MDN web docs moz://a

Sign in

English ▾

Arrow function expressions

An **arrow function expression** is a syntactically compact alternative to a regular function expression, although without its own bindings to the `this`, `arguments`, `super`, or `new.target` keywords. Arrow function expressions are ill suited as methods, and they cannot be used as constructors.

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

Basic syntax

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter name:
(singleParam) => { statements }
singleParam => { statements }

// The parameter list for a function with no parameters should be written
with a pair of parentheses.
() => { statements }
```

Advanced syntax

```
// Parenthesize the body of a function to return an object literal
expression:
params => ({foo: bar})

// Rest parameters and default parameters are supported
(param1, param2, ...rest) => { statements }
(param1 = defaultValue1, param2, …, paramN = defaultValueN) => {
statements }

// Destructuring within the parameter list is also supported
var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;
f(); // 6
```

Description

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

Two factors influenced the introduction of arrow functions: the need for shorter functions and the behavior of the `this` keyword.

Shorter functions

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

// This statement returns the array: [8, 6, 7, 9]
elements.map(function(element) {
  return element.length;
});

// The regular function above can be written as the arrow function below
elements.map((element) => {
  return element.length;
}); // [8, 6, 7, 9]

// When there is only one parameter, we can remove the surrounding parentheses
elements.map(element => {
  return element.length;
}); // [8, 6, 7, 9]

// When the only statement in an arrow function is `return`, we can remove `re
// the surrounding curly brackets
elements.map(element => element.length); // [8, 6, 7, 9]

// In this case, because we only need the length property, we can use destruct
// Notice that the `length` corresponds to the property we want to get whereas
// obviously non-special `lengthFooBArX` is just the name of a variable which
// to any valid variable name you want
elements.map(({ length: lengthFooBArX }) => lengthFooBArX); // [8, 6, 7, 9]

// This destructuring parameter assignment can also be written as seen below.
// this example we are not assigning `length` value to the made up property. I
// itself of the variable `length` is used as the property we want to retrieve
elements.map(({ length }) => length); // [8, 6, 7, 9]
```

No separate this

Before arrow functions, every new function defined its own `this` value based on how the function was called:

- A new object in the case of a constructor.
- `undefined` in strict mode function calls.
- The base object if the function was 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 an instance of itself.
  this.age = 0;

  setInterval(function growUp() {
    // In non-strict mode, the growUp() function defines `this`
    // as the global object (because it's where growUp() is executed.),
    // which is different from the `this`
    // defined by the Person() constructor.
    this.age++;
  }, 1000);
}

var p = new Person();
```

In ECMAScript 3/5, the `this` issue was fixable by assigning the value in `this` to a variable that could be closed over.

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

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

Alternatively, a bound function could be created so that a preassigned `this` value would be passed to the bound target function (the `growUp()` function in the example above).

An arrow function does not have its own `this`. The `this` value of the enclosing lexical scope is used; arrow functions follow the normal variable lookup rules. So while searching for `this` which is not present in the current scope, an arrow function ends up finding the `this` from its enclosing scope.

Thus, in the following code, the `this` within the function that is passed to `setInterval` has the same value as the `this` in the lexically enclosing function:

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

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

var p = new Person();
```

Relation with strict mode

Given that `this` comes from the surrounding lexical context, strict mode rules with regard to `this` are ignored.

```
var f = () => { 'use strict'; return this; };
f() === window; // or the global object
```

All other strict mode rules apply normally.

CORRECTION: START

NOTE: the previous statement seems false.

Strict mode should prevent creating global variables when assigning to an undeclared identifier in a function.

This code sample using Chrome 81 demonstrates that arrow functions allow the creation of global variables in such situations (both for a concise body and for a normal function body):

```
> f1 = x => { y = x; console.log(`x: ${x}, y: ${y}`); return x + 1; }
x => { y = x; console.log(`x: ${x}, y: ${y}`); return x + 1; }

> y
VM51587:1 Uncaught ReferenceError: y is not defined
    at <anonymous>:1:1
(anonymous) @ VM51587:1

> f1(3)
VM51533:1 x: 3, y: 3
4

> y
3

> f2 = x => { 'use strict'; z = x; console.log(`x: ${x}, z: ${z}`); retur
x => { 'use strict'; z = x; console.log(`x: ${x}, z: ${z}`); return x + 

> z
VM51757:1 Uncaught ReferenceError: z is not defined
    at <anonymous>:1:1
(anonymous) @ VM51757:1

```

```
> f2(4)
VM51712:1 Uncaught ReferenceError: z is not defined
    at f2 (<anonymous>:1:29)
    at <anonymous>:1:1
f2 @ VM51712:1
(anonymous) @ VM51800:1

> f3 = x => (z1 = x + 1)
x => (z1 = x + 1)

> z1
VM51891:1 Uncaught ReferenceError: z1 is not defined
    at <anonymous>:1:1
(anonymous) @ VM51891:1

> f3(10)
11

> z1
11
```

f2 illustrates that when explicitly setting the arrow function to apply strict mode, it does throw an error when attempting to assign an undeclared variable.

https://www.ecma-international.org/ecma-262/10.0/index.html#sec-strict-mode-code

https://www.ecma-international.org/ecma-262/10.0/index.html#sec-arrow-function-definitions-runtime-semantics-evaluation

CORRECTION: END

Invoked through call or apply

Since arrow functions do not have their own `this`, the methods `call()` and `apply()` can only pass in parameters. Any `this` argument is ignored.

```
var adder = {
  base: 1,

  add: function(a) {
    var f = v => v + this.base;
    return f(a);
  },

  addThruCall: function(a) {
    var f = v => v + this.base;
    var b = {
      base: 2
    };

    return f.call(b, a);
  }
};

console.log(adder.add(1));         // This would log 2
console.log(adder.addThruCall(1)); // This would log 2 still
```

No binding of arguments

Arrow functions do not have their own `arguments` object. Thus, in this example, `arguments` is simply 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. argumer
  return f();
}

foo(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
```

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' Wi
    return this.a + 10; // represents global object 'Window', therefore '
  }
});
```

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 (ECMA-262) The definition of 'Arrow Function Definitions' in that specification.

Browser compatibility

Update compatibility data on GitHub

Arrow functions

Chrome	45
Edge	12
Firefox	22
IE	No
Opera	32
Safari	10
WebView Android	45
Chrome Android	45

Firefox Android	22
Opera Android	32
Safari iOS	10
Samsung Internet Android	5.0
nodejs	Yes

Trailing comma in parameters

Chrome	58
Edge	12
Firefox	52
IE	No
Opera	45
Safari	10
WebView Android	58
Chrome Android	58
Firefox Android	52
Opera Android	43
Safari iOS	10
Samsung Internet Android	7.0
nodejs	Yes

What are we missing?

Full support

No support

See implementation notes.

See also

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

Last modified: Aug 4, 2020, by MDN contributors

Related Topics

JavaScript

Tutorials:

▸ Complete beginners

▸ JavaScript Guide

▸ Intermediate

▸ Advanced

References:

▸ Built-in objects

▸ Expressions & operators

▸ Statements & declarations

▾ Functions

Arrow function expressions

Default parameters

Method definitions

Rest parameters

The arguments object

getter

setter

- Classes
- Errors
- Misc