**PROGRAMMING LANGUAGE THEORY**
**CSC 620**

# IMPROVING JAVASCRIPT

By,
Priya Phapale -- 94662
Vidisha Kotamarti -- 92992
Lalit Singh Chauhan -- 84937
Eric Palma -- 95302

# TABLE OF CONTENT

## INTRODUCTION

JavaScript is one of the world’s most popular programming languages. It’s also one of the three main languages for web developers- HTML which lets you add content to a web page, CS which specifies the layout, style, alignment of web pages and JavaScript which improves the way web pages behave.

JavaScript has many advantages that make it different and better than other programming languages,

1. You do not need a compiler because web browsers interpret it with HTML
2. It’s easier to learn than other programming languages
3. Errors are easier to spot and therefore to correct
4. It can be assigned to certain web page elements or specific events such as clicks or mouseovers
5. JS works across multiple browsers, platforms, etc
6. You can use JavaScript to validate inputs and reduce the need for manual data checks
7. It makes websites more interactive and holds visitors’ attentions
8. It’s faster and more lightweight than other programming languages.

Along with these Javascript has some disadvantages as well.

1. Vulnerable to exploits
2. Can be used to execute malicious code on a user’s computer
3. Not always supported by different browsers and devices
4. JS code snippets are quite large
5. Can be rendered differently on different devices leading to inconsistency.

Our goal is to find some ways by which we can improve some functionalities in Javascript so that it will improve performance of any application in terms of memory and execution time.

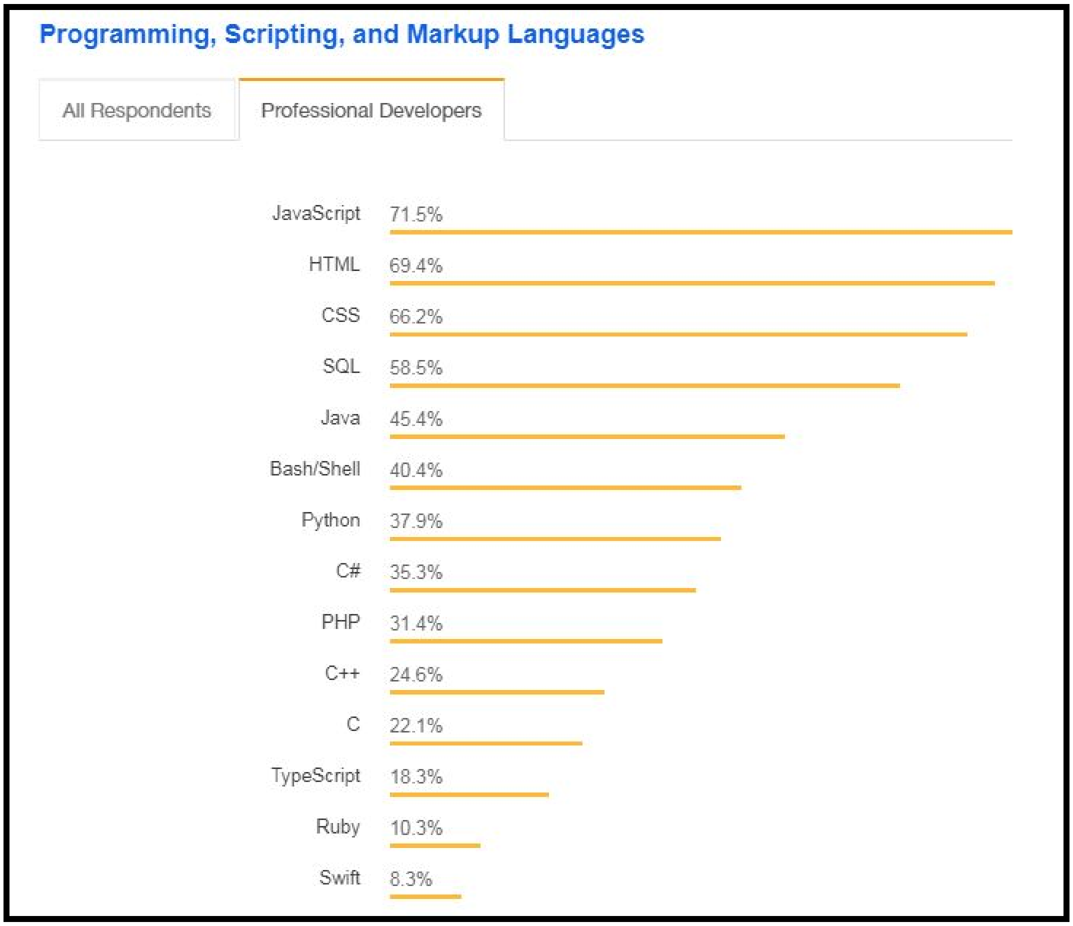


**Figure 1.** Stack Overflow 2018 Survey.
Source: https://insights.stackoverflow.com/survey/2018/

As JavaScript has been rapidly adopted by the web developer community due to its simplicity and browser compatibility, there are some features that would be good to evaluate to improve the language to make it stronger and more reliable. That is precisely what we want to discuss in this research.

## OBJECTIVE

The objective that we as a group have defined for this project are the following:

1. Analyze and determine 4 optimization opportunities for the computer language Javascript.
2. Present proposal to implement in the language to improve it or make more efficient.

## SCOPE

Due to the magnitude and complexity of the improvements that can take place in a high level computer language like JavaScript, we are going to focus just in the objectives presented before and its correspondent improvement proposals. This research does not pretend to give solution to all technical needs currently present. In general terms, we have find four optimization opportunities in JavaScript that we will enumerate as follows:

- Memory Optimization
- Loops Optimization
- Event Handling Optimization
- String Handling Optimization

# MEMORY OPTIMIZATION

To talk about memory optimization on JavaScript we have to mention some specific characteristics of the language.

JavaScript is a dynamic language which was designed to make it fast and easy for web developers to implement projects or update them.

In JavaScript is not needed to declare a type of variable ahead of time. The program assigns a default amount of memory to the variable declared. To check the type of any variable is possible to get this information with “typeof” operator.

## VARIABLE TYPE MANAGEMENT IN OTHER COMPUTER LANGUAGES

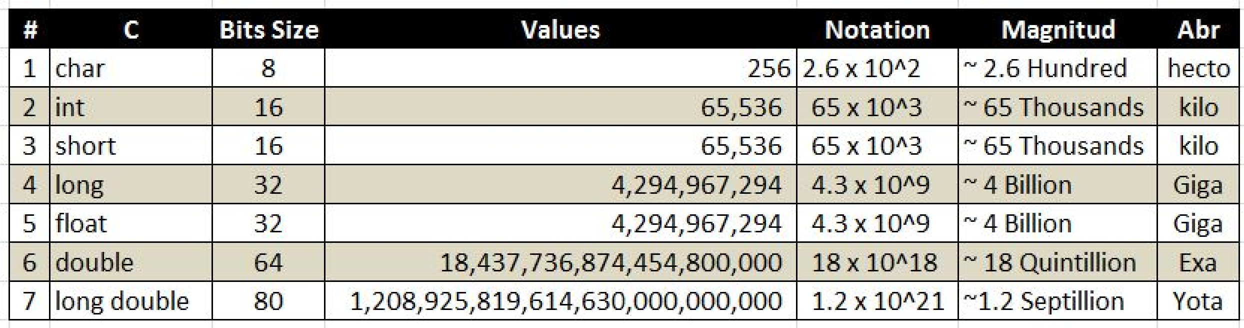
To put in context the way that JavaScript handles memory assignment, would be good to compare it with stable, low-level programming language to notice the differences. We are going to compare it with object oriented programs (OOP), Java and C.

In a low level programming language, this step is always needed, and is a way to efficiently assign resources, but at the same time is a source of errors too. Let us analyze the differences between these languages.

### The “C” Language

“C is one of the most powerful "modern" programming language, in that it allows direct access to memory and many "low level" computer operations. C source code is compiled into stand-alone executable programs. It forms (or is the basis for) the core of the modern languages Java and C++.” ~ H. James de St. Germain from the School of Computing at the University of Utah.

C has seven variable types and assigns different amount of space for each of them as detailed in table 1.



| # | C | Bits Size | Values | Notation | Magnitud | Abr |
|---|---|---|---|---|---|---|
| 1 | char | 8 | 256 | 2.6 x 10^2 | ~ 2.6 Hundred | hecto |
| 2 | int | 16 | 65,536 | 65 x 10^3 | ~ 65 Thousands | kilo |
| 3 | short | 16 | 65,536 | 65 x 10^3 | ~ 65 Thousands | kilo |
| 4 | long | 32 | 4,294,967,294 | 4.3 x 10^9 | ~ 4 Billion | Giga |
| 5 | float | 32 | 4,294,967,294 | 4.3 x 10^9 | ~ 4 Billion | Giga |
| 6 | double | 64 | 18,437,736,874,454,800,000 | 18 x 10^18 | ~ 18 Quintillion | Exa |
| 7 | long double | 80 | 1,208,925,819,614,630,000,000,000 | 1.2 x 10^21 | ~1.2 Septillion | Yota |

**Table 1.** C data types and their memory assignment.

### Java

Created was created “in the early 90s, Java, which originally went by the name Oak and then Green, was created by a team led by James Gosling for Sun Microsystems, a company now owned by Oracle." ~ Paul Leahy from ThoughtCo.

Java successfully implemented key principles like: Ease of use, robust, reliability, security, and platform independence. These features and continuous evolution have given Java’s popularity and acceptance in the developer community.
Figure 2 is a graphical representation of the different data types in Java.

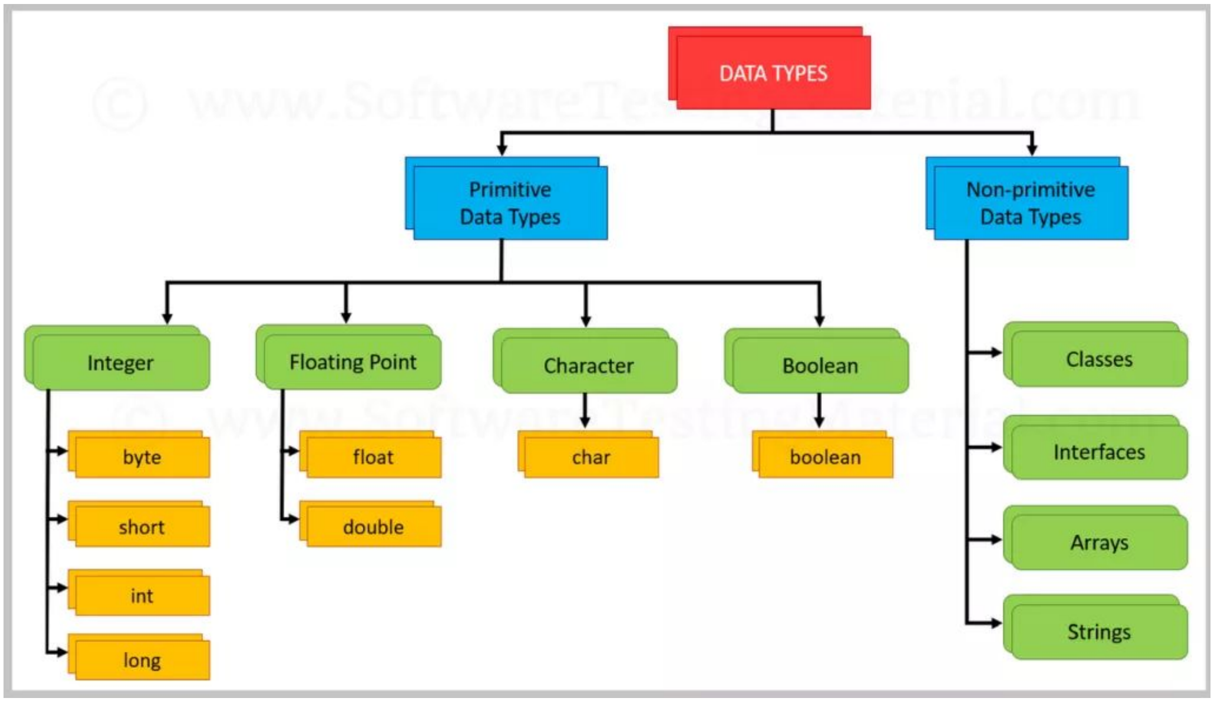
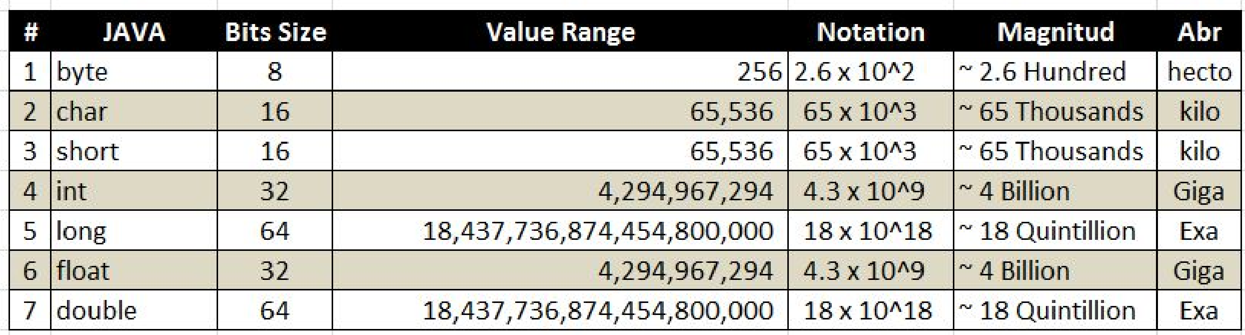


**Figure 2.** Data types in Java.

Now, how does Java handles data types to have such reliability and security features? Table 2 shows the different data types in Java and their memory assignment.

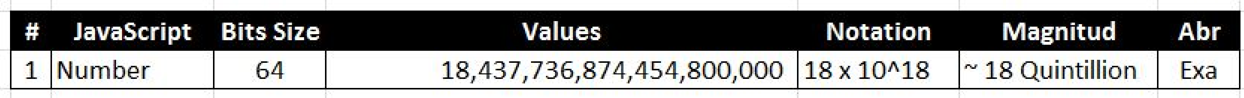


| # | JAVA | Bits Size | Value Range | Notation | Magnitud | Abr |
|---|---|---|---|---|---|---|
| 1 | byte | 8 | 256 | 2.6 x 10^2 | ~ 2.6 Hundred | hecto |
| 2 | char | 16 | 65,536 | 65 x 10^3 | ~ 65 Thousands | kilo |
| 3 | short | 16 | 65,536 | 65 x 10^3 | ~ 65 Thousands | kilo |
| 4 | int | 32 | 4,294,967,294 | 4.3 x 10^9 | ~ 4 Billion | Giga |
| 5 | long | 64 | 18,437,736,874,454,800,000 | 18 x 10^18 | ~ 18 Quintillion | Exa |
| 6 | float | 32 | 4,294,967,294 | 4.3 x 10^9 | ~ 4 Billion | Giga |
| 7 | double | 64 | 18,437,736,874,454,800,000 | 18 x 10^18 | ~ 18 Quintillion | Exa |

**Table 2.** Java data types and their memory assignment.
Source: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

### JavaScript

Now that we have seen the different data type in robust, reliable, secure, and widely used computer languages we have a context to compare JavaScript way to manage number data types. Table 3 show the memory assign for number data types.

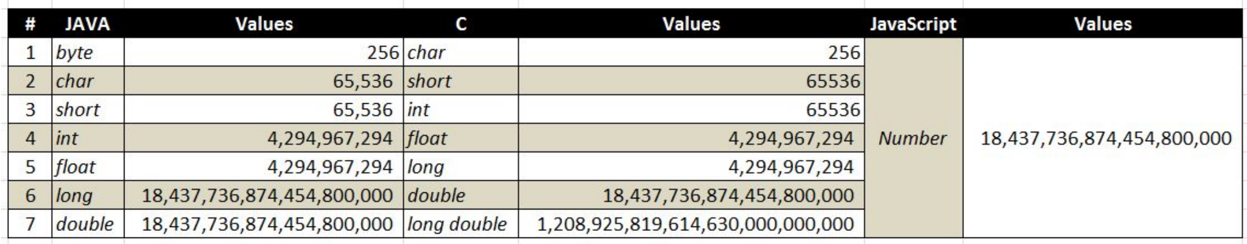


| # | JavaScript | Bits Size | Values | Notation | Magnitud | Abr |
|---|---|---|---|---|---|---|
| 1 | Number | 64 | 18,437,736,874,454,800,000 | 18 x 10^18 | ~ 18 Quintillion | Exa |

**Table 3.** JavaScript data type and memory assignment.
Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

### Computer Languages Comparison

Now let us compare in Table 4 the three languages side by side to analyze similarities and differences.



| # | JAVA | Values | C | Values | JavaScript | Values |
|---|---|---|---|---|---|---|
| 1 | *byte* | 256 | *char* | 256 | *Number* | 18,437,736,874,454,800,000 |
| 2 | *char* | 65,536 | *short* | 65536 | | |
| 3 | *short* | 65,536 | *int* | 65536 | | |
| 4 | *int* | 4,294,967,294 | *float* | 4,294,967,294 | | |
| 5 | *float* | 4,294,967,294 | *long* | 4,294,967,294 | | |
| 6 | *long* | 18,437,736,874,454,800,000 | *double* | 18,437,736,874,454,800,000 | | |
| 7 | *double* | 18,437,736,874,454,800,000 | *long double* | 1,208,925,819,614,630,000,000,000 | | |

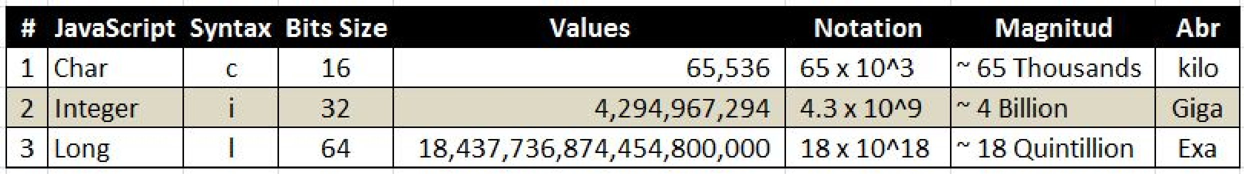
**Table 4.** Computer languages comparison in memory assignment in number data types.

This comparison allow us to confirm the reason of JavaScript been so popular among web developers. The reason that JavaScript only have one number data type against seven in Java or C, makes reduce fatal errors at the development moment. Nonetheless, this flexibility creates a lot memory waste, which traduces in reliability concerns. As always in computing there is a cost-benefit among flexibility and security.

## IMPROVEMENT OPPORTUNITY

We know variable declaration in Javascript does not need to specify the type of data. For example:
var nameOfVariable = 600;

Therefore, our proposal is to define just three number types in order to maintain flexible and simple, but at the same time more reliable. See Table 5 for details.



| # | JavaScript | Syntax | Bits Size | Values | Notation | Magnitud | Abr |
|---|---|---|---|---|---|---|---|
| 1 | Char | c | 16 | 65,536 | 65 x 10^3 | ~ 65 Thousands | kilo |
| 2 | Integer | i | 32 | 4,294,967,294 | 4.3 x 10^9 | ~ 4 Billion | Giga |
| 3 | Long | l | 64 | 18,437,736,874,454,800,000 | 18 x 10^18 | ~ 18 Quintillion | Exa |

**Table 5.** Numbers data types proposal.

## Backus-Naur Form (BNF) Notation

BNF notation for these number data type specification would be as follows:

```
<char> ::= | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<int> ::= <char>
<long> ::= <char>
```

## Variable Declaration

A variable declaration would present the following syntaxis.

For example:
<char> var ***c*** nameOfVariable = 60;
<int> var ***i*** nameOfVariable2 = 700;
<long> var ***l*** nameOfVariable3 = 10000000;

This way is remained simple, short to write, but give way more space managing opportunity.

# LOOPS OPTIMIZATION TO IMPROVE PERFORMANCE

The way a code is structured has a direct performance impact. It can speed up or slow down an operation based on how the code is written. Loops are one of the most used feature of any given programming languages and a majority of execution time of a program is spent within the loops. Hence it is essential to have the code for the loops written in the most optimized manner to speed up the execution and thus improving the performance on the whole.

a. <u>Choosing the correct loop type for an operation:</u>
In JavaScript, there are four different types of loops as the following:

*For Loop:*
For loop is the most used loop in JavaScript (similar to other languages). The syntax for For loop is as below:

```
for (var i=0; i < 10; i++){
        //loop body
}
```

As seen above, for loop has 4 parts – initialization, pretest condition, loop body and the post-execute condition.

*While Loop:*
While loop is a simple pre-test loop with a test condition and the loop body as below:

```
var i = 0;
while(i < 10){
        //loop body
        i++;
}
```

Any for loop can be converted into a while loop and vice-versa.
*Do- While Loop:*
This is a post-test loop, where the loop body is run at least once before evaluating the test condition.

```
var i = 0;
do {
        //loop body
} while (i++ < 10);
```

*For-in Loop:*
The For-in loop loops through the properties of an object. The syntax is as below:

```
for (var prop in object){
        //loop body
}
```

So, each time this loop gets executed, the prop variable is a defined property of the object until all the properties have been iterated upon. The returned properties are both inherited through the prototype chain and those that exist on the object itself.

Out of all the mentioned loops, the for-in is the slowest. This loop has more overhead since each iteration infers a property lookup from the object and the prototype chain. According to [8], for the same number of iterations, the for-in loop is seven times slower than the other loops. Hence, while using JavaScript, it is recommended to not use the for-in loop for operations that have finite number of properties to iterate over. Instead, we can use the other loops as below:

```
var props = ["prop1", "prop2"],
        i = 0;

while (i < props.length){
        process(object[props[i++]]);
}
```

Using the above modification, the code, instead of looking up every property of the object, focuses on the finite elements in the list, thus reducing the iteration overhead and improving the speed of execution.

Apart from choosing the right type of loops, there are other factors that can affect the performance. They are as below:

b. Reducing the amount of work per iteration:

If a single iteration takes a particular amount of time, then all the iterations take much longer. Thus, reducing the expensive work per iteration can hugely benefit us when it comes to looping over such iterations. Consider the following example from [8] :

```
//original loops
for (var i=0; i < items.length; i++){
        process(items[i]);
}
var j=0;
while (j < items.length){
        process(items[j++]);
}
var k=0;
do {
        process(items[k++]);
} while (k < items.length);
```

In all of the above loops, there are multiple operations that occur namely, property lookup (items.length), comparisons (i< items.length and if i<items.length is true), increment operation (i++, j++, k++), array lookup(items[i]) and function call (process(items[i]). Each of these operations burn some amount of time while performing the operation. Even if the above code seems simple, there is a lot going on per iteration. Thus reducing such operations can benefit the speed up and performance of the code overall.

One of the optimizations that can be done is reducing the number of array lookups. This can be achieved by making a call to the array lookup once and storing that value in a local variable and passing that local variable instead of the array. One part of the above code can be modified as below:

```
var k=0,
        num = items.length;
do {
        process(items[k++]);
} while (k < num);
```

Here the array items length is stored in a local variable and that is used for the comparison. By doing so, [8] believes that we can gain up to 25% of the total loop execution time in most browsers and up to 50% in Internet Explorer.

Another optimization technique to reduce the work per iteration is to simply reverse the order of comparisons.

### c. Reducing the number of Iterations:

The idea behind this technique is to finish up a significant amount of work per iteration, thus reducing the number of iterations. In fact, there is a well-known approach to limit the loop iterations called the Duff's device. This method ensures that each switch case gets at most 8 calls to the function to perform an operation, thus reducing the number of iterations overall. Depending on the size of the loop, this technique can provide a significant amount of speedup.

### d. Choosing and optimizing the condition statements:

Based on the number of iterations, it is advantageous to choose between the if-else and switch condition statements. If the iterations are large, it is better to go with the switch statement since it is easier to read and its performance is slightly better over the if-else statements. The difference in performance is due to the incremental cost due to the additional condition statement in if-else versus switch.

In order to optimize the if-else statement, it is essential to ensure that the most commonly occurring condition are placed first. This way the code need not run until it finds the correct conditional loop. Another optimization is use the nested if-else statements as possible. Using nested if-else statements over if-else, helps in cutting down the execution time to half.

e. Lookup Tables:

Lookup tables provide an alternate to cases where the number of conditions to test is significantly large. These can be built using a simple array in JavaScript. Since they eliminate the need for condition testing, the performance benefit is significant. Lookup tables are best used in cases like a Hash map/Hash Table, where there is one-to-one mapping.

The graph below shows a performance comparison between the if-else, switch and the lookup tables for a given number of test cases.

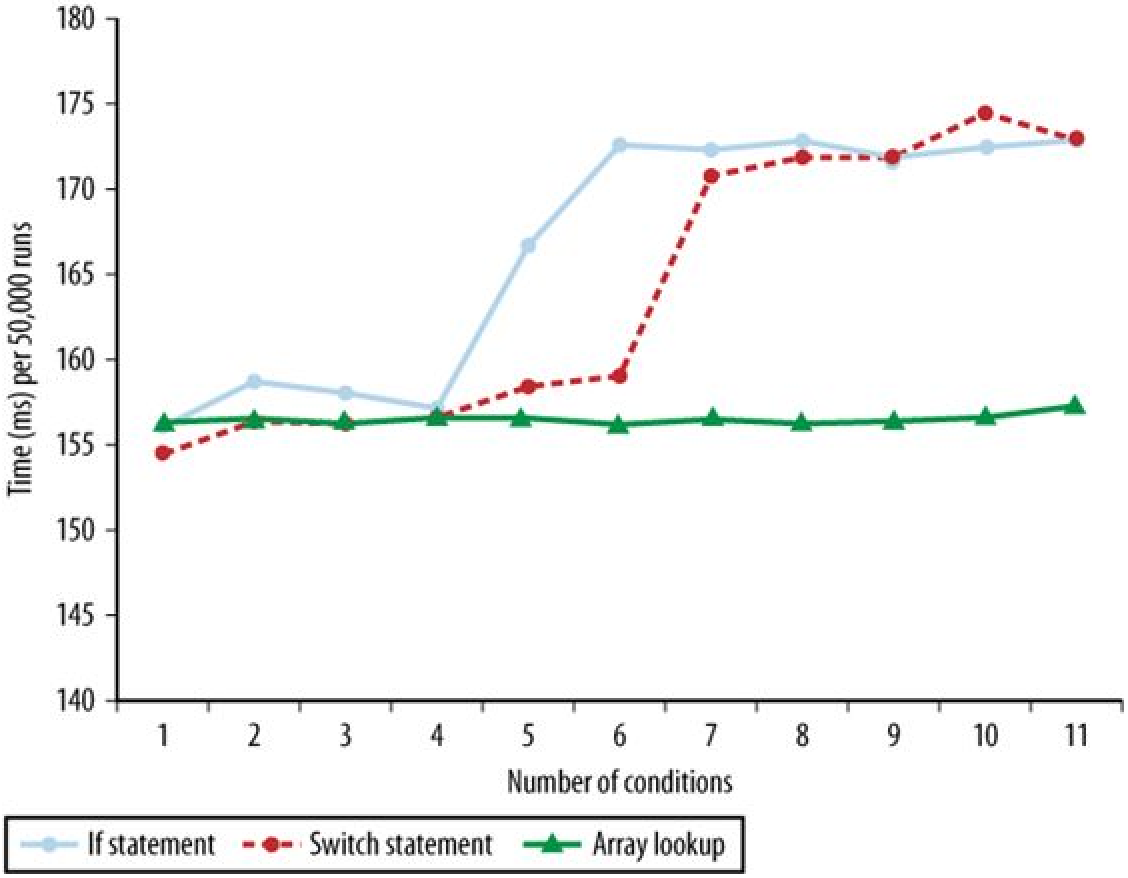


Figure: Array lookup vs if-else vs switch statements [8]

Apart from the above mentioned optimization techniques for loops, recursive and iteration functions that use loops to perform certain part of the function can also be optimized by using the optimized loops and avoiding call stack errors while using the recursive calls to a function. Another technique called Memoization which caches (stores) previous calculations for a later use can also be implemented specially in cases involving iterative functions. This can benefit the overall performance of the program.

Thus, similar to other programming languages, JavaScript programming language's performance also depends on the way a code is written and the algorithm used to implement a particular logic. Since most of the times, loops constitute a majority of the code, it is very important to write this code in the most optimized manner to achieve the desired functionality and performance benefit.

# EVENT AND EVENT HANDLING

**What is an event and event handling?**

Events are actions or occurrences that happen in the system. Using programming we can give a set of instructions to execute when such events are triggered. Events can be of different types for example page loads, mouseover, click on some button or image, pressing any key from the keyboard, closing a window, resizing a window, etc.

In response to those events, we can write a code to display some alerts or warnings messages to users, data to be validated, and virtually any other type of response imaginable. This is called as event handling.

Events are a part of the Document Object Model (DOM) and every HTML element contains a set of events which can trigger JavaScript Code. Here is an example of how we can handle events in javascript.

```
<form name="go">
 <input type="radio" name="C1" onclick="document.bgColor='lightblue'">
 <input type="radio" name="C2" onclick="document.bgColor='lightyellow'">
 <input type="radio" name="C3" onclick="document.bgColor='lightgreen'">
 <font face="Courier New">>
 </form></font>
```

In the above example we are using onclick event. When the user clicks on the radio button, it will change the background color. Here, clicking on the radio button is an event and changing background color is an action taken when the event fires.

The addEventListener() is an inbuilt function in JavaScript which takes the event to listen for, and a second argument to be called whenever the described event gets fired. Any number of event handlers can be added to a single element without overwriting existing event handlers.

**Syntax**

```
element.addEventListener(event, listener);
```

**Example using addEventListener()**

```
<!DOCTYPE html>
 <html>
 <body>
         <button id="try">Click here</button>
         <h1 id="text"></h1>
    <script>
    document.getElementById("try").addEventListener("click", function(){
    document.getElementById("text").innerText = "GeeksforGeeks";
     });
    </script>
 </body>
 </html>
```

**Problems in Javascript using event handling?**

Hard to keep track

Event handlers are an incredible tool for improving user experience and reducing the depth of the call stack especially when we have functions calling a function which calls another function then it becomes hard to keep track.

Poor performance

Event handlers work fine for small applications, but for large programs, it is time taking process. Performance gets degrade because of execution of hidden and repetitive event handlers for each element on a webpage.

The solution for all these problems is **Event delegation**

Event delegation allows you to avoid adding event listeners to specific nodes. Instead of doing that we can add an event listener to one parent. That event listener analyzes bubbled events to find a match on child elements

**Implementation-**

```
 <ul id="parent-list">
          <li id="post-1">Item 1</li>
          <li id="post-2">Item 2</li>
          <li id="post-3">Item 3</li>
          <li id="post-4">Item 4</li>
          <li id="post-5">Item 5</li>
          <li id="post-6">Item 6</li>
</ul>
```

Now assume that we need to execute some action when each child element is clicked. We can add a separate event listener to each individual LI element, but what if LI elements are frequently added and removed from the list? Adding and removing such kind of event listeners would be hard to maintain, especially if the addition and removal code is in different places within your app.

The better solution is to add an event listener to the parent UL element. But if we add the event listener to the parent, how would we know which element was clicked exactly? The answer is simple when the event bubbles up to the UL element, you check the event object's target property to get a reference of the clicked element.

Consider below example

```
// Get the element, add a click listener...
document.getElementById("parent-list").addEventListener("click", function(e)
{
// e.target is the clicked element!
// If it was a list item
if(e.target && e.target.nodeName == "LI")
              {
                      // List item found! Output the ID!
            console.log("List item ", e.target.id.replace("post-", ""), " was  clicked!");
                     }
});
```

Start by adding a click event listener to the parent element. When the event listener is triggered, check the event element to ensure it's the type of element to react to.  If it is a LI element, we have what we need!  If it's not an element that we want, the event can be ignored

**Advantages of using event delegation**

- It enables event handling by handling the objects other than ones which were generated by the events or their containers. It clearly separates component design and its usage.

- It performs much better in applications where more events are generated. It is because of the facts that, this model need not process unhandled events repeatedly, which is the case in the event-inheritance model.

- Event-delegation model allows a separation between a component's design and its use as it enables event handling to be handled by objects other than the ones that generate the events.

- As the event-delegation model does not have to repeatedly process unhandled events, it performs much better in applications where many events are generated. This is unlike the event-inheritance model.

# STRING HANDLING OPTIMIZATION

Different ways a new String object is created are implementation specific. As such, an obvious question one could ask is “since a primitive value String must be coerced to a String Object when trying to access a property, for example str.length, would it be faster if instead we had declared the variable as String Object?”. In other words, could declaring a variable as a String Object, ie var str = new String("hello"), rather than as a primitive value String, ie var str = "hello" potentially save the JS engine from having to create a new String Object on the fly so as to access its properties?

For our showcase, we will use mainly Firefox and Chrome; the results, though, would be similar if we chose any other web browser, as we are focusing not on a speed comparison between two different browser engines, but at a speed comparison between two different versions of the source code on each browser (one version having a primitive value string, and the other a String Object). In addition, we are interested in how the same cases compare in speed to subsequent versions of the same browser. The first sample of benchmarks was collected on the same machine, and then other machines with a different OS/hardware specs were added in order to validate the speed numbers.

For the benchmarks, the case is rather simple; we declare two string variables, one as a primitive value string and the other as an Object String, both of which have the same value:

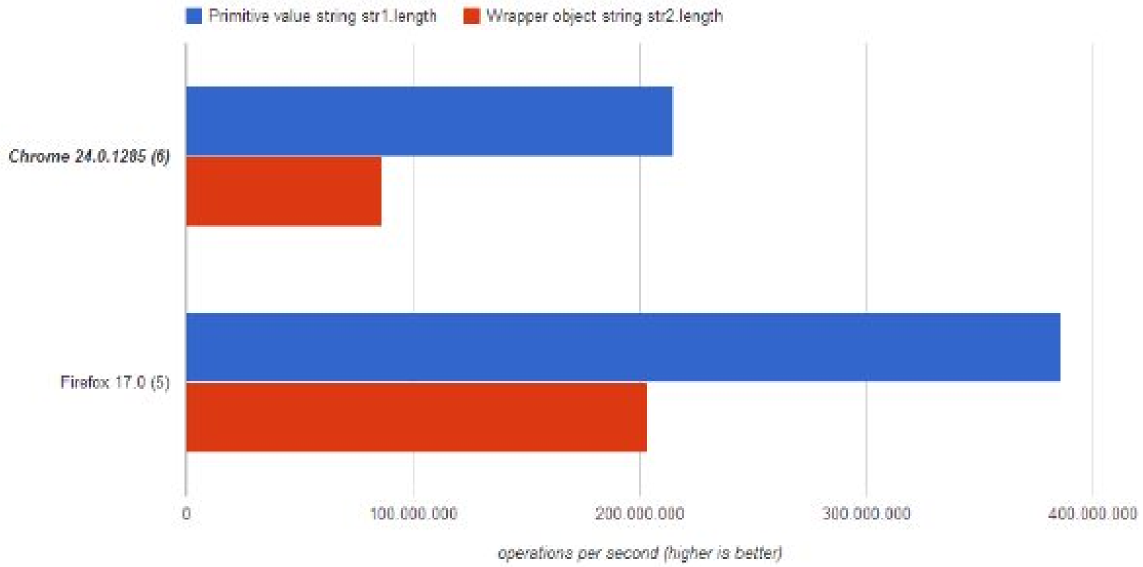
```
var strprimitive = "Hello";
var strobject    = new String("Hello");
```

and then we perform the same kind of tasks on them. (notice that in the jsperf pages strprimitive = str1, and strobject = str2)

1. length property

```
  var i = strprimitive.length;
  var k = strobject.length;
```

If we assume that during runtime the wrapper object created from the primitive value string strprimitive, is treated equally with the object string strobject by the JavaScript engine in terms of performance, then we should expect to see the same latency while trying to access each variable’s length property. Yet, as we can see in the following bar chart, accessing the length property is a lot faster on the primitive value string strprimitive, than in the object string strobject.

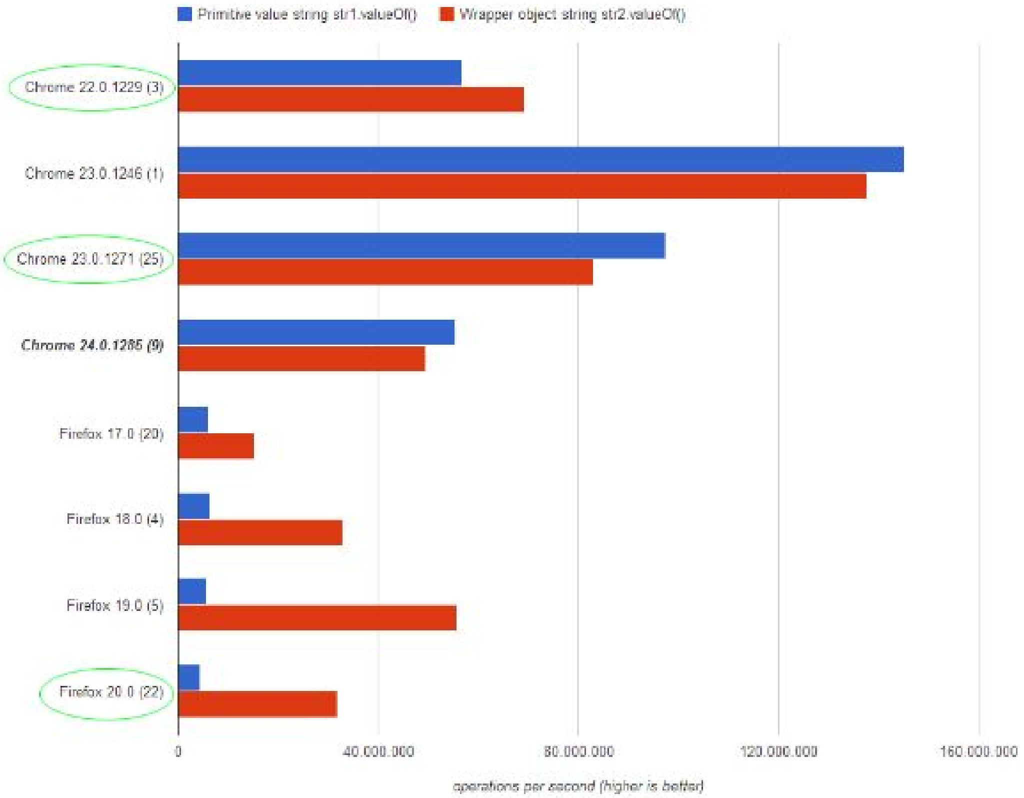


Actually, on Chrome 24.0.1285 calling strprimitive.length is 2.5x faster than calling strobject.length, and on Firefox 17 it is about 2x faster (but having more operations per second). Consequently, we realize that the corresponding browser JavaScript engines apply some “short paths” to access the length property when dealing with primitive string values, with special code blocks for each case

2. valueOf() method

```
 var i = strprimitive.valueOf();
 var k = strobject.valueOf();
```

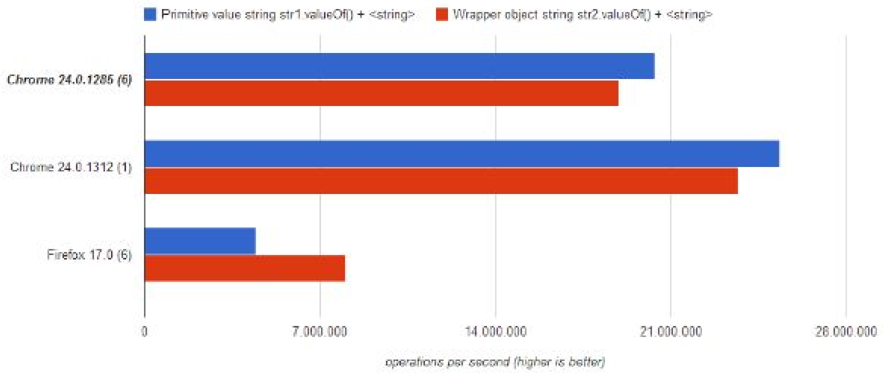
At this point it starts to get more interesting. So, what happens when we try to call the most common method of a string, it’s valueOf()? It seems like most browsers have a mechanism to determine whether it’s a primitive value string or an Object String, thus using a much faster way to get its value; surprizingly enough Firefox versions up to v20, seem to favour the Object String method call of strobject, with a 7x increased speed.



It’s also worth mentioning that Chrome 22.0.1229 seems to have favoured too the Object String, while in version 23.0.1271 a new way to get the content of primitive value strings has been implemented.

3. Adding two strings with valueOf()

```
  var i = strprimitive.valueOf() + " there";
  var k = strobject.valueOf() + " there";
```



Here we can see again that Firefox favours the strobject.valueOf(), since for strprimitive.valueOf() it moves up the inheritance tree and consequently creates a new wapper object for strprimitive. The effect this chained way of events has on the performance can also be seen in the next case.

## CONCLUSION

JavaScript is a high level computer programming language, very popular among the web development community.

Our approach to improve JavaScript focuses in four main points:

- Memory optimization
- Loops optimization
- Event Handling
- String Handling

Improving on these, JavaScript could become more robust, reliable, and stable without losing flexibility and ease of use that have made JavaScript among the web developer industry. Nonetheless, low level computing will always remain as the most secure programming language to use for projects that require high standards on this domain.

## REFERENCES

1. https://insights.stackoverflow.com/survey/2018/
2. https://developer.mozilla.org/en-US/docs/Web/JavaScript
3. https://www.cs.utah.edu/~germain/PPS/Topics/C_Language/the_C_language.html
4. https://www.thoughtco.com/what-is-java-2034117
5. https://intellipaat.com/tutorial/c-tutorial/c-data-types/
6. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
7. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures
8. https://www.oreilly.com/library/view/high-performance-javascript/9781449382308/ch04.html
9. https://www.w3schools.com/js/js_performance.asp
10. https://www.aivosto.com/articles/loopopt.html#loops
11. https://medium.com/@bretdoucette/part-4-what-is-event-delegation-in-javascript-f5c8c0de2983