**Functions traditional and Arrow**

The **function** declaration defines a function with the specified parameters.

You can also define functions using the [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) constructor and a [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function).

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#syntax)

function name(param0) {

statements

}

function name(param0, param1) {

statements

}

function name(param0, param1, /\* … ,\*/ paramN) {

statements

}

Copy to Clipboard

name

The function name.

param Optional

The name of an argument to be passed to the function. Maximum number of arguments varies in different engines.

statements Optional

The statements which comprise the body of the function.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#description)

A function created with a function declaration is a Function object and has all the properties, methods and behavior of Function objects. See [Function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) for detailed information on functions.

A function can also be created using an expression (see [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function)).

By default, functions return undefined. To return any other value, the function must have a [return](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/return) statement that specifies the value to return.

[**Block-level function declaration**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#block-level_function_declaration)

**Warning:** In [non-strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode), function declarations inside blocks behave strangely. Only declare functions in blocks if you are in strict mode.

Functions can be conditionally declared — that is, a function statement can be nested within an [if](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) statement. However, in non-strict mode, the results are inconsistent across implementations.

console.log(`'foo' name ${"foo" in globalThis ? "is" : "is not"} global. typeof foo is ${typeof foo}`);

if (false) {

function foo() { return 1; }

}

// In Chrome:

// 'foo' name is global. typeof foo is undefined

//

// In Firefox:

// 'foo' name is global. typeof foo is undefined

//

// In Safari:

// 'foo' name is global. typeof foo is function

Copy to Clipboard

The scoping and hoisting effect won't change regardless of whether the if body is actually executed.

console.log(`'foo' name ${"foo" in globalThis ? "is" : "is not"} global. typeof foo is ${typeof foo}`);

if (true) {

function foo() { return 1; }

}

// In Chrome:

// 'foo' name is global. typeof foo is undefined

//

// In Firefox:

// 'foo' name is global. typeof foo is undefined

//

// In Safari:

// 'foo' name is global. typeof foo is function

Copy to Clipboard

In [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode), [block](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block)-level function declarations are scoped to that block and are hoisted to the top of the block.

"use strict";

{

foo(); // Logs "foo"

function foo() {

console.log("foo");

}

}

console.log(`'foo' name ${"foo" in globalThis ? "is" : "is not"} global. typeof foo is ${typeof foo}`);

// 'foo' name is not global. typeof foo is undefined

Copy to Clipboard

[**Function declaration hoisting**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#function_declaration_hoisting)

Function declarations in JavaScript are [hoisted](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting) to the top of the enclosing function or global scope. You can use the function before you declared it:

hoisted(); // Logs "foo"

function hoisted() {

console.log('foo');

}

Copy to Clipboard

Note that [function expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) are not hoisted:

notHoisted(); // TypeError: notHoisted is not a function

var notHoisted = function() {

console.log('bar');

};

Copy to Clipboard

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#examples)

[**Using function**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function#using_function)

The following code declares a function that returns the total amount of sales, when given the number of units sold of three products.

function calcSales(unitsA, unitsB, unitsC) {

return unitsA \* 79 + unitsB \* 129 + unitsC \* 699;

}

Copy to Clipboard

# Arrow function expressions

An **arrow function expression** is a compact alternative to a traditional [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function), with some semantic differences and deliberate limitations in usage:

* Arrow functions don't have their own bindings to [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), or [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), and should not be used as [methods](https://developer.mozilla.org/en-US/docs/Glossary/Method).
* Arrow functions cannot be used as [constructors](https://developer.mozilla.org/en-US/docs/Glossary/Constructor). Calling them with [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) throws a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError). They also don't have access to the [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target) keyword.
* Arrow functions cannot use [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) within their body and cannot be created as generator functions.

## [Try it](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#try_it)

## [Syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#syntax)

param => expression

(param) => expression

(param1, paramN) => expression

param => {

statements

}

(param1, paramN) => {

statements

}

Copy to Clipboard

[Rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters), [default parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters), and [destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) within params are supported, and always require parentheses:

(a, b, ...r) => expression

(a = 400, b = 20, c) => expression

([a, b] = [10, 20]) => expression

({ a, b } = { a: 10, b: 20 }) => expression

Copy to Clipboard

Arrow functions can be [async](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) by prefixing the expression with the async keyword.

async param => expression

async (param1, param2, ...paramN) => {

statements

}

Copy to Clipboard

## [Description](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#description)

Let's decompose a traditional anonymous function down to the simplest arrow function step-by-step. Each step along the way is a valid arrow function.

**Note:** Traditional function expressions and arrow functions have more differences than their syntax. We will introduce their behavior differences in more detail in the next few subsections.

// Traditional anonymous function

(function (a) {

return a + 100;

});

// 1. Remove the word "function" and place arrow between the argument and opening body bracket

(a) => {

return a + 100;

};

// 2. Remove the body braces and word "return" — the return is implied.

(a) => a + 100;

// 3. Remove the parameter parentheses

a => a + 100;

Copy to Clipboard

In the example above, both the parentheses around the parameter and the braces around the function body may be omitted. However, they can only be omitted in certain cases.

The parentheses can only be omitted if the function has a single simple parameter. If it has multiple parameters, no parameters, or default, destructured, or rest parameters, the parentheses around the parameter list are required.

// Traditional anonymous function

(function (a, b) {

return a + b + 100;

});

// Arrow function

(a, b) => a + b + 100;

const a = 4;

const b = 2;

// Traditional anonymous function (no parameters)

(function() {

return a + b + 100;

});

// Arrow function (no arguments)

() => a + b + 100;

Copy to Clipboard

The braces can only be omitted if the function directly returns an expression. If the body has additional lines of processing, the braces are required — and so is the return keyword. Arrow functions cannot guess what or when you want to return.

// Traditional anonymous function

(function (a, b) {

const chuck = 42;

return a + b + chuck;

});

// Arrow function

(a, b) => {

const chuck = 42;

return a + b + chuck;

};

Copy to Clipboard

Arrow functions are always unnamed. If the arrow function needs to call itself, use a named function expression instead. You can also assign the arrow function to a variable so it has a name.

// Traditional Function

function bob(a) {

return a + 100;

}

// Arrow Function

const bob2 = (a) => a + 100;

Copy to Clipboard

### [Function body](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#function_body)

Arrow functions can have either a concise body or the usual block body.

In a concise body, only a single expression is specified, which becomes the implicit return value. In a block body, you must use an explicit return statement.

const func = (x) => x \* x;

// concise body syntax, implied "return"

const func2 = (x, y) => {

return x + y;

};

// with block body, explicit "return" needed

Copy to Clipboard

Returning object literals using the concise body syntax (params) => { object: literal } does not work as expected.

const func = () => { foo: 1 };

// Calling func() returns undefined!

const func2 = () => { foo: function () {} };

// SyntaxError: function statement requires a name

const func3 = () => { foo() {} };

// SyntaxError: Unexpected token '{'

This is because JavaScript only sees the arrow function as having a concise body if the token following the arrow is not a left brace, so the code inside braces ({}) is parsed as a sequence of statements, where foo is a [label](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/label), not a key in an object literal.

To fix this, wrap the object literal in parentheses:

const func = () => ({ foo: 1 });

### [Cannot be used as methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#cannot_be_used_as_methods)

Arrow function expressions should only be used for non-method functions because they do not have their own this. Let's see what happens when we try to use them as methods:

"use strict";

const obj = {

i: 10,

b: () => console.log(this.i, this),

c() {

console.log(this.i, this);

},

};

obj.b(); // logs undefined, Window { /\* … \*/ } (or the global object)

obj.c(); // logs 10, Object { /\* … \*/ }

Copy to Clipboard

Another example involving [Object.defineProperty()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty):

'use strict';

const obj = {

a: 10,

};

Object.defineProperty(obj, 'b', {

get: () => {

console.log(this.a, typeof this.a, this); // undefined 'undefined' Window { /\* … \*/ } (or the global object)

return this.a + 10; // represents global object 'Window', therefore 'this.a' returns 'undefined'

},

});

Copy to Clipboard

Because a [class](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)'s body has a this context, arrow functions as [class fields](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Public_class_fields) close over the class's this context, and the this inside the arrow function's body will correctly point to the instance (or the class itself, for [static fields](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static)). However, because it is a [closure](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures), not the function's own binding, the value of this will not change based on the execution context.

class C {

a = 1;

autoBoundMethod = () => {

console.log(this.a);

}

}

const c = new C();

c.autoBoundMethod(); // 1

const { autoBoundMethod } = c;

autoBoundMethod(); // 1

// If it were a normal method, it should be undefined in this case

Copy to Clipboard

Arrow function properties are often said to be "auto-bound methods", because the equivalent with normal methods is:

class C {

a = 1;

constructor() {

this.method = this.method.bind(this);

}

method() {

console.log(this.a);

}

}

Copy to Clipboard

**Note:** Class fields are defined on the instance, not on the prototype, so every instance creation would create a new function reference and allocate a new closure, potentially leading to more memory usage than a normal unbound method.

For similar reasons, the [call()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call), [apply()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply), and [bind()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) methods are not useful when called on arrow functions, because arrow functions establish this based on the scope the arrow function is defined within, and the this value does not change based on how the function is invoked.

### [No binding of arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#no_binding_of_arguments)

Arrow functions do not have their own [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) object. Thus, in this example, arguments is a reference to the arguments of the enclosing scope:

const arguments = [1, 2, 3];

const arr = () => arguments[0];

arr(); // 1

function foo(n) {

const f = () => arguments[0] + n; // foo's implicit arguments binding. arguments[0] is n

return f();

}

foo(3); // 3 + 3 = 6

Copy to Clipboard

**Note:** You cannot declare a variable called arguments in [strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode#making_eval_and_arguments_simpler), so the code above would be a syntax error. This makes the scoping effect of arguments much easier to comprehend.

In most cases, using [rest parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) is a good alternative to using an arguments object.

function foo(n) {

const f = (...args) => args[0] + n;

return f(10);

}

foo(1); // 11

Copy to Clipboard

### [Cannot be used as constructors](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#cannot_be_used_as_constructors)

Arrow functions cannot be used as constructors and will throw an error when called with [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new). They also do not have a [prototype](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/prototype) property.

const Foo = () => {};

const foo = new Foo(); // TypeError: Foo is not a constructor

console.log("prototype" in Foo); // false

Copy to Clipboard

### [Cannot be used as generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#cannot_be_used_as_generators)

The [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) keyword cannot be used in an arrow function's body (except when used within generator functions further nested within the arrow function). As a consequence, arrow functions cannot be used as generators.

### [Line break before arrow](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#line_break_before_arrow)

An arrow function cannot contain a line break between its parameters and its arrow.

const func = (a, b, c)

=> 1;

// SyntaxError: Unexpected token '=>'

For the purpose of formatting, you may put the line break after the arrow or use parentheses/braces around the function body, as shown below. You can also put line breaks between parameters.

const func = (a, b, c) =>

1;

const func2 = (a, b, c) => (

1

);

const func3 = (a, b, c) => {

return 1;

};

const func4 = (

a,

b,

c,

) => 1;

Copy to Clipboard

### [Precedence of arrow](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#precedence_of_arrow)

Although the arrow in an arrow function is not an operator, arrow functions have special parsing rules that interact differently with [operator precedence](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence) compared to regular functions.

let callback;

callback = callback || () => {};

// SyntaxError: invalid arrow-function arguments

Because => has a lower precedence than most operators, parentheses are necessary to avoid callback || () being parsed as the arguments list of the arrow function.

callback = callback || (() => {});

## [Examples](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#examples)

### [Using arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#using_arrow_functions)

// An empty arrow function returns undefined

const empty = () => {};

(() => 'foobar')();

// Returns "foobar"

// (this is an Immediately Invoked Function Expression)

const simple = (a) => a > 15 ? 15 : a;

simple(16); // 15

simple(10); // 10

const max = (a, b) => a > b ? a : b;

// Easy array filtering, mapping, etc.

const arr = [5, 6, 13, 0, 1, 18, 23];

const sum = arr.reduce((a, b) => a + b);

// 66

const even = arr.filter((v) => v % 2 === 0);

// [6, 0, 18]

const double = arr.map((v) => v \* 2);

// [10, 12, 26, 0, 2, 36, 46]

// More concise promise chains

promise

.then((a) => {

// …

})

.then((b) => {

// …

});

// Parameterless arrow functions that are visually easier to parse

setTimeout(() => {

console.log("I happen sooner");

setTimeout(() => {

// deeper code

console.log("I happen later");

}, 1);

}, 1);

Copy to Clipboard

### [Using call, bind, and apply](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#using_call_bind_and_apply)

The [call()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call), [apply()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply), and [bind()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) methods work as expected with traditional functions, because we establish the scope for each of the methods:

const obj = {

num: 100,

};

// Setting "num" on globalThis to show how it is NOT used.

globalThis.num = 42;

// A simple traditional function to operate on "this"

const add = function (a, b, c) {

return this.num + a + b + c;

};

console.log(add.call(obj, 1, 2, 3)); // 106

console.log(add.apply(obj, [1, 2, 3])); // 106

const boundAdd = add.bind(obj);

console.log(boundAdd(1, 2, 3)); // 106

Copy to Clipboard

With arrow functions, since our add function is essentially created on the globalThis (global) scope, it will assume this is the globalThis.

const obj = {

num: 100,

};

// Setting "num" on globalThis to show how it gets picked up.

globalThis.num = 42;

// Arrow function

const add = (a, b, c) => this.num + a + b + c;

console.log(add.call(obj, 1, 2, 3)); // 48

console.log(add.apply(obj, [1, 2, 3])); // 48

const boundAdd = add.bind(obj);

console.log(boundAdd(1, 2, 3)); // 48

Copy to Clipboard

Perhaps the greatest benefit of using arrow functions is with methods like [setTimeout()](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout) and [EventTarget.prototype.addEventListener()](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener) that usually require some kind of closure, call(), apply(), or bind() to ensure that the function is executed in the proper scope.

With traditional function expressions, code like this does not work as expected:

const obj = {

count: 10,

doSomethingLater() {

setTimeout(function () { // the function executes on the window scope

this.count++;

console.log(this.count);

}, 300);

},

};

obj.doSomethingLater(); // logs "NaN", because the property "count" is not in the window scope.

Copy to Clipboard

With arrow functions, the this scope is more easily preserved:

const obj = {

count: 10,

doSomethingLater() {

// The method syntax binds "this" to the "obj" context.

setTimeout(() => {

// Since the arrow function doesn't have its own binding and

// setTimeout (as a function call) doesn't create a binding

// itself, the "obj" context of the outer method is used.

this.count++;

console.log(this.count);

}, 300);

},

};

obj.doSomethingLater(); // logs 11

Copy to Clipboard

## [Specifications](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#specifications)