

[Technologies](#) ▼[References & Guides](#) ▼[Feedback](#) ▼[Sign in](#)  

# Functions

[« Previous](#)[Next »](#)

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a task or calculates a value. To use a function, you must define it somewhere in the scope from which you wish to call it.

See also the exhaustive reference chapter about JavaScript functions to get to know the details.

## Defining functions

### Function declarations

A **function definition** (also called a **function declaration**, or **function statement**) consists of the `function` keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.

- The JavaScript statements that define the function, enclosed in curly brackets, `{ }`.

For example, the following code defines a simple function named `square`:

```
1 | function square(number) {  
2 |     return number * number;  
3 | }
```

The function `square` takes one parameter, called `number`. The function consists of one statement that says to return the parameter of the function (that is, `number`) multiplied by itself. The statement `return` specifies the value returned by the function.

```
1 | return number * number;
```

Primitive parameters (such as a number) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.

If you pass an object (i.e. a non-primitive value, such as `Array` or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
1 | function myFunc(theObject) {  
2 |     theObject.make = 'Toyota';  
3 | }  
4 |  
5 | var mycar = {make: 'Honda', model: 'Accord', year: 1998};  
6 | var x, y;  
7 |  
8 | x = mycar.make; // x gets the value "Honda"  
9 |  
10 | myFunc(mycar);  
11 | y = mycar.make; // y gets the value "Toyota"  
12 |           // (the make property was changed by the function)
```

Function expressions 

While the function declaration above is syntactically a statement, functions can also be created by a function expression. Such a function can be **anonymous**; it does not have to have a name. For example, the function `square` could have been defined as:

```
1 | var square = function(number) { return number * number; };
2 | var x = square(4); // x gets the value 16
```

However, a name can be provided with a function expression and can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces:

```
1 | var factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1); };
2 |
3 | console.log(factorial(3));
```

Function expressions are convenient when passing a function as an argument to another function. The following example shows a `map` function that should receive a function as first argument and an array as second argument.

```
1 | function map(f, a) {
2 |     var result = [], i; // Create a new Array
3 |     for (i = 0; i !== a.length; i++)
4 |         result[i] = f(a[i]);
5 |     return result;
6 | }
```

In the following code our function receives a function defined by a function expression and executes it for every element of the array received as a second argument.

```
1 | function map(f, a) {
2 |     var result = []; // Create a new Array
3 |     var i; // Declare variable
4 |     for (i = 0; i !== a.length; i++)
5 |         result[i] = f(a[i]);
6 |     return result;
7 | }
8 | var f = function(x) {
9 |     return x * x * x;
10 | }
    var numbers = [0, 1, 2, 5, 10];
```

```
11 | var cube = map(f,numbers);  
12 | console.log(cube);  
13 |
```

Function returns: [0, 1, 8, 125, 1000].

In JavaScript, a function can be defined based on a condition. For example, the following function definition defines `myFunc` only if `num` equals 0:

```
1 | var myFunc;  
2 | if (num === 0) {  
3 |     myFunc = function(theObject) {  
4 |         theObject.make = 'Toyota';  
5 |     }  
6 | }
```

In addition to defining functions as described here, you can also use the `Function` constructor to create functions from a string at runtime, much like `eval()`.

A **method** is a function that is a property of an object. Read more about objects and methods in [Working with objects](#).

---

## Calling functions [↗](#)

Defining a function does not execute it. Defining the function simply names the function and specifies what to do when the function is called. **Calling** the function actually performs the specified actions with the indicated parameters. For example, if you define the function `square`, you could call it as follows:

```
1 | square(5);
```

The preceding statement calls the function with an argument of 5. The function executes its statements and returns the value 25.

Functions must be in scope when they are called, but the function declaration can be hoisted (appear below the call in the code), as in this example:

```
1 | console.log(square(5));
2 | /* ... */
3 | function square(n) { return n * n; }
```

The scope of a function is the function in which it is declared, or the entire program if it is declared at the top level.

**Note:** This works only when defining the function using the above syntax (i.e. `function funcName() {}`). The code below will not work. That means, function hoisting only works with function declaration and not with function expression.

```
1 | console.log(square); // square is hoisted with an initial value undefined.
2 | console.log(square(5)); // TypeError: square is not a function
3 | var square = function(n) {
4 |     return n * n;
5 | }
```

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function. The `show_props()` function (defined in *Working with objects*) is an example of a function that takes an object as an argument.

A function can call itself. For example, here is a function that computes factorials recursively:

```
1 | function factorial(n) {
2 |     if ((n === 0) || (n === 1))
3 |         return 1;
4 |     else
5 |         return (n * factorial(n - 1));
6 | }
```

You could then compute the factorials of one through five as follows:

```
1 | var a, b, c, d, e;
2 | a = factorial(1); // a gets the value 1
3 | b = factorial(2); // b gets the value 2
4 | c = factorial(3); // c gets the value 6
5 | d = factorial(4); // d gets the value 24
6 | e = factorial(5); // e gets the value 120
```

There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime. It turns out that functions are, themselves, objects, and these objects in turn have methods (see the [Function object](#)). One of these, the `apply()` method, can be used to achieve this goal.

---

## Function scope [🔗](#)

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined. In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access.

```
1 // The following variables are defined in the global scope
2 var num1 = 20,
3     num2 = 3,
4     name = 'Chamahk';
5
6 // This function is defined in the global scope
7 function multiply() {
8     return num1 * num2;
9 }
10
11 multiply(); // Returns 60
12
13 // A nested function example
14 function getScore() {
15     var num1 = 2,
16         num2 = 3;
17
18     function add() {
19         return name + ' scored ' + (num1 + num2);
20     }
21
22     return add();
23 }
```

```
24 | getScore(); // Returns "Chamahk scored 5"  
25 |
```

---

## Scope and the function stack [↗](#)

### Recursion [↗](#)

A function can refer to and call itself. There are three ways for a function to refer to itself:

1. the function's name
2. `arguments.callee`
3. an in-scope variable that refers to the function

For example, consider the following function definition:

```
1 | var foo = function bar() {  
2 |     // statements go here  
3 | };
```

Within the function body, the following are all equivalent:

1. `bar()`
2. `arguments.callee()`
3. `foo()`

A function that calls itself is called a *recursive function*. In some ways, recursion is analogous to a loop. Both execute the same code multiple times, and both require a condition (to avoid an infinite loop, or rather, infinite recursion in this case). For example, the following loop:

```
1 | var x = 0;  
2 | while (x < 10) { // "x < 10" is the loop condition  
3 |     // do stuff  
4 |     x++;  
5 | }
```

can be converted into a recursive function and a call to that function:

```
1 function loop(x) {  
2   if (x >= 10) // "x >= 10" is the exit condition (equivalent to "!(x < 10)")  
3     return;  
4   // do stuff  
5   loop(x + 1); // the recursive call  
6 }  
7 loop(0);
```

However, some algorithms cannot be simple iterative loops. For example, getting all the nodes of a tree structure (e.g. the DOM) is more easily done using recursion:

```
1 function walkTree(node) {  
2   if (node == null) //  
3     return;  
4   // do something with node  
5   for (var i = 0; i < node.childNodes.length; i++) {  
6     walkTree(node.childNodes[i]);  
7   }  
8 }
```

Compared to the function `loop`, each recursive call itself makes many recursive calls here.

It is possible to convert any recursive algorithm to a non-recursive one, but often the logic is much more complex and doing so requires the use of a stack. In fact, recursion itself uses a stack: the function stack.

The stack-like behavior can be seen in the following example:

```
1 function foo(i) {  
2   if (i < 0)  
3     return;  
4   console.log('begin: ' + i);  
5   foo(i - 1);  
6   console.log('end: ' + i);  
7 }  
8 foo(3);  
9  
10 // Output:  
11  
12 // begin: 3
```



```
13 // begin: 2
14 // begin: 1
15 // begin: 0
16 // end: 0
17 // end: 1
18 // end: 2
19 // end: 3
```

## Nested functions and closures

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function. It also forms a *closure*. A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.

To summarize:

- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

```
1 function addSquares(a, b) {
2   function square(x) {
3     return x * x;
4   }
5   return square(a) + square(b);
6 }
7 a = addSquares(2, 3); // returns 13
8 b = addSquares(3, 4); // returns 25
9 c = addSquares(4, 5); // returns 41
```

Since the inner function forms a closure, you can call the outer function and specify arguments for both the outer and inner function:

```
1  function outside(x) {  
2      function inside(y) {  
3          return x + y;  
4      }  
5      return inside;  
6  }  
7  fn_inside = outside(3); // Think of it like: give me a function that adds 3 to wha  
8                          // it  
9  result = fn_inside(5); // returns 8  
10  
11 result1 = outside(3)(5); // returns 8
```

## Preservation of variables [↗](#)

Notice how `x` is preserved when `inside` is returned. A closure must preserve the arguments and variables in all scopes it references. Since each call provides potentially different arguments, a new closure is created for each call to `outside`. The memory can be freed only when the returned `inside` is no longer accessible.

This is not different from storing references in other objects, but is often less obvious because one does not set the references directly and cannot inspect them.

## Multiply-nested functions [↗](#)

Functions can be multiply-nested, i.e. a function (A) containing a function (B) containing a function (C). Both functions B and C form closures here, so B can access A and C can access B. In addition, since C can access B which can access A, C can also access A. Thus, the closures can contain multiple scopes; they recursively contain the scope of the functions containing it. This is called *scope chaining*. (Why it is called "chaining" will be explained later.)

Consider the following example:

```
1  function A(x) {  
2      function B(y) {  
3          function C(z) {  
4              console.log(x + y + z);  
5          }  
6          C(3);  
7      }  
8      B(2);
```

```
9 | }  
10 | A(1); // logs 6 (1 + 2 + 3)
```

In this example, C accesses B's `y` and A's `x`. This can be done because:

1. B forms a closure including A, i.e. B can access A's arguments and variables.
2. C forms a closure including B.
3. Because B's closure includes A, C's closure includes A, C can access both B *and* A's arguments and variables. In other words, C *chains* the scopes of B and A in that order.

The reverse, however, is not true. A cannot access C, because A cannot access any argument or variable of B, which C is a variable of. Thus, C remains private to only B.

## Name conflicts

When two arguments or variables in the scopes of a closure have the same name, there is a *name conflict*. More inner scopes take precedence, so the inner-most scope takes the highest precedence, while the outer-most scope takes the lowest. This is the scope chain. The first on the chain is the inner-most scope, and the last is the outer-most scope. Consider the following:

```
1 | function outside() {  
2 |   var x = 5;  
3 |   function inside(x) {  
4 |     return x * 2;  
5 |   }  
6 |   return inside;  
7 | }  
8 |  
9 | outside()(10); // returns 20 instead of 10
```

The name conflict happens at the statement `return x` and is between `inside`'s parameter `x` and `outside`'s variable `x`. The scope chain here is `{inside, outside, global object}`. Therefore `inside`'s `x` takes precedences over `outside`'s `x`, and 20 (`inside`'s `x`) is returned instead of 10 (`outside`'s `x`).

---

## Closures

Closures are one of the most powerful features of JavaScript. JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to). However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function. Also, since the inner function has access to the scope of the outer function, the variables and functions defined in the outer function will live longer than the duration of the outer function execution, if the inner function manages to survive beyond the life of the outer function. A closure is created when the inner function is somehow made available to any scope outside the outer function.

```
1  var pet = function(name) {    // The outer function defines a variable called "name"
2      var getName = function() {
3          return name;          // The inner function has access to the "name" variab
4                                  //function
5      }
6      return getName;           // Return the inner function, thereby exposing it to
7  }
8  myPet = pet('Vivie');
9
10 myPet();                      // Returns "Vivie"
```

It can be much more complex than the code above. An object containing methods for manipulating the inner variables of the outer function can be returned.

```
1  var createPet = function(name) {
2      var sex;
3
4      return {
5
6          setName: function(newName) {
7              name = newName;
8          },
9
10         getName: function() {
11             return name;
12         },
13
14         getSex: function() {
15             return sex;
16         },
17     };
18 }
```

```

16
17     setSex: function(newSex) {
18         if(typeof newSex === 'string' && (newSex.toLowerCase() === 'male' ||
19             newSex.toLowerCase() === 'female')) {
20             sex = newSex;
21         }
22     }
23 }
24 }
25
26 var pet = createPet('Vivie');
27 pet.getName();           // Vivie
28
29 pet.setName('Oliver');
30 pet.setSex('male');
31 pet.getSex();            // male
32 pet.getName();          // Oliver

```

In the code above, the `name` variable of the outer function is accessible to the inner functions, and there is no other way to access the inner variables except through the inner functions. The inner variables of the inner functions act as safe stores for the outer arguments and variables. They hold "persistent" and "encapsulated" data for the inner functions to work with. The functions do not even have to be assigned to a variable, or have a name.

```

1  var getCode = (function() {
2      var apiCode = '0]Ea1(eh&2';    // A code we do not want outsiders to be able to
3
4      return function() {
5          return apiCode;
6      };
7  })();
8
9  getCode();    // Returns the apiCode

```

There are, however, a number of pitfalls to watch out for when using closures. If an enclosed function defines a variable with the same name as the name of a variable in the outer scope, there is no way to refer to the variable in the outer scope again.

```

1  var createPet = function(name) { // The outer function defines a variable called
2      return {

```

```
3      setName: function(name) {    // The enclosed function also defines a variable
4          name = name;              // How do we access the "name" defined by the out
5      }
6  }
7  }
```

## Using the arguments object [↗](#)

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

```
1  arguments[i]
```

where `i` is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be `arguments[0]`. The total number of arguments is indicated by `arguments.length`.

Using the `arguments` object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then access each argument using the `arguments` object.

For example, consider a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

```
1  function myConcat(separator) {
2      var result = ''; // initialize list
3      var i;
4      // iterate through arguments
5      for (i = 1; i < arguments.length; i++) {
6          result += arguments[i] + separator;
7      }
8      return result;
9  }
```



You can pass any number of arguments to this function, and it concatenates each argument into a string "list":

```
1 // returns "red, orange, blue, "  
2 myConcat(',', 'red', 'orange', 'blue');  
3  
4 // returns "elephant; giraffe; lion; cheetah; "  
5 myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');  
6  
7 // returns "sage. basil. oregano. pepper. parsley. "  
8 myConcat('. ', 'sage', 'basil', 'oregano', 'pepper', 'parsley');
```

**Note:** The `arguments` variable is "array-like", but not an array. It is array-like in that it has a numbered index and a `length` property. However, it does not possess all of the array-manipulation methods.

See the `Function` object in the JavaScript reference for more information.

---

## Function parameters

Starting with ECMAScript 2015, there are two new kinds of parameters: default parameters and rest parameters.

### Default parameters

In JavaScript, parameters of functions default to `undefined`. However, in some situations it might be useful to set a different default value. This is where default parameters can help.

In the past, the general strategy for setting defaults was to test parameter values in the body of the function and assign a value if they are `undefined`. If in the following example, no value is provided for `b` in the call, its value would be `undefined` when evaluating `a*b` and the call to `multiply` would have returned `NaN`. However, this is caught with the second line in this example:

```
1 function multiply(a, b) {  
2   b = typeof b !== 'undefined' ? b : 1;  
3  
4   return a * b;  
5 }  
6  
7 multiply(5); // 5
```

With default parameters, the check in the function body is no longer necessary. Now, you can simply put 1 as the default value for b in the function head:

```
1 function multiply(a, b = 1) {  
2   return a * b;  
3 }  
4  
5 multiply(5); // 5
```

For more details, see default parameters in the reference.

## Rest parameters [↗](#)

The rest parameter syntax allows us to represent an indefinite number of arguments as an array. In the example, we use the rest parameters to collect arguments from the second one to the end. We then multiply them by the first one. This example is using an arrow function, which is introduced in the next section.

```
1 function multiply(multiplier, ...theArgs) {  
2   return theArgs.map(x => multiplier * x);  
3 }  
4  
5 var arr = multiply(2, 1, 2, 3);  
6 console.log(arr); // [2, 4, 6]
```

## Arrow functions [↗](#)



An arrow function expression (previously, and now incorrectly known as **fat arrow function**) has a shorter syntax compared to function expressions and does not have its own `this`, `arguments`, `super`, or `new.target`. Arrow functions are always anonymous. See also this [hacks.mozilla.org](https://hacks.mozilla.org/2015/12/es6-in-depth-arrow-functions/) blog post: " ES6 In Depth: Arrow functions".

Two factors influenced the introduction of arrow functions: shorter functions and non-binding of `this`.

## Shorter functions [↗](#)

In some functional patterns, shorter functions are welcome. Compare:

```
1  var a = [  
2    'Hydrogen',  
3    'Helium',  
4    'Lithium',  
5    'Beryllium'  
6  ];  
7  
8  var a2 = a.map(function(s) { return s.length; });  
9  
10 console.log(a2); // logs [8, 6, 7, 9]  
11  
12 var a3 = a.map(s => s.length);  
13  
14 console.log(a3); // logs [8, 6, 7, 9]
```

## No separate this [↗](#)

Until arrow functions, every new function defined its own `this` value (a new object in the case of a constructor, undefined in strict mode function calls, the base object if the function is called as an "object method", etc.). This proved to be less than ideal with an object-oriented style of programming.

```
1  function Person() {  
2    // The Person() constructor defines `this` as itself.  
3    this.age = 0;  
4  
5    setInterval(function growUp() {  
6      // In nonstrict mode, the growUp() function defines `this`  
7      // as the global object, which is different from the `this`
```

```
8      // defined by the Person() constructor.
9      this.age++;
10     }, 1000);
11 }
12
13 var p = new Person();
```

In ECMAScript 3/5, this issue was fixed by assigning the value in `this` to a variable that could be closed over.

```
1 function Person() {
2     var self = this; // Some choose `that` instead of `self`.
3                       // Choose one and be consistent.
4     self.age = 0;
5
6     setInterval(function growUp() {
7         // The callback refers to the `self` variable of which
8         // the value is the expected object.
9         self.age++;
10     }, 1000);
11 }
```

Alternatively, a bound function could be created so that the proper `this` value would be passed to the `growUp()` function.

An arrow function does not have its own `this`; the `this` value of the enclosing execution context is used. Thus, in the following code, the `this` within the function that is passed to `setInterval` has the same value as `this` in the enclosing function:

```
1 function Person() {
2     this.age = 0;
3
4     setInterval(() => {
5         this.age++; // |this| properly refers to the person object
6     }, 1000);
7 }
8
9 var p = new Person();
```

# Predefined functions

JavaScript has several top-level, built-in functions:

## **eval()**

The **eval()** method evaluates JavaScript code represented as a string.

## **uneval()**

The **uneval()** method creates a string representation of the source code of an `Object`.

## **isFinite()**

The global **isFinite()** function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.

## **isNaN()**

The **isNaN()** function determines whether a value is `NaN` or not. Note: coercion inside the `isNaN` function has interesting rules; you may alternatively want to use `Number.isNaN()`, as defined in ECMAScript 2015, or you can use `typeof` to determine if the value is Not-A-Number.

## **parseFloat()**

The **parseFloat()** function parses a string argument and returns a floating point number.

## **parseInt()**

The **parseInt()** function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).

## **decodeURI()**

The **decodeURI()** function decodes a Uniform Resource Identifier (URI) previously created by `encodeURIComponent` or by a similar routine.

## **decodeURIComponent()**

The **decodeURIComponent()** method decodes a Uniform Resource Identifier (URI) component previously created by `encodeURIComponent` or by a similar routine.

## **encodeURIComponent()**

The **encodeURIComponent()** method encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

## **encodeURIComponent ( )**

The **encodeURIComponent ( )** method encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

## **escape ( )**

The deprecated **escape ( )** method computes a new string in which certain characters have been replaced by a hexadecimal escape sequence. Use `encodeURIComponent` or `encodeURIComponent` instead.

## **unescape ( )**

The deprecated **unescape ( )** method computes a new string in which hexadecimal escape sequences are replaced with the character that it represents. The escape sequences might be introduced by a function like `escape`. Because `unescape ( )` is deprecated, use `decodeURI ( )` or `decodeURIComponent` instead.

« Previous

Next »