What is Strict Mode in JavaScript? Explained with Examples

What is Strict Mode in JavaScript?

Strict mode is a feature in JavaScript that was introduced in ECMAScript 5. It lets you write code in such a way that follows stricter rules.

When strict mode is enabled, the JavaScript engine enforces additional constraints. This means you may be able to catch some common errors that would have otherwise gone unnoticed. It also helps you write cleaner and more secure code.

How to Use Strict Mode in JavaScript

You can use JavaScript strict mode in two ways. You can either enable strict mode for an entire JavaScript file, or you can enable it within the scope of a function.

To enable strict mode for the whole JavaScript file, you simply add the string `"use strict"` to the top of your code.

```
"use strict"

// Your code goes here...
```

For functions, you can enable strict mode by placing the `"use strict"` string at the top of your function body.

```
function myFunction() {
  "use strict"
  // Function body goes here...

}
```

You must place the `"use strict"` string at the top of the file or function body. Otherwise, it will not work.

But you are allowed to add comments above the `"use strict"` string. Apart from comments, everything else should be below `"use strict"` for it to work.

Difference Between Strict Mode and Regular JavaScript

Now, let's see some of the rules that strict mode enforces which aren't enforced when writing code in regular JavaScript.

Using an Undeclared Variable

When writing JavaScript code in strict mode, you must declare all variables and objects before use. This is useful because it helps to prevent creating global variables by accident which can lead to bugs.

Here's an example

```
// Regular JavaScript

function regularFunc() {
  username = "Marie"
  console.log(username)
}

regularFunc()
```

```
Marie
```

Regular JS throws no error for the username variable.

```
// Strict mode

function strictFunc() {
  "use strict"
  username = "Marie"
  console.log(username)
}

strictFunc()
```

Strict mode throws error for undeclared variables.

In the `regularFunc` function, the browser logs the `username` to the console without any error.

But with the `strictFunc` function which uses strict mode, it throws a reference error.

That's because the `username` variable hasn't been declared. And strict mode does not allow the use of variables before they are declared.

Duplicating a Parameter Name

In non-strict mode, duplicating a parameter name in a function is allowed. Only the last instance of the duplicates is allowed whiles the others are ignored.

But strict mode functions do not allow this. It throws a syntax error in the case of duplicates.

Here's an example:

```
function addNums (num, num) {
  console.log(num + num)
}

addNums(2, 3) // 6
```

In the example above, the first `num` parameter is ignored. Only the second one is used. This why the functions returns 6 because 3 + 3 is 6.

But when you define the same parameters in a function using strict mode, you cannot run it because it will throw a syntax error. Strict mode requires each parameter to have a unique name.

That is:

```
function addNums (num, num) {
  "use strict"
  console.log(num + num)
}
```

```
addNums(2, 3)
```

```
Uncaught SyntaxError: Duplicate parameter name not
allowed in this context (at index.js:3:24)
```

Browser throws an error for duplicate parameter name.

Using Reserved Future Keywords

In JavaScript, there are some keywords reserved for potential future use. And using these keywords as identifiers (such as variables or function names) may likely cause issues in future.

For example, `package` is one of such keywords. When you're not using strict mode, you can use it to name variables and functions.

```
const package = "This is a package"
console.log(package)
```

```
This is a package
```

Log result for using reserved keyword in regular JavaScript.

Strict mode doesn't allow the use of these reserved keywords. This ensures compatibility with future versions of JavaScript.

```
"use strict"
const package = "This is a package"
console.log(package)
```

```
Uncaught SyntaxError: Unexpected strict mode reserved
word (at index.js:3:7)
```

Using a reserved keyword in strict mode results in a syntax error.

Other future reserved keywords include:

- `implements`
- `interface`
- `private`
- `protected`
- `public`
- `static`
- `yield`
- `arguments`

Use of Deprecated Features

Strict mode restricts the use of JavaScript's deprecated features like `arguments.caller`, `arguments.callee`, and so on.

These are disallowed due to security and performance concerns. But non-strict mode allows using them.

Let's see an example of using `arguments.caller` in both strict and non-strict mode.

```
// non-strict mode

function outerFunction() {
  innerFunction();
}

function innerFunction() {
  console.log(innerFunction.caller); // Example of .caller
}

outerFunction();
```

```
f outerFunction() {
  innerFunction();
}
```

`arguments.caller` works in non-strict mode without any errors.

Using `.caller` on the `innerFunction` logs the `outerFunction` which calls the inner function to the console. It works alright in non-strict mode.

But using it in strict mode will cause JavaScript to throw an error like in the example below.

```
"use strict"

function outerFunction() {
  innerFunction();
}

function innerFunction() {
  console.log(innerFunction.caller); // Example of .caller
}

outerFunction();
```

```
▶ Uncaught TypeError: 'caller', 'callee', and 'arguments' index.js:8
properties may not be accessed on strict mode functions or the
arguments objects for calls to them
    at innerFunction (index.js:8:29)
    at outerFunction (index.js:4:3)
    at index.js:11:1
```

Strict mode throws an "Uncaught TypeError" for using deprecated features.

Assignment to a Read-Only Property

In non-strict mode, when you attempt to assign a new value to a read-only property, the assignment will not modify the property's value. But it will not throw any error. Instead, the assignment silently fails, and the property retains its original value.

```
const obj = {}

Object.defineProperty(obj, 'key', { value: 10, writable: false })
obj.key = 20

console.log(obj.key)
```

```
10
```

Log result for `object.key`

This example defines a new empty object `obj`. And using `Object.defineProperty`, a new property `key` is added to `obj`. This `key` property is set as a read-only property.

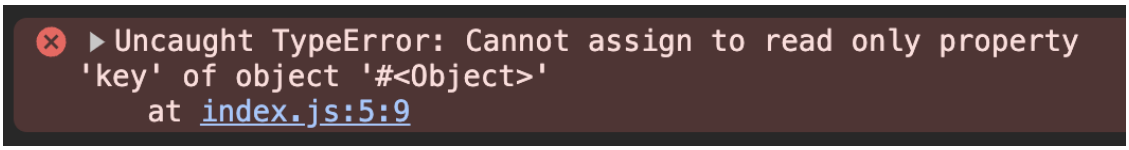
As you can see from log result, there is no error when you try to update the value of `key`. And the `key` property maintains the original value of `10`.

In strict mode, however, attempting to update the value of the property will result in a `TypeError`. This stricter enforcement helps catch potential errors early.

```
"use strict"
const obj = {}

Object.defineProperty(obj, 'key', { value: 10, writable: false })
obj.key = 20

console.log(obj.key)
```



Strict mode throws an error when assigning a new value to read-only property.

JavaScript Features that Use Strict Mode by Default

Some JavaScript features do not require you to explicitly invoke `"use strict"`. By default, strict mode is applied to prevent common errors and ensure compatibility with future JavaScript versions.

Examples of such contexts include the following:

- ES6 classes
- ES6 modules
- Arrow functions
- Tagged template literals