JAVASCRIPT

BEST PRACTICES TO PROGRAMMING CODE WITH JAVASCRIPT

Learn to Write Effective Programming Code in JavaScript!

CHARLIE MASTERSON

JavaScript:

Best Practices
to Programming Code
with JavaScript

Charlie Masterson

Table of Contents

Introduction

Welcome to our book, "JavaScript: Best Practices to Programming Code with JavaScript". Whether you are a beginner with coding or have been doing it for a while, you know how hard JavaScript can be! Even the simplest mistakes can mess up your entire code! We are here to tell you that it happens with everyone! You are not alone in this situation. It is how you handle these mistakes that could differentiate you from the other coders.

Do you just give up because of one mistake? Of course not! Life is all about making mistakes, and learning from them. Within these chapters, we will be teaching you some basic practices to enhance your skills within JavaScript. We will be giving you the best methods, processes, and techniques that will help lead to a better JavaScript programming code. As you learn, you will be able to apply your new skills to everyday life.

Our goal is to have our readers proficient in JavaScript code, so they can expand their knowledge by reading this book. If you are at this point, we expect that you have some previous knowledge of JavaScript, even if it is just the basics. Sometimes, JavaScript books can be frustrating. The code is not always an easy topic to explain clearly. We promise to make this coding process as simple as possible.

For our first chapter, we will be going into depth on some of the common JavaScript mistakes every coder has done. You may find that you aren't the only one making some of these errors. In fact, some seem silly to be messing up! Don't worry; sometimes it is the smallest mistakes that are the easiest to fix! Perhaps you are skipping attention to detail such as mismatching quotes or even changing your variable names. Were you aware that properties and variable names were case sensitive? If not, we have got you covered!

After the common mistakes, we will go over some general recommendations to help improve your code. For example, we will be going over how to define your local vars, how to avoid global variables and more! Once we have discussed the general recommendations, we move onto tips to improve your comments, logging, and overall performance. As we stated before, our primary goal is to develop your skills as a coder.

As you read, please feel free to stop and practice the codes we provide. After all, practice makes perfect! Eventually, coding will become more natural for you and you won't have to stop and think about the code as much. Of course, errors are bound to happen, but as you learn you may find yourself making less of them. We hope that you enjoy the read and will learn a thing or two while you're at it!

Chapter 1:
Common JavaScript Mistakes

Yes, you probably know all of the codes to follow. Just remember that coding in JavaScript can trip you up, no matter how long you've been working with code. The following examples are some of the common traps people fall into. When you are working, try to keep these models in your mind. This way, when you are coding or debugging, you will catch yourself sooner!

Mismatching Quotes, Curly Brace, and Parenthesis

If you do this a lot, there is an easy way to avoid the mistake. All you will need to do is to code your opening and closing elements at the same time! After you do this, you can enter your element that you want to add to your code. Here is an example:

```
CODE

var myString=" ";
function myFunction(){
    if(){

                        }
  }
```

- First line, you want to code the quotes before you even enter your string value. This way, you don't forget them.
- In your second line, close every bracket as soon as you open it.
- Within your code, be sure you double check and count the left and right parenthesis within your code. You will want to double check and make sure that they are equal
- Finally, if you open a parenthesis, be certain you close it!

Case Sensitive

In case you aren't already aware, functions and variables are both case sensitive. While you are working, it can be VERY easy to make this silent error. When you are picking names, we suggest you stick with it. As a small tip, just remember that the CSS properties within JavaScript are going to be camelCase. Here's a quick example:

```
getElementById("theId") != getElementByID("theId");
```

- Did you catch the mistake? Instead of "ID" it should have been "Id"

```
getElementById("theId") != getElementById("theID");
```

- What about this time? You got it! "Id" will not equal "ID." You can see how this is a common mistake, especially if you are trying to move quickly!

Line Breaks and Extra Commas

When you are coding, you will want to stay away from hard-line breaks. When you use line

breaks, they will be interpreted as a line-ending semicolon. This can also happen in a string. When you include a hard line break in your quotes, you are going to get an error. It will most likely be the 'unterminated string' parse error. Here is an example:

- Unterminated String Error:

```
var bad= <ul id="myId">
    <li>writing the text</li>
    <li>even more text </li>
    </ul>;
```

- Correct Line Break

```
var good= '<ul id="myId">' +
    '<li> writing the text </li>' +
    '<li> even more text </li>' +
    '</ul>';
```

- It should be noted that this semi-colon rule will not hold true with control structures. In control structures, a line break AFTER your closing parenthesis is not given by a semi-colon.
- To fix this mistake, you will want always to use semi-colons and parenthesis. This way, your code is easier to read, you will not get tripped up by the breaking lines, and when you move your code around, you will never have to worry about your initial statement closing.
- If you have extra commas, remember that your last property should not end with a comma. While some browsers won't be tripped up by the extra commas (such as Firefox), Internet explorer will. Just try and remember that it is better to be safe than sorry when you are working with code!

Conditional Statements

When you are coding, there are three common conditional statement mistakes. Here are some of the following:

1. When you are coding, try not to get tripped up by using an assignment operator. When you do this, you are conveying your second argument value to your first. Obviously, this will come back as a logic issue. When this happens, it will return true and will not throw an error if value in var2 is not zero.

   Example:

   ```
   if(var1 = var2) {}
   ```

   - This will return true or false, depends on value in var2. It assigns var2 to var1. You have to be careful in case this is something you did not mean!

2. Remember that all of your conditional comments have to be within parentheses. This may seem like a simple mistake. However, you would be surprised just how common it is!

   Example:

```
if(var1 == var2) {}
```

3. For most cases, JavaScript is loosely typed. This is not the fact when you switch statements. Your JavaScript will not be loosely typed when you use case comparisons.

   Example:

```
var myVar = 1;
if(myVar == '1') {
alert ("hey");

}
switch(myVar) {
case '1' :
alert("hey");
```

   - In the first line, your code will return true because JavaScript is loosely typed
   - As for your alert in the third lime, it will show because most of the time, JavaScript will not care about the data type
   - In the last line, your alert will NOT show because your data types will not match.
   - However, if you want to make javascript also check the data type, instead of two equals signs, you should use three (===). In this case, this will not alert:

```
var myVar = 1;
if(myVar === '1') {
alert ("hey");
}
```

Variable Scope

Not surprising, there are quite a few issues that can come from the variable scope. Some of these issues include overwriting the global variable from a local variable within your function or even thinking that a local variable is incorrectly a global variable. As you will be learning later, it is best just not to use any global variables. If you can't avoid them, at least know how to avoid them.

When you have global variables, remember that they will not be declared by the typical var keyword. Instead, you must declare your variables with the key term to keep them from having a global scope. Here is an example of a variable having a declared function and in return, getting a global scope:

CODE:

```
anonymousFunction1 = function(){
```

```
globalvar= 'global scope';
        return globalvar;
 } ();

 alert(globalvar);
```

- In this code, the "var" is missing, when this happens, globally will be declared.

CODE:

```
anonymousFunction2 = function(){
    var localvar = 'local scope' ;
    return localvar;
} ();

 alert(localvar);
```

- In this code, there is a locally declared "var"
- Last, there will be an error of "localvar is not defined." As you can see, this is because there is no globally defined localvar.

HTML id Conflicts

This little mistake is more of a best practice for CSS, but sometimes it can be helpful if you are having a JavaScript issue. An HTML id allows JavaScript to have DOM bindings to allow indexing. As you will recall, both properties and functions share a namespace in your coding. So, if your properties and function happen to have the same name, you may be creating a logic error. These errors are hard to track down! When you are using JavaScript to code, we suggest avoiding variable names that are also ID values. Also, if you are marking your HTML, try not to use a property name as a value. This will help avoid confusion.

Here is an example:

CODE:

```
var listitems= document.getElementsByTagName('a');
var aCount= listitems.length;
```

- If you happened to have <a id="length">, it would sometimes return as <a> instead of the count you asked for. This is why it is very important to remember this little tip.

String Replace

This string replace happens to be another very common mistake in the coding world. You may have made this mistake of assuming the behavior of your string replace method. When you make assumptions, it could impact all of your possible matches. If you weren't already aware, JavaScript will only change the first occurrence in your string replace method. If you want to replace all of your occurrences, it should be noted that you will need first to set a global modifier.

Here is a good example:

CODE:

```
var myString = "the string were talking about";
myString= myString.replace(/ /, "%10"); //
"the%10string were talking about"
myString = myString.replace(/ /g,"%10"); //
"the%10string%10were%10talking%10about"
```

‘this’

If you are used to using JavaScript for coding, you may have made the mistake of forgetting to use the ‘this’. If this isn’t ringing a bell for you, the functions in JavaScript are defined as object accessing properties. When you fail to use the ‘this’ identifier, things may get lost.

Here is an incorrect example:

```
function myFunction () {
var myObject = {
objProperty: " the text you are using",
objMethod: function () {
                    alert (objProperty);
                    }
};
myObject. objMethod ();
}
```

Here is the correct code when you use ‘this’:

```
function myFunction () {
var myObject = {
objProperty: " the text you are using",
objMethod: function () {
                    alert (this.objProperty);
                    }
};
myObject. objMethod ();
}
```

Missing Parameter

When you are coding, remember that you will want to update all of your function calls. This happens when you add the parameter to your function. Sometimes, people miss this and cause themselves some simple errors. Remember that if you do need to add a parameter to a special case, you will have to first set a default value for the parameter first for your function. This way, you won't miss updating your calls in any of your scripts.

Here is an example:

```
function addressFunction ( city, state, country){
                Country = country || "London" ;
    }
```

Overwriting/ Overloading Functions

When you are coding, if you declare a function more than once, your last declaration of the same function will overwrite any of the previous functions. This can be difficult because it will not give you any warnings or sign of error. In JavaScript, there is no overloading. This is why it is important to avoid any core names of JavaScript in your code. You will also want to avoid including JavaScript files in excess. If this happens, it could overwrite the same function in another script. Instead, we suggest using namespaces and functions instead, here is an example:

CODE:

```
// the creation of the namespace

// if the namespace does not exist, create it.

    if (!window.THENAMESPACE) {

    window [ 'THENAMESPACE' ] = {};                    }

// note that this function is only accessible in the anonymous function

    function myFunction (var1, var2){

//the local function code will go here }

    /* This will attach the local function to your namespace. This meaning that it will be accessible outside of your anonymous function when it uses the namespace*/

    window['THENAMESPACE']['myFunction'] = myFunction;

    };
//Note that the parenthesis will open an immediate execution //
```

- It is the parenthesis that encompasses the code that will make your anonymous function.

Using ‘For Each’ Correctly

As you will recall, JavaScript uses the "for" loop for your object attributes, properties, and methods. It uses the code to loop all your property names in an object. This loop will also include the functions and properties that you are not interested in having in your code. If you filter out these values, you will want to use the "hasOwnProperty" or the "typeof" function. While coding, we suggest staying away from the "each"; the only time you will want to use "each" is when you want to iterate methods and object properties.

Here are a few tips when coding:

- for (var myVar in myObject)The code from above is meant to iterate a specific variable in the properties of a specific object. This will not matter what order it is in. The for…in loop will not iterate the built-in properties.

- for (var i=0; i <myArray.length; i++)
  This code should be used for all elements of your array. It will iterate thru all of the above.
- for each (var myVar in myObject)
  This code is meant for specific variables in all of your object's properties
- If you wish to solve this issue, you will want to use the for…in if you are using it for objects. When you want to loop your array, just use for. Here is an example:
- listItems=document.getElementsByTagName('La');
  for each (var listitem in listItems){
  }
- Note: The code from above will go thru all of the methods of the object and all of the properties
- This will include any native properties and methods. Please also note that it will not go thru the array. Instead, it will throw an error.
- Due to the fact that there is the array of objects that you will want to look thru. Use the for loop such as followed:
- for (var i = 1; i <listItems.length; i++) {
  }

Function Location

When you are using JavaScript to code, remember that there are two ways that you are able to declare the function. First, you will declare a function if you want to form a variable assignment. When you do this, the location of the function will not matter.

Here is a simple example:

```
var myFunction = function (arg1, arg2){
// You can put any code here, again, it doesn't matter
};
```

- When the function is declared as function myFunction (arg1, arg2) {},this means that it can be used in your file, no matter where it is located. This is both before and after your function has been declared. However, if your function happens to be declared with a variable assignment, it can only be used by a code that will execute this assignment according to its declared function.

Undefined and Null

Going back to basic coding 101, it is important that you understand these terms are not one in the same! In case you forgot, null is used for an object in your code. Undefined will be used for the variables, methods, and the properties in your code. In order to be null, the object in question will need to be defined. If it is not, you can test to see if it is null. When an object is not defined, it cannot be tested and will throw an error instead.

Here is an example:

```
if (myObject ! == null & typeof(myObject) ! ==
'undefined') {
}
```

- Note: if myObject happens to be undefined, remember that it cannot be tested for null. Instead, it will throw an error.

```
if (typeof (myObject) ! == 'undefined' && myObject !==
null) {
}
```

- Note: This code will be in charge of handling myObject

Statement Switching

When you are using the switch statement in JavaScript, there are a few things you need to recognize. First, remember that there is no datatype conversion. Instead, there is a match, and at this points, all of the expressions are going to be executed until there is either a return statement or a break statement. In this case, you can include multiple cases in your code in just a single block! Remember as we stated earlier in the chapter, your Data Type is going to be very important. In comparisons, JavaScript is not loosely typed. Statements are something you will need to pay close attention to. We will be going over this practice in later chapters. Be sure to stick around and learn as much as you can.

Chapter 2:
General Recommendations

While you are reading through this book, we want it to be a learning tool. Essentially, we wish for you also to use this as a quick guide. Below, we will list a general recommendation checklist for you to peek at quickly while you are in the middle of coding. After, we will go into depth on each of the suggestions from below. Remember that everyone is going to make mistakes. When you are learning to code, it is much like learning a new language. This important factor is to catch your mistakes, learn from them, and then try to make them as little as possible.

Here is your checklist:

- DO NOT persist password locally
- DO NOT use hard code usernames (or passwords) in your client processed codes
- DO NOT use hard code strings in a program template or code
- DO NOT test for variables against the strings for any triggered events
- DO NOT use hyphens when naming a variable
- DO NOT leave a "debugger" statement in your code
- DO NOT use the function "eval()"
- DO NOT EVER use Global Variables
- DO NOT leave any dangling commas, especially in object definitions and lists
- DO define all of your local var's at the TOP of your function
- DO use (ie"===") explicit comparators.
- DO use shortcuts for your value assignment defaults
- DO place braces on the right side of your block definitions
- DO use semi-colons at the end of your lines
- DO follow naming conventions for your functions and variables
- DO use double quotes for your NLS'd Strings

1. **DO NOT persist password locally**

   As you could imagine, if you store any passwords in your code, this could create some major security holes. These are aspects you will want to avoid at all costs. The last thing you will want to do is be exploited by an attacker because you made a simple mistake. In any case, we suggest never persisting your passwords in sessionStorage, Cookies, or even localStorage. Instead, allow the user to get the password through normal mechanisms such as the browser. Just avoid putting it in code at all costs.

2. **DO NOT use hard code usernames (or passwords) in your client processed codes**

   Again, this is an easy way to create security holes. To some coders, this could make you just a lazy coder! Even if your code is just a test or a simple demo, never use your username and password in the code. You just never know who could get their hands on

the code. The internet is a very wide world.

3. **DO NOT use hard code strings in a program template or code**

When you are coding, you should just always assume that your code is going to be translated. If you want to avoid being lazy, at least create a map for the key values of your strings.

Here is a simple example:

- this.Text = {
  "greeting" : "Hello",
  "firstName" ; "First name",
  };

4. **DO NOT test for variables against the strings for any triggered events**

Remember that while you are condition checking in your code, these checks should be made against the variables or the reference node that is meant to trigger an event. We suggest not doing these checks against multilingual values. If you do this, it could help reduce the chance of throwing an error. By doing this, it will also help improve the maintainability of the code. When you are careful, the code will clearly show the actual conditions for the given action in the code. Here is an incorrect example followed by a better example:

- **Incorrect Example:**

```
if acct.type.match(/^A LINE OF CREDIT.*/) ||
    acct.type.match (/LINEA DE CREDITO.*/) ) {
```

- **Correct Example:**

```
if (acct.type === this.acct.TYPES.credit) {
```

- While you are coding, be sure to be aware of your patterns. Some have made the mistake of checking the target.label of an event to determine whether or not it is a source button. Instead, we suggest using an "class" or "id" identifier. This way, it will not break an application if it gets translated.

5. **DO NOT use hyphens when naming a variable**

Coding can be difficult; this is why we suggest not using hyphens in a variable name. if this happens, it could be confused as an attempt at subtraction. For example:

```
var hello-name = "Welcome";
```

- The way this code is set up, it will try to take the name away from the world. Obviously, this will throw an error.

6. **DO NOT leave a "debugger" statement in your code**

As a reminder, a "debugger" statement is used to force your browser to a breakpoint during any testing. This is the code that helps most developers to locate the bugs quickly and efficiently. However, it could build other failures if it is accidently left in the code. Instead, we suggest manually setting up a breakpoint by using a browser tool instead. This could save you the time and hassle at the end of the day.

7. **DO NOT use the function "eval()"**

There is a saying that some developers use that is known as "eval is evil." This should be an easy way to remember that you should rarely use eval, if ever. This code is a very easy way to make bad things happen in your code. Unless you truly know what you are doing with code, the source needs always to be trusted. If not, it could lead to vulnerabilities in your code. As you learned earlier, this is something you want to avoid at all costs!

8. **DO NOT EVER use Global Variables**

This is something we will say over and over again. When you are defining local variables, always use the var statement. If not, the variable will be placed into a global scope. You need to avoid this. Instead, we suggest using an AMD based loader. This way, it will eliminate any packages or modules that are globally namespaced. Please also note that this will not apply on browsers that provide global including console, navigator, window, or document.

9. **DO NOT leave any dangling commas, especially in object definitions and lists**

While most browsers will actually allow this with modern technology, it used to cause Internet Explorer to throw errors. It should be noted that by leaving dangling commas, it could create some sloppy programming. Most of the time, an issue like this is seen in define statements. It has also been found after the last method/function in a declared class. Here is an incorrect example:

```
var a = [1,2,3,];
var b = { a:1, b: "B", c: x,};
```

- As a general tip, you will want to generate your visual lists by using an Array. This way, you can join the results. Here is an example:

```
console.log ( "A List of Names: ", nameList.join ("firstname, lastname") );
```

10. **DO define all of your local var's at the TOP of your function**

While this isn't necessarily required, by defining your local variables at the top of your function, it can lead to a leaner code. The local variables are normally at the "function" scope. In other words, they are accessible within any enclosing function. This does not mean that they are only within the closing blocks. Instead, random "var" definitions that are placed throughout an entire function could lead to errors such as unintended variable re-assignment. Remember this will also apply for the "for (var I in…) blocks as well. While the "I" variable will not be scoped for the block, it will be scoped for the outer containing function. This can become very confusing as this is a common practice in other coding languages. Please note that this is not the case for JavaScript. Please just define your local variables at the top of your function to avoid any unintended side-effects.

11. **DO use (ie"===") explicit comparators.**

In JavaScript, you will always want to use the explicit comparators over any coerced comparators. By using the coerced comparators, you will be able to convert any value types then be able to match and compare them.

Here is a great example of doing this:

```
"Hey" == 2
" " == 1
1 == true
```

- By using the triple equals as compared to the !== this will force your code to create a pure comparison of the values.

12. **DO use shortcuts for your value assignment defaults**

When you are coding, your main goal is to make your code fast and readable. By using the logical OR during operation assignments, this will allow your code to have default values. Here is an example of putting simple validation on an input args. By doing this, it will ensure that the code will work on any expected object.

```
//"Args" will default to an empty object if it is not defined
args = args || {} ;
//As you know, args will be an object. Now that it is defined, it will be safe to do a var greeting such as:
var greeting = args. Greeting || "Hello";
```

13. **DO place braces on the right side of your block definitions**

While this is a personal style choice, it is always suggested to place the braces on the right of your block definitions to keep a safer code. As you start coding more, you will find a style that makes your work easier for you. Here are a couple of examples of placing the braces:

- **Bad:**

```
return              //If you do this, returns will be undefined and not expected by the object.
{                    //This will make sure that the block is valid. Please note that it will be no-ops!
        Ok : false
};
```

- **Good:**

```
return {
        ok : true
};
```

14. **DO use semi-colons at the end of your lines**
    While the use of semi-colons at the end of your lines is not TECHNICALLY required, your code will do better this way. Instead of guessing how the lines will be interpreted by your compiler, just place your semi-colons at the end, so there is no guessing game!
15. **DO follow naming conventions for your functions and variables**
    By doing this, it will help create a cleaner code. Remember that you will not always be the one reading/ writing your code. To keep your code consistent, use some of the following naming conventions to stay organized:

    - First, make sure that your constants are always uppercase
      Example: var A= 1.2

    - Next, when you are using PRIVATE variables, try and use leading underscore
      Example: _myUsername

    - Last, remember that normal variables will be labeled at the following:
      Example: camelCase

16. **DO use double quotes for your NLS'd Strings**

    In coding, there are many languages that will use single quotes heavily as part of that specific language. Most of the time, translators will not understand the "JSON" structure style. If this happens, it will introduce invalid strings if there are single quoted strings.

    Here is an example:

    English: { "end" : "The end of the story" }

Chapter 3:
Best Practices for Comments and Logging

When you are coding with JavaScript, there are a few tricks to make your comments and logging a bit easier. Remember that while you are working or practicing coding, you will want to do whatever you can to keep your coding clean and organized. Both loggings and comments should be used through your code. Please note that there are no performance penalties for using either. Most of the time, they will be removed during your building phase anyway. While some developers may note that logging can be a substitute for debugging skills, it is a valuable skill you will want to have to help aid your logic comprehension in coding.

Here, we will provide you with a quick list to remember when working with logging and comments.

- DO use proper logging levels
- DO use comment for all complex logic
- DO use standard logging for any useful details
- DO use JSDoc Syntax for any public functions, variables, or modules
- DO NOT use string concatenation

1. DO use proper logging levels

Please note that both WL.logger and Console both support a few different levels for logging. If you don’t recall, these include: warn, error, log, and debug. When you use a proper level in your coding for a certain situation, it will help you manage what you log in a better way. For example, if you are building code, your default for the Dojo is going to be removing the logging that appears below the "error." In this case, any console.error() would remain in the code. The issue with this is that after you build, any error conditions would be removed from the code and then it would be lost.

2. DO use comment for all complex logic

Most of the time, a code in JavaScript is readable if you have the skills. There are those times when the code gets too complex even to solve complex problems. For example, it isn't always practical to have a refractor code just so that someone else (such as a novice) can understand the code. This also stands for the developers who try to be "elegant" with their code. To keep it simple, and make everyone's lives easier, comment your complex code. At the end of the day, this will help aid everyone in the comprehension of the code.

3. DO use standard logging for any useful details

When you are creating code, your logging should include content that will be meaningful. Most developers will want to stay liberal when they are logging by using console.log (). When creating a log, be sure to provide the valuable content such as module, source, and function. Any message that you include should be descriptive and include useful variables. We also suggest avoiding

using alert () statements as a substitute for the logging. Most of the times, it could get left in the code, and it will make it more difficult for everyone.

4. DO use JSDoc Syntax for any public functions, variables, or modules

If you weren't already aware, JSDOC is a way that you can annotate your JavaScript code. This is a reason why it is vital to comment on your public variables and functions properly. By nature, private variables and functions can be documented, but they are less important. If you are using a reusable code, it could be contributed to a public or private library, and at this point, it will need to be properly documented.

5. DO NOT use string concatenation

While you are coding, we suggest using multiple arguments when you are logging. This way, you will be able to avoid using string concatenation. This is a very common anti-pattern within logging.

Here is a bad example:

```
Console.error ("Invalid arguments provided: " + args);
```

- In this case, the code will fail. The output has to run toString() on the objects. In this case, it will show as the following:

```
Invalid arguments provided: [Object]
```

- Instead, you will want to use multiple arguments to show the object you want. Here is the proper example:

```
Console.error ("Invalid arguments provided:", args);
```

- As you can see, it is a simple fix, but it something you could be making fairly easily.

Chapter 4:
Tips for Better Performance

Whether you are a novice in the coding world or have been practicing for a while, you can admit to yourself that you make mistakes. Remember that this is ok. There are always ways to help you perform better in coding.

In this chapter, we will be providing you with just a few ways to help improve your code. From minimizing your DOM access to staying suspicious of any logic inside of your loops, there is always room for improvement.

Here is your checklist if you are in a rush!

- DO ensure that your 3rd party libraries are always optimized
- DO minimize any DOM manipulation and DOM access
- DO keep your DOM at a minimum
- DO stay suspicious of the logic inside of your loops
- DO delay JavaScript Loading
- DO reduce activity in your loops

1. DO ensure that your 3rd party libraries are always optimized

While coding, you will want to be aware that you will ALWAYS need to optimize any custom JavaScript. On top of this, you must be sure that you are using an optimized version of any 3rd party libraries as well. As a general tip, minimized libraries will have ".min" in their title. While researching, you will want to look for websites that promise you are using an optimized version as opposed to a developer's source.

2. DO minimize any DOM manipulation and DOM access

If you weren't already aware, DOM is incredibly slow compared to any other code in your application. This is why it is vital for you to minimize any access to the DOM when it is possible. If you absolutely need to access a certain node, we suggest doing this once and then store your results into a local variable. Last, we suggest you build a large DOM fragment to insert once. This is a much better option compared to building nodes into your DOM and having to iterate it every time.

3. DO keep your DOM at a minimum

If you are coding for an application, you will want the code to be as quick and seamless as possible. Unfortunately, the performance will drop as the complexity and DOM size increase. By being proactive on your DOM size, you can improve the reflow of the operations. Luckily, you have two basic options to help minimize your DOM. First, you could destroy the view to save state. You will also need to destroy its children. While this can become complex to manage the

state and child widget, it is actually very unlikely that your user will revisit this view again. Instead, you will want to clean up any DOM resources or memory.

Your second option is to use the DomCache code. For this, the code will offload the view's node tree. In return, it will turn into a DomFragment variable and be transitioned out of the code. At this point, if you need the view again, you can check your cache and then reinsert it back into the DOM. If you have a view that is revisited, this is the best option for you.

4. DO stay suspicious of the logic inside of your loops

For most developers, it is normal to iterate through your elements and data by using a loop. However, you do not want to take the information as it happens within your loop. Remember that a number of potential negative performances could occur. You will want to keep your eye out for DOM access, storage access, math calculations, AJAC calls, and any other manipulations. The sooner you catch any of these outside of your loop, the better!

5. DO delay JavaScript Loading

If you put your script at the bottom of the page, it will allow your browser to load the page first. But, while your script is loading, your browser will not be able to start other downloads. If this happens, other rendering activity could be blocked. Instead, try using the code defer= "true" in your script tag. This attribute will be executed once your page has finished parsing. Please note that this will only work on external scripts. Try the following code example:

CODE:

```
<script>

window. onload = downScripts;


function downScripts ( ) {
        var element = document. createElement ("thescript");
        element.src = "theScript.js";
        document.body.appendChild(element);
}
</script>
```

6. Do reduce activity in your loops

Remember that the loops in your code will include a statement. This statement will be executed during each iteration of the loop. If your statements are placed outside of the loop, this will help you run the loop faster.

Here is an example of a bad code and a better code to help improve the performance:'

- Bad Code:

```
var I;
```

```
for (I = 1; I<arr.length; I++) {
```

- Better Code Performance:

```
var I;
var m = arr.length;
for ( I = 1; I < m; I++) {
```

Chapter 5:
Best JavaScript Tips for Beginners

Whether you like it or not, as a programmer or web or app developer, you need to learn JavaScript. It is considered the computer language of the future simply because it is an integral part of web ad app development.

Of course, it would be next to impossible to cover every trick to help you learn the best practices for JavaScript. Instead, we want to focus on the main attributes you will come across as you learn to code. In the chapter to follow, we will provide you with some of the simpler best practices to get you on the right path. Remember that there is always more information to learn about coding.

This is just an excellent start!

- DO NOT declare Boolean Objects, Strings, or Numbers
- DO Initialize Any Variables
- DO NOT use newObject ()
- DO use Parameter Defaults
- DO NOT use Automatic Type Conversions
- DO use "Timer" to Optimize Code
- DO NOT Pass String to "SetTimeOut" or "SetInterval"
- DO use "For in" Statements
- DO NOT use "With" Statements

1. DO NOT declare Boolean Objects, Strings, or Numbers

While you are coding, it is very important that you treat your Booleans, strings, and numbers as primitive values. You will not want to see these as objects. If you define any of these codes as objects, it could produce bad side effects and will just slow down the speed of your execution.

Here is a quick example:

```
var x= "Joe";

var y= new String ("Joe");

(x ===y)
```

- As you can see, this will be false due to the fact that x is a string while y is an object. It will throw an error, and this is something you can avoid!

2. DO Initialize Any Variables

Whether you are learning code, or are just looking to fix your skills, it is great practice to initialize your variables as you declare them. When you do this, it will help avoid undefined values, it will provide you with a single place to do this, and it will help give you a cleaner code.

3. DO NOT use newObject()

While this is something we touched on a bit earlier, it is important to remember. Here is a quick cheat sheet to help you remember which code to use compared to its similar code. By being organized, you can keep your code clean.

- DO USE: {} DO NOT USE: new Object()
- DO USE 0 DO NOT USE new Number()
- DO USE " " DO NOT USE new String()
- DO USE false DO NOT USE new Boolean()
- DO USE function () {} DO NOT USE new Function()
- DO USE /()/ DO NOT USE new RegExp()
- DO USE [] DO NOT USE new Array()

4. DO use Parameter Defaults

We invite you to ask a moment to ask yourself, what do I remember about functions? What do you call it when your function is missing an argument? That's right! The value of that missing argument will be called undefined. If you weren't already aware, these undefined values can and will break your code. To break this habit, try your best to assign values to your arguments automatically.

Here is an example:

```
function myFunction (x, y) {
 if ( x === undefined) {
 x= 1;
 }
}
```

5. DO NOT use Automatic Type Conversions

While you are coding, you will want to be aware of the numbers that could be converted to NaN or strings. If you need a refresher, NaN stands for Not a Number. This is something that can happen automatically due to the fact that JavaScript is typed loosely. The variables in your code can change the data type as well as contain a few different types of data.

Here is a quick example:

```
var x = "Hey";
```

- In this case, x is a string but if you were to continue;

```
x= 4;
```

- Then, this changes your x to a number. You can see where this could cause errors in your code.

You also should be aware that your JavaScript will convert numbers into strings if you are doing mathematical problems. If you type it the wrong way, it will give you an answer, but it may not be the one you wanted. Here is a quick example:

- In this example, typeof x will be a number

```
var x = 5+ 7;  // x.valueOf() will be 12
```

- In this next example, the typeof x will be a string

```
var x = 5+ "7"; // x.valueOf() will be 57
```

In a different case, you could try to use your function to subtract one string from another string. In this case, it will return as NaN.

Use this example:

- "Goodbye" – "Love" // This will return as NaN

6. DO use "Timer" to Optimize Code

This practice is super simple. Are there ever times that you need a quick way to determine exactly how long your operation will take? Luckily, Firebug has a "timer" feature that will log the results for you! Sometimes it helps to have some tricks on the side to help your coding along.

Here is the code for that:

```
function TimeTracker () {
  console.time ("MyTimer");
   for (x=4000; x > 0; x--) {}
  console.timeEnd ("MyTimer");
}
```

7. DO NOT Pass String to "SetTimeOut" or "SetInterval"

Remember when we told you not to use the "eval" function? This code will function the same way. This is why we suggest never passing the string to SetTimeOut or SetInterval. Unfortunately, this code is super inefficient.

Instead of this code:

```
setInterval (
```

```
"document.getElementById('container').innerHTML _+= 'The New Number: ' + I" , 2000
);
```

Use this one:

```
setInterval(someFunction, 2000);
```

8. DO use "For in" Statements

Sometimes when you loop items in an object, you may have noticed that at the same time, you retrieve the method functions as well. If you do not want this to happen, there is a way you can wrap your code as an if statement. This way, it will filter your information.

Here is an example:

```
for (key in object) {
 if(object. hasOwnProperty(key)) {
        //…do something…
   }
}
```

9. DO NOT use "With" Statements

Now, this coding tip is up for debate. On the one hand, "with" statements truly sound like a good idea to use. In retrospect, a "with" statement should be able to be used as a shorthand to access any objects that could be deeply nested in your code.

Here is an example:

```
with (person.woman.bodyparts) {
hand = true;
foot= true;
}
```

- As we noted, you would use these statements instead:

```
person.woman.bodyparts.hand= true;
person.woman.bodyparts.foot= true;
```

After some tests, it was found that these codes behaved badly, especially when setting up new members. Instead of using "with" you can use "var" and get the proper results.

Conclusion

There we go! We hope at this point in the book you feel a bit better and more confident about your coding skills.

True, it can be challenging at times, but with enough practice, you will only get better! Remember that the examples we included in this book just scratch the surface. There are always ways to improve your coding; just keep on working on it.

The book provided a lot of topics on applying Best Practices to your JavaScript programming code. It is important to be aware of this in order to have much better and efficient code.

In the first chapter, we went over some of the common JavaScript Mistakes. We want you to understand that simple mistakes that could be messing up your code.

After that, we went over some general recommendations to practice better coding. We also went over the best practices for a better performance and also the best practices for your comments and logging. Finally, we discussed the best tricks up our sleeves for beginners.

We hope you get a lot of valuable information.

We wish you the best of luck on your coding journey!!

About the Author

Charlie Masterson is a computer programmer and instructor who have developed several applications and computer programs.

As a computer science student, he got interested in programming early but got frustrated learning the highly complex subject matter.

Charlie wanted a teaching method that he could easily learn from and develop his programming skills. He soon discovered a teaching series that made him learn faster and better.

Applying the same approach, Charlie successfully learned different programming languages and is now teaching the subject matter through writing books.

With the books that he writes on computer programming, he hopes to provide great value and help readers interested to learn computer-related topics.