EcmaScript

Anubhav Saini

Published
with GitBook

Table of Contents

A book on JavaScript

This is **the book** you want to read up on JavaScript.

What people are saying about this book:

> **I wish I had this book in school.**
>
> **Everytime I read this book, it teaches me something new.**
>
> **Where were you 'A book on JavaScript'?**

And more.

Tell @hackecmascript how you felt about the book. Comment on the book content itself if you find a mistake.

Chapter 1 - Solving simple problems

This is the first chapter of the book and it talks about solving simple problems. There is no introduction to the material like JavaScript language and syntax etc. This is by design. The best way to learn programming is by doing it, by solving challenges and problems.

I have added enough explanation to get you through the problems.

By the way, this book is not for beginners. This book is for understanding JavaScript. It is about learning other ways to solve a problem than what we already know. You can be a beginner or an expert and you can still learn a new thing from this book.

Problem - Getting rid of repetition.

We want to welcome a few friends to JavaScript. But all we end up doing is repeating ourselves. Computers are supposed to do the boring, repetitive job and not humans.

We need to get rid of :

```
console.log('Hi, Anubhav');
console.log('Hi, Jake');
console.log('Hi, Ram');
...
```

as our way of greeting our friends.

We need sophisticated way, where we would use name of our friends and they will be greeted.

Solution

We are going to use function to solve our problem. We know that the `console.log('Hi,` and `');` part is common in each line. We are going to put that in a function and use variables where we need names.

```
function SayHi() {
    console.log('');
}
```

`SayHi()` is a function now. One thing to note is that it is not going to do anything all by itself. It is sleeping, sort of. We need to call it to wake it up.

`SayHi();` will call this function and it will work. There is nothing much to work here. When this function will run, it will print an empty single line and quit. It will go back to sleep.

Now, we need to make it say "Hi".

```
function SayHi() {
    console.log('Hi');
}
```

If we call our function now, we will get `Hi` as output of it. This is cool but still basic. We want to greet our friends. For that purpose we are going to use variable named `name`.

```
function SayHi(name) {
    console.log('Hi, ' + name);
}
```

Now, we need to call it with names of our friends.

```
SayHi("Anubhav");
SayHi("Jake");
SayHi("Ram");
...
```

And we are done.

More to read: -

- Variables.
- Functions.
- console.log.

Problem - Reverse a string

Given a string in a variable `str`, return a string that is reversed of `str`.

Examples and conditions:

- If we input str = "Anubhav", reverse(str) should return "vahbunA".
- If we input str = "a string", reverse(str) should return "gnirts a".
- We want to keep punctuation, spaces and every other thing intact except we want the string characters to be in reversed order.

There is no explanation why we need to reverse a string. I assure you that this is the one of the most common problem people ask each other when they are interviewing or learning to program.

We would learn following things from this exercise:

- How to make function return something that interests us.
- How to concatenate string.
- How to fetch characters in a string.
- How to fetch characters in a string backwards.
- How to know the length of the string.
- How to loop over a string.
- How to create a new string by way of concatenation.
- How to use inbuilt JavaScript functions to manipulate input.

Solution

Logic

A string is something like this: `"I am a string."` or `'I am a string.'` where `"` or `'` don't matter. (For this example).

Note: Do not mix `"` and `'`. Every beginning `"` needs ending `"` and same goes for `'` in matters of strings. Though we will see that we need to mix them, but that's when we understand strings better.

Code

```
function reverse(input) {
    var output = "";
    for(var i = 0; i < input.length; i++) {
        output += input.charAt(input.length -1 -i);
    }
    return output;
}
```

There is a lot to unpack here.

- We are using `function` which we have seen.
- We are using `for` loop. We are using `string concatenation via + operator`.
- We are using `charAt` function to get character at a specific position in string.
- We using convoluted mathematics `input.length - 1 -i` to get characters from end of the string.
- We are `return`-ing value from the function instead of `console.log`-ing it.

I suggest you read up on all the concepts before moving forward.

Advanced solution

```
function reverseString(str) {
  return str.split('').reverse().join('');
}

reverseString("hello");
```

Well, first thing to note here is the function `reverseString`.

```
function reverseString(str) {
    return str;
}
```

This function is very basic and has no real purpose. It takes a string in a variable `str` and just returns it.

Let's look at the logic we need to find the reversed string.

Logic

> Get a string.
> Iterate over it from back to front.
> return newly created string thus.

Here, instead of iterating, I am going to use javascript functions that are in built.

split( )

This function splits a string into array. An empty string `''` creates an array that contains every character from string.

String.prototype.split()

reverse( )

This function reverses an array in place. `[1, 2, 3, 4].reverse()` will return `[4, 3, 2, 1]`. Similar thing happens to string.

`"Anubhav".split('')` will split string into an Array `[ "A", "n", "u", "b", "h", "a", "v" ]`.

`[ "A", "n", "u", "b", "h", "a", "v" ].reverse()` will return reversed array: `[ "v", "a", "h", "b", "u", "n", "A" ]`

join( )

Join joins the elements of an array and creates a string.

`[ "v", "a", "h", "b", "u", "n", "A" ].join('')` returns `"vahbunA"`.

Thus, we have succeeded in reversing a string by using `split()`, `reverse()` and `join()`.

Problem - Creating an empty array with length > 0

When we create array, we follow the syntax `var arr = [];` as it is succinct. But this creates an array with `length = 0` which can be found out by using `arr.length`.

The question then becomes is how to initialize an empty array with `length = 5`?

We are going to learn following things from this problem:

- How to create arrays.
- How to create empty arrays.
- How to create arrays with values and empty slots.
- What happens when we access array with index that is not initialized.

Solution

Suppose we need 10 digits in an array. We write array like `var array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];`. This is fairly short and readable way to initialize an array.

`Array` in JavaScript have properties such as `length` which specifically tells us amount of elements it is holding.

Continuing from last example, `array.length` will get us `10`.

```
var names = ["Anubhav", "Jake", "Hyde", "Thorax"];
console.log(names.length);
```

Will give us 4 as there are four names in the array.

But when we go for an empty array, we get `length = 0` and this is the problem that we have to solve.

Code

```
var arr = [ , , , , , ];
```

Yup, `5` commas between `[` , `]` and you are all set.

If you go to console (say firefox's) by hitting F12, you will see that when you try to access this array it shows:

```
var arr = [ , , , , , ];

arr;
Array [ <5 empty slots> ]
```

Which tells us that the array has memory for five items.

You can also mix data with non data and create arrays that have spaces for future data.

```
var arr = [ 1, , 3, , 5, ];

arr;
Array [ 1, <1 empty slot>, 3, <1 empty slot>, 5 ]

arr[0]
1
arr[1]
undefined
arr[2]
3
```

In code above, length of the array `arr` is still 5, but only 3 slots are filled and 2 are empty.

Problem - Get longest name in an array

We are given a bunch of names: `["Ecma", "Script", "Hacker", "Fill Haac", "John Snokeet"]` and we want to know which one of the given names is longest one.

We are going to learn following things from this exercise:

- How Arrays can be sorted.
- How to provide our own implementation for comparing elements if we need to.
- How to sort strings in an array alphabetically or length-wise.
- How to get the last item out of an array via `pop()`.
- Functions can be written inside other functions.
- Functions can be passed as variables.

Solution

Logic

Think of array as storage space but for data and we access it usually sequentially. We have some stuff in our storage and we want to move things around.

We are going to move things based on their size. In our exercise, size of stuff can be calculated by `names[index].length` and if we compare every datum with other, we can find out the longest name in the array.

For every name, get it's length. Compare length to other name's length. Get the index of the name that has longest length. Return the name @ that index in names.

Pseudo code *This is hint of how to solve the problem.*

```
maxLengthName = first name in names.
for-each( name in names from second name ) {
    if name.length is greater than other name's length
        update maxLengthName to current name.
}
return maxLengthName
```

Which would translate into JavaScript as:

Code *A monstrous implementation*

```
function getLongestName(names) {
    var i = 0;
    var maxLengthNameIndex = 0;
    for(i = 1; i < names.length; i++) {
        if(names[i].length > names[maxLengthNameIndex].length){
            maxLengthNameIndex = i;
        }
    }
    return names[maxLengthNameIndex];
}

var names = ["Ecma", "Script", "Hacker", "Fill Haac", "John Snokeet"];

getLongestName(names);
"John Snokeet"        // I got this output since I am in FireFox's console.
```

You can understand the code easily. All we are doing is:

- Assuming first name is longest. `// var maxLengthNameIndex = 0;`
- Looping from second name to last
    - If length of name comes longer than first names'
        - We update `maxLengthNameIndex` with current name's index.
- Return name with longest length.

Advanced, Sexy but Inefficient Solution

Code *Short-Sweet-and-Sexy way of doing the thing above*

```
function getLongestName(names) {
    return names.sort(function(a, b){ return a.length - b.length; }).pop();
}
```

I guess I can break it down a bit.

```
function getLongestName(names) {
    return names.sort(
        function(a, b) {
            return a.length - b.length;
        }
    ).pop();
}
```

We can get compare function out of sort call and re-write it in getLongestName function itself.

```
function getLongestName(names) {
    function compareByLength(a, b) {
        return a.length - b.length;
    }
    return names.sort(compareByLength).pop();
}
```

Note: *All the examples above (in advanced section) deals with Function-In-Function scenario. This is very important tool and in future we will find it almost indispensable.*

We can de-tangle two functions and write a bit more readable code as follows.

Code *compare functionality moved outside of* `getLongestName()`

```
function compareByLength(a, b) {
    return a.length - b.length;
}

function getLongestName(names) {
    return names.sort(compareByLength).pop();
}
```

I wouldn't go into how compare helps except that you can rewrite compare as follow to have same effect.

Code *A bit more verbose version of compare function.*

```
function compareByLength(a, b) {
    if(a.length < b.length) {
        return -1;
    } else if(a.length > b.length) {
        return 1;
    } else {
        return 0;
    }
}

function getLongestName(names) {
    return names.sort(compareByLength).pop();
}
```

There's a catch! Sorting whole array just to find highest/longest element is an overkill. It can be justified for arrays containing few elements; but for a sizable array this will be slower, inefficient and not necessary.

Only benefit to sorting array is when there are going to be next queries on sorted data set.

Example:

```
[ "Range", "Strange", "Rave", "Brave", "Is", "New", "Stupid" ]
```

If you sort array on first query like "get the smallest word". Then "get the largest word" becomes a constant time operation.

Advanced and Efficient solution

```
function getLongestName(names){
  var longest = names[0];
  names.forEach(function(x) {
    if(longest.length < x.length){
      longest = x;
    }
  });
  return longest;
};

getLongestName(names);
```

A challenge for you: Find a peculiar behavior of this program when there are names that have same length.

Problem - Replace a character in a string

We are given a string, say, `var str = "It's my turn now, it's my turn now.";` and we want to replace every `y` with `e`. Don't know why we wanna do that; it's a good song. But that's the problem. Replace a given character in a given string.

Solution

`var newString = str.replace('y', 'e');`

The code above works. It replaces the `y` with `e`, but with a problem: It doesn't replace all of the `y`'s into `e`s.

Code - *Running in FireFox console.*

```
var str = "It's my turn now, it's my turn now.";
undefined

var newString = str.replace('y', 'e');
undefined

newString
"It's me turn now, it's my turn now."
```

So, what we do?

Well, we can call replace again on the returned value.

`var newString = newString.replace('y', 'e');`

But you will see that it's a lousy way to do so. Obvious question is: what if there are more than 2 `y`s?

Though we can put the replacement in a loop with terminating condition being string not equal to last iteration.

Code - *Loopy way to write char-replacement code*

```
function Replacer(str, toReplace, replaceWith){
    var newString = str;
    var oldString = "";
    while(newString !== oldString){
        oldString = newString;
        newString = oldString.replace(toReplace, replaceWith);
    }
    return newString;
}


Replacer("It's my turn now, it's my turn now.", 'y', 'e')
"It's me turn now, it's me turn now."
```

As we can see the code and logic work. But this is not a good way to do so, even though it solves the problem completely.

We have to move to advanced topic: Regular Expressions.

Advanced Solution

Check this code below, if you are unfamiliar with RegEx, you are going to be amazed at it's succinct-ness.

Code - *Replace all instances of a character in a string*

```
function Replacer(str, toReplace, replaceWith) {
    var toReplaceRegex = new RegExp(toReplace, "g");
    return str.replace(toReplaceRegex, replaceWith);
}
```

A word of caution: Remember we are replacing characters. Can you find ways to hack this code, by putting weird input perhaps? What other ways we can break this function? What do you think about capitalization and punctuation?

Now let's see this function's output.

Output

```
Replacer("It's my turn now, it's my turn now.", 'y', 'e')
"It's me turn now, it's me turn now."
```

In two lines, we have managed to remove all the instances of `y` with `e`.

This is cool.

You'll need to read up on:

- RegExp
- filters for RegExp: g, i, etc.

Problem - Convert HTML entities to character entities

You are given an input from user as `<big>I am great @ "javascript". Best's Best.!</big>`. As you can see that it contains, `<`, `>`, `'`, `"`, `@` and `!`. We might do not care about it here, but servers do. Suppose, a user pasted a malicious script in comment section and web server serves that as output to every other user. Do you think it will create a security risk?

That's a rhetorical question. Of course, it does. Moving on.

We need to sanitize the input from the user and make it digestible for web server.

Using Character Entity reference chart we are going to convert unsafe characters to their safer versions.

Solution

One way to code this problem is as we saw in `Replace a character in a string`.

```
function sanitize(str) {
  return str.replace(/&/g, '&amp;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;')
            .replace(/"/g, '&quot;')
            .replace(/'/g, '&apos;');
}
```

Code - *Sanitizes unsafe characters*

But, this code suffers from a problem. It's too specific. What if we do not want to sanitize `'`? or `"`. If there's no variability in our requirements then this code is fine. But if there is, it is not a good way to handle such requirements.

A better way to do so would be to create small functions that do one thing. That we can compose our solution as we want.

```
var
sanitizeAmpersand = function(input) { return input.replace(/&/g, '&amp;'); },
sanitizeGreaterThan = function(input) { return input.replace(/>/g, '&gt;'); },
sanitizeSmallerThan = function(input) { return input.replace(/</g, '&lt;'); },
sanitizeQuote = function(input) { return input.replace(/"/g, '&quot;'); },
sanitizeApos = function(input) { return input.replace(/'/g, '&apos;'); };

var sanitized = sanitizeSmallerThan(sanitizeGreaterThan(sanitizeAmpersand("<big>I am

sanitized
"&lt;big&gt;I am great @ "javascript". Best's Best.!&lt;/big&gt;"
```

Code - *The way we want to work if we have variability in our requirements.*

One way to handle the variability and ease of coding is to hack the `String` variable.

```
String.prototype.sanitizeAmpersand = function() { return this.replace(/&/g, '&amp;');
function String.prototype.sanitizeAmpersand()
"You & I".sanitizeAmpersand()
"You &amp; I"
```

Code - *To test if hacking String is possible - yes it is.*

This way we can hack string prototype and write our code and get the output fairly straight forward way.

```
String.prototype.sanitizeAmpersand = function() { return this.replace(/&/g, '&amp;');
String.prototype.sanitizeGreaterThan = function() { return this.replace(/>/g, '&gt;')
String.prototype.sanitizeSmallerThan = function() { return this.replace(/</g, '&lt;')
String.prototype.sanitizeQuote = function() { return this.replace(/"/g, '&quot;'); };
String.prototype.sanitizeApos = function() { return this.replace(/'/g, '&apos;'); };


var input = "<big>I am great @ \"javascript\". Best's Best.!</big>";
undefined

input.sanitizeAmpersand().sanitizeGreaterThan().sanitizeSmallerThan()
"&lt;big&gt;I am great @ "javascript". Best's Best.!&lt;/big&gt;"

input.sanitizeAmpersand().sanitizeGreaterThan().sanitizeSmallerThan().sanitizeQuote()
"&lt;big&gt;I am great @ &quot;javascript&quot;. Best's Best.!&lt;/big&gt;"

input.sanitizeQuote().sanitizeApos()
"<big>I am great @ &quot;javascript&quot;. Best&apos;s Best.!</big>"
```

But this is not a good solution.

We are hacking `String` and wishing that no one else has done so. If some other developer or library decides to hack `String` , one of us would be out of luck. Every now and then we'd get a strange output. A strange output is the least of our problems. A wrong output might wipe out something important or just get's thrown out or blocked by server.

We need to find a way where we return a string after every such operation and we do not end up cluttering `String` .

TODO: Add more content here.

Problem - FizzBuzz

You are given numbers 1 to 100. You have to print `fizz` , `buzz` or `fizzbuzz` as per given following conditions.

1. If number is multiple of 3, print out `fizz` .
2. If number is multiple of 5, print out `buzz` .
3. If number is multiple of 15, print out `fizzbuzz` .
4. If nothing of above conditions satisfy then just print out the number.

There is nothing more to do and nothing less will work.

Solution

A very simple solution is hidden in the problem itself.

```
function FizzBuzzEr(){
    for(var i = 1; i <= 100; i+=1){
        if(i % 15 === 0){
            console.log('fizzbuzz');
        } else if(i % 5 === 0){
            console.log('buzz');
        } else if(i % 3 === 0){
            console.log('fizz');
        } else {
            console.log(i);
        }
    }
}

FizzBuzzEr();
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
16
17
fizz
....
many other lines
```

Code - *A very first and simple solution to fizzbuzz problem.*

You see, that it works; but it's very absurd. What happens when we add another option? Even before that, the code we wrote is rather ugly.

Even though ugly it is better than the following one.

```
function FizzBuzzEr(){
    for(var i = 1; i <= 100; i+=1){
        if(i % 3 === 0){
            console.log('fizz');
        } else if(i % 5 === 0){
            console.log('buzz');
        } else if(i % 15 === 0){
            console.log('fizzbuzz');
        } else {
            console.log(i);
        }
    }
}
```

Code - *Challenge - What is wrong with this code?*

You have to figure out what is wrong with code above and that is your task. Keep in mind that it is your task and you have to do it. It should be done by you and alone. You shall not seek any help on this task and neither shall it be provided to you.

Moving on.

There is better way to handle this problem.

A bit better solution.

```
function FizzBuzzEr(){
    var output = '';
    for(var i = 1; i <= 100; i+=1){
        if(i % 3 === 0){
            output += 'fizz';
        }
        if(i % 5 === 0){
            output += 'buzz';
        }
        output = output.length === 0 ? i : output;
        console.log(output);
        output = '';
    }
}

FizzBuzzEr();
1
2
fizz
4
buzz
fizz
7
8
fizz
buzz
11
fizz
13
14
fizzbuzz
16
17
fizz
```

Code - *Challenge - How does this piece of code works?*

The code above works. It gets loaded into the memory and then executes finely. After which, it creates the same series of fizz buzz fizzbuzz and numbers as the first program did. You have to figure out what's it doing.

Challenge - *What if add the condition for* `i % 15 === 0` *and then put* `output += "fizzbuzz";` *?*

A kick-ass way of solving FizzBuzz

Instead of we sealing fate of FizzBuzzEr function by letting it squeak `fizz` , `buzz` , `fizzbuzz` or number only, we can give it superpower to deal with the changing requirements.

```
var FizzBuzzOptions = [
    function(num) { return num % 3 === 0 ? 'fizz' : ''; },
    function(num) { return num % 5 === 0 ? 'buzz' : ''; }
];

var FizzBuzzEr = function(options){
    var output = '';
    for(var i = 1; i <= 100; i++){
        options.forEach(function(x){
            output += x(i);
        });
        output = output.length === 0 ? i : output;
        console.log(output);
        output = '';
    }
}

FizzBuzzEr(FizzBuzzOptions);
```

Code - *KickAss version of FizzBuzz*

Go ahead and run this code. This works and is beautiful.

Advantages of the code above are:

1. You can switch the fizz, buzz at your will.
2. You can add more conditions as you like. All you need to do is modify `FizzBuzzOptions` or pass your own Options.
3. We have separated the actual-work from logic. We are providing logic in `options` argument.

That's pretty much it about FizzBuzz.

Problem - Implement stack.

You will be given data in a comma delimited string and you will have to implement a stack based on the values.

We will learn following thigs:

- How to `split()` a string.
- How to create stack in JS.
- How to hide data and make it private.

Solution.

We need to break down the problem.

1. Break string on comma.
2. Use broken string data as input for our stack.
3. Deploy push() and pop() on our stack object.

Let's just break the string.

```
function getData(inputString) {
    var values = inputString.split(',');
    return values;
};
```

Code - *returns the values after breaking a comma delimited string.*

If it was any other language, we might be writing the object required for implementing stack or using built in one. But since it's JS, we don't need to. Let's implement our functionality for operating a stack.

```
function Stack(inputString) {
    this.stack = getData(inputString);
}

Stack.prototype.pop = function() { return this.stack.pop(); }
Stack.prototype.push = function(val) { this.stack.push(val); }
```

Code - *Implementation of stack in JS.*

That's It! We can reduce the functionality even further as we do not need a `getData()` function. We need to provide a display function to print stack and a way to join it too. We need to hide the data too.

```
function Stack(inputString) {
    this.stack = inputString.split(',');
}

Stack.prototype.pop = function() { return this.stack.pop(); }
Stack.prototype.push = function(val) { this.stack.push(val); }
Stack.prototype.join = function(joinVia) { return this.stack.join(joinVia); }
Stack.prototype.display = function(processor) { this.stack.forEach(processor); }

var s = new Stack("A,B,C,D,E,F,G,H");
undefined

s.pop();
"H"

s.push(10); s.push("Anubhav");
undefined

s.display(function(x){ console.log(x); } );
undefined
A
B
C
D
E
F
G
10
Anubhav

s.join('-');
"A-B-C-D-E-F-G-10-Anubhav"
```

Code - *`display()` , `join()` implmented. Security left.*

We have managed to display the data easily and join the data easily. What we are missing is security. We can access `s.stack` out in the open.

```
s.stack
Array [ "A", "B", "C", "D", "E", "F", "G", 10, "Anubhav" ]

s.stack = [1,2,3,null, undefined, "great", "A"]
Array [ 1, 2, 3, null, undefined, "great", "A" ]

s.join('-')
"1-2-3---great-A"
```

Execution of code above - *shows that we can access internal variable easily and change it as we need, maliciously or otherwise.*

Below is the secure version. If doesn't make sense at first, do not be disheartened. Pick apart the code piece by piece.

```
function Stack(input) {
    var data = input.split(',');
    return (function(store) {
        return {
            pop: function() { return store.pop(); },
            push: function(x) { store.push(x); },
            display: function() { store.forEach(function(x) { console.log(x); }); },
            join: function(c) { c = c || '-'; return store.join(c); }
        };
    })(data);
}

var stack = new Stack("a,b,c,d,e");
stack.pop()
"e"

stack.display()
a
b
c
d

stack.join()
"a-b-c-d"

stack.data
undefined
stack.store
undefined

var r = Stack('a,b, c,d,e');

r.join()
"a-b- c-d-e"

r.data
undefined

r.store
undefined
```

Code - *Shows that stack's data/input now cannot be retrieved via instance.*

The code above is secure in a sense that the data cannot be tempered with via direct access. Thus, we have learned a way to make data private.

Problem - Until you find meaning of life.

Given is an array of numbers and you have to process each number until you see 42. As soon as you see 42, stop the whole process. You have found the meaning of life and everything else, go home, sip mohitos and relax.

Given:

- An array of numbers as input argument data.
- A function `processor` that takes a number.
- Write a function named `untilAnswer()` that will run and find solution.

Solution

Let's break down the problem.

An array of number is given as argument `data` a function named `processor` and our function has to be named `untilAnswer`.

```
function untilAnswer(data, processor) {
    //functionality here.
}
```

Code - *Basic structure.*

Now, we need to iterate `data` array that will produce the number turn-by-turn and feed that number to `processor` function.

`forEach` seems like a good enough function for iteration and passing down the values.

```
function untilAnswer(data, processor) {
    data.forEach(function(x){
        if(x === 42){
            return;
        }
        processor(x);
    });
}
```

Code - *Done! But...*

This, code above, is a clean solution; except that it's inaccurate.

Try running it for the cases where there are numbers other than 42 after a 42 in an array e.g. `[1, 2, 3, 4, 5, 42, 43, 44]` .

This teaches a valuable lesson about `forEach` and concept of `Function-In-Function` . `forEach` runs for every element in the given array. When we return from the inner function it returns, without processing, only when input `x` is 42. In all other cases, it will process the number passed to it.

We need to keep track of whether we have seen a 42 or not.

```
function untilAnswer(data, processor) {
    var seen = false;
    data.forEach(function(x){
        if(x === 42){
            seen = true;
        }
        if(!seen){
            processor(x);
        }
    });
}
```

Code - *Done! But...*

We have used a boolean variable `seen` to know if we have seen a 42 and then if we have, we do not do anything.

Figure Out: *There is a problem with the code above that you have to figure out. Think what* `forEach` *does and why our processing is inefficient.*

Below is a better way to do what we did above. It's clean and accurate. Take a look at code above and below and figure out the difference.

```
// var input = [1, 2, 88, 42, 99];

var untilAnswer = function (data, processor) {
    for(var i = 0, len = data.length; i < len && data[i] !== 42; i += 1) {
        processor(data[i]);
    }
}

// var processor = function(x) {
//    console.log(x);
// }

// untilAnswer(input, processor);
```

Code - *Right way to go about doing it.*

Problem - Generate range of numbers.

We need to create a function `range` that generates numbers in the range `x` and `y` passed to it.

Conditions

- It should be able to handle negative to positive, positive to positive, positive to negative, negative to negative number generation.
- It should be able to generate numbers that are `step` argument away.
- Numbers generated should be inclusive.

In this problem you will learn:

- How to handle wrong inputs.
- Moral dillemmas of choosing solution strategy.
- Issues that arise from a bad design choice.
- How to create `range` functionality modularly.

Solution

Let's start by writing down the conditions.

Case	start from	end at	step	action	Example
1	zero	positive	positive	generate	range(0, 10, 1) => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2	zero	negative	negative	generate	range(0, -4, -1) => [0, -1, -2, -3, -4]
3	zero	positive	negative	fail	range(0, 10, -1) => throw error
4	zero	negative	positive	fail	range(0, -4, 1) => throw error
5	positive (smaller)	positive (greater)	positive	generate	range(9, 10, 1) => [9, 10]
6	negative (bigger)	negative (smaller)	negative	generate	range(-3, -4, -1) => [-3, -4]
7	positive (greater)	positive (smaller)	negative	generate	range(10, 9, -1) => [10, -9]
8	negative (smaller)	negative (bigger)	positive	generate	range(-10, -9, 1) => [-10, -9]
9	positive (smaller)	positive (greater)	negative	fail	range(9, 10, -1) => throw error
10	negative (bigger)	negative (smaller)	positive	fail	range(-3, -4, 1) => throw error
11	positive (greater)	positive (smaller)	positive	fail	range(10, 9, 1) => throw error
12	negative (smaller)	negative (bigger)	negative	fail	range(-10, -9, -1) => throw error

Table - *Conditions of operation of program.*

Now, we can begin. We need to generalize the above 12 cases.

We can put cases in two categories by their execution result: Failure and Success.

We can generalize that:

step value	start <relation> end	output
positive	start < end	generate
positive	start > end	fail
negative	start > end	generate
negative	start end	fail

This creates two specific issues for us:

1. What to do in case where step is zero?
2. What to do in case where start is equal to end?

We are going to define what does it means to have step equal to zero.

When step is zero, we know the start and end but we are being told that we do not have to move in any direction. We can interpret this in two ways.

- Return [start, end] or
- Throw error.

Let's tackle first interpretation. When step is zero, we cannot move in any direction. That means we cannot generate range. But, we know that we are supposed to generate a range that includes starting and ending numbers. This is a moral dilemma. *The way we have structured our program, we have created this problem*. Consider if we have made conditions in which both the numbers were not included:

> range(0, 10, 1) generates [1, 2, 3, 4, 5, 6, 7, 8, 9]

But, instead we went for:

> range(0, 10, 1) generates [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

And thus our problem right now is:

Should we generate `[1, 10]` or `throw error` for `range(1, 10, 0)` ?

I leave you to decide your implementation. I am going to throw error, rationale being that it's not returning a range. You could argue against it and I welcome it.

This brings us to next thing.

When start === end, this pose another moral problem.

> Should we generate `[10]` or `throw error` for `range(10, 10, anyvalue-for-step)` ?

Again, we have ourselves created this problem. If we had excluded the end or start or both, we could have avoided this problem.

I am going to generate `[10]` . You are welcome to critize this decision and come up with your own.

```
var range = function(startFrom, endAt, step) {
    var result = [];
    if(step === 0) {
        throw new Error('Step cannot be zero');
    } else if(step > 0) {
        /* Tackle cases where we are going upwards from start to end. */
    } else if(step < 0) {
        /* Tackle cases where we are going downwards from start to end */
    }
};
```

Code - *Skeleton of what we are going to do.*

We are here. We are again at the decision-intersection: should we put all of the code in single `range` function or create multiple functions?

I am going to create multiple ones. You can choose whichever suits you best.

```
var rangeUpwards = function(start, end, step) {
    var result = [];
    for(var i = start; i <= end; i += step) {
        result.push(i);
    }
    return result;
};
var rangeDownwards = function(start, end, step){
    var result = [];
    for(var i = start; i >=end; i += step) {
        result.push(i);
    }
    return result;
};
var range = function(start, end, step) {
    var result = [];
    if(step === 0) {
        throw new Error("Step cannot be zero.");
    } else if(step > 0) {
        if(start <= end){
            return rangeUpwards(start, end, step);
        } else {
            throw new Error("Start cannot be greater than end, if stepping is positiv
        }
    } else {
        if(start >= end) {
            return rangeDownwards(start, end, step);
        } else {
            throw new Error("Start cannot be smaller than end, if stepping is negativ
        }
    }
};
```

Code - *Full functionality*.

```
range(0, 10, 1)
Array [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1 more… ]

range(10, 0, -1)
Array [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 1 more… ]

range(10, 10, 1)
Array [ 10 ]

range(0, 10, 0)
Error: Step cannot be zero.

range(10, -10, 1)
Error: Start cannot be greater than end, if stepping is positive.

range(-10, 10, -1)
Error: Start cannot be smaller than end, if stepping is negative.
```

Output - *As expected.*

Let's go back to code for while. I hope you noticed that `rangeUpwards` and `rangeDownwards` are exactly the same except terminating condition. I want to tell you that it's deliberate till now. I wanted you to understand that in both cases, we need to go from start to end and step needs to be added.

We need to pass the terminating condition as argument.

Now that you know, you can remove the redundant functionality. You can try to club the conditions together too.

Here's how I'd do it.

```
var generateRange = function(start, step, endCondition) {
    var result = [];
    for(var i = start; endCondition(i); i += step) {
        result.push(i);
    }
    return result;
};

var range = function(start, end, step) {
    var result = [];
    if(step === 0) {
        throw new Error("Step cannot be zero.");
    } else if(step > 0) {
        if(start <= end){
            return generateRange(start, step, function(i) { return i <= end; });
        } else {
            throw new Error("Start cannot be greater than end, if stepping is positiv
        }
    } else {
        if(start >= end) {
            return generateRange(start, step, function(i) { return i >= end; });
        } else {
            throw new Error("Start cannot be smaller than end, if stepping is negativ
        }
    }
};
```

Code - *Full functionality*.

```
range(0, 5, 1)
Array [ 0, 1, 2, 3, 4, 5 ]

range(10, 5, -1)
Array [ 10, 9, 8, 7, 6, 5 ]

range(10, 10, 0)
Error: Step cannot be zero.

range(10, 10, 1)
Array [ 10 ]

range(10, -10, 1)
Error: Start cannot be greater than end, if stepping is positive.

range(-10, 10, -1)
Error: Start cannot be smaller than end, if stepping is negative.
```

We have successfully generated a range function that covers all or our cases.

Problem - Returning arithmetic functions based on input operators.

Input format: string containing, on operator and two operands.

Example: 10+20; 30-49; 19*9 etc.

Output: 30 (`function definition`); -11 (`function definition`); 171 (`function definition`) etc.

In this problem we will learn:

- How to get numbers out of string using regex.
- How to find either of +,-,*,/,% in string via regex.
- How to return a function.
- How to do a multiway switch.
- How to sort an array of numbers.
- How to sort an array of strings.

Solution for Returning arithmetic functions based on input operators.

Logic

The problem can be divided in two sub-problems.

1. Find what operator is used.
2. Return appropriate function.

We will need Regular Expressions for first sub-problem. For the second problem we need to understand `switch` and functions.

Let's start from functions. Functions are first order citizens of JS language. They can be stored as variables, called via another variable, returned from functions and passed around in/to functions.

One simpler example is sorting an array.

```
var arr = range(10, 1, -1);
/* range is from Generate range of numbers problem */
arr;
Array [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]

function comparer(a, b) { return a-b; }
var sortNumberFunction = comparer;
arr.sort(sortNumberFunction);

arr;
Array [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

Code - *Example where we pass a function as an argument*.

But we have yet to return a function. Let's make a decider-function that looks at the type of elements in an array and then returns appropriate compare function.

```
function decideWhichCompare(arr) {
    var compareFunction = function() { };
    if(typeof arr[0] === typeof 10){
        compareFunction = function(a, b) { return a - b; };
    } else if(typeof arr[0] === typeof "string") {
        compareFunction = function(a, b) { return a.length - b.length; };
    }
    console.log("Returning: ", compareFunction);
    return compareFunction;
}

var numbers = range(10, 1, -1);
var strings = ["He", "Like", "I", "Likes", "JavaScript", "."];

var sortMyData = function(dataArray) {
    var compareFunction = decideWhichCompare(dataArray);
    dataArray; //Since I am in FireFox's console.
    dataArray.sort(compareFunction);
    dataArray; //Since I am in FireFox's console.
};

sortMyData(numbers);
sortMyData(strings);
```

Code - *That returns a function based upon input variable's data type*.

We are successful in returning a function in JavaScript. We can use this technique to solve our problem. Finally!!!

```
var whichFunction = function(op) {
    switch(op) {
    case "+": return function(a, b) { return a + b; };
    case "-": return function(a, b) { return a - b; };
    case "*": return function(a, b) { return a * b; };
    case "/": return function(a, b) { return a / b; };
    case "%": return function(a, b) { return a % b; };
    default: return function() { console.log("Unknown operation."); };
    }
}
```

Code - *Function that returns arithmetic functions based upon input operators*.

Now, we need to figure out a way to know which `op` is passed to us in the input string.

```
var outputFunction = function(input) {
    var regex = new RegExp(/(\d+)([+-/*%])(\d+)/);
    var match = input.match(regex);
    var opFunction = whichFunction(match[2]);
    console.log(opFunction(parseInt(match[1]), parseInt(match[3])), "(" + opFunction
};

outputFunction("10+20")
30 (function (a, b) { return a + b; })

outputFunction("10/20")
0.5 (function (a, b) { return a / b; })

outputFunction("10%20")
10 (function (a, b) { return a % b; })

outputFunction("10-20")
-10 (function (a, b) { return a - b; })

outputFunction("10*20")
200 (function (a, b) { return a * b; })

outputFunction("10/0")
Infinity (function (a, b) { return a / b; })

outputFunction("0/0")
NaN (function (a, b) { return a / b; })
```

Code - *Working code*.

TODO: Add explanation why match works with index above as it did. TODO: Explain how passing "10+20+30" will not result in 60 but only 30. First operation will be performed.

Problem - Summing it up.

We have general sum function.

```
function sum(x, y) {
    return x + y;
}
```

We want to be able to add a lot more numbers, but right now we can add only two numbers at a time.

1. Make us able to call sum function like `sum(2, 3, 4, 5, 6, 7, 8)` and get the output.
2. Make us able to create functions that add with a specific number, `var add4To = sum(4);` should create a new function such that `add4To(6) === 10`.

Solution

Initial code is fixed-arity code. We need to fix that first, so we will use arguments.

```
var sum = function(start) {
    var sum = 0;
    for(var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
```

Code - *Where we find the sum of the numbers given to us.*

The code above has a flaw. We are not using `start`. Let's quickly fix that.

```
var sum = function(start) {
    var sum = start;
    for(var i = 1; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
```

Code : *Fixed code.*

[TODO:]

Document Object Model

JavaScript is a programming language. HTML is a markup language. DOM is bridge between the two. DOM is what glues browsers and users together in an interactive web application. Without DOM, a web page is just a text document.

DOM + JavaScript provide us the ability to create interactive web applications.

Problem - Fetch all the elements of a specific type from the loaded page.

We want to fetch all the `div` in the web page.

We are going to use Developer's Console that you can activate via F12 or browser settings or browser menu.

Solution

```
var allDivsInPage = document.getElementsByTagName('div');
undefined

allDivsInPage
HTMLCollection [ <div#newtab-customize-overlay>, <div.newtab-customize-panel-containe
```

Code - *Get all the `div` element in the document.

We can see that, the page I had, had more than 51 `div` elements on it. Printing them here doesn't make any sense, so we will skip it. We are going to *try* still; to make a point.

```
document.getElementsByTagName('div').forEach(function(x) {
    console.log(x);
})
TypeError: document.getElementsByTagName(...).forEach is not a function
```

Code - *Fails because returned value though look like an array, is not an array*

That's a bummer. Does it have a length property?

```
document.getElementsByTagName('div').length
61
```

Code - *Shows that our object does have a property named length, but doesn't iterate over it.*

Well, now if you remember we did a similar problem in Chapter 1. In that problem we had an object that was not array, had a length property and access via numerical indices.

We need to check if it can be accessed via index.

```
document.getElementsByTagName('div')[0]
<div id="newtab-customize-overlay">
```

Code - *Shows us that data can be accessed via index*

Now, we know the trick.

```
[].forEach.call(allDivsInPage, function(x) { console.log(x); })
undefined
<div id="newtab-customize-overlay">
<div class="newtab-customize-panel-container">
<div id="newtab-customize-panel" orient="vertical">
...
...
... a lot of lines
```

Code - *Shows that we can still loop through the returned array-like object.*

Chapter 3 - Canvas API

Canvas API is what HTML 5 brought to us: a playground for creating lines and shapes in browser. It can do more.

Advantages

- It let's you create paint like applications in browsers.
- It's accessible via JavaScript, thus making it a good tool for us JS developers.

Disadvantages

- It is stateful.

We will learn Canvas API via projects. One such project is JSPaint; I wrote js-paint application to learn about canvas api.

We will go through the history of projects and re-create our own applications.

Chapter 4 - CoffeeScript

CoffeeScript is a language that compiles into JavaScript, which makes it JS on steroids. It's cool and takes less space.

Advantages

- Compiles into correct JavaScript.
- Once mastered, you won't want to go back to just JavaScript.

Disadvantages

- One more style of syntax to be learned.
- Learning half of it makes for very buggy output.

Resources

In this chapter, you will find all the resources you need about JavaScript.

- jQuery
    - Source code

jQuery

You can start from here.

1. Official site
2. Source code

Go ahead.

Building jQuery source code.

1. fork the source code.
2. `git clone` it.
3. `cd` to dir && `npm install` && `grunt` it.

There you have freshly made jQuery in your `dist/` folder.

What is jQuery?

- It is a cross browser library.
- It has very good API.
- It is (a tad bit) slower than just-JavaScript.
- It helps in web development greatly.
- It is lightweight.
- It is CSS3 compliant.

From jQuery.com

> jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

EcmaScript Programming Style Guides

These guides are from online resources.

There are two main parts:

- Programming Style Guide.
  > These guides tell how to style your code. These are choices.
  >
  > - For example, `function (){ };` vs `function () { };` - notice the space after `)`.
- Programming Language Guide.
  > These guides tell what to use and what not to use from language.
  > These are general truths; and pain points if ignored.
  >
  > - Do not use `with`. It's context dependent.
  > - Use `var` in ES 5. Not using it is context dependent.

There are guides for version 5 and onward:

- ES 5
- ES 2015 (ES 6 / JS 6 - please don't use these names)
- ES 2016
- ES 2017

Programming Style Guide for ES 5

> Note: This is not preaching. You are free to pick what you like. End goal is to stick to what you pick. Look, **programming is hard**. There are enough problems to solve already; do not create problems by `reckless patching`. Pick a style that suits you and your team and you **intended code-audience** and `stick to it`.

This guide serves a simple purpose: to let you have consistency in your code.

The information in this chapter can be summed up in two categories: `Do` and `Don't Do`.

Table - *Do and Don't do of JavaScript.*

Do	Don't
Use var to declare local variables.	Initialize variables in function without var.
Use camelCase for variables and functions.	Mix camelCase with PascalCase and under_score_case.
Use a `Constants` Object to define constants.	Use UPPER_CASE_FOR_CONSTANTS for constants. Really?
Get rid of `eval()` and `with()` if you don't really **need** them.	Use `eval()` to execute user-input code.
Use `options` object if argument list becomes lengthy.	Create functions and APIs with 10, 20 etc. arguments. That's just competing with older Windows API.
Create your own subtypes that consume built in types.	Add stuff to built in objects.
Put 'use strict'; at top of your script.	Ditch strict mode because it's easy.
Use closures to hide access to data.	Put everything at global scope or outside it's intended usage scope.

Table - *Do and Don't do of Programming in general.*

Do	Don't
Talk to your team.	Force your team to use your style.
Write functions to avoid code duplication.	Copy-paste code.
Create as many objects as logically required.	Put everything in a master object or create objects out of free will.
Practice DRY principle	Copy-paste-hack code.
Practice YAGNI principle mercilessly.	Create interfaces that you might one day need.

Use `var`.

Use var for variables. Without them some environments put variables in global scope. Bad Bad Thing.

Use camelCase.

Use camelCase for variables and functions. Objects and Constructor functions should be kept PascalCased. JavaScript doesn't understand hyphenated-casing, so that's a bust. However underscore_casing is not a bad idea generally, but people are accustomed to see camelCase every where in JS.

Use Constants Object for constants.

Contrary to popular belief in and status of UPPER_CASE_CONSTANTS, use a Constants object and put your constants there.

Semi-Colons.

The rules related to being able to deliberately leave a semi-colon are not that much. But please don't. Use semi-colons as if they were mandatory. There is no point in showing off your special-case-retentive-memory.

Nested functions.

JavaScript wouldn't be JavaScript without them.

Wrapper objects for primitive types.

Please don't.

`eval()` .

Please don't.

`with()`

Please don't.

BackSlash for multiline string.

Please don't.

Adding stuff to prototypes of built-in types.

Please don't.

Argument list.

Upto three arguments are fine in argument list. If argument list contains more than three arguments, consider options object.

Defining functions in a loop.

Please don't.

Google Style Guide says:

> BE CONSISTENT.
>
> If you're editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around all their arithmetic operators, you should too. If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.
>
> The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you're saying rather than on how you're saying it. We present global style rules here so people know the vocabulary, but local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Avoid this.

ES 5 - Programming Style Rules

ES 5 - Programming Language Rules

> Use `var` for variables that you declare in functions.

- Cleaner `global` object.
- `var` makes variables local.
- Not using `var` makes your code context dependent and fragile.

var

What happens when you do not use var?

In **non-strict / sloppy mode**, your variable is defined in global scope. In browsers, this is usually `window`.

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

console.log(add(10, 11), sum, window.sum, sum === window.sum);

21 21 21 true // both in firefox/firebug and chrome console.
```

Code - *Above code manages to create a sum variable in global scope i.e. window.*

In **strict mode**, not using `var` to declare a variable is an error.

```
(function(){
  'use strict';
  function add2(a, b){
    sum2 = a + b;
    return sum2;
  }

  console.log(add2(10, 11), sum2, window.sum2, sum2 === window.sum2);

  ReferenceError: assignment to undeclared variable sum2 // In firefox/firebug

  Uncaught ReferenceError: sum2 is not defined // In chrome console.

})();
```

Code - *Above code manages to throw an error in strict mode.*

> try with: `var sum2 = a + b;` instead of `sum2 = a + b;` .

In strict mode, we are not even going to get output without var. Function itself will throw error on execution. If we add var, then output of function will be achieved, but even then, that variable is going to stay local. That means less clutter at global level.

Output of not using `var` depends upon whether your code is in **sloppy** or **strict** mode. Your code might not have freedom of knowing which mode it will execute in. Thus, making your code fragile. Don't code fragile.

Verdict

> Use `var` for variables that you declare in functions.

- Cleaner `global` object.
- `var` makes variables local.
- Not using `var` makes your code context dependent and fragile.

ES 2015

ES 2016