MDN web docs moz://a

Sign in

English ▾

The arguments object

`arguments` is an `Array`-like object accessible inside functions that contains the values of the arguments passed to that function.

JavaScript Demo: Functions Arguments

```
function func1(a, b, c) {
  console.log(arguments[0]);
  // expected output: 1

  console.log(arguments[1]);
  // expected output: 2

  console.log(arguments[2]);
  // expected output: 3
}

func1(1, 2, 3);

```

Run ›

Reset

Description

Note: If you're writing ES6 compatible code, then rest parameters should be preferred.

> **Note:** "Array-like" means that `arguments` has a `length` property and properties indexed from zero, but it doesn't have `Array`'s built-in methods like `forEach()` or `map()`. See §Description for details.

The `arguments` object is a local variable available within all non-arrow functions. You can refer to a function's arguments inside that function by using its `arguments` object. It has entries for each argument the function was called with, with the first entry's index at `0`.

For example, if a function is passed 3 arguments, you can access them as follows:

```
arguments[0] // first argument
arguments[1] // second argument
arguments[2] // third argument
```

Each argument can also be set or reassigned:

```
arguments[1] = 'new value';
```

The `arguments` object is not an `Array`. It is similar, but lacks all `Array` properties except `length`. For example, it does not have the `pop()` method.

However, it can be converted to a real `Array`:

```
var args = Array.prototype.slice.call(arguments);
// Using an array literal is shorter than above but allocates an empty ar
var args = [].slice.call(arguments);
```

As you can do with any Array-like object, you can use ES2015's `Array.from()` method or spread syntax to convert `arguments` to a real Array:

```
let args = Array.from(arguments);
// or
let args = [...arguments];
```

The `arguments` object is useful for functions called with more arguments than they are formally declared to accept. This technique is useful for functions that can be passed a variable number of arguments, such as `Math.min()`. This example function accepts any number of string arguments and returns the longest one:

```
function longestString() {
  var longest = '';
  for (var i=0; i < arguments.length; i++) {
    if (arguments[i].length > longest.length) {
      longest = arguments[i];
    }
  }
  return longest;
}
```

You can use `arguments.length` to count how many arguments the function was called with. If you instead want to count how many parameters a function is declared to accept, inspect that function's `length` property.

Using typeof with arguments

The `typeof` operator returns `'object'` when used with `arguments`

```
console.log(typeof arguments); // 'object'
```

The type of individual arguments can be determined by indexing `arguments`:

```
console.log(typeof arguments[0]); // returns the type of the first argume
```

Properties

`arguments.callee`
  Reference to the currently executing function that the arguments belong to. Forbidden in strict mode.

`arguments.length`
  The number of arguments that were passed to the function.

`arguments[@@iterator]`
  Returns a new Array iterator object that contains the values for each index in `arguments`.

Examples

Defining a function that concatenates several strings

This example defines a function that concatenates several strings. The function's only formal argument is a string containing the characters that separate the items to concatenate.

```
function myConcat(separator) {
  let args = Array.prototype.slice.call(arguments, 1);
  return args.join(separator);
}
```

You can pass as many arguments as you like to this function. It returns a string list using each argument in the list:

```
// returns "red, orange, blue"
myConcat(', ', 'red', 'orange', 'blue');

// returns "elephant; giraffe; lion; cheetah"
myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');

// returns "sage. basil. oregano. pepper. parsley"
myConcat('. ', 'sage', 'basil', 'oregano', 'pepper', 'parsley');
```

Defining a function that creates HTML lists

This example defines a function that creates a string containing HTML for a list. The only formal argument for the function is a string that is "`u`" if the list is to be unordered (bulleted), or "`o`" if the list is to be ordered (numbered). The function is defined as follows:

```
function list(type) {
  var html = '<' + type + 'l><li>';
  var args = Array.prototype.slice.call(arguments, 1);
  html += args.join('</li><li>');
  html += '</li></' + type + 'l>'; // end list
  return html;
}
```

You can pass any number of arguments to this function, and it adds each argument as a list item to a list of the type indicated. For example:

```
let listHTML = list('u', 'One', 'Two', 'Three');

/* listHTML is:
"<ul><li>One</li><li>Two</li><li>Three</li></ul>"
*/
```

Rest, default, and destructured parameters

The `arguments` object can be used in conjunction with rest, default, and destructured parameters.

```
function foo(...args) {
  return args;
}
foo(1, 2, 3); // [1, 2, 3]
```

While the presence of rest, default, or destructured parameters does not alter the behavior of the `arguments` object in strict mode code, there are subtle differences for non-strict code.

In strict-mode code, the `arguments` object behaves the same whether or not a function is passed rest, default, or destructured parameters. That is, assigning new values to variables in the body of the function will not affect the `arguments` object. Nor will assigning new variables to the `arguments` object affect the value of variables.

> **Note:** You cannot write a `"use strict";` directive in the body of a function definition that accepts rest, default, or destructured parameters. Doing so will throw a syntax error.

Non-strict functions that are passed only simple parameters (that is, not rest, default, or restructured parameters) will sync the value of variables new values in the body of the function with the `arguments` object, and vice versa:

```
function func(a) {
  arguments[0] = 99; // updating arguments[0] also updates a
  console.log(a);
}
func(10); // 99
```

And also:

```
function func(a) {
  a = 99; // updating a also updates arguments[0]
  console.log(arguments[0]);
}
func(10); // 99
```

Conversely, non-strict functions that **are** passed rest, default, or destructured parameters **will not** sync new values assigned to argument variables in the function body with the `arguments` object. Instead, the `arguments` object in non-strict functions with complex parameters **will always** reflect the values passed to the function when the function was called (this is the same behavior as exhibited by all strict-mode functions, regardless of the type of variables they are passed):

```
function func(a = 55) {
  arguments[0] = 99; // updating arguments[0] does not also update a
  console.log(a);
}
func(10); // 10
```

And also:

```
function func(a = 55) {
  a = 99; // updating a does not also update arguments[0]
  console.log(arguments[0]);
}
func(10); // 10
```

And also:

```
// An untracked default parameter
function func(a = 55) {
  console.log(arguments[0]);
}
func(); // undefined
```

Specifications

Specification
ECMAScript (ECMA-262) The definition of 'Arguments Exotic Objects' in that specification.

Browser compatibility

Update compatibility data on GitHub

`arguments`

Chrome	1
Edge	12
Firefox	1
IE	3
Opera	3
Safari	1
WebView Android	1
Chrome Android	18
Firefox Android	4
Opera Android	10.1
Safari iOS	1
Samsung Internet Android	1.0
nodejs	Yes

`callee`

Chrome	1
Edge	12
Firefox	1
IE	6
Opera	4
Safari	1
WebView Android	1
Chrome Android	18
Firefox Android	4
Opera Android	10.1
Safari iOS	1
Samsung Internet Android	1.0

nodejs	Yes

`length`

Chrome	1
Edge	12
Firefox	1
IE	4
Opera	4
Safari	1
WebView Android	1
Chrome Android	18
Firefox Android	4
Opera Android	10.1
Safari iOS	1
Samsung Internet Android	1.0
nodejs	Yes

`@@iterator`

Chrome	52
Edge	12
Firefox	46
IE	No
Opera	39
Safari	9
WebView Android	52
Chrome Android	52
Firefox Android	46
Opera Android	41

Safari iOS	9
Samsung Internet Android	6.0
nodejs	Yes

What are we missing?

☐	Full support
☐	No support

See also

- `Function`
- Rest parameters

Last modified: Jul 19, 2020, by MDN contributors

Related Topics

JavaScript

Tutorials:

- ▸ Complete beginners
- ▸ JavaScript Guide
- ▸ Intermediate
- ▸ Advanced

References: