Arrow functions

An **arrow function expression** has a shorter syntax than a function expression and does not have its own `this`, arguments, super, or new.target. These function expressions are best suited for non-method functions, and they cannot be used as constructors.

JavaScript Demo: Functions =>

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

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

```

Run ›

Reset

```
> Array [8, 6, 7, 9]
```

Syntax

Basic Syntax

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

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


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

Advanced Syntax

```
// Parenthesize the body of 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
let 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: shorter functions and non-binding of `this`.

Shorter functions

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

materials.map(function(material) {
  return material.length;
}); // [8, 6, 7, 9]

materials.map((material) => {
  return material.length;
}); // [8, 6, 7, 9]

materials.map(({length}) => length); // [8, 6, 7, 9]
```

No separate `this`

Until arrow functions, every new function defined its own `this` value (a new object in the case of a constructor, undefined in strict mode function calls, the base object if the function is 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, 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 execution context is used. Thus, in the following code, the `this` within the function that is passed to `setInterval` has the same value as `this` in the 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.

Invoked through call or apply

Since arrow functions do not have their own `this`, the methods `call()` or `apply()` can only pass in parameters. `thisArg` 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 to 2
console.log(adder.addThruCall(1)); // This would log to 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 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
  return f();
}

foo(1); // 2
```

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 = {
  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);
    return this.a + 10; // represents global object 'Window', therefo
  }
});
```

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 explicit 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).

Remember to 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 = ()
           => 1;
// SyntaxError: expected expression, got '=>'
```

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
```

More examples

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

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

```
// see 'IIFE' in glossary)

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	Status	Comment
ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'Arrow Function Definitions' in that specification.	ST Standard	Initial definition.
ECMAScript Latest Draft (ECMA-262) The definition of 'Arrow Function Definitions' in that specification.	D Draft	

Browser compatibility

Desktop Mobile

Feature	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari
Basic support	45	Yes	22[1] [2]	No	32	10
Trailing comma in parameters	?	?	52	No	?	?

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

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