayed execution.



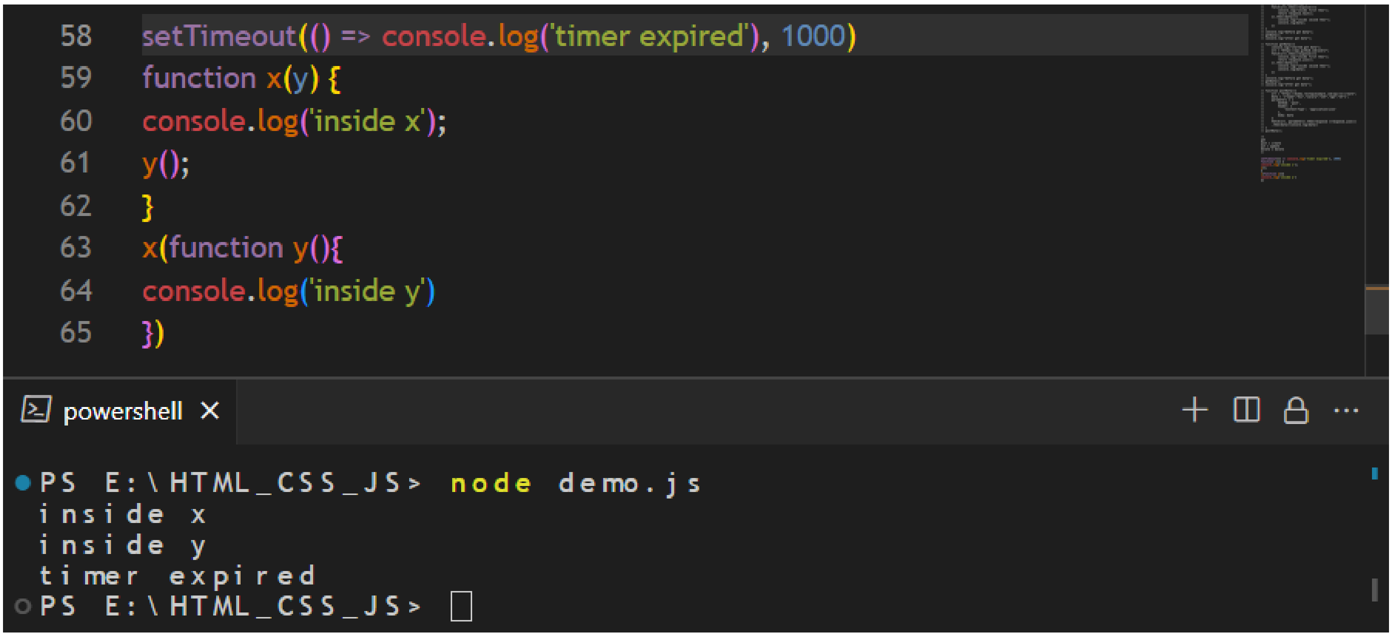
```
const prompt = require("prompt-sync")();
let ar = [10,20,30,40,50,60];
ar.forEach((num) =>{
    console.log(num + 10);
});
setTimeout(()=>{
    console.log("This is delayed by 4 second");
}, 4000);
```

```
PS E:\HTML_CSS_JS> node demo.js
20
30
40
50
60
70
This is delayed by 4 second
PS E:\HTML_CSS_JS> 
```



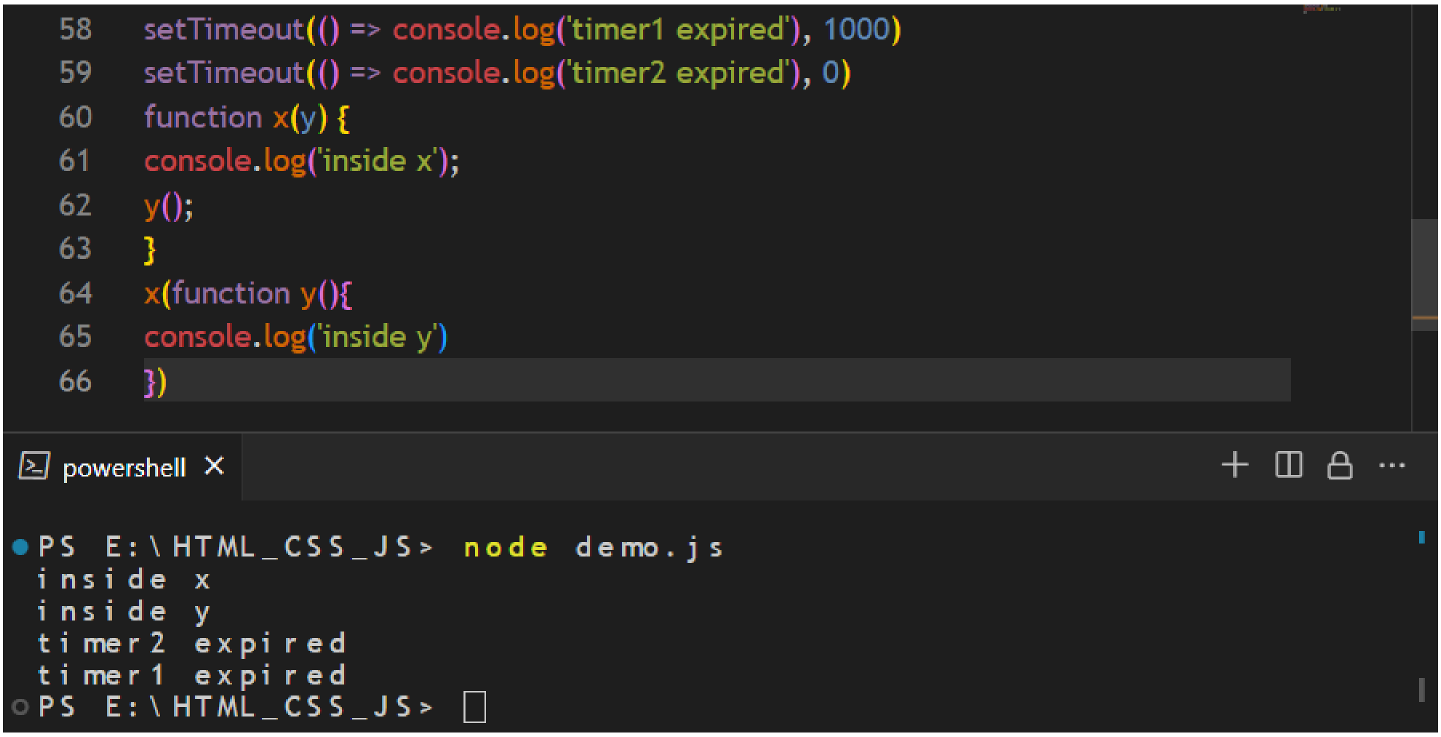
```
const prompt = require("prompt-sync")();

function func1(){
    console.log("func1 function");
}
function func2(){
    console.log("func2 function");
}

function func3(num1, num2, callback){
    console.log(num1 * num2);
    callback();
}
func3(10, 20, func1);
func3(50, 60, func2);
func3(60, 80, function(){
    console.log("func3 function");
})
```

```
PS E:\HTML_CSS_JS> node demo.js
200
func1 function
3000
func2 function
4800
func3 function
PS E:\HTML_CSS_JS> 
```



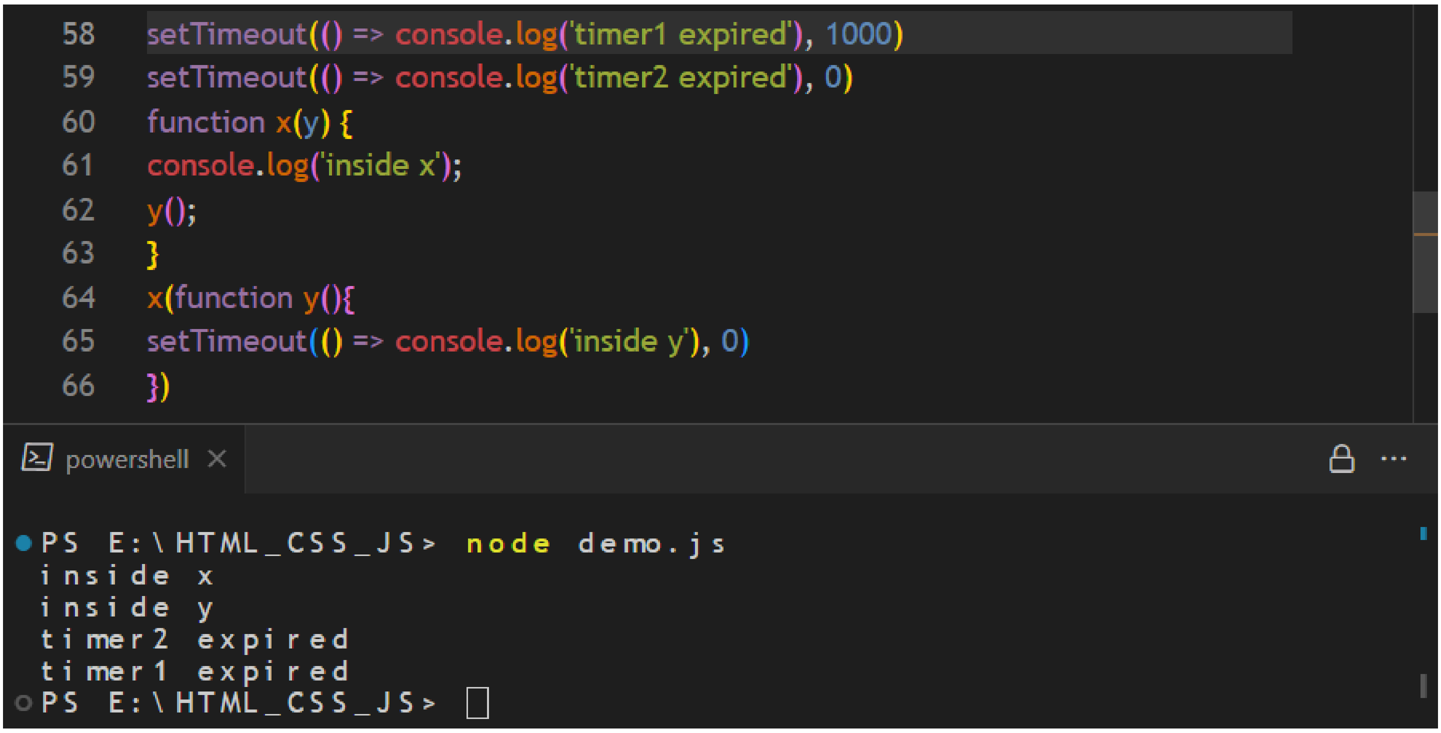
```
setTimeout(() => console.log('timer expired'), 1000)
function x(y) {
console.log('inside x');
y();
}
x(function y(){
console.log('inside y')
})
```

```
PS E:\HTML_CSS_JS> node demo.js
inside x
inside y
timer expired
PS E:\HTML_CSS_JS> 
```



```
setTimeout(() => console.log('timer1 expired'), 1000)
setTimeout(() => console.log('timer2 expired'), 0)
function x(y) {
console.log('inside x');
y();
}
x(function y(){
console.log('inside y')
})
```

```
PS E:\HTML_CSS_JS> node demo.js
inside x
inside y
timer2 expired
timer1 expired
PS E:\HTML_CSS_JS>
```

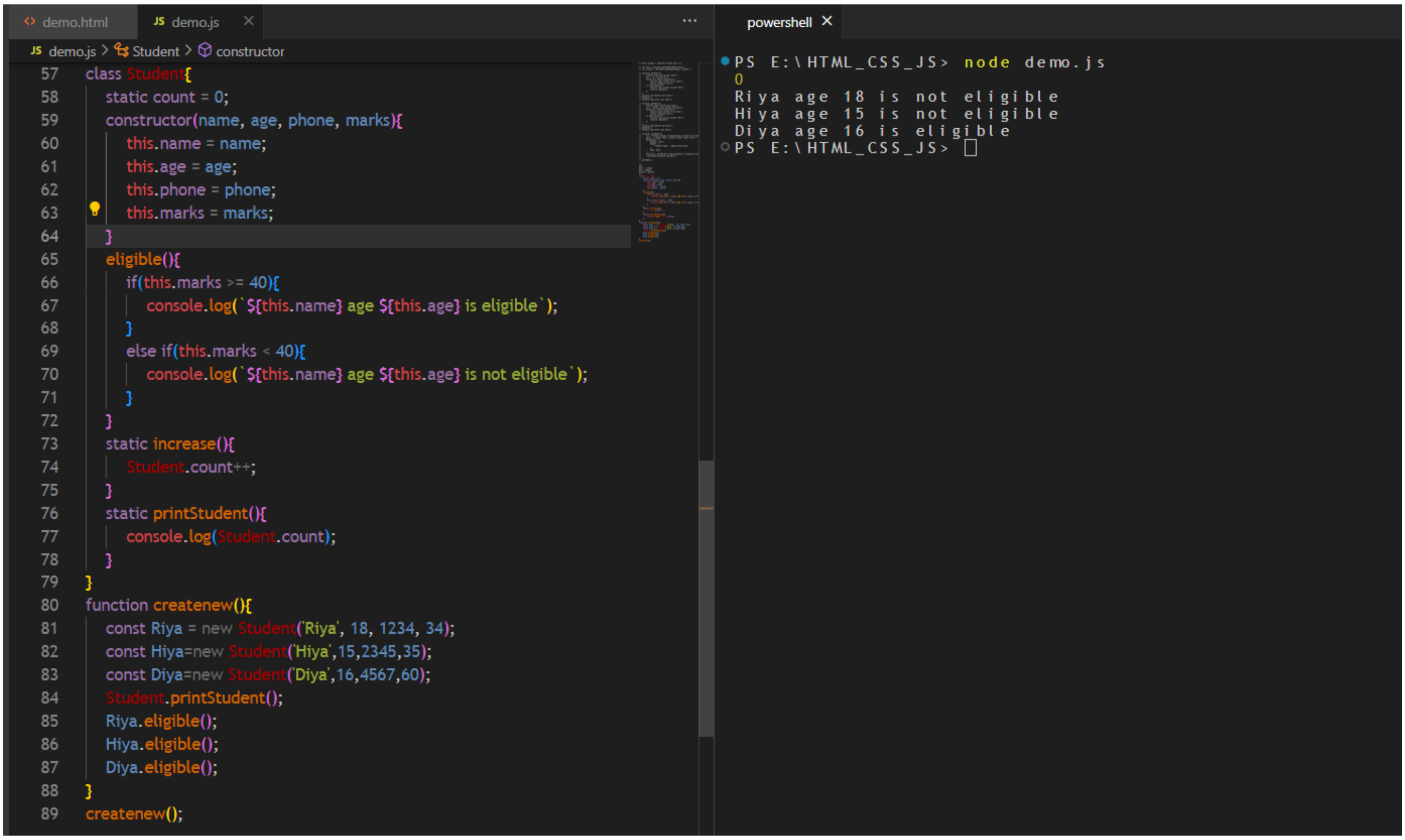


```
setTimeout(() => console.log('timer1 expired'), 1000)
setTimeout(() => console.log('timer2 expired'), 0)
function x(y) {
console.log('inside x');
y();
}
x(function y(){
setTimeout(() => console.log('inside y'), 0)
})
```

```
PS E:\HTML_CSS_JS> node demo.js
inside x
inside y
timer2 expired
timer1 expired
PS E:\HTML_CSS_JS>
```

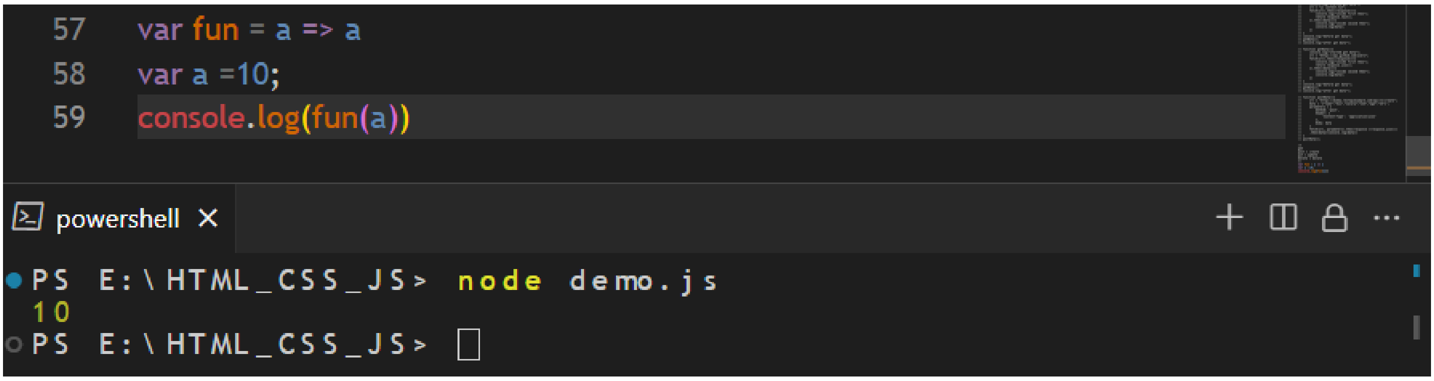
Question:

Design a student class which stores the name, age, phone number board marks of each student. Complete the constructor to initialize the values. Complete the function eligible to check whether the student is eligible or not for college. If board marks > 40 = eligible if less than 40 = not eligible.



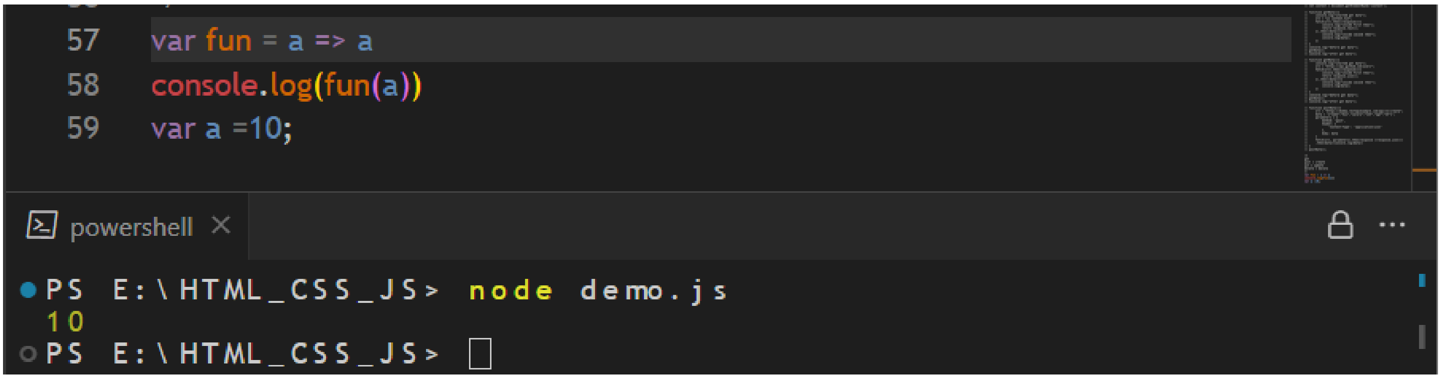
```
class Student{
    static count = 0;
    constructor(name, age, phone, marks){
        this.name = name;
        this.age = age;
        this.phone = phone;
        this.marks = marks;
    }
    eligible(){
        if(this.marks >= 40){
            console.log(`${this.name} age ${this.age} is eligible`);
        }
        else if(this.marks < 40){
            console.log(`${this.name} age ${this.age} is not eligible`);
        }
    }
    static increase(){
        Student.count++;
    }
    static printStudent(){
        console.log(Student.count);
    }
}
function createnew(){
    const Riya = new Student('Riya', 18, 1234, 34);
    const Hiya=new Student('Hiya',15,2345,35);
    const Diya=new Student('Diya',16,4567,60);
    Student.printStudent();
    Riya.eligible();
    Hiya.eligible();
    Diya.eligible();
}
createnew();
```

```
PS E:\HTML_CSS_JS> node demo.js
0
Riya age 18 is not eligible
Hiya age 15 is not eligible
Diya age 16 is eligible
PS E:\HTML_CSS_JS>
```



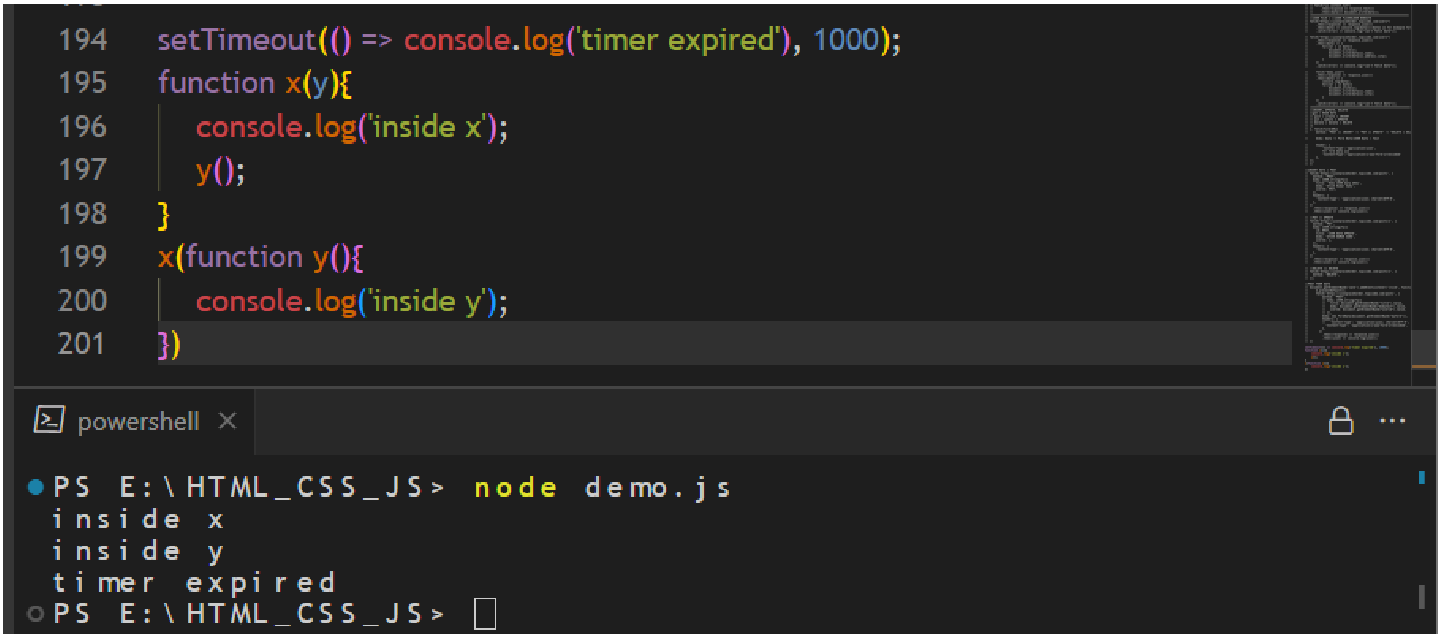
```
var fun = a => a
var a =10;
console.log(fun(a))
```

```
PS E:\HTML_CSS_JS> node demo.js
10
PS E:\HTML_CSS_JS>
```



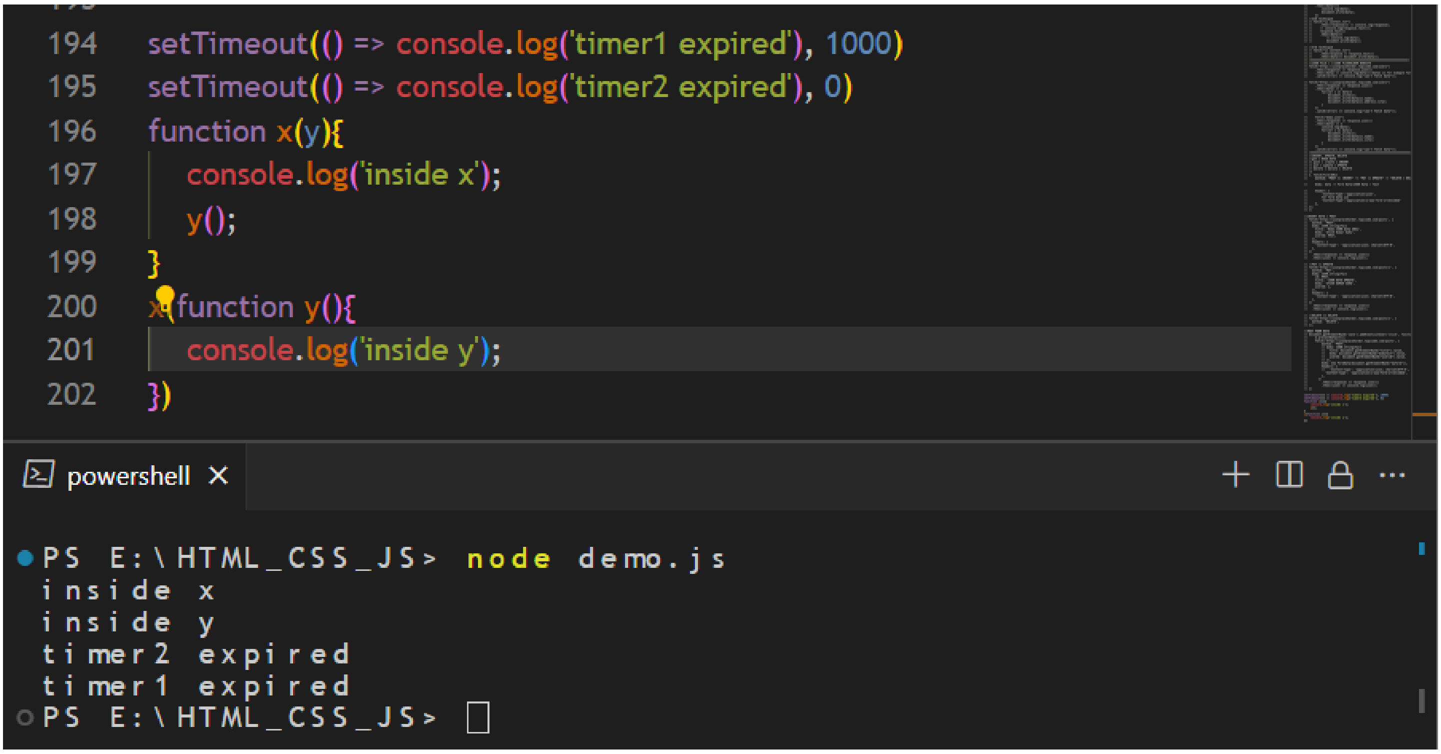
```
var fun = a => a
console.log(fun(a))
var a =10;
```

```
PS E:\HTML_CSS_JS> node demo.js
10
PS E:\HTML_CSS_JS>
```



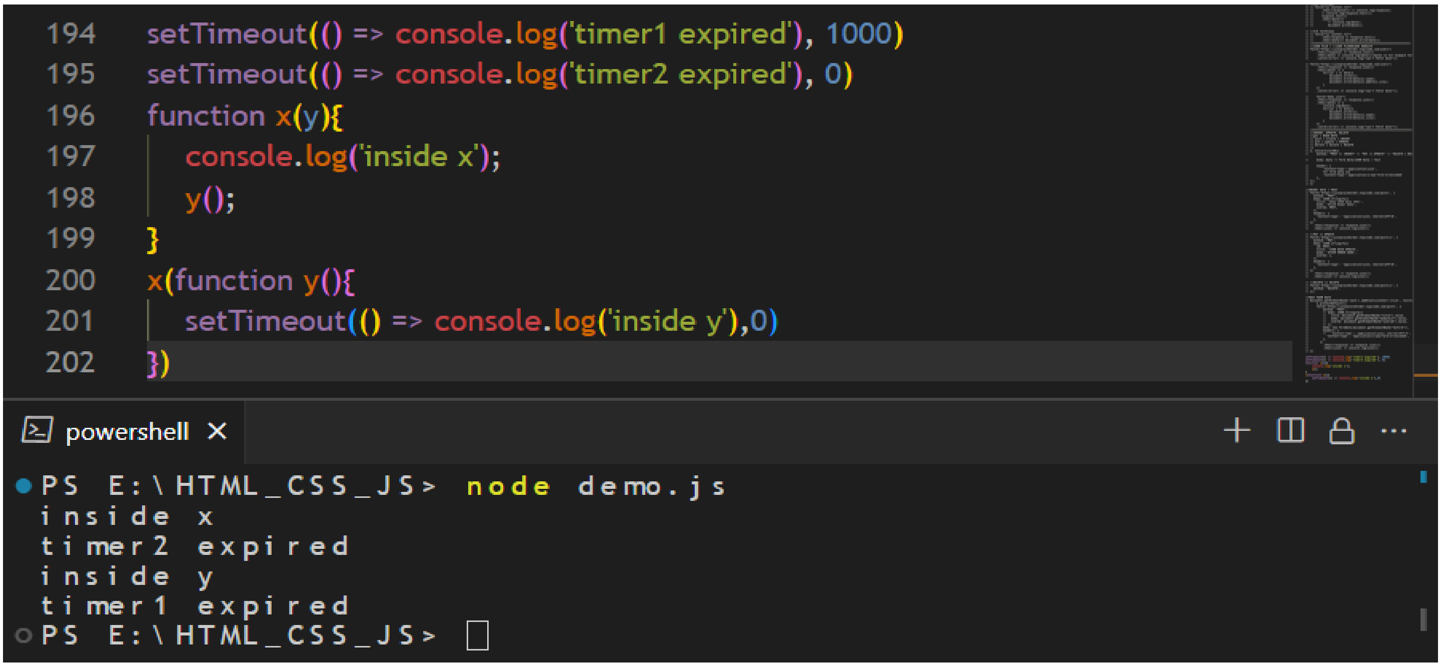
```javascript
setTimeout(() => console.log('timer expired'), 1000);
function x(y){
  console.log('inside x');
  y();
}
x(function y(){
  console.log('inside y');
})
```

```
PS E:\HTML_CSS_JS> node demo.js
inside x
inside y
timer expired
PS E:\HTML_CSS_JS>
```



```javascript
setTimeout(() => console.log('timer1 expired'), 1000)
setTimeout(() => console.log('timer2 expired'), 0)
function x(y){
  console.log('inside x');
  y();
}
x(function y(){
  console.log('inside y');
})
```

```
PS E:\HTML_CSS_JS> node demo.js
inside x
inside y
timer2 expired
timer1 expired
PS E:\HTML_CSS_JS>
```



```javascript
setTimeout(() => console.log('timer1 expired'), 1000)
setTimeout(() => console.log('timer2 expired'), 0)
function x(y){
  console.log('inside x');
  y();
}
x(function y(){
  setTimeout(() => console.log('inside y'),0)
})
```

```
PS E:\HTML_CSS_JS> node demo.js
inside x
timer2 expired
inside y
timer1 expired
PS E:\HTML_CSS_JS>
```

Add an event listener called DOMCONTENTLOADED inside this add a callback function which console “DOM has loaded”. More than one answer can be correct.

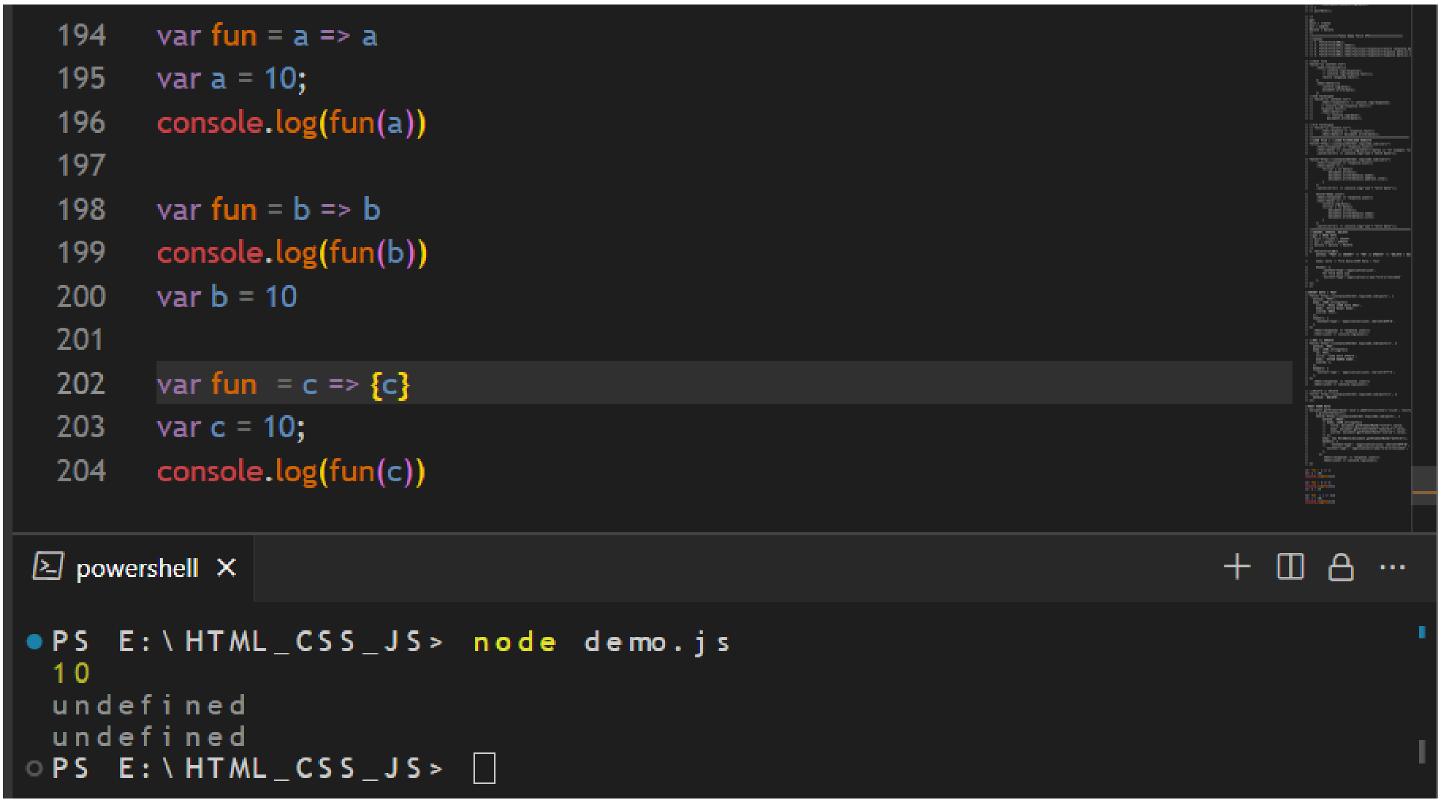
window.addEventListener(“DOMContentLoaded”,() => {console.log(‘DOM HAS LOADED’)})

document.addEventListener(“DOMContentLoaded”, () => {console.log(‘DOM HAS LOADED’)})

JavaScript Garbage:

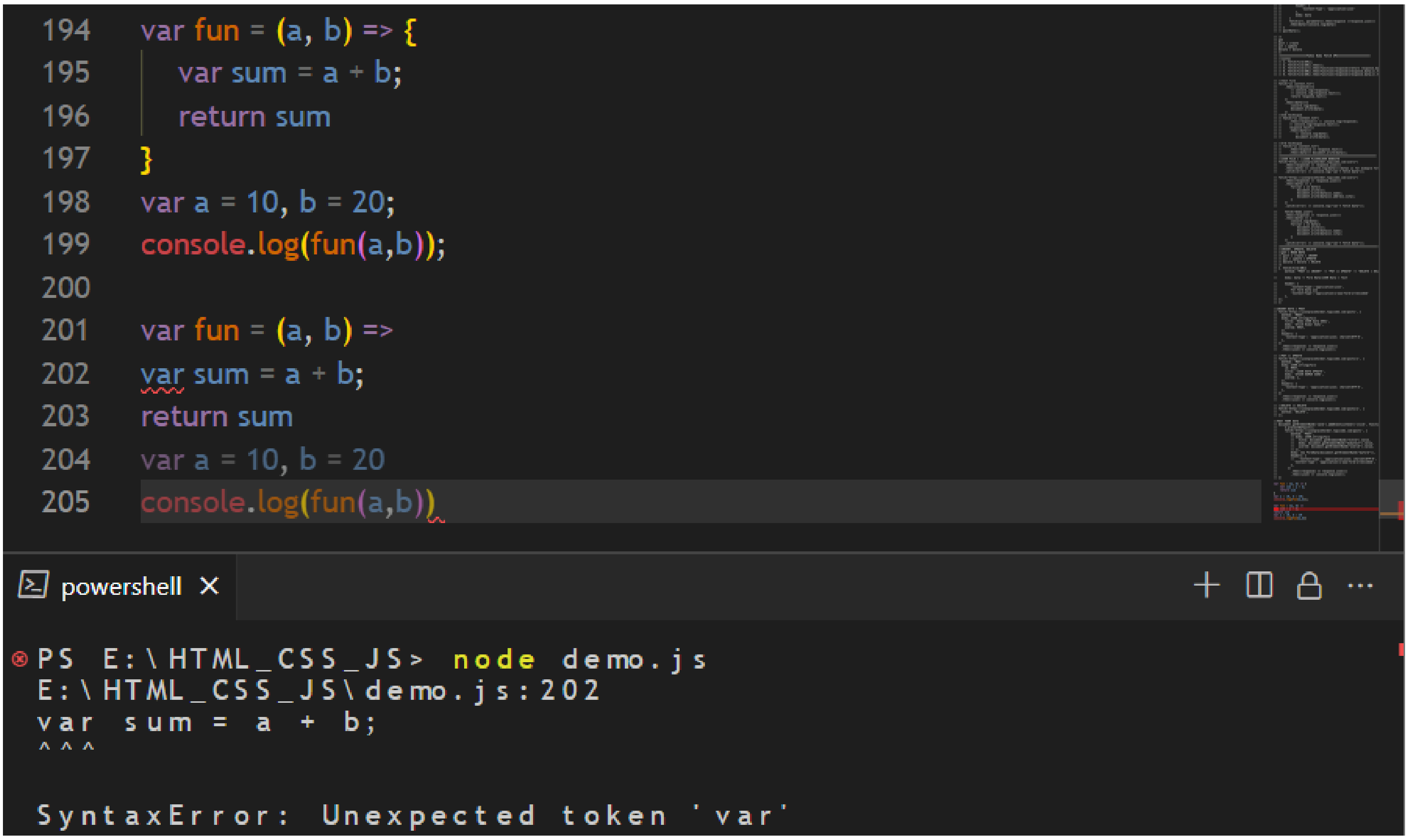
Garbage collection is a crucial process in computer programming, specifically in memory management. It's like the automated cleanup crew of a program that's responsible for identifying and reclaiming memory that's no longer in use or referenced by the program. Imagine you have a room, and you keep adding items to it. Over time, some of these items become unnecessary or forgotten, and your room gets cluttered.

Garbage collection is like a system that periodically checks the room, identifies the items that are no longer needed, and clears them out, making room for new items and ensuring efficient use of available space. In programming, it helps prevent memory leaks and ensures that the program doesn't use more memory than it actually needs, improving overall performance and stability. We need garbage collection because, without garbage collection we will run out of memory eventually. To release memory from variable which are out of scope now. To avoid memory leakage.



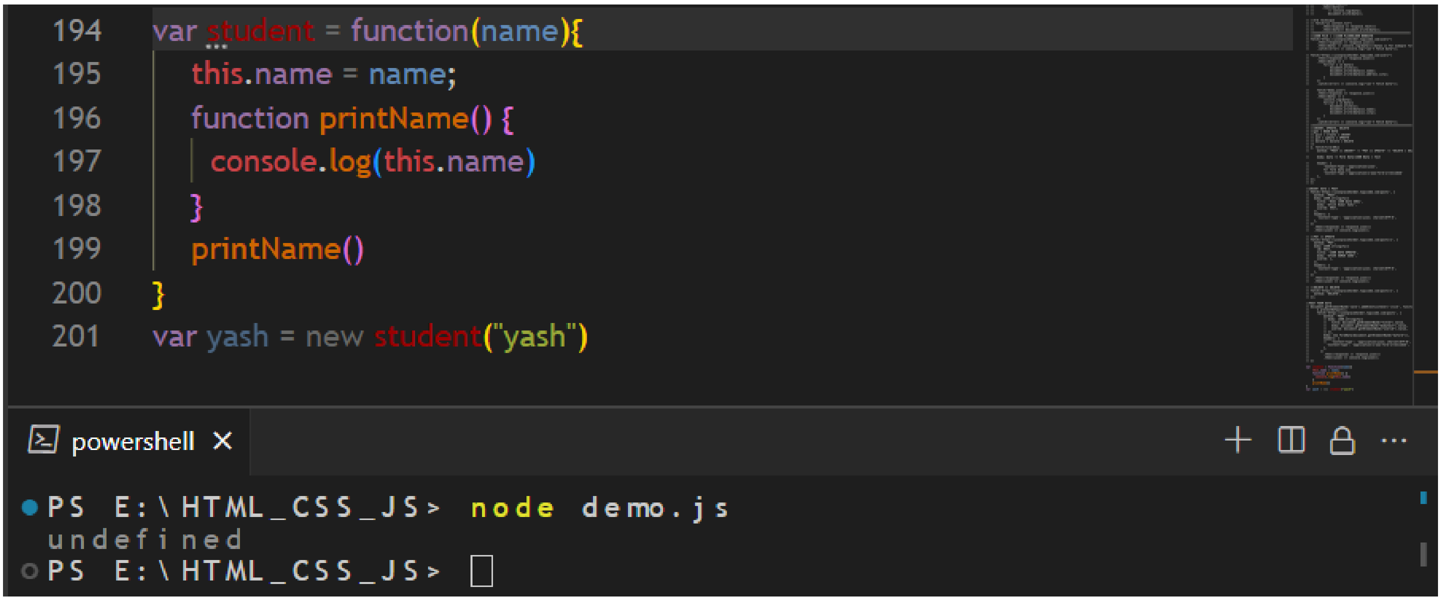
```
var fun = a => a
var a = 10;
console.log(fun(a))

var fun = b => b
console.log(fun(b))
var b = 10

var fun  = c => {c}
var c = 10;
console.log(fun(c))
```

```
PS E:\HTML_CSS_JS> node demo.js
10
undefined
undefined
PS E:\HTML_CSS_JS>
```



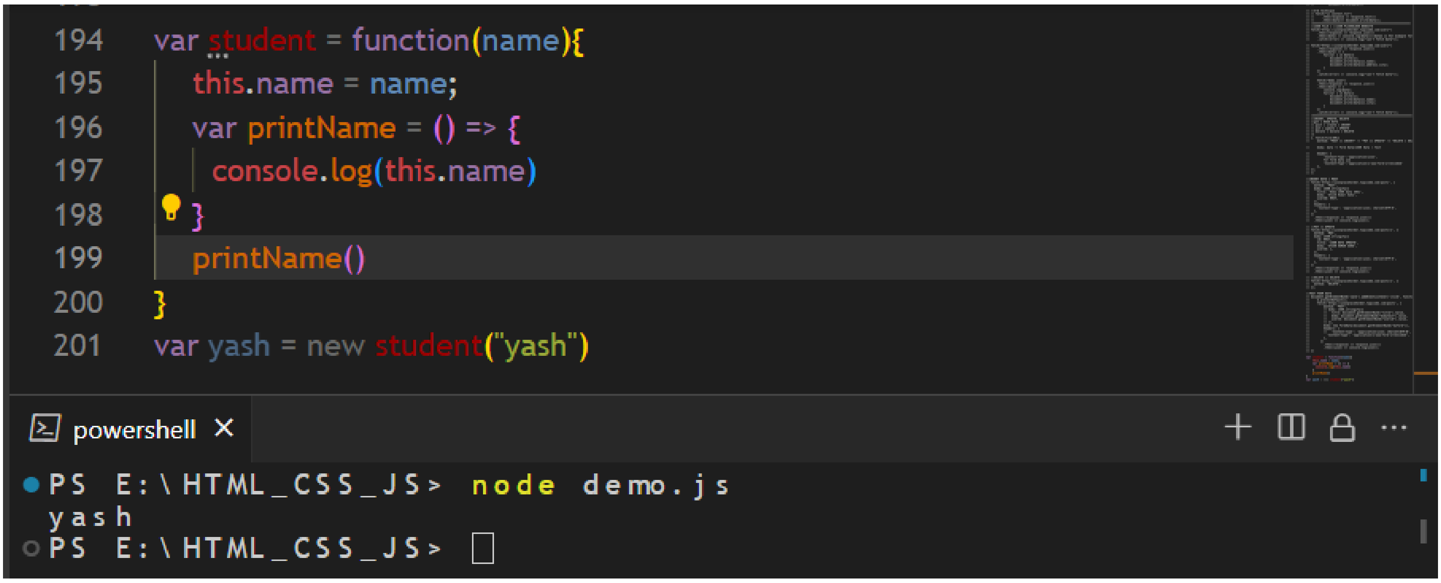
```
var fun = (a, b) => {
    var sum = a + b;
    return sum
}
var a = 10, b = 20;
console.log(fun(a,b));

var fun = (a, b) =>
var sum = a + b;
return sum
var a = 10, b = 20
console.log(fun(a,b))
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:202
var sum = a + b;
^^^

SyntaxError: Unexpected token 'var'
```



```
var student = function(name){
    this.name = name;
    function printName() {
        console.log(this.name)
    }
    printName()
}
var yash = new student("yash")
```

```
PS E:\HTML_CSS_JS> node demo.js
undefined
PS E:\HTML_CSS_JS> 
```

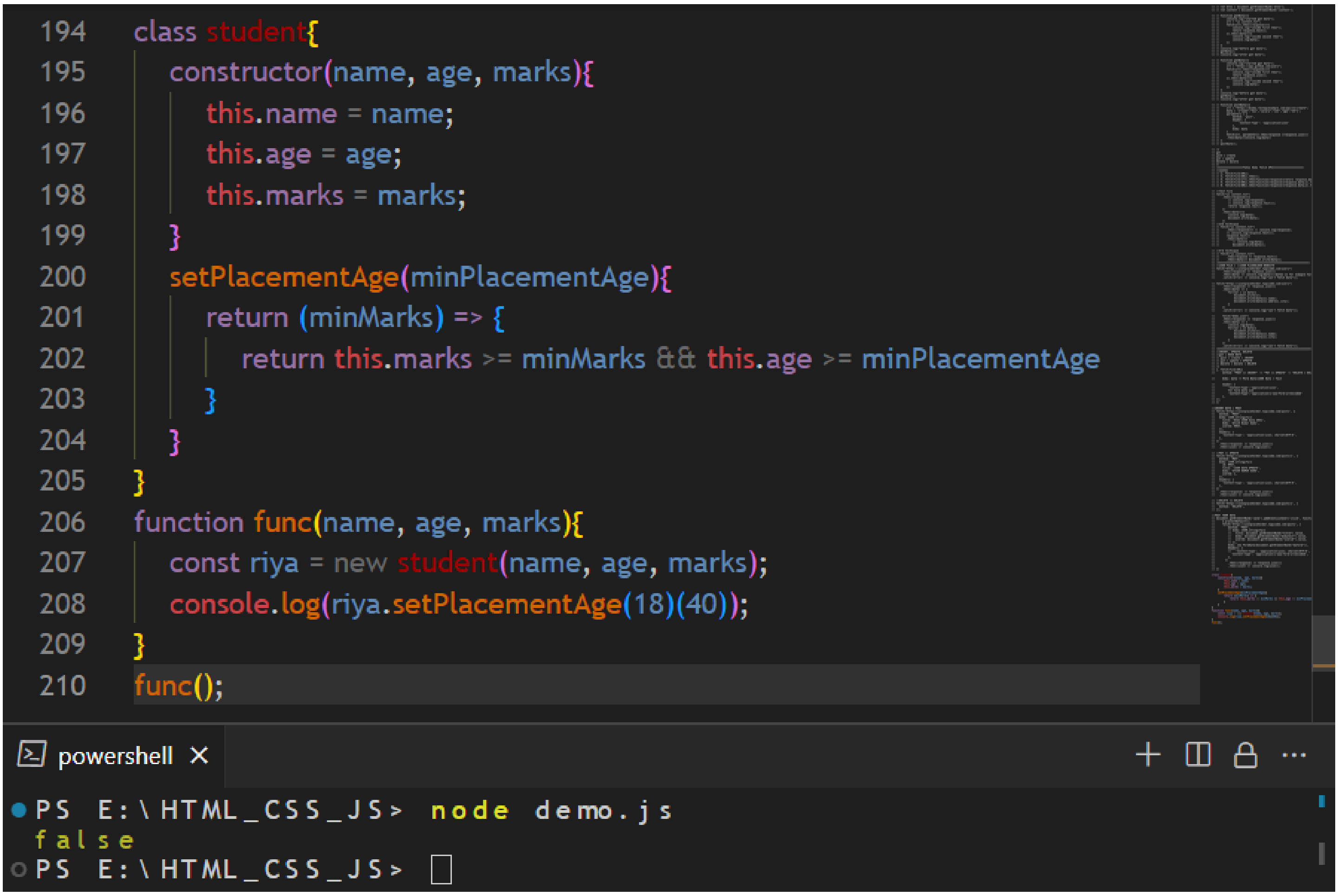


```
var student = function(name){
    this.name = name;
    var printName = () => {
        console.log(this.name)
    }
    printName()
}
var yash = new student("yash")
```

```
PS E:\HTML_CSS_JS> node demo.js
yash
PS E:\HTML_CSS_JS> 
```

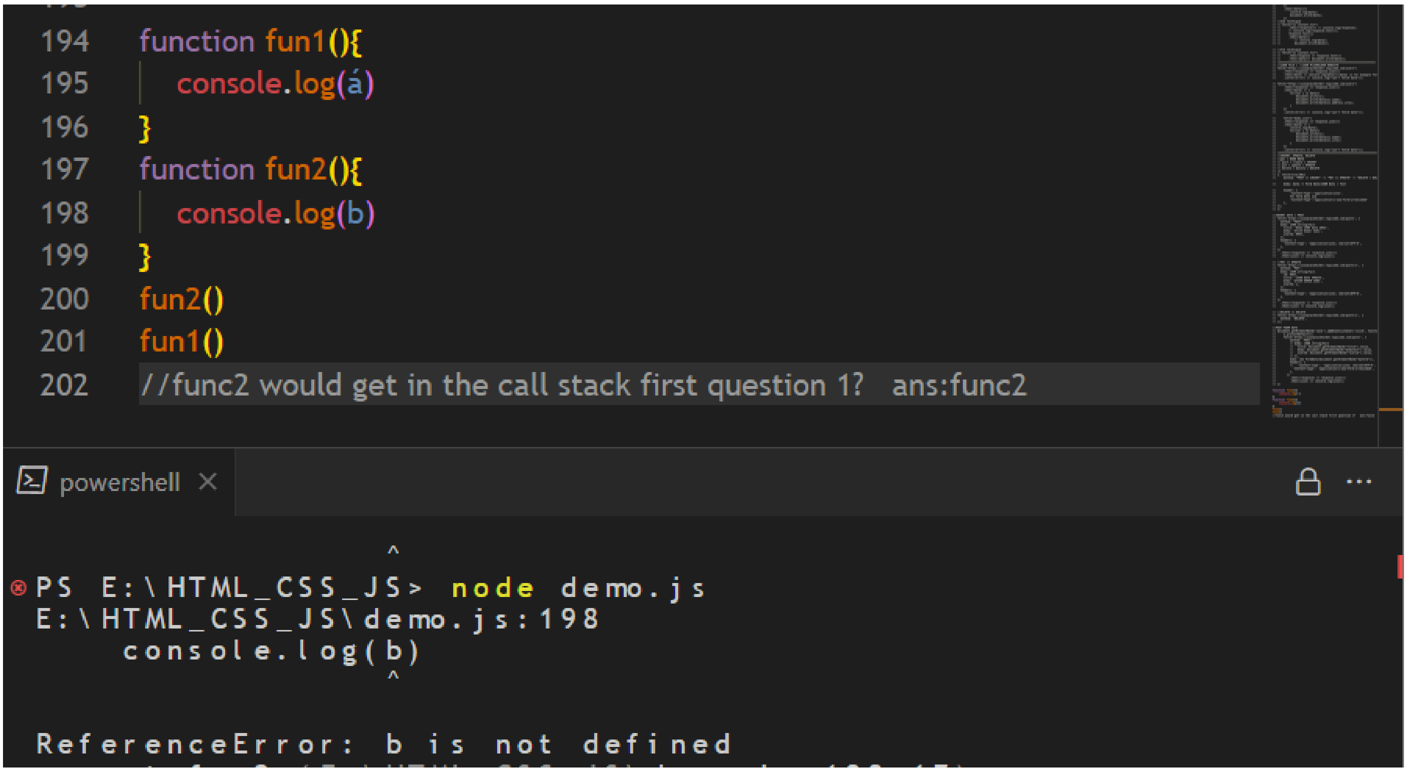
Question:

Take the student class and write a function called setPlacementAge which basically takes the minimum board marks required by a candidate to sit for the company coming for placement and it returns a function. The function takes minAge as an argument and returns true if the given student has board marks greater than minimum board marks required by company and is above the required age set by the company.



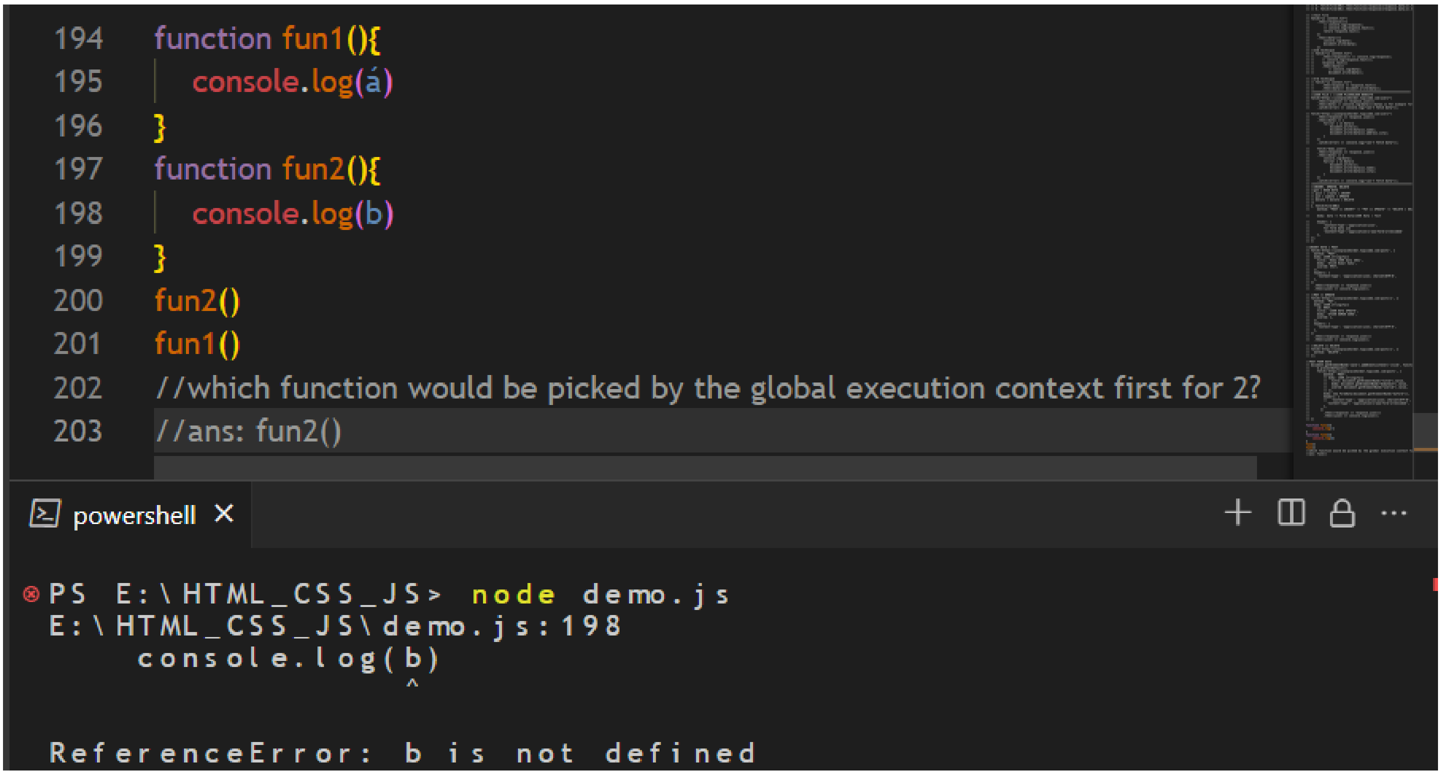
```
class student{
    constructor(name, age, marks){
        this.name = name;
        this.age = age;
        this.marks = marks;
    }
    setPlacementAge(minPlacementAge){
        return (minMarks) => {
            return this.marks >= minMarks && this.age >= minPlacementAge
        }
    }
}
function func(name, age, marks){
    const riya = new student(name, age, marks);
    console.log(riya.setPlacementAge(18)(40));
}
func();
```

```
PS E:\HTML_CSS_JS> node demo.js
false
PS E:\HTML_CSS_JS> 
```



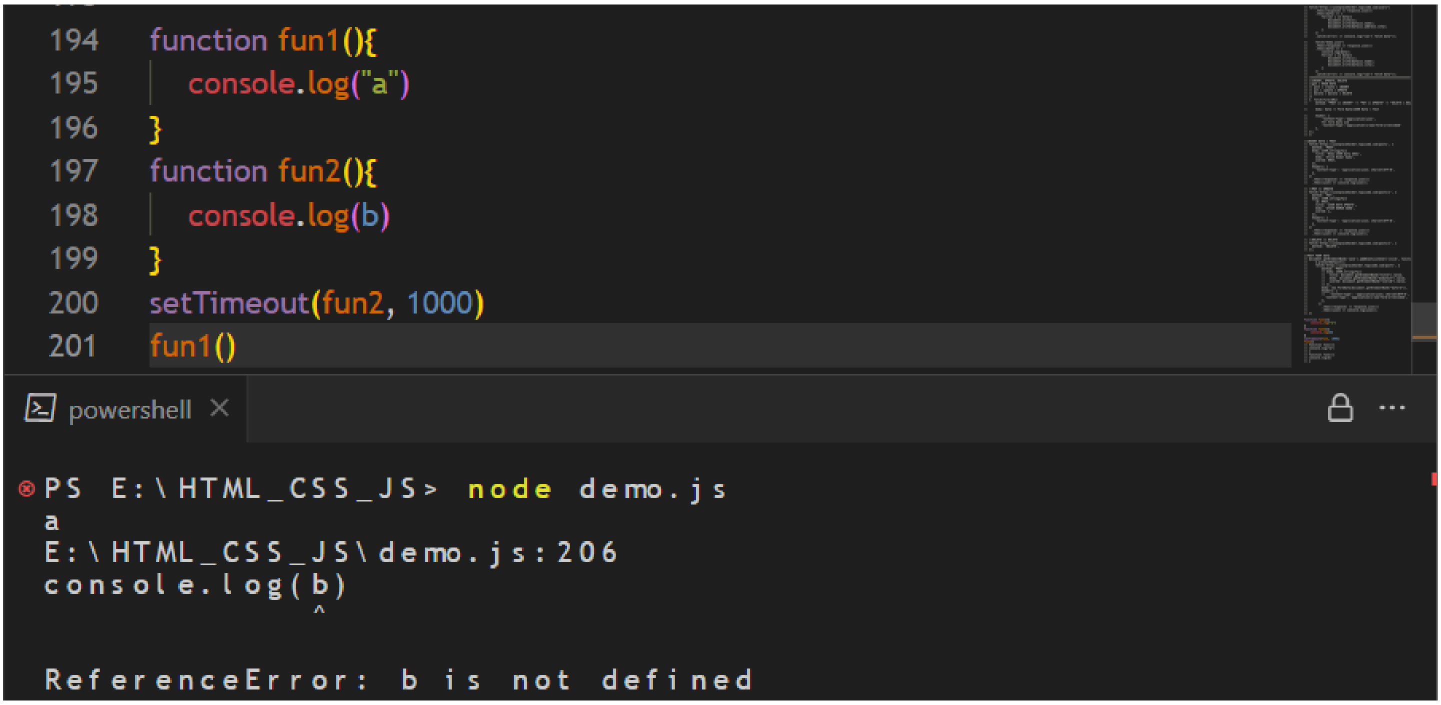
```
function fun1(){
    console.log(á)
}
function fun2(){
    console.log(b)
}
fun2()
fun1()
//func2 would get in the call stack first question 1?   ans:func2
```

```
                    ^
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:198
    console.log(b)
                ^

ReferenceError: b is not defined
```



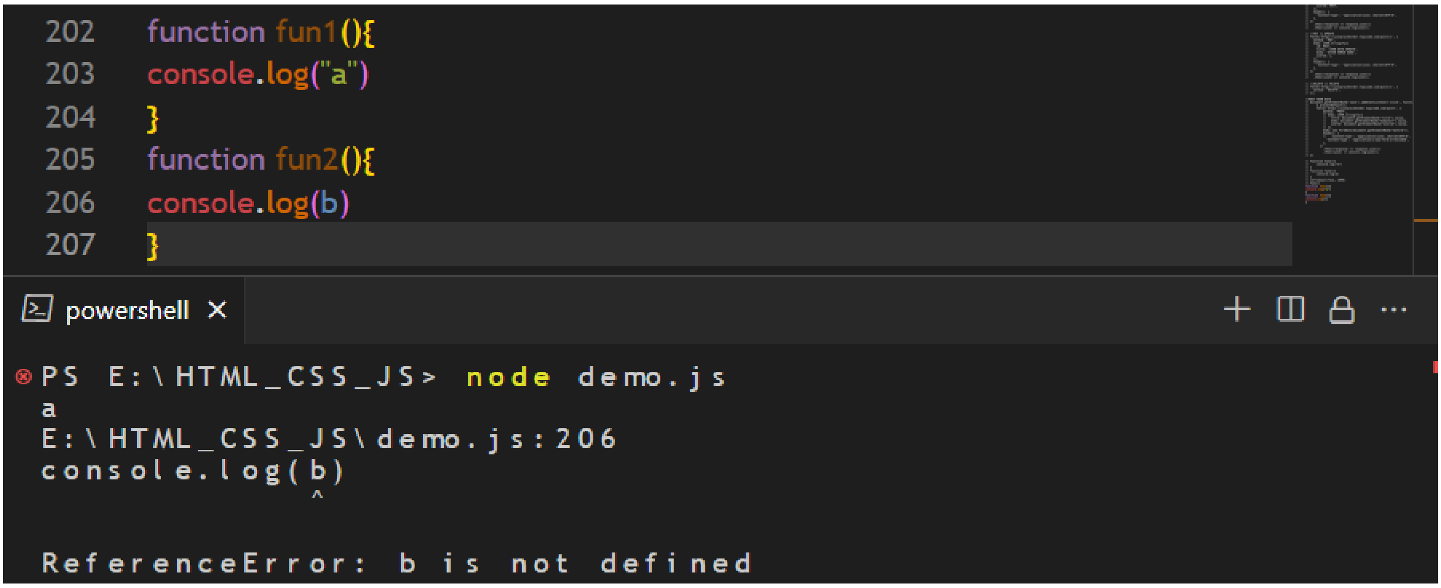
```
function fun1(){
    console.log(á)
}
function fun2(){
    console.log(b)
}
fun2()
fun1()
//which function would be picked by the global execution context first for 2?
//ans: fun2()
```

```
PS E:\HTML_CSS_JS> node demo.js
E:\HTML_CSS_JS\demo.js:198
    console.log(b)
                ^

ReferenceError: b is not defined
```



```
function fun1(){
    console.log("a")
}
function fun2(){
    console.log(b)
}
setTimeout(fun2, 1000)
fun1()
```

```
PS E:\HTML_CSS_JS> node demo.js
a
E:\HTML_CSS_JS\demo.js:206
console.log(b)
            ^

ReferenceError: b is not defined
```



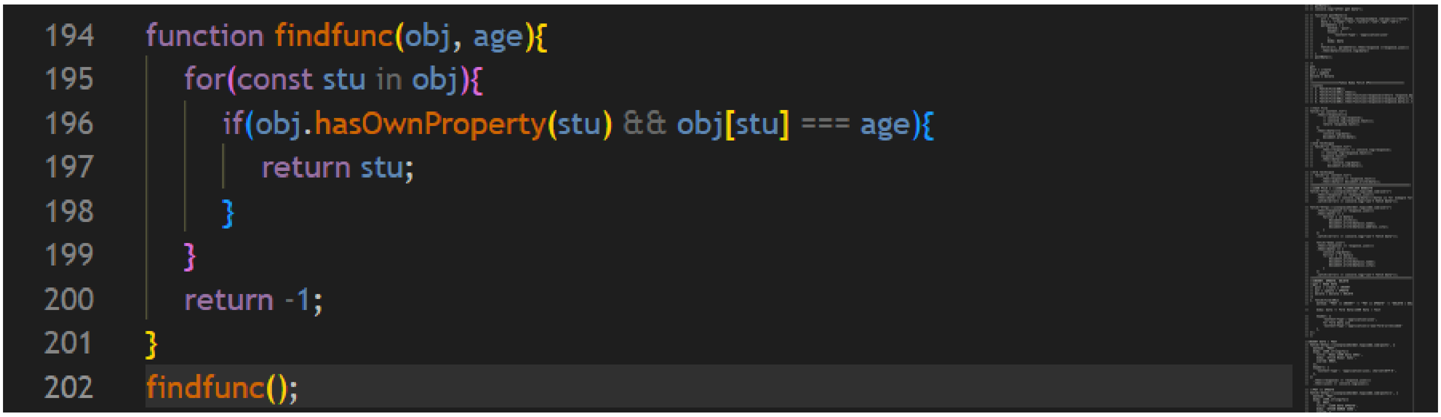
```
function fun1(){
console.log("a")
}
function fun2(){
console.log(b)
}
```

```
PS E:\HTML_CSS_JS> node demo.js
a
E:\HTML_CSS_JS\demo.js:206
console.log(b)
            ^

ReferenceError: b is not defined
```

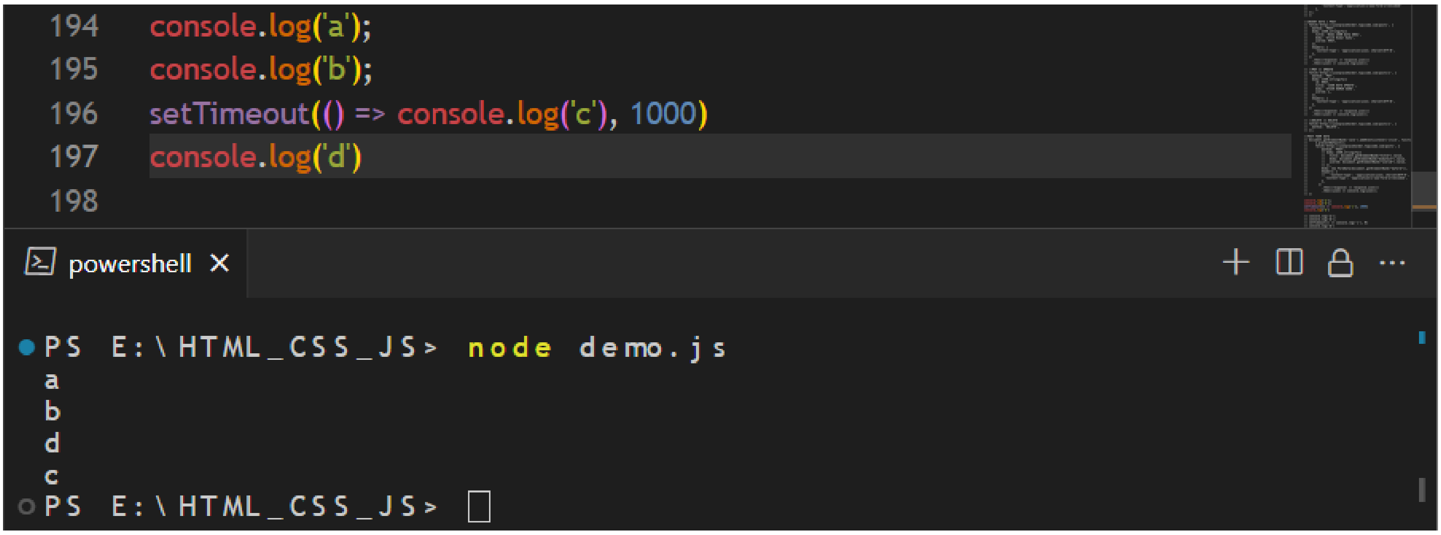
Question:

Given a student object as shown below can you write code to iterate through object and find the student's name who is of age n.

Input: stu = {‘yash’: 26, ‘vaibhav’: 24, ‘rajesh’: 25}   age = 25   Output: rajesh

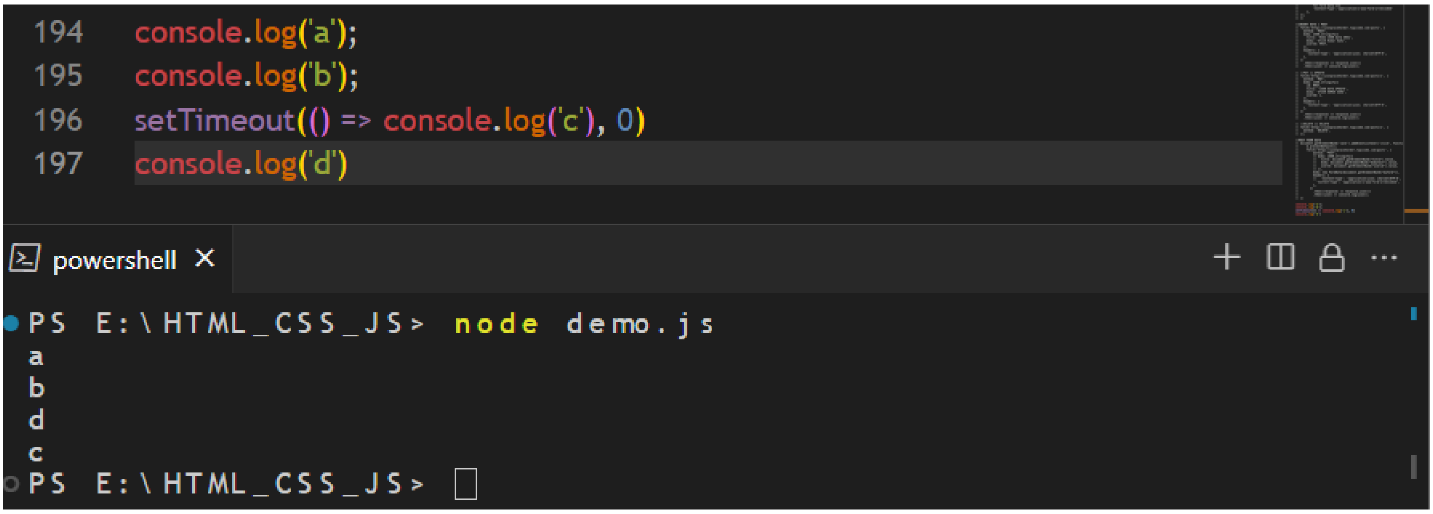


```
function findfunc(obj, age){
    for(const stu in obj){
        if(obj.hasOwnProperty(stu) && obj[stu] === age){
            return stu;
        }
    }
    return -1;
}
findfunc();
```



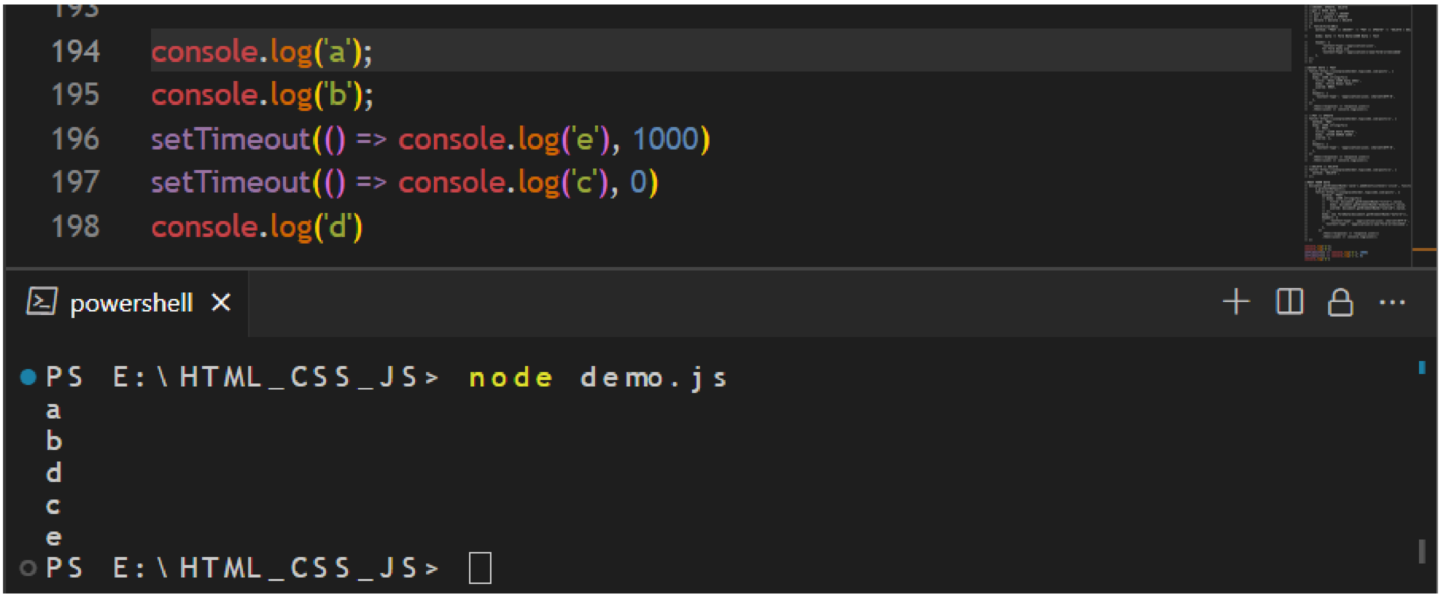
```
console.log('a');
console.log('b');
setTimeout(() => console.log('c'), 1000)
console.log('d')

```

```
PS E:\HTML_CSS_JS> node demo.js
a
b
d
c
PS E:\HTML_CSS_JS>
```



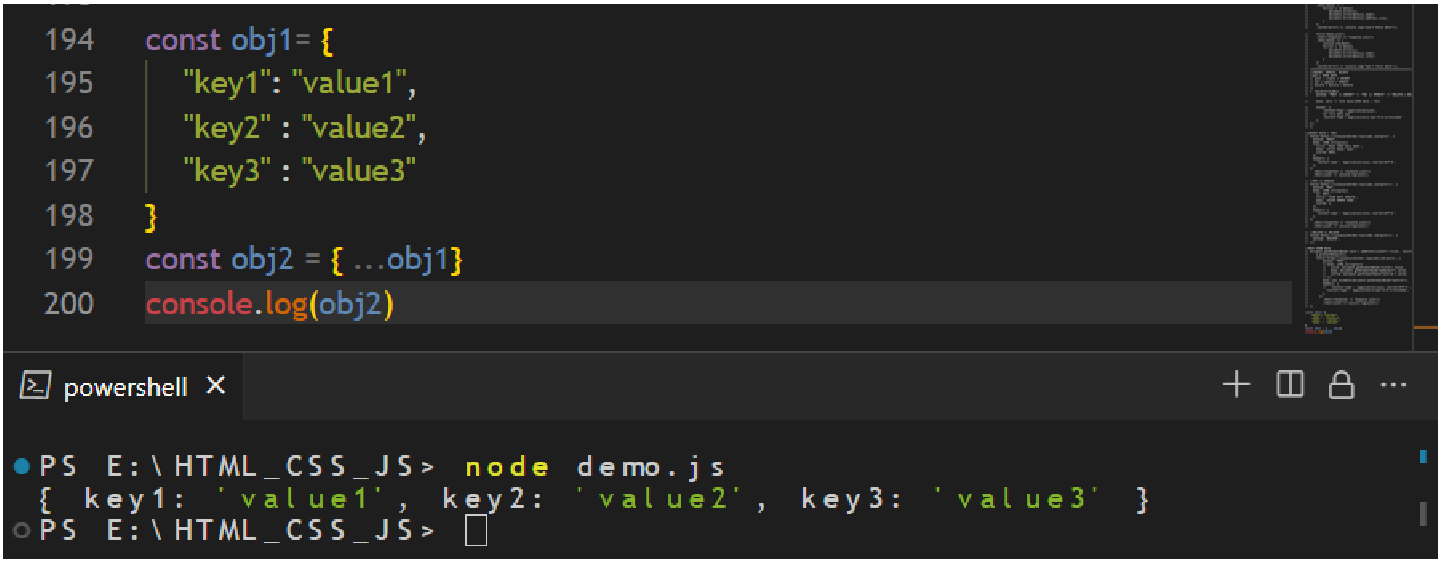
```
console.log('a');
console.log('b');
setTimeout(() => console.log('c'), 0)
console.log('d')
```

```
PS E:\HTML_CSS_JS> node demo.js
a
b
d
c
PS E:\HTML_CSS_JS>
```



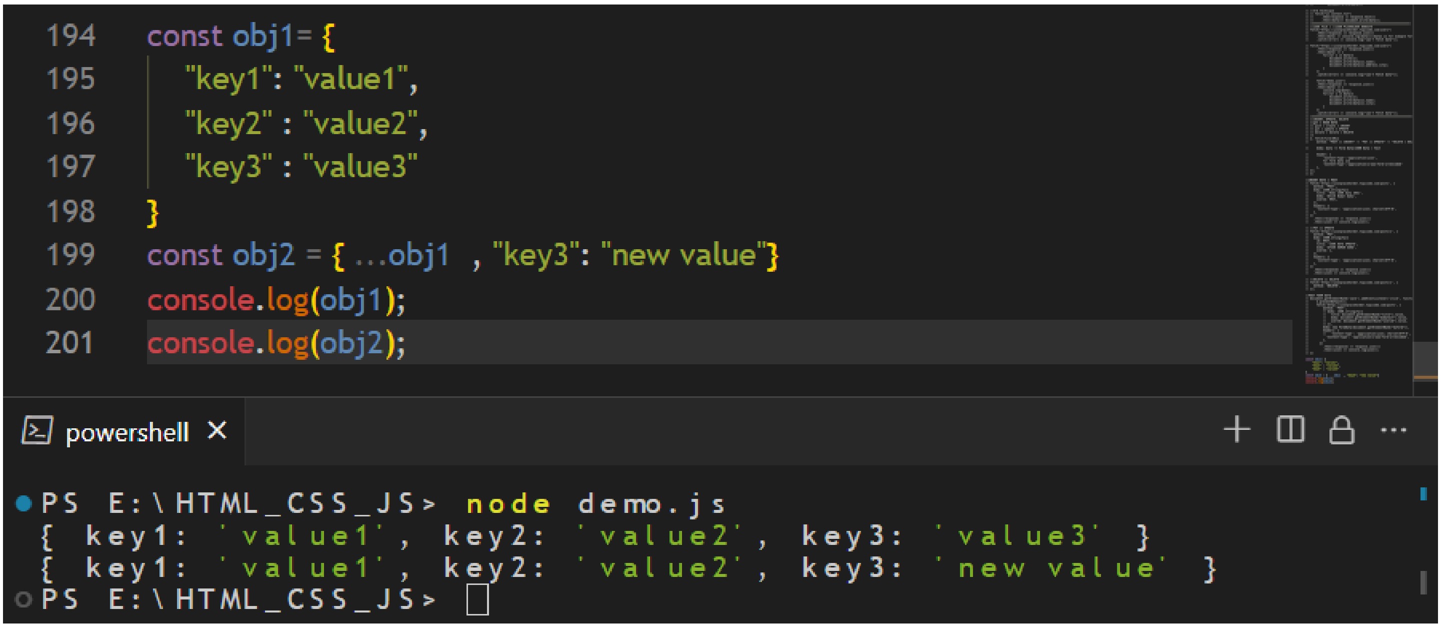
```javascript
console.log('a');
console.log('b');
setTimeout(() => console.log('e'), 1000)
setTimeout(() => console.log('c'), 0)
console.log('d')
```

```
PS E:\HTML_CSS_JS> node demo.js
a
b
d
c
e
PS E:\HTML_CSS_JS>
```



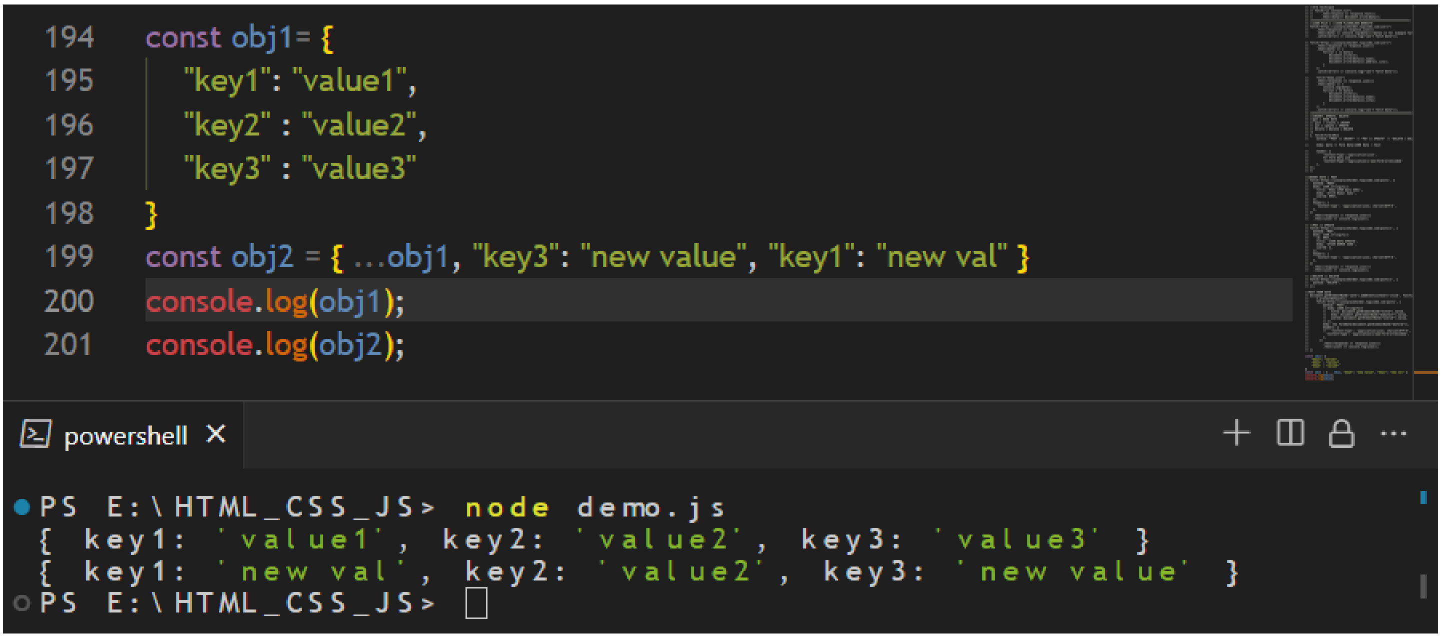
```javascript
const obj1= {
    "key1": "value1",
    "key2" : "value2",
    "key3" : "value3"
}
const obj2 = { ...obj1}
console.log(obj2)
```

```
PS E:\HTML_CSS_JS> node demo.js
{ key1: 'value1', key2: 'value2', key3: 'value3' }
PS E:\HTML_CSS_JS>
```



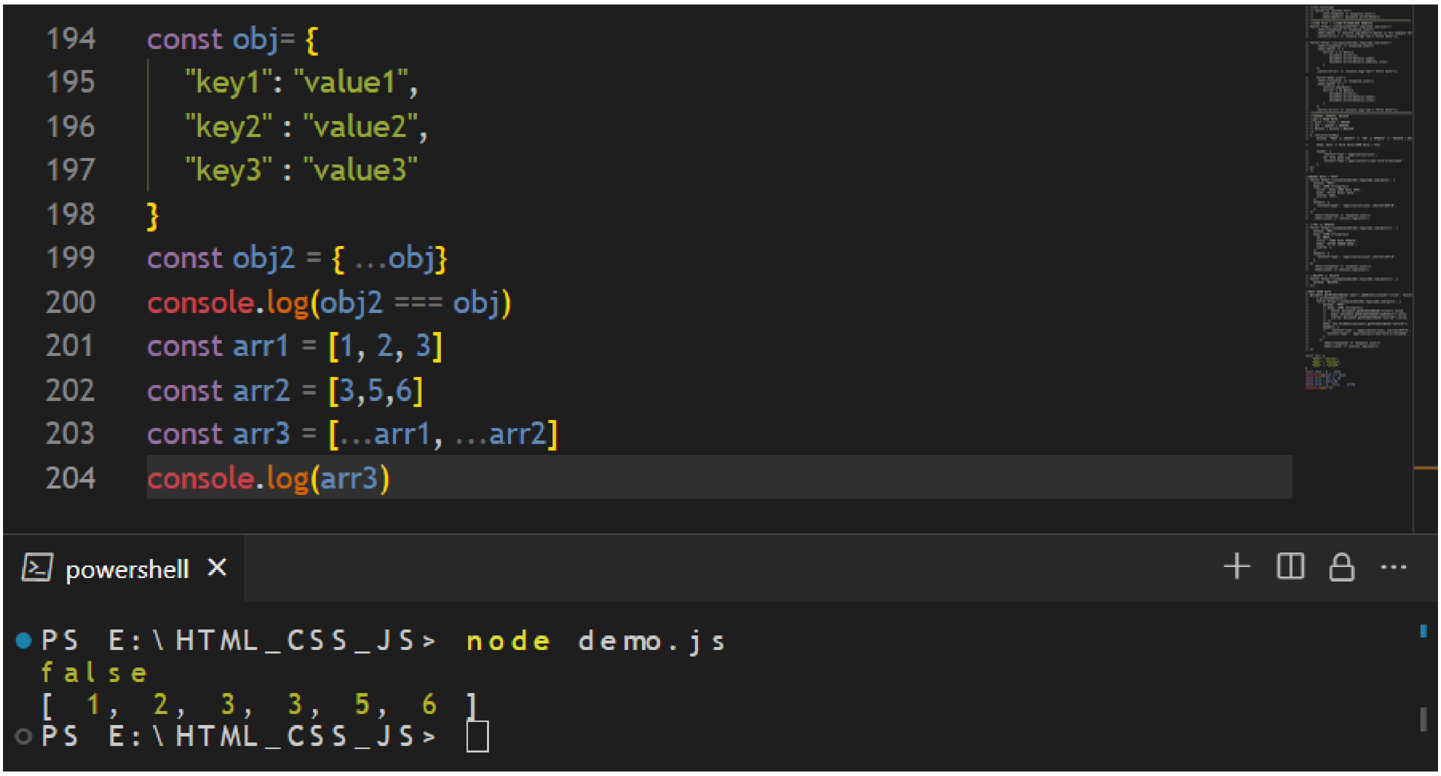
```javascript
const obj1= {
    "key1": "value1",
    "key2" : "value2",
    "key3" : "value3"
}
const obj2 = { ...obj1 , "key3": "new value"}
console.log(obj1);
console.log(obj2);
```

```
PS E:\HTML_CSS_JS> node demo.js
{ key1: 'value1', key2: 'value2', key3: 'value3' }
{ key1: 'value1', key2: 'value2', key3: 'new value' }
PS E:\HTML_CSS_JS>
```



```
const obj1= {
    "key1": "value1",
    "key2" : "value2",
    "key3" : "value3"
}
const obj2 = { ...obj1, "key3": "new value", "key1": "new val" }
console.log(obj1);
console.log(obj2);
```

```
PS E:\HTML_CSS_JS> node demo.js
{ key1: 'value1', key2: 'value2', key3: 'value3' }
{ key1: 'new val', key2: 'value2', key3: 'new value' }
PS E:\HTML_CSS_JS> 
```



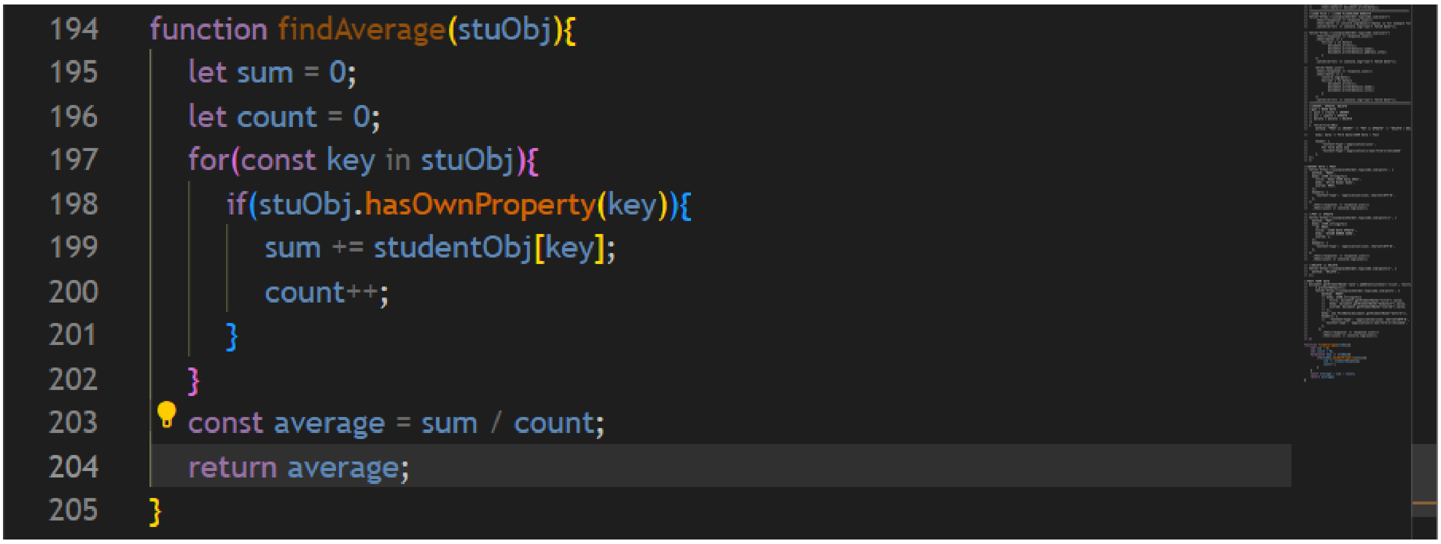
```
const obj= {
    "key1": "value1",
    "key2" : "value2",
    "key3" : "value3"
}
const obj2 = { ...obj}
console.log(obj2 === obj)
const arr1 = [1, 2, 3]
const arr2 = [3,5,6]
const arr3 = [...arr1, ...arr2]
console.log(arr3)
```

```
PS E:\HTML_CSS_JS> node demo.js
false
[ 1, 2, 3, 3, 5, 6 ]
PS E:\HTML_CSS_JS> 
```

Question:

Given a student object as shown below can you write code to iterate through object and find average age of students.

Input: studentobj = {'yash': 26, 'vaibhav': 24, 'rajesh' : 25}   Output: Average age = 25



```
function findAverage(stuObj){
    let sum = 0;
    let count = 0;
    for(const key in stuObj){
        if(stuObj.hasOwnProperty(key)){
            sum += studentObj[key];
            count++;
        }
    }
    const average = sum / count;
    return average;
}
```