RELATED TOPICS

JavaScript

Tutorials:

▶ Complete beginners

▼ JavaScript Guide

- Introduction
- Grammar and types
- Control flow and error handling
- Loops and iteration
- Functions
- Expressions and operators
- Numbers and dates
- Text formatting
- Regular expressions
- Indexed collections
- Keyed collections
- Working with objects
- Details of the object model
- Using promises
- Iterators and generators
- Meta programming
- JavaScript modules

▶ Intermediate

▶ Advanced

References:

▶ Built-in objects

▶ Expressions & operators

▶ Statements & declarations

▼ Functions

- The arguments object
- Arrow function expressions
- Default parameters
- getter
- Method definitions
- Rest parameters
- setter

▶ Classes

▶ Errors

▶ Misc

Functions

« Previous Next »

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output. 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`:

```
function square(number) {
  return number * number;
}
```

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:

```
return number * number;
```

Parameters are essentially passed to functions **by value** — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, **the change is not reflected globally or in the code which called that function**.

When you pass an object as a parameter, if the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}

var mycar = {make: 'Honda', model: 'Accord', year: 1998};
var x, y;

x = mycar.make; // x gets the value "Honda"

myFunc(mycar);
y = mycar.make; // y gets the value "Toyota"
                // (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:

```
const square = function(number) { return number * number }
var x = square(4) // x gets the value 16
```

However, a name *can* be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

```
const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n
- 1) }

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:

```
function map(f, a) {
  let result = []; // Create a new Array
  let i; // Declare variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
```

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

```
function map(f, a) {
  let result = []; // Create a new Array
  let i; // Declare variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
const f = function(x) {
   return x * x * x;
}
let numbers = [0, 1, 2, 5, 10];
let cube = map(f,numbers);
console.log(cube);
```

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`:

```
var myFunc;
if (num === 0) {
  myFunc = function(theObject) {
    theObject.make = 'Toyota';
  }
}
```

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 it 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:

```
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:

```
console.log(square(5));
/* ... */
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.
>
> This means that function hoisting only works with function *declarations*—not with function *expressions*.
>
> ```
> console.log(square)    // square is hoisted with an initial value
> undefined.
> console.log(square(5)) // Uncaught TypeError: square is not a function
> const square = function(n) {
>   return n * n;
> }
> ```

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function. The `showProps()` 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:

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

You could then compute the factorials of `1` through `5` as follows:

```
var a, b, c, d, e;
a = factorial(1); // a gets the value 1
b = factorial(2); // b gets the value 2
c = factorial(3); // c gets the value 6
d = factorial(4); // d gets the value 24
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 in turn, these objects 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 variables to which the parent function has access.

```
// The following variables are defined in the global scope
var num1 = 20,
    num2 = 3,
    name = 'Chamakh';

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
}

multiply(); // Returns 60

// A nested function example
function getScore() {
  var num1 = 2,
      num2 = 3;

  function add() {
    return name + ' scored ' + (num1 + num2);
  }

  return add();
}

getScore(); // Returns "Chamakh scored 5"
```

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:

```
var foo = function bar() {
   // statements go here
}
```

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...

```
var x = 0;
while (x < 10) { // "x < 10" is the loop condition
   // do stuff
   x++;
}
```

...can be converted into a recursive function declaration, followed by a call to that function:

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

However, some algorithms cannot be simple iterative loops. For example, getting all the nodes of a tree structure (such as the DOM) is easier via recursion:

```
function walkTree(node) {
  if (node == null) //
    return;
  // do something with node
  for (var i = 0; i < node.childNodes.length; i++) {
```

```
    walkTree(node.childNodes[i]);
  }
}
```

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 the logic is often 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:

```
function foo(i) {
  if (i < 0)
    return;
  console.log('begin: ' + i);
  foo(i - 1);
  console.log('end: ' + i);
}
foo(3);

// Output:

// begin: 3
// begin: 2
// begin: 1
// begin: 0
// end: 0
// end: 1
// end: 2
// end: 3
```

Nested functions and closures

You may nest a function within another function. The nested (inner) function is private to its containing (outer) function.

It also forms a *closure*. A closure is an expression (most commonly, 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:

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
a = addSquares(2, 3); // returns 13
b = addSquares(3, 4); // returns 25
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:

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
fn_inside = outside(3); // Think of it like: give me a function
that adds 3 to whatever you give
                        // it
result = fn_inside(5); // returns 8

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. For example:

- A function (`A`) contains a function (`B`), which itself contains 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*. (The reason it is called "chaining" is explained later.)

Consider the following example:

```
function A(x) {
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
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 nested 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:

```
function outside() {
  var x = 5;
  function inside(x) {
    return x * 2;
  }
  return inside;
}

outside()(10); // returns 20 instead of 10
```

The name conflict happens at the statement `return x * 2` 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.

```
var pet = function(name) {    // The outer function defines a
variable called "name"
  var getName = function() {
    return name;              // The inner function has access to
the "name" variable of the outer
                              //function
  }
  return getName;             // Return the inner function,
thereby exposing it to outer scopes
}
myPet = pet('Vivie');

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.

```
var createPet = function(name) {
  var sex;

  return {
    setName: function(newName) {
      name = newName;
    },

    getName: function() {
      return name;
    },

    getSex: function() {
      return sex;
    },

    setSex: function(newSex) {
      if(typeof newSex === 'string' && (newSex.toLowerCase() ===
'male' ||
        newSex.toLowerCase() === 'female')) {
        sex = newSex;
      }
    }
  }
}
```

```
var pet = createPet('Vivie');
pet.getName();                  // Vivie

pet.setName('Oliver');
pet.setSex('male');
pet.getSex();                   // male
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.

```
var getCode = (function() {
  var apiCode = '0]Eal(eh&2';    // A code we do not want
outsiders to be able to modify...

  return function() {
    return apiCode;
  };
})();

getCode();    // Returns the apiCode
```

> **Note:** There are a number of pitfalls to watch out for when using closures!
>
> If an enclosed function defines a variable with the same name as a variable in the outer scope, then there is no way to refer to the variable in the outer scope again. (The inner scope variable "overrides" the outer one, until the program exits the inner scope.)

```
var createPet = function(name) {  // The outer function defines a variable
called "name".
  return {
    setName: function(name) {    // The enclosed function also defines a
variable called "name".
      name = name;               // How do we access the "name" defined by
the outer function?
    }
  }
}
```

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:

```
arguments[i]
```

where `i` is the ordinal number of the argument, starting at `0`. 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:

```
function myConcat(separator) {
   var result = ''; // initialize list
   var i;
   // iterate through arguments
   for (i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
   }
   return result;
}
```

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

```
// returns "red, orange, blue, "
myConcat(', ', 'red', 'orange', 'blue');

// returns "elephant; giraffe; lion; cheetah; "
myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');

// returns "sage. basil. oregano. pepper. parsley. "
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 exactly what default parameters do.

Without default parameters (pre-ECMAScript 2015)

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`.

In the following example, if no value is provided for `b`, its value would be `undefined` when evaluating `a*b`, and a call to `multiply` would normally have returned `NaN`. However, this is prevented by the second line in this example:

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a * b;
}

multiply(5); // 5
```

With default parameters (post-ECMAScript 2015)

With *default parameters*, a manual check in the function body is no longer necessary. You can put `1` as the default value for `b` in the function head:

```
function multiply(a, b = 1) {
  return a * b;
}

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 following example, the function `multiply` uses *rest parameters* to collect arguments from the second one to the end. The function then multiplies these by the first argument.

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(x => multiplier * x);
}

var arr = multiply(2, 1, 2, 3);
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 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:

```
var a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

var a2 = a.map(function(s) { return s.length; });

console.log(a2); // logs [8, 6, 7, 9]

var a3 = a.map(s => s.length);

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.

```
function Person() {
  // The Person() constructor defines `this` as itself.
  this.age = 0;

  setInterval(function growUp() {
    // In nonstrict mode, the growUp() function defines `this`
    // as the global object, which is different from the `this`
    // defined by the Person() constructor.
    this.age++;
  }, 1000);
}
```

```
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.

```
function Person() {
  var self = this; // Some choose `that` instead of `self`.
                   // Choose one and be consistent.
  self.age = 0;

  setInterval(function growUp() {
    // The callback refers to the `self` variable of which
    // the value is the expected object.
    self.age++;
  }, 1000);
}
```

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:

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the person object
  }, 1000);
}

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 `encodeURI` 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.

`encodeURI()`

The **`encodeURI()`** 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 `encodeURI` 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 »

Found a problem with this page?

- Edit on **GitHub**
- Source on **GitHub**
- Report a problem with this content on **GitHub**
- Want to fix the problem yourself? See our Contribution guide.

Last modified: Feb 2, 2022, by MDN contributors