mdn web docs_

Arrow function expressions

An **arrow function expression** is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

Differences & Limitations:

- Does not have its own bindings to `this` or `super`, and should not be used as `methods`.
- Does not have `new.target` keyword.
- Not suitable for `call`, `apply` and `bind` methods, which generally rely on establishing a scope.
- Can not be used as constructors.
- Can not use `yield`, within its body.

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

Comparing traditional functions to arrow functions

Let's decompose a "traditional anonymous function" down to the simplest "arrow function" step-by-step:

> **Note:** Each step along the way is a valid "arrow function".

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

// Arrow Function Break Down

// 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 argument parentheses
a => a + 100;
```

The { braces } and (parentheses) and "return" are required in some cases.

For example, if you have **multiple arguments** or **no arguments**, you'll need to re-introduce parentheses around the arguments:

```
// Traditional Anonymous Function
function (a, b){
  return a + b + 100;
}

// Arrow Function
(a, b) => a + b + 100;

// Traditional Anonymous Function (no arguments)
```

```
let a = 4;
let b = 2;
function (){
  return a + b + 100;
}

// Arrow Function (no arguments)
let a = 4;
let b = 2;
() => a + b + 100;
```

Likewise, if the body requires **additional lines** of processing, you'll need to re-introduce braces **PLUS the "return"** (arrow functions do not magically guess what or when you want to "return"):

```
// Traditional Anonymous Function
function (a, b){
  let chuck = 42;
  return a + b + chuck;
}

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

And finally, for **named functions** we treat arrow expressions like variables:

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

// Arrow Function
let bob = a => a + 100;
```

Syntax

Basic syntax

One param. With simple expression return is not needed:

```
param => expression
```

Multiple params require parentheses. With simple expression return is not needed:

```
(param1, paramN) => expression
```

Multiline statements require body braces and return:

```
param => {
  let a = 1;
  return a + param;
}
```

Multiple params require parentheses. Multiline statements require body braces and return:

```
(param1, paramN) => {
   let a = 1;
   return a + param1 + paramN;
}
```

Advanced syntax

To return an object literal expression requires parentheses around expression:

```
params => ({foo: "a"}) // returning the object {foo: "a"}
```

Rest parameters are supported:

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

Default parameters are supported:

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

Destructuring within params supported:

```
([a, b] = [10, 20]) => a + b;  // result is 30
({ a, b } = { a: 10, b: 20 }) => a + b; // result is 30
```

Description

Arrow functions used as methods

As stated previously, arrow function expressions are best suited for non-method functions. Let's see what happens when we try to use them as methods:

```
'use strict';

var obj = { // does not create a new scope
  i: 10,
  b: () => console.log(this.i, this),
  c: function() {
    console.log(this.i, this);
  }
}

obj.b(); // prints undefined, Window {...} (or the global object)
obj.c(); // prints 10, Object {...}
```

Arrow functions do not have their own `this`. Another example involving `Object.defineProperty()`:

```
'use strict';

var 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'
  }
});
```

call, apply and bind

The `call`, `apply` and `bind` methods are **NOT suitable** for Arrow functions -- as they were designed to allow methods to execute within different scopes -- because **Arrow functions establish "this" based on the scope the Arrow function is defined within.**

For example `call`, `apply` and `bind` work as expected with Traditional functions, because we establish the scope for each of the methods:

```
// ----------------------
// Traditional Example
// ----------------------
// A simplistic object with its very own "this".
var obj = {
    num: 100
}

// Setting "num" on window to show how it is NOT used.
window.num = 2020; // yikes!

// A simple traditional function to operate on "this"
var add = function (a, b, c) {
  return this.num + a + b + c;
}

// call
var result = add.call(obj, 1, 2, 3) // establishing the scope as "obj"
console.log(result) // result 106

// apply
const arr = [1, 2, 3]
```

```
var result = add.apply(obj, arr) // establishing the scope as "obj"
console.log(result) // result 106

// bind
var result = add.bind(obj) // establishing the scope as "obj"
console.log(result(1, 2, 3)) // result 106
```

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

```
// ----------------------
// Arrow Example
// ----------------------

// A simplistic object with its very own "this".
var obj = {
    num: 100
}

// Setting "num" on window to show how it gets picked up.
window.num = 2020; // yikes!

// Arrow Function
var add = (a, b, c) => this.num + a + b + c;

// call
console.log(add.call(obj, 1, 2, 3)) // result 2026

// apply
const arr = [1, 2, 3]
console.log(add.apply(obj, arr)) // result 2026

// bind
```

```
const bound = add.bind(obj)
console.log(bound(1, 2, 3)) // result 2026
```

Perhaps the greatest benefit of using Arrow functions is with DOM-level methods (setTimeout, setInterval, addEventListener) that usually required some kind of closure, call, apply or bind to ensure the function executed in the proper scope.

Traditional Example:

```
var obj = {
    count : 10,
    doSomethingLater : function (){
        setTimeout(function(){ // the function executes on the window scope
            this.count++;
            console.log(this.count);
        }, 300);
    }
}

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

Arrow Example:

```
var obj = {
    count : 10,
    doSomethingLater : function(){
        // The traditional function 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 traditional function will
            // be used within.
            this.count++;
            console.log(this.count);
        }, 300);
    }
}

obj.doSomethingLater();
```

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:

```
var arguments = [1, 2, 3];
var arr = () => arguments[0];

arr(); // 1

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

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

In most cases, using rest parameters is a good alternative to using an `arguments` object.

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

```
foo(1); // 11
```

Use of the `new` operator

Arrow functions cannot be used as constructors and will throw an error when used with `new`.

```
var Foo = () => {};
var foo = new Foo(); // TypeError: Foo is not a constructor
```

Use of `prototype` property

Arrow functions do not have a `prototype` property.

```
var Foo = () => {};
console.log(Foo.prototype); // undefined
```

Use of the `yield` keyword

The `yield` keyword may not be used in an arrow function's body (except when permitted within functions further nested within it). As a consequence, arrow functions cannot be used as generators.

Function body

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

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

```
var func = x => x * x;
// concise body syntax, implied "return"

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

Returning object literals

Keep in mind that returning object literals using the concise body syntax `params => {object:literal}` will not work as expected.

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

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

This is because the code inside braces ({}) is parsed as a sequence of statements (i.e. `foo` is treated like a label, not a key in an object literal).

You must wrap the object literal in parentheses:

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

Line breaks

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

```
var func = (a, b, c)
  => 1;
// SyntaxError: expected expression, got '=>'
```

However, this can be amended by putting the line break after the arrow or using parentheses/braces as seen below to ensure that the code stays pretty and fluffy. You can also put line breaks between arguments.

```
var func = (a, b, c) =>
  1;

var func = (a, b, c) => (
  1
);

var func = (a, b, c) => {
  return 1
};

var func = (
  a,
  b,
  c
) => 1;

// no SyntaxError thrown
```

Parsing order

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.

```
let callback;

callback = callback || function() {}; // ok

callback = callback || () => {};
// SyntaxError: invalid arrow-function arguments

callback = callback || (() => {});    // ok
```

Examples

Basic usage

```
// An empty arrow function returns undefined
let empty = () => {};

(() => 'foobar')();
// Returns "foobar"
// (this is an Immediately Invoked Function Expression)

var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10

let max = (a, b) => a > b ? a : b;
```

```
// Easy array filtering, mapping, ...

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

var sum = arr.reduce((a, b) => a + b);
// 66

var even = arr.filter(v => v % 2 == 0);
// [6, 0, 18]

var 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);
```

Specifications

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

Browser compatibility

Report problems with this compatibility data on GitHub

Arrow functions
✓ Chrome 45
✓ Edge 12
✓ Firefox 22
✗ Internet Explorer No
✓ Opera 32
✓ Safari 10
✓ WebView Android 45
✓ Chrome Android 45
✓ Firefox for Android 22
✓ Opera Android 32
✓ Safari on iOS 10
✓ Samsung Internet 5.0
✓ Deno 1.0

✓ Node.js 4.0.0

Trailing comma in parameters

✓ Chrome 58

✓ Edge 12

✓ Firefox 52

✗ Internet Explorer No

✓ Opera 45

✓ Safari 10

✓ WebView Android 58

✓ Chrome Android 58

✓ Firefox for Android 52

✓ Opera Android 43

✓ Safari on iOS 10

✓ Samsung Internet 7.0

✓ Deno 1.0

✓ Node.js 8.0.0

✓ Full support ◆ Partial support ✗ No support ✱ See implementation notes. ⚑ User must explicitly enable this feature.

See also

- "ES6 In Depth: Arrow functions" on hacks.mozilla.org

Last modified: Feb 18, 2022, by MDN contributors