**FUNCTIONS**

JavaScript Function Definitions

[❮ Previous](https://www.w3schools.com/js/js_object_es5.asp)[Next ❯](https://www.w3schools.com/js/js_function_parameters.asp)

JavaScript functions are **defined** with the function keyword.

You can use a function **declaration** or a function **expression**.

Function Declarations

Earlier in this tutorial, you learned that functions are **declared** with the following syntax:

function *functionName*(*parameters*) {  
  // *code to be executed*  
}

Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).

Example

function myFunction(a, b) {  
  return a \* b;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_return)

Semicolons are used to separate executable JavaScript statements.  
Since a function **declaration** is not an executable statement, it is not common to end it with a semicolon.

Function Expressions

A JavaScript function can also be defined using an **expression**.

A function expression can be stored in a variable:

Example

const x = function (a, b) {return a \* b};

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_expression)

After a function expression has been stored in a variable, the variable can be used as a function:

Example

const x = function (a, b) {return a \* b};  
let z = x(4, 3);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_expression_variable)

The function above is actually an **anonymous function** (a function without a name).

Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

The function above ends with a semicolon because it is a part of an executable statement.

ADVERTISEMENT

The Function() Constructor

As you have seen in the previous examples, JavaScript functions are defined with the function keyword.

Functions can also be defined with a built-in JavaScript function constructor called Function().

Example

const myFunction = new Function("a", "b", "return a \* b");  
  
let x = myFunction(4, 3);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_constructor)

You actually don't have to use the function constructor. The example above is the same as writing:

Example

const myFunction = function (a, b) {return a \* b};  
  
let x = myFunction(4, 3);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_constructor2)

Most of the time, you can avoid using the new keyword in JavaScript.

Function Hoisting

Earlier in this tutorial, you learned about "hoisting" ([JavaScript Hoisting](https://www.w3schools.com/js/js_hoisting.asp)).

Hoisting is JavaScript's default behavior of moving **declarations** to the top of the current scope.

Hoisting applies to variable declarations and to function declarations.

Because of this, JavaScript functions can be called before they are declared:

myFunction(5);  
  
function myFunction(y) {  
  return y \* y;  
}

Functions defined using an expression are not hoisted.

Self-Invoking Functions

Function expressions can be made "self-invoking".

A self-invoking expression is invoked (started) automatically, without being called.

Function expressions will execute automatically if the expression is followed by ().

You cannot self-invoke a function declaration.

You have to add parentheses around the function to indicate that it is a function expression:

Example

(function () {  
  let x = "Hello!!";  // I will invoke myself  
})();

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_expression_self)

The function above is actually an **anonymous self-invoking function** (function without name).

Functions Can Be Used as Values

JavaScript functions can be used as values:

Example

function myFunction(a, b) {  
  return a \* b;  
}  
  
let x = myFunction(4, 3);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_value)

JavaScript functions can be used in expressions:

Example

function myFunction(a, b) {  
  return a \* b;  
}  
  
let x = myFunction(4, 3) \* 2;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_value2)

Functions are Objects

The typeof operator in JavaScript returns "function" for functions.

But, JavaScript functions can best be described as objects.

JavaScript functions have both **properties** and **methods**.

The arguments.length property returns the number of arguments received when the function was invoked:

Example

function myFunction(a, b) {  
  return arguments.length;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_length)

The toString() method returns the function as a string:

Example

function myFunction(a, b) {  
  return a \* b;  
}  
  
let text = myFunction.toString();

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_tostring)

A function defined as the property of an object, is called a method to the object.  
A function designed to create new objects, is called an object constructor.

Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

Example

// ES5  
var x = function(x, y) {  
  return x \* y;  
}  
  
// ES6  
const x = (x, y) => x \* y;

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_es6_arrow)

Arrow functions do not have their own this. They are not well suited for defining **object methods**.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using const is safer than using var, because a function expression is always constant value.

You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

Example

const x = (x, y) => { return x \* y };

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_es6_arrow_safe)

JavaScript Function Parameters

[❮ Previous](https://www.w3schools.com/js/js_function_definition.asp)[Next ❯](https://www.w3schools.com/js/js_function_invocation.asp)

A JavaScript function does not perform any checking on parameter values (arguments).

Function Parameters and Arguments

Earlier in this tutorial, you learned that functions can have **parameters**:

function *functionName*(*parameter1, parameter2, parameter3*) {  
  // *code to be executed*  
}

Function **parameters** are the **names** listed in the function definition.

Function **arguments** are the real **values** passed to (and received by) the function.

Parameter Rules

JavaScript function definitions do not specify data types for parameters.

JavaScript functions do not perform type checking on the passed arguments.

JavaScript functions do not check the number of arguments received.

Default Parameters

If a function is called with **missing arguments** (less than declared), the missing values are set to undefined.

Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter:

Example

function myFunction(x, y) {  
  if (y === undefined) {  
    y = 2;  
  }  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_undefined)

[ECMAScript 2015](https://www.w3schools.com/js/js_es6.asp) allows default parameter values in the function declaration:

function myFunction(x, y = 2) {  
  // function code  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_default)

ADVERTISEMENT

The Arguments Object

JavaScript functions have a built-in object called the arguments object.

The argument object contains an array of the arguments used when the function was called (invoked).

This way you can simply use a function to find (for instance) the highest value in a list of numbers:

Example

x = findMax(1, 123, 500, 115, 44, 88);  
  
function findMax() {  
  let max = -Infinity;  
  for (let i = 0; i < arguments.length; i++) {  
    if (arguments[i] > max) {  
      max = arguments[i];  
    }  
  }  
  return max;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_arguments)

Or create a function to sum all input values:

Example

x = sumAll(1, 123, 500, 115, 44, 88);  
  
function sumAll() {  
  let sum = 0;  
  for (let i = 0; i < arguments.length; i++) {  
    sum += arguments[i];  
  }  
  return sum;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_arguments_sum)

If a function is called with **too many arguments** (more than declared), these arguments can be reached using **the arguments object**.

Arguments are Passed by Value

The parameters, in a function call, are the function's arguments.

JavaScript arguments are passed by **value**: The function only gets to know the values, not the argument's locations.

If a function changes an argument's value, it does not change the parameter's original value.

**Changes to arguments are not visible (reflected) outside the function.**

JavaScript Function Invocation

[❮ Previous](https://www.w3schools.com/js/js_function_parameters.asp)[Next ❯](https://www.w3schools.com/js/js_function_call.asp)

The code inside a JavaScript function will execute when "something" invokes it.

Invoking a JavaScript Function

The code inside a function is not executed when the function is **defined**.

The code inside a function is executed when the function is **invoked**.

It is common to use the term "**call a function**" instead of "**invoke a function**".

It is also common to say "call upon a function", "start a function", or "execute a function".

In this tutorial, we will use **invoke**, because a JavaScript function can be invoked without being called.

Invoking a Function as a Function

Example

function myFunction(a, b) {  
  return a \* b;  
}  
myFunction(10, 2);           // Will return 20

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_invoke_function)

The function above does not belong to any object. But in JavaScript there is always a default global object.

In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.

In a browser the page object is the browser window. The function above automatically becomes a window function.

Note

This is a common way to invoke a JavaScript function, but not a very good practice.  
Global variables, methods, or functions can easily create name conflicts and bugs in the global object.

myFunction() and window.myFunction() is the same function:

Example

function myFunction(a, b) {  
  return a \* b;  
}  
window.myFunction(10, 2);    // Will also return 20

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_invoke_function_2)

What is **this**?

In JavaScript, the this keyword refers to an **object**.

**Which** object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

|  |
| --- |
| In an object method, this refers to the **object**. |
| Alone, this refers to the **global object**. |
| In a function, this refers to the **global object**. |
| In a function, in strict mode, this is undefined. |
| In an event, this refers to the **element** that received the event. |
| Methods like call(), apply(), and bind() can refer this to **any object**. |

Note

this is not a variable. It is a keyword. You cannot change the value of this.

See Also:

[The JavaScript **this** Tutorial](https://www.w3schools.com/js/js_this.asp)

ADVERTISEMENT

The Global Object

When a function is called without an owner object, the value of this becomes the global object.

In a web browser the global object is the browser window.

This example returns the window object as the value of this:

Example

let x = myFunction();            // x will be the window object  
  
function myFunction() {  
  return this;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_invoke_function_3)

Invoking a function as a global function, causes the value of **this** to be the global object.  
Using the window object as a variable can easily crash your program.

Invoking a Function as a Method

In JavaScript you can define functions as object methods.

The following example creates an object (**myObject**), with two properties (**firstName** and **lastName**), and a method (**fullName**):

Example

const myObject = {  
  firstName:"John",  
  lastName: "Doe",  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  }  
}  
myObject.fullName();         // Will return "John Doe"

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_invoke_method)

The **fullName** method is a function. The function belongs to the object. **myObject** is the owner of the function.

The thing called this, is the object that "owns" the JavaScript code. In this case the value of this is **myObject**.

Test it! Change the **fullName** method to return the value of this:

Example

const myObject = {  
  firstName:"John",  
  lastName: "Doe",  
  fullName: function () {  
    return this;  
  }  
}  
  
// This will return [object Object] (the owner object)  
myObject.fullName();

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_invoke_method_2)

Invoking a function as an object method, causes the value of this to be the object itself.

Invoking a Function with a Function Constructor

If a function invocation is preceded with the new keyword, it is a constructor invocation.

It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

Example

// This is a function constructor:  
function myFunction(arg1, arg2) {  
  this.firstName = arg1;  
  this.lastName  = arg2;  
}  
  
// This creates a new object  
const myObj = new myFunction("John", "Doe");  
  
// This will return "John"  
myObj.firstName;

JavaScript Function call()

[❮ Previous](https://www.w3schools.com/js/js_function_invocation.asp)[Next ❯](https://www.w3schools.com/js/js_function_apply.asp)

Method Reuse

With the call() method, you can write a method that can be used on different objects.

All Functions are Methods

In JavaScript all functions are object methods.

If a function is not a method of a JavaScript object, it is a function of the global object (see previous chapter).

The example below creates an object with 3 properties, firstName, lastName, fullName.

Example

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  }  
}  
  
// This will return "John Doe":  
person.fullName();

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_method)

In the example above, this refers to the **person object**.

**this.firstName** means the **firstName** property of **this**.

Same as:

**this.firstName** means the **firstName** property of **person**.

What is **this**?

In JavaScript, the this keyword refers to an **object**.

**Which** object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

|  |
| --- |
| In an object method, this refers to the **object**. |
| Alone, this refers to the **global object**. |
| In a function, this refers to the **global object**. |
| In a function, in strict mode, this is undefined. |
| In an event, this refers to the **element** that received the event. |
| Methods like call(), apply(), and bind() can refer this to **any object**. |

Note

this is not a variable. It is a keyword. You cannot change the value of this.

See Also:

[The JavaScript **this** Tutorial](https://www.w3schools.com/js/js_this.asp)

ADVERTISEMENT

The JavaScript call() Method

The call() method is a predefined JavaScript method.

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With call(), an object can use a method belonging to another object.

This example calls the **fullName** method of person, using it on **person1**:

Example

const person = {  
  **fullName**: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
const person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
const person2 = {  
  firstName:"Mary",  
  lastName: "Doe"  
}  
  
// This will return "John Doe":  
person.fullName.call(**person1**);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_call)

This example calls the **fullName** method of person, using it on **person2**:

Example

const person = {  
  **fullName**: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
const person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
const person2 = {  
  firstName:"Mary",  
  lastName: "Doe"  
}  
  
// This will return "Mary Doe"  
person.fullName.call(**person2**);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_call2)

The call() Method with Arguments

The call() method can accept arguments:

Example

const person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
  
const person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
  
person.fullName.call(person1, "Oslo", "Norway");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_arguments)

JavaScript Function apply()

[❮ Previous](https://www.w3schools.com/js/js_function_call.asp)[Next ❯](https://www.w3schools.com/js/js_function_bind.asp)

Method Reuse

With the apply() method, you can write a method that can be used on different objects.

The JavaScript apply() Method

The apply() method is similar to the call() method (previous chapter).

In this example the **fullName** method of **person** is **applied** on **person1**:

Example

const person = {  
  fullName: function() {  
    return this.firstName + " " + this.lastName;  
  }  
}  
  
const person1 = {  
  firstName: "Mary",  
  lastName: "Doe"  
}  
  
// This will return "Mary Doe":  
person.fullName.apply(person1);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_apply)

The Difference Between call() and apply()

The difference is:

The call() method takes arguments **separately**.

The apply() method takes arguments as an **array**.

The apply() method is very handy if you want to use an array instead of an argument list.

The apply() Method with Arguments

The apply() method accepts arguments in an array:

Example

const person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
  
const person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
  
person.fullName.apply(person1, ["Oslo", "Norway"]);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_apply_arguments)

Compared with the call() method:

Example

const person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
  
const person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
  
person.fullName.call(person1, "Oslo", "Norway");

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_arguments)

ADVERTISEMENT

Simulate a Max Method on Arrays

You can find the largest number (in a list of numbers) using the Math.max() method:

Example

Math.max(1,2,3);  // Will return 3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_math)

Since JavaScript **arrays** do not have a max() method, you can apply the Math.max() method instead.

Example

Math.max.apply(null, [1,2,3]); // Will also return 3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_math_apply)

The first argument (null) does not matter. It is not used in this example.

These examples will give the same result:

Example

Math.max.apply(Math, [1,2,3]); // Will also return 3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_math_apply1)

Example

Math.max.apply(" ", [1,2,3]); // Will also return 3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_math_apply2)

Example

Math.max.apply(0, [1,2,3]); // Will also return 3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_call_math_apply3)

JavaScript Strict Mode

In JavaScript strict mode, if the first argument of the apply() method is not an object, it becomes the owner (object) of the invoked function. In "non-strict" mode, it becomes the global object.

JavaScript Function bind()

[❮ Previous](https://www.w3schools.com/js/js_function_apply.asp)[Next ❯](https://www.w3schools.com/js/js_function_closures.asp)

Function Borrowing

With the bind() method, an object can borrow a method from another object.

The example below creates 2 objects (person and member).

The member object borrows the fullname method from the person object:

Example

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  }  
}  
  
const member = {  
  firstName:"Hege",  
  lastName: "Nilsen",  
}  
  
let fullName = person.fullName.bind(member);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_bind_borrow)

Preserving **this**

Sometimes the bind() method has to be used to prevent loosing **this**.

In the following example, the person object has a display method. In the display method, **this** refers to the person object:

Example

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  display: function () {  
    let x = document.getElementById("demo");  
    x.innerHTML = this.firstName + " " + this.lastName;  
  }  
}  
  
person.display();

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_bind1)

When a function is used as a callback, **this** is lost.

This example will try to display the person name after 3 seconds, but it will display **undefined** instead:

Example

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  display: function () {  
    let x = document.getElementById("demo");  
    x.innerHTML = this.firstName + " " + this.lastName;  
  }  
}  
  
setTimeout(person.display, 3000);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_bind)

The bind() method solves this problem.

In the following example, the bind() method is used to bind person.display to person.

This example will display the person name after 3 seconds:

Example

const person = {  
  firstName:"John",  
  lastName: "Doe",  
  display: function () {  
    let x = document.getElementById("demo");  
    x.innerHTML = this.firstName + " " + this.lastName;  
  }  
}  
  
let display = person.display.bind(person);  
setTimeout(display, 3000);

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_bind2)

ADVERTISEMENT

What is **this**?

In JavaScript, the this keyword refers to an **object**.

**Which** object depends on how this is being invoked (used or called).

The this keyword refers to different objects depending on how it is used:

|  |
| --- |
| In an object method, this refers to the **object**. |
| Alone, this refers to the **global object**. |
| In a function, this refers to the **global object**. |
| In a function, in strict mode, this is undefined. |
| In an event, this refers to the **element** that received the event. |
| Methods like call(), apply(), and bind() can refer this to **any object**. |

JavaScript Closures

[❮ Previous](https://www.w3schools.com/js/js_function_bind.asp)[Next ❯](https://www.w3schools.com/js/js_class_intro.asp)

JavaScript variables can belong to the **local** or **global** scope.

Global variables can be made local (private) with **closures**.

Global Variables

A function can access all variables defined **inside** the function, like this:

Example

function myFunction() {  
  let a = 4;  
  return a \* a;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_scope2)

But a function can also access variables defined **outside** the function, like this:

Example

let a = 4;  
function myFunction() {  
  return a \* a;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_scope1)

In the last example, **a** is a **global** variable.

In a web page, global variables belong to the window object.

Global variables can be used (and changed) by all scripts in the page (and in the window).

In the first example, **a** is a **local** variable.

A local variable can only be used inside the function where it is defined. It is hidden from other functions and other scripting code.

Global and local variables with the same name are different variables. Modifying one, does not modify the other.

Variables created **without** a declaration keyword (var, let, or const) are always global, even if they are created inside a function.

Example

function myFunction() {  
  a = 4;  
}

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_scope4)

ADVERTISEMENT

Variable Lifetime

Global variables live until the page is discarded, like when you navigate to another page or close the window.

Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

A Counter Dilemma

Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.

You could use a global variable, and a function to increase the counter:

Example

// Initiate counter  
let counter = 0;  
  
// Function to increment counter  
function add() {  
  counter += 1;  
}  
  
// Call add() 3 times  
add();  
add();  
add();  
  
// The counter should now be 3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_closures1)

There is a problem with the solution above: Any code on the page can change the counter, without calling add().

The counter should be local to the add() function, to prevent other code from changing it:

Example

// Initiate counter  
let counter = 0;  
  
// Function to increment counter  
function add() {  
  let counter = 0;  
  counter += 1;  
}  
  
// Call add() 3 times  
add();  
add();  
add();  
  
//The counter should now be 3. But it is 0

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_closures2)

It did not work because we display the global counter instead of the local counter.

We can remove the global counter and access the local counter by letting the function return it:

Example

// Function to increment counter  
function add() {  
  let counter = 0;  
  counter += 1;  
  return counter;  
}  
  
// Call add() 3 times  
add();  
add();  
add();  
  
//The counter should now be 3. But it is 1.

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_closures3)

It did not work because we reset the local counter every time we call the function.

**A JavaScript inner function can solve this.**

JavaScript Nested Functions

All functions have access to the global scope.

In fact, in JavaScript, all functions have access to the scope "above" them.

JavaScript supports nested functions. Nested functions have access to the scope "above" them.

In this example, the inner function plus() has access to the counter variable in the parent function:

Example

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

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_closures4)

This could have solved the counter dilemma, if we could reach the plus() function from the outside.

We also need to find a way to execute counter = 0 only once.

**We need a closure.**

JavaScript Closures

Remember self-invoking functions? What does this function do?

Example

const add = (function () {  
  let counter = 0;  
  return function () {counter += 1; return counter}  
})();  
  
add();  
add();  
add();  
  
// the counter is now 3

[Try it Yourself »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_function_closures5)

Example Explained

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 (0), and returns a function expression.

This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.

This is called a JavaScript **closure.** It makes it 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.