Functions

Generally speaking, a function is a "subprogram" that can be *called* by code external (or internal in the case of recursion) to the function. Like the program itself, a function is composed of a sequence of statements called the *function body*. Values can be *passed* to a function, and the function will *return* a value.

In JavaScript, functions are first-class objects, because they can have properties and methods just like any other object. What distinguishes them from other objects is that functions can be called. In brief, they are `Function` objects.

For more examples and explanations, see also the JavaScript guide about functions.

Every function in JavaScript is a `Function` object. See `Function` for information on properties and methods of `Function` objects.

To return a value other than the default, a function must have a `return` statement that specifies the value to return. A function without a return statement will return a default value. In the case of a constructor called with the `new` keyword, the default value is the value of its `this` parameter. For all other functions, the default return value is `undefined`.

The parameters of a function call are the function's *arguments*. Arguments are passed to functions *by value*. If the function changes the value of an argument, this change is not reflected globally or in the calling function. However, object references are values, too, and they are special: if the function changes the referred object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
   theObject.brand = "Toyota";
 }


 var mycar = {
   brand: "Honda",
   model: "Accord",
   year: 1998
 };


 console.log(mycar.brand);


 myFunc(mycar);


 console.log(mycar.brand);
```

The `this` keyword does not refer to the currently executing function, so you must refer to `Function` objects by name, even within the function body.

Defining functions

There are several ways to define functions:

The function declaration (`function` statement)

There is a special syntax for declaring functions (see function statement for details):

```
function name([param[, param[, ... param]]]) {
   statements
}
```

`name`

The function name.

`param`

The name of an argument to be passed to the function. A function can have up to 255 arguments.

`statements`

The statements comprising the body of the function.

The function expression (`function` expression)

A function expression is similar to and has the same syntax as a function declaration (see function expression for details). A function expression may be a part of a larger expression. One can define "named" function expressions (where the name of the expression might be used in the call stack for example) or "anonymous" function expressions. Function expressions are not *hoisted* onto the beginning of the scope, therefore they cannot be used before they appear in the code.

```
function [name]([param[, param[, ... param]]]) {
   statements
}
```

`name`

The function name. Can be omitted, in which case the function becomes known as an anonymous function.

`param`

The name of an argument to be passed to the function. A function can have up to 255 arguments.

`statements`

The statements comprising the body of the function.

Here is an example of an **anonymous** function expression (the `name` is not used):

```
var myFunction = function() {
    statements
}
```

It is also possible to provide a name inside the definition in order to create a **named** function expression:

```
var myFunction = function namedFunction(){
    statements
}
```

One of the benefit of creating a named function expression is that in case we encounted an error, the stack trace will contain the name of the function, making it easier to find the origin of the error.

As we can see, both examples do not start with the `function` keyword. Statements involving functions which do not start with `function` are function expressions.

When functions are used only once, a common pattern is an **IIFE (*Immediately Invokable Function Expression*)**.

```
(function() {
    statements
})();
```

IIFE are function expressions that are invoked as soon as the function is declared.

The generator function declaration (`function*` statement)

There is a special syntax for generator function declarations (see [`function* statement`]() for details):

```
function* name([param[, param[, ... param]]]) {
   statements
}
```

`name`
The function name.

`param`
The name of an argument to be passed to the function. A function can have up to 255 arguments.

`statements`
The statements comprising the body of the function.

The generator function expression (`function*` expression)

A generator function expression is similar to and has the same syntax as a generator function declaration (see [`function* expression`]() for details):

```
function* [name]([param[, param[, ... param]]]) {
   statements
}
```

`name`
The function name. Can be omitted, in which case the function becomes known as an anonymous function.

`param`

The name of an argument to be passed to the function. A function can have up to 255 arguments.

`statements`

The statements comprising the body of the function.

The arrow function expression (=>)

An arrow function expression has a shorter syntax and lexically binds its `this` value (see arrow functions for details):

```
([param[, param]]) => {
   statements
}

param => expression
```

`param`

The name of an argument. Zero arguments need to be indicated with `()`. For only one argument, the parentheses are not required. (like `foo => 1`)

`statements or expression`

Multiple statements need to be enclosed in brackets. A single expression requires no brackets. The expression is also the implicit return value of the function.

The `Function` constructor

Note: Using the `Function` constructor to create functions is not recommended since it needs the function body as a string which may prevent some JS engine optimizations and can also cause other problems.

As all other objects, `Function` objects can be created using the `new` operator:

```
new Function (arg1, arg2, ... argN, functionBody)
```

`arg1, arg2, ... argN`

Zero or more names to be used by the function as formal parameters. Each must be a proper JavaScript identifier.

`functionBody`

A string containing the JavaScript statements comprising the function body.

Invoking the `Function` constructor as a function (without using the `new` operator) has the same effect as invoking it as a constructor.

The `GeneratorFunction` constructor

Note: `GeneratorFunction` is not a global object, but could be obtained from generator function instance (see `GeneratorFunction` for more detail).

Note: Using the `GeneratorFunction` constructor to create functions is not recommended since it needs the function body as a string which may prevent some JS engine optimizations and can also cause other problems.

As all other objects, `GeneratorFunction` objects can be created using the `new` operator:

```
new GeneratorFunction (arg1, arg2, ... argN, functionBody)
```

`arg1, arg2, ... argN`

Zero or more names to be used by the function as formal argument names. Each must be a string that conforms to the rules for a valid JavaScript identifier or a list of such strings separated with a comma; for example "`x`", "`theValue`", or "`a,b`".

`functionBody`

A string containing the JavaScript statements comprising the function definition.

Invoking the `Function` constructor as a function (without using the `new` operator) has the same effect as invoking it as a constructor.

Function parameters

Default parameters

Default function parameters allow formal parameters to be initialized with default values if no value or `undefined` is passed. For more details, see default parameters.

Rest parameters

The rest parameter syntax allows to represent an indefinite number of arguments as an array. For more details, see rest parameters.

The `arguments` object

You can refer to a function's arguments within the function by using the `arguments` object. See arguments.

- `arguments`: An array-like object containing the arguments passed to the currently executing function.
- `arguments.callee` : The currently executing function.
- `arguments.caller` : The function that invoked the currently executing function.
- `arguments.length`: The number of arguments passed to the function.

Defining method functions

Getter and setter functions

You can define getters (accessor methods) and setters (mutator methods) on any standard built-in object or user-defined object that supports the addition of new properties. The syntax for defining getters and setters uses the object literal syntax.

get

Binds an object property to a function that will be called when that property is looked up.

set

Binds an object property to a function to be called when there is an attempt to set that property.

Method definition syntax

Starting with ECMAScript 2015, you are able to define own methods in a shorter syntax, similar to the getters and setters. See method definitions for more information.

```
var obj = {
  foo() {},
  bar() {}
};
```

`Function` constructor vs. function declaration vs. function expression

Compare the following:

A function defined with the `Function` constructor assigned to the variable `multiply`:

```
var multiply = new Function('x', 'y', 'return x * y');
```

A *function declaration* of a function named `multiply`:

```
function multiply(x, y) {
   return x * y;
}
```

A *function expression* of an anonymous function assigned to the variable `multiply`:

```
var multiply = function(x, y) {
   return x * y;
};
```

A *function expression* of a function named `func_name` assigned to the variable `multiply`:

```
var multiply = function func_name(x, y) {
   return x * y;
};
```

Differences

All do approximately the same thing, with a few subtle differences:

There is a distinction between the function name and the variable the function is assigned to. The function name cannot be changed, while the variable the function is assigned to can be reassigned. The function name can be used only within the function's body. Attempting to use it outside the function's body results in an error (or `undefined` if the function name was previously declared via a `var` statement). For example:

```
var y = function x() {};
alert(x);
```

The function name also appears when the function is serialized via [`Function`'s toString method](#).

On the other hand, the variable the function is assigned to is limited only by its scope, which is guaranteed to include the scope in which the function is declared.

As the 4th example shows, the function name can be different from the variable the function is assigned to. They have no relation to each other. A function declaration also creates a variable with the same name as the function name. Thus, unlike those defined by function expressions, functions defined by function declarations can be accessed by their name in the scope they were defined in:

A function defined by '`new Function`' does not have a function name. However, in the [SpiderMonkey](#) JavaScript engine, the serialized form of the function shows as if it has the name "anonymous". For example, `alert(new Function())` outputs:

```
function anonymous() {
}
```

Since the function actually does not have a name, `anonymous` is not a variable that can be accessed within the function. For example, the following would result in an error:

```
var foo = new Function("alert(anonymous);");
foo();
```

Unlike functions defined by function expressions or by the `Function` constructor, a function defined by a function declaration can be used before the function declaration itself. For example:

```
foo();
function foo() {
   alert('FOO!');
}
```

A function defined by a function expression or by a function declaration inherits the current scope. That is, the function forms a closure. On the other hand, a function defined by a `Function` constructor does not inherit any scope other than the global scope (which all functions inherit).

```
var p = 5;
function myFunc() {
    var p = 9;

    function decl() {
        console.log(p);
    }
    var expr = function() {
        console.log(p);
    };
    var cons = new Function('\tconsole.log(p);');

    decl();
    expr();
    cons();
}
myFunc();
```

Functions defined by function expressions and function declarations are parsed only once, while those defined by the `Function` constructor are not. That is, the function body string passed to the `Function` constructor must be parsed each and every time the constructor is called. Although a function expression creates a closure every time, the function body is not reparsed, so function expressions are still faster than "`new Function(...)`". Therefore the `Function` constructor should generally be avoided whenever possible.

It should be noted, however, that function expressions and function declarations nested within the function generated by parsing a `Function constructor` 's string aren't parsed repeatedly. For example:

```
var foo = (new Function("var bar = \'FOO!\';\nreturn(function() {\n\talert(bar
foo();
```

A function declaration is very easily (and often unintentionally) turned into a function expression. A function declaration ceases to be one when it either:

- becomes part of an expression
- is no longer a "source element" of a function or the script itself. A "source element" is a non-nested statement in the script or a function body:

```
var x = 0;
if (x === 0) {
   x = 10;
   function boo() {}
}
function foo() {
   var y = 20;
   function bar() {}
   while (y === 10) {
      function blah() {}
      y++;
   }
}
```

Examples

```
function foo() {}


(function bar() {})


x = function hello() {}


if (x) {

   function world() {}
}



function a() {

   function b() {}
   if (0) {

      function c() {}
   }
}
```

Block-level functions

In strict mode, starting with ES2015, functions inside blocks are now scoped to that block. Prior to ES2015, block-level functions were forbidden in strict mode.

```
'use strict';
```

```
function f() {
  return 1;
}

{
  function f() {
    return 2;
  }
}

f() === 1;
```

Block-level functions in non-strict code

In a word: Don't.

In non-strict code, function declarations inside blocks behave strangely. For example:

```
if (shouldDefineZero) {
   function zero() {
      console.log("This is zero.");
   }
}
```

ES2015 says that if `shouldDefineZero` is false, then `zero` should never be defined, since the block never executes. However, it's a new part of the standard. Historically, this was left unspecified, and some browsers would define `zero` whether the block executed or not.

In strict mode, all browsers that support ES2015 handle this the same way: `zero` is defined only if `shouldDefineZero` is true, and only in the scope of the `if`-block.

A safer way to define functions conditionally is to assign a function expression to a variable:

```
var zero;
if (shouldDefineZero) {
   zero = function() {
      console.log("This is zero.");
   };
}
```

Examples

Returning a formatted number

The following function returns a string containing the formatted representation of a number padded with leading zeros.

```
function padZeros(num, totalLen) {
   var numStr = num.toString();
   var numZeros = totalLen - numStr.length;
   for (var i = 1; i <= numZeros; i++) {
      numStr = "0" + numStr;
   }
   return numStr;
}
```

The following statements call the padZeros function.

```
var result;
result = padZeros(42,4);
result = padZeros(42,2);
result = padZeros(5,4);
```

Determining whether a function exists

You can determine whether a function exists by using the `typeof` operator. In the following example, a test is performed to determine if the `window` object has a property called `noFunc` that is a function. If so, it is used; otherwise some other action is taken.

```
if ('function' === typeof window.noFunc) {

 } else {

 }
```

Note that in the `if` test, a reference to `noFunc` is used—there are no brackets "()" after the function name so the actual function is not called.

Specifications

Specification	Status	Comment
ECMAScript 1st Edition (ECMA-262)	Standard	Initial definition. Implemented in JavaScript 1.0
ECMAScript 5.1 (ECMA-262) The definition of 'Function Definition' in that specification.	Standard	
ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'Function definitions' in that specification.	Standard	New: Arrow functions, Generator functions, default parameters, rest parameters.
ECMAScript Latest Draft (ECMA-262) The definition of 'Function definitions' in that specification.	Living Standard	

Browser compatibility

1. The initial implementation of arrow functions in Firefox made them automatically strict. This has been changed as of Firefox 24. The use of `'use`

`strict';` is now required.

2. Prior to Firefox 39, a line terminator (`\n`) was incorrectly allowed after arrow function arguments. This has been fixed to conform to the ES2015 specification and code like `() \n => {}` will now throw a `SyntaxError` in this and later versions.

See also

- `function statement`
- `function expression`
- `function* statement`
- `function* expression`
- `Function`
- `GeneratorFunction`
- `Arrow functions`
- `Default parameters`
- `Rest parameters`
- `Arguments object`
- `getter`
- `setter`
- `Method definitions`
- Functions and function scope

Was this article helpful?

Thank you!