Strict mode

> Sometimes you'll see the default, non-strict mode referred to as **"sloppy mode"**. This isn't an official term, but be aware of it, just in case.

ECMAScript 5's strict mode is a way to *opt in* to a restricted variant of JavaScript, thereby implicitly opting-out of "sloppy mode". Strict mode isn't just a subset: it *intentionally* has different semantics from normal code. Browsers not supporting strict mode will run strict mode code with different behavior from browsers that do, so don't rely on strict mode without feature-testing for support for the relevant aspects of strict mode. Strict mode code and non-strict mode code can coexist, so scripts can opt into strict mode incrementally.

Strict mode makes several changes to normal JavaScript semantics:

1. Eliminates some JavaScript silent errors by changing them to throw errors.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.
3. Prohibits some syntax likely to be defined in future versions of ECMAScript.

See transitioning to strict mode, if you want to change your code to work in the restricted variant of JavaScript.

Invoking strict mode

Strict mode applies to *entire scripts* or to *individual functions*. It doesn't apply to block statements enclosed in `{}` braces; attempting to apply it to such contexts does nothing. `eval` code, `Function` code, event handler attributes, strings passed to `WindowTimers.setTimeout()`, and the like are entire scripts, and invoking strict mode in them works as expected.

Strict mode for scripts

To invoke strict mode for an entire script, put the *exact* statement `"use strict";` (or `'use strict';`) before any other statements.

```
1 // Whole-script strict mode syntax
2 'use strict';
3 var v = "Hi! I'm a strict mode script!";
```

This syntax has a trap that has already bitten a major site: it isn't possible to blindly concatenate non-conflicting scripts. Consider concatenating a strict mode script with a non-strict mode script: the entire concatenation looks strict! The inverse is also true: non-strict plus strict looks non-strict. Concatenation of strict mode scripts with each other is fine, and concatenation of non-strict mode scripts is fine. Only concatenating strict and non-strict scripts is problematic. It is thus recommended that you enable strict mode on a function-by-function basis (at least during the transition period).

You can also take the approach of wrapping the entire contents of a script in a function and having that outer function use strict mode. This eliminates the concatenation problem but it means that you have to explicitly export any global variables out of the function scope.

Strict mode for functions

Likewise, to invoke strict mode for a function, put the *exact* statement `"use strict";` (or `'use strict';`) in the function's body before any other statements.

```
1 function strict() {
2   // Function-level strict mode syntax
3   'use strict';
4   function nested() { return 'And so am I!'; }
5   return "Hi!  I'm a strict mode function!  " + nested();
6 }
```

```
function notStrict() { return "I'm not strict."; }
```

Changes in strict mode

Strict mode changes both syntax and runtime behavior. Changes generally fall into these categories: changes converting mistakes into errors (as syntax errors or at runtime), changes simplifying how the particular variable for a given use of a name is computed, changes simplifying `eval` and `arguments`, changes making it easier to write "secure" JavaScript, and changes anticipating future ECMAScript evolution.

Converting mistakes into errors

Strict mode changes some previously-accepted mistakes into errors. JavaScript was designed to be easy for novice developers, and sometimes it gives operations which should be errors non-error semantics. Sometimes this fixes the immediate problem, but sometimes this creates worse problems in the future. Strict mode treats these mistakes as errors so that they're discovered and promptly fixed.

First, strict mode makes it impossible to accidentally create global variables. In normal JavaScript mistyping a variable in an assignment creates a new property on the global object and continues to "work" (although future failure is possible: likely, in modern JavaScript). Assignments which would accidentally create global variables instead throw in strict mode:

```
'use strict';
                        // Assuming a global variable mistypedVariable exis
mistypeVariable = 17; // this line throws a ReferenceError due to the
                        // misspelling of variable
```

Second, strict mode makes assignments which would otherwise silently fail to throw an exception. For example, `NaN` is a non-writable global variable. In normal code assigning to `NaN` does nothing; the developer receives no failure feedback. In strict mode assigning to `NaN` throws an exception. Any assignment that silently fails in normal code (assignment to a non-writable global or property, assignment to a getter-only property, assignment to a new property on a non-extensible object) will throw in strict mode:

```
'use strict';
```

```
'use strict';

// Assignment to a non-writable global
var undefined = 5; // throws a TypeError
var Infinity = 5; // throws a TypeError

// Assignment to a non-writable property
var obj1 = {};
Object.defineProperty(obj1, 'x', { value: 42, writable: false });
obj1.x = 9; // throws a TypeError

// Assignment to a getter-only property
var obj2 = { get x() { return 17; } };
obj2.x = 5; // throws a TypeError

// Assignment to a new property on a non-extensible object
var fixed = {};

Object.preventExtensions(fixed);
fixed.newProp = 'ohai'; // throws a TypeError
```

Third, strict mode makes attempts to delete undeletable properties throw (where before the attempt would simply have no effect):

```
'use strict';
delete Object.prototype; // throws a TypeError
```

Fourth, strict mode prior to Gecko 34 requires that all properties named in an object literal be unique. The normal code may duplicate property names, with the last one determining the property's value. But since only the last one does anything, the duplication is simply a vector for bugs, if the code is modified to change the property value other than by changing the last instance. Duplicate property names are a syntax error in strict mode:

> This is no longer the case in ECMAScript 2015 (bug 1041128).

```
'use strict';
var o = { p: 1, p: 2 }; // !!! syntax error
```

Fifth, strict mode requires that function parameter names be unique. In normal code the last duplicated argument hides previous identically-named arguments. Those previous arguments remain available through `arguments[i]`, so they're not completely inaccessible. Still, this hiding makes little sense and is probably undesirable (it might hide a typo, for example), so in strict mode duplicate argument names are a syntax error:

```
function sum(a, a, c) { // !!! syntax error
  'use strict';
  return a + a + c; // wrong if this code ran
}
```

Sixth, a strict mode in ECMAScript 5 forbids octal syntax. The octal syntax isn't part of ECMAScript 5, but it's supported in all browsers by prefixing the octal number with a zero: `0644 === 420` and `"\045" === "%"`. In ECMAScript 2015 Octal number is supported by prefixing a number with "`0o`". i.e.

```
var a = 0o10; // ES2015: Octal
```

Novice developers sometimes believe a leading zero prefix has no semantic meaning, so they use it as an alignment device — but this changes the number's meaning! A leading zero syntax for the octals is rarely useful and can be mistakenly used, so strict mode makes it a syntax error:

```
'use strict';
var sum = 015 + // !!! syntax error
          197 +
          142;

var sumWithOctal = 0o10 + 8;
console.log(sumWithOctal); // 16
```

Seventh, strict mode in ECMAScript 2015 forbids setting properties on primitive values. Without strict mode, setting properties is simply ignored (no-op), with strict mode, however, a `TypeError` is thrown.

```
(function() {
'use strict';

false.true = '';         // TypeError
```

```
(14).sailing = 'home';     // TypeError
'with'.you = 'far away'; // TypeError

})();
```

Simplifying variable uses

Strict mode simplifies how variable names map to particular variable definitions in the code. Many compiler optimizations rely on the ability to say that variable *X* is stored in *that* location: this is critical to fully optimizing JavaScript code. JavaScript sometimes makes this basic mapping of name to variable definition in the code impossible to perform until runtime. Strict mode removes most cases where this happens, so the compiler can better optimize strict mode code.

First, strict mode prohibits `with`. The problem with `with` is that any name inside the block might map either to a property of the object passed to it, or to a variable in surrounding (or even global) scope, at runtime: it's impossible to know which beforehand. Strict mode makes `with` a syntax error, so there's no chance for a name in a `with` to refer to an unknown location at runtime:

```
'use strict';
var x = 17;
with (obj) { // !!! syntax error
  // If this weren't strict mode, would this be var x, or
  // would it instead be obj.x?  It's impossible in general
  // to say without running the code, so the name can't be
  // optimized.
  x;
}
```

The simple alternative of assigning the object to a short name variable, then accessing the corresponding property on that variable, stands ready to replace `with`.

Second, `eval` of strict mode code does not introduce new variables into the surrounding scope. In normal code `eval("var x;")` introduces a variable `x` into the surrounding function or the global scope. This means that, in general, in a function containing a call to `eval` every name not referring to an argument or local variable must be mapped to a particular definition at runtime (because that `eval` might have introduced a new variable that would hide the outer variable). In strict mode `eval` creates variables only for the code being evaluated, so `eval` can't affect whether a name refers to an outer variable or some local variable:

```
var x = 17;
var evalX = eval("'use strict'; var x = 42; x;");
console.assert(x === 17);
console.assert(evalX === 42);
```

If the function `eval` is invoked by an expression of the form `eval(...)` in strict mode code, the code will be evaluated as strict mode code. The code may explicitly invoke strict mode, but it's unnecessary to do so.

```
function strict1(str) {
  'use strict';
  return eval(str); // str will be treated as strict mode code
}
function strict2(f, str) {
  'use strict';
  return f(str); // not eval(...): str is strict if and only
                 // if it invokes strict mode
}
function nonstrict(str) {
  return eval(str); // str is strict if and only
                    // if it invokes strict mode
}

strict1("'Strict mode code!'");
strict1("'use strict'; 'Strict mode code!'");
strict2(eval, "'Non-strict code.'");
strict2(eval, "'use strict'; 'Strict mode code!'");
nonstrict("'Non-strict code.'");
nonstrict("'use strict'; 'Strict mode code!'");
```

Thus names in strict mode `eval` code behave identically to names in strict mode code not being evaluated as the result of `eval`.

Third, strict mode forbids deleting plain names. `delete name` in strict mode is a syntax error:

```
'use strict';

var x;
delete x; // !!! syntax error
```

```
5
6 eval('var y; delete y;'); // !!! syntax error
```

Making `eval` and `arguments` simpler

Strict mode makes `arguments` and `eval` less bizarrely magical. Both involve a considerable amount of magical behavior in normal code: `eval` to add or remove bindings and to change binding values, and `arguments` by its indexed properties aliasing named arguments. Strict mode makes great strides toward treating `eval` and `arguments` as keywords, although full fixes will not come until a future edition of ECMAScript.

First, the names `eval` and `arguments` can't be bound or assigned in language syntax. All these attempts to do so are syntax errors:

```
1  'use strict';
2  eval = 17;
3  arguments++;
4  ++eval;
5  var obj = { set p(arguments) { } };
6  var eval;
7  try { } catch (arguments) { }
8  function x(eval) { }
9  function arguments() { }
10 var y = function eval() { };
11 var f = new Function('arguments', "'use strict'; return 17;");
```

Second, strict mode code doesn't alias properties of `arguments` objects created within it. In normal code within a function whose first argument is `arg`, setting `arg` also sets `arguments[0]`, and vice versa (unless no arguments were provided or `arguments[0]` is deleted). `arguments` objects for strict mode functions store the original arguments when the function was invoked. `arguments[i]` does not track the value of the corresponding named argument, nor does a named argument track the value in the corresponding `arguments[i]`.

```
1  function f(a) {
2    'use strict';
3    a = 42;
4    return [a, arguments[0]];
5  }
6  var pair = f(17);
```

```
7 console.assert(pair[0] === 42);
8 console.assert(pair[1] === 17);
```

Third, `arguments.callee` is no longer supported. In normal code `arguments.callee` refers to the enclosing function. This use case is weak: simply name the enclosing function! Moreover, `arguments.callee` substantially hinders optimizations like inlining functions, because it must be made possible to provide a reference to the un-inlined function if `arguments.callee` is accessed. `arguments.callee` for strict mode functions is a non-deletable property which throws when set or retrieved:

```
1 'use strict';
2 var f = function() { return arguments.callee; };
3 f(); // throws a TypeError
```

"Securing" JavaScript

Strict mode makes it easier to write "secure" JavaScript. Some websites now provide ways for users to write JavaScript which will be run by the website *on behalf of other users*. JavaScript in browsers can access the user's private information, so such JavaScript must be partially transformed before it is run, to censor access to forbidden functionality. JavaScript's flexibility makes it effectively impossible to do this without many runtime checks. Certain language functions are so pervasive that performing runtime checks has a considerable performance cost. A few strict mode tweaks, plus requiring that user-submitted JavaScript be strict mode code and that it be invoked in a certain manner, substantially reduce the need for those runtime checks.

First, the value passed as `this` to a function in strict mode is not forced into being an object (a.k.a. "boxed"). For a normal function, `this` is always an object: either the provided object if called with an object-valued `this`; the value, boxed, if called with a Boolean, string, or number `this`; or the global object if called with an `undefined` or `null` `this`. (Use `call`, `apply`, or `bind` to specify a particular `this`.) Not only is automatic boxing a performance cost, but exposing the global object in browsers is a security hazard because the global object provides access to functionality that "secure" JavaScript environments must restrict. Thus for a strict mode function, the specified `this` is not boxed into an object, and if unspecified, `this` will be `undefined`:

```
1 'use strict';
2 function fun() { return this; }
3 console.assert(fun() === undefined);
4 console.assert(fun.call(2) === 2);
5 console.assert(fun.apply(null) === null);
```

```
console.assert(fun.call(undefined) === undefined);
console.assert(fun.bind(true)() === true);
```

That means, among other things, that in browsers it's no longer possible to reference the `window` object through `this` inside a strict mode function.

Second, in strict mode it's no longer possible to "walk" the JavaScript stack via commonly-implemented extensions to ECMAScript. In normal code with these extensions, when a function `fun` is in the middle of being called, `fun.caller` is the function that most recently called `fun`, and `fun.arguments` is the `arguments` for that invocation of `fun`. Both extensions are problematic for "secure" JavaScript because they allow "secured" code to access "privileged" functions and their (potentially unsecured) arguments. If `fun` is in strict mode, both `fun.caller` and `fun.arguments` are non-deletable properties which throw when set or retrieved:

```
function restricted() {
  'use strict';
  restricted.caller;    // throws a TypeError
  restricted.arguments; // throws a TypeError
}
function privilegedInvoker() {
  return restricted();
}
privilegedInvoker();
```

Third, `arguments` for strict mode functions no longer provide access to the corresponding function call's variables. In some old ECMAScript implementations `arguments.caller` was an object whose properties aliased variables in that function. This is a security hazard because it breaks the ability to hide privileged values via function abstraction; it also precludes most optimizations. For these reasons no recent browsers implement it. Yet because of its historical functionality, `arguments.caller` for a strict mode function is also a non-deletable property which throws when set or retrieved:

```
'use strict';
function fun(a, b) {
  'use strict';
  var v = 12;
  return arguments.caller; // throws a TypeError
}
fun(1, 2); // doesn't expose v (or a or b)
```

Paving the way for future ECMAScript versions

Future ECMAScript versions will likely introduce new syntax, and strict mode in ECMAScript 5 applies some restrictions to ease the transition. It will be easier to make some changes if the foundations of those changes are prohibited in strict mode.

First, in strict mode, a short list of identifiers become reserved keywords. These words are `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, and `yield`. In strict mode, then, you can't name or use variables or arguments with these names.

```
function package(protected) { // !!!
  'use strict';
  var implements; // !!!

  interface: // !!!
  while (true) {
    break interface; // !!!
  }

  function private() { } // !!!
}
function fun(static) { 'use strict'; } // !!!
```

Two Mozilla-specific caveats: First, if your code is JavaScript 1.7 or greater (for example in chrome code or when using the right `<script type="">`) and is strict mode code, `let` and `yield` have the functionality they've had since those keywords were first introduced. But strict mode code on the web, loaded with `<script src="">` or `<script>...</script>`, won't be able to use `let/yield` as identifiers. Second, while ES5 unconditionally reserves the words `class`, `enum`, `export`, `extends`, `import`, and `super`, before Firefox 5 Mozilla reserved them only in strict mode.

Second, strict mode prohibits function statements, not at the top level of a script or function. In normal mode in browsers, function statements are permitted "everywhere". *This is not part of ES5 (or even ES3)!* It's an extension with incompatible semantics in different browsers. Note that function statements outside top level are permitted in ES2015.

```
'use strict';
if (true) {
  function f() { } // !!! syntax error
```

```
  f();
}

for (var i = 0; i < 5; i++) {
  function f2() { } // !!! syntax error
  f2();
}

function baz() { // kosher
  function eit() { } // also kosher
}
```

This prohibition isn't strict mode proper because such function statements are an extension of basic ES5. But it is the recommendation of the ECMAScript committee, and browsers will implement it.

Strict mode in browsers

The major browsers now implement strict mode. However, don't blindly depend on it since there still are numerous Browser versions used in the wild that only have partial support for strict mode or do not support it at all (e.g. Internet Explorer below version 10!). *Strict mode changes semantics.* Relying on those changes will cause mistakes and errors in browsers which don't implement strict mode. Exercise caution in using strict mode, and back up reliance on strict mode with feature tests that check whether relevant parts of strict mode are implemented. Finally, make sure to *test your code in browsers that do and don't support strict mode*. If you test only in browsers that don't support strict mode, you're very likely to have problems in browsers that do, and vice versa.

Specifications

Specification	Status	Comment
ECMAScript 5.1 (ECMA-262) The definition of 'Strict Mode Code' in that specification.	ST Standard	Initial definition. See also: Strict mode restriction and exceptions

ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'Strict Mode Code' in that specification.	ST Standard	Strict mode restriction and exceptions
ECMAScript Latest Draft (ECMA-262) The definition of 'Strict Mode Code' in that specification.	D Draft	Strict mode restriction and exceptions

See also

- Where's Walden? » New ES5 strict mode support: now with poison pills!
- Where's Walden? » New ES5 strict mode requirement: function statements not at top level of a program or function are prohibited
- Where's Walden? » New ES5 strict mode support: new vars created by strict mode eval code are local to that code only
- JavaScript "use strict" tutorial for beginners.
- John Resig - ECMAScript 5 Strict Mode, JSON, and More
- ECMA-262-5 in detail. Chapter 2. Strict Mode.
- Strict mode compatibility table
- Transitioning to strict mode