mdn web docs_

Arrow function expressions

An **arrow function expression** is a compact alternative to a traditional function expression, with some semantic differences and deliberate limitations in usage:

- Arrow functions don't have their own bindings to `this`, `arguments`, or `super`, and should not be used as methods.
- Arrow functions cannot be used as constructors. Calling them with `new` throws a `TypeError`. They also don't have access to the `new.target` keyword.
- Arrow functions cannot use `yield` within their body and cannot be created as generator functions.

Try it

JavaScript Demo: Functions ⇒

```
const materials = ['Hydrogen', 'Helium', 'Lithium', 'Beryllium'];

console.log(materials.map((material) => material.length));
// Expected output: Array [8, 6, 7, 9]

```

Run › Reset

Syntax

JS

```
() => expression

param => expression

(param) => expression

(param1, paramN) => expression

() => {
  statements
}

param => {
  statements
}

(param1, paramN) => {
  statements
}
```

Rest parameters, default parameters, and destructuring within params are supported, and always require parentheses:

JS
```js
(a, b, ...r) => expression
(a = 400, b = 20, c) => expression
([a, b] = [10, 20]) => expression
({ a, b } = { a: 10, b: 20 }) => expression
```

Arrow functions can be `async` by prefixing the expression with the `async` keyword.

JS
```js
async param => expression
async (param1, param2, ...paramN) => {
  statements
}
```

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.

JS
```js
// Traditional anonymous function
(function (a) {
  return a + 100;
});

// 1. Remove the word "function" and place arrow between the argument and opening body brace
(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;
```

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.

JS
```js
// 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 parameters)
() => a + b + 100;
```

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.

JS

```
// Traditional anonymous function
(function (a, b) {
  const chuck = 42;
  return a + b + chuck;
});

// Arrow function
(a, b) => {
  const chuck = 42;
  return a + b + chuck;
};
```

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.

JS

```
// Traditional Function
function bob(a) {
  return a + 100;
}

// Arrow Function
const bob2 = (a) => a + 100;
```

Function body

Arrow functions can have either an *expression body* or the usual *block body*.

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

JS

```
const func = (x) => x * x;
// expression body syntax, implied "return"

const func2 = (x, y) => {
  return x + y;
};
// with block body, explicit "return" needed
```

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

JS

```
const func = () => { foo: 1 };
// Calling func() returns undefined!

const func2 = () => { foo: function () {} };
// SyntaxError: function statement requires a name
```

```js
const func3 = () => { foo() {} };
// SyntaxError: Unexpected token '{'
```

This is because JavaScript only sees the arrow function as having an expression 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, not a key in an object literal.

To fix this, wrap the object literal in parentheses:

JS

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

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:

JS

```js
"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 { /* … */ }
```

Another example involving `Object.defineProperty()`:

JS

```js
"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'
  },
});
```

Because a class's body has a `this` context, arrow functions as 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). However, because it is a closure, not the function's own binding, the value of `this` will not change based on the execution context.

JS

```js
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
```

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

JS

```
class C {
  a = 1;
  constructor() {
    this.method = this.method.bind(this);
  }
  method() {
    console.log(this.a);
  }
}
```

> **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()`, `apply()`, and `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

Arrow functions do not have their own `arguments` object. Thus, in this example, `arguments` is a reference to the arguments of the enclosing scope:

JS

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

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

> **Note:** You cannot declare a variable called `arguments` in strict mode, 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 is a good alternative to using an `arguments` object.

JS

```
function foo(n) {
  const f = (...args) => args[0] + n;
  return f(10);
}

foo(1); // 11
```

Cannot be used as constructors

Arrow functions cannot be used as constructors and will throw an error when called with `new`. They also do not have a `prototype` property.

JS

```js
const Foo = () => {};
const foo = new Foo(); // TypeError: Foo is not a constructor
console.log("prototype" in Foo); // false
```

Cannot be used as generators

The `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

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

JS

```js
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.

JS

```js
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;
```

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 compared to regular functions.

JS

```js
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.

JS

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

Examples

Using arrow functions

JS

```
// 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);
```

Using call, bind, and apply

The `call()` , `apply()` , and `bind()` methods work as expected with traditional functions, because we establish the scope for each of the methods:

JS

```
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;
};
```

```js
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
```

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

JS

```js
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
```

Perhaps the greatest benefit of using arrow functions is with methods like `setTimeout()` and `EventTarget.prototype.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:

JS

```js
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.
```

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

JS

```js
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
```

Specifications

Specification
ECMAScript Language Specification # sec-arrow-function-definitions

Browser compatibility

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android
Arrow functions	Chrome 45	Edge 12	Firefox 22	Opera 32	Safari 10	Chrome Android 45	Firefox for Android 22	Opera Android
Trailing comma in parameters	Chrome 58	Edge 12	Firefox 52	Opera 45	Safari 10	Chrome Android 58	Firefox for Android 52	Opera Android

Tip: you can click/tap on a cell for more information.

Full support See implementation notes.

See also

- Functions guide
- Functions
- `function`
- `function` expression
- ES6 In Depth: Arrow functions on hacks.mozilla.org (2015)

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Oct 5, 2023 by MDN contributors.