JavaScript – W2S2
More on functions

Cyrille Jegourel – Singapore University of Technology and Design

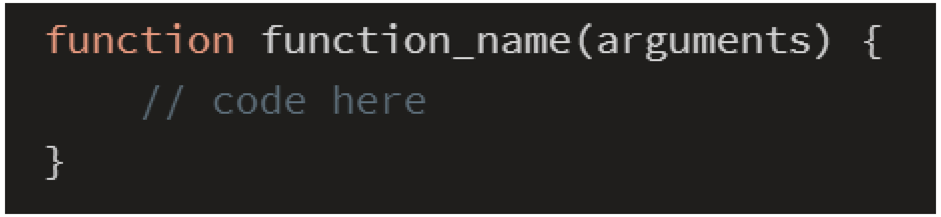
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Outline (Week 2, Session 2)

- Other ways to declare functions
- Recursion
- Arguments object and the rest argument
- Closures
- Callbacks

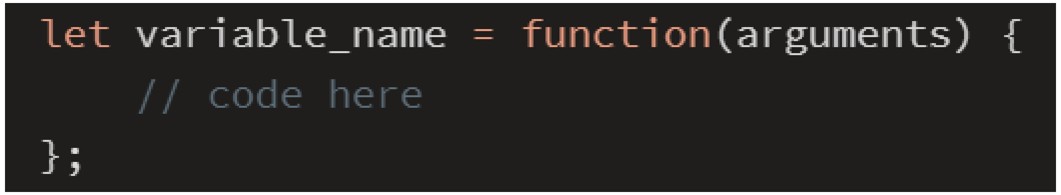
Several ways to declare functions

- Function declaration
    - Method we have used so far



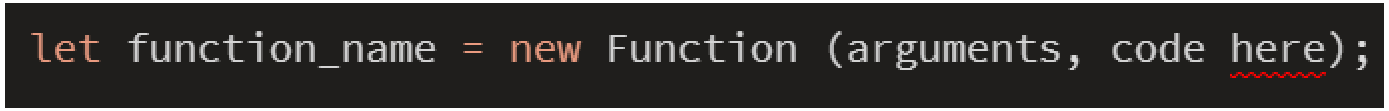
```
function function_name(arguments) {
    // code here
}
```

- Function expression



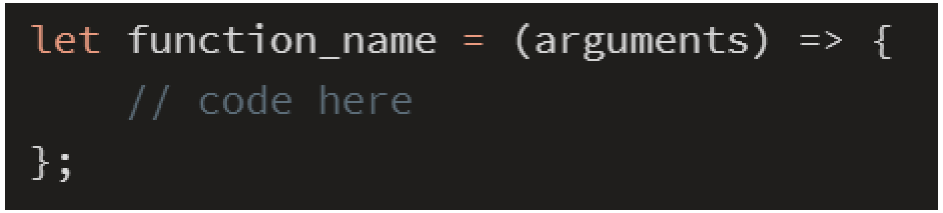
```
let variable_name = function(arguments) {
    // code here
};
```

- Function constructor



```
let function_name = new Function (arguments, code here);
```

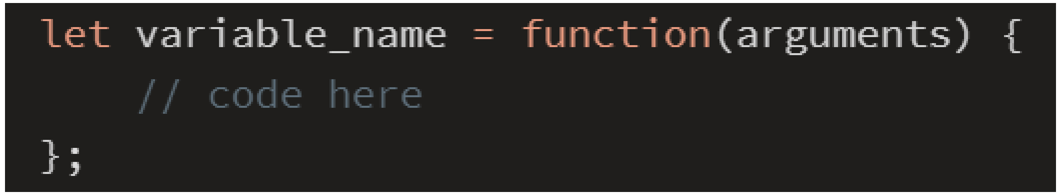
- Arrow functions



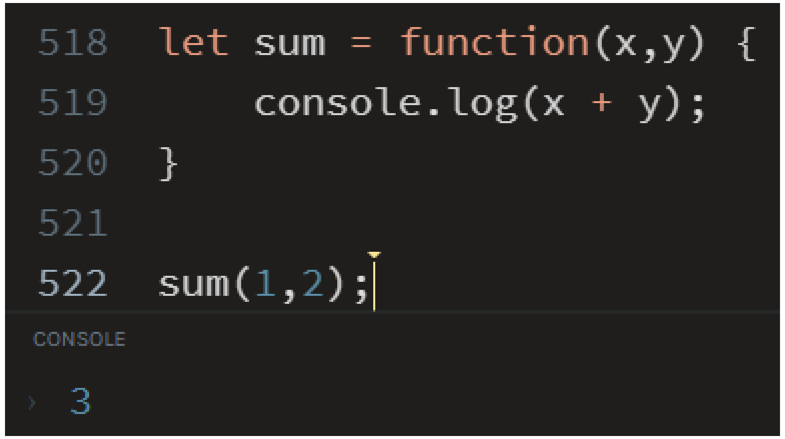
```
let function_name = (arguments) => {
    // code here
};
```

Function expression

- A function expression assigns a function to a variable name.
- Note the semi-colon at the end of statement.
- There is no name after the function keyword. This function is said to be anonymous and can be called using the variable name.



```
let variable_name = function(arguments) {
    // code here
};
```



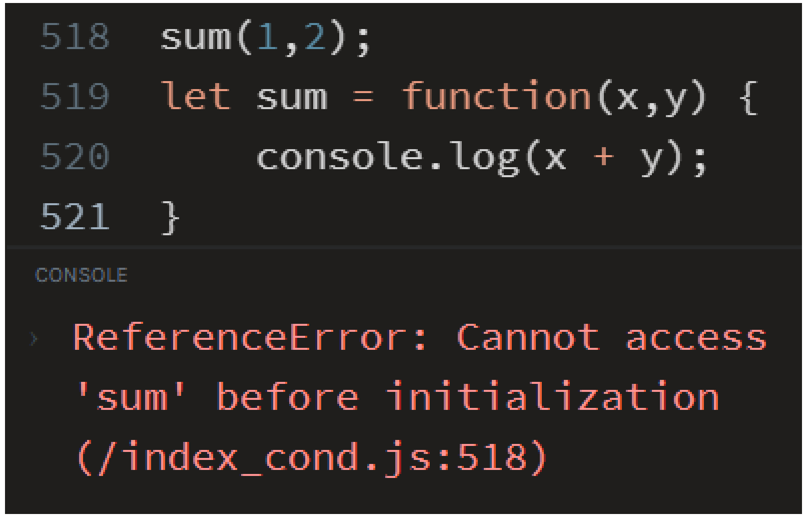
```
let sum = function(x,y) {
    console.log(x + y);
}

sum(1,2);
```

CONSOLE

```
› 3
```

Function expression

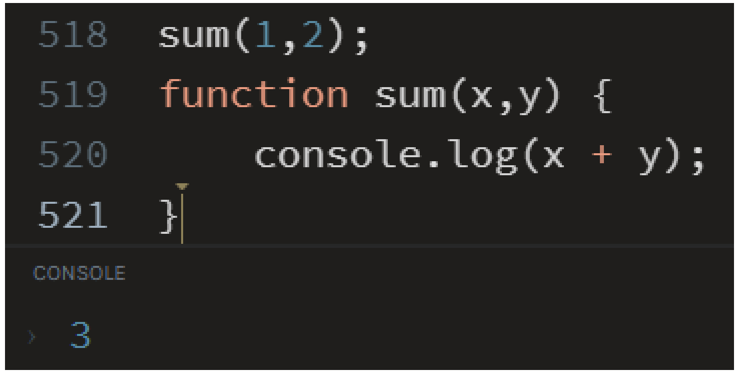
- A function expression assigns a function to a variable name.
- Note the semi-colon at the end of statement.
- There is no name after the function keyword. This function is said to be anonymous and can be called using the variable name.
- A function expression **does not make use of declaration hoisting**. It cannot be called before being defined.



```
518  sum(1,2);
519  let sum = function(x,y) {
520      console.log(x + y);
521  }
```

CONSOLE

```
› ReferenceError: Cannot access
  'sum' before initialization
  (/index_cond.js:518)
```



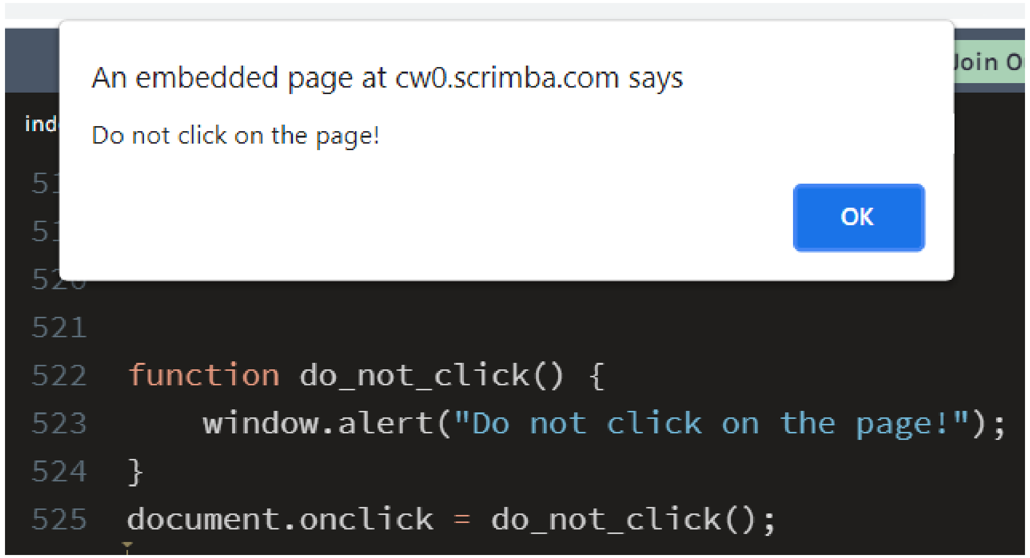
```
518  sum(1,2);
519  function sum(x,y) {
520      console.log(x + y);
521  }
```

CONSOLE

```
› 3
```

Function expression

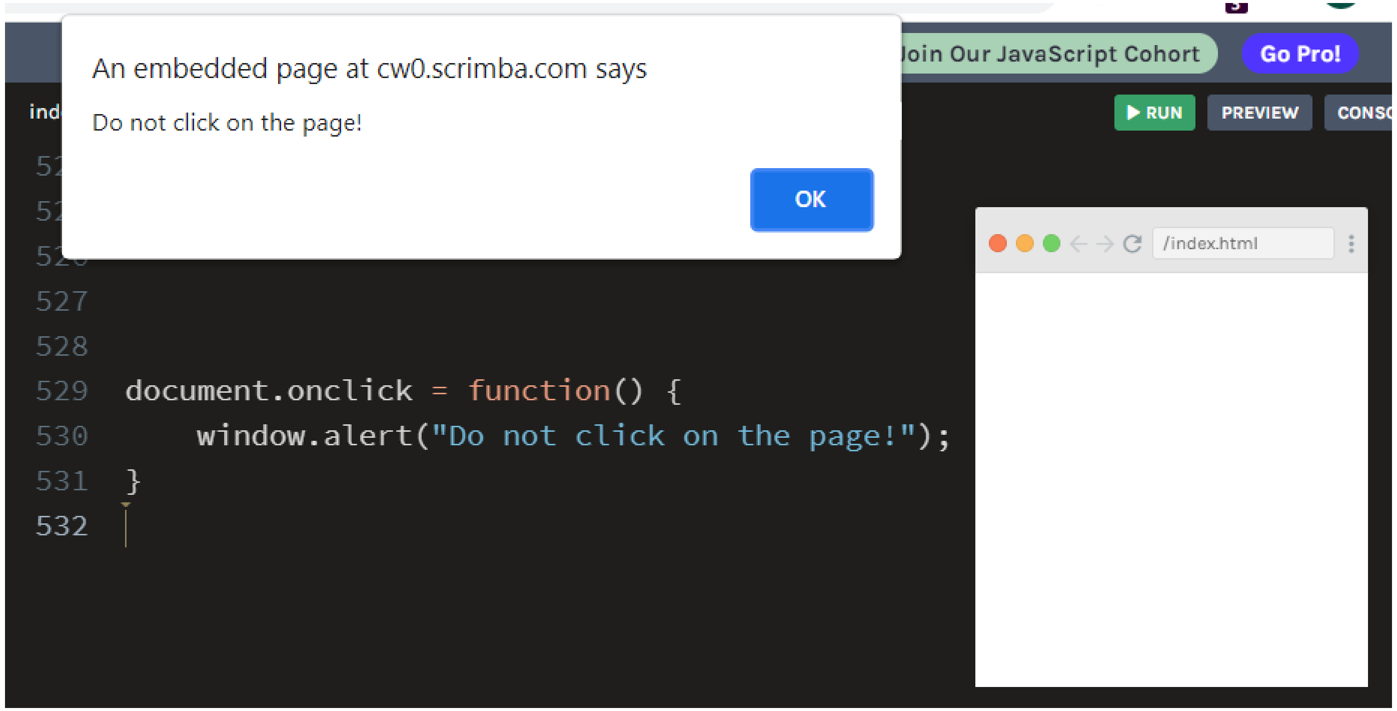
- Mmh, it does not seem so useful then…
- In fact, anonymous functions are a quite handy way to handle events without the need of declaring a function somewhere and then call it, especially if the event only occurs in one place of the JS code.
- In the example, function expression declares and calls a function in one statement. Quite convenient!



An embedded page at cw0.scrimba.com says

Do not click on the page!

```
function do_not_click() {
    window.alert("Do not click on the page!");
}
document.onclick = do_not_click();
```

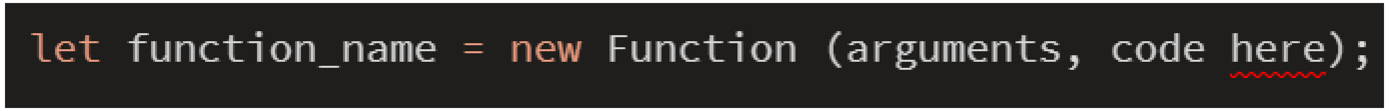


An embedded page at cw0.scrimba.com says

Do not click on the page!

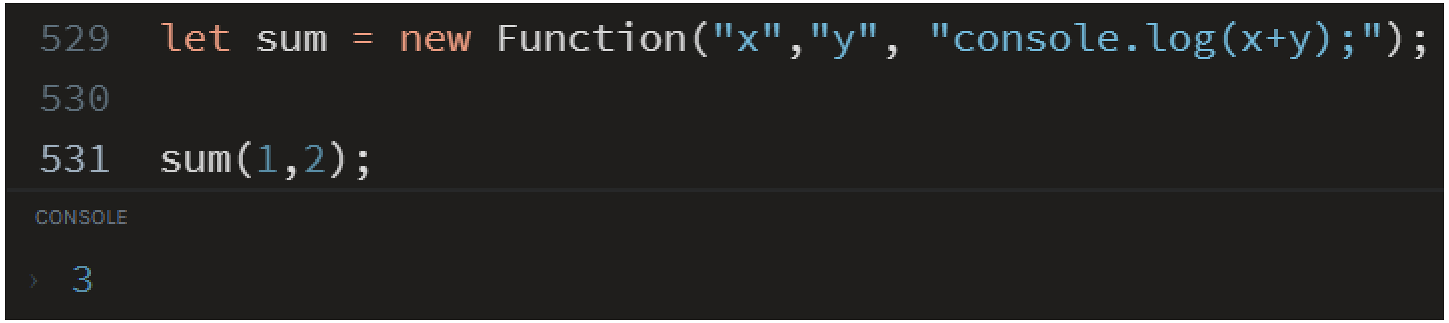
```
document.onclick = function() {
    window.alert("Do not click on the page!");
}
```

Function constructor



```
let function_name = new Function (arguments, code here);
```

- Similar to the creation of Arrays (or Objects in next lecture).
- Poorer performance than other methods because it is evaluated each time it is used instead of being parsed once.
- Only given just in case and for the sake of completeness.
- The three methods are preferred for creation of functions.



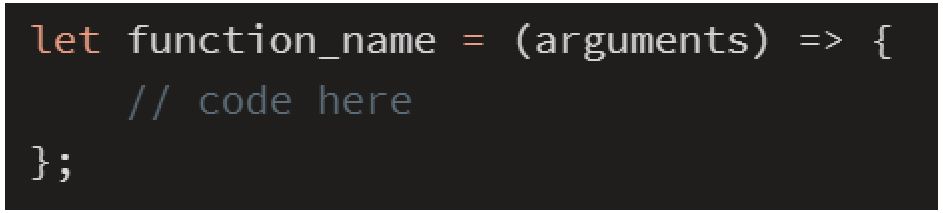
```
let sum = new Function("x","y", "console.log(x+y);");

sum(1,2);
```

CONSOLE

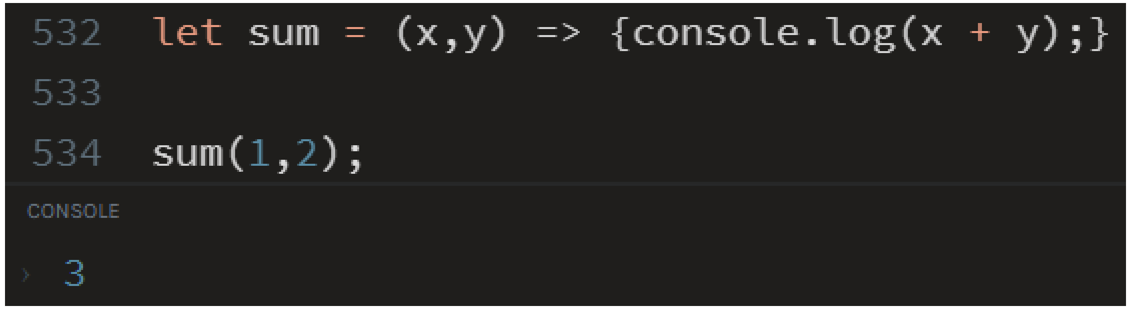
```
› 3
```

Arrow functions

- Arrow functions have been introduced in ECMAScript 6 (ES6).
- Shortens and simplifies the syntax for function expressions.
- Widely used in node modules or on the Web. So, better get familiar with it now. ☺



```
let function_name = (arguments) => {
    // code here
};
```



```
let sum = (x,y) => {console.log(x + y);}

sum(1,2);
```

```
› 3
```

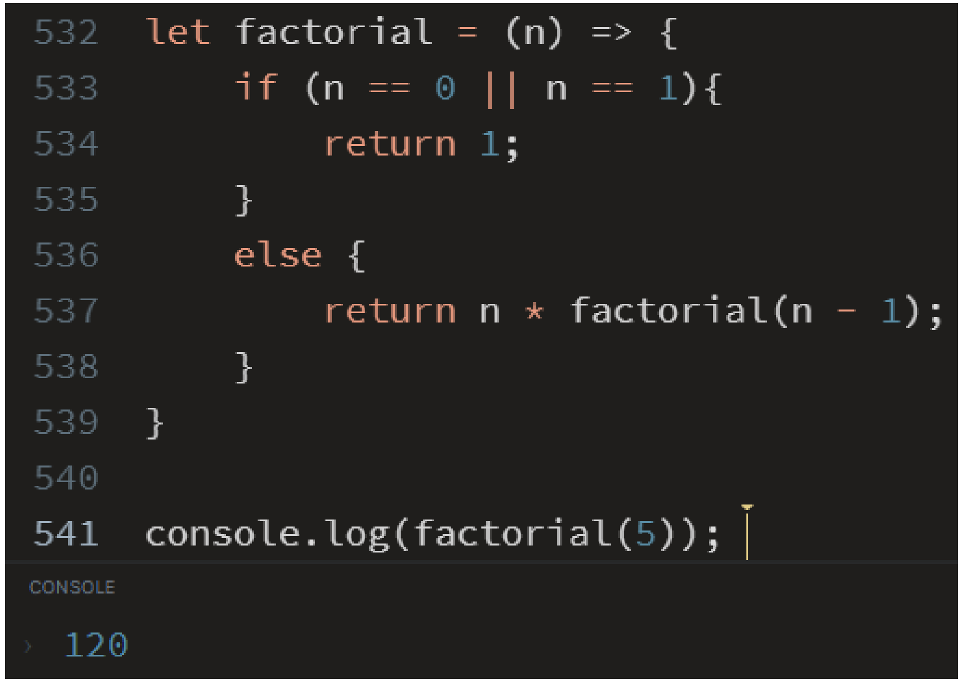
Recursive sequences: an example

- Fibonacci sequence:
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Given two initial numbers (0 and 1), the n-th Fibonacci number is defined by the sum of the two previous numbers.
- In mathematics, a sequence can be defined by a **recurrence relation** (an equation that expresses each element of a sequence as a function of the preceding ones) **and an initialisation**.
- Fibonacci(0) = 0, Fibonacci(1) = 1,

  and for n>1, Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

Recursive functions

- A recursive function is a function defined in terms of itself via self-referential expressions.
- This means that the function will continue to call itself and repeat its behaviour until some condition is met to return a result. All recursive functions share a common structure made up of two parts: base case and recursive case.



```
let factorial = (n) => {
    if (n == 0 || n == 1){
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}

console.log(factorial(5));
```

```
› 120
```

Recursive functions in JS: Pros and Cons

- Pros:
  - Provides an (easy) way of solving complex problems
  - Might be faster than an iterative solution

- Cons:
  - Might be slower
  - Requires more memory storage
  - Often throw a Stack Overflow Exception when processing or operations are too large.
  - Can be difficult to trace and debug

Activity #1: Fibonacci sequence

- Recall: Fibonacci sequence:
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Given two initial numbers (0 and 1), the n-th Fibonacci number is defined by the sum of the two previous numbers.
- Goal: Implement the Fibonacci function using recursion. The function takes an integer as input and returns the n-th Fibonacci number.

Activity #2: McCarthy 91 function

- McCarthy 91 function is defined as follows:

$$M(n) = \begin{cases} n - 10 & \text{if } n > 100, \\ M(M(n+11)) & \text{if } n \le 100. \end{cases}$$

- Implement the McCarthy function using recursion.
- Evaluate the function for every integer between -10 and 110.

Activity #3: Euclide Algorithm for the GCD

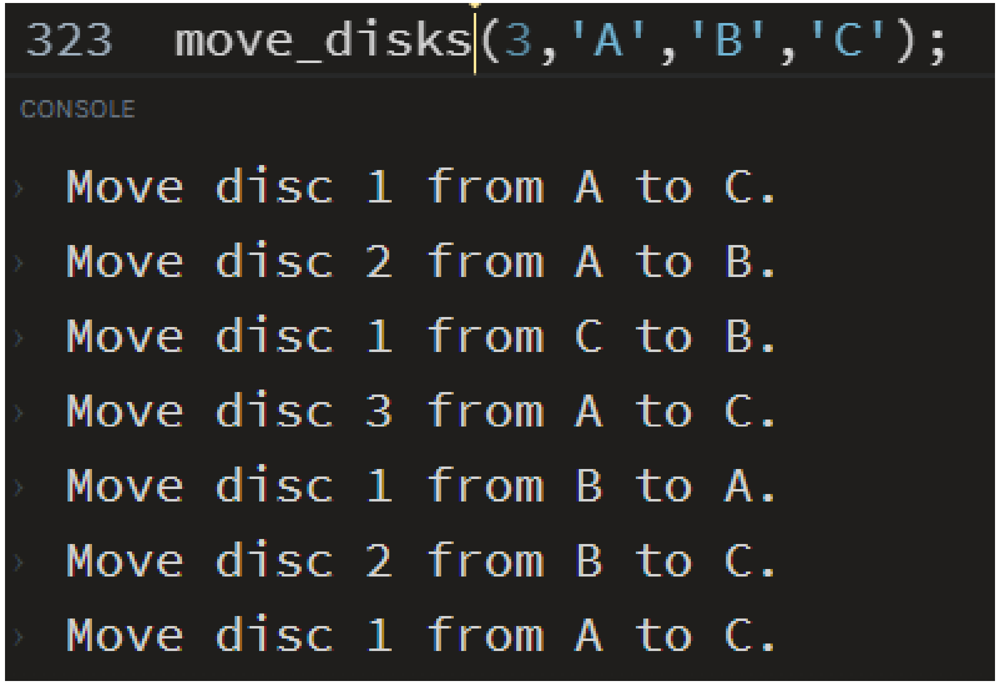
The gcd of two numbers is the greatest common divisor existing between them. For example, the gcd of 30 and 50 is 10.

Write a recursive function gcd that, given two integers a and b, returns the gcd of a and b.

Activity #4: Towers of Hanoi

Write a function move_disks(n, from_tower, aux_tower, to_tower) which returns the movement of disks that solves the Towers of Hanoi problem.

- The first argument n is an Integer input that gives information on the number of disks.
- The second argument from_tower is a String which is the label of the origin tower.
- The third argument aux_tower is a String which is the label of the auxilary tower.
- The last argument to_tower is a String which is the label of the destination tower.

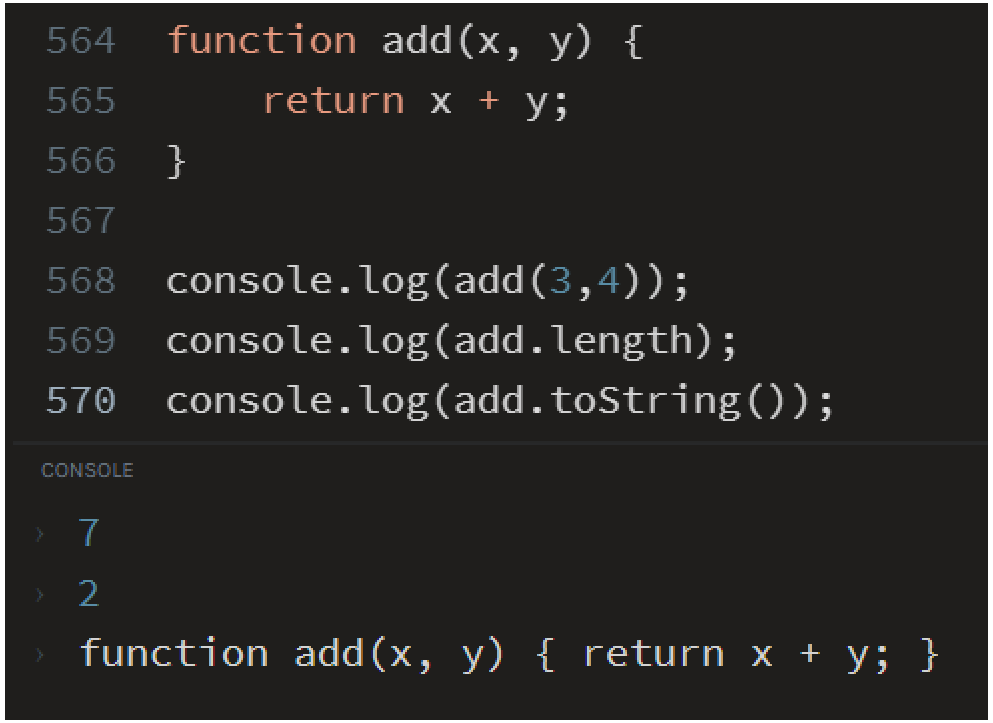


```
move_disks(3,'A','B','C');
```

```
Move disc 1 from A to C.
Move disc 2 from A to B.
Move disc 1 from C to B.
Move disc 3 from A to C.
Move disc 1 from B to A.
Move disc 2 from B to C.
Move disc 1 from A to C.
```

A few function properties

- The length property applied on a function returns the minimum number of arguments necessary to invoke this function.
- You can convert a function into a string using the **toString()** method. Can be used e.g., for debugging purposes, to determine the symbolic names of the arguments, etc.



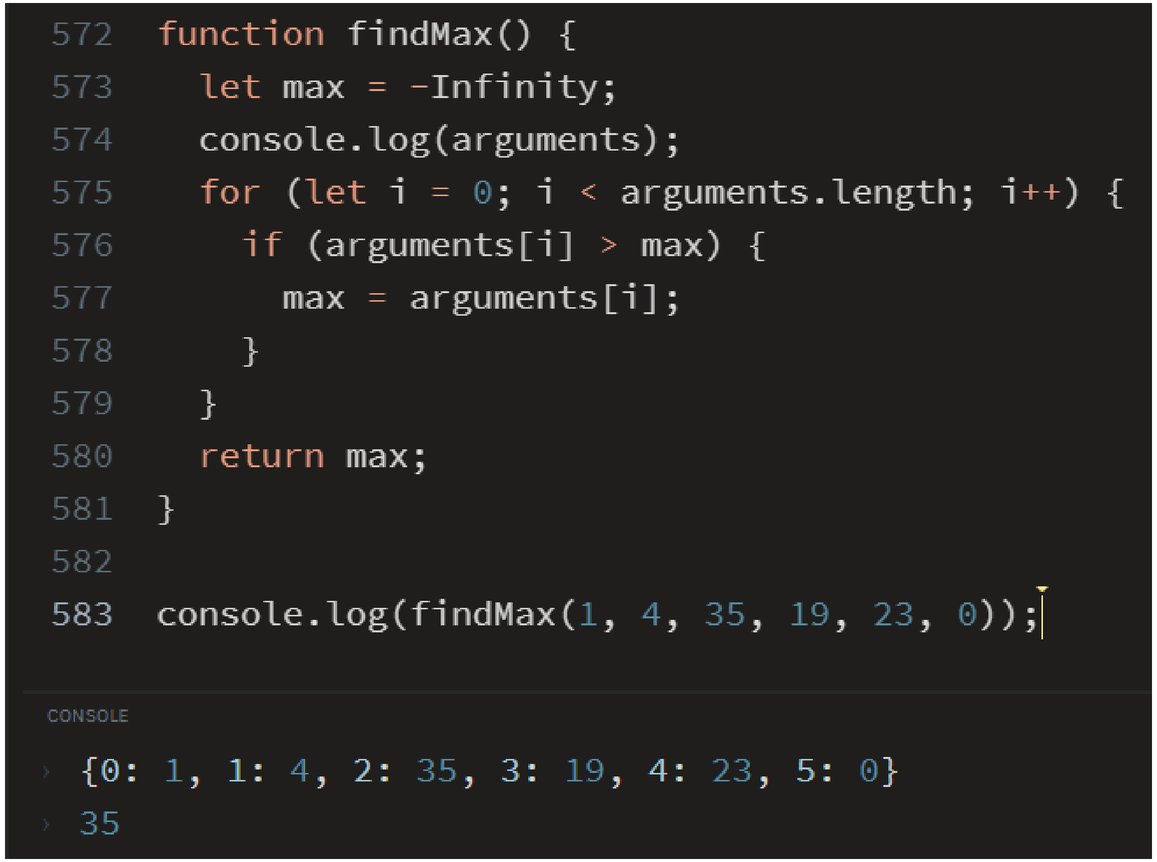
```
function add(x, y) {
    return x + y;
}

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

CONSOLE

```
7
2
function add(x, y) { return x + y; }
```

Indefinite number of arguments

- A function can have a variable number of arguments.
- Any (non-arrow) JS function has a built-in **argument** object which contains an array of the arguments used when the function was invoked.

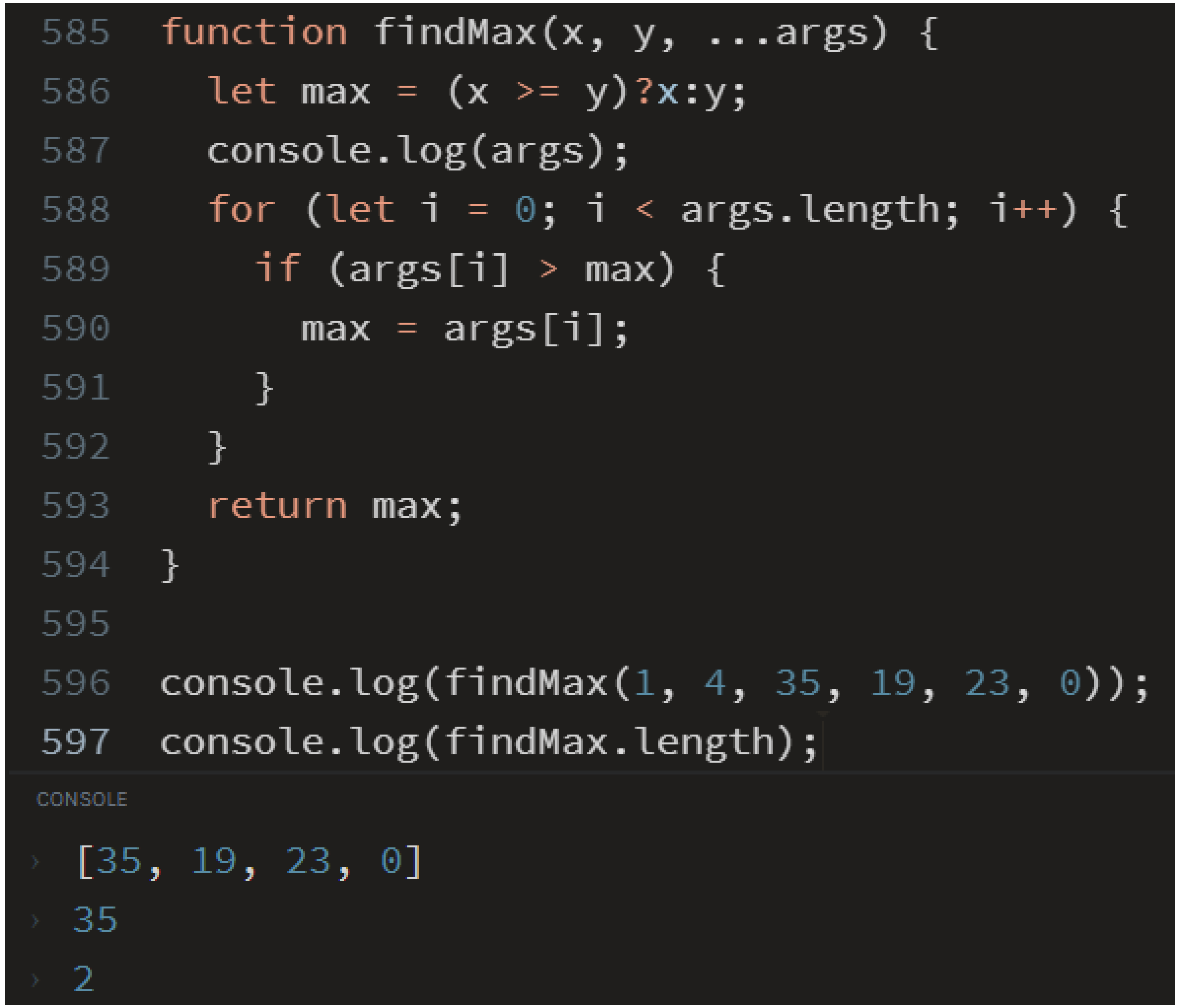


```
function findMax() {
  let max = -Infinity;
  console.log(arguments);
  for (let i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}

console.log(findMax(1, 4, 35, 19, 23, 0));
```

```
{0: 1, 1: 4, 2: 35, 3: 19, 4: 23, 5: 0}
35
```

The rest parameter (ES6)

- The rest parameter provides an easier way of working with an indefinite number of arguments.
- The syntax consists of writing 3 dots before an argument of a function. This argument is called the rest (of the inputs).
- Main difference with the arguments object:
    - Here args is an array, so you can use all the Array methods on args.
    - Works for arrow functions
    - More straightforward.

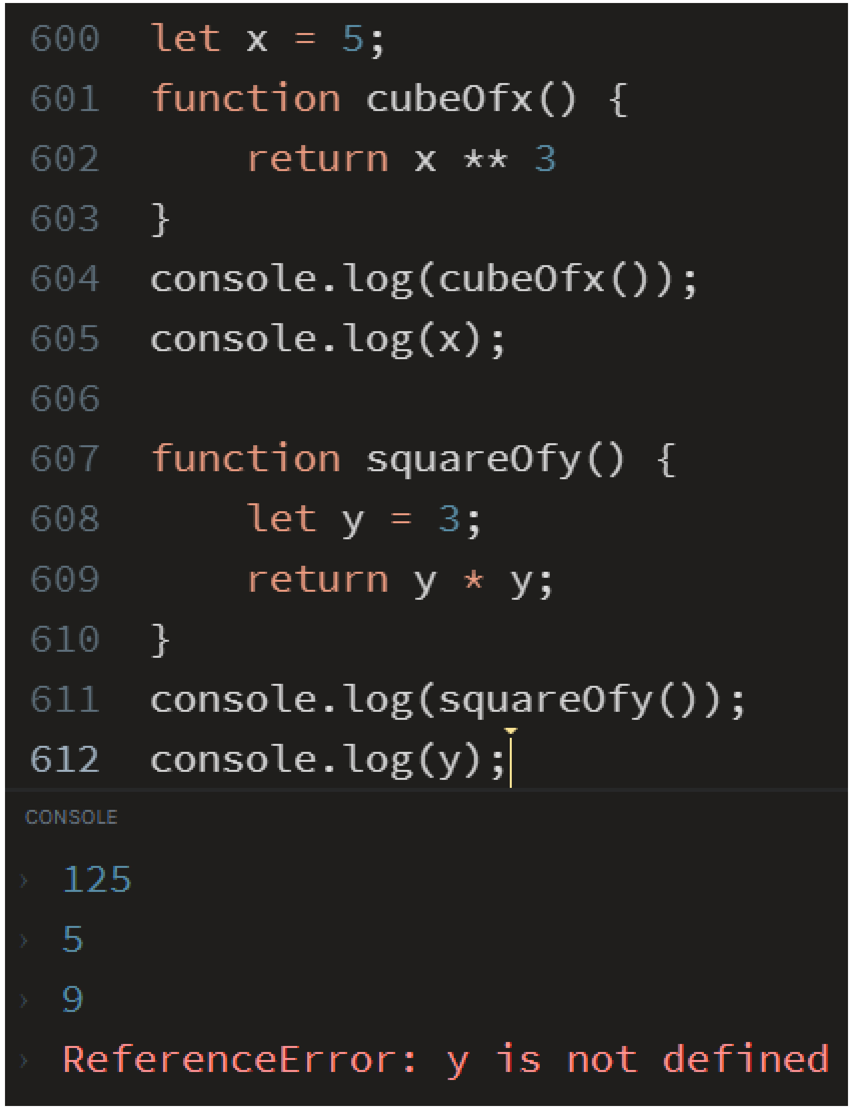


```
function findMax(x, y, ...args) {
  let max = (x >= y)?x:y;
  console.log(args);
  for (let i = 0; i < args.length; i++) {
    if (args[i] > max) {
      max = args[i];
    }
  }
  return max;
}

console.log(findMax(1, 4, 35, 19, 23, 0));
console.log(findMax.length);
```

```
[35, 19, 23, 0]
35
2
```

Recall: global vs local variables

- Global variables are accessible inside and outside of a function (e.g., x is a global variable)
- But the lifetime of a local variable is limited to the scope of its function. In the example, y is created when the function is called but it is deleted after its execution and can not be accessed anymore in the main program.



```
let x = 5;
function cubeOfx() {
    return x ** 3
}
console.log(cubeOfx());
console.log(x);

function squareOfy() {
    let y = 3;
    return y * y;
}
console.log(squareOfy());
console.log(y);
```

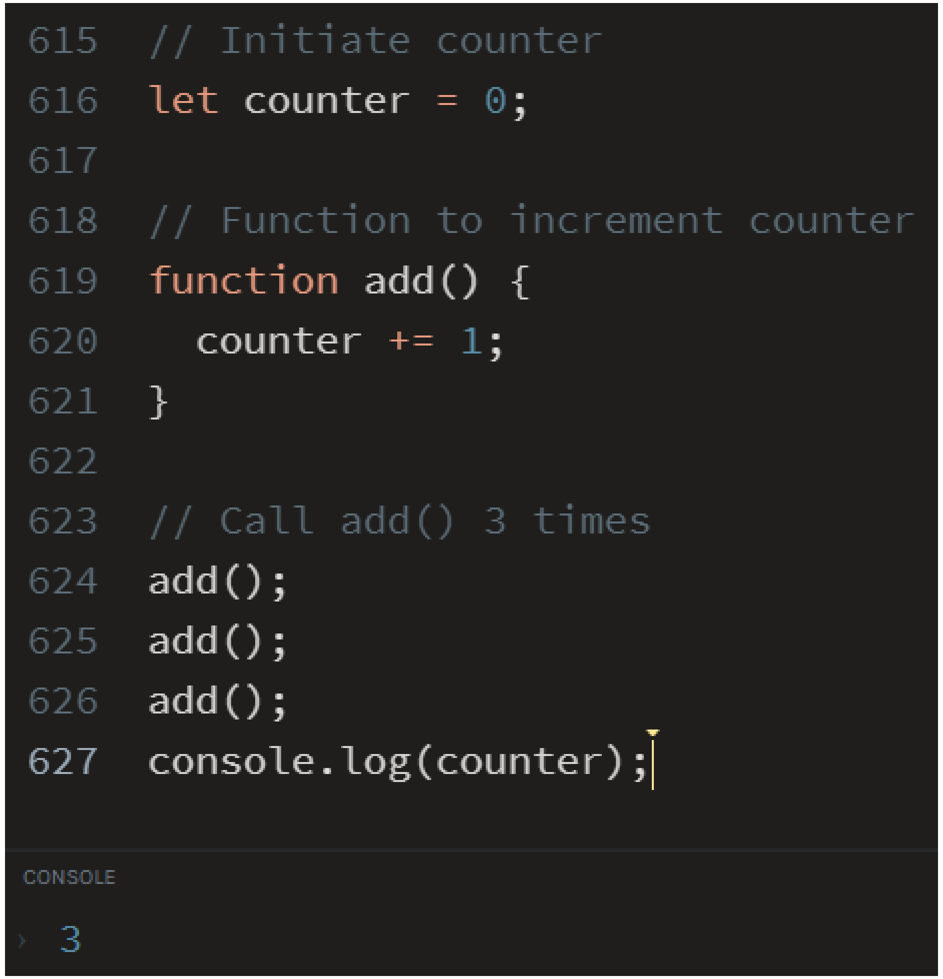
CONSOLE

```
125
5
9
ReferenceError: y is not defined
```

Closure motivation (I)

- Let counter be a variable, incremented by the add() function, which should remain readable at any time by other functions.
- Look at the example on the right.
- Does it work as expected?

Example taken from:
https://www.w3schools.com/js/js_function_closures.asp

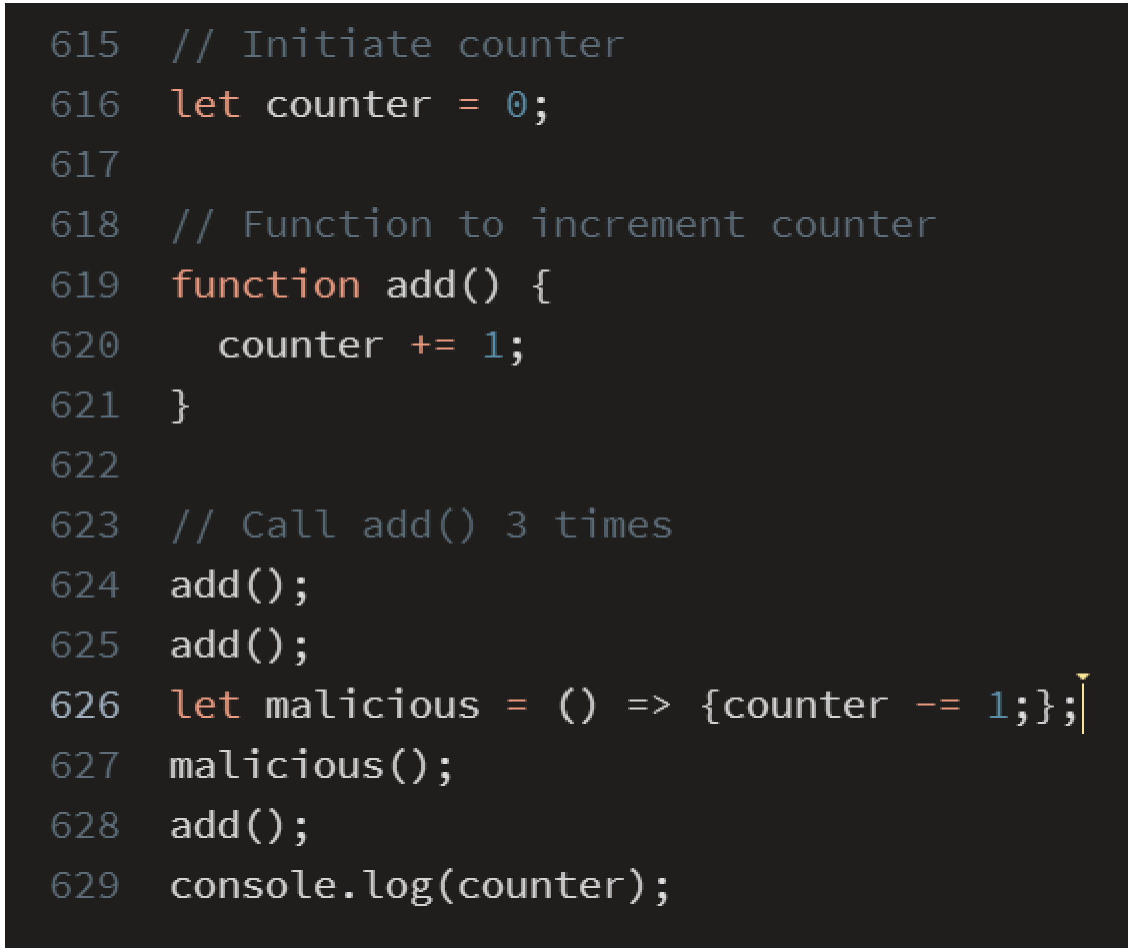


```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}

// Call add() 3 times
add();
add();
add();
console.log(counter);
```

```
› 3
```

Closure motivation (II)

- Let counter be a variable, incremented by the add() function, which should remain readable at any time by other functions.
- Look at the example on the right.
- Does it work as expected?
- Not really because any external or malicious function could modify the value of the counter.
- Here, my malicious() function accesses and decrements the global counter.

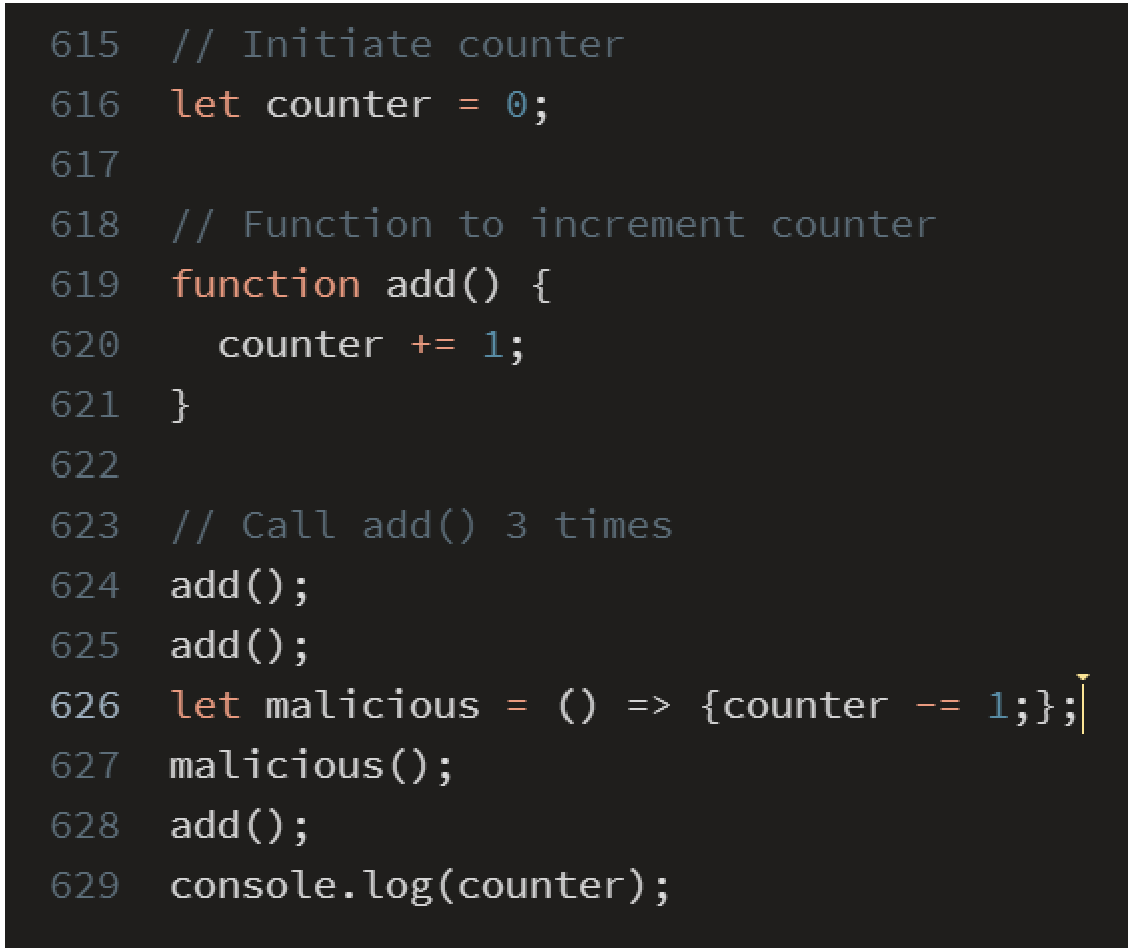


```javascript
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}

// Call add() 3 times
add();
add();
let malicious = () => {counter -= 1;};
malicious();
add();
console.log(counter);
```

Example taken from: https://www.w3schools.com/js/js_function_closures.asp

Closure motivation (III)

- Let counter be a variable, incremented by the add() function, which should remain readable at any time by other functions.
- It does not work because every function have access to the global scope.
- Making counter local to add() won’t change anything because counter will be deleted after each call to add().



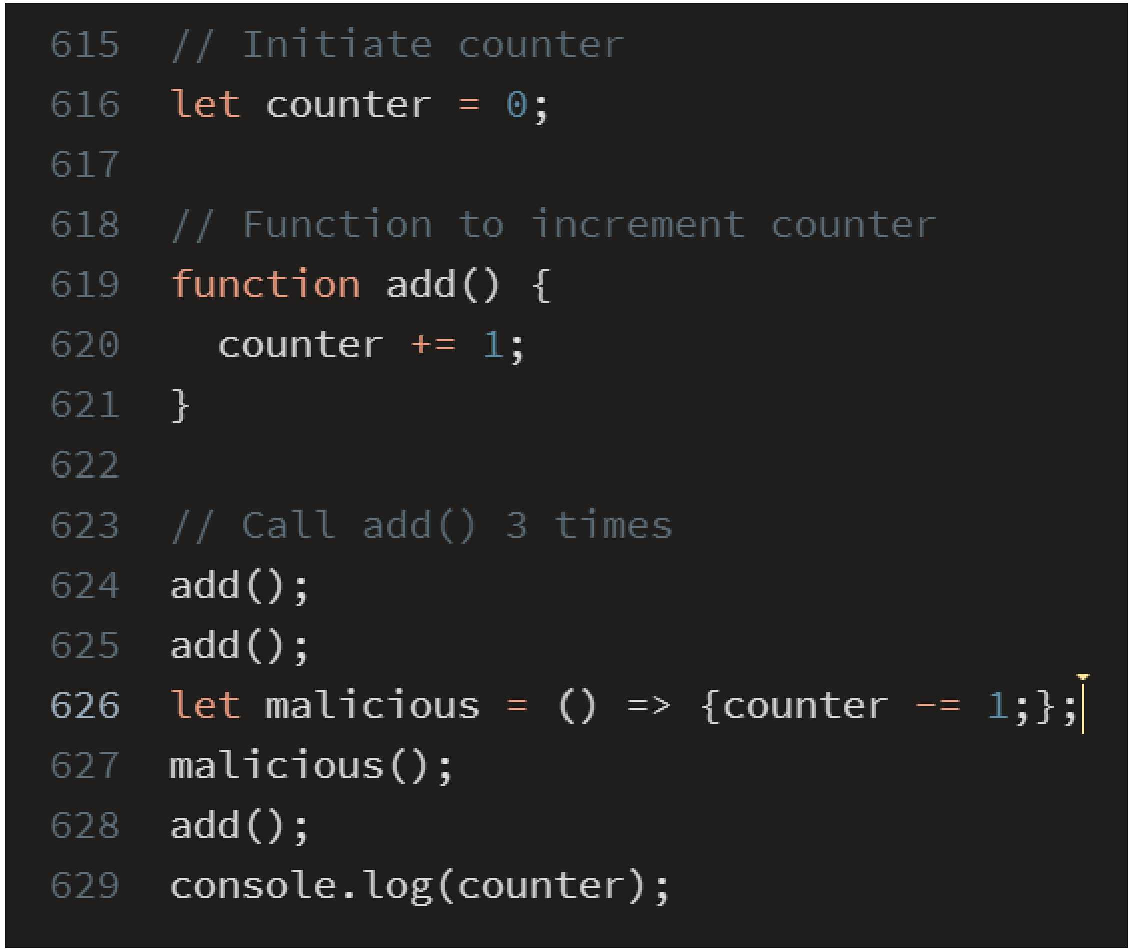
```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}

// Call add() 3 times
add();
add();
let malicious = () => {counter -= 1;};
malicious();
add();
console.log(counter);
```

Example taken from:
https://www.w3schools.com/js/js_function_closures.asp

Closure motivation (IV)

- Let counter be a variable, incremented by the add() function, which should remain readable at any time by other functions.
- It does not work because every function have access to the global scope.
- Making counter local to add() won’t change anything because counter will be deleted after each call to add().
- Only possibility: using closure with recursive functions.

Example taken from: https://www.w3schools.com/js/js_function_closures.asp

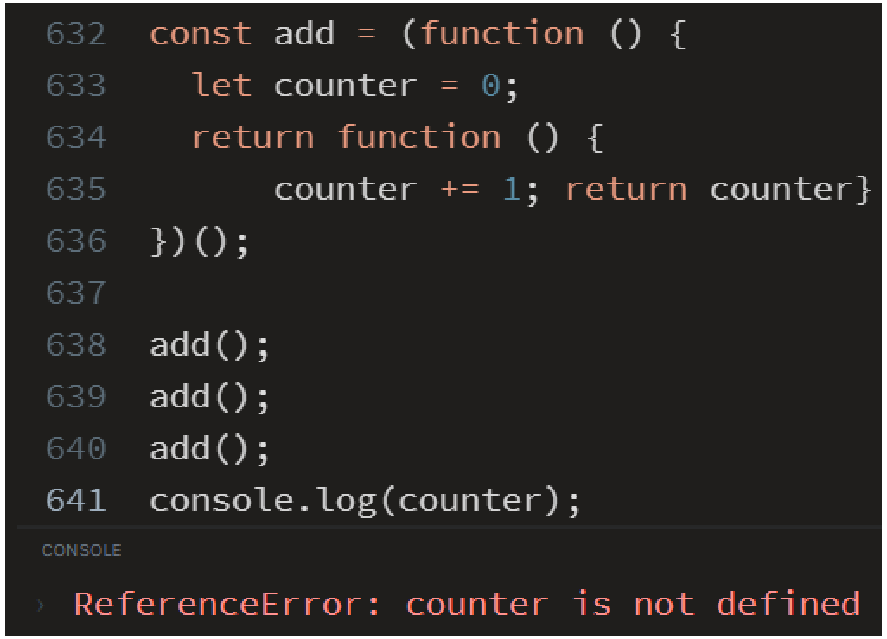


```
// Initiate counter
let counter = 0;

// Function to increment counter
function add() {
  counter += 1;
}

// Call add() 3 times
add();
add();
let malicious = () => {counter -= 1;};
malicious();
add();
console.log(counter);
```

Closure

- Let counter be a variable, incremented by the add() function, which should remain readable at any time by other functions.
- Only possibility: using closure with recursive functions.
- A closure is a function having access to the parent scope, even after the parent function has closed.
- Let's have a look to the code on the right.

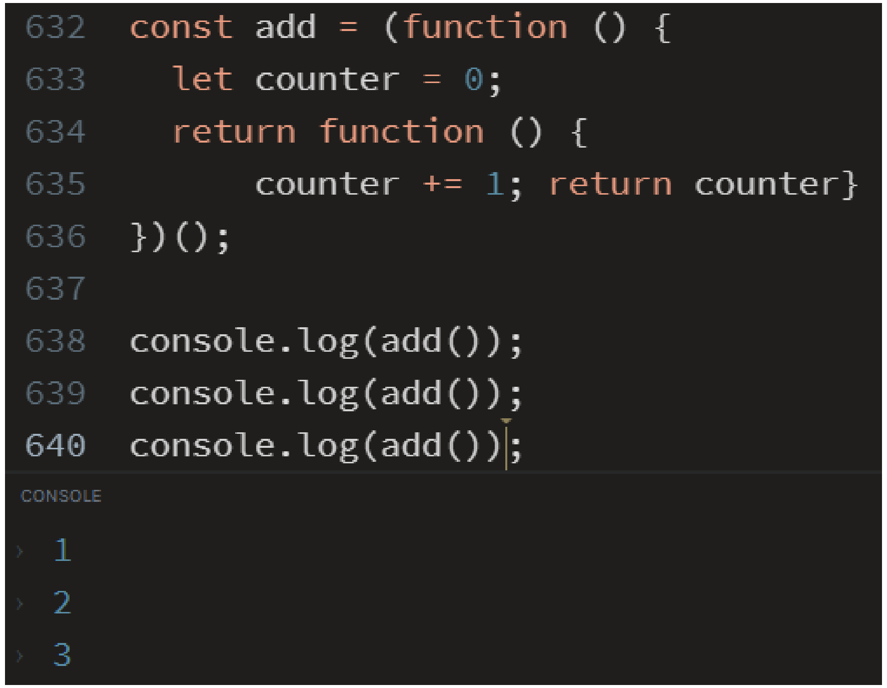
Example taken from:
https://www.w3schools.com/js/js_function_closures.asp



```
const add = (function () {
  let counter = 0;
  return function () {
      counter += 1; return counter}
})();

add();
add();
add();
console.log(counter);
```

CONSOLE

```
› ReferenceError: counter is not defined
```



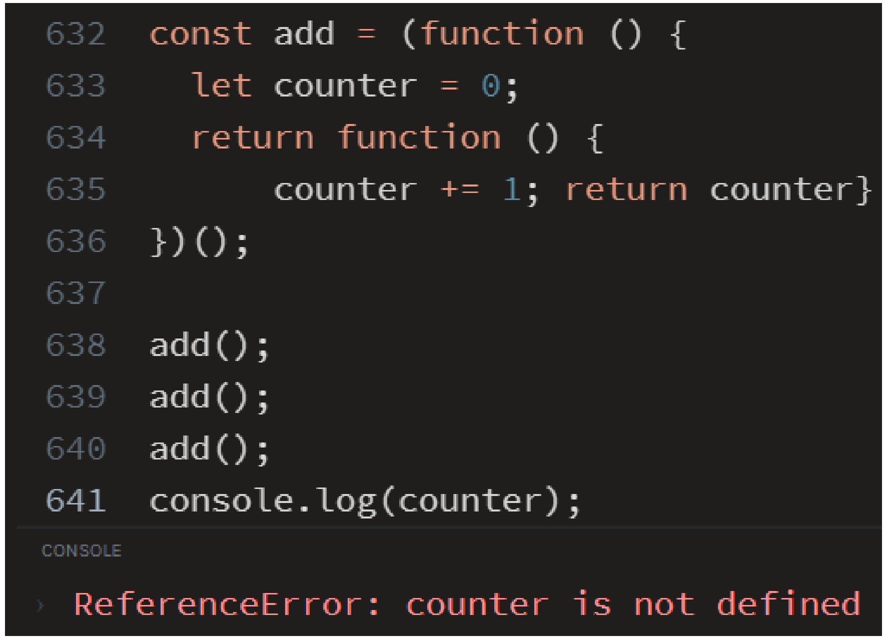
```
const add = (function () {
  let counter = 0;
  return function () {
      counter += 1; return counter}
})();

console.log(add());
console.log(add());
console.log(add());
```

CONSOLE

```
› 1
› 2
› 3
```

Closure

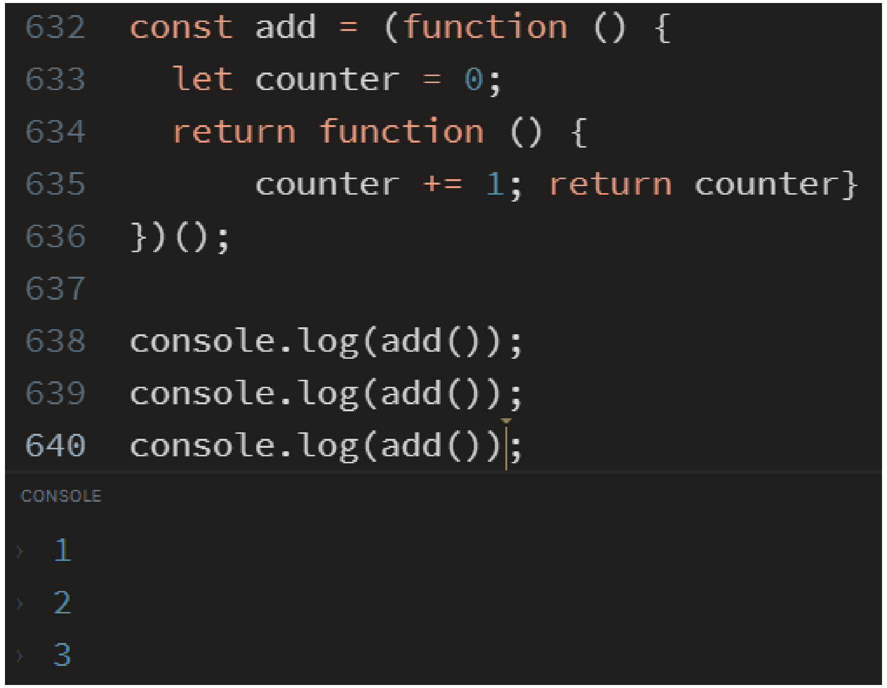
- In these blocks, **counter** is not a global variable accessible by any function.
- The variable **add** is assigned to the return value of a self-invoking function.
- The self-invoking function only runs once. It sets the counter to zero, and returns a function expression.
- This way, **add** becomes a function that can access the counter in the parent scope!
- This closure makes possible for a function to have "private" variables.
- The counter is protected by the scope of the anonymous function, and can only be changed using the add function.



```
const add = (function () {
  let counter = 0;
  return function () {
      counter += 1; return counter}
})();

add();
add();
add();
console.log(counter);
```

CONSOLE

```
› ReferenceError: counter is not defined
```



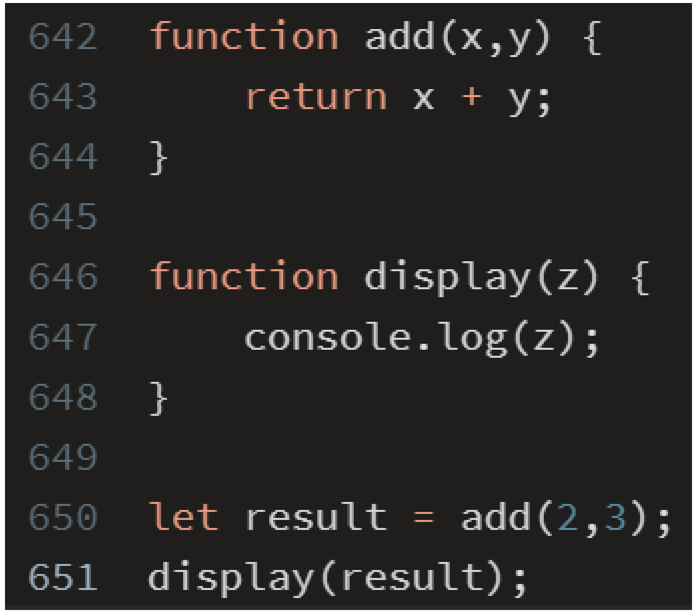
```
const add = (function () {
  let counter = 0;
  return function () {
      counter += 1; return counter}
})();

console.log(add());
console.log(add());
console.log(add());
```

CONSOLE

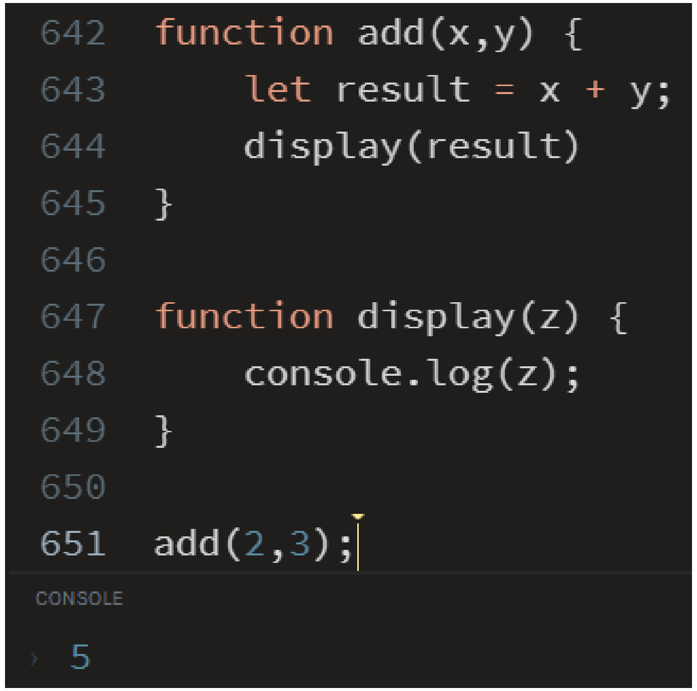
```
› 1
› 2
› 3
```

Callback motivation

- Let’s perform a calculation and then let’s display it.
- In the 1st example, we need to make two calls after the function declarations, one to add() and one to display().
- In the 2nd example, we only need to make one call to add() since display() is already called inside add().
- However, you can’t prevent add() to display the result in the 2nd example.



```
function add(x,y) {
    return x + y;
}

function display(z) {
    console.log(z);
}

let result = add(2,3);
display(result);
```

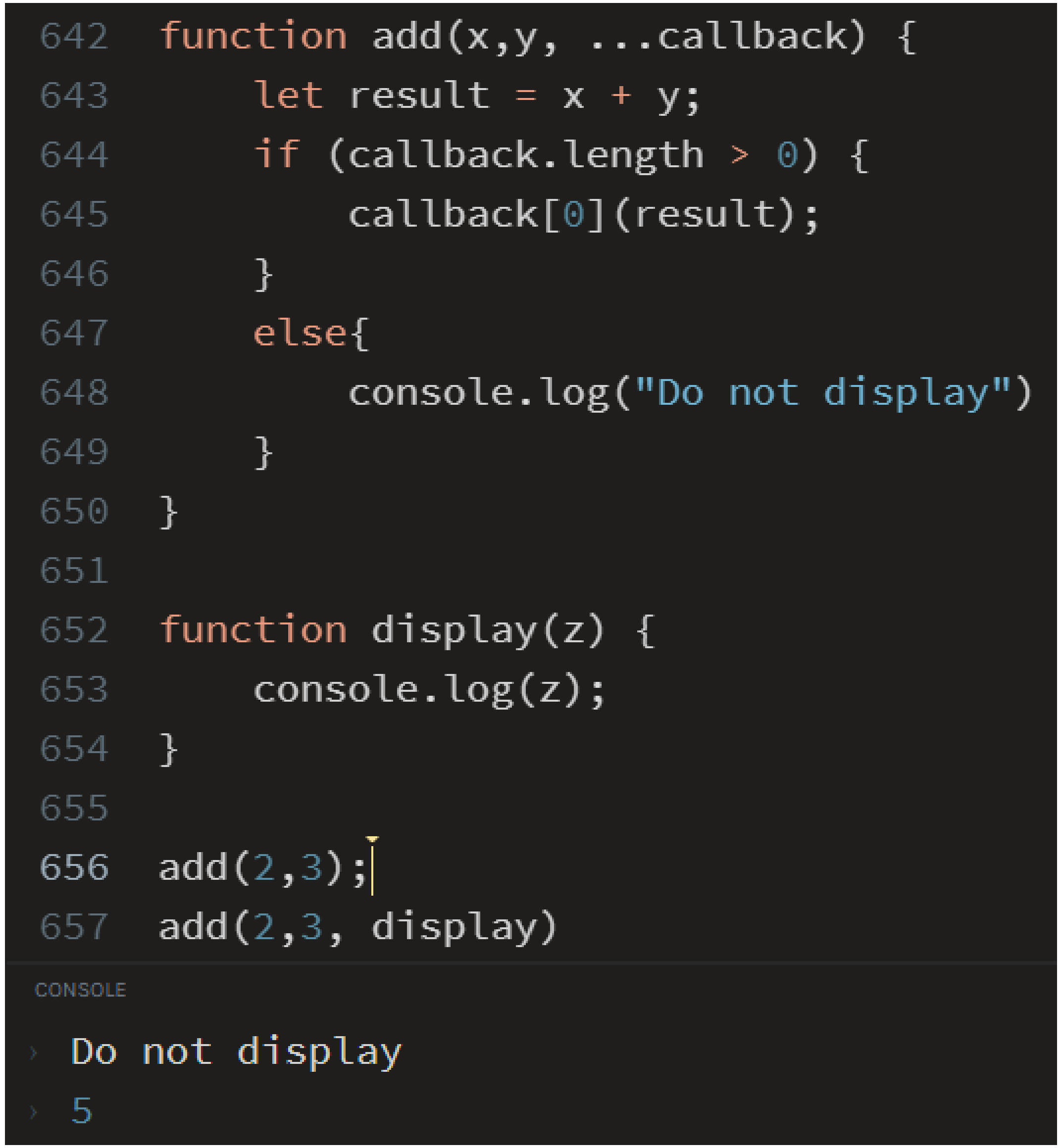


```
function add(x,y) {
    let result = x + y;
    display(result)
}

function display(z) {
    console.log(z);
}

add(2,3);
```

```
CONSOLE
› 5
```

Callback

- For more flexibility, you can use a **callback**.
- A **callback** is a function passed as an argument to another function.
- In this example, **display** is the name of a function passed as a possible argument of **add**().
- Note that we pass display, not **display**() which is a call to the display function.
- The **add**() function performs a calculation and afterwards, if specified, runs a callback function. In this example, it will run display.

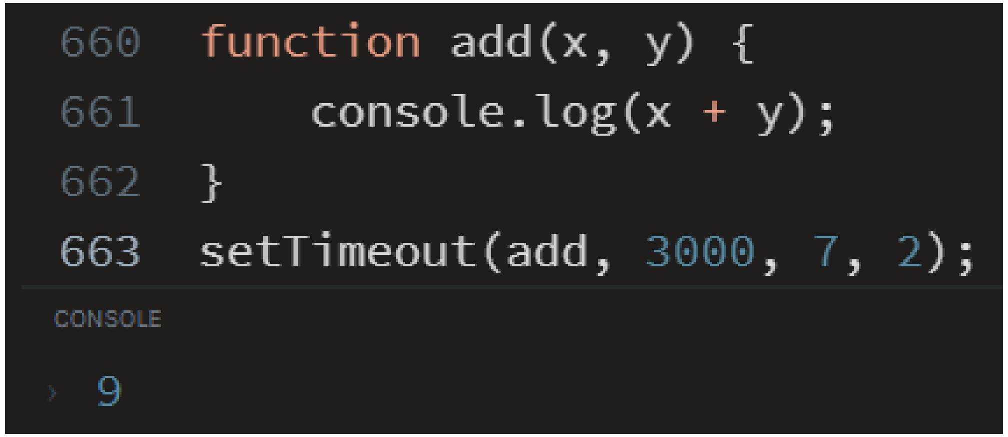


```javascript
function add(x,y, ...callback) {
    let result = x + y;
    if (callback.length > 0) {
        callback[0](result);
    }
    else{
        console.log("Do not display")
    }
}

function display(z) {
    console.log(z);
}

add(2,3);
add(2,3, display)
```

```
Do not display
5
```

Typical use of callbacks

- A typical example of callback occurs when you want to execute a function on time-out.
- setTimeout(some_func, some_time, some_func_arguments) is a built-in function which, after some_time milliseconds, calls the some_func function with the arguments specified by some_func_arguments if necessary.
- **add** is used as a callback of setTimeout().
- In the example, 9 is displayed in the console after 3 seconds.
- Callbacks are commonly used with asynchronous function (more on that next week).



```javascript
function add(x, y) {
    console.log(x + y);
}
setTimeout(add, 3000, 7, 2);
```

```
› 9
```

Conclusion

- In this session, we have covered more advanced concepts related to functions, notably:
  - Function expressions
  - Arrow functions
  - Rest parameters
  - Recursive (and/or self-invoked) functions
  - Closure
  - Callback

To do

- Don’t forget to submit all the activities in a js file in your respective folders before next lesson!
- Preferably 1 js file with the functions followed by a few tests per functions.