JavaScript Best Practices

Avoid global variables, avoid new, avoid ==, avoid eval()

Avoid Global Variables

Minimize the use of global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

Use local variables instead and learn how to use closures.

Always Declare Local Variables

All variables used in a function should be declared as local variables.

Local variables must be declared with the var, the let, or the const keyword, otherwise they will become global variables.

Strict mode does not allow undeclared variables.

Declarations on Top

It is a good coding practice to put all declarations at the top of each script or function.

This will:

- **Give cleaner code**
- **Provide a single place to look for local variables**
- **Make it easier to avoid unwanted (implied) global variables**
- **Reduce the possibility of unwanted re-declarations**

```
// Declare at the beginning
let firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";
```

price = 19.90;
discount = 0.10;

fullPrice = price - discount;

This also goes for loop variables:

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

Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

- **Give cleaner code**
- **Provide a single place to initialize variables**
- **Avoid undefined values**

```
// Declare and initiate at the beginning
let firstName = "";
let lastName = "";
let price = 0;
let discount = 0;
let fullPrice = 0,
const myArray = [];
const myObject = {};
```

Initializing variables provides an idea of the intended use (and intended data type).

Declare Objects with const

Declaring objects with const will prevent any accidental change of type:

Example

```
let car = {type:"Fiat", model:"500", color:"white"};
car = "Fiat";     // Changes object to string
const car = {type:"Fiat", model:"500", color:"white"};
car = "Fiat";     // Not possible
```

Declare Arrays with const

Declaring arrays with const will prevent any accidential change of type:

Example

```
let cars = ["Saab", "Volvo", "BMW"];
cars = 3;    // Changes array to number


const cars = ["Saab", "Volvo", "BMW"];
cars = 3;    // Not possible
```

Don't Use new Object()

- **Use "" instead of new String()**
- **Use 0 instead of new Number()**
- **Use false instead of new Boolean()**
- **Use {} instead of new Object()**
- **Use [] instead of new Array()**
- **Use /()/ instead of new RegExp()**
- **Use function (){} instead of new Function()**

Example

```
let x1 = "";          // new primitive string
let x2 = 0;           // new primitive number
let x3 = false;       // new primitive boolean
const x4 = {};        // new object
const x5 = [];        // new array object
const x6 = /()/;      // new regexp object
const x7 = function(){}; // new function object
```

Beware of Automatic Type Conversions

JavaScript is loosely typed.

A variable can contain all data types.

A variable can change its data type:

Example

```
let x = "Hello";    // typeof x is a string
x = 5;              // changes typeof x to a number
```

Beware that numbers can accidentally be converted to strings or NaN (Not a Number).

When doing mathematical operations, JavaScript can convert numbers to strings:

Example

```
let x = 5 + 7;      // x.valueOf() is 12,  typeof x is a number
let x = 5 + "7";    // x.valueOf() is 57,  typeof x is a string
let x = "5" + 7;    // x.valueOf() is 57,  typeof x is a string
let x = 5 - 7;      // x.valueOf() is -2,  typeof x is a number
let x = 5 - "7";    // x.valueOf() is -2,  typeof x is a number
let x = "5" - 7;    // x.valueOf() is -2,  typeof x is a number
let x = 5 - "x";    // x.valueOf() is NaN, typeof x is a number
```

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number):

Example

```
"Hello" - "Dolly"    // returns NaN
```

Use === Comparison

The == comparison operator always converts (to matching types) before comparison.

The === operator forces comparison of values and type:

Example

```
0 == "";      // true
1 == "1";     // true
1 == true;    // true

0 === "";     // false
1 === "1";    // false
1 === true;   // false
```

Use Parameter Defaults

If a function is called with a missing argument, the value of the missing argument is set to undefined.

Undefined values can break your code. It is a good habit to assign default values to arguments.

Example

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 0;
  }
}
```

ECMAScript 2015 allows default parameters in the function definition:

```
function (a=1, b=1) { /*function code*/ }
```

End Your Switches with Defaults

Always end your switch statements with a default. Even if you think there is no need for it.

Example

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday";
    break;
  case 1:
    day = "Monday";
    break;
  case 2:
    day = "Tuesday";
    break;
  case 3:
    day = "Wednesday";
    break;
  case 4:
    day = "Thursday";
    break;
  case 5:
    day = "Friday";
    break;
  case 6:
    day = "Saturday";
    break;
  default:
    day = "Unknown";
}
```

Avoid Number, String, and Boolean as Objects

Always treat numbers, strings, or booleans as primitive values. Not as objects.

Declaring these types as objects, slows down execution speed, and produces nasty side effects:

Example

```
let x = "John";
let y = new String("John");
(x === y) // is false because x is a string and y is an object.
```

Or even worse:

Example

```
let x = new String("John");
let y = new String("John");
(x == y) // is false because you cannot compare objects.
```

Avoid Using eval()

The eval() function is used to run text as code. In almost all cases, it should not be necessary to use it.

Because it allows arbitrary code to be run, it also represents a security problem.