Best Practices

Contents

Overview

JavaScript is interesting in that it enforces no particular structure upon us.

"Bring your own organisation", if you will. How we structure JavaScript is very important because:

- Done right, it makes code easier to understand for others and yourself re-visiting your own code.
- Having a decided-upon structure helps keep future code clean and encourages the declared best practices.
- It makes code testable.

Tutorial

Variables

For variable names use **lowerCamelCasing**, human-readable, semantic names where appropriate:

```
let playerScore =0;

let speed = distance / time;
```

As opposed to:

```
let thisIsAVeryLongVariablePlaceHolderForPlayersScoresHEHEHE = 0;
let s = d/t;
```

Declaring variables

When declaring variables and constants, use the `let` and `const` keywords, NOT `var`.

If a variable will not be reassigned, use `const`:

```
const myName = 'Savannah';
console.log(myName);
```

Otherwise, use `let`:

```
let myAge = '22';
myAge++;
console.log('Happy Birthday!');

// What happens if CONST is used here?
const myAge = '22';
myAge++;
console.log('Happy Birthday!');

// The reassignment will throw an error - you cannot reassign a constant.
```

At all cost, avoid `var` unless really needed - it hoists the variable and is much more trouble to deal with.

Operators and comparison

Ternary Operators

Ternary operators should be put on a single line:

```
let status = (age >= 18) ? 'adult' : 'minor';
```

Not nested:

```
let status = (age >= 18)
? 'adult'
: 'minor';
```

We can agree that the nested version is much harder to read.

Strict Equality

Always use strict equality and inequality:

```
name === 'Savannah'
age !== 25
```

Not this:

```
name == 'Savannah'
age != 25;
```

Control Statements

- There should be *no space* between a control statement keyword and its opening parenthesis.
- There should be *a space* between the parenthesis and the opening curly brace.

Write control statements like this:

```
if(icecream) {
    alert('Woooooooohoo!');
}
```

Not this:

```
if (icecram){
    alert('Woooooooohoo!');
}
```

Statements

Statements are syntax constructs and commands that perform actions.

We can have as many statements in our code as we want.

Statements can be separated with a semicolon (`;`).

For example, here we split "Hello World" into two alerts:

```
alert("Hello");alert("World");
```

Usually, statements are written on separate lines to make the code more readable:

```
alert("Hello");
alert("World");
```

Semicolons

A semicolon may be omitted in most cases when a line break exists.

This would also work:

```
alert("Hello")
alert("World")
```

Here, JavaScript interprets the line break as an "implicit" semicolon. This is called automatic semicolon insertion.

In most cases, a newline implies a semicolon. But 'in most cases' doesn't mean 'always'!

There are cases when a new line doesn't mean a semicolon. For example:

```
alert(3+
1
+2);
```

The code outputs `6` because JavaScript does not insert semicolons here.

It is intuitively obvious that if the line ends with a `'+'` then it is an "incomplete expression", so the semicolon is not required.

And in this case that works as intended. However, there are situations where JavaScript 'fails' to assume a semicolon where it is really needed.

Errors which occur in such cases are quite hard to find and fix.

```
[1,2].forEach(alert)
```

This will output `1` followed by `2`.
Now lets add an `alert` before the code and *not* finish it with a semicolon:

```
alert("There will be an error")

[1,2].forEach(alert)
```

Now, if we run the code, only the first alert is shown and then we have an error!
But everything is fine again if we add a semicolon after `alert`:

```
alert("All good now");

[1,2].forEach(alert)
```

Now, the result is `"All good now"` followed by `1` and `2`.

The error in the no-semicolon variant occurs because JS does not assume a semicolon before the square brackets. So, because the semicolon is not auto-inserted, the code in the first example is treated as a single statement. This is how it is interpreted:

```
alert("There will be an error")[1,2].forEach(alert)
```

It is recommended that semicolons are inserted between statements even if they are separated by new lines.

Comment

As time goes on, programs become more and more complex. It becomes prudent to add *comments* which describe what the code does and why.

Comments can be placed anywhere. They do not affect its execution because the engine simply ignores them.

One-line Comments

One line comments start with two forward slash characters `//`.

The rest of the line is a comment, it may occupy a full line of its own or follow a statement:

```
// Example 1 - this is a comment of its own.
alert("Hello");

alert("World"); // Example 2 - This comment follows a statement
```

Multiline Comments

Multiline comments start with a forward slash and an asterisk `/*` and end with an asterisk followed by a forward slash `*/`:

```
/* This is an example with a multi-line comment,
I can continue here.
And here.
And here.
All good :)
*/
alert("Hello");
alert("World");
```

Strings

When writing a string in JavaScript, you can use double quotes `("")`, single quotes `('')` or even back-ticks ``(``)``.

All of them are okay to use, as long as it is consistent through out the JavaScript document.

The multitude of choice is beneficial as it allows for use of the character within a string:

```
console.log("I can use an apostrophe here because I'm using double quotes on the
outside");
```

The alternative would be to use the escape character `(\)` within a string:

```
console.log(' \'Woof\' - Sir Barks A Lot ');
```

For inserting values into string, use string/template literals:

```
let myName = 'Savannah';
console.log(`Hi! I'm ${myName}`);
```

Defining Functions

Where possible use the function declaration to define functions over function expressions:

```
function sum(a,b) {
    return a+b;
}
```

This is suboptimal by comparison:

```
let sum = function(a,b){
    return a+b;
}
```

When using anonymous functions inside a method that requires a function as parameter, it is acceptable (although not required) to use an arrow function to make the code shorter and cleaner:

```
const array1 = [1,2,3,4,5];
let sum = array.reduce((a,b) => a+b);
```

By comparison, this is suboptimal:

```
const array1 = [1,2,3,4,5];
let sum = array.reduce(function(a,b) {
    return a+b;
});
```

In summary, ensure standards are followed, and adhered to across the organisation. For more information, look at the Mozilla Code Guidelines

Exercises

There are no exercises for this module.