MDN Web Docs
moz://a

# eval()

> **Warning:** Executing JavaScript from a string is an enormous security risk. It is far too easy for a bad actor to run arbitrary code when you use `eval()`. See Never use eval()!, below.

The **`eval()`** function evaluates JavaScript code represented as a string.

**JavaScript Demo: Standard built-in objects - eval()**

```
1  console.log(eval('2 + 2'));
2  // expected output: 4
3
4  console.log(eval(new String('2 + 2')));
5  // expected output: 2 + 2
6
7  console.log(eval('2 + 2') === eval('4'));
8  // expected output: true
9
10 console.log(eval('2 + 2') === eval(new String('2 + 2')));
11 // expected output: false
12
```

Run ›

Reset

## Syntax

```
eval(string)
```

### Parameters

`string`

A string representing a JavaScript expression, statement, or sequence of statements. The expression can include variables and properties of existing objects.

### Return value

The completion value of evaluating the given code. If the completion value is empty, `undefined` is returned.

## Description

`eval()` is a function property of the global object.

The argument of the `eval()` function is a string. If the string represents an expression, `eval()` evaluates the expression. If the argument represents one or more JavaScript statements, `eval()` evaluates the statements. Do not call `eval()` to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

If you construct an arithmetic expression as a string, you can use `eval()` to evaluate it at a later time. For example, suppose you have a variable `x`. You can postpone evaluation of an expression involving `x` by assigning the string value of the expression, say "`3 * x + 2`", to a variable, and then calling `eval()` at a later point in your script.

If the argument of `eval()` is not a string, `eval()` returns the argument unchanged. In the following example, the `String` constructor is specified and `eval()` returns a `String` object rather than evaluating the string.

```
eval(new String('2 + 2')); // returns a String object containing "2 + 2"
eval('2 + 2');             // returns 4
```

You can work around this limitation in a generic fashion by using `toString()`.

```
var expression = new String('2 + 2');
eval(expression.toString());            // returns 4
```

If you use the `eval` function *indirectly,* by invoking it via a reference other than `eval`, as of ECMAScript 5 it works in the global scope rather than the local scope. This means, for

instance, that function declarations create global functions, and that the code being evaluated doesn't have access to local variables within the scope where it's being called.

```
function test() {
  var x = 2, y = 4;
  // Direct call, uses local scope
  console.log(eval('x + y')); // Result is 6
  // Indirect call using the comma operator to return eval
  console.log((0, eval)('x + y')); // Uses global scope, throws because x is u
  // Indirect call using a variable to store and return eval
  var geval = eval;
  console.log(geval('x + y')); // Uses global scope, throws because x is undef
}
```

## Never use eval()!

`eval()` is a dangerous function, which executes the code it's passed with the privileges of the caller. If you run `eval()` with a string that could be affected by a malicious party, you may end up running malicious code on the user's machine with the permissions of your webpage / extension. More importantly, a third-party code can see the scope in which `eval()` was invoked, which can lead to possible attacks in ways to which the similar `Function` is not susceptible.

`eval()` is also slower than the alternatives, since it has to invoke the JavaScript interpreter, while many other constructs are optimized by modern JS engines.

Additionally, modern javascript interpreters convert javascript to machine code. This means that any concept of variable naming gets obliterated. Thus, any use of `eval()` will force the browser to do long expensive variable name lookups to figure out where the variable exists in the machine code and set its value. Additionally, new things can be introduced to that variable through `eval()` such as changing the type of that variable, forcing the browser to re-evaluate all of the generated machine code to compensate.

Fortunately, there's a very good alternative to `eval()`: using `window.Function()`. See this example of how to convert code using a dangerous `eval()` to using `Function()`, see below.

Bad code with `eval()`:

```
function looseJsonParse(obj){
```

```
    return eval("(" + obj + ")");
}
console.log(looseJsonParse(
   "{a:(4-1), b:function(){}, c:new Date()}"
))
```

Better code without `eval()`:

```
function looseJsonParse(obj){
    return Function('"use strict";return (' + obj + ')')();
}
console.log(looseJsonParse(
   "{a:(4-1), b:function(){}, c:new Date()}"
))
```

Comparing the two code snippets above, the two code snippets might seem to work the same way, but think again: the `eval()` one is a great deal slower. Notice `c: new Date()` in the evaluated object. In the function without the `eval()`, the object is being evaluated in the global scope, so it is safe for the browser to assume that `Date` refers to `window.Date()` instead of a local variable called `Date`. But, in the code using `eval()`, the browser cannot assume this since what if your code looked like the following:

```
function Date(n){
    return ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday",
}
function looseJsonParse(obj){
    return eval("(" + obj + ")");
}
console.log(looseJsonParse(
   "{a:(4-1), b:function(){}, c:new Date()}"
))
```

Thus, in the `eval()` version of the code, the browser is forced to make the expensive lookup call to check to see if there are any local variables called `Date()`. This is incredibly inefficient compared to `Function()`.

In a related circumstance, what if you actually wanted your `Date()` function to be able to be called from the code inside `Function()`. Should you just take the easy way out and fall back to `eval()`? No! Never. Instead try the approach below.

```
function Date(n){
    return ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturday",
}
function runCodeWithDateFunction(obj){
    return Function('"use strict";return (' + obj + ')')()(
        Date
    );
}
console.log(runCodeWithDateFunction(
   "function(Date){ return Date(5) }"
))
```

The code above may seem inefficiently slow because of the triple nested function, but let's analyze the benefits of the above efficient method:

- It allows the code in the string passed to `runCodeWithDateFunction()` to be minified.
- Function call overhead is minimal, making the far smaller code size well worth the benefit
- `Function()` more easily allows your code to benefit from the possible performance improvements provided by `"use strict";`
- The code does not use `eval()`, making it orders of magnitude faster than otherwise.

Lastly, let's examine minification. With using `Function()` as shown above, you can minify the code string passed to `runCodeWithDateFunction()` far more efficiently because the function arguments names can be minified too as seen in the minified code below.

```
console.log(Function('"use strict";return(function(a){return a(5)})')()(fu
return"Monday Tuesday Wednesday Thursday Friday Saturday Sunday".split(" "
```

There are also additional safer (and faster!) alternatives to `eval()` or `Function()` for common use-cases.

## Accessing member properties

You should not use `eval()` to convert property names into properties. Consider the following example where the property of the object to be accessed is not known until the code is executed. This can be done with `eval()`:

```
var obj = { a: 20, b: 30 };
var propName = getPropName();  // returns "a" or "b"

eval( 'var result = obj.' + propName );
```

However, `eval()` is not necessary here. In fact, its use here is discouraged. Instead, use the property accessors, which are much faster and safer:

```
var obj = { a: 20, b: 30 };
var propName = getPropName();  // returns "a" or "b"
var result = obj[ propName ];  //  obj[ "a" ] is the same as obj.a
```

You can even use this method to access descendant properties. Using `eval()` this would look like:

```
var obj = {a: {b: {c: 0}}};
var propPath = getPropPath();  // returns e.g. "a.b.c"

eval( 'var result = obj.' + propPath );
```

Avoiding `eval()` here could be done by splitting the property path and looping through the different properties:

```
function getDescendantProp(obj, desc) {
  var arr = desc.split('.');
  while (arr.length) {
    obj = obj[arr.shift()];
  }
  return obj;
}

var obj = {a: {b: {c: 0}}};
var propPath = getPropPath();  // returns e.g. "a.b.c"
var result = getDescendantProp(obj, propPath);
```

Setting a property that way works similarly:

```
function setDescendantProp(obj, desc, value) {
  var arr = desc.split('.');
  while (arr.length > 1) {
    obj = obj[arr.shift()];
```

```
  }
  return obj[arr[0]] = value;
}

var obj = {a: {b: {c: 0}}};
var propPath = getPropPath();  // returns e.g. "a.b.c"
var result = setDescendantProp(obj, propPath, 1);  // obj.a.b.c will now be 1
```

## Use functions instead of evaluating snippets of code

JavaScript has first-class functions , which means you can pass functions as arguments to other APIs, store them in variables and objects' properties, and so on. Many DOM APIs are designed with this in mind, so you can (and should) write:

```
// instead of setTimeout(" ... ", 1000) use:
setTimeout(function() { ... }, 1000);

// instead of elt.setAttribute("onclick", "...") use:
elt.addEventListener('click', function() { ... } , false);
```

Closures are also helpful as a way to create parameterized functions without concatenating strings.

## Parsing JSON (converting strings to JavaScript objects)

If the string you're calling `eval()` on contains data (for example, an array: `"[1, 2, 3]"`), as opposed to code, you should consider switching to JSON, which allows the string to use a subset of JavaScript syntax to represent data.

Note that since JSON syntax is limited compared to JavaScript syntax, many valid JavaScript literals will not parse as JSON. For example, trailing commas are not allowed in JSON, and property names (keys) in object literals must be enclosed in quotes. Be sure to use a JSON serializer to generate strings that will be later parsed as JSON.

## Pass data instead of code

For example, an extension designed to scrape contents of web-pages could have the scraping rules defined in XPath instead of JavaScript code.

## Examples

## Examples

### Using `eval`

In the following code, both of the statements containing `eval()` return 42. The first evaluates the string "`x + y + 1`"; the second evaluates the string "`42`".

```
var x = 2;
var y = 39;
var z = '42';
eval('x + y + 1'); // returns 42
eval(z);           // returns 42
```

### Using `eval` to evaluate a string of JavaScript statements

The following example uses `eval()` to evaluate the string `str`. This string consists of JavaScript statements that assigns `z` a value of 42 if `x` is five, and assigns 0 to `z` otherwise. When the second statement is executed, `eval()` will cause these statements to be performed, and it will also evaluate the set of statements and return the value that is assigned to `z`.

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42;} else z = 0;";

console.log('z is ', eval(str));
```

If you define multiple values then the last value is returned.

```
var x = 5;
var str = "if (x == 5) {console.log('z is 42'); z = 42; x = 420; } else z
;'

console.log('x is ', eval(str)); // z is 42  x is 420
```

### Last expression is evaluated

`eval()` returns the value of the last expression evaluated.

```
var str = 'if ( a ) { 1 + 1; } else { 1 + 2; }';
var a = true;
var b = eval(str);  // returns 2
```

```
console.log('b is : ' + b);

a = false;
b = eval(str);  // returns 3

console.log('b is : ' + b);
```

`eval` as a string defining function requires "(" and ")" as prefix and suffix

```
var fctStr1 = 'function a() {}'
var fctStr2 = '(function a() {})'
var fct1 = eval(fctStr1)  // return undefined
var fct2 = eval(fctStr2)  // return a function
```

## Specifications

| Specification |
| --- |
| ECMAScript Language Specification (ECMAScript)<br># sec-eval-x |

## Browser compatibility

Report problems with this compatibility data on GitHub

| `eval` | |
| --- | --- |
| Chrome | 1 |
| Edge | 12 |
| Firefox | 1 |
| Internet Explorer | 3 |
| Opera | 3 |
| Safari | 1 |