Top 10 Tips to write better JavaScript Code

```
const numbers = [1, 2, 3, 4];

// Using a for loop
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}

// Using forEach
numbers.forEach((number) => {
  console.log(number);
});
```


Using == instead of === 4
Not using const and let 4
Not using semicolons 6
Not handling errors 7
Not using strict mode 7
Not using curly braces 8
Not using Array.prototype.forEach() 8
Not using const when defining functions 9
Not using the let keyword in for loops 10
Not using === or !== with null or undefined 11

Not declaring variables.

When a variable is not declared, it is considered a global variable. This means that it can be accessed from anywhere in the code, which can lead to confusion and errors. To avoid this mistake, always declare variables using the `var` keyword. For example:

Code snippet

```
var myVar = 1;
```

Use code with caution. Learn more

copy

2. Using global variables.

As mentioned above, global variables can be accessed from anywhere in the code, which can lead to confusion and errors. It is best to avoid using global variables whenever possible. If you do need to use a global variable, make sure to give it a descriptive name so that you know what it is used for.

3. Not using strict mode.

Strict mode is a JavaScript feature that helps to prevent errors and security vulnerabilities. It is enabled by using the `"use strict";` statement at the top of your code. For example:

Code snippet

```
"use strict";
```

Use code with caution. Learn more

copy

4. Not handling errors properly.

All JavaScript code should be written to handle errors gracefully. This means that you should have a plan for what to do if an error occurs. For example, you could log the error to the console, display a message to the user, or try to recover from the error.

5. Using synchronous functions in an asynchronous context.

Synchronous functions are functions that block the execution of the code until they are finished. Asynchronous functions, on the other hand, do not block the execution of the code. It is important to use the correct type of function in the correct context. For example, you should not use a synchronous function in an asynchronous context, such as an event handler.

6. Not using the right data types.

JavaScript has a variety of data types, and it is important to use the right data type for the job. For example, you should not use a string to represent a number.

7. Not using comments.

Comments are a great way to explain what your code is doing. They can also be used to disable code temporarily.

8. Not using indentation.

Indentation makes your code more readable. It is a good practice to use indentation consistently.

9. Not testing your code.

It is important to test your code regularly to make sure that it is working correctly. There are a variety of tools available to help you test your code.

10. Not learning from your mistakes.

Everyone makes mistakes when they are coding. The important thing is to learn from your mistakes and avoid making them again.

Here are some additional tips for avoiding common JavaScript coding mistakes:

- Read the documentation. The documentation for JavaScript is very comprehensive and can help you avoid making mistakes.
- Use a linter. A linter is a tool that can help you find potential errors in your code.
- Ask for help. If you are stuck on something, don't be afraid to ask for help from a more experienced developer.

By following these tips, you can avoid common JavaScript coding mistakes and write better code.

More common Coding Mistakes with JavaScript

Using == instead of ===

Using == instead of === can lead to unexpected behavior in your code. The == operator does type coercion, which means it converts the operands to a common type before comparing them. This can lead to unintended results, especially when comparing different types.

For example:

```
console.log(1 == '1'); // true

console.log(1 === '1'); // false
```

To avoid this mistake, always use === for strict equality comparison. This operator compares both value and type, so it will only return true if the operands are of the same type and have the same value.

Not using const and let

Declaring variables with var can lead to unexpected behavior in your code because var has function scope, which means the variable is accessible anywhere within the function, even outside of the block it was declared in. This can lead to bugs and make it harder to reason about your code.

Instead, use const and let for block-scoped variables. const should be used for variables that won't be reassigned, while let should be used for variables that will be reassigned.

For example:

```
// Using var

function foo() {

  var x = 1;

  if (true) {

    var x = 2;

  }
```

```
  console.log(x); // 2
}


// Using let
function bar() {
  let x = 1;
  if (true) {
    let x = 2;
  }
  console.log(x); // 1
}


// Using const
const PI = 3.14;
```

Not using semicolons

JavaScript automatically inserts semicolons in some cases, but relying on this can lead to bugs and unexpected behavior. It's best practice to always use semicolons to avoid these issues.

For example:

```
// Without semicolons

const x = 1

const y = 2



// With semicolons

const x = 1;

const y = 2;
```

Not handling errors

Failing to handle errors can cause your program to crash or behave unexpectedly. Always make sure to handle errors properly with try-catch blocks or error callbacks.

For example:

```
try {

  // Some code that might throw an error

} catch (err) {

  console.error('An error occurred:', err);

}
```

Not using strict mode

Strict mode enforces stricter rules on your code, catching common mistakes and preventing bad practices. Always use strict mode in your code to catch errors early and improve code quality.

To enable strict mode, add the following statement at the beginning of your JavaScript file or function:

```
'use strict';
```

Not using curly braces

Using curly braces in your code can make it easier to read and understand. Always use curly braces for blocks, even if they contain only one statement.

For example:

```
// Without curly braces

if (true)

  console.log('Hello');



// With curly braces

if (true) {

  console.log('Hello');

}
```

Not using Array.prototype.forEach()

Iterating over arrays with for loops can be error-prone and hard to read. Use Array.prototype.forEach() instead to simplify your code.

For example:

```
const numbers = [1, 2, 3, 4];

// Using a for loop

for (let i = 0; i < numbers.length; i++) {

  console.log(numbers[i]);

}



// Using forEach

numbers.forEach((number) => {

  console.log(number);

});
```

Not using const when defining functions

When defining functions, always use const to ensure the function is not accidentally reassigned or modified elsewhere in your code.

For example:

```
// Using const to define a function

const add = (a, b) => {

  return a + b;

};



// Using let to define a function (avoid)

let subtract = function(a, b) {

  return a - b;

};
```

Not using the let keyword in for loops

Using var instead of let in for loops can lead to unexpected behavior because var has function scope, meaning the variable is accessible outside of the loop. This can cause issues when reusing the variable in other parts of your code.

Instead, use let to declare the variable in the for loop, making it block-scoped and preventing it from being accessed outside of the loop.

For example:

```
// Using var in a for loop

for (var i = 0; i < 5; i++) {

  console.log(i);

}

console.log(i); // 5



// Using let in a for loop

for (let j = 0; j < 5; j++) {

  console.log(j);

}

console.log(j); // ReferenceError: j is not defined
```

Not using === or !== with null or undefined

When checking for null or undefined, always use the strict equality operators (=== and !==) to avoid unexpected behavior.

For example:

```
// Using == with null or undefined

console.log(null == undefined); // true
```

```
console.log(null == 0); // false

console.log(undefined == 0); // false



// Using === with null or undefined

console.log(null === undefined); // false

console.log(null === 0); // false

console.log(undefined === 0); // false



// Checking for null or undefined

const value = null;

if (value === null || value === undefined) {

  console.log('Value is null or undefined');

}
```

By avoiding these common coding mistakes, you can write more reliable and maintainable JavaScript code.