



learnersbucket.com

JavaScript

Interview

Guide



with 120+ Solved  
Interview Questions



**PRASHANT YADAV**

Senior Software Engineer

## Introduction

I am neither an expert book writer nor a great frontend engineer, I am someone who is curious and trying to fill a few gaps.

There are many awesome books in the market that help you to get ready and crack the general Software Engineering or Backend interviews, but a very few with the focus on the Frontend. This is where the idea to create this book has hit.

In 2019, I decided to run a blog ([learnersbucket.com](https://learnersbucket.com)) with the sole aim of writing about frontend stuff that I couldn't find on the web. I started with Data Structures & Algorithms in JavaScript and then ES6 concepts and later solved solutions for frontend interview questions gathered over the time from the web.

Over 3 years I managed to get a decent exposure to my blog and had around 400 articles written. After getting some good writing experience I decided to consolidate things in the book format so that It will be easy to refer and use as a “JavaScript Interview Guide”.

This book contains

- 100 JavaScript questions
- 20 React questions
- 2 System Design questions

with solved solutions

Along with these, I have tried to explain a few important concepts of JavaScript and there are many that you will learn along with each question.

I don't own the questions, along with the solution many things in the book you will find are referred from different sources.

- Prashant Yadav (Senior Software Engineer)

© JavaScript Interview Guide | learnersbucket.com

2

JavaScript Interview Guide

Copyright © 2022 Prashant Yadav | Learnersbucket.com

All Rights Reserved.

Author - Prashant Yadav.

Self - Published by Prashant Yadav | Learnersbucket.com

ISBN - 978-93-5777-237-2

Published on - 1 st December 2022

No part of this book shall be reproduced or transmitted in any form or by Any means, electronic or mechanical, including photocopying, recording, or by any information retrieval system without written permission by publisher and/or author.

The information contained in this eBook is for informational purposes only. I am not a certified teacher. Any legal or educational advice that I give is my opinion based on my own experience. You should always seek the advice of a certified professional before acting on something that I have published or recommended.

[OceanofPDF.com](http://OceanofPDF.com)

# First Edition.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

3

## Legal Disclaimer & Copyright

*The information contained in this eBook is for informational purposes only. I am not a certified teacher. Any legal or educational advice that I give is my opinion based on my own experience. You should always seek the advice of a certified professional before acting on something that I have published or recommended.*

*The material in this guide may include information, products, or services by third parties.*

*Third-Party Materials comprise the products and opinions expressed by their owners. As such, I do not assume responsibility or liability for any Third-Party material or opinions.*

*The publication of such Third-Party Materials does not constitute my guarantee of any information, instruction, opinion, products, or services contained within the Third-Party Material.*

*The problems solved in this book and solutions provided are referenced from different articles on the web. I don't own them, you are free to use the code examples for your practice, but shall not be reproduced, transmitted, or sold in whole or in part in any form.*

*No part of this publication shall be reproduced, transmitted, or sold in whole or in part in any form, without the prior written consent of the author. All trademarks and registered trademarks appearing in this guide are the property of their respective owners.*

*By reading this eBook, you agree that I and/or my company is not responsible for the success or failure of your career/business*

*decisions relating to any information presented in this eBook.*

© JavaScript Interview Guide | learnersbucket.com

4

*Copyright ©2022 by learnersbucket.com All rights reserved.*

*Other than the code, No part of this book may be reproduced or used in any manner without the written permission of the copyright*

*owner except for the use of quotations in a book review.*

*For more information, please address:*

*books.learnersbucket@gmail.com*

**FIRST EDITION .**

© JavaScript Interview Guide | learnersbucket.com

5

Table of Content

Concepts

1.

[Closure](#)

2. [Array.reduce\(\) method](#)

3. [Promises](#)

4. [Function & this](#)

## JavaScript Questions

1.

[Promise.all\(\).polyfill](#)

2.

[Promise.any\(\).polyfill](#)

3.

[Promise.race\(\).polyfill](#)

4.

[Promise.finally\(\).polyfill](#)

5.

[Promise.allSettled\(\).polyfill](#)

6.

[Custom Promise](#)

7.

[Execute async functions in Series](#)

8.

[Execute async functions in Parallel](#)

9.

[Retry promises N number of times](#)

10. [Implement mapSeries async function](#)

11. [Implement mapLimit async function](#)
12. [Implement async filter function](#)
13. [Implement async reject function](#)
14. [Execute promises with priority](#)
15. [Dependent async tasks](#)
16. [Create pausable auto incrementor](#)
17. [Implement queue using stack](#)
18. [Implement stack using queue](#)
19. [Implement stack with min and max method](#)
20. [Implement two stacks with an array](#)
21. [Implement Priority Queue](#)
22. [Implement LRU cache](#)

© JavaScript Interview Guide | learnersbucket.com

6

23. [Implement debounce function](#)
24. [Implement debounce with immediate flag](#)
25. [Implement throttle function](#)
26. [Implement custom Instanceof](#)
27. [Check if function is called with new keyword](#)
28. [Implement hashSet](#)



29. [Create a toggle function](#)
30. [Create a sampling function](#)
31. [Make function sleep](#)
32. [Remove cycle from the object](#)
33. [Filter multidimensional array](#)
34. [Count element in multidimensional array](#)
35. [Convert HEX to RGB](#)
36. [Convert RGB to HEX](#)
37. [In-memory filesystem library](#)
38. [Basic implementations of streams API](#)
39. [Create a memoizer function](#)
40. [Method chaining - part 1](#)
41. [Method chaining - part 2](#)
42. [Implement clearTimeout](#)
43. [Implement clearInterval](#)
44. [Create a fake setTimeout](#)
45. [Currying - problem 1](#)
46. [Currying - problem 2](#)
47. [Currying - problem 3](#)
48. [Convert time to 24 hours format](#)

- 49. [Convert time to 12 hours format](#)
- 50. [Create a digital clock](#)
- 51. [Chop array in chunks of given size](#)
- 52. [Chop string in chunks of given size](#)
- 53. [Deep flatten object](#)
- 54. [Restrict modifications of objects](#)
- 55. [Merge objects](#)

© JavaScript Interview Guide | learnersbucket.com

7

- 56. [Implement browser history](#)
- 57. [Singleton design pattern](#)
- 58. [Observer design pattern](#)
- 59. [Implement groupBy\(\) method](#)
- 60. [Compare two array or object](#)
- 61. [Array iterator](#)
- 62. [Array with event listeners](#)
- 63. [Filter array of objects on value or index](#)
- 64. [Aggregate array of objects on the given key](#)
- 65. [Convert entity relation array to ancestry tree string](#)
- 66. [Get object value from string path](#)

67. [Set object value on string.path](#)
68. [Polyfill for JSON.stringify\(\)](#)
69. [Polyfill for JSON.parse\(\)](#)
70. [HTML encoding of the string](#)
71. [CSS selector generator](#)
72. [Aggregate the input values](#)
73. [Fetch request and response Interceptors](#)
74. [Cache API call with expiry time](#)
75. [Polyfill for getElementByClassName\(\)](#)
76. [Polyfill for getElementByClassNameHierarchy\(\)](#)
77. [Find the element with the given color property](#)
78. [Throttle an array of tasks](#)
79. [Decode a string](#)
80. [Trie data structure](#)
81. [First and last occurrence of a number in the sorted array](#)
82. [Piping function - part 1](#)
83. [Piping function - part 2](#)
84. [Create analytics SDK](#)
85. [Check if given binary tree is full](#)
86. [Get height and width of a binary tree](#)

87. [Polyfill for extend](#)

88. [Animate elements in sequence](#)

© JavaScript Interview Guide | learnersbucket.com

8

89. [LocalStorage with expiry](#)

90. [Custom cookie](#)

91. [Create an immutability helper - part 1](#)

92. [Create an immutability helper - part 2](#)

93. [Make high priority API call](#)

94. [Convert JSON to HTML](#)

95. [Convert HTML to JSON](#)

96. [Concurrent history tracking system](#)

97. [Implement an in-memory search engine](#)

98. [Implement a fuzzy search function](#)

99. [Cancel an API request](#)

100. [Highlight words in the string](#)

React Questions

1.

[usePrevious\(\) hook](#)

2.

[useIdle\(\) hook](#)

3.

[useAsync\(\) hook](#)

4.

[useDebounce\(\) hook](#)

5.

[useThrottle\(\) hook](#)

6.

[useResponsive\(\) hook](#)

7.

[useWhyDidYouUpdate\(\) hook](#)

8.

[useOnScreen\(\) hook](#)

9.

[useScript\(\) hook](#)

10. [useOnClickOutside\(\) hook](#)

11. [useHasFocus\(\) hook](#)

12. [useToggle\(\) hook](#)

13. [useCopy\(\) hook](#)

14. [useLockedBody\(\) hook](#)

15. [Number Increment counter](#)
16. [Capture product visible in viewport](#)
17. [Highlight text on selection](#)
18. [Batch API calls in sequence](#)

© JavaScript Interview Guide | learnersbucket.com

9

19. [Time in human readable format](#)
20. [Detect overlapping circles](#)

System design Questions

1.

[Maker checker flow](#)

2.

[Online coding judge](#)

© JavaScript Interview Guide | learnersbucket.com

10

Concepts

© JavaScript Interview Guide | learnersbucket.com

11

Closure

Closures are one of the essential concepts of JavaScript that make the language very flexible and resilient to work with, having a good

understanding of it is really important.

In simple terms, closure is a bundling of two or more functions where inner functions have access to the properties and methods of the outer functions even after the execution of the external function is done.

function

example

```
() {
```

```
let
```

```
blog =
```

```
"learnersbucket"
```

```
;
```

```
function
```

```
displayBlog
```

```
() {
```

```
console
```

```
.log(blog);
```

```
}
```

```
displayBlog();
```

```
}
```

```
example();
```

```
// "learnersbucket"
```

If you notice the variable `blog` is declared above the function

`displayBlog()` , but it can be accessed inside it.

This is because variables are lexical scope in JavaScript and then can be accessed anywhere inside that scope unless and until they are

overridden. In the above example, the variable is function scoped and it can be accessed anywhere within the function body (even in the

inner functions).

function

example

```
() {
```

```
// outer scoped
```

```
let
```

```
blog =
```

```
"learnersbucket"
```

```
;
```

```
function
```

```
displayBlog
```

```
() {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

12

```
// new variable with the same name
```



```
// declared in a new scope
```

```
// this will be printed
```

```
let
```

```
blog =
```

```
"hello"
```

```
;
```

```
console
```

```
.log(blog);
```

```
}
```

```
displayBlog();
```

```
}
```

```
example();
```

```
// hello
```

Preference is always given to the nearest declared one.

This feature makes the closures extremely powerful as even if we

return the inner function, it will have access to the variables (it will be alive) in the outer scope and perform all sorts of operations.

```
function
```

```
sum
```

```
() {
```

```
let
```

```
a =  
10  
;  
function  
add  
(  
b  
) {  
  return  
  a + b  
}  
return  
add;  
}  
  
// sum will return a function that will accept an argument  
  
const  
fn = sum();  
  
// pass the argument and it will sum with  
  
// the value of variable 'a' that is in outer scope  
  
// and return the total
```

```
let  
total = fn(  
20  
);  
console  
.log(total);  
// 30
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

13

Not only variables but can also access all the other properties and methods of the outer function, not just parent but at any level in the scope it is declared.

```
function
```

```
x
```

```
(
```

```
a
```

```
) {
```

```
function
```

```
y
```

```
(
```

```
b
```

```
) {  
  function  
    z  
    (  
      c  
    ) {  
      return  
        a + b + c;  
    };  
    return  
      z;  
  }  
  return  
    y;  
}  
  
// the outer function accepts an argument and returns a function  
  
const  
a = x(  
  10  
);
```

```
// the inner function also accepts an argument and returns the total of both
// outer and inner argument

let
b = a(
20
);
let
c = b(
30
);
console
.log(c);
// 60
```

If you notice, in this, the inner function has access to the outer function's arguments and can use it for each instance it is created.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

14

Array.reduce() method

Array.reduce() is one of the most powerful methods available that can be used to perform different types of actions like segregation,

aggregation, running things in sequence/series, etc.

## Anatomy of Array.reduce()

// verbose

```
arr.reduce(callbackfn, initialValue);
```

// simplified

// callback function with parameters

```
arr.reduce((  
  previousValue, currentValue, currentIndex,  
  array  
) => {  
  const  
    nextValue = previousValue + currentValue;  
  return  
    nextValue;  
}, initialValue);
```

Array.reduce() accepts a callback function and initial value as an input (initial value is optional). The function will be called for each element of the array with the initial value at the beginning and then with the value returned from the last call of the same function.

The callback function has 4 parameters, ( previousValue , currentValue , currentIndex , array ).

- *previousValue* – The value returned from the last call of the same function or the initial value at the beginning.
- *currentValue* – Current value of the array.

- *currentIndex* – Current index position of the iteration.
- *array* – The array itself.

Using this method we can perform different types of operations.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

15

## Aggregation

Sum all the elements of the array or multiply all the elements of the array.

```
const
```

```
arr = [
```

```
  1
```

```
,
```

```
  2
```

```
,
```

```
  3
```

```
,
```

```
  4
```

```
];
```

```
const
```

```
sum = arr.reduce((
```

```
  previousValue, currentValue
```

```
)  
  
=> {  
  
  const  
  
  nextValue = previousValue + currentValue;  
  
  return  
  
  nextValue;  
  
  },  
  
  0  
  
);  
  
console  
  
.log(sum);  
  
// 10  
  
const  
  
product = arr.reduce((  
  
  previousValue, currentValue  
  
  )  
  
=> {  
  
  const  
  
  nextValue = previousValue * currentValue;  
  
  return
```



```
nextValue;  
  
},  
  
1  
  
);  
  
console  
  
.log(product);  
  
// 24
```

## Segregation

We can group a certain set of values depending on our requirements.

```
const  
  
arr = [  
  
  1.1  
  
  ,  
  
  1.2  
  
  ,  
  
  1.3  
  
  ,  
  
  2.2  
  
  ,  
  
  2.3
```

,

2.4

];

const

segregate = arr.reduce((

previousValue, currentValue

)

=> {

// round of the value

const

floored =

Math

.floor(currentValue);

// if the key is not present

// create a new entry with the array value

if

(!previousValue[floored]){

previousValue[floored] = [];

}

// segregate the current value in the respective

key

© JavaScript Interview Guide | learnersbucket.com

16

```
previousValue[floored].push(currentValue);
```

```
// return the updated value
```

```
return
```

```
previousValue;
```

```
}, {});
```

```
console
```

```
.log(seggregate);
```

```
/*
```

```
{
```

```
1: [1.1, 1.2, 1.3],
```

```
2: [2.2, 2.3, 2.4]
```

```
}
```

```
*/
```

Run in sequence

Let's say we have an array of functions and a value, the value has to be passed through these functions in a pipe. Like the initial value has to be passed to the first function and then the returned value from the first function has to be passed to the next function and so on.

```
// functions
```

```
const  
upperCase = (  
  str  
) => {  
  return  
    str.toUpperCase();  
};
```

```
const  
reverse = (  
  str  
) => {  
  return  
    str.split(  
      "  
    ).reverse().join(  
      "  
    );  
};
```

```
const  
append = (
```

str

) => {

return

"Hello "

+ str;

};

// array of functions to be piped

const

arr = [upperCase, reverse, append];

// initial value

const

initialValue =

"learnersbucket"

;

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

17

const

finalValue = arr.reduce((

previousValue, currentElement

)

```
=> {  
  // pass the value through each function  
  // currentElement is the function from the array  
  const  
    newValue = currentElement(previousValue);  
  // return the value received from the function  
  return  
    newValue;  
}, initialValue);  
console  
.log(finalValue);  
// "Hello TEKUBSRENRAEL"
```

Similarly, if we want to run a promise in a sequence we can do the same with this.

```
// helper function to create a promise  
// that resolves after a certain time  
const  
asyncTask =  
function  
(
```

```
time
) {
return
new
Promise
((
resolve, reject
) => {
setTimeout(
()
=> resolve(
`Completing ${time}`
),
100
*
time
);
});
}

// create an array of task
```

```
const
promises = [
  asyncTask(
    3
  ),
  asyncTask(
    1
  ),
  asyncTask(
    7
  ),
  asyncTask(
    2
  ),
  asyncTask(
    5
  ),
];

// main function to run promise in series

const
```



```
asyncSeriesExecuter =  
function  
(  
  promises  
) {  
  promises.reduce((  
    acc, curr  
  ) => {  
    // return when previous promise is resolved  
    return  
    acc.then(  
      ()  
    ) => {  
      // run the current promise  
      return  
      curr.then(  
        val  
      ) => {  
        console  
        .log(val)}});
```

18

```
});
```

```
},
```

Promise

```
.resolve());
```

```
};
```

```
asyncSeriesExecuter(promises);
```

```
"Completing 3"
```

```
"Completing 1"
```

```
"Completing 7"
```

```
"Completing 2"
```

```
"Completing 5"
```

19

Promises

Everyone has their own claims about whether JavaScript is a synchronous programming language or asynchronous, blocking, or non-blocking code, but not everyone is sure about it (even I am not ).

Let us try to get this thing clear and understand promises and how it works.

JavaScript is a synchronous programming language. However, callback functions enable us to transform it into an asynchronous programming language.

And promises are to help to get out of “callback hell” while dealing with the asynchronous code and do much more.

In simple terms, JavaScript promises are similar to the promises made in human life.

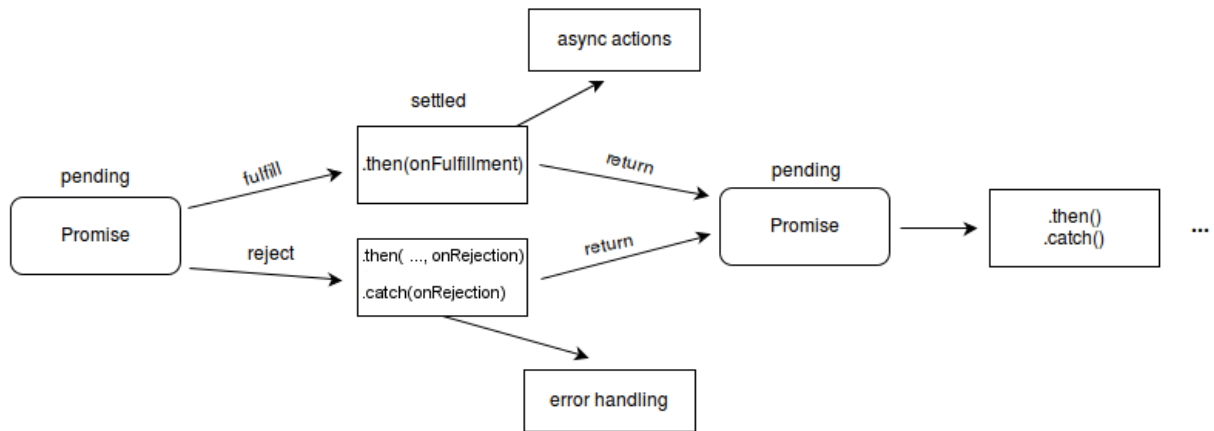
The dictionary definition of promises is –

“Assurance that one will do something or that a particular thing will happen.”

JavaScript promises also work in the same way.

- When a promise is created, there are only two outcomes to that promise.
- Either it will be fulfilled (resolved) or it will be rejected.
- By the time promises are not fulfilled or rejected, they will be in a pending state.
- Promises are fulfilled with a certain value, that value can be further processed (if the value also is a promise) or given back raw.

© JavaScript Interview Guide | learnersbucket.com



- Promises are rejected with the reason that caused them to be rejected.
- After either of the results, we can also perform the next set of operations.

*Working of promises MDN reference*

Anatomy of promise

const

promise =

new

Promise

((

resolve, reject

) => {

// resolve or reject

});

Promise has three methods available to it ( then , catch , & finally ) that can be used once it is settled ( resolved or rejected ). Each method accepts a

callback function that is invoked depending on the state of the promise.

- *then(onResolvedFn, onRejectedFn)* – This will be called either when the promise is rejected or resolved. Depending upon the state,

appropriate callback functions will be invoked with the value.

- *catch(onRejectFn)* – This will be called when the promise is rejected with the reason.

- *finally(onFinallyFn)* – This will be called every time after then and catch.

Promise

```
.prototype.then(onResolvedFn, onRejectedFn);
```

© JavaScript Interview Guide | learnersbucket.com

21

Promise

```
.prototype.catch(onRejectedFn);
```

Promise

```
.prototype.finally(onFinallyFn);
```

Working of promise

Create a promise that will resolve after 5 seconds.

```
const
```

```
promise =
```

```
new
```

Promise

```
((
  resolve, reject
) => {
  // a promise that will resolve after
  // 5 second
  setTimeout(
    ()
    => {
      resolve(
        "Hello World!"
      );
    },
    5000
  );
});
```

Initially, the promise will be in the pending state.

```
console
.log(promise);

/*
```

```
Promise { : "pending" }
```

```
: "pending"
```

```
: Promise.prototype { ... }
```

```
*/
```

After 5 seconds, the state of the promise will be updated.

```
setTimeout(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(promise);
```

```
  },
```

```
  6000
```

```
);
```

```
/*
```

```
  Promise { : "fulfilled", : "Hello World!" }
```

```
  : "fulfilled"
```

```
  : "Hello World!"
```

```
  : Promise.prototype { ... }
```

```
*/
```

We can assign the `.then( onResolvedFn , onRejectedFn )` method to the promise but the *onResolvedFn* callback function will be called only after the promise is resolved and will have the value.

22

```
promise.then((
```

```
  val
```

```
) => {
```

```
  console
```

```
    .log(val); });
```

// "Hello World!" // after the promise is resolved that is after 5 seconds  
Thenable promises can be chained further.

```
promise
```

```
  .then((
```

```
    val
```

```
) => {
```

```
  return
```

```
    "ABC "
```

```
    + val; })
```

```
  .then((
```

```
    val
```

```
) => {
```

```
  console
```

```
    .log(val); });
```



```
// "ABC Hello World!"
```

We can attach a finally block independently to the then , as well as catch , and it will be invoked at the end.

```
promise
```

```
.then((
```

```
val
```

```
) => {
```

```
return
```

```
"ABC "
```

```
+ val; })
```

```
.then((
```

```
val
```

```
) => {
```

```
console
```

```
.log(val); })
```

```
.finally(
```

```
()
```

```
=> {
```

```
console
```

```
.log(
```

```
"task done"
```

```
); });
```

```
// "ABC Hello World!"
```

```
// "task done"
```

Similarly, let's say we reject a promise after 5 seconds, then we can either use the `.then(null, onRejectedFn)` or `.catch(onRejectedFn)`.

```
const
```

```
promise =
```

```
new
```

```
Promise
```

```
((
```

```
resolve, reject
```

```
) => {
```

```
// a promise that will reject after
```

```
// 5 second
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
reject(
```

```
"Error 404"
```

```
);
```

```
},
```

```
5000

);

});

promise.then(

null

, (error) => {

console

.error(

"Called from then method"

, error);

});

// "Called from then method" "Error 404"

promise.catch((

error

) => {

console

.error(

"Called from catch method"

, error);

});
```

23

```
// "Called from catch method" "Error 404"
```

As you can notice multiple handlers can be assigned on the same promise and `.then()` will execute in the order of assignment.

The catch block can also be extended further using `.then()`.

```
promise.then(  
  null
```

```
, (error) => {
```

```
  return
```

```
    error;
```

```
  }).then((  
    val
```

```
  ) => {
```

```
    console
```

```
      .log(  
        "I am chained from then"
```

```
      , val);
```

```
    });
```

```
  // "I am chained from then" "Error 404"
```

```
promise.catch((  
  error  
) => {  
  return  
    error;  
  }).then((  
    val  
) => {  
    console  
      .log(  
        "I am chained from catch"  
      , val);  
    });  
  // "I am chained from catch" "Error 404"
```

And .finally() can be attached to both of these.

```
promise.then(  
  null  
  , (error) => {  
    return  
      error;
```

```
}).then((  
  val  
) => {  
  console  
    .log(  
      "I am chained from then"  
      , val);  
    }).finally(  
      ()  
      => {  
        console  
          .log(  
            " Then block finally done"  
          );  
        });  
    promise.catch((  
      error  
    ) => {  
      return  
        error;
```

```
}).then((  
  val  
) => {  
  console  
    .log(  
      "I am chained from catch"  
    , val);  
  }).finally(  
    ()  
    => {  
      console  
        .log(  
          " Catch block finally done"  
        );  
    });  
  "I am chained from then"  
  "Error 404"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

"I am chained from catch"

```
"Error 404"
```

```
" Then block finally done"
```

```
" Catch block finally done"
```

Notice the order of execution, the first error is handled in `.then` and then in `.catch` and then finally blocks of both are called in order.

### Helper methods

The promise object has many static methods. Some are helper's methods while others help to process the promise better.

`Promise.resolve(value)` creates a resolved promise.

#### Promise

```
.resolve(
```

```
"I am resolved"
```

```
)
```

```
.then((
```

```
val
```

```
) => {
```

```
console
```

```
.log(val); });
```

```
// "I am resolved"
```

Similarly, `Promise.reject(reason)` creates a rejected promise.

#### Promise



```
.reject(  
  "I am throwing error"  
)  
.catch((  
  error  
) => {  
  console  
    .error(error); });  
// "I am throwing error"
```

## Process methods

These methods help to process async task concurrency. We have covered each of them in the problems section.

- [Promise.all\(\)](#)
- [Promise.allSettled\(\)](#)
- [Promise.any\(\)](#)
- [Promise.race\(\)](#)

© JavaScript Interview Guide | learnersbucket.com

25

## Async...await

This is new syntax introduced in ES6 that helps to process the promise better.

```
const
promise =
Promise
.resolve(
"I am resolved"
);
async
function
example
(promise){
// promise is wrapped in a try-catch block
// to handle it better
try
{
const
resp =
await
promise;
console
.log(resp);
```

```
}  
  
catch  
  
(error){  
  
  console  
  
    .error(error);  
  
}  
  
finally  
  
{  
  
  console  
  
    .log(  
  
      "Task done"  
  
    );  
  
}  
  
}  
  
example(promise);  
  
// "I am resolved"  
  
// "Task done"
```

To use it we have to mark the function with the `async` keyword and then we can use the `await` keyword inside the `async` function.

The code is wrapped inside a `try-catch-finally` block for frictionless execution.

async keyword with different function declaration.

```
// fat arrow
```

```
const
```

```
example =
```

```
async
```

```
() => {
```

```
// await can be used
```

```
};
```

```
// assigning the function variable
```

```
const
```

```
example =
```

```
async
```

```
function
```

```
() {
```

```
// await can be used
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

26

```
};
```

A function declared with the async keyword returns a promise.

```
const
```

```
promise =  
  Promise  
    .resolve(  
      "I am resolved"  
    );  
  // fat arrow  
  const  
  example =  
    async  
    (promise) => {  
      // promise is wrapped in a try-catch block  
      // to handle it better  
      try  
      {  
        const  
        resp =  
          await  
            promise;  
        return  
          resp;
```

```
}  
  
catch  
  
(error){  
  
  console  
  
    .error(error);  
  
}  
  
finally  
  
{  
  
  console  
  
    .log(  
  
      "Task done"  
  
    );  
  
}  
  
};  
  
console  
  
  .log(example(promise));  
  
// Promise { : "fulfilled", : "I am resolved" }  
  
// "Task done"  
  
example(promise).then((  
  
  val
```

```
) => {  
  
  console  
  
  .log(val);  
  
  });  
  
  //"Task done"  
  
  //"I am resolved"
```

Notice the order of execution here, the try and finally block will be executed, thus content in the finally block is printed and the value returned is accessed in the .then that is why "Task done" is printed before "I am resolved" .

Read more about [promises on MDN](#) .

© JavaScript Interview Guide | learnersbucket.com

27

## Function & this

Functions are the building blocks of JavaScript, it is one of the programming languages that uses functional programming at the core.

As easy as it is to use the functions, understanding this keyword is that complex. Because the value of this is decided at the execution time, unlike other programming languages.

Majorly there are four different ways to invoke a function in Javascript.

1. As a normal function.

function

example

```
(){
```

```
  console
```

```
    .log(
```

```
      "Hello World!"
```

```
    );
```

```
  };
```

```
example();
```

```
// "Hello World!"
```

2. As a method.

```
const
```

```
obj = {
```

```
  blog:
```

```
    "learnersbucket"
```

```
,
```

```
  displayBlog:
```

```
    function
```

```
    (){
```

```
      console
```

```
        .log(
```



```
this  
    .blog);  
}  
};  
obj.displayBlog();  
// "learnersbucket"
```

3. As a constructor.

```
const  
number =  
new  
Number  
(  
"10"  
);  
console  
.log(number);  
// 10
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

28

4. Indirectly using call , apply , & bind .

```
function
example
(
  arg1, arg2
){
  console
  .log(arg1 + arg2);
};
example.call(
  undefined
,
  10
,
  20
);
// 30
```

The value of this is decided upon how the function is invoked, each invocation creates its own context and the context decides the value of this. Also the “[strict mode](#)” affects the behavior of this too.

Value of this when invoked as a normal function

The value of this in the function invocation refers to the global object.

[window](#) in the browser and [global](#) in Nodejs.

function

example

() {

// in browser this refers to window

console

.log(

this

===

window

);

}

example();

// true

Because this refers to the window object, if we assign any property to it we can access it outside.

function

example

() {

// in strict mode this refers to undefined

this

```
.blog =  
"learnersbucket"  
;  
this  
.displayBlog =  
function  
{  
  console  
.log(  
  `Awesome ${  
this  
.blog}`  
)  
}  
}  
example();  
console  
.log(  
this  
.blog);
```

```
// "learnersbucket"
```

```
this
```

```
.displayBlog();
```

© JavaScript Interview Guide | learnersbucket.com

29

```
// "Awesome learnersbucket"
```

Strict mode

If you invoke the function with the strict mode the value of this will be undefined .

function

example

```
() {
```

```
"use strict"
```

```
// in strict mode this refers to undefined
```

```
console
```

```
.log(
```

```
this
```

===

```
undefined
```

```
);
```

```
}
```

```
example();
```

```
// true
```

It also affects all the inner functions that are defined in the function which is declared in strict mode.

```
function
```

```
example
```

```
() {
```

```
"use strict"
```

```
// in strict mode this refers to undefined
```

```
console
```

```
.log(
```

```
this
```

```
===
```

```
undefined
```

```
);
```

```
// inner function
```

```
function
```

```
inner
```

```
() {
```

```
// in strict mode this refers to undefined
```

```
console
```

```

.log(
this

=====

undefined

);

}

// invoke the inner function

inner();

}

example();

// true

// true

```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

30

## IIFE (Immediately Invoked Function Expression)

When we immediately invoke the function, it is invoked as a normal function thus depending upon the mode, the value of this inside it is decided.

```

// normal mode

(

function

```

example

```
(){
```

```
// in strict mode this refers to undefined
```

```
console
```

```
.log(
```

```
this
```

===

```
window
```

```
);
```

```
})();
```

```
// true
```

```
// strict mode
```

```
(
```

```
function
```

example

```
(){
```

```
"use strict"
```

```
// in strict mode this refers to undefined
```

```
console
```

```
.log(
```



this

===

undefined

);

})();

// true

Value of this when invoked as a method

When a function is declared inside an object the value of this inside that function will refer to the object it is declared in.

const

example = {

blog:

'learnersbucket'

,

displayBlog:

function

() {

// this refers to the current object

console

.log(

this

```
=== example);  
  
console  
  
.log(  
  
this  
  
.blog);  
  
}  
  
};  
  
example.displayBlog();  
  
// true  
  
//"learnersbucket"
```

The context is set at the time of invocation, thus if we update the value of the object property value, it will be reflected.

```
const  
  
example = {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

31

```
blog:  
  
'learnersbucket'  
  
,  
  
displayBlog:  
  
function
```

```
(){  
  // this refers to the current object  
  
  console  
  
  .log(  
  
  this  
  
  === example);  
  
  console  
  
  .log(  
  
  this  
  
  .blog);  
  
  }  
  
  };  
  
  example.blog =  
  
  "MDN"  
  
  ;  
  
  example.displayBlog();  
  
  // true  
  
  // "MDN"
```

If the object is passed as a reference, then the context is shared between both the variables, the original and the one that has the

reference.

const

example = {

blog:

'learnersbucket'

,

displayBlog:

function

() {

// this refers to the current object

console

.log(

this

=== example);

console

.log(

this

=== example2);

console

.log(

```
this
.blog);
}
};

const
example2 = example;
example2.blog =
"MDN"
;

example.displayBlog();

// true

// true

// "MDN"

example2.displayBlog();

// true

// true

// "MDN"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

32

But, if we extract the method and save it in a variable, and then invoke the variable, the outcome will change.

```
const
example = {
  blog:
    'learnersbucket'
  ,
  displayBlog:
    function
    () {
      console
      .log(
        this
        === example);
      console
      .log(
        this
        .blog);
    }
  };
const
display = example.displayBlog;
```

```
display();
```

```
// false
```

```
// undefined
```

This is because when extracted to a variable and invoked it will be treated as a normal function.

```
const
```

```
example = {
```

```
  blog:
```

```
    'learnersbucket'
```

```
,
```

```
  displayBlog:
```

```
    function
```

```
    (){
```

```
      // this refers to the window object
```

```
      console
```

```
      .log(
```

```
      this
```

```
      ===
```

```
      window
```

```
    );
```

```
console
.log(
this
.blog);
}
};

const
display = example.displayBlog;
display();

// true

// undefined
```

The same happens when you pass the methods to the timers i.e

[setTimeout](#) and [setInterval](#) . Timers invoke the function as a normal function or throw errors in strict mode.

```
const
example = {
  blog:
'learnersbucket'
,
  displayBlog:
function
```



```
(){
```

```
// this refers to the window object
```

```
console
```

```
.log(
```

```
this
```

```
===
```

```
window
```

```
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
33
```

```
console
```

```
.log(
```

```
this
```

```
.blog);
```

```
}
```

```
};
```

```
setTimeout(example.displayBlog,
```

```
200
```

```
);
```

```
// true
```

```
// undefined
```

If there are any inner functions inside the methods, the value of this inside them depends upon how the inner function is invoked.

```
const
```

```
example = {
```

```
  blog:
```

```
    'learnersbucket'
```

```
,
```

```
  displayBlog:
```

```
    function
```

```
    () {
```

```
      function
```

```
      inner
```

```
      () {
```

```
        // this refers to the window object
```

```
        console
```

```
        .log(
```

```
        this
```

```
        ===
```

```
        window
```

```
      );
```

```
console
.log(
this
.blog);
};
inner();
}
};
example.displayBlog();

// true

// undefined
```

Because the inner function is invoked as a normal function the value of this is a window object.

To access the value of the parent we can use either the [Fat arrow function](#) or indirect invocation technique using call & apply .

### Fat arrow function

The fat arrow function does not have this of its own, it accesses this in its nearest scope.

© JavaScript Interview Guide | learnersbucket.com

34

In the below example, the fat arrow's this refers to the this of displayBlog() which refers to the object it is part of.

```
const
example = {
  blog:
    'learnersbucket'
  ,
  displayBlog:
    function
    () {
      const
      inner =
      ()
      => {
        // this refers to the example object
        console
        .log(
        this
        === example);
        console
        .log(
        this
```

```
.blog);  
  
};  
  
inner();  
  
}  
  
};  
  
example.displayBlog();  
  
// true  
  
// "learnersbucket"
```

Using call() method

We can change the value of this inside a function by calling it indirectly with the call method.

```
const  
  
example = {  
  
  blog:  
  
    'learnersbucket'  
  
  ,  
  
  displayBlog:  
  
    function  
  
    () {  
  
      function
```

```
inner
(){
// this refers to the example object
console
.log(
this
=== example);
console
.log(
this
.blog);
};
inner.call(
this
);
}
};
example.displayBlog();
// true
// "learnersbucket"
```

35

Value of this when invoked as a constructor

The value of this in the function invoked as a constructor refers to a new object which has the value passed as an argument. Each instance creates a new object.

function

Example

```
(  
  blog  
) {  
  this  
    .blog = blog;  
  this  
    .displayBlog =  
  function  
    () {  
    console  
      .log(  
        this  
          .blog);  
    }  
  }  
}
```

```
};
```

```
};
```

```
const
```

```
example =
```

```
new
```

```
Example(
```

```
"learnersbucket"
```

```
);
```

```
example.displayBlog();
```

```
///  
"learnersbucket"
```

```
const
```

```
example2 =
```

```
new
```

```
Example(
```

```
"MDN"
```

```
);
```

```
example2.displayBlog();
```

```
///  
"MDN"
```

```
console
```

```
.log(example === example2);
```



```
//false
```

Note – There are some methods in JavaScript that when invoked normally behave as a constructor.

```
const
```

```
reg1 =
```

```
RegExp
```

```
(
```

```
'\\w+'
```

```
);
```

```
const
```

```
reg2 =
```

```
RegExp
```

```
(
```

```
'\\w+'
```

```
);
```

```
console
```

```
.log(reg1 === reg2);
```

```
// false
```

To avoid this we can add a check to the function which we want to be invoked as a constructor only.

```
function
```

Example

```
(  
  blog  
) {  
  if  
    (!(  
      this  
        instanceof  
          Example)) {  
            throw  
              Error  
                (  
                  'Can be invoked only as a constructor'  
                );  
          }  
    this  
      .blog = blog;  
  };  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
const
example =
new
Example(
"learnersbucket"
);
console
.log(example.blog);
const
example2 = Example(
"MDN"
);
```

// Error: Can be invoked only as a constructor

Value of this when invoked indirectly

When the function is invoked indirectly the value of this is what is passed as an argument to the call , apply , & bind method.

Run time binding

Using the call and apply methods we can invoke the function with the new context. The values will be attached to that execution only.

```
const
exampleObj = {
```

```
blog:
'learnersbucket'

,

};

function
example
(
name
) {
console
.log(
`${name} runs ${
this
.blog}`
);
};

example.call(exampleObj,
'Prashant'
);

// "Prashant runs learnersbucket"
```

```
example.apply(exampleObj, [  
  'Prashant'  
]);  
  
// "Prashant runs learnersbucket"
```

The difference between call and apply is that apply accepts arguments in an array, while call accepts it normally.

### Permanent binding

When using bind, we can create a new function with the new values and store it in a variable, and then use it further. It creates fresh permanent binding without affecting the original function.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

37

```
const  
  
exampleObj = {  
  
  name:  
  
  'prashant'  
  
  ,  
  
};  
  
function  
  
example  
  
(
```

```
blog
) {
  console
  .log(
    `${
      this
      .name} runs ${blog}`
    );
};

const
bounded = example.bind(exampleObj);

bounded(
  'learnersbucket'
);

// "Prashant runs learnersbucket"

bounded(
  'MDN'
);

// "Prashant runs MDN"
```

38

JavaScript Questions

© JavaScript Interview Guide | learnersbucket.com

39

Promise.all() polyfill

Definition

According to MDN –

The Promise.all() accepts an array of promises and returns a promise that resolves when all of the promises in the array are fulfilled or when the iterable contains no promises. It rejects with the reason of the first promise that rejects.

After reading the definition of Promise.all() we can break down the problem in sub-problem and tackle it one by one.

- It will return a promise.
- The promise will resolve with the result of all the passed promises or reject with the error message of the first failed promise.
- The results are returned in the same order as the promises are in the given array.

Implementation

const

myPromiseAll =

function

```
(  
  taskList  
) {  
  //to store results  
  
  const  
  results = [];  
  
  //to track how many promises have completed  
  
  let  
  promisesCompleted =  
  
  0  
  
  ;  
  
  // return new promise  
  
  return  
  
  new  
  
  Promise  
  
  ((  
    resolve, reject  
  ) => {  
    taskList.forEach((  
      promise, index
```



```
) => {  
  //if promise passes  
  promise.then((  
    val  
  ) => {  
    //store its outcome and increment the count  
    results[index] = val;  
    promisesCompleted +=  
    1  
  }  
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

40

```
//if all the promises are completed,  
//resolve and return the result  
if  
(promisesCompleted === taskList.length)  
{  
  resolve(results)  
}  
})
```

```
//if any promise fails, reject.
```

```
.catch(
```

```
error
```

```
=> {
```

```
reject(error)
```

```
})
```

```
})
```

```
});
```

```
}
```

Test case 1

Input:

```
function
```

```
task
```

```
(
```

```
time
```

```
) {
```

```
return
```

```
new
```

```
Promise
```

```
(
```

```
function
(
  resolve, reject
) {
  setTimeout(
    function
    () {
      resolve(time);
    }, time);
  });
}

const
taskList = [task(
  1000
), task(
  5000
), task(
  3000
)];

//run promise.all
```

```
myPromiseAll(taskList)
```

```
.then(
```

```
results
```

```
=> {
```

```
console
```

```
.log(
```

```
"got results"
```

```
, results)
```

```
})
```

```
.catch(
```

```
console
```

```
.error);
```

Output:

```
//"got results" [1000,5000,3000]
```

Test case 2

Input:

```
function
```

```
task
```

```
(
```

```
time
```

```
) {  
  return  
  new  
  Promise  
  (  
    function  
    (  
      resolve, reject  
    ) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

41

```
setTimeout(  
  function  
  () {  
    if  
    (time <  
      3000  
    ){  
      reject(  
        "Rejected"
```

```
);  
}  
else  
{  
  resolve(time);  
}  
}, time);  
});  
}  
  
const  
taskList = [task(  
  1000  
, task(  
  5000  
, task(  
  3000  
  )]);  
  
//run promise.all  
myPromiseAll(taskList)  
  .then(  
    
```

```
results  
  
=> {  
  
  console  
  
    .log(  
  
      "got results"  
  
      , results)  
  
    })  
  
  .catch(  
  
    console  
  
    .error);
```

Output:

"Rejected"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

42

## Promise.any() Polyfill

### Definition

According to MDN –

Promise.any() takes an iterable of Promise objects. It returns a single promise that fulfills as soon as any of the promises in the iterable fulfills, with the value of the fulfilled promise. If no promises in the iterable fulfill (if all of the given promises are rejected), then the returned promise is rejected with an AggregateError, a new subclass of Error that groups together individual errors.

In simple terms Promise.any() is just opposite of Promise.all() .

Reading the definition we can break the problem statement into multiple sub-problems and then tackle them individually to implement the polyfill.

- Function takes an array of promises as input and returns a new promise.
- The returned promise is resolved as soon as any of the input promises resolves.
- Else if all of the input promises are rejected then the returned promise is rejected with the array of all the input promises reasons.

## Implementation

```
const
```

```
any =
```

```
function
```

```
(
```

```
promisesArray
```

```
) {
```

```
const
```

```
promiseErrors =
```

```
new
```



Array

(promisesArray.length);

let

counter =

0

;

//return a new promise

return

new

Promise

((

resolve, reject

) => {

promisesArray.forEach((

promise

) => {

Promise

.resolve(promise)

.then(resolve)

// resolve, when any of the input

promise resolves

© JavaScript Interview Guide | learnersbucket.com

43

```
.catch((  
    error  
  ) => {  
    promiseErrors[counter] = error;  
    counter = counter +  
    1  
    ;  
    if  
    (counter === promisesArray.length) {  
      // all promises rejected, reject outer  
      promise  
      reject(promiseErrors);  
    }  
  });  
});  
});  
};
```

Test case 1

Input:

```
const
```

```
test1 =
```

```
new
```

```
Promise
```

```
(
```

```
function
```

```
(
```

```
resolve, reject
```

```
)
```

```
{
```

```
setTimeout(reject,
```

```
500
```

```
,
```

```
'one'
```

```
);
```

```
});
```

```
const
```

```
test2 =
```

new

Promise

(

function

(

resolve, reject

)

{

setTimeout(resolve,

600

,

'two'

);

});

const

test3 =

new

Promise

(

function

```
(
  resolve, reject
)
{
  setTimeout(reject,
    200
    ,
    'three'
  );
});
any([test1, test2, test3]).then(
  function
  (
    value
  ) {
    // first and third fails, 2nd resolves
    console
    .log(value);
  }).catch(
    function
```

```
(  
  err  
) {  
  console  
    .log(err);  
});
```

Output:

"two"

Test case 2

Input:

const

test1 =

new

Promise

```
(  
  function  
    (  
      resolve, reject  
    )  
    {
```

44

```
setTimeout(reject,
```

```
500
```

```
,
```

```
'one'
```

```
);
```

```
});
```

```
const
```

```
test2 =
```

```
new
```

```
Promise
```

```
(
```

```
function
```

```
(
```

```
resolve, reject
```

```
)
```

```
{
```

```
setTimeout(reject,
```

```
600
```

```
,  
'two'  
);  
});  
  
const  
test3 =  
  
new  
Promise  
(  
  function  
  (  
    resolve, reject  
  )  
  {  
    setTimeout(reject,  
      200  
    ,  
    'three'  
  );  
});
```



```
any([test1, test2, test3]).then(
```

```
function
```

```
(
```

```
value
```

```
) {
```

```
console
```

```
.log(value);
```

```
}).catch(
```

```
function
```

```
(
```

```
err
```

```
){
```

```
// all three fails
```

```
console
```

```
.log(err);
```

```
});
```

Output:

```
[
```

```
"three"
```

```
,
```

```
"one"
```

```
,
```

```
"two"
```

```
]
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

45

## Promise.race() polyfill

### Definition

According to MDN –

The `Promise.race()` method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

Reading the definition, we can break the problem statement into sub-problems to implement the `Promise.race()` method.

- It returns a promise.
- The returned promise fulfills or rejects as soon as any one of the input promises fulfills or rejects.
- Returned promise resolves with the value of the input promise or rejects with the reason of the input promise.

Thus we can create a function that will take an array of promises as input and return a new promise, inside this returned promise

iterate the input promises and resolve or reject as soon as any of them resolves or rejects.

## Implementation

```
const
```

```
race =
```

```
function
```

```
(
```

```
promisesArray
```

```
) {
```

```
  return
```

```
    new
```

```
    Promise
```

```
    ((
```

```
      resolve, reject
```

```
    ) => {
```

```
      promisesArray.forEach((
```

```
        promise
```

```
      ) => {
```

```
        Promise
```

```
        .resolve(promise)
```

```
// resolve, when any of the input promise resolves
.then(resolve, reject)

// reject, when any of the input promise rejects
.catch(reject);

});

});

};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

46

Test case 1

Input:

const

test1 =

new

Promise

(

function

(

resolve, reject

)

```
{
  setTimeout(resolve,
    500
  ,
    'one'
  );
});

const
test2 =
  new
  Promise
  (
    function
    (
      resolve, reject
    )
    {
      setTimeout(resolve,
        100
      ,
```

```
'two'  
  
);  
  
});  
  
const  
  
test3 =  
  
new  
  
Promise  
  
(  
  
function  
  
(  
  
resolve, reject  
  
)  
  
{  
  
  setTimeout(reject,  
  
    200  
  
    ,  
  
    'three'  
  
  );  
  
});  
  
race([test1, test2, test3]).then(
```

```
function
(
value
)
{
// first two resolve, 3rd fails, but promise2 is
faster
console
.log(value);
}).catch(
function
(
err
){
console
.log(err);
});
```

Output:

"two"

Test case 2

Input:

```
const
```

```
test1 =
```

```
new
```

```
Promise
```

```
(
```

```
function
```

```
(
```

```
resolve, reject
```

```
)
```

```
{
```

```
setTimeout(resolve,
```

```
500
```

```
,
```

```
'one'
```

```
);
```

```
});
```

```
const
```

```
test2 =
```

```
new
```



Promise

```
(  
  function  
  (  
    resolve, reject  
  )  
  {  
    setTimeout(resolve,  
      100  
    ,  
    'two'  
  );  
});
```

const

test3 =

new

Promise

```
(  
  function  
  (  

```

```
resolve, reject
)
{
setTimeout(reject,
40
,
'three'
);
});
race([test1, test2, test3]).then(
function
(
value
)
{
// first two resolve, 3rd fails, but promise3 is
faster
console
.log(value);
```

47

```
}).catch(  
function  
(  
err  
) {  
  console  
.log(err);  
});
```

Output:

"three"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

48

### Promise.finally() polyfill

#### Definition

According to MDN –

The `finally()` method of a Promise schedules a function, the callback function, to be called when the promise is settled. Like `then()` and `catch()`, it immediately returns an equivalent Promise object, allowing you to chain calls to another promise method, an operation called composition.

Same as the `try ... catch ... finally` block where no matter whether code runs in a `try` block or `catch` block, the `finally` block will always be executed at the end, which can be used for cleanup operations.

The same way for Promises we have .then() for when promise resolves and .catch() for when promise rejects and .finally() block which will always run after any of those.

Example

Input:

```
function
```

```
  checkMail
```

```
() {
```

```
  return
```

```
    new
```

```
    Promise
```

```
    ((
```

```
      resolve, reject
```

```
    ) => {
```

```
      if
```

```
      (
```

```
        Math
```

```
        .random() >
```

```
        0.5
```

```
      ) {
```

```
        resolve(
```

```
'Mail has arrived'
```

```
);
```

```
}
```

```
else
```

```
{
```

```
reject(
```

```
new
```

```
Error
```

```
(
```

```
'Failed to arrive'
```

```
));
```

```
}
```

```
});
```

```
}
```

```
checkMail()
```

```
.then((
```

```
mail
```

```
) => {
```

```
console
```

```
.log(mail);
```

```
})  
.catch((  
err  
) => {  
console  
.error(err);  
})
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

49

```
.finally(  
()  
=> {  
console  
.log(  
'Experiment completed'  
);  
});
```

Output:

Error

: Failed to arrive

"Experiment completed"

From the definition, we can conclude that to implement `.finally()`

- We have to take a callback function as an input and call this callback function when the promise is settled which is either after resolve or reject.
- Since there is no reliable way to tell if the promise was accepted or refused, a finally callback will not receive any argument.
- It will provide you with a promise that you can use to compose calls to other promise methods in a chain.

Implementation

Promise

`.prototype.finally =`

`function`

`(`

`callback`

`) {`

`if`

`(`

`typeof`

```
callback !==  
'function'  
) {  
  return  
  this  
  .then(callback, callback);  
}  
  
// get the current promise or a new one  
  
const  
  
P =  
  
this  
  .constructor ||  
  
Promise  
  
;  
  
// return the promise and call the callback function  
  
// as soon as the promise is rejected or resolved  
with its value  
  
return  
  
this  
  .then(
```



value

```
=> P.resolve(callback()).then(
```

```
()
```

```
=> value),
```

```
err => P.resolve(callback()).then(
```

```
()
```

```
=> {
```

```
throw
```

```
err; })
```

```
);
```

```
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

50

Test Case

Input:

// This test case is from stack overflow.

```
const
```

```
logger = (
```

```
label, start = Date.now(
```

```
)) => (...values)
```

```
=> {  
  console  
    .log(label, ...values,  
    after ${  
      Date  
        .now()  
      - start}ms`  
    );  
  };  
  const  
    delay = (  
      value, ms  
    ) =>  
      new  
        Promise  
          (  
            resolve  
              =>  
                {  
                  setTimeout(resolve, ms, value);
```

```
});  
  
function  
test  
  
(  
  impl  
) {  
  const  
  log =  
  ordinal  
  => state => logger(  
    `${ordinal}  
    ${impl} ${state}`  
  );  
  const  
  first = log(  
    'first'  
  );  
  // test propagation of resolved value  
  delay(  
    2
```

```
,  
1000  
)  
  
.finally(first(  
  'settled'  
))  
  
.then(first(  
  'fulfilled'  
), first(  
  'rejected'  
));  
  
const  
second = log(  
  'second'  
);  
  
// test propagation of rejected value  
delay(  
  Promise  
    .reject(  
      3
```

```
),  
2000  
)  
.finally(second(  
'settled'  
))  
.then(second(  
'fulfilled'  
) , second(  
'rejected'  
));  
  
const  
third = log(  
'third'  
);  
  
// test adoption of resolved promise  
delay(  
4  
,  
3000
```

```
)  
  .finally(third(  
    'settled'  
  ))  
  .finally(  
    ()  
    => delay(  
      6  
      ,  
      500  
    ))  
  .then(third(  
    'fulfilled'  
  ), third(  
    'rejected'  
  ));  
const  
fourth = log(  
  'fourth'  
);
```

```
// test adoption of rejected promise
```

```
delay(
```

```
5
```

```
,
```

```
4000
```

```
)
```

```
.finally(fourth(
```

```
'settled'
```

```
))
```

```
.finally(
```

```
()
```

```
=> delay(
```

```
Promise
```

```
.reject(
```

```
7
```

```
),
```

```
500
```

```
))
```

```
.then(fourth(
```

```
'fulfilled'
```

```
), fourth(
```

```
'rejected'
```

```
));
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

51

```
}
```

```
test(
```

```
'polyfill'
```

```
);
```

Output:

```
"first polyfill settled"
```

```
"after 1005ms"
```

```
"first polyfill fulfilled"
```

2

```
"after 1007ms"
```

```
"second polyfill settled"
```

```
"after 2006ms"
```

```
"second polyfill rejected"
```

3

```
"after 2008ms"
```



"third polyfill settled"

"after 3006ms"

"third polyfill fulfilled"

4

"after 3512ms"

"fourth polyfill settled"

"after 4000ms"

"fourth polyfill rejected"

7

"after 4506ms"

Edge Case

//This will be resolved with undefined

Promise

.resolve(

2

).then(

()

=> {}, () => {}).then((

val

)

=>

{

console

.log(val)});

// undefined

//This will be resolved with 2

Promise

.resolve(

2

).finally(

()

=> {}).then((

val

) =>

{

console

.log(val)});

// 2

//Similarly, This will be fulfilled with undefined

Promise

```
.reject(  
  3  
)  
.then(  
  ()  
  => {}, () => {})  
)  
.then((  
  val  
)  
=>  
  {  
    console  
      .log(val)});  
// undefined  
//This will be fulfilled with 3  
Promise  
  .reject(  
    3  
  )  
  .finally(  
    ()  
    => {})  
  ).then((  
    val
```

```
) =>
{
  console
    .log(val)});
// 3

//A throw (or returning a rejected promise) in the finally callback will reject
//the new promise with the rejection reason specified when calling throw()
Promise
    .reject(
      2
    ).finally(
      ()
    ) => {
  throw
    'Parameter
      is not a number!'
    }.then((
      val
    ) => {
  console
    .log(val)});
```

```
// 'Parameter is not a number!'
```

© JavaScript Interview Guide | learnersbucket.com

52

## Promise.allSettled() polyfill

### Definition

According to MDN –

The `Promise.allSettled()` method returns a promise that fulfills after all of the given promises have either fulfilled or rejected, with an array of objects that each describes the outcome of each promise.

`Promise.allSettled()` takes an array of promises as input and returns an array with the result of all the promises whether they are rejected or resolved.

Reading the problem statement we can break it down into sub-problems and tackle them individually.

- Map the array of promises to return an object with status and value/error depending upon the promised settlement.
- Pass this map to the `Promise.all` to run them at once and return the result.

### Implementation

```
const
```

```
allSettled = (
```

```
promises

) => {

// map the promises to return a custom response.

const

mappedPromises = promises.map((

p)

=>

Promise

.resolve(p)

.then(

val

=> ({ status:

'fulfilled'

, value: val }),

err => ({ status:

'rejected'

, reason: err })

)

);

// run all the promises once with .all
```

```
return
```

```
Promise
```

```
.all(mappedPromises);
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

53

Test Case

Input:

```
const
```

```
a =
```

```
new
```

```
Promise
```

```
((
```

```
resolve
```

```
) => setTimeout(
```

```
()
```

```
=>
```

```
{ resolve(
```

```
3
```

```
) },
```

200

));

const

b =

new

Promise

((

resolve,reject

) => reject(

9

));

const

c =

new

Promise

((

resolve

) => resolve(

5

));



```
allSettled([a, b, c]).then((
```

```
val
```

```
)=> {
```

```
console
```

```
.log(val)});
```

Output:

```
[
```

```
{
```

```
"status"
```

```
:
```

```
"fulfilled"
```

```
,
```

```
"value"
```

```
:
```

```
3
```

```
},
```

```
{
```

```
"status"
```

```
:
```

```
"rejected"
```

```
,  
  "reason"  
:  
  9  
},  
{  
  "status"  
:  
  "fulfilled"  
,  
  "value"  
:  
  5  
}  
]
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

54

Custom promise

Problem Statement -

Write a function in JavaScript that works similar to the [original](#)

[promise](#) .

Promises in JavaScript allow you to execute non-blocking (asynchronous) code and produce a value if the operation is successful or throws an error when the process fails.

In short, the eventual success (or failure) of an asynchronous operation and its associated value are represented by the Promise object.

Anatomy of Promise

```
const
```

```
promise =
```

```
new
```

```
Promise
```

```
((
```

```
resolve, reject
```

```
) => {
```

```
// time-consuming async operation
```

```
// initial state will be pending
```

```
// any one of the below operations can occur at
```

```
any given time
```

```
// this will resolve or fulfill the promise
```

```
resolve(value);
```

```
// this will reject the promise

reject(reason);

});

// this will be invoked when a promise is resolved

promise.then((

value

) => {

});

// this will be invoked when a promise is rejected

promise.catch((

value

) => {

});
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

55

```
// this will always be invoked after any of the above operation

promise.finally((

value

) => {

});
```

## Working of Promise

```
const
```

```
promise =
```

```
new
```

```
Promise
```

```
((
```

```
resolve, reject
```

```
) => {
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
resolve(
```

```
"hello"
```

```
);
```

```
},
```

```
1000
```

```
);
```

```
});
```

```
promise.then((
```

```
value
```

```
) => {  
  
  console  
  
  .log(value);  
  
  });
```

We have to implement a custom function MyPromise that will be similar to the original promise.

To implement this we will use the [observer pattern](#).

Use two handlers onSuccess and onError and assign this them to the .then , .catch , .finally methods.

Whenever the resolve or reject methods are invoked, run all the handlers in sequence and pass down the values to the next.

```
// enum of states
```

```
const
```

```
states = {
```

```
PENDING:
```

```
0
```

```
,
```

```
FULFILLED:
```

```
1
```

```
,
```

```
REJECTED:
```

2

}

class

MyPromise

{

// initialize the promise

constructor

(callback) {

this

.state = states.PENDING;

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

56

this

.value =

undefined

;

this

.handlers = [];

try

{

```
callback(  
  this  
    ._resolve,  
  this  
    ._reject);  
}  
  
catch  
(error) {  
  this  
    ._reject(error);  
}  
}  
  
// helper function for resolve  
_resolve = (  
  value  
) => {  
  this  
    ._handleUpdate(states.FULFILLED, value);  
}  
  
// helper function for reject
```



```
_reject = (  
  value  
) => {  
  this  
    ._handleUpdate(states.REJECTED, value);  
}  
  
// handle the state change  
_handleUpdate = (  
  state, value  
) => {  
  if  
    (state === states.PENDING) {  
    return  
    ;  
  }  
  setTimeout(  
    ()  
    => {  
    if  
      (value
```

```
instanceof
MyPromise) {
value.then(
this
._resolve,
this
._reject);
}
this
.state = state;
this
.value = value;
this
._executeHandlers();
},
0
)
}

// execute all the handlers
// depending on the current state
```

```
_executeHandlers =  
  
()  
  
=> {  
  
  if  
  
  (  
  
    this  
  
    .state === states.PENDING) {  
  
    © JavaScript Interview Guide | learnersbucket.com  
  
    57  
  
    return  
  
    ;  
  
    }  
  
    this  
  
    .handlers.forEach((  
  
    handler  
  
    ) => {  
  
    if  
  
    (  
  
    this  
  
    .state === states.FULFILLED) {
```

```
return  
  handler.onSuccess(  
    this  
      .value);  
  }  
  return  
    handler.onFailure(  
      this  
        .value);  
    })  
  this  
    .handlers = [];  
  }  
  // add handlers  
  // execute all if any new handler is added  
  _addHandler = (  
    handler  
  ) => {  
    this  
      .handlers.push(handler);
```

```
this

._executeHandlers();

}

// then handler

// creates a new promise

// assigned the handler

then = (

onSuccess, onFailure

) => {

// invoke the constructor

// and new handler

return

new

MyPromise((

resolve, reject

) =>

{

this

._addHandler({

onSuccess: (
```

```
value
) => {
  if
    (!onSuccess) {
      return
      resolve(value);
    }
  try
    {
      return
      resolve(onSuccess(value));
    }
  catch
    (error) {
      reject(error);
    }
  },
  onFailure: (
    value
  ) => {
```

```
if  
  
(!onFailure) {  
  
return  
  
reject(value);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

58

```
}  
  
try  
  
{  
  
return  
  
reject(onFailure(value));  
  
}  
  
catch  
  
(error) {  
  
return  
  
reject(error);  
  
}  
  
}  
  
})  
  
})
```

```
};  
  
// add catch handler  
  
catch  
  
= (  
  
onFailure  
  
) => {  
  
return  
  
this  
  
.then(  
  
null  
  
, onFailure);  
  
};  
  
// add the finally handler  
  
finally  
  
= (  
  
callback  
  
) => {  
  
// create a new constructor  
  
// listen the then and catch method  
  
// finally perform the action
```



```
return  
  
new  
MyPromise((  
  resolve, reject  
) =>  
  {  
    let  
      wasResolved;  
    let  
      value;  
    this  
      .then((  
        val  
      ) => {  
        value = val;  
        wasResolved =  
          true  
      }  
      ;  
    return  
      callback();
```

```
}).catch((  
  err  
) => {  
  value = err;  
  wasResolved =  
    false  
  ;  
  return  
    callback();  
  })  
  if  
    (wasResolved) {  
    resolve(value);  
  }  
  else  
    {  
    reject(value);  
  }  
  })
```

59

```
};
```

```
};
```

Test Case

Input:

```
const
```

```
promise =
```

```
new
```

```
MyPromise((
```

```
resolve, reject
```

```
) =>
```

```
{
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
resolve(
```

```
"hello"
```

```
);
```

```
},
```

```
1000
```

```
);  
});  
promise.then((  
value  
) => {  
console  
.log(value);  
});
```

Output:

"hello"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

60

## Execute async functions in Series

### Problem Statement -

Implement a function that takes a list of async functions as input and executes them in a series that is one at a time. The next task is

executed only when the previous task is completed.

### Example

Input:

[

asyncTask(

```
3
),
asyncTask(
1
),
asyncTask(
2
)
]
```

Output:

```
3
1
2
```

We will see three different approaches to solve this problem.

Approach 1 - Using async/await

async/await is the new keyword introduced in [ES6](#) that helps us to handle the promises by avoiding the callback chaining.

[For...of](#) loop allows using await keyword performing the next iteration only when the previous one is finished.

To handle the rejection, we wrap the async/await in the try-catch block.

61

Implementation

const

asyncSeriesExecuter =

async

function

(

promises

) {

for

(

let

promise

of

promises) {

try

{

const

result =

await

```
promise;  
  
console  
.log(result);  
  
}  
  
catch  
  
(e){  
  
console  
.log(e);  
  
}  
  
}  
  
}
```

Test Case

Input:

```
const  
  
asyncTask =  
  
function  
  
(  
  
i  
  
) {  
  
return
```

new

Promise

((

resolve, reject

) => {

setTimeout(

()

=> resolve(

`Completing \${i}`

),

100

\*i)

});

}

const

promises = [

asyncTask(

3

),

asyncTask(



```
1
),
asyncTask(
7
),
asyncTask(
2
),
asyncTask(
5
),
];
asyncSeriesExecuter(promises);
```

Output:

"Completing 3"

"Completing 1"

"Completing 7"

"Completing 2"

"Completing 5"

62

## Approach 2 - Using recursion

We can execute the async tasks in series by recursively calling the same function after the current task is executed.

### Implementation

```
const
asyncSeriesExecuter =
function
(
promises
) {
// get the top task
let
promise = promises.shift();
//execute the task
promise.then((
data
) => {
//print the result
console
```

```
.log(data);  
  
//recursively call the same function  
  
if  
  
(promises.length >  
  
0  
  
) {  
  
  asyncSeriesExecuter(promises);  
  
}  
  
});  
  
}
```

Test Case

Input:

```
const  
  
asyncTask =  
  
function  
  
(  
  
i  
  
) {  
  
  return  
  
  new
```

Promise

```
((  
  resolve, reject  
) => {  
  setTimeout(  
    ()  
    => resolve(  
      `Completing ${i}`  
    ),  
    100  
    *i)  
  });  
}  
  
const  
promises = [  
  asyncTask(  
    3  
  ),  
  asyncTask(  
    1
```

```
),  
  asyncTask(  
    7  
  ),  
  asyncTask(  
    2  
  ),  
  asyncTask(  
    5  
  ),  
];  
  
asyncSeriesExecuter(promises);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

63

Output:

"Completing 3"

"Completing 1"

"Completing 7"

"Completing 2"

"Completing 5"

### Approach 3 - Using Array.reduce()

Rather than calling the function recursively, we can use the Array.reduce to do the execution in series.

#### Implementation

```
const
asyncSeriesExecuter =
function
(
promises
) {
promises.reduce((
acc, curr
) => {
return
acc.then(
()
=> {
return
curr.then(
val
```

```
=> {  
  console  
    .log(val));  
  });  
},  
Promise  
  .resolve();  
}
```

Test Case

Input:

```
const  
asyncTask =  
function  
(  
  i  
) {  
  return  
    new  
    Promise  
    ((
```

resolve, reject

) => {

setTimeout(

()

=> resolve(

`Completing \${i}`

),

100

\*i)

});

}

const

promises = [

asyncTask(

3

),

asyncTask(

1

),

asyncTask(



7

),

asyncTask(

2

),

asyncTask(

5

),

];

© JavaScript Interview Guide | learnersbucket.com

64

asyncSeriesExecuter(promises);

Output:

"Completing 3"

"Completing 1"

"Completing 7"

"Completing 2"

"Completing 5"

© JavaScript Interview Guide | learnersbucket.com

65

## Execute async functions in Parallel

### Problem Statement -

Implement a function that takes a list of async functions as input and a callback function and executes the input tasks in parallel i.e all at once and invokes the callback after every task is finished.

### Example

Input:

```
executeParallel(  
  [asyncTask(  
    3  
  ), asyncTask(  
    1  
  ), asyncTask(  
    2  
  )],  
  (result) => {  
    console  
      .log(result);  
  });  
Output:  
// output in the order of execution  
[
```

2

,

1

,

3

]

We can use the simple `forEach` loop on each task and execute them

parallel. We just need to use a tracker to determine the number of

tasks that have been finished so that when each task is executed, we can invoke the callback function.

Invoking the callback function at the end of the loop won't work as the async tasks may finish at different intervals.

function

`asyncParallel`

(

`tasks, callback`

) {

`// store the result`

`const`

`results = [];`

`// track the task executed`

```
let
tasksCompleted =
0
;
// run each task
tasks.forEach(
  asyncTask
=> {
  // invoke the async task
  © JavaScript Interview Guide | learnersbucket.com
66
  // it can be a promise as well
  // for a promise you can chain it with then
  asyncTask(
    value
    => {
    // store the output of the task
    results.push(value);
    // increment the tracker
    tasksCompleted++;
```

```
// if all tasks are executed

// invoke the callback

if

(tasksCompleted >= tasks.length) {

callback(results);

}

});

});

};
```

## Test Case

To create an async task, we have created a function that accepts a callback and runs a setTimeout for a random time and invokes this callback inside the timeout.

```
function

createAsyncTask

() {

const

value =

Math

.floor(
```

Math

.random() \*

10

);

return

function

(

callback

) {

setTimeout(

()

=> {

callback(value);

}, value \*

1000

);

};

}

Using this we can create multiple tasks and then test the async parallel.

Input:

```
const
```

```
taskList = [
```

```
  createAsyncTask(),
```

```
  createAsyncTask(),
```

```
  © JavaScript Interview Guide | learnersbucket.com
```

```
67
```

```
  createAsyncTask(),
```

```
  createAsyncTask(),
```

```
  createAsyncTask(),
```

```
  createAsyncTask()
```

```
];
```

```
asyncParallel(taskList, result => {
```

```
  console
```

```
    .log(
```

```
      'results'
```

```
    , result);
```

```
  });
```

Output:

```
"results"
```

```
// [object Array] (6)
```

```
[
```

```
1
```

```
,
```

```
6
```

```
,
```

```
7
```

```
,
```

```
7
```

```
,
```

```
9
```

```
,
```

```
9
```

```
]
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

68

Retry promises N number of times

Problem Statement -

Implement a function in JavaScript that retries promises N number of times with a delay between each call.

Example



Input:

```
retry(asyncFn, retries =  
3  
, delay =  
50  
, finalError  
=  
'Failed'  
);
```

Output:

```
... attempt  
1  
-> failed  
... attempt  
2  
-> retry after  
50  
ms -> failed  
... attempt  
3
```

-> retry after

50

ms -> failed

... Failed.

In short, we have to create a retry function that Keeps on retrying until the promise resolves with delay and max retries.

We will see two code examples one with [thenable](#) promise and the second with [async...await](#).

Let us first create the delay function so that we can take a pause for a specified amount of time.

Delay function

We can create a delay function by creating a new promise and resolve it after given time using setTimeout .

```
//delay func
```

```
const
```

```
wait =
```

```
ms
```

```
=>
```

```
new
```

```
Promise
```

```
((
```

```
resolve
```

```
) => {  
  
  setTimeout(  
  
    ()  
  
    => resolve(), ms);  
  
  });
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

69

### Approach 1 - Using then...catch

To retry the promise we have to call the same function recursively with reduced max tries, if the promise fails then in the .catch() method check if there are number of tries left then recursively call the same function or else reject with the final error.

### Implementation

```
const  
  
retryWithDelay = (  
  
  operation,  
  
  retries =  
  
  3  
  
  ,  
  
  delay =  
  
  50  
  
  ,
```

```
finalErr =  
'Retry failed'  
)=>  
new  
Promise  
((  
resolve, reject  
)  
=> {  
return  
operation()  
.then(resolve)  
.catch((  
reason  
)=> {  
//if retries are left  
if  
(retries >  
0  
) {
```

```
//delay the next call

return

wait(delay)

//recursively call the same function

//to retry with max retries - 1

.then(

retryWithDelay.bind(

null

,

operation,

retries -

1

,

delay,

finalErr)

)

.then(resolve)

.catch(reject);

}

// throw final error
```

```
return  
  
reject(finalErr);  
  
});
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
70
```

```
});
```

### Test Case

To test we can create a test function that keeps throwing errors if called less than 5 times. So if we call it 6 or more times, it will pass.

Input:

```
// Test function
```

```
const
```

```
getTestFunc =
```

```
()
```

```
=> {
```

```
let
```

```
callCounter =
```

```
0
```

```
;
```

```
return
```

```
async
() => {
  callCounter +=
    1
  ;
  // if called less than 5 times
  // throw error
  if
    (callCounter <
      5
    ) {
    throw
      new
        Error
          (
            'Not yet'
          );
    }
  }
}
```

```
// Test the code

const
test =
  async
  () => {
    await
    retryWithDelay(getTestFunc(),
    10
    );
    console
    .log(
    'success'
    );
    await
    retryWithDelay(getTestFunc(),
    3
    );
    console
    .log(
    'will fail before getting here'
```



```
);  
}  
  
// Print the result  
  
test().catch(  
  
console  
  
.error);
```

Output:

```
"success"  
  
// 1st test  
  
"Retry failed"  
  
//2nd test
```

Approach 2 - Using `async...await`.

We can implement the same function using `async-await`. When using `async-await`, we need to wrap the code inside `try..catch` block to handle

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

71

the error, thus in the catch block, we can check if the max retries are still left then recursively call the same function or else throw the final error.

Implementation

```
const
```

```
retryWithDelay =
```

```
async
(
  fn, retries =
    3
    , interval =
      50
      ,
      finalErr =
        'Retry failed'
    ) => {
  try
    {
      // try
      await
      fn();
    }
    catch
    (err) {
      // if no retries left
      // throw error
```

```
if
(retries <=
0
) {
return
Promise
.reject(finalErr);
}
//delay the next call
await
wait(interval);
//recursively call the same func
return
retryWithDelay(fn, (retries -
1
), interval,
finalErr);
}
}
```

Test Case

Input:

```
// Test function
```

```
const
```

```
getTestFunc =
```

```
()
```

```
=> {
```

```
let
```

```
callCounter =
```

```
0
```

```
;
```

```
return
```

```
async
```

```
() => {
```

```
callCounter +=
```

```
1
```

```
;
```

```
// if called less than 5 times
```

```
// throw error
```

```
if
```

```
(callCounter <
```

5

) {

throw

new

Error

(

'Not yet'

);

}

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

72

}

}

// Test the code

const

test =

async

() => {

await

retryWithDelay(getTestFunc(),

10

);

console

.log(

'success'

);

await

retryWithDelay(getTestFunc(),

3

);

console

.log(

'will fail before getting here'

);

}

// Print the result

test().catch(

console

.error);

Output:

"success"

// 1st test

"Retry failed"

//2nd test

Alternatively, you can also update the code and keep on retrying the API call based on some test.

© JavaScript Interview Guide | learnersbucket.com

73

Implement mapSeries async function

Problem Statement -

Implement a mapSeries async function that is similar to the

Array.map() but returns a promise that resolves on the list of output by mapping each input through an asynchronous iteratee function or

rejects it if any error occurs. The inputs are run in a sequence that is one after another.

The asynchronous iteratee function will accept an input and a

callback. The callback function will be called when the input is

finished processing, the first argument of the callback will be the error flag and the second will be the result.

Example

Input:

let

```
numPromise = mapSeries([
  1
  ,
  2
  ,
  3
  ,
  4
  ,
  5
],
function
(
  num, callback
) {
  // i am async iteratee function
  // do async operations here
  setTimeout(
    function
    () {
```



```
num = num *
```

```
2
```

```
;
```

```
console
```

```
.log(num);
```

```
// throw error
```

```
if
```

```
(num ===
```

```
12
```

```
){
```

```
callback(
```

```
true
```

```
);
```

```
}
```

```
else
```

```
{
```

```
callback(
```

```
null
```

```
, num);
```

```
}
```

```
    },  
    1000  
  );  
});  
numPromise  
  .then((  
    result  
  ) =>  
    console  
      .log(  
        "success:"  
        + result))  
    .catch(  
      ()  
    =>  
      console  
        .log(  
          "no success"  
        )  
    );
```

74

Output:

```
// each number will be printed after a delay of 2 seconds
```

2

4

6

8

10

```
"success:2,4,6,8,10"
```

```
// this will be printed immediately
```

after last

The implementation of this problem will be derived from the

[Async.series\(\)](#), the only difference will be that each input will be passed to the asynchronous iteratee function and the processed output from it will be returned in the output array.

We will return a new promise and inside this promise, iterate each

input value in the series using the `Array.reduce()` .

Pass the `Promise.resolve([])` with an empty array as the initial

accumulator to the `Array.reduce()` and inside the reduce, once the previous promise is resolved listen to it, create a new promise and pass the input value to the asynchronous iteratee function, based on the

callback result from this iteratee function, either resolve or reject the current promise.

This will go in sequence one after another.

```
const
mapSeries = (
arr, fn
) => {
// return a new promise
return
new
Promise
((
resolve, reject
) => {
const
output = [];
// for all the values in the input
// run it in series
// that is one after another
// initially it will take an empty array to resolve
// merge the output of the current with the previous
// and pass it on to the next
```

75

const

final = arr.reduce((

acc, current

) => {

return

acc.then((

val

) => {

// return a new promise

return

new

Promise

((

resolve, reject

) => {

// based on the callback value of

// async iteratee function

// resolve or reject

```
fn(current, (error, result) => {
```

```
  if
```

```
    (error){
```

```
      reject(error);
```

```
    }
```

```
    else
```

```
    {
```

```
      resolve([...val, result]);
```

```
    }
```

```
  });
```

```
});
```

```
});
```

```
},
```

```
Promise
```

```
  .resolve([]));
```

```
// based on the final promise state
```

```
// invoke the final promise.
```

```
final
```

```
  .then((
```

```
    result
```

```
) => {  
  resolve(result);  
})  
.catch((  
  error  
  ) => {  
    reject(error);  
  });  
});  
};
```

Test Case 1. All successful

Input:

```
let  
numPromise = mapSeries([  
  1  
  ,  
  2  
  ,  
  3  
  ,
```

4

,

5

],

function

(

num, callback

) {

setTimeout(

function

() {

num = num \*

2

;

console

.log(num);

// throw error

if

(num ===

12



```
) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
76
```

```
callback(
```

```
true
```

```
);
```

```
}
```

```
else
```

```
{
```

```
callback(
```

```
null
```

```
, num);
```

```
}
```

```
},
```

```
1000
```

```
);
```

```
});
```

```
numPromise
```

```
.then((
```

```
result
```

```
) =>  
console  
.log(  
"success:"  
+ result))  
.catch(  
()
```

```
=>  
console  
.log(  
"no success"  
));
```

Output:

```
// each number will be printed after a delay of 2 seconds
```

```
2
```

```
4
```

```
6
```

```
8
```

```
10
```

```
"success:2,4,6,8,10"
```

```
// this will be printed immediately
```

after last

Test Case 2. One fails

Input:

```
let
```

```
numPromise = mapSeries([
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
,
```

```
5
```

```
],
```

```
function
```

```
(
```

```
num, callback
```

```
) {
```

```
setTimeout(  
  function  
  () {  
    num = num *  
    2  
    ;  
    console  
    .log(num);  
    // throw error  
    if  
    (num ===  
    8  
    ){  
      callback(  
        true  
      );  
    }  
    else  
    {  
      callback(  

```

```
    null
    , num);
  }
},
2000
);
});
numPromise
.then((
result
) =>
console
.log(
"success:"
+ result))
.catch(
()
=>
console
.log(
```

```
"no success"
```

```
));
```

© JavaScript Interview Guide | learnersbucket.com

77

Output:

```
// each number will be printed after a delay of 2 seconds
```

```
2
```

```
4
```

```
6
```

```
8
```

```
"no success"
```

```
// this will be printed immediately after
```

```
last
```

© JavaScript Interview Guide | learnersbucket.com

78

Implement mapLimit async function

Problem Statement -

Implement a mapLimit function that is similar to the Array.map() but returns a promise that resolves on the list of output by mapping each input through an asynchronous iteratee function or rejects it if any error occurs. It also accepts a limit to decide how many operations can occur at a time.

The asynchronous iteratee function will accept an input and a

callback. The callback function will be called when the input is

finished processing, the first argument of the callback will be the error flag and the second will be the result.

Example

Input:

let

numPromise = mapLimit([

1

,

2

,

3

,

4

,

5

],

3

,

function

(

```
num, callback
) {
// i am async iteratee function
// do async operations here
setTimeout(
function
() {
num = num *
2
;
console
.log(num);
callback(
null
, num);
},
1000
);
});
numPromise
```



```
.then((  
  result  
) =>  
  console  
    .log(  
      "success:"  
      + result))  
  .catch(  
    ()  
    =>  
    console  
      .log(  
        "no success"  
      ));
```

Output:

```
/// first batch
```

```
2
```

```
4
```

```
6
```

```
/// second batch
```

79

8

10

/// final result

"success: [2, 4, 6, 8, 10]

To implement this function we will have to use the combination of both [Async.parallel](#) and Async.series .

- First [chop the input array](#) into the subarrays of the given limit.

This will return us an array of arrays like [[1, 2, 3], [4, 5]] .

- The parent array will run in series, that is the next subarray will execute only after the current subarray is done.
- All the elements of each sub-array will run in parallel.
- Accumulate all the results of each sub-array element and resolve the promise with this.
- If there is any error, reject.

// helper function to chop array in chunks of given size

Array

.prototype.chop =

function

(

```
size
) {
//temp array
const
temp = [...this];
//if the size is not defined
if
(!size) {
return
temp;
}
//output
const
output = [];
let
i =
0
;
//iterate the array
while
```

```
(i < temp.length) {  
  //slice the sub-array of a given size  
  //and push them in output array  
  output.push(temp.slice(i, i + size));  
  i = i + size;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

80

```
}
```

```
return
```

```
output;
```

```
};
```

```
const
```

```
mapLimit = (
```

```
arr, limit, fn
```

```
) => {
```

```
  // return a new promise
```

```
  return
```

```
  new
```

```
  Promise
```

```
((
```

resolve, reject

) => {

// chop the input array into the subarray of limit

// [[1, 2, 3], [1, 2, 3]]

let

chopped = arr.chop(limit);

// for all the subarrays of chopped

// run it in series

// that is one after another

// initially it will take an empty array to resolve

// merge the output of the subarray and pass it

on to the next

const

final = chopped.reduce((

a, b

) => {

// listen on the response of previous value

return

a.then((

val

```
) => {  
  // run the sub-array values in parallel  
  // pass each input to the iteratee function  
  // and store their outputs  
  // after all the tasks are executed  
  // merge the output with the previous one  
  and resolve  
  return  
  new  
  Promise  
  ((  
    resolve, reject  
  ) => {  
    const  
    results = [];  
    let  
    tasksCompleted =  
    0  
    ;  
    b.forEach((
```

e

) => {

// depending upon the output from

// async iteratee function

// reject or resolve

fn(e, (error, value) => {

if

(error){

reject(error);

}

else

{

results.push(value);

© JavaScript Interview Guide | learnersbucket.com

81

tasksCompleted++;

if

(tasksCompleted >= b.length) {

resolve([...val, ...results]);

}

```
}  
});  
});  
});  
});  
},
```

Promise

```
.resolve([]));  
  
// based on final promise state  
  
// invoke the final promise.  
  
final  
  
.then((  
  
result  
  
) => {  
  
resolve(result);  
  
})  
  
.catch((  
  
e  
  
) => {  
  
reject(e);
```



```
});
```

```
});
```

```
};
```

Test Case 1: All the inputs resolve.

Input:

```
let
```

```
numPromise = mapLimit([
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
,
```

```
5
```

```
],
```

```
3
```

```
,
```

```
function
```

```
(  
  num, callback  
) {  
  setTimeout(  
    function  
    () {  
      num = num *  
      2  
      ;  
      console  
      .log(num);  
      callback(  
        null  
        , num);  
      },  
      2000  
    );  
  });  
  numPromise
```

```
.then((  
  result  
) =>  
  console  
    .log(  
      "success:"  
      + result))  
  .catch(  
    ()  
    =>  
    console  
      .log(  
        "no success"  
      ));
```

Output:

```
// first batch
```

```
2
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

82

4

6

// second batch

8

10

"success:2,4,6,8,10"

Test Case 2: Rejects.

Input:

let

numPromise = mapLimit([

1

,

2

,

3

,

4

,

5

],

3

```
,  
function  
(  
  num, callback  
) {  
  setTimeout(  
    function  
    () {  
      num = num *  
        2  
    };  
    console  
      .log(num);  
    // throw error  
    if  
      (num ===  
        6  
      ){  
        callback(  
          true
```

```
);  
}  
else  
{  
  callback(  
    null  
    , num);  
}  
},  
2000  
);  
});  
numPromise  
  .then((  
    result  
  ) =>  
    console  
      .log(  
        "success:"  
        + result))
```

```
.catch(  
  ()  
=>  
  console  
    .log(  
      "no success"  
    ));
```

Output:

```
// first batch  
  
2  
  
4  
  
6  
  
"no success"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

83

Implement async filter function

Problem Statement -

Implement a function that takes an array of input and an async

iteratee function and returns a promise that resolves with the list of inputs that has passed the test through the iteratee function.

The inputs will run in parallel, but the output will be in the same order as the original.

The asynchronous iteratee function will accept an input and a callback. The callback function will be called when the input is finished processing, the first argument of the callback will be the error flag and the second will be the result.

Example

Input:

let

numPromise = filter([

1

,

2

,

3

,

4

,

5

],

function



```
(  
  num, callback  
) {  
  setTimeout(  
    function  
    () {  
      num = num *  
      2  
      ;  
      console  
      .log(num);  
      // throw error  
      if  
      (num ===  
      7  
      ){  
        callback(  
          true  
        );  
      }  
    }  
  )  
}
```

```
else
{
callback(
null
, num !==
4
);
}
},
2000
);
});
numPromise
.then((
result
) =>
console
.log(
"success:"
+ result))
```

```
.catch(  
  
()  
  
=>  
  
console  
  
.log(  
  
"no success"  
  
));
```

Output:

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

84

2

4

6

8

10

"success:1,3,4,5"

This function is a simple extension of `Array.filter()` to handle the async operations.

To implement this we will create a new promise and return it, inside this promise, run all the input values in parallel using the

`Array.forEach()` and inside the `forEach` pass each value to the iteratee function.

Inside the iteratee function's callback if the result is true which means the input has passed the test, then add that input to the result at the current index.

In the end, if we are at the last element of the input array, resolve the promise with the result.

Filter the final output to remove the gaps as we have to maintain the order for the passed values. There will be no value at the indexes of unpassed values in the output array.

```
const
filter = (
arr, fn
) => {
// return a new promise
return
new
Promise
((
resolve, reject
) => {
const
output = [];
let
track =
```

0

;

arr.forEach((

e, i

) => {

fn(e, (error, result) => {

// reject on error

if(error){

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

85

reject(error);

}

// track the no of inputs processed

track++;

// if the input passes the test

// add it to the current index

if

(result){

output[i] = e;

}

```
// if the last element of the input array
if
(track >= arr.length){
// filter the final output with truthy values
// to return the value in order
resolve(output.filter(
Boolean
));
}
});
});
});
};
```

Test Case

Input:

let

numPromise = filter([

1

,

2

```
,  
3  
  
,  
4  
  
,  
5  
  
],  
function  
(  
  num, callback  
) {  
  setTimeout(  
    function  
    () {  
      num = num *  
      2  
      ;  
      console  
      .log(num);  
      // throw error
```

```
if
(num ===
7
){
callback(
true
);
}
else
{
callback(
null
, num !==
4
);
}
},
2000
);
});
```



86

```
numPromise
```

```
.then((
```

```
result
```

```
) =>
```

```
console
```

```
.log(
```

```
"success:"
```

```
+ result))
```

```
.catch(
```

```
()
```

```
=>
```

```
console
```

```
.log(
```

```
"no success"
```

```
));
```

Output:

2

4

6

8

10

"success:1,3,4,5"

© JavaScript Interview Guide | learnersbucket.com

87

Implement async reject function

Problem Statement -

Implement a function that takes an array of input and an async

iteratee function and returns a promise that resolves with the list of inputs that has failed the test through the iteratee function. This

function is exactly the opposite of the [Async Filter](#).

The inputs will run in parallel, but the output will be in the same order as the original.

The asynchronous iteratee function will accept an input and a

callback. The callback function will be called when the input is

finished processing, the first argument of the callback will be the error flag and the second will be the result.

Example

Input:

let

numPromise = reject([

1

,

2

,

3

,

4

,

5

],

function

(

num, callback

) {

setTimeout(

function

() {

num = num \*

2

;

```
console
.log(num);
// throw error
if
(num ===
7
){
callback(
true
);
}
else
{
callback(
null
, num !==
4
);
}
},
```

2000

);

});

numPromise

.then((

result

) =>

console

.log(

"success:"

+ result))

.catch(

()

=>

console

.log(

"no success"

));

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

Output:

2

4

6

8

10

"success:2"

We can use the exact same implementation of the Async filter, only changing the logic to add the value to the output array only when it fails the test.

```
const
```

```
reject = (
```

```
arr, fn
```

```
) => {
```

```
// return a new promise
```

```
return
```

```
new
```

```
Promise
```

```
((
```

```
resolve, reject
```

```
) => {
```

```
const
output = [];
let
track =
0
;
arr.forEach((
e, i
) => {
fn(e, (error, result) => {
// reject on error
if
(error){
reject(error);
}
// track the no of inputs processed
track++;
// if input fails the test
// add it to the current index
if
```

```
(!result){  
  output[i] = e;  
}  
  
// if the last element of the input array  
if  
  
(track >= arr.length){  
  // filter the final output with truthy values  
  // to return the value in order  
  resolve(output.filter(  
    Boolean  
  ));  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

89

```
});
```

```
});
```

```
});
```

```
};
```

Test Case

let



```
numPromise = reject([
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
,
```

```
5
```

```
],
```

```
function
```

```
(
```

```
num, callback
```

```
) {
```

```
  setTimeout(
```

```
    function
```

```
    () {
```

```
      num = num *
```

```
2
```

```
;
console
.log(num);
// throw error
if
(num ===
7
){
callback(
true
);
}
else
{
callback(
null
, num !==
4
);
}
```

```
    },  
    2000  
  );  
});  
  
numPromise  
  .then((  
    result  
  ) =>  
    console  
      .log(  
        "success:"  
        + result))  
    .catch(  
      ()  
    =>  
      console  
        .log(  
          "no success"  
        ));
```

Output:

2

4

6

8

10

"success:2"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

90

Execute promises with the Priority

Problem Statement -

Given a list of promises and their priorities, call them parallelly and resolve with the value of the first promise with the most priority. If all the promises fail then reject with a custom error.

Example

```
const
```

```
promises = [
```

```
{status:
```

```
'resolve'
```

```
, priority:
```

```
4
```

```
},
```

```
{status:
  'reject'
, priority:
  1
},
{status:
  'resolve'
, priority:
  2
},
{status:
  'reject'
, priority:
  3
}
];

resolvePromisesWithPriority(promises);

// {status: 'resolve', priority: 2}
```

To solve this, in the `resolvePromisesWithPriority` function we will create a new promise and return it.

Sort the input promises in ascending order of priority and then

execute all of these in [parallel](#).

While executing, use a variable to track the most priority promise, if any promise rejects and it is the promise of most priority, update the variable to the next in priority.

In the end, if any promise resolves and it is of most priority, resolve it, else if all the promises fail then reject with the custom error.

To track if all the promises are finished, we will use another variable named `taskCompleted`.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

91

function

`resolvePromisesWithPriority`

(

`promises`

) {

`// sort the promises based on priority`

`promises.sort((`

`a, b`

`) => a.priority - b.priority);`

`// track the rejected promise`

`let`

`rejected = {};`

```
// track the result

let

result = {};

// track the position of the most priority

let

mostPriority =

0

;

// track the no of promises executed

let

taskCompleted =

0

;

// return a new promise

return

new

Promise

((

resolve, reject

) => {
```

```
// run each task in parallel

promises.forEach((

{task, priority}, i

) => {

// if the task is done

// store it in the result

task.then((

value

) => {

result[priority] = value;

}).catch((

error

) => {

// if the promise is rejected

// track the rejected promises just for

reference

rejected[priority] =

true

;

// if the rejected task is the least priority
```



```

one

// move to the next least priority

if

(priority === promises[mostPriority].priority){

mostPriority++;

}

}).finally(

()

=> {

// if the value priority is not reject

// and is the least priority

//resolve with these value

if

(!rejected[priority] && priority ===

promises[mostPriority].priority){

© JavaScript Interview Guide | learnersbucket.com

92

console

.log(rejected);

resolve(priority);

```

```
}  
  
// track the no of tasks completed  
  
taskCompleted++;  
  
// if all the tasks are finished and none  
of them have been  
resolved  
  
// reject with custom error  
  
if  
  
(taskCompleted === promises.length){  
reject(  
"All Apis Failed"  
);  
}  
  
});  
  
});  
  
});  
  
};
```

Test Case

Input:

```
// create a promise that rejects or resolves
```

```
// randomly after some time
```

```
function
```

```
createAsyncTask
```

```
() {
```

```
  const
```

```
  value =
```

```
    Math
```

```
    .floor(
```

```
    Math
```

```
    .random() *
```

```
    10
```

```
  );
```

```
  return
```

```
    new
```

```
    Promise
```

```
    ((
```

```
      resolve, reject
```

```
    ) => {
```

```
      setTimeout(
```

```
    )
```

```
=> {  
  if  
    (value <  
      7  
    ){  
      reject();  
    }  
  else  
    {  
      resolve(value);  
    }  
  }, value *  
    100  
  );  
});  
};  
  
const  
promises = [  
  {task: createAsyncTask(), priority:  
    1
```

```
},  
  
{task: createAsyncTask(), priority:  
4  
  
},  
  
{task: createAsyncTask(), priority:  
3  
  
},  
  
{task: createAsyncTask(), priority:  
2  
  
}  
  
];
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

93

```
resolvePromisesWithPriority(promises).then((  
result  
  
)=>{  
  
console  
  
.log(result);  
  
}, (error) => {  
  
console
```

```
.log(error);
```

```
});
```

Output:

```
/*
```

```
// log
```

```
// rejected promise as per their priority
```

```
{
```

```
"1": true,
```

```
"3": true,
```

```
"4": true
```

```
}
```

```
// log
```

```
// resolve promise as per their priority
```

```
2
```

```
*/
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

94

Dependent async tasks

Problem Statement -

Consider we have multiple async tasks A, B, C, D, and E ( not

promises). A, B, and C are independent tasks while D depends on A and B to perform its task while E depends on D and C to perform its task. Write a task function/class to solve this problem.

To solve this problem. We create a class that will take the dependencies and a callback function as input.

In the dependencies list, we will check if all the tasks in the list are completed or not, if it is completed then filter them out as we no longer need to run them.

If there are no items in dependencies, invoke the callback directly.

Otherwise, if there are dependencies pending, push them into a list, and execute them one by one. Once all are completed, invoke the callback.

Use a flag to determine the state of the task, i.e completed or not.

class

Task

{

// accept the dependencies list

// and the callback

constructor

(dependencies = [], job) {

// filter the dependencies that are not yet

```
completed
this
.dependencies = dependencies ? dependencies.filter(
dependency
=> dependency
instanceof
Task && !dependency.isCompleted)
: [];
this
.currentDependencyCount =
this
.dependencies.length;
// the callback
this
.job = job;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

95

```
// if current task is done
```

```
this
.isCompleted =
```



```
false

;

// store the dependencies list callback

// to execute is sequence

this

.subscribedList = [];

// start the job

this

.processJob();

}

processJob() {

// if there is dependency

// subscribe to each of them

if

(

this

.dependencies &&

this

.dependencies.length)

{
```

```
for
(
let
dependency
of
this
.dependencies)
{
dependency.subscribe(
this
.trackDependency.bind(
this
));
}
}

// else invoke the callback directly
else
{
this
.job(
```

```
this
.done.bind(
this
));
}
}

// if all the dependencies are executed
// invoke the callback
trackDependency() {
this
.currentDependencyCount--;
if
(
this
.currentDependencyCount ===
0
) {
this
.job(
this
```

```
.done.bind(  
  this  
));  
  
}  
  
}  
  
// push the callback to the list  
subscribe(cb) {  
  this  
    .subscribedList.push(cb);  
}  
  
// if the current task is done
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

96

```
// mark it as complete  
  
// invoke all the dependency callbacks.  
  
// to print it in sequence  
done() {  
  this  
    .isCompleted =  
  true
```

```
;
for
(
const
callback
of
this
.subscribedList)
{
callback();
}
}
```

Test Case

Input:

```
const
processA =
new
Task(
null
```

```
, (done) => {  
  setTimeout(  
    ()  
    => {  
      console  
        .log(  
          'Process A'  
        );  
      done();  
    },  
    100  
  );  
});  
  
const  
processB =  
new  
Task([processA], (done) => {  
  setTimeout(  
    ()  
    => {
```

```
console
.log(
'Process B'
);
done();
},
1500
);
});
const
processC =
new
Task(
null
, (done) => {
setTimeout(
()
=> {
console
.log(
```

'Process C'

);

done();

},

1000

);

});

const

processD =

new

Task([processA, processB], (done)

=> {

setTimeout(

()

=> {

console

.log(

'Process D'

);

done());



```
},
```

```
1000
```

```
);
```

```
});
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

97

```
const
```

```
processE =
```

```
new
```

```
Task([processC, processD], (done)
```

```
=> {
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
console
```

```
.log(
```

```
'Process E'
```

```
);
```

```
done();
```

```
},
```

100

);

});

const

createAllDoneInstance = (

allDoneCallback

) =>

new

Task([processA,

processB, processC, processD, processE], allDoneCallback);

createAllDoneInstance((

done

) => {

console

.log(

'All is done!'

);

done());

});

Output:

"Process A"

"Process C"

"Process B"

"Process D"

"Process E"

"All is done!"

© JavaScript Interview Guide | learnersbucket.com

98

Create Pausable auto incrementer

Problem Statement -

Create a pausable auto incrementor in JavaScript, which takes an

initial value and steps as input and increments the initial value with given steps every second. The incrementer can be paused and resumed

back.

It is one of the classic problems which use two of the trickiest concepts of JavaScript.

1. Timers.

2. Closure.

Use the [setInterval](#) to auto increment the values, whereas wrapping the start and stop function inside the closure and returning them, so that the incrementer can be controlled while still maintaining the

value.

Defining the function body

Our incrementor function will take two inputs, initial value, and steps.

And within the function, we will need two variables,

1. To track the value.
2. For storing the IntervalId, so that it can be cleared to stop the timer and resume when needed.

```
const
```

```
timer = (
```

```
init = 0, step = 1
```

```
) => {
```

```
let
```

```
intervalId;
```

```
let
```

```
count = init;
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

99

Start function

In the start function, setInterval will be invoked at an interval of 1

second, and in each interval call, the initial value will be increased by the given step and it will be logged in the console.

setInterval's id will be stored In the intervalId variable.

```
const
```

```
startTimer =
```

```
()
```

```
=> {
```

```
if
```

```
(!intervalId){
```

```
intervalId = setInterval(
```

```
()
```

```
=> {
```

```
console
```

```
.log(count);
```

```
count += step;
```

```
},
```

```
1000
```

```
);
```

```
}
```

```
}
```

There is a condition to check if intervalId is having any value or not, just to make sure we don't start multiple intervals.

Stop function

Inside the stop function we stop the increment by invoking the `clearInterval` by passing the `intervalId` to it and also updating the `intervalId` to `null`.

```
const  
stopTimer =  
(  
=> {  
  clearInterval(intervalId);  
  intervalId =  
  null  
  ;  
})
```

At the end return the `startTimer` and `stopTimer`.

```
return  
{  
  startTimer,  
  stopTimer,  
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

## Implementation

```
const
```

```
timer = (
```

```
init = 0, step = 1
```

```
) => {
```

```
let
```

```
intervalId;
```

```
let
```

```
count = init;
```

```
const
```

```
startTimer =
```

```
()
```

```
=> {
```

```
if
```

```
(!intervalId){
```

```
intervalId = setInterval(
```

```
()
```

```
=> {
```

```
console
```

```
.log(count);
```

```
count += step;

},

1000

);

}

}

const

stopTimer =

()

=> {

clearInterval(intervalId);

intervalId =

null

;

}

return

{

startTimer,

stopTimer,

};
```



```
}
```

Test Case

Input:

```
const
```

```
timerObj = timer(
```

```
10
```

```
,
```

```
10
```

```
);
```

```
//start
```

```
timerObj.startTimer();
```

```
//stop
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
timerObj.stopTimer();
```

```
},
```

```
5000
```

```
);
```

Output:

10

20

30

40

© JavaScript Interview Guide | learnersbucket.com

101

Implement queue using two stack

Problem Statement -

Implement [queue](#) data structure using two different [stacks](#) instead of using an array or object as a linked list.

List of operations performed on Priority Queue

- *enqueue(val)* : Adds the element in the queue.
- *dequeue()* : Removes the element from the queue.
- *peek()* : Returns the top element.

There are two different approaches which can be used to create a queue using stack.

1. Making the enqueue operation costly.
2. Making the dequeue operation costly.

Generally the enqueue and dequeue operation of the queue is performed in  $O(1)$  time.

But as we are using stack rather than array or linked list, either one of the operations will be running in  $O(n)$  time. It is up to us to decide which way we want to proceed.

Making enqueue operation costly

Every time an item is added to the queue first copy all the items from the stack1 to the stack2 then add the new element to the stack2.

Then again copy all the items back from stack2 to stack1.

This way we keep FIFO principal in place by keeping the first element at front and last element at end.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

102

Implementation

//Enqueue costly

class

QueueUsingStack

{

constructor

() {

this

.stack1 =

new

Stack();

```
this
.stack2 =
new
Stack();
}
enqueue = (
x
) => {
while
(!
this
.stack1.isEmpty()){
this
.stack2.push(
this
.stack1.pop());
}
this
.stack2.push(x);
while
```

```
(!  
this  
.stack2.isEmpty()){  
this  
.stack1.push(  
this  
.stack2.pop());  
}  
}  
dequeue =  
()  
=> {  
return  
this  
.stack1.pop();  
}  
peek =  
()  
=> {  
return
```

this

.stack1.peek();

}

size =

()

=> {

return

this

.stack1.size();

}

isEmpty =

()

=> {

return

this

.stack1.isEmpty();

}

clear =

()

=> {

```
this
.stack1 =
new
Stack();
this
.stack2 =
new
Stack();
}
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

103

Test Case

Input:

```
const
queue =
new
QueueUsingStack();
queue.enqueue(
```

10

```
);  
queue.enqueue(  
20  
);  
queue.enqueue(  
30  
);  
queue.enqueue(  
40  
);  
queue.enqueue(  
50  
);  
console  
.log(queue.peek());  
console  
.log(queue.dequeue());  
console  
.log(queue.peek());  
console
```



```
.log(queue.dequeue());
```

console

```
.log(queue.peek());
```

Output:

10

10

20

20

30

This is the most efficient way because once we have stored the data in an organized way all other operations work perfectly, we don't need to change them.

Making dequeue operation costly

In this we will store the element in LIFO order using stack, but while removing the element we will have to copy all the elements into

another stack so that the first element is at the top and can be removed.

And then copy all the elements back to the first stack.

The only problem with this way is that for the peek operation we have to do the same process, this makes two operations run in  $O(n)$  time

which makes it highly inefficient.

104

It is always better to store the data in an organized way so that all the later processes don't get affected.

Implementation

class

QueueUsingStack

{

constructor

() {

this

.stack1 =

new

Stack();

this

.stack2 =

new

Stack();

}

enqueue = (

x

) => {

this

.stack1.push(x);

}

dequeue =

()

=> {

while

(!

this

.stack1.isEmpty()){

this

.stack2.push(

this

.stack1.pop());

}

const

item =

this

.stack2.pop();

while

```
(!  
this  
.stack2.isEmpty()){  
this  
.stack1.push(  
this  
.stack2.pop());  
}  
return  
item;  
}  
peek =  
()  
=> {  
while  
(!  
this  
.stack1.isEmpty()){  
this  
.stack2.push(  

```

```
this
.stack1.pop();
}
const
item =
this
.stack2.peek();
while
(!
this
.stack2.isEmpty()){
this
.stack1.push(
this
.stack2.pop());
}
return
item;
}
```

105

size =

()

=> {

return

this

.stack1.size();

}

isEmpty =

()

=> {

return

this

.stack1.isEmpty();

}

clear =

()

=> {

this

.stack1 =

```
new  
Stack();  
this  
.stack2 =
```

```
new  
Stack();  
}  
}
```

Test Case

Input:

```
const  
queue =  
new  
QueueUsingStack();  
queue.enqueue(  
10  
);  
queue.enqueue(  
20  
);
```

```
queue.enqueue(  
30  
);  
queue.enqueue(  
40  
);  
queue.enqueue(  
50  
);  
console  
.log(queue.peek());  
console  
.log(queue.dequeue());  
console  
.log(queue.peek());  
console  
.log(queue.dequeue());  
console  
.log(queue.peek());
```

Output:



10

10

20

20

30

© JavaScript Interview Guide | learnersbucket.com

106

Implement a Stack using Queue

Problem Statement -

Implement a [stack](#) using a single [queue](#) .

Approach

- Every time we will add new data into the queue, we will move the existing data behind new data by repeatedly removing the first

data and pushing it after at the end of the queue.

- This way we will be able to mimic the stack implementation

using the queue operations.

Implementation

function

Stack

() {

let

```
queue =  
new  
Queue();  
//Push  
this  
.push =  
function  
(  
elm  
) {  
let  
size = queue.size();  
queue.enqueue(elm);  
//Move old data after the new data  
for  
(  
let  
i =  
0  
; i < size; i++) {
```

```
let  
  
x = queue.dequeue();  
  
queue.enqueue(x);  
  
}  
  
}  
  
//Pop  
  
this  
  
.pop =  
  
function  
  
(){  
  
if  
  
(queue.isEmpty()){  
  
return  
  
null  
  
;  
  
}  
  
return  
  
queue.dequeue();  
  
}
```

107

```
//Peek
```

```
this
```

```
.peek =
```

```
function
```

```
() {
```

```
  if
```

```
    (queue.isEmpty()) {
```

```
      return
```

```
      null
```

```
    ;
```

```
  }
```

```
  return
```

```
    queue.front();
```

```
  }
```

```
//Size
```

```
this
```

```
.size =
```

```
function
```

```
() {
```

```
return  
queue.size();  
}  
  
//IsEmpty  
  
this  
.isEmpty =  
  
function  
  
(){  
return  
queue.isEmpty();  
}  
  
//Clear  
  
this  
.clear =  
  
function  
  
(){  
queue.clear();  
return  
true  
;  
}
```

```
}  
  
//ToArray  
  
this  
  
.toArray =  
  
function  
  
(){  
  
return  
  
queue.toArray();  
  
}  
  
}
```

Test Case

Input:

```
let  
  
stack =  
  
new  
  
Stack();  
  
//creating new instance  
  
of Stack  
  
stack.push(  
  
1
```

```
);  
stack.push(  
2  
);  
stack.push(  
3  
);  
console  
.log(stack.peek());
```

```
console  
.log(stack.isEmpty());
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

108

```
console  
.log(stack.size());  
console  
.log(stack.pop());  
console  
.log(stack.toArray());  
console
```

```
.log(stack.size());  
stack.clear();  
//clear the stack  
console  
.log(stack.isEmpty());
```

Output:

```
3  
false  
3  
3  
[  
  2  
  ,  
  1  
]  
2  
true
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

109

Implement stack with max and min function



## Problem Statement -

Implement a [stack data structure](#) in which we can get the max and min value through function in  $O(1)$  time.

### Example

Input:

2

5

17

23

88

54

1

22

Output:

max:

88

min:

1

### Approach

- Instead of storing a single value in the stack we will store an object with current , max and min values.

- While adding the new value in the stack we will check if the stack is empty or not.
- If it is empty then add the current value as current , max and min values.
- Else get the previous value and compare it with the current item, if it is greater than the existing then replace the max , If it is less than the existing then replace the min .

## Implementation

function

stackWithMinMax

() {

let

items = [];

let

length =

0

;

this

.push = (

item

) => {

//Check if stack is empty

//Then add the current value at all place

```
if
```

```
(length ===
```

```
0
```

```
){
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
110
```

```
items[length++] = {value: item, max: item, min: item};
```

```
}
```

```
else
```

```
{
```

```
//Else get the top data from stack
```

```
const
```

```
data =
```

```
this
```

```
.peek();
```

```
let
```

```
{max, min} = data;
```

```
//If it is greater than previous then update
```

```
the max
```

```
max = max < item ? item : max;
```

```
//If it is lesser than previous then update
the min
min = min > item ? item : min;

//Add the new data
items[length++] = {value: item, max, min};
}
}

//Remove the item from the stack
this
.pop =
()
=> {
return
items[--length];
}

//Get the top data
this
.peek =
()
=> {
```

```
return  
  
items[length -  
  
1  
  
];  
  
}  
  
//Get the max value  
  
this  
  
.max =  
  
()  
  
=> {  
  
return  
  
items[length -  
  
1  
  
].max;  
  
}  
  
//Get the min value  
  
this  
  
.min =  
  
()  
  
=> {
```

```
return  
items[length -  
1  
].min;  
}  
  
//Get the size  
this  
.size =  
()  
=> {  
return  
length;  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

111

```
//Check if its empty  
this  
.isEmpty =  
()  
=> {
```

```
return
```

```
length ===
```

```
0
```

```
;
```

```
}
```

```
//Clear the stack
```

```
this
```

```
.clear =
```

```
()
```

```
=> {
```

```
length =
```

```
0
```

```
;
```

```
items = [];
```

```
}
```

```
}
```

Test Case

Input:

```
let
```

```
SM =
```

new

stackWithMinMax

();

SM.push(

4

);

SM.push(

7

);

SM.push(

11

);

SM.push(

23

);

SM.push(

77

);

SM.push(

3



```
);  
SM.push(  
1  
);  
SM.pop();  
console  
.log(  
`max: ${SM.max()}`  
,  
`min: ${SM.min()}`  
);
```

Output:

"max: 77"

"min: 3"

© JavaScript Interview Guide | learnersbucket.com

112

Implement two stack with an array

Problem Statement -

Create a data structure called twoStacks which will be using only a single array to store the data but will act as two different [stacks](#) .

The twoStacks data structure will perform following operations.

- *push1(elm)* : This will add data in the first stack.
- *push2(elm)* : This will add data in the second stack.
- *pop1()* : This will remove the data from the first stack.
- *pop2()* : This will remove the data from the second stack.

Example

Input:

let

stack =

new

twoStacks(

10

);

//Push data in first stack

stack.push1(

'stack1'

);

//Push data in second stack

stack.push2(

'stack2'

);

```
//Pop data from first stack  
  
console  
  
.log(stack.pop1());  
  
//Pop data from second stack  
  
console  
  
.log(stack.pop2());
```

Output:

"stack1"

"stack2"

There are two different ways in which we can implement this.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

113

Method 1: By dividing the array in two equal halves One of the approaches to implementing two stacks in an array is by

dividing the array in two equal halves and using these halves as two different stacks to store the data.

This method works fine, however it is not space efficient because

suppose we have two stacks with **4** and **6** elements and our array is of **10** length. No if we divide our array in two equal halves then it is going to have two stacks of length **5** . If we push only **4** items in the first stack then it has one space vacant and when we try to push **6** items in the second stack it will overflow because it only has a capacity of **5** . We could have used the **1** vacant space of the first stack to store the data.

Method 2: A space efficient method

This method is very space efficient and it does not overflow if there is space available in the array or any of the stack.

The concept we use here is we store the data on the two different ends in the array (from start and from end ).

The first stack stores the data from the front that is at the index **0** and the second stack stores the data from the end that is the index size-1.

Both stack push and pop data from opposite ends and to prevent the overflow we just need to check if there is space in the array.

Implementation

class

twoStacks

{

//Initialize the size of the stack

constructor

(n){

this

.size = n;

this

.top1 =

-1

;

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

114

this

.top2 = n;

this

.arr = [];

}

//Push in stack1

push1 = (

elm

) => {

//Check if there is space in array

//Push at the start of the array

if

(

this

.top1 <

this

.top2 -

1

){

```
this
.arr[++
this
.top1] = elm;
}
else
{
console
.log(
'Stack overflow'
);
return
false
;
}
}
//Push in stack2
push2 = (
elm
) => {
```

```
//Check if there is space in array
```

```
//Push at the end of the array
```

```
if
```

```
(
```

```
this
```

```
.top1 <
```

```
this
```

```
.top2 -
```

```
1
```

```
){
```

```
this
```

```
.arr[--
```

```
this
```

```
.top2] = elm;
```

```
}
```

```
else
```

```
{
```

```
console
```

```
.log(
```

```
'Stack overflow'
```

```
);  
return  
false  
;  
}  
}  
  
//Pop from the stack 1  
pop1 =  
()  
=> {  
//Check if stack1 has data  
//Remove it from the front of the stack  
if  
(  
this  
.top1 >=  
0  
) {  
let  
elm =
```



```
this
.arr[
this
.top1];
this
.top1--;
return
elm;
}
else
{
console
.log(
'stack underflow'
);
return
false
;
}
}
```

115

```
//Pop from the stack 2
```

```
pop2 =
```

```
()
```

```
=> {
```

```
//Check if stack2 has data
```

```
//Remove it from the end of the array
```

```
if
```

```
(
```

```
this
```

```
.top2 <
```

```
this
```

```
.size){
```

```
let
```

```
elm =
```

```
this
```

```
.arr[
```

```
this
```

```
.top2];
```

```
this
.top2++;
return
elm;
}
else
{
console
.log(
'stack underflow'
);
return
false
;
}
}
```

Test Case

Input:

let

```
stack =  
new  
twoStacks(  
10  
);  
//push in first stack  
stack.push1(  
'stack1'  
);  
//push in second stack  
stack.push2(  
'stack2'  
);  
//pop from first stack  
console  
.log(stack.pop1());  
//pop from second stack  
console  
.log(stack.pop2());
```

Output:

"stack1"

"stack2"

© JavaScript Interview Guide | learnersbucket.com

116

## Implement Priority Queue

What is a priority queue?

As queues are widely used in computer programming and in real lives as well, there was a need for some different models of original [queue data structure](#) to process the data more efficiently.

A priority queue is one of the variants of the original queue. These elements are added and removed based on their priorities. It is an

abstract data type that captures the idea of a container whose elements have priorities attached to them. An element of highest priority always appears at the front of the queue. If that element is removed, the next highest priority element advances to the front.

A real-life example of the priority queue are the patients in the hospitals, the one with at most priority are treated first and then the others.

Another example is people standing in a queue at the boarding line at the airport, first and second class (Business class) passengers get priority over the coach class (Economy).

Why do we need a priority queue?

It is used when we have to choose between the same values who have different priorities or weight.

- *Dijkstra's Shortest Path Algorithm using priority queue* : When the graph is stored in the form of adjacency list or matrix,

priority queue can be used to extract minimum efficiently when

implementing Dijkstra's algorithm.

- *Prim's algorithm* : to store keys of nodes and extract minimum key nodes at every step.

© JavaScript Interview Guide | learnersbucket.com

117

- *Data compression* : It is used in Huffman Codes which is used to compress data.

- *Operating system* : It is used by operating systems for load balancing.

List of operations performed on Priority Queue

- *enqueue()* : Adds an item at the tail of the queue.
- *dequeue()* : Removes an item from the head of the queue.
- *front()* : Returns the first item in the queue.
- *rear()* : Returns the last item in the queue.
- *size()* : Returns the size of the queue.
- *isEmpty()* : Returns true if the queue is empty, false otherwise.

There are two ways of implementing a priority queue.

- Add elements at appropriate places based on their priorities.
- Queue elements as they are added and remove them according to their priorities.

We will be using the first approach as we just have to place the elements at the appropriate place and then it can be dequeued normally.

## Implementation

function

PriorityQueue

() {

let

items = [];

//Container

function

QueueElement

(

element, priority

) {

this

.element = element;

this

.priority = priority;

}

```
//Add a new element in queue
```

```
this
```

```
.enqueue =
```

```
function
```

```
(
```

```
element, priority
```

```
){
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
118
```

```
let
```

```
queueElement =
```

```
new
```

```
QueueElement(element, priority);
```

```
//To check if element is added
```

```
let
```

```
added =
```

```
false
```

```
;
```

```
for
```

```
(
```



```
let
i =
0
; i < items.length; i++){
//We are using giving priority to higher numbers
//If new element has more priority then add
it at that place
if
(queueElement.priority > items[i].priority){
items.splice(i,
0
, queueElement);
//Mark the flag true
added =
true
;
break
;
}
}
```

```
//If element is not added

//Then add it to the end of the queue

if

(!added){

items.push(queueElement);

}

}

//Remove element from the queue

this

.dequeue =

()

=> {

return

items.shift();

}

//Return the first element from the queue

this

.front =

()

=> {
```

```
return  
items[  
0  
];  
}  
  
//Return the last element from the queue  
this  
.rear =  
()  
=> {  
return  
items[items.length -  
1  
];  
}  
  
//Check if queue is empty  
this  
.isEmpty =  
()  
=> {
```

```
return  
  
items.length ==  
  
0  
  
;  
  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

119

```
//Return the size of the queue  
  
this  
  
.size =  
  
()  
  
=> {  
  
return  
  
items.length;  
  
}  
  
//Print the queue  
  
this  
  
.print =  
  
function  
  
(){
```

```
for
(
let
i =
0
; i < items.length; i++){
console
.log(
`${items[i].element} - ${items[i].priority}`
);
}
}
}
```

Test Case

Input:

```
let
pQ =
new
PriorityQueue();
pQ.enqueue(
```

1

,

3

);

pQ.enqueue(

5

,

2

);

pQ.enqueue(

6

,

1

);

pQ.enqueue(

11

,

1

);

pQ.enqueue(

13

,

1

);

pQ.enqueue(

10

,

3

);

pQ.dequeue();

pQ.print();

Output:

"10 - 3"

"5 - 2"

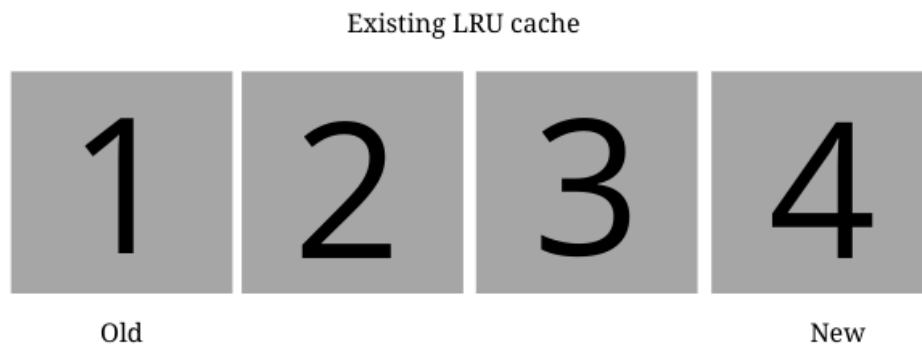
"6 - 1"

"11 - 1"

"13 - 1"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

120



## Implement LRU cache

What is a LRU cache?

LRU cache, briefly known as *Least Recently Used* cache is a caching policy that is used to evict elements from the cache to make room for new elements when the cache is full. It removes the least recently used elements first before adding the new one.

Cache, as you know, is one of the important concepts of computer

science, because it drastically speeds up things by storing things in memory. Thus it can store a single element or whole index to optimize it.

How does LRU cache work?

Let us understand how LRU cache works by taking an example of a cache of 4 elements, say 1, 2, 3, 4.

In order to add a new element to this (say 5), we will have to first remove the least recently used which in this case is 1.

© JavaScript Interview Guide | learnersbucket.com



Adding new element 5 in LRU cache, so we will have to remove the older to add new one.



Adding 2 back in the LRU cache



Now when you try to access the 2 again it will be shifted to the end again as it will be the most recently used one and 3 will become the least recently used.

### Implementing LRU cache in Javascript?

We use two data structures together to implement a LRU Cache.

- [Queue](#) : which is implemented using a [doubly-linked list](#) . The most recently used items will be near the front end and the least recent

pages will be near the rear end.

- [HashMap](#) : With item / page number as key and address of the corresponding queue node as value.

© JavaScript Interview Guide | learnersbucket.com

Following is the list of operations that will be formed on the LRU cache.

- *get(key)* : Returns the cache value for the given item / page number.
- *put(key, val)* : Adds a new value in the cache.
- *use(key)* : Uses one of the existing values and re-arranges the cache by marking the used one as most recently one.
- *evict()* : Removes a value from the cache.
- *Insert(key, val)* : A helper function to add value in cache while performing put

Base Function

class

Node

{

constructor

(key, val) {

this

.key = key;

this

.val = val;

this

.prev =

```
    null
    ;
    this
    .next =
    null
    ;
  }
}

const
LRUCache =
function
(
  cap
) {
  this
  .cap = cap;
  this
  .count =
  0
  ;
```

```
this
.head =
null
;
this
.tail =
null
;
this
.cache =
new
Map
();
//other methods will go here
}
```

Use(Key)

- First get the queue node of the key from the hashmap.
- Then rearrange the doubly linked list to push the current node at the end or tail marking it as most recently used.

//Uses the cache with given key and marks it as most recently used

123

this

.use =

function

(

key

) {

const

node =

this

.cache.get(key);

if

(node ===

this

.head) {

return

;

}

else

if

```
(node ===  
this  
.tail) {  
node.prev.next =  
null  
;  
this  
.tail = node.prev;  
node.prev =  
null  
;  
node.next =  
this  
.head;  
this  
.head.prev = node;  
this  
.head = node;  
}  
else
```

```
{  
  if  
    (node.prev) {  
      node.prev.next = node.next;  
    }  
  if  
    (node.next) {  
      node.next.prev = node.prev;  
    }  
  node.next =  
    this  
      .head;  
  node.prev =  
    null  
  ;  
  this  
    .head.prev = node;  
  this  
    .head = node;  
}
```

```
};
```

```
Evict
```

- First get the queue node from the hashmap and then remove this node from it and re-arrange all the nodes.
- Then delete it from the hashmap as well.

```
//Removes the least recently used cache
```

```
this
```

```
.evict =
```

```
function
```

```
() {
```

```
const
```

```
keyToEvict =
```

```
this
```

```
.tail ?
```

```
this
```

```
.tail.key
```

```
:
```

```
null
```

```
;
```

```
if
```



```
(!  
this  
.tail) {  
return  
;  
}  
else  
if  
(  
this  
.head ===  
this  
.tail) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

124

```
this  
.head =  
null  
;  
this
```

```
.tail =  
null  
;  
}  
else  
{  
this  
.tail.prev.next =  
null  
;  
this  
.tail =  
this  
.tail.prev;  
}  
if  
(keyToEvict) {  
this  
.count--;  
this
```

```
.cache.delete(keyToEvict);
```

```
}
```

```
};
```

```
Insert(key, val)
```

Adds a new element at the appropriate position in the queue and also inserts it in the hashmap.

```
//Helper function to add new cache in the queue
```

```
this
```

```
.insert =
```

```
function
```

```
(
```

```
key, val
```

```
) {
```

```
const
```

```
node =
```

```
new
```

```
Node(key, val);
```

```
this
```

```
.count++;
```

```
this
```

```
.cache.set(key, node);
```

```
if
(
!
this
.head) {
this
.head = node;
this
.tail = node;
}
else
{
this
.head.prev = node;
node.next =
this
.head;
this
.head = node;
}
};
```

Put(key, val)

- If the key already exists then use it and mark it as the most recent one.
- If the capacity is exceeded then remove the least recent one and then insert the new key.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

125

//Adds new item in the list

this

.put =

function

(

key, val

) {

if

(

this

.cache.has(key)) {

const

node =

this

.cache.get(key);

```
node.val = val;

this

.use(key);

this

.cache.set(key, node);

}

else

{

if

(

this

.count >=

this

.cap) {

this

.evict();

}

this

.insert(key, val);

this
```

```
.use(key);
```

```
// may not be needed
```

```
}
```

```
};
```

```
Get(key)
```

Returns the queue node of the associated key.

```
//Returns the value of the given key
```

```
this
```

```
.get =
```

```
function
```

```
(
```

```
key
```

```
) {
```

```
if
```

```
(!
```

```
this
```

```
.cache.has(key)) {
```

```
return
```

```
-1
```

```
;
```

```
}  
  
const  
node =  
  
this  
  
.cache.get(key);  
  
this  
  
.use(key);  
  
return  
  
node.val;  
  
};
```

Display()

Prints all the items of the queue in the least to most order along with its value.

```
//Display the list
```

```
this  
  
.display =  
  
function  
  
{  
  
let  
  
current =  
  
this
```



```
.head;
```

```
while
```

```
(current){
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

126

```
console
```

```
.log(current.key, current.val);
```

```
current = current.next;
```

```
}
```

```
}
```

Complete code of LRU cache implementation

```
class
```

```
Node
```

```
{
```

```
constructor
```

```
(key, val) {
```

```
this
```

```
.key = key;
```

```
this
```

```
.val = val;
```

```
this
    .prev =
    null
    ;
    this
    .next =
    null
    ;
}
}

const
LRUCache =
function
(
    cap
) {
    this
    .cap = cap;
    this
    .count =
```

0

;

this

.head =

null

;

this

.tail =

null

;

this

.cache =

new

Map

();

//Returns the value of the given key

this

.get =

function

(

```
key
) {
  if
  (!
  this
  .cache.has(key)) {
    return
    -1
    ;
  }
  const
  node =
  this
  .cache.get(key);
  this
  .use(key);
  return
  node.val;
};

//Adds new item in the list
```

```
this

.put =

function

(

key, val

) {

if

(

this

.cache.has(key)) {

const

node =

this

.cache.get(key);

node.val = val;

this

.use(key);

this

.cache.set(key, node);

}
```

```
else
```

```
{
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
127
```

```
if
```

```
(
```

```
this
```

```
.count >=
```

```
this
```

```
.cap) {
```

```
this
```

```
.evict();
```

```
}
```

```
this
```

```
.insert(key, val);
```

```
this
```

```
.use(key);
```

```
// may not be needed
```

```
}
```

```
};
```

```
//Uses the cache with given key and marks it as  
most recently used  
  
this  
  
.use =  
  
function  
  
(  
key  
  
) {  
    const  
  
    node =  
  
    this  
  
.cache.get(key);  
  
    if  
  
(node ===  
  
this  
  
.head) {  
  
        return  
  
        ;  
  
    }  
  
    else
```

```
if
(node ===
this
.tail) {
node.prev.next =
null
;
this
.tail = node.prev;
node.prev =
null
;
node.next =
this
.head;
this
.head.prev = node;
this
.head = node;
}
```



```
else
{
if
(node.prev) {
node.prev.next = node.next;
}
if
(node.next) {
node.next.prev = node.prev;
}
node.next =
this
.head;
node.prev =
null
;
this
.head.prev = node;
this
.head = node;
```

```
}
```

```
};
```

```
//Removes the least recently used cache
```

```
this
```

```
.evict =
```

```
function
```

```
() {
```

```
const
```

```
keyToEvict =
```

```
this
```

```
.tail ?
```

```
this
```

```
.tail.key
```

```
:
```

```
null
```

```
;
```

```
if
```

```
(!
```

```
this
```

```
.tail) {
```

128

return

;

}

else

if

(

this

.head ===

this

.tail) {

this

.head =

null

;

this

.tail =

null

;

```
}  
  
else  
  
{  
  
this  
  
.tail.prev.next =  
  
null  
  
;  
  
this  
  
.tail =  
  
this  
  
.tail.prev;  
  
}  
  
if  
  
(keyToEvict) {  
  
this  
  
.count--;  
  
this  
  
.cache.delete(keyToEvict);  
  
}  
  
};
```

```
//Helper function to add new cache in the queue
```

```
this
```

```
.insert =
```

```
function
```

```
(
```

```
key, val
```

```
) {
```

```
const
```

```
node =
```

```
new
```

```
Node(key, val);
```

```
this
```

```
.count++;
```

```
this
```

```
.cache.set(key, node);
```

```
if
```

```
(!
```

```
this
```

```
.head) {
```

```
this
```

```
.head = node;

this

.tail = node;

}

else

{

this

.head.prev = node;

node.next =

this

.head;

this

.head = node;

}

};

//Display the list

this

.display =

function

(){
```

```
let
current =
this
.head;
while
(current){
console
.log(current.key, current.val);
current = current.next;
}
}
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

129

Test Case

Input:

```
const
```

```
lru =
```

```
new
```

```
LRUCache(
```

4

);

lru.put(

1

,

'a'

);

lru.put(

2

,

'b'

);

lru.put(

3

,

'c'

);

lru.put(

4

,



'd'

);

lru.display();

lru.use(

2

);

lru.display();

Output:

//LRU

4

"d"

3

"c"

2

"b"

1

"a"

//After using 2

2

"b"

4

"d"

3

"c"

1

"a"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

130

Implement debounce function

What is debouncing a function?

Debouncing is a method or a way to execute a function when it is made sure that no further repeated event will be triggered in a given frame of time.

A technical example is when we search something on any eCommerce

site, we don't want to trigger a search function and make a request to the server as the user keeps typing each letter. We want the user to finish typing and then wait for a specified window of time to see if the user is not going to type anything else or has finished typing then

make the request and return the result.

Excessively invoking the function majorly hampers the performance

and is considered as one of the key hurdles.

Implementation

const

```
debounce = (  
  func, delay  
) => {  
  // 'private' variable to store the instance  
  // in closure each timer will be assigned to it  
  let  
    inDebounce;  
  // debounce returns a new anonymous function (closure)  
  return  
    function  
      () {  
        // reference the context and args for the setTimeout  
        function  
          const  
            context =  
              this  
            ;  
          const  
            args =  
              arguments
```

```
;
// base case

// clear the timeout to assign the new timeout
to it.

// when event is fired repeatedly then this helps
to reset

clearTimeout(inDebounce);

// set the new timeout and call the original function
with apply

inDebounce = setTimeout(

()

=> func.apply(context,

args), delay);

© JavaScript Interview Guide | learnersbucket.com

131

};

};
```

### Explanation

We created a function that will return a function. The outer function uses a variable to keep track of timerId for the execution of the inner function.

The inner function will be called only after a specified window of time, to achieve this we use `setTimeout` function.

We store the reference of the `setTimeout` function so that we can clear it if the outer function is re-invoked before the specified time

`clearTimeout(inDebounce)` and again recall it.

If we are invoking for the first time, our function will execute at the end of our delay. If we invoke and then reinvoke again before the end of our delay, the delay restarts.

Test Case

Input:

```
// print the mouse position
```

```
const
```

```
onMouseMove = (
```

```
e
```

```
) => {
```

```
  console
```

```
    .clear();
```

```
  console
```

```
    .log(e.x, e.y);
```

```
}
```

```
// define the debounced function
```

```
const
```

```
debouncedMouseMove = debounce(onMouseMove,  
50  
);  
// call the debounced function on every mouse move  
window  
.addEventListener(  
'mousemove'  
, debouncedMouseMove);
```

Output:

300 400

© JavaScript Interview Guide | learnersbucket.com

132

Implement debounce function with

Immediate Flag

Debouncing is a method or a way to execute a function when it is made sure that no further repeated event will be triggered in a given frame of time.

In the last example we had seen how to implement the classic

[debounce function with delay](#). In this implementation we will add one

extra flag called immediate to execute the function immediately

without any further delay. After initial execution it won't run again till the delay.

This flag will be optional which means if it is set to false then the debounce function will behave normally.

## Implementation

```
const
debounce = (
func, wait, immediate
) => {
// 'private' variable to store the instance
// in closure each timer will be assigned to it
let
timeout;
// debounce returns a new anonymous function (closure)
return
function
() {
// reference the context and args for the setTimeout
function
let
context =
this
,
```

```
args =  
arguments  
;  
  
// should the function be called now? If immediate  
is true  
  
// and not already in a timeout then the answer  
is: Yes  
  
const  
callNow = immediate && !timeout;  
  
// base case  
  
// clear the timeout to assign the new timeout  
to it.  
  
// when event is fired repeatedly then this helps  
to reset  
  
clearTimeout(timeout);  
  
© JavaScript Interview Guide | learnersbucket.com  
  
133  
  
// set the new timeout  
  
timeout = setTimeout(  
function
```



```
() {  
  // Inside the timeout function, clear the timeout  
  variable  
  
  // which will let the next execution run when  
  in 'immediate' mode  
  
  timeout =  
  
  null  
  
  ;  
  
  // check if the function already ran with the  
  immediate flag  
  
  if  
  
  (!immediate) {  
    // call the original function with apply  
    func.apply(context, args);  
  }  
  
  }, wait);  
  
  // immediate mode and no wait timer? Execute the  
  function immediately  
  
  if  
  
  (callNow) func.apply(context, args);
```

```
}
```

```
}
```

### Explanation

We use a private variable to store the timerId, and return a closure from it which will execute the callback function after debouncing.

Inside the closure, store the context of the current function and its arguments and pass it later to the callback function while executing it with the help of apply method.

Also create a flag to check if the debounce should happen immediately or not `const callNow = immediate && !timeout`; this will help to execute the callback immediately only when it is not in the timeout

and immediate flag is true.

Then assign the timer and execute the function only when the immediate flag is false.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

134

At the end, execute the callback if callNow is true.

### Test Case

Input:

```
// print the mouse position
```

```
const
```

```
onMouseMove = (
```

```
e
```

```
) => {  
  console  
  .clear();  
  console  
  .log(e.x, e.y);  
}  
  
// define the debounced function  
  
const  
  
debouncedMouseMove = debounce(onMouseMove,  
  
50  
  
);  
  
// call the debounced function on every mouse move  
  
window  
  
  .addEventListener(  
  
  'mousemove'  
  
  , debouncedMouseMove);
```

Output:

314 419

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

Implement throttling function

What is throttling a function?

Throttling is a way/technique to restrict the number of function execution/call in the specified amount of time.

Excessive function invocations in applications hamper the performance drastically. To optimize an app we need to handle this correctly.

There are scenarios where we may invoke functions when it isn't necessary. For example, consider a scenario where we want to make an API call to the server on a button click.

If the user spam's the click then this will make an API call on each click. This is not what we want, we want to restrict the number of API calls that can be made. The other call will be made only after a specified interval of time.

Throttling helps us to gain control over the rate at which function is called or executes.

Implementation

```
const
```

```
throttle = (
```

```
func, limit
```

```
) => {
```

```
// track the timerid and time
```

```
let
lastFunc;

let
lastRan;

return

function

() {
// capture the context and arguments

const

context =

this

;

const

args =

arguments

;

// if not yet run, run it once
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

136

if

```
(!lastRan) {  
    func.apply(context, args);  
    lastRan =  
        Date  
            .now();  
}  
else  
{  
    // reset the timer  
    clearTimeout(lastFunc);  
    // start it again  
    lastFunc = setTimeout(  
        function  
        () {  
            if  
            ((  
                Date  
                    .now() - lastRan) >= limit) {  
                func.apply(context, args);  
                lastRan =
```

Date

```
.now();
```

```
}
```

```
}, limit - (
```

Date

```
.now() - lastRan));
```

```
}
```

```
}
```

```
}
```

Test Case

Input:

```
const
```

```
print =
```

```
()
```

```
=> {
```

```
console
```

```
.log(
```

```
"hello"
```

```
);
```

```
}
```

```
const  
throttled = throttle(print,  
2500  
);  
window  
.addEventListener(  
'mousemove'  
, throttled,  
false  
);
```

Output:

```
// "hello" at the beginning  
// "hello" after 2500 millisecond  
// "hello" after 2500 millisecond
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

137

What is the difference between debouncing and throttling?

- *Debouncing* :- It is used to invoke/call/execute functions only when things have stopped happening for a given specific time.

For example, Call a search function to fetch the result when the user has stopped typing in the search box. If the user keeps on



typing then reset the function.

- *Throttling* :- It is used to restrict the no of time a function can be called/invoked/executes. For example, making an API call to the server on the user's click. If the user spam's the click then also there will be specific calls only. Like, make each call after 10 seconds.

© JavaScript Interview Guide | learnersbucket.com

138

Implement custom instanceof method

Definition

According to MDN –

The instanceof operator tests to see if the prototype property of a constructor appears anywhere in the prototype chain of an object. The return value is a boolean value.

From the definition, we can derive that to create a custom instanceof method we will have to check if the prototype property of a constructor appears anywhere in the prototype chain of an object .

Thus we will have to check first, if the provided input is an object or not and later keep checking if the prototype property appears anywhere in the prototype chain.

To get the prototype of a constructor we can use `__proto__` or `getPrototypeOf(object)` method.

Iterative implementation with `__proto__`

```
const
instanceOf = (
obj, target
) => {
// if provided input is not object type, return
false
if
(obj ===
null
||
typeof
obj !==
'object'
)
return
false
;
// keep checking in the prototype chain
while
(obj){
```

```
if
(obj.__proto__ === target.prototype)

return

true

;

obj = obj.__proto__;

}

return

false

;

}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

139

Test Case

Input:

class

P

{}

class

Q

extends

P

{}

const

q =

new

Q()

console

.log(instanceOf(q, Q))

// true

console

.log(instanceOf(q, P))

// true

console

.log(instanceOf(q,

Object

))

// true

function

R

```
() {}
```

```
console
```

```
.log(instanceOf(q, R))
```

```
// false
```

```
R.prototype = Q.prototype
```

```
console
```

```
.log(instanceOf(q, R))
```

```
// true
```

```
R.prototype = {}
```

```
console
```

```
.log(instanceOf(q, R))
```

```
// false
```

Recursive implementation with `getPrototypeOf(object)`

```
const
```

```
instanceOf = (
```

```
  obj, target
```

```
) => {
```

```
  // if provided input is not object type, return
```

```
  false
```

```
  if
```

```
(obj ==  
null  
||  
typeof  
obj !==  
'object'  
)  
return  
false  
;  
  
// get the prototype  
  
const  
proto =  
Object  
.getPrototypeOf(obj);  
  
// recursively test if prototype matches to the  
target's prototype  
  
return  
proto === target.prototype ?  
true
```

```
: instanceof(proto,  
target);  
}
```

Test Case

Input:

```
class
```

```
P
```

```
{}
```

```
class
```

```
Q
```

```
extends
```

```
P
```

```
{}
```

```
const
```

```
q =
```

```
new
```

```
Q()
```

```
console
```

```
.log(instanceOf(q, Q))
```

```
// true
```

```
console
.log(instanceOf(q, P))
// true
```

```
console
.log(instanceOf(q,
Object
))
// true
```

```
function
```

```
R
```

```
() {}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

140

```
console
.log(instanceOf(q, R))
// false
```

```
R.prototype = Q.prototype
```

```
console
.log(instanceOf(q, R))
// true
```



```
R.prototype = {}  
  
console  
  
.log(instanceOf(q, R))  
  
// false
```

© JavaScript Interview Guide | learnersbucket.com

141

Check if function is called with new keyword

There are multiple ways to invoke functions in JavaScript and one of the ways is with the “new” keyword calling function as a constructor.

Determine if the function is called with the “new” keyword.

Method 1: Using instanceof

The simplest way is by using the [instanceof](#) method, but it really does not cover all the edge cases.

```
function
```

```
A
```

```
() {
```

```
if
```

```
((
```

```
this
```

```
instanceof
```

```
arguments
```

```
.callee) ) {  
  console  
  .log(  
    "OK, new"  
  );  
}  
else  
{  
  console  
  .log(  
    "OK, function"  
  );  
}  
}
```

Test Case

Input:

var

Z =

new

A();

```
Z.lolol = A;
```

```
Z.lolol();
```

Output:

```
// OK, new
```

Even though `Z.lolol()` is called as a normal function it is invoked as a constructor. The above code doesn't work if you assign the constructor to the same object it has created.

To solve this we will need some additional checks along with the `instanceOf` method.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

142

Using one extra flag to determine if the function is constructed or not does the work.

```
function
```

```
A
```

```
() {
```

```
// You could use arguments.callee instead of x
```

```
here,
```

```
// except in EcmaScript 5 strict mode.
```

```
if
```

```
(
```

```
this
```

```
instanceof
A && !
this
._constructed )
{
this
._constructed =
true
;
console
.log(
'OK, new'
);
}
else
{
console
.log(
'OK, function'
);
```

```
}
```

```
}
```

Test Case

Input:

```
A();
```

```
// Ok, function
```

```
new
```

```
A();
```

```
// OK, new
```

```
new
```

```
A();
```

```
// Ok, new
```

```
A(
```

```
4
```

```
);
```

```
// OK, function
```

```
var
```

```
X =
```

```
new
```

```
A(
```

4

);

// OK, new

var

Z =

new

A();

// OK, new

Z.lolol = A;

Z.lolol();

// OK, function

var

Y = A;

Y();

// OK, function

var

y =

new

Y();

// OK, new

```
y.lolol = Y;
```

```
y.lolol();
```

```
// OK, function
```

Output:

```
"OK, function"
```

```
"OK, new"
```

```
"OK, new"
```

```
"OK, function"
```

```
"OK, new"
```

```
"OK, new"
```

```
"OK, function"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

143

```
"OK, function"
```

```
"OK, new"
```

```
"OK, function"
```

Method 2: Using `new.target`

From ES6 we can use the `new.target`.

As per MDN –

The `new.target` pseudo-property lets you detect whether a function or constructor was called using the `new` operator. In constructors and

functions invoked using the new operator, new.target returns a reference to the constructor or function. In normal function calls, new.target is undefined.

```
function
```

```
  A
```

```
  (
```

```
    B
```

```
  ) {
```

```
    if
```

```
    (
```

```
      new
```

```
        .target ) {
```

```
          console
```

```
            .log(
```

```
              'OK, new'
```

```
            );
```

```
          }
```

```
        else
```

```
        {
```

```
          console
```



```
.log(  
'OK, function'  
);  
  
}  
  
}
```

Test Case

Input:

```
A();  
  
// Ok, function  
  
new  
  
A();  
  
// OK, new  
  
new  
  
A();  
  
// Ok, new  
  
A(  
  
4  
  
);  
  
// OK, function  
  
var
```

```
X =  
new  
A(  
4  
);  
// OK, new  
  
var  
Z =  
new  
A();  
// OK, new  
  
Z.lolol = A;  
Z.lolol();  
// OK, function  
  
var  
Y = A;  
Y();  
// OK, function
```

```
var  
  
y =  
  
new  
  
Y();  
  
// OK, new  
  
y.lolol = Y;  
  
y.lolol();  
  
// OK, function
```

Output:

"OK, function"

"OK, new"

"OK, new"

"OK, function"

"OK, new"

"OK, new"

"OK, function"

"OK, function"

"OK, new"

"OK, function"

145

Implement store class (hashSet) in JavaScript

Problem Statement -

Create a simple store class (hashSet) with set(key, value) , get(key) , & has(key) methods.

Example

```
const
```

```
store =
```

```
new
```

```
Store();
```

```
store.set(
```

```
'a'
```

```
,
```

```
10
```

```
);
```

```
store.set(
```

```
'b'
```

```
,
```

```
20
```

```
);
```

```
store.set(
```

```
'c'  
  
,  
  
30  
  
);  
  
store.get(  
  
'b'  
  
);  
  
// 20  
  
store.has(  
  
'c'  
  
);  
  
// true
```

Reading the problem statement we can derive that we have to create a function that will store the list of key-value pairs and will expose two methods, one to check if the key exists in the store and the second to get the value associated with the key, apart from these one more

method to store the key-values.

We can do this by simply creating a function with an object that will store the key-value and these methods.

```
const  
  
Store =  
  
function
```

```
(){  
  //store the data  
  
  this  
  
  .list = {};  
  
  //set the key-value in list  
  
  this  
  
  .set =  
  
  function  
  
  (  
  
  key, value  
  
  ){  
  
  this  
  
  .list[key] = value;  
  
  }  
  
  //get the value of the given key  
  
  this  
  
  .get =  
  
  function  
  
  (  
  
  key
```

```
) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

146

```
return
```

```
this
```

```
.list[key];
```

```
}
```

```
//check if key exists
```

```
this
```

```
.has =
```

```
function
```

```
(
```

```
key
```

```
) {
```

```
return
```

```
!!
```

```
this
```

```
.list[key];
```

```
}
```

```
}
```

## Test Case

Input:

```
const
```

```
store =
```

```
new
```

```
Store();
```

```
store.set(
```

```
'a'
```

```
,
```

```
10
```

```
);
```

```
store.set(
```

```
'b'
```

```
,
```

```
20
```

```
);
```

```
store.set(
```

```
'c'
```

```
,
```

```
30
```



```
);  
console  
.log(store.get(  
'b'  
));
```

```
console  
.log(store.has(  
'c'  
));
```

```
console  
.log(store.get(  
'd'  
));
```

```
console  
.log(store.has(  
'e'  
));
```

Output:

20

true

undefined

false

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

147

Create a toggle function

Problem Statement -

Create a toggle function that accepts a list of arguments and toggles each of them when invoked in a cycle.

Example

```
let
```

```
hello = toggle(
```

```
"hello"
```

```
);
```

```
hello()
```

```
// "hello";
```

```
hello()
```

```
// "hello";
```

```
let
```

```
onOff = toggle(
```

```
"on"
```

```
,
```

```
"off"  
  
);  
  
onOff()  
  
// "on"  
  
onOff()  
  
// "off"  
  
onOff()  
  
// "on"
```

The toggle function returns each value clockwise on each call of the function and the same can be done by returning a function from the

toggle function forming a closure over the values to track the cycle.

### Implementation

```
const  
  
toggle = (  
  
...list  
  
) => {  
  
  // to track the cycle  
  
  let  
  
  current =  
  
  -1  
  
  ;
```

```
const
length = list.length;

return

function

(){

//moves to next element, resets to 0 when current
> length

current = (current +

1

) % length;

return

list[current];

}

}
```

Test Case

```
const

hello = toggle(

"1"

,

"2"
```

```
);  
  
console  
.log(hello());  
  
// "1"  
  
console  
.log(hello());  
  
// "2"  
  
console  
.log(hello());  
  
// "1"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

148

Sampling function

Problem Statement -

Create a function that accepts a function as input and a count and executes that input function once for a given count of calls. Known as a sampling function.

Example

function

message

() {

```
console
.log(
"hello"
);
}

const
sample = sampler(message,
4
);

sample();
sample();
sample();
sample(); // "hello"
this will be executed
sample();
sample();
sample();
sample();

// "hello" this will be executed
```

The sampling function is different from [throttling](#) as throttling limits the execution of the function once in a given amount of time while

sampling limits the execution by executing function once in a given number of calls.

To create a sampling function we can create a closure that will track how many times the function has been called and once it reaches the count, execute the input function and reset the counter.

Implementation

```
function
```

```
sampler
```

```
(
```

```
fn, count, context
```

```
){
```

```
let
```

```
counter =
```

```
0
```

```
;
```

```
return
```

```
function
```

```
(
```

```
...args
```

```
){
```

149

```
// set the counters

let

lastArgs = args;

context =

this

?? context;

// invoke only when number of calls is equal
to the counts

if

(++counter !== count)

return

;

fn.apply(context, args);

counter =

0

;

};

}
```

Test Case



```
function
```

```
message
```

```
() {
```

```
  console
```

```
    .log(
```

```
      "hello"
```

```
    );
```

```
  }
```

```
const
```

```
sample = sampler(message,
```

```
  4
```

```
);
```

```
sample();
```

```
sample();
```

```
sample();
```

```
sample();
```

```
// hello
```

```
sample();
```

```
sample();
```

```
sample();
```

```
sample();
```

```
// hello
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

150

Make function sleep

Many programming languages have inbuilt functions available which can halt the function execution for the given amount of time.

- *C or PHP* : sleep(2)
- *JAVA* : Thread.sleep(2000)
- *Python* : time.sleep(2)
- *Go* : time.Sleep(2 \* time.Second)

Javascript does not have any inbuilt function for this, But thanks to the introduction of promises and async-await in ES2018, we can now

implement such a feature without any hassle and use it seamlessly.

Implementation

```
const
```

```
sleep = (
```

```
milliseconds
```

```
) => {
```

```
return
```

```
new
```

Promise

```
(  
  resolve  
=> setTimeout(resolve,  
  milliseconds))  
};
```

This will create a wrapper function which will resolve the promise after the given milliseconds.

We can use this to prevent function execution for a certain amount of time.

Test Case 1: Using .then()

```
sleep(  
  500  
).then(  
  ()  
=> {  
  //do stuff  
  console  
  .log(  
    'I run after 500 milliseconds'  
  );
```

```
});
```

Test Case 2: Using async...await

```
const
```

```
performAction =
```

```
async
```

```
() => {
```

```
  await
```

```
    sleep(
```

```
      2000
```

```
    );
```

```
  //do stuff
```

© JavaScript Interview Guide | learnersbucket.com

```
151
```

```
}
```

```
performAction();
```

This works well, however due to [how javascript works](#) this does not stop the entire program execution like it does in other programming

languages, instead it will just make our current function sleep.

© JavaScript Interview Guide | learnersbucket.com

```
152
```

Remove cycle from the object

Problem Statement -

Given an object with a cycle, remove the cycle or circular reference from it.

Example

Input:

const

List =

function

(

val

){

this

.next =

null

;

this

.val = val;

};

const

item1 =

new

```
List(  
  10  
);  
  
const  
item2 =  
  
new  
List(  
  20  
);  
  
const  
item3 =  
  
new  
List(  
  30  
);  
  
item1.next = item2;  
item2.next = item3;  
item3.next = item1;  
  
// this form a cycle, if you console.log this you will see a circular object,  
// like, item1 -> item2 -> item3 -> item1 -> so on.
```

Output:

```
// removes cycle
```

```
// item1 -> item2 -> item3
```

If you see the above example, we have created a list object, that accepts a value and pointer to the next item in the list, similar to a [linked list](#) , and using this we have created the circular object.

We have to create a function that will break this cycle, in this example to break the cycle we will have to delete the next pointer of the *item3* .

© JavaScript Interview Guide | learnersbucket.com

153

There are two places where this cycle removal can take place.

1. For normal use of Object.
2. While creating the JSON string.

Normal use

We can use [WeakSet](#) which is used to store only unique object references and detect if the given object was previously detected or not, if it was detected then delete it.

This cycle removal always takes in-place.

```
const
```

```
removeCycle = (
```

```
obj
```

```
) => {
```

```
//set store
```

```
const
set =
new
WeakSet
([obj]);
//recursively detects and deletes the object references
(
function
iterateObj
(
obj
) {
for
(
let
key
in
obj) {
// if the key is not present in prototype
chain
```



```
if
(obj.hasOwnProperty(key)) {
  if
  (
  typeof
  obj[key] ===
  'object'
  ){
    // if the set has object reference
    // then delete it
    if
    (set.has(obj[key])){
      delete
      obj[key];
    }
    else
    {
      //store the object reference
      set.add(obj[key]);
      //recursively iterate the next
```

objects

```
iterateObj(obj[key]);
```

```
}
```

```
}
```

```
}
```

```
}
```

```
})(obj);
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

154

Test Case

Input:

```
const
```

```
List =
```

```
function
```

```
(
```

```
val
```

```
){
```

```
this
```

```
.next =
```

```
null  
;  
this  
.val = val;  
};  
const  
item1 =  
new  
List(  
10  
);  
const  
item2 =  
new  
List(  
20  
);  
const  
item3 =  
new
```

```
List(  
  30  
);  
item1.next = item2;  
item2.next = item3;  
item3.next = item1;  
removeCycle(item1);  
console  
.log(item1);
```

Output:

```
/*  
{val: 10, next: {val: 20, next: {val: 30}}}  
*/
```

Using JSON.stringify while creating JSON

[JSON.stringify](#) accepts a replacer function that can be used to alter the value of the stringification process.

We can use the same function to detect and remove the cycle from the object.

```
const  
getCircularReplacer =  
()
```

```
=> {  
  
  //form a closure and use this  
  
  //weakset to monitor object reference.  
  
  const  
  
  seen =  
  
  new  
  
  WeakSet  
  
  ();  
  
  //return the replacer function  
  
  return  
  
  (  
  
    key, value  
  
  ) => {  
  
    if  
  
    (  
  
      typeof  
  
      value ===  
  
      'object'  
  
      && value !==  
  
      null
```

```
) {
```

```
if
```

```
(seen.has(value)) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

155

```
return
```

```
;
```

```
}
```

```
seen.add(value);
```

```
}
```

```
return
```

```
value;
```

```
};
```

```
};
```

Test Case

Input:

```
const
```

```
List =
```

```
function
```

```
(
```

```
val
){
this
.next =
null
;
this
.val = val;
};
const
item1 =
new
List(
10
);
const
item2 =
new
List(
20
```

```
);  
  
const  
item3 =  
  
new  
List(  
30  
  
);  
  
item1.next = item2;  
item2.next = item3;  
item3.next = item1;  
  
console  
.log(  
  
JSON  
.stringify(item1, getCircularReplacer()));
```

Output:

```
"{'next':{'next':{'val':30},'val':20},'val':10}"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

156

## Filter Multidimensional Array

### Problem Statement -



Given multiple dimensional arrays, create a filter function that takes a callback function as input and returns a new array with all elements that have passed the test implemented in the callback function.

Example

Input:

```
const
```

```
arr = [[
```

```
  1
```

```
, [
```

```
  2
```

```
, [
```

```
  3
```

```
,
```

```
'foo'
```

```
, {
```

```
'a'
```

```
:
```

```
  1
```

```
,
```

```
'b'
```

```
:
```

2

}],

'bar'

]);

const

filtered = filter(arr, (

e

) =>

typeof

e ===

'string'

);

console

.log(JSON.stringify(filtered));

Output:

[[[

"foo"

],

"bar"

]]'

To filter a multi-dimensional array, we will have to filter each sub-array recursively and merge their results.

```
const
filter = (
arr, test
) => {
// Store the output
const
result = [];
//iterate the array
for
(
let
a
of
arr) {
//if sub-array
if
(
Array
```

```
.isArray(a)) {  
  //recursively filter the sub-array  
  
  const  
  
  output = filter(a, test);  
  
  //store the result  
  
  result.push(output);  
  
  }  
  
  else  
  
  {  
  
    //if not an array  
  
    //test the element  
  
    //if it passes the test, store its result  
  
    if  
  
    (test(a)) {  
  
      result.push(a);  
  
    }  
  
  }  
  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

157

}

}

}

```
//return the result
```

```
return
```

```
result;
```

```
};
```

Test Case

Input:

```
const
```

```
arr = [[
```

```
1
```

```
, [
```

```
2
```

```
, [
```

```
3
```

```
,
```

```
"foo"
```

```
, { a:
```

```
1
```

```
, b:
```

```
2
```

```
}],
```

```
"bar"
```

```
];
```

```
const
```

```
filtered = filter(arr, (e) =>
```

```
typeof
```

```
e ===
```

```
"number"
```

```
);
```

```
console
```

```
.log(
```

```
JSON
```

```
.stringify(filtered));
```

Output:

```
[[
```

```
1
```

```
,[
```

```
2
```

```
,[
```

```
3
```

```
]]]]
```

We can extend this method and implement a multiFilter on the array's prototype itself that will take a callback function and filter the array based on the callback's result.

Array

.prototype.multiFilter =

function

[OceanofPDF.com](http://OceanofPDF.com)

```
(  
test  
) {  
  //original array;  
  const  
  originalArray =  
  this  
  ;  
  const  
  filter = (  
    arr, test  
  ) => {  
    // Store the output  
    const  
    result = [];  
    //iterate the array  
    for  
    (  
      let  
      a
```



```
of
arr) {
//if sub-array
if
(
Array
.isArray(a)) {
//recursively filter the sub-array
const
output = filter(a, test);
//store the result
result.push(output);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

158

```
}
```

```
else
```

```
{
```

```
//if not an array
```

```
//test the element
```

```
//if it passes the test, store its
```

```
result
if
(test(a)) {
result.push(a);
}
}
}

//return the result

return
result;
};

//filter and return

return
filter(originalArray, test);
};
```

Test Case

Input:

```
const
```

```
arr = [[
```

```
1
```

, [

2

, [

3

,

"foo"

, { a:

1

, b:

2

}]],

"bar"

]];

const

filtered = arr.multiFilter((

e

) =>

typeof

e ===

"number"

```
);  
console  
.log(  
JSON  
.stringify(filtered));
```

Output:

```
[[  
  1  
  ,  
  2  
  ,  
  3  
]]]
```

© JavaScript Interview Guide | learnersbucket.com

159

Count elements in nested array

Problem Statement -

Given a nested array and a callback function, count all the elements that pass the test in the callback and return the count.

This problem is similar to the above [multiDimensional array filter](#), [the](#)

only difference is rather than filtering the array we have to just return the count of elements in the array that passes the test.

Example

Input:

```
const
```

```
arr = [[
```

```
  1
```

```
, [
```

```
  2
```

```
, [
```

```
  3
```

```
,
```

```
  4
```

```
,
```

```
  "foo"
```

```
, { a:
```

```
  1
```

```
, b:
```

```
  2
```

```
  ]],
```

```
  "bar"
```

```
    ]];  
  
    const  
    count = countInArray(arr, (e) =>  
    typeof  
    e ===  
    "number"  
    );  
  
    console  
    .log(count);
```

Output:

4

All we have to do is create a closure with an inner function. The parent function will hold the variable which will be incrementing the count every time the array element passes the test in the callback function and the inner function will recursively search the nested array.

```
let  
  
countInArray =  
  
function  
  
(  
  
inputArr, test  
  
) {
```

```
//track the count

let

count =

0

;

const

search = (

arr, test

) => {

//iterate the array

for

(

let

a

of

arr) {

//if not an array

//test the element

//if it passes the test, store its result

if
```

```
(test(a)) {
```

```
count +=
```

```
1
```

```
;
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
160
```

```
}
```

```
//if sub-array
```

```
if
```

```
(
```

```
Array
```

```
.isArray(a)) {
```

```
//recursively filter the sub-array
```

```
search(a, test);
```

```
}
```

```
}
```

```
};
```

```
//search
```

```
search(inputArr, test);
```

```
//return
```



```
return
```

```
count;
```

```
};
```

Test Case

Input:

```
const
```

```
arr = [[
```

```
1
```

```
, [
```

```
2
```

```
, [
```

```
3
```

```
,
```

```
4
```

```
,
```

```
"foo"
```

```
, { a:
```

```
1
```

```
, b:
```

```
2
```

```
    }],  
    "bar"  
  ]];  
  
const  
count = countInArray(arr, (e) =>  
  typeof  
  e ===  
  "number"  
);  
  
console  
.log(count);
```

Output:

4

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

161

## Convert HEX color to RGB

### Problem Statement -

Implement a function that converts the HEXA color codes to RGB numbers.

### Example

Input:

"#ff33ff"

Output:

{

"r"

:

255

,

"g"

:

51

,

"b"

:

255

}

Hexadecimal uses 16 unique symbols, representing values as “0 – 9”

for values between 0 – 9 and “A – F” or “a – f” for values between “10 – 15”.

RGB format is a combination of three colors, red, green, and blue in the range of 0 – 255. A hex color code is the hexadecimal

representation of the RGB numbers.

There are multiple ways in which we can convert the HEXA color codes to RGB numbers.

Approach 1. Using slice() method

A HEXA color code '#ff33ff' starts with '#' followed by six

alpha-numeric characters ff,33,ff, two of them each representing, R, G, & B.

© JavaScript Interview Guide | learnersbucket.com

162

We can use the slice() to get the two numbers and then use the parseInt() method that accepts a radix value and converts the string to a number.

```
const
```

```
hex2rgb = (
```

```
  hex
```

```
) => {
```

```
  const
```

```
    r =
```

```
    parseInt
```

```
    (hex.slice(
```

```
      1
```

```
    ,
```

3

),

16

);

const

g =

parseInt

(hex.slice(

3

,

5

),

16

);

const

b =

parseInt

(hex.slice(

5

,

```
7
),
16
);
// return {r, g, b}

return
{ r, g, b };
}

console
.log(hex2rgb(
"#ff33ff"
));

/*
{
  "r": 255,
  "g": 51,
  "b": 255
}
*/
```

In case we are given a short form of Hexa code like #f3f , we will have to convert it to the original form.

```
//create full hex
```

```
const
```

```
fullHex = (
```

```
hex
```

```
) => {
```

```
let
```

```
r = hex.slice(
```

```
1
```

```
,
```

```
2
```

```
);
```

```
let
```

```
g = hex.slice(
```

```
2
```

```
,
```

```
3
```

```
);
```

```
let
```

```
b = hex.slice(
```

```
3
```

```
,  
4  
);  
r =  
parseInt  
(r+r,  
16  
);  
g =  
parseInt  
(g+g,  
16  
);  
b =  
parseInt  
(b+b,  
16  
);  
// return {r, g, b}  
return
```



```
{ r, g, b };
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

163

```
//convert hex to rgb
```

```
const
```

```
hex2rgb = (
```

```
hex
```

```
) => {
```

```
if
```

```
(hex.length ===
```

```
4
```

```
){
```

```
return
```

```
fullHex(hex);
```

```
}
```

```
const
```

```
r =
```

```
parseInt
```

```
(hex.slice(
```

1

,

3

),

16

);

const

g =

parseInt

(hex.slice(

3

,

5

),

16

);

const

b =

parseInt

(hex.slice(

5

,

7

),

16

);

// return {r, g, b}

return

{ r, g, b };

}

console

.log(hex2rgb(

"#f3f"

));

/\*

{

"r": 255,

"g": 51,

"b": 255

}

\*/

## Approach 2. Using regex

The [match\(\)](#) method of string accepts a regex and returns the array of matching elements.

We can then use this array to parse the HEXA values to RGB.

```
const
```

```
hex2rgb = (
```

```
hex
```

```
) => {
```

```
const
```

```
rgbChar = [
```

```
'r'
```

```
,
```

```
'g'
```

```
,
```

```
'b'
```

```
];
```

```
const
```

```
normal = hex.match(
```

```
/^#([0-9a-f]{2})([0-9a-f]{2})([0-9a-f]{2})$/i
```

```
);
```

```
if
(normal) {
return
normal.slice(
1
).reduce((
a, e, i
) => {
a[rgbChar[i]] =
parseInt
(e,
16
);
return
a;
}, {});
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

164

}

const

```
shorthand = hex.match(
/^#([0-9a-f])([0-9a-f])([0-9a-f])$/i
);
if
(shorthand) {
return
shorthand.slice(
1
).reduce((
a, e, i
) =>
{
a[rgbChar[i]] =
0x11
*
parseInt
(e,
16
);
return
```

```
a;  
  
}, {});  
  
}  
  
return  
  
null  
  
;  
  
}  
  
console  
  
.log(hex2rgb(  
  
"#ff33ff"  
  
));  
  
/*  
  
{  
  
"r": 255,  
  
"g": 51,  
  
"b": 255  
  
}  
  
*/  
  
console  
  
.log(hex2rgb(
```

```
"#f3f"  
  
));  
  
/*  
  
{  
  
"r": 255,  
  
"g": 51,  
  
"b": 255  
  
}  
  
*/
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

165

Convert RGB to HEX color

Problem Statement -

Implement a function that converts the RGB number to HEXA color codes.

Example

Input:

255

,

51



,

255

Output:

"#ff33ff"

RGB format is a combination of three colors, red, green, and blue in the range of 0 – 255. A hex color code is the hexadecimal

representation of the RGB numbers.

Hexadecimal uses 16 unique symbols, representing values as “0 – 9”

for value between 0 – 9 and “A – F” or “a – f” for value between “10 – 15”.

There are multiple ways in which we can convert a given number to a HEXA value, just pad the number with 0 if it is a single digit and append it with “#”.

Approach 1. Using toString() method

The toString() method accepts a radix value and converts the value to that.

const

componentToHex = (

c

) => {

const

hex = c.toString(

16

);

return

hex.length ==

1

?

"0"

+ hex : hex;

}

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

166

const

rgbToHex = (

r, g, b

) => {

return

"#"

+ componentToHex(r) + componentToHex(g)

+ componentToHex(b);

}

console

.log(rgbToHex(

255

,

51

,

255

));

///`#ff33ff`

Approach 2. Using [left shift](#) (<<) operator const

rgbToHex = (

r, g, b

) => {

return

`"#"`

+ ((

1

<<

24

) + (r <<

```
16
) + (g <<
8
) +
b).toString(
16
).slice(
1
);
}
console
.log(rgbToHex(
255
,
51
,
255
));
//"#ff33ff"
```

Approach 3. Using Array.map()

```
const
rgbToHex = (
  r, g, b
) =>
'#'
+ [r, g, b]
.map(
  x
=> x.toString(
    16
  ).padStart(
    2
    ,
    '0'
  )).join(
  ''
)
console
.log(rgbToHex(
  255
```

,

51

,

255

));

//"#ff33ff"

© JavaScript Interview Guide | learnersbucket.com

167

In-memory filesystem library

Problem Statement -

Implement a simple in-memory filesystem library that supports the following functionalities.

- *createDirectory(name)* – Creates a new directory at the current path.
- *changeDirectory(path)* – Changes the directory path.
- *addFile(name)* – Adds a new file at the current path.
- *deleteFile(name)* – Deletes the file with the given name at the current path.
- *deleteDirectory(name)* – Deletes the directory with the given name at the given path.
- *getRootDirectory* – Returns the root directory and all its nested childs.
- *getCurDirectory* – Returns the items of the current directory.

- *getCurDirectoryPath* – Returns the path of the current directory.

To implement this function we will be extensively using [Objects](#), working with objects is a complex task especially if you are not thoroughly aware of how objects as a reference can be used effectively.

Boilerplate

```
const
FileSystem =
function
(){
this
.directory = {
"root"
: {}
};
this
.currentDir =
this
.directory[
"root"
];
```

this

.currentDirPath =

"root"

;

};

© JavaScript Interview Guide | learnersbucket.com

168

Here we have defined 3 variables

- *directory* – Holds the root directory.
- *currentDir* – Holds the reference to the root directory.
- *currentDirPath* – Holds the path of the current directory.

Create a new directory

A new directory is just a new object added to the given path.

this

.createDirectory =

function

(

name

){

this



```
.currentDir[name] = {};  
  
}
```

Change path of the directory

We accept the directory path as an absolute value separated by hyphen path-subpath-subpath . Once the directory changes, we also update the currentDir object reference. This way when you create a new directory or add a new file it will be added to the currentDir only.

Note – we are not checking if the directory exists or not.

this

```
.changeDirectory =
```

```
function
```

```
(
```

```
path
```

```
) {
```

```
this
```

```
.currentDir =
```

```
this
```

```
._changeDirectoryHelper(path);
```

```
this
```

```
.currentDirPath = path;
```

```
}
```

```
this

._changeDirectoryHelper =
function
(
path
) {
const
paths = path.split(
"_"
);
let
current =
this
.directory;
for
(
let
key
of
paths){
```

```
current = current[key];  
}  
  
return  
  
current;  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

169

Get the current directory

Returns the current directory object.

```
this  
  
.getCurDirectoryPath =  
  
function  
  
(){  
  
return  
  
this  
  
.currentDirPath;  
  
}
```

Get the current directory path

```
this  
  
.getCurDirectory =
```

```
function  
  
(){  
  
return  
  
this  
  
.currentDir;  
  
}
```

Add file

At the current path the files will be added to the key “files” as an array making it easy to add and remove them.

```
this  
  
.addFile =  
  
function  
  
(  
  
fileName  
  
){  
  
if  
  
(  
  
this  
  
.currentDir.files){  
  
this  
  
.currentDir.files.push(fileName);
```

```
}  
else  
{  
  this  
    .currentDir[  
      "files"  
    ] = [fileName];  
}  
return  
true  
;  
}
```

## Remove file

To remove files simply filter the files array at the current directory.

```
this  
  .deleteFile =  
  function  
  (  
    fileName  
  ){
```

```
this
.currentDir.files =
this
.currentDir.files.filter((
e
)
=> e !==
fileName);
return
true
;
}
```

Delete the directory

To delete the directory, change to the current path and delete any of the sub-directory.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

170

```
this
.deleteDirectory =
function
(
```

name

) {

delete

this

.currentDir[name];

}

Get the root directory

Returns the root object.

this

.getRootDirectory =

function

() {

return

this

.directory;

}

Complete code

const

FileSystem =

function

```
(){  
  this  
  .directory = {  
    "root"  
    : {}};  
  this  
  .currentDir =  
  this  
  .directory[  
    "root"  
  ];  
  this  
  .currentDirPath =  
  "root"  
  ;  
  this  
  .createDirectory =  
  function  
  (  
    name
```



```
) {  
  this  
    .currentDir[name] = {};  
}  
this  
  .changeDirectory =  
    function  
      (  
        path  
      ) {  
        this  
          .currentDir =  
            this  
              ._changeDirectoryHelper(path);  
        this  
          .currentDirPath = path;  
      }  
    this  
      ._changeDirectoryHelper =  
        function
```

```
(  
  path  
) {  
  const  
    paths = path.split(  
      "_"  
    );  
  let  
    current =  
      this  
        .directory;  
  for  
    (  
      let  
        key  
      of  
        paths){  
        current = current[key];  
      }  
  return
```

```
current;
}
this
.getCurDirectoryPath =
function
(){
return
this
.currentDirPath;
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

171

```
this
.getCurDirectory =
function
(){
return
this
.currentDir;
}
```

```
this

.addFile =

function

(

fileName

){

if

(

this

.currentDir.files){

this

.currentDir.files.push(fileName);

}

else

{

this

.currentDir[

"files"

] = [fileName];

}
```

```
return
true
;
}
this
.deleteFile =
function
(
fileName
){
this
.currentDir.files =
this
.currentDir.files.filter((
e
)
=> e !==
fileName);
return
true
```

```
;
}
this
.deleteDirectory =
function
(
name
){
delete
this
.currentDir[name];
}
this
.getRootDirectory =
function
(){
return
this
.directory;
}
```

```
}
```

Test Case

Input:

```
const
```

```
dir =
```

```
new
```

```
FileSystem();
```

```
dir.createDirectory(
```

```
'prashant'
```

```
);
```

```
dir.changeDirectory(
```

```
'root-prashant'
```

```
);
```

```
dir.addFile(
```

```
'index.html'
```

```
);
```

```
dir.addFile(
```

```
'app.js'
```

```
);
```

```
dir.changeDirectory(
```

```
'root'  
  
);  
  
dir.createDirectory(  
  
  'practice'  
  
);  
  
dir.changeDirectory(  
  
  'root-practice'  
  
);  
  
dir.addFile(  
  
  'index.html'  
  
);  
  
dir.addFile(  
  
  'app.js'  
  
);  
  
dir.createDirectory(  
  
  'build'  
  
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
dir.changeDirectory(  
  
  'root-practice'  
  
);
```



```
'root-practice-build'

);

dir.addFile(

'a.png'

);

dir.addFile(

'b.jpg'

);

dir.deleteFile(

'a.png'

);

dir.changeDirectory(

'root'

);

dir.deleteDirectory(

'prashant'

);

console

.log(dir.getRootDirectory());
```

Output:

```
{  
  "root"  
  : {  
    "practice"  
    : {  
      "files"  
      : [  
        "index.html"  
      ,  
        "app.js"  
      ],  
      "build"  
      : {  
        "files"  
        : [  
          "b.jpg"  
        ]  
      }  
    }  
  }  
}
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

173

Create a basic implementation of a streams

API

Problem Statement -

Create a basic implementation of a streams API. The user should be able to create multiple subscriptions and whenever a value is pushed to the stream, all the subscriptions should run.

Example

Input:

```
const
```

```
z =
```

```
new
```

```
Stream();
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value));
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value *
```

```
2
```

```
));
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value *
```

```
3
```

```
));
```

```
z.push(
```

```
2
```

```
);
```

```
Output:
```

```
2
```

```
4
```

```
6
```

We have to create a basic implementation of a streams API.

[Streams API](#) passes the data in chunks which means as we keep pushing values it should go through all the subscriptions.

To create this all we have to do is cache the subscriptions and whenever a value is pushed, call all the subscription methods with this value.

We will see two different implementations of these.

© JavaScript Interview Guide | learnersbucket.com

174

Class-based implementation

In the constructor we will initialize the cache and then create two methods inside the class, subscribe and push .

subscribe will cache all the methods passed to it and in the push method we will call all the subscription methods with the value received as an argument.

class

Stream

{

constructor

() {

this

```
.subscriptions = [];  
  
}  
  
subscribe(method){  
  
  if  
  
  (  
  
    typeof  
  
    method !==  
  
    'function'  
  
  ){  
  
    throw  
  
    new  
  
    Error  
  
    (  
  
      'Invalid method!.'  
  
    );  
  
  }  
  
  this  
  
  .subscriptions.push(method);  
  
}  
  
push(val){
```

```
this
.subscriptions.forEach((
method
) => {
method.call(
this
, val);
});
}
}
```

Test Case

Input:

```
const
z =
new
Stream();
z.subscribe((
value
) =>
console
```

```
.log(value));
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value *
```

```
2
```

```
));
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value *
```

```
3
```

```
));
```

```
z.push(
```

```
2
```

```
);
```

Output:

```
2
```



4

6

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

175

Function-based implementation

function

Stream

() {

let

subscriptions = [];

this

.subscribe = (

method

) => {

if

(

typeof

method !==

'function'

){

throw

new

Error

(

'Invalid method!.'

);

}

subscriptions.push(method);

}

this

.push = (

val

) => {

subscriptions.forEach((

method

) => {

method.call(

this

, val);

});

```
}
```

```
}
```

Test Case

Input:

```
const
```

```
z =
```

```
new
```

```
Stream();
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value));
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value *
```

```
2
```

```
));
```

```
z.subscribe((
```

```
value
```

```
) =>
```

```
console
```

```
.log(value *
```

```
3
```

```
));
```

```
z.push(
```

```
2
```

```
);
```

Output:

```
2
```

```
4
```

```
6
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

176

Create a memoizer function

Problem Statement -

Create a function that memorizes or caches the result for the given input so that the subsequent calls for the same inputs will be faster.

## Example

```
slowFunc(params)
```

```
// normal call, slow output
```

```
const
```

```
memoized = memoize(slowFunc);
```

```
memoized(params)
```

```
//first call, extremely slow -> caches
```

```
the result
```

```
memoized(params)
```

```
//second call, very fast.
```

```
//all the subsequents call for the same input will be faster.
```

```
memoized(differentParams)
```

```
//first call, extremely
```

```
slow -> caches the result
```

```
memoized(differentParams)
```

```
//second call, very fast.
```

[Memoization](#) is an optimization technique prominently used in computer science to avoid recomputation in intensive computation,

we cache the result for the given input and return it for subsequent call of the same input rather than performing the expensive computation

again.

We will create a closure with the higher-order function that will cache the result for the given input. If a call is made and the result for the input is stored in the cache then we will return it, otherwise, execute the function and cache its result.

```
const
```

```
memoize =
```

```
function
```

```
(
```

```
fn
```

```
) {
```

```
  const
```

```
  cache = {};
```

```
  return
```

```
  function
```

```
  () {
```

```
    //arg as key to store the result
```

```
    const
```

```
    KEY =
```

```
    JSON
```

```
    .stringify(
```

```
    arguments
```

```
);
```

```
//if the result is present for the given key return
```

```
it
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
177
```

```
if
```

```
(cache[KEY]) {
```

```
return
```

```
cache[KEY]
```

```
}
```

```
//else compute and store the result and return
```

```
the result
```

```
const
```

```
evaluatedValue = fn(...arguments);
```

```
cache[KEY] = evaluatedValue;
```

```
return
```

```
evaluatedValue;
```

```
}
```

```
};
```

```
Test Case
```

Input:

function

factorial

(

n

) {

if

(n ==

0

|| n ==

1

) {

return

1

}

return

factorial(n

-1

) \* n;

};



```
const
memoizedFactorial = memoize(factorial);

let
a = memoizedFactorial(
100
)
// first call, slow

console
.log(a);

let
b = memoizedFactorial(
100
)
// cached call, faster

console
.log(b);

Output:
9.33262154439441e+157

// slow
9.33262154439441e+157
```

// faster

© JavaScript Interview Guide | learnersbucket.com

178

## Method chaining - Part 1

### Problem Statement -

Explain method chaining in JavaScript by implementing a calculator that performs the basic actions like add, subtract, divide, and multiply.

### Example

```
calculator.add(  
  10  
)  
.subtract(  
  2  
)  
.divide(  
  2  
)  
.multiply(  
  5  
)  
;  
console  
.log(calculator.total);  
//20
```

Method chaining is an object-oriented paradigm, in which the methods usually share the same reference, which in JavaScript is done by sharing [this](#) (current context) from each method.

We are going to see the two different implementations of method chaining.

1. Using objects.
2. With functions.

#### Method 1: Using objects

Methods inside the object can refer to the current object using this keyword, thus we can use the same to perform our operations and return them so that they can be shared around the chain.

```
const  
  
calculator = {  
  
  total:  
  
  0  
  
  ,  
  
  add:  
  
  function  
  
  (  
  
  val  
  
  ){
```

```
this  
  
.total += val;  
  
return  
  
this  
  
;  
  
},  
  
subtract:  
  
function  
  
(  
  
val  
  
) {  
  
this  
  
.total -= val;  
  
return  
  
this  
  
;  

```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

179

},

divide:

```
function
(
val
){
this
.total /= val;
return
this
;
},
multiply:
function
(
val
){
this
.total *= val;
return
this
;
}
```

```
}
```

```
};
```

Test Case

Input:

```
calculator.add(
```

```
10
```

```
).subtract(
```

```
2
```

```
).divide(
```

```
2
```

```
).multiply(
```

```
5
```

```
);
```

```
console
```

```
.log(calculator.total);
```

Output:

```
20
```

Method 2: With functions

The problem with objects is that we cannot create a new instance of them. But it can be solved using functions.

We just have to return this keyword from each method, so that next chaining can be done on the current function.

```
const
```

```
CALC =
```

```
function
```

```
() {
```

```
  this
```

```
    .total =
```

```
    0
```

```
  ;
```

```
  this
```

```
    .add = (
```

```
    val
```

```
  ) => {
```

```
    this
```

```
      .total += val;
```

```
    return
```

```
      this
```

```
    ;
```

```
  }
```

```
  this
```

```
.subtract = (  
  val  
) => {  
  this  
    .total -= val;  
  return  
    this  
  ;  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

180

```
this  
  
.multiply = (  
  val  
) => {  
  this  
    .total *= val;  
  return  
    this  
  ;
```



```
}  
  
this  
  
.divide = (  
  val  
  
) => {  
  
  this  
  
  .total /= val;  
  
  return  
  
  this  
  
  ;  
  
}  
  
this  
  
.value =  
  
()  
  
=>  
  
this  
  
.total;  
  
}
```

Test Case

Input:

```
const  
calculator =  
new  
CALC();  
calculator.add(  
10  
)  
.subtract(  
2  
)  
.divide(  
2  
)  
.multiply(  
5  
)  
;  
console  
.log(calculator.total);
```

Output:

20

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

181

Method chaining - Part 2

Problem Statement -

Showcase a working demo of method chaining in JavaScript by implementing the following example.

Example

Input:

```
computeAmount().lacs(
```

```
15
```

```
).crore(
```

```
5
```

```
).crore(
```

```
2
```

```
).lacs(
```

```
20
```

```
).thousand(
```

```
45
```

```
).crore(
```

```
7
```

```
).value
```

```
());
```

Output:

143545000

In the previous problem we have already seen an example of method chaining in which we used object-based and function-based implementations.

Similarly we will solve this problem with different approaches as well.

Method 1: Using function as constructor

We will create a constructor function that will help us to create a new instance every time and perform the operations.

```
const
```

```
ComputeAmount =
```

```
function
```

```
() {
```

```
  this
```

```
    .store =
```

```
    0
```

```
  ;
```

```
  this
```

```
    .crore =
```

```
    function
```

```
    (
```

```
      val
```

```
) {  
  this  
    .store += val *  
    Math  
      .pow(  
        10  
      ,  
        7  
      );  
  return  
    this  
      ;  
};  
this  
  .lacs =  
    function  
      (  
        val  
      ) {  
        this
```

```
.store += val *
```

```
Math
```

```
.pow(
```

```
10
```

```
,
```

```
5
```

```
);
```

```
return
```

```
this
```

```
;
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
182
```

```
this
```

```
.thousand =
```

```
function
```

```
(
```

```
val
```

```
){
```

```
this
```

```
.store += val *
```

```
Math
```

```
.pow(
```

```
10
```

```
,
```

```
3
```

```
);
```

```
return
```

```
this
```

```
;
```

```
}
```

```
this
```

```
.hundred =
```

```
function
```

```
(
```

```
val
```

```
){
```

```
this
```

```
.store += val *
```

```
Math
```

```
.pow(  
  10  
  
  ,  
  2  
);  
return  
this  
;  
}  
this  
.ten =  
function  
(  
  val  
) {  
  this  
.store += val *  
  10  
;  
return
```



this

;

}

this

.unit =

function

(

val

){

this

.store += val;

return

this

;

}

this

.value =

function

() {

return

this

.store;

}

}

Test Case

Input:

const

computeAmount =

new

ComputeAmount();

const

amount =

computeAmount.lacs(

15

).crore(

5

).crore(

2

).lacs(

20

```
).thousand(  
45  
).crore(  
7  
).val  
ue();  
console  
.log(amount ===  
143545000  
);
```

Output:

true

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

183

## Method 2: Using function as closure

We will form closure and return a new object from it that will have all the logic encapsulated. This way we won't have to create a constructor everytime and the data won't duplicate too.

const

ComputeAmount =

function

```
() {  
    return  
    {  
        store:  
        0  
        ,  
        crore:  
        function  
        (  
            val  
        ) {  
            this  
            .store += val *  
            Math  
            .pow(  
            10  
            ,  
            7  
            );  
            return
```

this

;

},

lacs:

function

(

val

) {

this

.store += val \*

Math

.pow(

10

,

5

);

return

this

;

},

thousand:

function

(

val

) {

this

.store += val \*

Math

.pow(

10

,

3

);

return

this

;

},

hundred:

function

(

```
val
){
this
.store += val *
Math
.pow(
10
,
2
);
return
this
;
},
ten:
function
(
val
){
this
```

```
.store += val *
```

```
10
```

```
;
```

```
return
```

```
this
```

```
;
```

```
},
```

```
unit:
```

```
function
```

```
(
```

```
val
```

```
){
```

```
this
```

```
.store += val;
```

```
return
```

```
this
```

```
;
```

```
},
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)



value:

function

() {

return

this

.store;

}

}

}

Test Case

Input:

const

amount =

ComputeAmount().lacs(

9

).lacs(

1

).thousand(

10

).ten(

1

).unit(

1

).value());

console

.log(amount ===

1010011

);

const

amount2 =

ComputeAmount().lacs(

15

).crore(

5

).crore(

2

).lacs(

20

).thousand(

45

```
).crore(  
7  
)  
.value()  
console  
.log(amount2 ===  
143545000  
);
```

Output:

true

true

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

185

Implement clearTimeout

Problem Statement -

Implement a ClearAllTimeout function that will stop all the running  
setTimeout at once.

Example

Input:

```
setTimeout(  

```

```
()  
=> {  
  console  
    .log(  
      "hello"  
    )},  
  2000  
);  
setTimeout(  
  ()  
    => {  
      console  
        .log(  
          "hello1"  
        )},  
      3000  
    );  
  setTimeout(  
    ()  
      => {
```

```
console
```

```
.log(
```

```
"hello2"
```

```
)),
```

```
4000
```

```
);
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
console
```

```
.log(
```

```
"hello3"
```

```
)),
```

```
5000
```

```
);
```

```
clearAllTimeout();
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
console
```

```
.log(  
"hello4"  
}),  
5000  
);
```

Output:

```
"hello4"
```

[setTimeout](#) is an asynchronous function that executes a function or a piece of code after a specified amount of time.

setTimeout method returns a unique Id when it is invoked, which can be used to cancel the timer anytime using the clearTimeout method which is provided by BOM (Browser Object Model).

Reading about the problem statement we can understand that all we have to do is to clear all the active timers and the same can be done by clearing all timeoutIds using clearTimeout .

```
window  
  
.clearAllTimeout =  
  
function  
  
(){  
  
//clear all timeouts  
  
while
```

```
(timeoutIds.length){  
  clearTimeout(timeoutIds.pop());  
}  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

186

But to clear all the timeoutIds at once, we will need to store them somewhere, let's say in an array. For which we will override the existing setTimeout method and collect all the timeoutIds in an array.

```
window  
  .timeoutIds = [];  
  
  // store the original method  
  
  const  
    originalTimeoutFn =  
    window  
      .setTimeout;  
  
  //over-writing the original method  
  
  window  
    .setTimeout =  
  
    function  
    (  

```

```
fn, delay
) {
  const
  id = originalTimeoutFn(fn, delay);
  timeoutIds.push(id);
  //return the id so that it can be originally cleared
  return
  id;
}
```

Complete code

```
window
.timeoutIds = [];
// store the original method
const
originalTimeoutFn =
window
.setTimeout;
//over-writing the original method
window
.setTimeout =
```



```
function
(
fn, delay
) {
const
id = originalTimeoutFn(fn, delay);
timeoutIds.push(id);
//return the id so that it can be originally cleared
return
id;
}
window
.clearAllTimeout =
function
(){
//clear all timeouts
while
(timeoutIds.length){
clearTimeout(timeoutIds.pop());
}
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

187

Test Case

```
setTimeout(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      "hello"
```

```
    )},
```

```
  2000
```

```
);
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      "hello1"
```

```
    )},
```

3000

);

setTimeout(

()

=> {

console

.log(

"hello2"

}),

4000

);

setTimeout(

()

=> {

console

.log(

"hello3"

}),

5000

);

```
clearAllTimeout();
```

If we test this, this runs as expected. It will clear all the timeouts, as `setTimeout` is an Asynchronous function, meaning that the timer

function will not pause execution of other functions in the functions stack, thus `clearAllTimeout` runs and cancels them before they can be executed.

If you notice, here we have added a global variable `timeoutIds` which we are using to store the ids of each `setTimeout` and later to cancel all of them, using the global variable is bad practice as it can be

overridden.

One thing we could do over here is to wrap these inside a closure or higher-order function or an Object to keep it restricted.

This way we won't be interfering with existing methods and can still get our work done.

```
const
```

```
MY_TIMER = {
```

```
  timeoutIds : [],
```

```
  //global timeout id arrays
```

```
  //create a MY_TIMER's timeout
```

```
  setTimeout :
```

```
  function
```

```
  (
```

```
    fn,delay
```

```
  ){
```

```
let  
id = setTimeout(fn,delay);  
this  
.timeoutIds.push(id);  
return  
id;  
},
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

188

```
//MY_TIMER's clearAllTimeout
```

```
clearAllTimeout :  
function  
{  
while  
(  
this  
.timeoutIds.length){  
clearTimeout(  
this  
.timeoutIds.pop());
```

```
}
```

```
}
```

```
};
```

Test Case

Input:

```
const
```

```
id = MY_TIMER.setTimeout(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      "hello"
```

```
    )},
```

```
    1000
```

```
  );
```

```
  console
```

```
    .log(id);
```

```
  MY_TIMER.clearAllTimeout();
```

Output:

```
13
```

```
//timeoutId
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

189

## Implement ClearAllInterval

### Problem Statement -

Implement a ClearAllInterval function that will stop all the running setInterval at once.

### Example

Input:

```
setInterval(  
  ()  
=> {  
  console  
    .log(  
      "Hello"  
    );  
  },  
  2000  
);  
setInterval(  
  ()  
=> {  
  console  
    .log(  
      "Hello"  
    );  
  },  
  2000  
);
```

```
=> {  
  console  
    .log(  
      "Hello2"  
    );  
  },  
  500  
);  
clearAllInterval(); // clears first two intervals  
setInterval(  
  ()  
=> {  
  console  
    .log(  
      "Hello3"  
    );  
  },  
  1000  
)
```

Output:



"Hello3"

// last one, after every ~1 sec

JavaScript has a timer function [setInterval](#) which repeatedly executes a function after a specified amount of time. Each setInterval method returns a unique id which can be used to cancel or clear the interval using clearInterval .

Similarly, as we have implemented the [clearAllTimeout](#) method, the same logic can be used to implement the clearAllInterval.

First, to clear all the intervals at once, we need to store all of their ids so that they can be cleared one by one using the clearInterval method.

© JavaScript Interview Guide | learnersbucket.com

190

Thus we can define a global variable intervalIds that will store the ids of the setIntervals and override the existing setInterval function to push the ids in this global variable.

//to store all the interval ids

window

.intervalIds = [];

//original interval function

const

originalIntervalFn =

window

.setInterval;

```
//overriding the original  
window  
.setInterval =  
function  
(  
fn, delay  
) {  
  const  
  id = originalIntervalFn(fn, delay);  
  //storing the id of each interval  
  intervalIds.push(id);  
  return  
  id;  
}  
  
//clear all interval  
window  
.clearAllInterval =  
function  
(  
){  
  while
```

```
(intervalIds.length){  
  clearInterval(intervalIds.pop());  
}  
}
```

Test Case

Input:

```
setInterval(  
  ()  
=> {  
  console  
    .log(  
      "Hello"  
    );  
  },  
  2000  
);  
setInterval(  
  ()  
=> {  
  console
```

```
.log(  
  "Hello2"  
);  
},  
500  
);  
clearAllInterval();  
setInterval(  
  ()  
=> {  
  console  
    .log(  
      "Hello3"  
    );
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
191
```

```
},
```

```
1000
```

```
)
```

Output:

```
"Hello3"
```

```
// after every ~1 sec
```

If you notice, here we have added a global variable `intervalIds` which we are using to store the ids of each `setInterval` and later to cancel all of them. Using the global variable is bad practice as it can be

overridden.

One thing you could do over here is to wrap these inside a closure or higher-order function or an Object to keep it restricted.

This way we won't be interfering with existing methods and can still get our work done.

```
const
```

```
MY_TIMER = {
```

```
  intervalIds : [],
```

```
  //global interval id's arrays
```

```
  //create a MY_TIMER's interval
```

```
  setInterval :
```

```
  function
```

```
  (
```

```
    fn,delay
```

```
  ){
```

```
    let
```

```
    id = setInterval(fn,delay);
```

```
this
.intervalIds.push(id);
return
id;
},
//MY_TIMER's clearAllTimeout
clearAllInterval :
function
(){
while
(
this
.intervalIds.length){
clearTimeout(
this
.intervalIds.pop());
}
}
};
```

Test Case

Input:

```
MY_TIMER.setInterval(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      "Hello"
```

```
    );
```

```
  },
```

```
  2000
```

```
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

192

```
MY_TIMER.setInterval(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      "Hello2"
```

```
    );
```

```
},  
  
500  
  
);  
  
MY_TIMER.clearAllInterval();  
  
MY_TIMER.setInterval(  
  
()  
  
=> {  
  
  console  
  
    .log(  
  
      "Hello3"  
  
    );  
  
  },  
  
  1000  
  
);
```

Output:

"Hello3"

// last one, after every ~1 sec

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

193

Create a fake setTimeout



## Problem Statement -

Create a fake setTimeout that should work somehow similar to the original setTimeout method.

## Example

```
MY_TIMER.setTimeout(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      1
```

```
    )
```

```
  },
```

```
  2500
```

```
);
```

```
MY_TIMER.setTimeout(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      2
```

```
)  
},  
2000  
);  
MY_TIMER.run();
```

Output:

```
2  
  
// will be printed after 2 seconds  
  
1  
  
// will be printed 500 milliseconds after the 1st
```

Well, there is no definite way to implement the custom setTimeout, but we can do a workaround.

Create a custom object that will have setTimeout , clearTimeout , and run method.

In the setTimeout , store each entry in the queue, for the delay, add the input to the current time to determine when it should be invoked. Also after each entry, sort the queue in ascending order based on the time.

Return a unique id at the end.

Using the timer id, we can remove the entry from the queue in clearTimeout .

In the run method, we will run an infinite while loop, in each iteration, we will get the first element from the queue (as it will be with the least time), check if its time is past the current time then invoke it, else push it back into the queue.

Do it for all the entries in the queue. Add a condition to check if there are no more timers (the queue is empty) to run then break the loop.

```
const
```

```
MY_TIMER = {
```

```
  timerId:
```

```
    1
```

```
  ,
```

```
  queue: [],
```

```
  // create a new timer
```

```
  setTimeout:
```

```
    function
```

```
    (
```

```
      cb, time, ...args
```

```
    ){
```

```
      const
```

```
      id =
```

```
      this
```

```
.timerId++;

// add a new entry to the queue

// the time at which it will run

// will be added to the current date

// so that it will run next

this

.queue.push({

id,

cb,

time:

Date

.now() + time,

args,

});

// sort the queue in the ascending order of time

this

.queue.sort((

a, b

) => a.time - b.time);

// return the id
```

```
return  
  
id;  
  
},  
  
// to stop the timer  
  
clearTimeout:  
  
function  
  
(  
  
removeId  
  
) {  
  
// remove the entry with the given id  
  
this  
  
.queue =  
  
this  
  
.queue.filter((  
  
{ id }  
  
) => id  
  
!== removeId);  
  
},  
  
// start running the timer
```

195

run:

function

() {

// this will continuously run the loop

// till all the entry in the queue are invoked

while

(

true

) {

const

entry =

this

.queue.shift();

const

{ cb, time, args } = entry;

// if time has passed

// invoke it

if

(

Date

```
.now() > time){
```

```
  cb(...args);
```

```
}
```

```
// else push it back into the queue
```

```
else
```

```
{
```

```
  this
```

```
  .queue.unshift(entry);
```

```
}
```

```
// if there are no further entries
```

```
// break the loop
```

```
if
```

```
(
```

```
  this
```

```
  .queue.length ===
```

```
  0
```

```
){
```

```
  break
```

```
;
```

```
}
```

```
}
```

```
}
```

```
};
```

Test Case

Input:

```
MY_TIMER.setTimeout(
```

```
()
```

```
=> {
```

```
  console
```

```
    .log(
```

```
      1
```

```
    )
```

```
  },
```

```
  2500
```

```
);
```

```
MY_TIMER.setTimeout(
```

```
()
```

```
=> {
```

```
  console
```



```
.log(  
2  
)  
},  
2000  
);
```

```
MY_TIMER.setTimeout(  
()  
=> {
```

```
console
```

```
.log(  
3  
)  
},  
2500  
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

196

```
MY_TIMER.setTimeout(  
()  

```

```
=> {  
  console  
    .log(  
      4  
    )  
  },  
  3000  
);  
MY_TIMER.run();
```

Output:

```
2  
1  
3  
4
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

197

### Currying - Problem 1

What is currying?

Currying in JavaScript is a concept of functional programming in which we can pass functions as arguments (callbacks) and return

functions without any side effects (changes to program states).

In simple terms, in currying we return a function for each function invoked which accepts the next argument inline. With the help of currying we can transform a function with multiple arguments into a sequence of nesting functions.

Example

```
//normal function
```

```
sum(
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
)
```

```
//should return 6
```

```
//currying function
```

```
sum(
```

```
1
```

```
)(
```

```
2
```

```
)
```

```
3
```

```
)
```

```
//should return 6
```

If you notice in the currying function for each function call `sum(1)` we are returning a function which accepts the next argument `sum(1)(2)`

and it again returns a function which accepts argument `sum(1)(2)(3)` and so on.

There is no specified limit to how many times you return a function, also there are different variations to currying like the first function will accept the 2 arguments and the next function can accept any number

of arguments and so.

```
//variations of currying
```

```
sum(
```

```
1
```

```
)
```

```
2
```

```
)
```

```
3
```

```
)
```

```
sum(
```

```
1
```

,

2

)(

3

)

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

198

sum(

1

)(

2

,

3

)

sum(

1

,

2

,

3

)

//all of them should return same output

//6

Now you may be wondering that each function call returns a new

function then how the value is returned from it?. Well for that we have to decide a base condition that should return the output.

For example, if no argument is passed in the next function call then return the value or if we have reached 5 arguments then return the

value, etc.

sum(

1

)(

2

)(

3

)()

sum(

1

,

2

)(

3

)()

sum(

1

)(

2

,

3

)()

sum(

1

,

2

,

3

)()

//all of these should return 6

//OR

//when we reach 5 arguments then return the value rather than new function

sum(

1

,

2

,

3

,

4

,

5

)

sum(

1

,

2

)(

3

,

4

,

5

)



sum(

1

)(

2

,

3

,

4

,

5

)

sum(

1

,

2

,

3

)(

4

,

5

)

sum(

1

)(

2

)(

3

)(

4

)(

5

)

sum(

1

,

2

,

3

,

4

)(

5

)

//all should return 15

Problem Statement -

Implement a currying function for 4 arguments. When we have reached the limit, return the value.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

199

Example

sum(

1

,

2

,

3

,

4

)

sum(

1

)(

2

)(

3

)(

4

)

sum(

1

,

2

)(

3

,

4

)

sum(

1

```
,  
2  
  
,  
3  
  
)(  
4  
  
)  
sum(  
1  
  
)(  
2  
  
,  
3  
  
,  
4  
  
)
```

//all should return 10

First, let's handle the base case.

When the function is invoked in normal style `sum(1, 2, 3, 4)` . In this all we have to do is check the number of arguments passed, if it is the

same as the limit provided return the sum of them.

```
const
sum = (
...args
) => {
//spread the arguments in storage array
const
storage = [...args];
//base case
if
(storage.length ===
4
){
return
storage.reduce((
a, b
) => a + b,
0
);
}
}
```

`...args` is the rest operator which aggregates all the passed arguments as an array.

Once we have the array of arguments we have stored that in a variable inside the function `storage = [...args]` .

The purpose of using the variable is that, when the arguments passed is less than the limit then we will use this further to store the next argument in the closure.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

200

Now if the length of the storage is 4 which means we have 4 arguments so return their sum.

Otherwise we have to return a new function every time until we have reached the limit.

```
const
```

```
sum = (
```

```
...args
```

```
) => {
```

```
//spread the arguments in storage array
```

```
const
```

```
storage = [...args];
```

```
//base case
```

```
//if we have reached the limit
```

```
if
(storage.length ===
4
){
return
storage.reduce((
a, b
) => a + b,
0
);
}
//otherwise return a function
else
{
//create an inner function
const
temp =
function
(
...args2
```



```
) {  
  //get the arguments of inner function  
  //merge them in existing storage  
  storage.push(...args2);  
  //if we have reached the limit  
  //return the value  
  if  
    (storage.length ===  
     4  
    ) {  
    return  
    storage.reduce((  
      a, b  
    ) => a + b,  
    0  
    );  
  }  
  //else return the same function again  
  else  
  {
```

```
return  
  
temp;  
  
}  
  
}  
  
//return the function  
  
return  
  
temp;  
  
}  
  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

201

We merge the arguments of inner function in the existing storage and check if we have reached the limit or not in each call.

If we have reached the limit return the sum of them otherwise return the same function again.

Test Case

Input:

```
const  
  
res = sum(  
  
1  
  
,
```

2

,

3

,

4

);

const

res2 = sum(

1

)(

2

)(

3

)(

4

);

const

res3 = sum(

1

,

2

)(

3

,

4

);

const

res4 = sum(

1

,

2

,

3

)(

4

);

const

res5 = sum(

1

)(

2

,

3

,

4

);

console

.log(res, res2, res3, res4, res5);

Output:

10

10

10

10

10

Special Case: Return the value when invoked with empty

arguments

For the last input when no argument is passed it should return 0.

The solution for this is quite straightforward, all we have to do is update the condition and instead of checking the number of

arguments available in the storage, we have to check if there is any argument passed or not. If it is not passed then return the sum of

arguments we have in storage.

```
const
```

```
sum = (
```

```
...args
```

```
) => {
```

```
//spread the arguments in storage array
```

```
const
```

```
storage = [...args];
```

```
//base case
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

202

```
//if invoked without any argument
```

```
if
```

```
(storage.length ===
```

```
0
```

```
){
```

```
return
```

```
0
```

```
;
```

```
}
```

```
//closure

else

{

//create an inner function

const

temp =

function

(

...args2

){

//get the arguments of inner function

//merge them in existing storage

storage.push(...args2);

//if no arguments are passed

//return the value

if

(args2.length ===

0

){

return
```

```
storage.reduce((  
  a, b  
  ) => a + b,  
  0  
);  
  
}  
  
//else return the same function again  
  
else  
  
{  
  
  return  
  
  temp;  
  
}  
  
}  
  
//return the function  
  
return  
  
temp;  
  
}  
  
}
```

Test Case

Input:



```
const
```

```
res = sum(
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
);
```

```
const
```

```
res2 = sum(
```

```
1
```

```
)(
```

```
2
```

```
)(
```

```
3
```

```
)(
```

```
4
```

```
);
```

```
const
```

```
res3 = sum(
```

```
1
```

```
,
```

```
2
```

```
)(
```

```
3
```

```
,
```

```
4
```

```
);
```

```
const
```

```
res4 = sum(
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
)(
```

```
4
```

```
);
```

```
const
```

```
res5 = sum(
```

```
1
```

```
)(
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
);
```

```
const
```

```
res6 = sum();
```

```
console
```

```
.log(res, res2, res3, res4, res5, res6);
```

Output:

```
10
```

```
10
```

```
10
```

```
10
```

```
10
```

0

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

203

Currying - Problem 2

Problem Statement -

Create a javascript function that will remember its previously passed values and return the sum of the current and previous value.

Example

```
sum(
```

```
5
```

```
);
```

```
// 5
```

```
sum(
```

```
3
```

```
);
```

```
// 8
```

```
sum(
```

```
4
```

```
);
```

```
// 12
```

```
sum(
```

```
0
```

```
);
```

```
// 12
```

Javascript functions have access to the state (properties & methods) of its parent function even after it is executed.

So to create a function that will return the sum of the previous values in javascript we will use this technique of closure.

```
const
```

```
curry =
```

```
()
```

```
=> {
```

```
//To store the previous values
```

```
let
```

```
sum =
```

```
0
```

```
;
```

```
//Return an inner function
```

```
//Which will have access to its parent function's
```

```
store
```

```
return
```

```
function
```

```
(  
  num = 0  
) {  
  sum += num;  
  return  
  sum;  
};  
};
```

Here we have created a function named curry which has a store to keep track of the previous value and returns an inner function.

The powerful feature of the javascript function is that the inner function still has access to the parent function's properties and methods even after it is returned.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

204

We now call this function which will return the inner function and we can use the returned inner function for our action.

Test Case

//Returns and stores the inner function.

let

sum = curry();

```
console
.log(sum(
5
));
//5
```

```
console
.log(sum(
3
));
//8
```

```
console
.log(sum(
4
));
//12
```

```
console
.log(sum(
0
));
//12
```

console

.log(sum());

//12

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

205

## Currying - Problem 3

### Problem Statement -

Write a function that satisfies the following.

add(

1

)(

2

).value() =

3

;

add(

1

,

2

)(



3

).value() =

6

;

add(

1

)(

2

)(

3

).value() =

6

;

add(

1

)(

2

) +

3

=

6;

This is a little tricky question and requires us to use and modify the [valueOf\(\)](#) method.

When JavaScript wants to turn an object into a primitive value, it uses the valueOf() method. JavaScript automatically calls the valueOf() method when it comes across an object where a primitive value is

anticipated, so you don't ever need to do it yourself.

Example

function

MyNumberType

(

number

) {

this

.number = number;

}

MyNumberType.prototype.valueOf =

function

() {

return

this

```
.number +  
1  
;  
};  
  
const  
myObj =  
new  
MyNumberType(  
4  
);  
  
myObj +  
3  
;  
  
// 8
```

Thus we can form a closure and track the arguments in an Array and return a new function everytime that will accept new arguments.

We will also override the `valueOf()` method and return the sum of all the arguments for each primitive action, also add a new method

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

value() that will reference the valueOf() thus when invoked will return the sum of arguments.

```
function
```

```
add
```

```
(
```

```
...current
```

```
){
```

```
// store the current arguments
```

```
let
```

```
sum = current;
```

```
function
```

```
resultFn
```

```
(
```

```
...rest
```

```
){
```

```
// merge the new arguments
```

```
sum = [...sum, ...rest];
```

```
return
```

```
resultFn;
```

```
}
```

```
// override the valueOf to return sum
```

```
resultFn.valueOf =  
function  
{  
  return  
    sum.reduce((  
      acc, current  
    ) => acc + current,  
    0  
  );  
};  
  
// extend the valueOf  
resultFn.value = resultFn.valueOf;  
  
// return the inner function  
  
// on any primitive action .valueOf will be invoked  
  
// and it will return the value  
  
return  
resultFn;  
}
```

Test Case

Input:

```
console
.log(add(
1
)
2
).value() ==
3
);
```

```
console
.log(add(
1
,
2
)
3
).value() ==
6
);
```

```
console
.log(add(
```

1

)(

2

)(

3

).value() ==

6

);

console

.log(add(

1

)(

2

) +

3

);

Output:

true

true

true

Convert time in 24 hours format.

Problem Statement -

Given a time in 12 hours format, convert it into 24 hours format.

Example

Input:

"12:10AM"

"12:33PM"

Output:

00

:

10

12

:

33

We will have to check the input string in which period of 12 hours

format i.e “AM” or “PM” and then accordingly format them.

JavaScript [string](#) has method `endsWith` which can be used to check if the time is ending with “AM” or “PM”.



Split the time into hours and minutes to easily convert them accordingly based on which period they belong to.

There is a special case we need to handle for 12 AM as its value will be 0. For the remaining we can return the same number if it is AM or add twelve and return the new number if it is PM.

```
const  
  
formatTime = (  
  
time  
  
) => {  
  
  //convert the input to lowercase  
  
  const  
  
  timeLowerCased = time.toLowerCase();  
  
  //split the hours and mins  
  
  let  
  
  [hours, mins] = timeLowerCased.split(  
  
    ":"  
  
  );  
  
  // Special case
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

208

```
// 12 has to be handled for both AM and PM.
```

```
if
(timeLowerCased.endsWith(
"am"
)){
hours = hours ==
12
?
"0"
: hours;
}
else
if
(timeLowerCased.endsWith(
"pm"
)){
hours = hours ==
12
? hours :
String
(+hours
```

```
+  
12  
);  
}  
return  
`${hours.padStart(  
2  
,  
0  
)}:${mins.slice(  
0  
,  
-2  
).padStart(  
2  
,  
0  
)}`  
;  
}
```

Test Case

Input:

console

```
.log(formatTime(
```

```
"12:10AM"
```

```
));
```

console

```
.log(formatTime(
```

```
"12:33PM"
```

```
));
```

Output:

00

:

10

12

:

33

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

209

Convert time in 12 hours format.

## Problem Statement -

Given a time in 24 hours format, convert it into 12 hours format.

### Example

Input:

"00:00"

"12:33"

Output:

"12:00 AM"

"12:33 PM"

We will split the input string on “:” and get the hour and minutes.

By default keep the time Meridiem to “AM”.

Check if the hour is greater than 12 then reset the hour minusing 12 from it and set time Meridiem to “PM”.

If the hour is “00” then simply reset the hour to 12.

Finally return the string along with the minutes.

```
const
```

```
formatTime = (
```

```
time
```

```
) => {
```

```
// split the time
```

```
const
timeSplitted = time.split(
":"
);
// default is AM
let
ampm =
'AM'
;
// hour is greater than 12 then its PM
if
(timeSplitted[
0
] >=
12
) {
ampm =
'PM'
;
}
```

210

```
// reset the hour if time is greater than 12
```

```
if
```

```
(time_splitted[
```

```
0
```

```
12
```

```
) {
```

```
time_splitted[
```

```
0
```

```
] = time_splitted[
```

```
0
```

```
]-
```

```
12
```

```
;
```

```
}
```

```
// if hour is zero, reset the hour to 12
```

```
if
```

```
(time_splitted[
```

```
0
] ==
0
){
time_splitted[
0
] =
12
;
}
// return the converted time
return
time_splitted[
0
] +
':'
+ time_splitted[
1
]
+
```



```
''
```

```
+ ampm;
```

```
}
```

Test Case

Input:

console

```
.log(formatTime(
```

```
"12:33"
```

```
));
```

console

```
.log(formatTime(
```

```
"00:33"
```

```
));
```

Output:

```
"12:33 PM"
```

```
"12:33 AM"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

211

Create a digital clock.

Problem Statement -

Create a digital clock that shows the current time in HH:MM:SS  
format.

Example

10

:

57

:

23

10

:

57

:

24

10

:

57

:

25

10

:

57

:

26

10

:

57

:

27

Date function returns the single moment in time in a platform independent format. There are different methods available with the

Date function which we will be using to create our clock.

- *getHours()* - This method returns the hours of date in 24hrs number format like (0 – 23).
- *getMinutes()* - This will return the minutes of the hours in number format like from (0 – 59).
- *getSeconds()* - This method returns the seconds of the minutes in number format like (0 – 59).
- *getMilliseconds()* - This method returns the milliseconds of the seconds in number format like (0 – 999).

We will create a function which will be using these methods to get the current time in HH:MM:SS format.

const

```
clock =  
  
()  
  
=> {  
  
  const  
  
  time =  
  
  new  
  
  Date  
  
  (),  
  
  hours = time.getHours(),  
  
  minutes = time.getMinutes(),  
  
  seconds = time.getSeconds();  
  
  return  
  
  pad(hours) +  
  
  ':'  
  
  + pad(minutes) +  
  
  ':'  
  
  + pad(seconds);  
  
  © JavaScript Interview Guide | learnersbucket.com  
  
  212  
  
  }
```

We are using this pad() helper function which will format the input by appending 0 if the number is a single digit.

```
const
pad = (
inp
) => {
return
String
(inp).length ==
1
?
'0'
+ inp : inp;
};
```

Now when we call the clock function it will return the single instance of time at the moment it was called.

```
console
.log(clock());

//10:59:23
```

But to make it work like a clock we will need to call this function

repeatedly after 1 second. For this we will use the setInterval function which repeatedly calls the function after a given interval of time.

```
setInterval(  
  function  
  () {  
    console  
      .log(clock());  
  },  
  1000  
);  
//10:59:23  
//10:59:24  
//10:59:25  
//10:59:26
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

213

### Chop array into chunks of given length

#### Problem Statement -

Write a function to chop an array into chunks of a given length and return each chunk as an array without modifying the input array.

#### Example

Input:

[

1

,

2

,

3

,

4

,

5

,

6

,

7

,

8

,

9

,

10

]

3

Output:

[[

1

,

2

,

3

], [

4

,

5

,

6

], [

7

,

8



,

9

], [

10

]]

Let us see the two different implementations of this problem.

1. A normal function that will take the input and return the output.
2. We will extend the JavaScript array and add a new method chop, which will do the same.

Normal function

We will traverse till there is an element in the array, in each iteration slice the sub-array of the given size and push them to the output array.

```
const
```

```
chop = (
```

```
arr, size = arr.length
```

```
) => {
```

```
//temp array
```

```
const
```

```
temp = [...arr];
```

```
//output
```

```
const
```

```
output = [];  
  
let  
  
i =  
  
0  
  
;  
  
//iterate the array  
  
while  
  
(i < temp.length) {  
  
//slice the sub-array of given size  
  
© JavaScript Interview Guide | learnersbucket.com  
  
214  
  
//and push them in output array  
  
output.push(temp.slice(i, i + size));  
  
i = i + size;  
  
}  
  
return  
  
output;  
  
}
```

Test Case

Input:

console

.log(chop([

1

,

2

,

3

,

4

,

5

,

6

,

7

,

8

,

9

,

10

],

3

));

Output:

[[

1

,

2

,

3

], [

4

,

5

,

6

], [

7

,

8

,

9

], [

10

]]

Adding a new method to the array by extending it

We can use the same logic and add a new method to the array by extending its prototype.

Deep copy all the elements of the array so that we don't mutate the original array and if the size is not defined then return this deep copy, else return the array of chunks.

Array

.prototype.chop =

function

(

size

){

//temp array

const

temp = [...this];

```
//if size is not defined
```

```
if
```

```
(!size){
```

```
return
```

```
temp;
```

```
}
```

```
//output
```

```
const
```

```
output = [];
```

```
let
```

```
i =
```

```
0
```

```
;
```

```
//iterate the array
```

```
while
```

```
(i < temp.length) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

215

```
//slice the sub-array of given size
```

```
//and push them in output array
```

```
output.push(temp.slice(i, i + size));
```

```
i = i + size;
```

```
}
```

```
return
```

```
output;
```

```
}
```

Test Case

Input:

```
const
```

```
arr = [
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
,
```

```
5
```

```
,
```

6

,

7

,

8

,

9

,

10

];

const

output = arr.chop(

3

);

console

.log(output);

Output:

[[

1

,



2

,

3

], [

4

,

5

,

6

], [

7

,

8

,

9

], [

10

]]

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

Chop string into chunks of given length

Problem Statement -

Write a function to chop string into chunks of given length and return it as an array.

Example

Input:

'javascript'

3

Output:

[

'jav'

,

'asc'

,

'rip'

,

't'

]

We can solve this problem with two different approaches.

1. Bruteforce.

2. Regex.

## Bruteforce approach

It is extremely straightforward, iterate each character of the string and keep on slicing the characters of the given size and pushing it into the output array.

```
const
chop = (
str, size = str.length
) => {
const
arr = [];
let
i =
0
;
//iterate the string
while
(i < str.length) {
//slice the characters of given size
//and push them in output array
arr.push(str.slice(i, i + size));
i = i + size;
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

217

```
return
```

```
arr;
```

```
}
```

Test Case

Input:

```
console
```

```
.log(chop(
```

```
'javascript'
```

```
,
```

```
3
```

```
));
```

Output:

```
[
```

```
"jav"
```

```
,
```

```
"asc"
```

```
,
```

```
"rip"
```

```
,
```

```
"t"
```

```
]
```

## Using Regex

We often tend to ignore the Regex based solution as it is not easy to remember the expressions, but the Regex method accepts the size

which can be used to extract at-most n-sized sub strings.

### Regex expression

```
str.match(
```

```
/.{1,n}/g
```

```
);
```

```
// Replace n with the size of
```

```
the substring
```

If the string contains any newlines or carriage returns, then use this expression.

```
str.match(
```

```
/([^\r\n]){1,n}/g
```

```
);
```

```
// Replace n with the
```

```
size of the substring
```

String .prototype .match() returns an array of matching strings with a regular expression. Passing the regex to it will return the array of n-sized sub strings.

```
const
```

```
chop = (
```

```
str, size = str.length
```

```
) => {
```

```
  return
```

```
    str.match(
```

```
      new
```

```
      RegExp
```

```
    (
```

```
      '{1,'
```

```
      + size +
```

```
      '}'
```

```
    ,
```

```
    'g'
```

```
  ));
```

```
}
```

Test Case

Input:

218

console

.log(chop(

'javascript'

,

3

));

Output:

[

"jav"

,

"asc"

,

"rip"

,

"t"

]

219

## Deep flatten object

### Problem Statement -

Given an nested object which can have any type of object, deep flatten it and return the new object in Javascript.

### Example

Input:

```
{
```

A:

"12"

,

B:

23

,

C: {

P:

23

,

O: {

L:

56

```
},
```



Q: [

1

,

2

]

}

}

Output:

{

"A"

:

"12"

"B"

:

23

,

"C.O.L"

:

56

,

```
"C.P"  
:  
23  
  
,  
"C.Q.0"  
:  
1  
  
,  
"C.Q.1"  
:  
2  
  
,  
}
```

In the output if you notice, when we have nested objects, the key is concatenated till there is a non-object value, similar for the array, the key is concatenated on the index.

We can solve this using the following steps.

1. Check if the value of the given key is an object or not.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

220

2. If it is not an object then add that value to the output.

3. Else, check if the value is an array or not.
4. If it is an object or array then recursively call the same function with value and pass the key to be used as prefix and return the output in the existing result.
5. Otherwise, iterate the value and use the array's index along with the existing key as a new key and then store it in the output.
6. Alternatively, we can convert the array to object and then recursively call the same function as we did in step 4.

```
const  
flatten = (  
  obj, prefix  
) => {  
  //store the result  
  let  
  output = {};  
  //iterate the object  
  for  
  (  
  let  
  k  
  in
```

```
obj){  
  let  
  val = obj[k];  
  //new key  
  const  
  newKey = prefix ? prefix +  
  "."  
  + k : k;  
  //array and object both are object in js  
  if  
  (  
  typeof  
  val ===  
  "object"  
  ){  
    // if it is array  
    if  
    (  
    Array  
    .isArray(val)){
```

```
//use rest & spread together to convert
```

```
//array to object
```

```
const
```

```
{ ...arrToObj } = val;
```

```
const
```

```
newObj = flatten(arrToObj, newKey);
```

```
output = {...output, ...newObj};
```

```
}
```

```
//if it is object
```

```
else
```

```
{
```

```
const
```

```
newObj = flatten(val, newKey);
```

```
output = {...output, ...newObj};
```

```
}
```

```
}
```

```
// normal value
```

```
else
```

```
{
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

221

```
output = {...output, [newKey]: val};
```

```
}
```

```
}
```

```
return
```

```
output;
```

```
}
```

Test Case

Input:

```
const
```

```
nested = {
```

A:

"12"

,

B:

23

,

C: {

P:

23

```
,  
O: {  
  L:  
    56  
  },  
  Q: [  
    1  
  ],  
  }  
};  
  
console  
.log(flatten(nested));
```

Output:

```
{  
  "A"  
  :  
    "12"  
  "B"
```

:

23

,

"C.O.L"

:

56

,

"C.P"

:

23

,

"C.Q.0"

:

1

,

"C.Q.1"

:

2

,

}



222

## Restrict modification of object properties

### Problem Statement -

Restrict adding, removing and changing or simply the modification of object properties.

Objects are the backbone of JavaScript as most of its features are actually objects, like arrays, functions, etc.

To make objects powerful it was obvious that it should be allowed to be modified or extended so that it can be used in different forms.

But there are scenarios where we want to restrict this modification to some limit or completely.

We will see two different ways in which we can achieve the same.

Using `Object.seal()` to restrict object extending

For example, we should be able to modify the existing properties or methods of the objects but cannot add a new one. `Object.seal()` can be used to achieve the same but it also marks all existing properties as non-configurable like we cannot delete them but just update their value if it is writable.

```
const
```

```
obj = {
```

```
  prop:
```

42

```
};
```

Object

```
.seal(obj);
```

```
obj.prop =
```

33

```
;
```

console

```
.log(obj.prop);
```

```
// 33
```

delete

```
obj.prop;
```

```
// cannot delete when sealed
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

223

console

```
.log(obj.prop);
```

```
// 33
```

But we cannot restrict the modification of nested objects with

Object.seal() .

```
const
obj = {
  prop:
    42
  ,
  nested: {
    a:
      1
    ,
    b:
      2
    }
  };
//Seal the object
Object
.seal(obj);
obj.nested.a =
  2
;
delete
```

```
obj.nested.a;  
  
console  
  
.log(obj.nested.a);  
  
// undefined
```

However, we can create another helper function which will deep seal or seal the nested objects as well.

```
function  
  
deepSeal  
  
(  
  object  
) {  
  // Retrieve the property names defined on object  
  
  let  
  
  propNames =  
  
    Object  
  
    .getOwnPropertyNames(object);  
  
  // Seal properties before Sealing self  
  
  for  
  
  (  
  
    let
```

```
name
of
propNames) {
let
value = object[name];
object[name] = value &&
typeof
value ===
"object"
?
deepSeal(value) : value;
}
return
Object
.seal(object);
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

224

Now this will seal the nested objects as well.

```
const
```

```
obj = {  
  prop:  
    42  
  ,  
  nested: {  
    a:  
      1  
    ,  
    b:  
      2  
    }  
  };  
  //Seal the object  
  deepSeal(obj);  
  obj.nested.a =  
    2  
  ;  
  delete  
  obj.nested.a;  
  console
```

```
.log(obj.nested.a);
```

```
// 2
```

We can use `Object.isSealed()` to confirm if an object is sealed or not.

```
const
```

```
obj = {
```

```
  prop:
```

```
    42
```

```
,
```

```
  nested: {
```

```
    a:
```

```
      1
```

```
,
```

```
    b:
```

```
      2
```

```
  }
```

```
};
```

```
//Seal the object
```

```
deepSeal(obj);
```

```
console
```

```
.log(
```

Object

```
.isSealed(obj));
```

```
//true
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

225

Using Object.freeze() to restrict modification of object properties.

Unlike Object.seal() , Object.freeze() freezes the object completely. It does not even allow changing of object properties.

```
const
```

```
obj = {
```

```
prop:
```

```
42
```

```
};
```

Object

```
.freeze(obj);
```

```
obj.prop =
```

```
33
```

```
;
```

```
// Throws an error in strict mode
```

```
console
```

```
.log(obj.prop);
```



```
// 42
```

But this also only shallowly freezes the nested object properties.

```
const
```

```
obj = {
```

```
  prop:
```

```
    42
```

```
,
```

```
  nested: {
```

```
    a:
```

```
      1
```

```
,
```

```
    b:
```

```
      2
```

```
  }
```

```
};
```

```
Object
```

```
  .freeze(obj);
```

```
obj.nested.a =
```

```
  33
```

```
;
```

```
// Updates the value
```

```
console
```

```
.log(obj.nested.a);
```

```
// 33
```

Like `deepSeal()` we can also create a `deepFreeze()` function that will freeze the nested objects as well.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

226

```
function
```

```
deepFreeze
```

```
(
```

```
object
```

```
) {
```

```
// Retrieve the property names defined on object
```

```
var
```

```
propNames =
```

```
Object
```

```
.getOwnPropertyNames(object);
```

```
// Freeze properties before freezing self
```

```
for
```

```
(
```

```
let
  name
  of
  propNames) {
  let
    value = object[name];
    object[name] = value &&
    typeof
    value ===
    "object"
    ?
    deepFreeze(value) : value;
  }
  return
  Object
  .freeze(object);
}
const
  obj = {
  prop:
```

42

,

nested: {

a:

1

,

b:

2

}

};

deepFreeze(obj);

obj.nested.a =

33

;

// Updates the value

console

.log(obj.nested.a);

// 1

You can use `Object.isFrozen()` to check if an object is frozen or not.

const

```
obj = {  
  prop:  
    42  
  ,  
  nested: {  
    a:  
      1  
    ,  
    b:  
      2  
    }  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

227

```
};
```

```
//Seal the object
```

```
deepFreeze(obj);
```

```
console
```

```
.log(
```

```
Object
```

```
.isFrozen(obj));
```

//true

© JavaScript Interview Guide | learnersbucket.com

228

Merge objects

Problem Statement -

Write a program to merge two or more objects without using any native methods.

There are two different types of merge that can be performed on the objects.

1. *Shallow* : In shallow merge only the properties owned by the object will be merged, it will not merge the extended properties

or methods.

2. *Deep* : In deep merge properties and extended properties of the objects are merged as well, if it exists.

Shallow merging

We will iterate all the keys of the source object and check if the property belongs to the source object then only copy it to the target.

let

merge = (

...arguments

) => {

```
// Create a new object

let

target = {};

// Merge the object into the target object

let

merger = (

obj

) => {

for

(

let

prop

in

obj) {

if

(obj.hasOwnProperty(prop)) {

// Push each value from `obj` to `target`

target[prop] = obj[prop];

}

}

}
```

```
};  
  
// Loop through each object and conduct a merge  
for  
  
(  
  
let  
  
i =  
  
0  
  
; i <  
  
arguments  
  
.length; i++) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

229

```
merger(  
  
arguments  
  
[i]);  
  
}  
  
return  
  
target;  
  
}
```

Test Case 1



Input:

```
let
```

```
obj1 = {
```

```
  name:
```

```
  'prashant'
```

```
,
```

```
  age:
```

```
  23
```

```
,
```

```
}
```

```
let
```

```
obj2 = {
```

```
  qualification:
```

```
  'BSC CS'
```

```
,
```

```
  loves:
```

```
  'Javascript'
```

```
}
```

```
let
```

```
merged = merge(obj1, obj2);
```

console

.log(merged);

Output:

```
/*
```

```
Object {
```

```
  age: 23,
```

```
  loves: "Javascript",
```

```
  name: "prashant",
```

```
  qualification: "BSC CS"
```

```
}
```

```
*/
```

Test Case 2

In this if you notice the skills value is overridden by the last object.

Input:

```
let
```

```
obj1 = {
```

```
  name:
```

```
  'prashant'
```

```
,
```

```
  age:
```

23

,

skills : {programming:

'JavaScript'

}

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

230

}

let

obj2 = {

qualification:

'BSC CS'

,

loves:

'Javascript'

,

skills : {sports:

'swimming'

}

}

```
let
```

```
merged = merge(obj1, obj2);
```

```
console
```

```
.log(merged);
```

Output:

```
/*
```

```
Object {
```

```
  age: 23,
```

```
  loves: "Javascript",
```

[OceanofPDF.com](https://oceanofpdf.com)

```
name: "prashant",  
qualification: "BSC CS",  
skills: {  
  sports:  
    "swimming"}  
}  
*/
```

## Deep merging

To deep merge an object we have to copy the own properties and extended properties as well.

```
let  
merge = (  
  ...arguments  
) => {  
  // Variables  
  let  
  target = {};  
  // Merge the object into the target object  
  let  
  merger = (
```

```

obj
) => {
  for
  (
  let
  prop
  in
  obj) {
    if
    (obj.hasOwnProperty(prop)) {
      if
      (
      Object
      .prototype.toString.call(obj[prop])
      ===
      '[object
      Object]'
      ) {
        // If we're doing a deep merge and the property
        is an object

```

231

```
target[prop] = merge(target[prop], obj[prop]);  
  
}  
  
else  
  
{  
  
// Otherwise, do a regular merge  
  
target[prop] = obj[prop];  
  
}  
  
}  
  
}  
  
};  
  
//Loop through each object and conduct a merge  
  
for  
  
(  
  
let  
  
i =  
  
0  
  
; i <  
  
arguments
```

```
.length; i++) {  
  merger(  
    arguments  
    [i]);  
}  
return  
target;  
};
```

## Test Case

In this if you notice the nested objects with different values are merged.

```
let  
obj1 = {  
  name:  
    'prashant'  
  ,  
  age:  
    23  
  ,  
  nature: {
```



"helping"

:

true

,

"shy"

:

false

}

}

let

obj2 = {

qualification:

'BSC CS'

,

loves:

'Javascript'

,

nature: {

"angry"

:

false

,

"shy"

:

true

}

}

console

.log(merge(obj1, obj2));

/\*

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

232

Object {

age: 23,

loves: "Javascript",

name: "prashant",

nature: Object {

angry: false,

helping: true,

shy: true

```
},
```

```
qualification: "BSC CS"
```

```
}
```

```
*/
```

Common function for shallow and deep merging

We can combine both the function for shallow copy and deep copy

together to create a single function which will perform merge based on the arguments passed.

If we will pass true as first argument then it will perform deep merge else it will perform shallow merge.

```
let
```

```
merge = (
```

```
...arguments
```

```
) => {
```

```
// Variables
```

```
let
```

```
target = {};
```

```
let
```

```
deep =
```

```
false
```

```
;
```

```
let
i =
0
;
// Check if a deep merge
if
(
typeof
(
arguments
[
0
]) ===
'boolean'
) {
deep =
arguments
[
0
];
```

```
i++;  
  
}  
  
// Merge the object into the target object  
  
let  
  
merger = (  
  
obj  
  
) => {  
  
for  
  
(  
  
let  
  
prop  
  
in  
  
obj) {  
  
if  
  
(obj.hasOwnProperty(prop)) {  
  
if  
  
(deep &&  
  
Object  
  
.prototype.toString.call(obj[prop])
```

===

233

'[object Object]'

) {

// If we're doing a deep merge and the property

is

an object

target[prop] = merge(target[prop], obj[prop]);

}

else

{

// Otherwise, do a regular merge

target[prop] = obj[prop];

}

}

}

};

//Loop through each object and conduct a merge

for

(; i <

arguments

.length; i++) {

merger(

arguments

[i]);

}

return

target;

};

Test Case

let

obj1 = {

name:

'prashant'

,

age:

23

,

nature: {

"helping"

:

true

,

"shy"

:

false

}

}

let

obj2 = {

qualification:

'BSC CS'

,

loves:

'Javascript'

,

nature: {

"angry"

:

false



```
,  
"shy"
```

```
:
```

```
true
```

```
}
```

```
}
```

```
//Shallow merge
```

```
console
```

```
.log(merge(obj1, obj2));
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
234
```

```
/*
```

```
Object {
```

```
age: 23,
```

```
loves: "Javascript",
```

```
name: "prashant",
```

```
nature: Object {
```

```
angry: false,
```

```
shy: true
```

```
},
```

```
qualification: "BSC CS"
```

```
}
```

```
*/
```

```
//Deep merge
```

```
console
```

```
.log(merge(
```

```
true
```

```
, obj1, obj2));
```

```
/*
```

```
Object {
```

```
age: 23,
```

```
loves: "Javascript",
```

```
name: "prashant",
```

```
nature: Object {
```

```
angry: false,
```

```
helping: true,
```

```
shy: true
```

```
},
```

```
qualification: "BSC CS"
```

```
}
```

\*/

© JavaScript Interview Guide | learnersbucket.com

235

Implement browser history

Problem Statement -

You must be familiar with browser history and its functionality where you can navigate through the browsed history. Implement the same.

It will have the following functionality

- *visit(url)* : Marks the entry of the URL in the history.
- *current()* : Returns the URL of the current page.
- *backward()* : Navigate to the previous url.
- *forward()* : Navigate to the next url.

Example

Input:

const

bh =

new

BrowserHistory();

bh.visit(

'A'

);

console

```
.log(bh.current());
```

```
bh.visit(
```

```
'B'
```

```
);
```

console

```
.log(bh.current());
```

```
bh.visit(
```

```
'C'
```

```
);
```

console

```
.log(bh.current());
```

```
bh.goBack();
```

console

```
.log(bh.current());
```

```
bh.visit(
```

```
'D'
```

```
);
```

console

```
.log(bh.current());
```

Output:

"A"

"B"

"C"

"B"

"D"

We can implement this with the help of an array and index tracker for navigation.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

236

For each visit, add the URL at the next index. while navigating backward return the URL of the previous index, going forward return

the URL at the next index. Add checks to prevent the under and

overflowing of the indexes.

function

BrowserHistory

() {

// track history

this

.history = [];

this

```
.index =  
-1  
;  
// add new url at next index  
this  
.visit =  
function  
(  
url  
) {  
this  
.history[++  
this  
.index] = url;  
}  
// return the url of the current index  
this  
.current =  
function  
( ) {
```

```
return  
  
this  
  
.history[  
  
this  
  
.index];  
  
}  
  
// go to previous entry  
  
this  
  
.backward =  
  
function  
  
() {  
  
this  
  
.index =  
  
Math  
  
.max(  
  
0  
  
, --  
  
this  
  
.index);  
  
}
```

```
// go to next entry
```

```
this
```

```
.forward =
```

```
function
```

```
() {
```

```
this
```

```
.index =
```

```
Math
```

```
.min(
```

```
this
```

```
.history.length -
```

```
1
```

```
,
```

```
++
```

```
this
```

```
.index);
```

```
}
```

```
}
```

Test Case

Input:



```
const  
bh =  
new  
BrowserHistory();  
bh.visit(  
'A'  
);  
console  
.log(bh.current());
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

237

```
bh.visit(  
'B'  
);  
console  
.log(bh.current());  
bh.visit(  
'C'  
);  
console
```

```
.log(bh.current());
```

```
bh.backward();
```

```
console
```

```
.log(bh.current());
```

```
bh.visit(
```

```
'D'
```

```
);
```

```
console
```

```
.log(bh.current());
```

```
bh.backward();
```

```
console
```

```
.log(bh.current());
```

```
bh.forward();
```

```
console
```

```
.log(bh.current());
```

Output:

```
"A"
```

```
"B"
```

```
"C"
```

```
"B"
```

"D"

"B"

"D"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

238

Singleton design pattern

Problem Statement -

Create an implementation of a singleton design pattern in JavaScript.

In a singleton design pattern, only one object is created for each interface (class or function) and the same object is returned every time when called.

It is really useful in scenarios where only one object is needed to coordinate actions across the system. For example, a notification object, which sends notification across the system.

Example

```
const
```

```
object1 = singleton.getInstance();
```

```
const
```

```
object2 = singleton.getInstance();
```

```
console
```

```
.log(object1 === object2);
```

```
//true
```

We can implement the singleton pattern by creating a closure with a variable that stores the created instance and returns it every time.

```
const
```

```
Singleton = (
```

```
function
```

```
() {
```

```
let
```

```
instance;
```

```
function
```

```
createInstance
```

```
() {
```

```
const
```

```
object =
```

```
new
```

```
Object
```

```
(
```

```
"I am the instance"
```

```
);
```

```
return
```

```
object;
}
return
{
  getInstance:
  function
  () {
    if
    (!instance) {
      instance = createInstance();
    }
    return
    instance;
  }
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

239

```
}
};
})();
```

Test Case

```
const
```

```
object1 = singleton.getInstance();
```

```
const
```

```
object2 = singleton.getInstance();
```

```
console
```

```
.log(object1 === object2);
```

```
//true
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

240

## Observer design pattern

### Problem Statement -

Create an implementation of a observer design pattern in JavaScript.

Observer design also known as pub/sub pattern short for

publication/subscription. It is clear from the name itself that if you are subscribed to the publication and if something is published in the

publication it will notify the subscriber.

In other words, A subscription model in which an object subscribes to a host and the host notifies the object whenever an event occurs is

known as the observer pattern. It is one of the important pillars of event-driven programming and JavaScript is one of the most popular

event-driven programming languages. This pattern promotes loose

coupling facilitating good object-oriented design.

JavaScript is the most avid follower of observer pattern and we use it almost regularly while building web apps. We write event handlers by creating event listeners that listen to an event and notify them every time the event is fired with some additional details of the event.

For example, when a click event is triggered you can access the event object to get all the event details about the click like its position on the screen, etc.

You can also remove the listener (unsubscribe) to stop listening if you want.

### Creating observer design pattern in JavaScript

To create the observer design pattern, we need to have two types of participants.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

241

#### 1. Host

- It will maintain the list of observers.
- Provides options to subscribe and unsubscribe to the observers.
- Notifies the observer when state changes.

#### 2. Observer

Has a function that gets called/invoked every time a state changes.

Keeping these two things in mind, we can create the Observer design pattern in JavaScript.

const

```
Move =  
  
function  
  
{  
  
  this  
  
    .handlers = [];  
  
  this  
  
    .subscribe =  
  
    function  
  
    (  
  
      fn  
  
    ) {  
  
      this  
  
        .handlers.push(fn);  
  
    };  
  
  this  
  
    .unsubscribe =  
  
    function  
  
    (  
  
      fn  
  
    ) {
```



this

.handlers =

this

.handlers.filter((

item

) =>

item !== fn);

};

this

.fire =

function

(

o, thisObj

) {

const

scope = thisObj ||

window

;

this

.handlers.forEach((

```
item
```

```
) => {
```

```
item.call(scope, o);
```

```
});
```

```
}
```

```
}
```

Test Case

Input:

```
// 1st observer
```

```
const
```

```
moveHandler =
```

```
function
```

```
(
```

```
item
```

```
) {
```

```
console
```

```
.log(
```

```
"fired: "
```

```
+ item);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

242

```
};
```

```
// 2nd observer
```

```
const
```

```
moveHandler2 =
```

```
function
```

```
(
```

```
item
```

```
) {
```

```
console
```

```
.log(
```

```
"Moved: "
```

```
+ item);
```

```
};
```

```
const
```

```
move =
```

```
new
```

```
Move();
```

```
// subscribe 1st observer
```

```
move.subscribe(moveHandler);
```

```
move.fire(  
  'event #1'  
);  
  
// unsubscribe 1st observer  
move.unsubscribe(moveHandler);  
  
move.fire(  
  'event #2'  
);  
  
// subscribe 1st & 2nd observer  
move.subscribe(moveHandler);  
move.subscribe(moveHandler2);  
  
move.fire(  
  'event #3'  
);
```

Output:

"fired: event #1"

"fired: event #3"

"Moved: event #3"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

Implement `groupBy()` method

Problem Statement -

Write the polyfill for the `groupBy()` method that accepts a collection and iteratee as arguments and returns the object that has grouped the collection values using iteratee's value as the key.

Example

Input:

```
groupBy([
```

```
  6.1
```

```
,
```

```
  4.2
```

```
,
```

```
  6.3
```

```
],
```

```
Math
```

```
.floor);
```

```
groupBy([
```

```
  "one"
```

```
,
```

```
  "two"
```

```
,
```

```
"three"
```

```
],
```

```
"length"
```

```
);
```

Output:

```
// { 6: [6.1, 6.3], 4: [4.2] }
```

```
// { 3: ['one', 'two'], 5: ['three'] }
```

The whole logic can be abstracted inside the [Array.reduce\(\)](#) method and we can easily aggregate the values.

Here the iteratee can be a function or a property, thus we will have to check the type of iteratee, and depending upon that we can get the key from it.

```
const
```

```
groupBy = (
```

```
values, keyFinder
```

```
) => {
```

```
// using reduce to aggregate values
```

```
return
```

```
values.reduce((
```

```
a, b
```

```
) => {
```

```
// depending upon the type of keyFinder
```

```
// if it is function, pass the value to it

// if it is a property, access the property

const

key =

typeof

keyFinder ===

'function'

?

keyFinder(b) :

b[keyFinder];

// aggregate values based on the keys

if

(!a[key]){

a[key] = [b];

}

else

{

© JavaScript Interview Guide | learnersbucket.com

244

a[key] = [...a[key], b];
```

```
}
```

```
return
```

```
a;
```

```
}, {});
```

```
};
```

Test Case

Input:

console

```
.log(groupBy([
```

```
6.1
```

```
,
```

```
4.2
```

```
,
```

```
6.3
```

```
],
```

Math

```
.floor));
```

console

```
.log(groupBy([
```

```
"one"
```



```
,  
"two"  
  
,  
"three"  
  
],  
"length"  
  
));
```

Output:

```
// { 6: [6.1, 6.3], 4: [4.2] }  
  
// { 3: ['one', 'two'], 5: ['three'] }
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

245

Compare two array or object

Problem Statement -

Write a program to deeply compare two arrays or objects.

There are no specific built-in methods available to us to compare two different arrays or objects in javascript. We will have to create our own custom function which will compare two different objects.

Method 1: Using JSON.stringify()

The most basic approach is to convert the whole array or the object to a string then compare if those strings are equal or not.

To convert an array or object we will be using [JSON.stringify\(\)](#).

```
let
```

```
a = {a:
```

```
  1
```

```
, b:
```

```
  2
```

```
, c:
```

```
  3
```

```
};
```

```
let
```

```
b = {a:
```

```
  1
```

```
, b:
```

```
  2
```

```
, c:
```

```
  3
```

```
};
```

```
/"{'a':1,'b':2,'c':3}" "{'a':1,'b':2,'c':3}"
```

```
console
```

```
.log(
```

JSON

```
.stringify(a) ===
```

JSON

```
.stringify(b));
```

```
//true
```

This may seem to be working fine, but this approach fails when we change the order of the elements.

```
let
```

```
a = {a:
```

```
1
```

```
, b:
```

```
2
```

```
, c:
```

```
3
```

```
};
```

```
let
```

```
b = {b:
```

```
2
```

```
, a:
```

```
1
```

```
, c:
3
};
//{"a":1,'b':2,'c':3}" {"b":2,'a':1,'c':3}"
console
.log(
JSON
.stringify(a) ===
JSON
.stringify(b));
//false
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

246

**Method 2: Recursively deep check arrays or objects** As arrays are basically objects only in JavaScript, we can use the

object's methods on it. `Object.keys([1, 2])` will return the indexes as key and thus we can use it to get the values from the array.

- Get the keys of both the objects.
- If the number of keys are not the same, return false.
- Iterate the keys, if the values of both the objects for the given keys are objects, recursively call the same function with values

and deep check, if they are not equal, return false.

- If the values are non-iterable, do the equality check, if they are not equal return false.
- If all the cases are passed, return true at the end.

```
const
```

```
deepEqual = (
```

```
object1, object2
```

```
) => {
```

```
// get object keys
```

```
const
```

```
keys1 =
```

```
Object
```

```
.keys(object1);
```

```
const
```

```
keys2 =
```

```
Object
```

```
.keys(object2);
```

```
// if mismatched keys
```

```
if
```

```
(keys1.length !== keys2.length) {
```

```
return
```

```
false  
;  
}  
for  
(  
  const  
  key  
  of  
  keys1) {  
  // get the values  
  const  
  val1 = object1[key];  
  const  
  val2 = object2[key];  
  // if both values are objects  
  const  
  areObjects = val1 &&  
  typeof  
  val1 ===  
  "object"
```

```
&& val2 &&
```

```
typeof
```

```
val2 ===
```

```
"object"
```

```
;
```

```
// if are objects
```

```
if
```

```
(areObjects){
```

```
// deep check again
```

```
if
```

```
(!deepEqual(val1, val2)){
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
247
```

```
return
```

```
false
```

```
;
```

```
}
```

```
}
```

```
// if are not objects
```

```
// compare the values
```

else

if

(!areObjects && val1 !== val2){

return

false

;

}

}

return

true

;

}

Test Case

Input:

const

obj1 = {

name:

"learnersbucket"

,

details: {



```
x: [
```

```
1
```

```
,
```

```
2
```

```
],
```

```
y:
```

```
2
```

```
,
```

```
},
```

```
};
```

```
const
```

```
obj2 = {
```

```
name:
```

```
"learnersbucket"
```

```
,
```

```
details: {
```

```
y:
```

```
2
```

```
,
```

```
x: [
```

1

,

2

],

},

};

console

.log(deepEqual(obj1, obj2));

Output:

true

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

248

Array iterator method

Problem Statement -

Create an iterator method that accepts an array and returns a new method that will return the next array value on each invocation.

Example

let

iterator = helper([

1

```
,  
2  
  
,  
"hello"  
]);  
  
console  
.log(iterator.next());  
  
// 1  
  
console  
.log(iterator.next());  
  
// 2  
  
console  
.log(iterator.done());  
  
// false  
  
console  
.log(iterator.next());  
  
// "hello"  
  
console  
.log(iterator.done());  
  
// true
```

```
console
```

```
.log(iterator.next());
```

```
// "null"
```

We can create this helper function by forming a [closure](#) and returning an object with two methods next and done.

Create an index tracker in the outer function that will help to return the next value from the next() method.

In the done() return boolean flag depending upon the index position on the input array's size.

```
const
```

```
helper = (
```

```
array
```

```
) => {
```

```
// track the index
```

```
let
```

```
nextIndex =
```

```
0
```

```
;
```

```
// return two methods
```

```
return
```

```
{
```

```
// return the next value
```

```
// or null
```

```
next:
```

```
function
```

```
() {
```

```
  return
```

```
    nextIndex < array.length
```

```
    ? array[nextIndex++]
```

```
    :
```

```
    null
```

```
  ;
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
249
```

```
},
```

```
// returns boolean value
```

```
// if all the values are returned from array
```

```
done:
```

```
function
```

```
() {
```

```
  return
```

```
nextIndex >= array.length;
```

```
}
```

```
};
```

```
};
```

Test Case

```
let
```

```
iterator = helper([
```

```
1
```

```
,
```

```
2
```

```
,
```

```
"hello"
```

```
]);
```

```
console
```

```
.log(iterator.next());
```

```
// 1
```

```
console
```

```
.log(iterator.next());
```

```
// 2
```

```
console
```

```
.log(iterator.done());
```

```
// false
```

```
console
```

```
.log(iterator.next());
```

```
// "hello"
```

```
console
```

```
.log(iterator.done());
```

```
// true
```

```
console
```

```
.log(iterator.next());
```

```
// "null"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

250

Array with event listeners

Problem Statement -

Extend the arrays in javascript such that an event gets dispatched whenever an item is added or removed.

Example

Input:

```
const
```

```
arr = [];  
  
arr.addListener(  
  'add'  
  , (eventName, items, array) =>  
    {  
      console  
        .log(  
          'items were added'  
          , items);  
    });  
  
arr.addListener(  
  'remove'  
  , (eventName, item, array)  
    => {  
      console  
        .log(item,  
          ' was removed'  
        );  
    });  
  
arr.pushWithEvent(  
  'add',  
  items,  
  array);  
}
```



```
'add'  
  
 , [  
 4  
  
 ,  
 5  
  
 ]);  
  
arr.popWithEvent(  
  'remove'  
  
 );
```

Output:

```
"items were added again"
```

```
// [object Array] (2)
```

```
[
```

```
 4
```

```
 ,
```

```
 5
```

```
]
```

```
5
```

```
" was removed"
```

We will extend the array prototype and add the required methods to it

- *listeners* : This will store the list of event listeners associated with the event name.
- *addListener(eventName, callback)* : This will add a callback to the event.
- *pushWithEvent(eventName, items)* : Adds all the items in the array and triggers the event with the given name.
- *popWithEvent(eventName)* : Removes the last items from the array and triggers the event with the given name.

© JavaScript Interview Guide | learnersbucket.com

251

- *triggerEvent(eventName, args)* : A helper function that triggers all the callbacks associated with the given event name.
- *removeListener(eventName, callback)* : Removes the callback attached to the eventName. Note: It won't work for anonymous functions.

// to track the events and their callbacks

Array

.prototype.listeners = {};

// to add/assign a new event with listener

Array

.prototype.addListener =

function

(

```
name, callback
){
// if there are no listener present
// create a new one
// we will invoke all the callbacks when event is
triggered
if
(!
this
.listeners[name]) {
this
.listeners[name] = [];
}
this
.listeners[name].push(callback);
}
// add a new method that triggers an event on push
// Calls trigger event
Array
.prototype.pushWithEvent =
```

```
function
(
event, args
)
{
// push the new values
this
.push(...args);
// trigger add event
this
.triggerEvent(event, args);
};

// add a new method that triggers an event on pop
// Calls trigger event
Array
.prototype.popWithEvent =
function
(
event, args
)
```

```
{  
  // push the new values  
  
  const  
  element =  
  
  this  
  
  .pop();  
  
  // trigger add event  
  
  this  
  
  .triggerEvent(event, element);  
  
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

252

Array

```
.prototype.triggerEvent =  
function  
(  
  eventName, elements  
) {  
  // if the event is present  
  
  // trigger all the callbacks with the value
```

```
if
(
this
.listeners[eventName]) {
this
.listeners[eventName].forEach(
callback
=>
callback(eventName, elements,
this
)
);
}
};

Array
.prototype.removeListener =
function
(
eventName,
callback
```

```
) {  
  // if event exists  
  if  
  (  
    this  
    .listeners[eventName]){  
    // filter out the listener  
    // note: this won't work for anonymous function.  
    this  
    .listeners[eventName] =  
    this  
    .listeners[eventName].filter((  
      e  
    )  
    => e  
    !== callback);  
  }  
}
```

Test Case

Input:

```
const
```

```
arr = [];
```

```
const
```

```
onAdd = (
```

```
eventName, items, array
```

```
) => {
```

```
console
```

```
.log(
```

```
'items were added'
```

```
, items);
```

```
}
```

```
const
```

```
onAddAgain = (
```

```
eventName, items, array
```

```
) => {
```

```
console
```

```
.log(
```

```
'items were added again'
```

```
, items);
```

```
}
```



```
arr.addListener(  
  'add'  
  , onAdd);  
arr.addListener(  
  'add'  
  , onAddAgain);  
arr.addListener(  
  'remove'  
  , (eventName, item, array)  
  => {  
    console  
      .log(item,  
        ' was removed'  
      );  
  });  
arr.pushWithEvent(  
  'add'  
  , [  
    1  
  ,
```

2

,

3

,

'a'

,

'b'

]);

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

253

```
arr.removeListener(
```

```
'add'
```

```
, onAddAgain);
```

```
// removes
```

```
the second listener
```

```
arr.pushWithEvent(
```

```
'add'
```

```
, [
```

4

,

5

]);

arr.popWithEvent(

'remove'

);

console

.log(arr);

Output:

"items were added"

// [object Array] (5)

[

1

,

2

,

3

,

"a"

,

"b"

```
]
```

```
"items were added again"
```

```
// [object Array] (5)
```

```
[
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
"a"
```

```
,
```

```
"b"
```

```
]
```

```
"items were added"
```

```
// [object Array] (2)
```

```
[
```

```
4
```

```
,
```

```
5
```

]

5

" was removed"

// [object Array] (6)

[

1

,

2

,

3

,

"a"

,

"b"

,

4

]

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

254

Filter array of objects on value or index

Problem Statement -

Implement a function in JavaScript that filters an array of objects based on the value or index.

Example

```
const
```

```
arr = [
```

```
{ name:
```

```
"Amir"
```

```
, id:
```

```
"1"
```

```
},
```

```
{ name:
```

```
"Samlan"
```

```
, id:
```

```
"2"
```

```
},
```

```
{ name:
```

```
"Shahrukh"
```

```
, id:
```

```
"0"
```

```
},  
];  
  
console  
.log(filterObject(arr,  
0  
));  
  
// { name: "Amir",  
id: "1" }  
  
console  
.log(filterObject(arr,  
"Amir"  
));  
  
// { name:  
"Amir", id: "1" }  
  
console  
.log(filterObject(arr,  
"0"  
));  
  
// { name: "Shahrukh",  
id: "0" }
```

One way to solve this is by using the [Proxy](#) and overriding the get method, but the problem with using this is it converts the input values to a string, thus it is not easy to determine if we have to filter on index or value.

We can use an alternative approach where we will create a function

that will accept the array of objects and filter value as input and based on the type of filter, it will check in the object and return the

appropriate value. If nothing is found we will return undefined.

```
const
```

```
filterObject = (
```

```
arr, filter
```

```
) => {
```

```
// if the value of the filter is a string
```

```
// check in the values of the object
```

```
if
```

```
(
```

```
typeof
```

```
filter ===
```

```
"string"
```

```
){
```

```
for
```

```
(
```



```
const
entry
of
arr){
// traverse each entry and check on value
for
(
const
[key, val]
of
Object
.entries(entry)){
if
(val === filter){
return
entry;
}
}
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

255

```
}  
  
}  
  
// if filter is number and can be accessed in arr  
else  
  
if  
  
(filter  
  
in  
  
arr){  
  
return  
  
arr[filter];  
  
}  
  
// if nothing is found  
else  
  
{  
  
return  
  
undefined  
  
;  
  
}  
  
};
```

Test Case

Input:

```
const
```

```
arr = [
```

```
{ name:
```

```
"Amir"
```

```
, id:
```

```
"1"
```

```
},
```

```
{ name:
```

```
"Samlan"
```

```
, id:
```

```
"2"
```

```
},
```

```
{ name:
```

```
"Shahrukh"
```

```
, id:
```

```
"0"
```

```
},
```

```
];
```

```
console
```

```
.log(filterObject(arr,
```

```
0
```

```
));
```

```
console
```

```
.log(filterObject(arr,
```

```
"Amir"
```

```
));
```

```
console
```

```
.log(filterObject(arr,
```

```
"0"
```

```
));
```

```
console
```

```
.log(filterObject(arr,
```

```
"-1"
```

```
));
```

Output:

```
// { name: "Amir", id: "1" }
```

```
// { name: "Amir", id: "1" }
```

```
// { name: "Shahrukh", id: "0" }
```

```
// undefined
```

256

Aggregate array of objects on the given keys

Problem Statement -

Given an array of objects and two keys “on” and “who”, aggregate the “who” values on the “on” values.

Example

Input:

```
const
```

```
endorsements = [
```

```
{ skill:
```

```
'css'
```

```
, user:
```

```
'Bill'
```

```
},
```

```
{ skill:
```

```
'javascript'
```

```
, user:
```

```
'Chad'
```

```
},
```

```
{ skill:
'javascript'
, user:
'Bill'
},
{ skill:
'css'
, user:
'Sue'
},
{ skill:
'javascript'
, user:
'Sue'
},
{ skill:
'html'
, user:
'Sue'
}
```

```
];  
  
console  
.log(aggregate(endorsements,  
"user"  
  
,  
"skill"  
));
```

Output:

```
[  
  
  {  
  
    "user"  
  
    :  
  
    "Bill"  
  
    ,  
  
    "skill"  
  
    : [  
  
      "css"  
  
      ,  
  
      "javascript"  
  
    ]  
  }  
]
```

},

{

"user"

:

"Chad"

,

"skill"

:[

"javascript"

]

},

{

"user"

:

"Sue"

,

"skill"

:[

"css"

,



```
"javascript"
```

```
,
```

```
"html"
```

© JavaScript Interview Guide | learnersbucket.com

```
257
```

```
]
```

```
}
```

```
]
```

We can use [Array.reduce\(\)](#) to aggregate the values of the array of objects. All we have to do is,

- Get the value of the “ on ” key and aggregate the values of the “ who ” key in the format in which we have to return output.
- Then only return the values from the aggregation as we are expecting an array of objects as output .

```
const
```

```
aggregate = (
```

```
arr, on, who
```

```
) => {
```

```
// using reduce() method to aggregate
```

```
const
```

```
agg = arr.reduce((
```

a, b

) => {

// get the value of both the keys

const

onValue = b[on];

const

whoValue = b[who];

// if there is already a key present

// merge its value

if

(a[onValue]){

a[onValue] = {

[on]: onValue,

[who]: [...a[onValue][who], whoValue]

}

}

// create a new entry on the key

else

{

a[onValue] = {

```
[on]: onValue,
```

```
[who]: [whoValue]
```

```
}
```

```
}
```

```
// return the aggregation
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

258

```
return
```

```
a;
```

```
}, {});
```

```
// return only values after aggregation
```

```
return
```

```
Object
```

```
.values(agg);
```

```
}
```

Test Case

Input:

```
const
```

```
endorsements = [
```

```
{ skill:
```

'css'

, user:

'Bill'

},

{ skill:

'javascript'

, user:

'Chad'

},

{ skill:

'javascript'

, user:

'Bill'

},

{ skill:

'css'

, user:

'Sue'

},

{ skill:

```
'javascript'
```

```
, user:
```

```
'Sue'
```

```
},
```

```
{ skill:
```

```
'html'
```

```
, user:
```

```
'Sue'
```

```
}
```

```
];
```

```
console
```

```
.log(aggregate(endorsements,
```

```
"skill"
```

```
,
```

```
"user"
```

```
));
```

```
Output:
```

```
[{
```

```
"skill"
```

```
:
```

"css"

,

"user"

:[

"Bill"

,

"Sue"

]

},

{

"skill"

:

"javascript"

,

"user"

:[

"Chad"

,

"Bill"

,

```
"Sue"
```

```
]
```

```
},
```

```
{
```

```
"skill"
```

```
:
```

```
"html"
```

```
,
```

```
"user"
```

```
: [
```

```
"Sue"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

259

```
]
```

```
}]
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

260

Convert entity relation array to ancestry tree

Problem Statement -

Given an array with two entries, parent and child relation, convert the array to a relation string (parent -> child -> grandchild).

The input array will contain relations for many ancestries in random order, We must return the array of strings representing different relationships.

For example, in the below case, the topmost ancestor is an animal.

Input:

```
[
  [
    "lion"
  ],
  [
    "cat"
  ],
  [
    "cat"
  ],
  [
    "mammal"
  ],
  [
    "dog"
  ],
  [
    "mammal"
  ],
  [
```



"mammal"

,

"animal"

],

[

"fish"

,

"animal"

],

[

"shark"

,

"fish"

],

];

Output:

[

"animal -> mammal -> cat -> lion"

,

"animal -> mammal -> cat"

```
,  
"animal -> mammal -> dog"  
  
,  
"animal -> mammal"  
  
,  
"animal -> fish"  
  
,  
"animal -> fish -> shark"  
  
]
```

To solve this, the first thing we have to do is to convert the array to an object of the parent-child relationship for better processing.

```
// aggregate parent / child relation
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

261

```
const
```

```
aggregate = (  
arr
```

```
) => {
```

```
// aggregate the values for easier processing
```

```
return
```

```
arr.reduce((
```

a, b

) => {

const

[child, parent] = b;

// aggregating on child

a[child] = parent;

return

a;

}, {});

};

const

arr = [

[

"lion"

,

"cat"

],

[

"cat"

,

"mammal"

],

[

"dog"

,

"mammal"

],

[

"mammal"

,

"animal"

],

[

"fish"

,

"animal"

],

[

"shark"

,

```
"fish"  
  
],  
  
];  
  
console  
  
.log(aggregate(arr));  
  
/*  
  
{  
  
  "lion": "cat",  
  
  "cat": "mammal",  
  
  "dog": "mammal",  
  
  "mammal": "animal",  
  
  "fish": "animal",  
  
  "shark": "fish"  
  
}  
  
*/
```

We have aggregated the values on the child as one child will have only one parent, this way a single key-value pair is formed which will be faster to process and in forming relationships.

Now, all we have to do is keep on traversing this map and form the relation string. For this, we will get the value of the current key and

262

recursively call the same function with this value and the map to get its ancestry and so on.

```
// for a relationship from the aggregated value
```

```
const
```

```
convert = (
```

```
  obj
```

```
) => {
```

```
  return
```

```
    Object
```

```
      .keys(obj).reduce((
```

```
        a, b
```

```
) => {
```

```
  a.push(getKey(obj, b));
```

```
  return
```

```
    a;
```

```
  }, []);
```

```
};
```

```
// helper function to form the string
```

```
// till the last hierarchy
```

```
const
```

```
getKey = (  
  obj, key  
) => {  
  // access the  
  
  const  
  
  val = obj[key];  
  
  // the formation can be reversed by changing the  
  order of the keys  
  
  // child -> parent | parent -> child  
  
  if  
  
  (val  
  
  in  
  
  obj){  
    return  
  
    getKey(obj, val) +  
  
    " -> "  
  
    + key;  
  
  }  
  
  else  
  
  {
```

```
return
```

```
val +
```

```
" -> "
```

```
+ key;
```

```
}
```

```
};
```

```
// map after aggregation
```

```
const
```

```
map = {
```

```
"lion"
```

```
:
```

```
"cat"
```

```
,
```

```
"cat"
```

```
:
```

```
"mammal"
```

```
,
```

```
"dog"
```

```
:
```

```
"mammal"
```



```
,  
"mammal"  
:  
"animal"  
  
,  
"fish"  
:  
"animal"  
  
,  
"shark"  
:  
"fish"  
};  
console  
.log(convert(map));  
/*  
[
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

263

"animal -> mammal -> cat -> lion",

```
"animal -> mammal -> cat",  
"animal -> mammal -> dog",  
"animal -> mammal",  
"animal -> fish",  
"animal -> fish -> shark"  
]  
*/
```

Combining it together.

```
const  
ancestry = (  
  arr  
) => {  
  // aggregate parent / child relation  
  const  
  aggregate = (  
    arr  
  ) => {  
    // aggregate the values for easier processing  
    return  
    arr.reduce((
```

```
a, b
) => {
  const
  [child, parent] = b;
  // aggregating on child
  a[child] = parent;
  return
  a;
}, {});
};

// for a relationship from the aggregated value
const
convert = (
  obj
) => {
  return
  Object
  .keys(obj).reduce((
    a, b
  ) => {
```

```

a.push(getKey(obj, b));

return

a;

}, []);

};

// helper function to form the string

// till the last hierarchy

const

getKey = (

obj, key

) => {

// access the

const

val = obj[key];

```

© JavaScript Interview Guide | learnersbucket.com

264

```

// the formation can be reversed by changing the order of the keys

// child -> parent | parent -> child

if

(val

```

```
in
obj){
return
getKey(obj, val) +
" -> "
+ key;
}
else
{
return
val +
" -> "
+ key;
}
};

// get the aggregated map
const
aggregatedMap = aggregate(arr);

// return the ancestry

return
```

```
convert(aggregatedMap);
```

```
};
```

Test Case

Input:

```
const
```

```
arr = [
```

```
[
```

```
"lion"
```

```
,
```

```
"cat"
```

```
],
```

```
[
```

```
"cat"
```

```
,
```

```
"mammal"
```

```
],
```

```
[
```

```
"dog"
```

```
,
```

```
"mammal"
```

```
],  
[  
  "mammal"  
  ,  
  "animal"  
],  
[  
  "fish"  
  ,  
  "animal"  
],  
[  
  "shark"  
  ,  
  "fish"  
],  
];  
  
console  
.log(ancestry(arr));
```

Output:

```
[  
  "animal -> mammal -> cat -> lion"  
  
  ,  
  
  "animal -> mammal -> cat"  
  
  ,  
  
  "animal -> mammal -> dog"  
  
  ,  
  
  "animal -> mammal"  
  
  ,  
  
  "animal -> fish"  
  
  ,  
  
  "animal -> fish -> shark"  
  
]
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

265

Get object value from string path

Problem Statement -

Implement a method in Javascript that will take an object and a string or array of strings as a path and return the value at that path. If

nothing is found, return undefined .

This is basically to write polyfill for [lodash. \\_get\(\)](https://lodash.com/docs/4.17.15#_get) .



## Example

Input:

```
const
```

```
obj = {
```

```
  a: {
```

```
    b: {
```

```
      c: [
```

```
        1
```

```
        ,
```

```
        2
```

```
        ,
```

```
        3
```

```
      ]
```

```
    }
```

```
  }
```

```
};
```

```
console
```

```
.log(get(obj,
```

```
'a.b.c'
```

```
));
```

```
console
```

```
.log(get(obj,
```

```
'a.b.c.0'
```

```
));
```

```
console
```

```
.log(get(obj,
```

```
'a.b.c[1]'
```

```
));
```

```
console
```

```
.log(get(obj,
```

```
'a.b.c[3]'
```

```
));
```

Output:

```
// [1,2,3]
```

```
// 1
```

```
// 2
```

```
// undefined
```

We can implement a function that will handle the following cases.

- Check the type of path, if it is an array of strings, concatenate it and form a string.

- Traverse the string and get the actual values from it ignoring the special characters like [, ], and dot.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

266

- Once we have the exact path, deeply traverse the input nested object to find the value.

```
const
```

```
get = (
```

```
obj, path
```

```
) => {
```

```
// if path is not a string or array of string
```

```
if
```

```
(path ===
```

```
"
```

```
|| path.length ==
```

```
0
```

```
)
```

```
return
```

```
undefined
```

```
;
```

```
// if path is an array, concatenate it and form
```

```
a string

// to handle a single case of string

if

(

Array

.isArray(path)) path = path.join(

'

');

// filter out the brackets and dot

let

exactPath = [];

for

(

let

i =

0

; i < path.length; i++) {

if

(path[i] !==

'['
```

```
&& path[i] !==  
''  
&& path[i]  
!==  
''  
) {  
  exactPath.push(path[i]);  
}  
}  
  
// get the value of the path in the sequence  
const  
value = exactPath.reduce((  
  source, path  
) =>  
  source[path], obj);  
  
// if not found return undefined  
return  
value ? value :  
undefined  
;
```

```
};
```

Test Case

Input:

```
const
```

```
obj = {
```

```
  a: {
```

```
    b: {
```

```
      c: [
```

```
        1
```

```
        ,
```

```
        2
```

```
        ,
```

```
        3
```

```
      ]
```

```
    }
```

```
  }
```

```
};
```

```
console
```

```
.log(get(obj,
```

```
'a.b.c'
```

```
));
```

```
console
```

```
.log(get(obj,
```

```
'a.b.c.0'
```

```
));
```

```
console
```

```
.log(get(obj,
```

```
'a.b.c[1]'
```

```
));
```

```
console
```

```
.log(get(obj, [
```

```
'a'
```

```
,
```

```
'b'
```

```
,
```

```
'c'
```

```
,
```

```
'2'
```

```
]]);
```

```
console
```

```
.log(get(obj,  
'a.b.c[3]'  
));
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

267

console

```
.log(get(obj,  
'a.c'  
));
```

Output:

```
// [1,2,3]
```

```
// 1
```

```
// 2
```

```
// 3
```

```
// undefined
```

```
// 'bfe'
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

268

Set object value at the string path

Problem Statement -



Given an object, a path in the string or array of strings format, and a value, update the value at the given path in the object.

This is a polyfill for [lodash.set\(\)](#) method and is opposite of [lodash.get\(\)](#) method.

Example

```
const
object = {
  'a'
: [{
  'b'
: {
  'c'
:
3
} } ] };
set(object,
'a[0].b.c'
,
4
);
console
```

```
.log(object.a[
```

```
0
```

```
].b.c);
```

```
// 4
```

```
set(object, [
```

```
'x'
```

```
,
```

```
'0'
```

```
,
```

```
'y'
```

```
,
```

```
'z'
```

```
],
```

```
5
```

```
);
```

```
console
```

```
.log(object.x[
```

```
0
```

```
].y.z);
```

```
// 5
```

To implement this function, we will first check if the provided path is a string or an array of strings.

If it is a string then filter all the special characters like [ , ] and split the string on dot(.) to get all the path keys in an array.

Then using a helper function we can assign the value to the provided path.

- Get only the first key from the path array and aggregate the rest of the keys.
- If there are no more keys left to update, assign the value to the current key.

© JavaScript Interview Guide | learnersbucket.com

269

- Else recursively call the same function with the current value for the next path.
- While moving to the next path, check the type of key, if it is numeric, then value should be an array thus pass array, else if it is a string pass the object.

Note:- This will override the existing value and assign a new one.

const

helper = (

obj, path, value

) => {

// get the current and the remaining keys from

the path

let

```
[current, ...rest] = path;
```

```
// if there are more keys
```

```
// add the value as an object or array
```

```
// depending upon the typeof key
```

```
if
```

```
(rest.length >
```

```
0
```

```
){
```

```
// if there is no key present
```

```
// create a new one
```

```
if
```

```
(!obj[current]){
```

```
// if the key is numeric
```

```
// add an array
```

```
// else add an object
```

```
const
```

```
isNumber =
```

```
`${+rest[
```

```
0
  ]}`
  === rest[
0
  ];
  obj[current] = isNumber ? [] : {};
}
// recursively update the remaining path
// if the last path is not of object type
// but key is then
// create an object or array based on the
key
// and update the value
if
(
  typeof
  obj[current] !==
  'object'
){
  // determine if the key is string or numeric
```

```

const
isNumber =
`${+rest[
0
]}`
=== rest[
0
];
obj[current] = helper(isNumber ? [] : {}, rest, value)
}
// else directly update value
else
{
obj[current] = helper(obj[current], rest, value);
}
© JavaScript Interview Guide | learnersbucket.com
270
}
// else directly assign the value to the key
else

```

```
{  
  obj[current] = value;  
}  
  
// return the updated obj  
return  
obj;  
}  
  
const  
set = (  
  obj, path, value  
) => {  
  let  
    pathArr = path;  
    // if path is of string type  
    // replace the special characters  
    // and split the string on . to get the path keys  
    array  
    if  
    (  
      typeof
```

```
path ===  
'string'  
{  
  pathArr = path.replace(  
    '['  
    ,  
    '.'  
  ).replace(  
    ']'  
    ,  
    ''  
  ).split(  
    ''  
  );  
}
```

// use the helper function to update

```
helper(obj, pathArr, value);  
};
```

Test Case

Input:



```
const
```

```
abc = {
```

```
  a: {
```

```
    b: {
```

```
      c: [
```

```
        1
```

```
        ,
```

```
        2
```

```
        ,
```

```
        3
```

```
      ]
```

```
    },
```

```
    d: {
```

```
      a:
```

```
        "hello"
```

```
    }
```

```
  }
```

```
};
```

```
const
```

```
instance1 =
```

JSON

.parse(

JSON

.stringify(abc));

set(instance1,

'a.b.c'

,

'learnersbucket'

);

console

.log(instance1.a.b.c);

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

271

const

instance2 =

JSON

.parse(

JSON

.stringify(abc));

set(instance2,

```
'a.b.c.0'
```

```
,
```

```
'learnersbucket'
```

```
);
```

```
console
```

```
.log(instance2.a.b.c[
```

```
0
```

```
]);
```

```
const
```

```
instance3 =
```

```
JSON
```

```
.parse(
```

```
JSON
```

```
.stringify(abc));
```

```
set(instance3,
```

```
'a.b.c[1]'
```

```
,
```

```
'learnersbucket'
```

```
);
```

```
console
```

```
.log(instance3.a.b.c[
```

```
1
```

```
]);
```

```
const
```

```
instance4 =
```

```
JSON
```

```
.parse(
```

```
JSON
```

```
.stringify(abc));
```

```
set(instance4, [
```

```
'a'
```

```
,
```

```
'b'
```

```
,
```

```
'c'
```

```
,
```

```
'2'
```

```
],
```

```
'learnersbucket'
```

```
);
```

console

.log(instance4.a.b.c[

2

]);

const

instance5 =

JSON

.parse(

JSON

.stringify(abc));

set(instance5,

'a.b.c[3]'

,

'learnersbucket'

);

console

.log(instance5.a.b.c[

3

])

const

```
instance6 =  
JSON  
.parse(  
JSON  
.stringify(abc));  
set(instance6,  
'a.c.d[0]'  
,  
'learnersbucket'  
);  
// valid digits treated as array elements  
console  
.log(instance6.a.c.d[  
0  
]);  
const  
instance7 =  
JSON  
.parse(  
JSON
```

```
.stringify(abc));  
  
set(instance7,  
  'a.d.01'  
  ,  
  'learnersbucket'  
);  
  
// invalid digits treated as property string  
console  
  .log(instance7.a.d[  
    '01'  
  ]);  
  
const  
  object = {  
    'a'  
      : [{  
        'b'  
          : {  
            'c'  
              :  
                3
```

```
} } ] };
```

```
set(object,
```

```
'a[0].b.c'
```

```
,
```

```
4
```

```
);
```

```
console
```

```
.log(object.a[
```

```
0
```

```
].b.c);
```

```
set(object, [
```

```
'x'
```

```
,
```

```
'0'
```

```
,
```

```
'y'
```

```
,
```

```
'z'
```

```
],
```

```
5
```



```
);  
console  
.log(object.x[  
0  
].y.z);
```

Output:

```
"learnersbucket"  
"learnersbucket"  
"learnersbucket"  
"learnersbucket"  
"learnersbucket"  
"learnersbucket"  
"learnersbucket"  
4
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

272

5

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

273

Implement JSON stringify method.

## Problem Statement -

Implement a simple polyfill for [JSON.stringify\(\)](#) in JavaScript.

### Example

console

.log(

JSON

.stringify([ { x:

5

, y:

6

});

// expected output: "[{"x":5,"y":6}]"

JSON.stringify() converts almost each javascript value to a string, except for a few.

To implement this, we will break down problems into two subproblems and tackle them separately.

First, determine the type of value and accordingly convert it to the string.

- For function , symbol , undefined return "null".
- For number , [if the value is finite](#) return the value as it is else return "null".

- For boolean return it as it is and for string return the value in double quotes.
- The last thing left is the object, there are multiple cases to handle for the object.
- If it is date , convert it to ISO string.
- If it is a constructor of String, Boolean, or Number, convert it to those values only.
- If it is an array, convert each value of the array and return it.
- If it is a nested object, recursively call the same function to stringify further.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

274

// helper method

// handle all the value types

// and stringify accordingly

static

value(val) {

switch

(

typeof

val) {

case

'boolean'

:

case

'number'

:

// if the value is finite number return the

number as it is

// else return null

return

isFinite

(val) ?

`\${val}`

:

`null`

;

case

'string'

:

return

```
`"${val}"`  
  
;  
  
// return null for anything else  
  
case  
  
'function'  
  
:  
  
case  
  
'symbol'  
  
:  
  
case  
  
'undefined'  
  
:  
  
return  
  
'null'  
  
;  
  
// for object, check again to determine the  
object's actual type  
  
case  
  
'object'  
  
:
```

```
// if the value is date, convert date to string
if
(
val
instanceof
Date
) {
return
`${val.toISOString()}`
;
}

// if value is a string generated as constructor,
// new String(value)
else
if
(
val.constructor ===
String
){
return
`${val}`
;
}
```

```

}

// if value is a number or boolean generated
as constructor,

// new String(value), new Boolean(true)

else

if

(val.constructor ===

Number

|| val.constructor

===

Boolean

){

return

isFinite

(val) ?

`${val}`

:

`null`

;

}

```

```
// if value is a array, return key values
```

```
as
```

```
// string inside [] brackets
```

```
else
```

```
if
```

```
(
```

```
Array
```

```
.isArray(val)) {
```

```
return
```

```
`[${val.map(value =>
```

```
this
```

```
.value(value)).join(',')}]`
```

```
;
```

```
}
```

```
// recursively stringify nested values
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
275
```

```
return
```

```
this
```

```
.stringify(val);
```



```
}
```

```
}
```

Second, we will handle some base cases like, if it is a null value, return

'null ', if it is an object, get the appropriate value from the above method. At the end wrap the value inside curly braces and return

them.

```
// main method
```

```
static
```

```
stringify(obj) {
```

```
// if value is not an actual object, but it is
```

```
undefined or an array
```

```
// stringify it directly based on the type of
```

```
value
```

```
if
```

```
(
```

```
typeof
```

```
obj !==
```

```
'object'
```

```
|| obj ===
```

```
undefined
```

```
|| obj
```

```
instanceof  
Array  
) {  
    return  
    this  
    .value(obj);  
}  
// if value is null return null  
else  
if  
(obj ===  
null  
) {  
    return  
    `null`  
;  
}  
// remove the cycle of object  
// if it exists  
this
```

```
.removeCycle(obj);

// traverse the object and stringify at each level

let

objString =

Object

.keys(obj).map((

k

) => {

return

(

typeof

obj[k] ===

'function'

) ?

null

:

`${k}: ${

this

.value(obj[k])}`

;

}
```

```
});  
  
// return the stringified output  
  
return  
  
`${objString}`  
  
;  
  
}
```

The final case to handle the circular object, for that we will be using the [removeCycle\(obj\)](#) method to remove the cycle from the objects.

© JavaScript Interview Guide | learnersbucket.com

276

```
// helper method to remove cycle  
  
static  
  
removeCycle = (  
  
obj  
  
) => {  
  
//set store  
  
const  
  
set =  
  
new  
  
WeakSet  
  
([obj]);
```

```
//recursively detects and deletes the object
```

```
references
```

```
(
```

```
function
```

```
iterateObj
```

```
(
```

```
obj
```

```
) {
```

```
for
```

```
(
```

```
let
```

```
key
```

```
in
```

```
obj) {
```

```
// if the key is not present in prototype
```

```
chain
```

```
if
```

```
(obj.hasOwnProperty(key)) {
```

```
if
```

```
(
```

```
typeof
obj[key] ===
'object'
){
// if the set has object reference
// then delete it
if
(set.has(obj[key])){
delete
obj[key];
}
else
{
//store the object reference
set.add(obj[key]);
//recursively iterate the
next objects
iterateObj(obj[key]);
}
}
```

```
}
```

```
}
```

```
})(obj);
```

```
}
```

Complete code

```
class
```

```
JSON
```

```
{
```

```
// main method
```

```
static
```

```
stringify(obj) {
```

```
// if value is not an actual object, but it is
```

```
undefined or an array
```

```
// stringify it directly based on the type of
```

```
value
```

```
if
```

```
(
```

```
typeof
```

```
obj !==
```

```
'object'
```

```
|| obj ===
```

```
undefined
```

```
|| obj
```

```
instanceof
```

```
Array
```

```
) {
```

```
return
```

```
this
```

```
.value(obj);
```

```
}
```

```
// if value is null return null
```

```
else
```

```
if
```

```
(obj ===
```

```
null
```

```
) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

277

```
return
```

```
`null`
```



```
;
}

// remove the cycle of object

// if it exists

this

.removeCycle(obj);

// traverse the object and stringify at each level

let

objString =

Object

.keys(obj).map((

k

) => {

return

(

typeof

obj[k] ===

'function'

)?

null
```

```
:  
    `"${k}": ${  
    this  
    .value(obj[k])}`  
;  
});  
  
// return the stringified output  
return  
    `${objString}`  
;  
}  
  
// helper method  
  
// handle all the value types  
// and stringify accordingly  
  
static  
value(val) {  
    switch  
    (  
        typeof  
        val) {
```

case

'boolean'

:

case

'number'

:

// if the value is finite number return the

number as it is

// else return null

return

isFinite

(val) ?

`\${val}`

:

`null`

;

case

'string'

:

return

```
`"${val}"`  
  
;  
  
// return null for anything else  
  
case  
  
'function'  
  
:  
  
case  
  
'symbol'  
  
:  
  
case  
  
'undefined'  
  
:  
  
return  
  
'null'  
  
;  
  
// for object, check again to determine the  
object's actual type  
  
case  
  
'object'  
  
:
```

```
// if the value is date, convert date to string
```

```
if
```

```
(val
```

```
instanceof
```

```
Date
```

```
) {
```

```
return
```

```
`"${val.toISOString()}"`
```

```
;
```

```
}
```

```
// if value is a string generated as constructor,
```

```
// new String(value)
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
278
```

```
else
```

```
if
```

```
(val.constructor ===
```

```
String
```

```
){
```

```
return
```

```

`"${val}"`

;

}

// if value is a number or boolean generated
as constructor,

// new String(value), new Boolean(true)

else

if

(val.constructor ===

Number

|| val.constructor

===

Boolean

){

return

isFinite

(val) ?

`${val}`

:

`null`

```

```
;
}

// if value is a array, return key values
// as string inside [] brackets

else

if

(

Array

.isArray(val)) {

return

`${val.map(value =>

this

.value(value)).join(',')}`

;

}

// recursively stringify nested values

return

this

.stringify(val);

}
```

```
}  
  
// helper method to remove cycle  
  
static  
  
removeCycle = (  
  
  obj  
  
  ) => {  
  
    //set store  
  
    const  
  
    set =  
  
    new  
  
    WeakSet  
  
    ([obj]);  
  
    //recursively detects and deletes the object  
    references  
  
    (  
  
    function  
  
    iterateObj  
  
    (  
  
      obj  
  
    ) {
```



```
for
(
let
key
in
obj) {
// if the key is not present in prototype
chain
if
(obj.hasOwnProperty(key)) {
if
(
typeof
obj[key] ===
'object'
){
// if the set has object reference
// then delete it
if
(set.has(obj[key])){
```

```
delete
obj[key];
}
else
{
//store the object reference
set.add(obj[key]);
//recursively iterate the
next objects
iterateObj(obj[key]);
}
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

279

```
}
}
})(obj);
}
};
```

Test Case

Input:

let

obj1 = {

a:

1

,

b: {

c:

2

,

d:

-3

,

e: {

f: {

g:

-4

,

},

},

```
h: {
```

```
  i:
```

```
    5
```

```
  ,
```

```
  j:
```

```
    6
```

```
  ,
```

```
},
```

```
}
```

```
}
```

```
let
```

```
obj2 = {
```

```
  a:
```

```
    1
```

```
  ,
```

```
  b: {
```

```
    c:
```

```
      'Hello World'
```

```
  ,
```

```
  d:
```

2

,

e: {

f: {

g:

-4

,

},

},

h:

'Good Night Moon'

,

},

}

// cricular object

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

280

const

List =

function

```
(  
  val  
) {  
  this  
    .next =  
    null  
  ;  
  this  
    .val = val;  
};  
const  
item1 =  
  new  
    List(  
      10  
    );  
const  
item2 =  
  new  
    List(  

```

20

);

const

item3 =

new

List(

30

);

item1.next = item2;

item2.next = item3;

item3.next = item1;

console

.log(

JSON

.stringify(item1));

console

.log(

JSON

.stringify(obj1));

console

```
.log(  
JSON  
.  
stringify(obj2));  
console  
  
.log(  
JSON  
.  
stringify([ { x:  
5  
, y:  
6  
} ]));  
// expected output: "[{"x":5,"y":6}]"  
console  
  
.log(  
JSON  
.  
stringify([  
new  
Number  
(  
3
```



```
),  
new  
String  
(  
'false'  
)  
new  
Boolean  
(  
false  
)  
new  
Number  
(  
Infinity  
))));  
// expected output: "[3,\"false\",false]"  
console  
.log(  
JSON
```

```

.stringify({ x: [
  10
,
  undefined
,
  function
(){}
,
  Symbol
(
  "
)]]));
// expected output: '{"x":[10,null,null,null]}'
console
.log(
JSON
.stringify({a:
Infinity
}));
Output:
"{'next': {'next': {'val': 30}, 'val': 20}, 'val': 10}"

```

```
"{'a': 1,'b': {'c': 2,'d': -3,'e': {'f': {'g': -4}},'h': {'i': 5,'j': 6}}}"
```

```
"{'a': 1,'b': {'c': 'Hello World','d': 2,'e': {'f': {'g': -4}},'h': 'Good Night Moon'}}"
```

```
"[{'x': 5,'y': 6}]"
```

```
"[3,'false',false,null]"
```

© JavaScript Interview Guide | learnersbucket.com

281

```
"{'x': [10,null,null,null]}"
```

```
"{'a': null}"
```

© JavaScript Interview Guide | learnersbucket.com

282

Implement JSON parse method.

Problem Statement -

Implement a simple polyfill for [JSON.parse\(\)](#) in JavaScript.

Example

```
const
```

```
json =
```

```
'{"result":true, "count":42}'
```

```
;
```

```
const
```

```
obj =
```

JSON

```
.parse(json);
```

console

```
.log(obj);
```

```
// expected output: {"result": true, "count": 42}
```

JSON.parse() method is exactly opposite of the [JSON.stringify\(\)](#). It takes a string as input and parses it to javascript value if possible, else throws an error.

This can be implemented in a step-by-step manner.

1. Remove all the spaces from the string.
2. If the string is empty or has invalid characters like starting with single quotes, then throw an error.
3. If the string has only [], {}, true, false, or number, convert and return the respective values.
4. The last part is to handle the nested values like arrays, objects, and arrays of objects. For this, we will first get their individual

values on comma separation by removing the braces (without []

and {}).

5. Then if it is an array, parse each value by calling the same

function recursively and return as an array, else if it is an object, split the values on : and parse the key as well as the value both and return them as object.

static

```
parse(string) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

283

```
// remove the space from the string
```

```
string = string.replace(
```

```
  / /g
```

```
,
```

```
"
```

```
);
```

```
//convert each value accordingly
```

```
switch
```

```
(string) {
```

```
  case
```

```
    "
```

```
  :
```

```
    throw
```

```
    new
```

```
    Error
```

```
    ();
```

```
  case
```

'null'

:

return

null

;

case

'{'

:

return

{};

case

'[]'

:

return

[];

case

'true'

:

return

true

```
;
case
'false'
:
return
false
;
default
:
// if number return as number
if
(+string === +string) {
return
Number
(string);
}
// if escaped single quotes, throw error
else
if
(string[
```

```
0
] ===
\"
) {
throw
new
Error
();
}
// if escaped double quotes, throw error
else
if
(string[
0
] ===
\"")
) {
// same as string.substr(1, string.length-2);
return
string.slice(
```



```
1
,
-1
);
}
else
{
// if [] || {}
// get the inner string
const
innerString = string.slice(
1
,
-1
);
// get the values from the string
// array of pairs if {}
// array of single values if []
const
subStrings =
```

```
this
.stringSplitByComma(innerString);
// if it is array
if
(string[
0
] ===
'['
) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

284

```
// parse each value
return
subStrings.map(
item
=>
this
.parse(item));
}
else
```

```
if
(string[
0
] ===
'{'
) {
// if it object
// get the key and value by splitting
on :
// parse the key and value individually
return
subStrings.reduce((
acc, item
) =>
{
if
(item.indexOf(
':'
) >
-1
```

```
) {  
  const  
  index = item.indexOf(  
    ':'  
  );  
  const  
  thisKey = item.substring(  
    0  
    ,  
    index);  
  const  
  thisValue = item.substring(index  
    +  
    1  
  );  
  acc[  
    this  
      .parse(thisKey)] =  
    this  
      .parse(thisValue);
```

```

}
return
acc;
}, {});
}
}
}
}
}
}

```

To get the comma-separated values from the array or the object, we will be using the sliding window technique.

1. Track the opening and closing parentheses [] as well as the curly braces {} .
2. Whenever a comma is spotted while traversing the string, get the value between the opening and closing parentheses and/or curly

braces and push it into an array so that each of these values can be parsed individually.

```
// helper function
```

```
// to get the comma separated values of array or
objects
```

```
static
```

```
stringSplitByComma(string) {
```

```
const
```

```
allStrs = [];
```

```
// lParen tracks the parentheses []
```

```
// lCurly tracks the curly braces {}
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
285
```

```
let
```

```
lParen =
```

```
0
```

```
, lCurly =
```

```
0
```

```
;
```

```
let
```

```
left =
```

```
0
```

```
, right =
```

```
0
```

```
;
```

```
// traverse the string
```

```
// whenever a comma is spotted
```

```
// store the value

while

(right <= string.length) {

const

rChar = string[right];

// track the index for the content

// inside the array [] or object {}

if

(rChar ===

'['

) lParen++;

if

(rChar ===

'{'

) lCurly++;

if

(rChar ===

']'

) lParen--;

if
```

```

(rChar ===
  '}'
) lCurly--;
// if a comma is spotted
if
((rChar ===
  ','
  && lParen ===
  0
  && lCurly
    ===
  0
  ) ||
right === string.length)
{
  // get the value in between and store it
  const
  thisStr = string.substring(left, right);
  allStrs.push(thisStr);
  left = right +

```



```
1  
;  
}  
right++;  
}  
return  
allStrs;  
}
```

Complete code

```
class  
JSON  
{  
static  
parse(string) {  
// remove the space from the string  
string = string.replace(  
/ /g  
,  
"  
);
```

```
// convert each value accordingly
```

```
switch
```

```
(string) {
```

```
case
```

```
"
```

```
:
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
286
```

```
throw
```

```
new
```

```
Error
```

```
();
```

```
case
```

```
'null'
```

```
:
```

```
return
```

```
null
```

```
;
```

```
case
```

```
'{'
```

:

return

{};

case

'[]'

:

return

[];

case

'true'

:

return

true

;

case

'false'

:

return

false

;

```
default
:
// if number return as number
if
(+string === +string) {
return
Number
(string);
}
// if escaped single quotes, throw error
else
if
(string[
0
] ===
"\")
) {
throw
new
Error
```

```
);  
}  
// if escaped double quotes, throw error  
else  
if  
(string[  
0  
] ===  
"\"")  
{  
// same as string.substr(1, string.length-2);  
return  
string.slice(  
1  
,  
-1  
);  
}  
else  
{
```

```
// if [] || {}

// get the inner string

const

innerString = string.slice(

1

,

-1

);

// get the values from the string

// array of pairs if {}

// array of single values if []

const

subStrings =

this

.stringSplitByComma(innerString);

// if it is array

if

(string[

0

] ===
```

```
'['  
) {  
    // parse each value  
    return  
    subStrings.map(  
        item  
        =>  
        this  
        .parse(item));  
    }  
    else  
    if  
    (string[  
        0  
        ] ===  
        '{'  
    ) {  
        // if it object  
        // get the key and value by splitting  
        on :
```

287

```
// parse the key and value individually
```

```
return
```

```
subStrings.reduce((
```

```
acc, item
```

```
) =>
```

```
{
```

```
if
```

```
(item.indexOf(
```

```
':'
```

```
) >
```

```
-1
```

```
) {
```

```
const
```

```
index = item.indexOf(
```

```
':'
```

```
);
```

```
const
```

```
thisKey = item.substring(
```



```
0
,
index);
const
thisValue = item.substring(index
+
1
);
acc[
this
.parse(thisKey)] =
this
.parse(thisValue);
}
return
acc;
}, {});
}
}
}
```

```
}
```

```
// helper function
```

```
// to get the comma separated values of array or  
objects
```

```
static
```

```
stringSplitByComma(string) {
```

```
const
```

```
allStrs = [];
```

```
// lParen tracks the parentheses []
```

```
// lCurly tracks the curly braces {}
```

```
let
```

```
lParen =
```

```
0
```

```
, lCurly =
```

```
0
```

```
;
```

```
let
```

```
left =
```

```
0
```

```
, right =
```

0

;

// traverse the string

// whenever a comma is spotted

// store the value

while

(right <= string.length) {

const

rChar = string[right];

// track the index for the content

// inside the array [] or object {}

if

(rChar ===

'['

) lParen++;

if

(rChar ===

'{'

) lCurly++;

if

```

(rChar ===
']'
) lParen--;
if
(rChar ===
'}'
) lCurly--;
// if a comma is spotted
if
((rChar ===
','
&& lParen ===
0
&& lCurly
===
0
) ||
right === string.length)
{

```

288

```
// get the value in between and store it
```

```
const
```

```
thisStr = string.substring(left, right);
```

```
allStrs.push(thisStr);
```

```
left = right +
```

```
1
```

```
;
```

```
}
```

```
right++;
```

```
}
```

```
return
```

```
allStrs;
```

```
}
```

```
}
```

Test Case

```
console
```

```
.log(
```

```
JSON
```

```
.parse(
```

```
'[{"result":true,"count":42}]'
```

```
));
```

```
/*
```

```
[
```

```
{
```

```
"result": true,
```

```
"count": 42
```

```
}
```

```
]
```

```
*/
```

```
console
```

```
.log(
```

```
JSON
```

```
.parse(
```

```
'{"next": {"next": {"val":
```

```
30},"val": 20},"val": 10
```

```
}'
```

```
));
```

```
/*
```

```
{
```

```

"next": {
  "next": {
    "val": 30
  },
  "val": 20
},
"val": 10
}
*/

console
.log(
JSON
.parse(
'{"a": 1,"b": {"c": 2,"d":
-3,"e": {"f": {"g":
-4}}, "h": {"i": 5,"j": 6}}}'
));

/*

```

```
{  
  "a": 1,  
  "b": {  
    "c": 2,  
    "d": -3,  
    "e": {  
      "f": {  
        "g": -4  
      }  
    },  
    "h": {  
      "i": 5,  
      "j": 6  
    }  
  }  
}  
*/
```

console

.log(

JSON



```
.parse(
'{"a": 1,"b": {"c": "Hello
World","d": 2,"e": {"f":
{"g": -4}}},"h": "Good Night Moon"}}'
));

/*
{
"a": 1,
"b": {
"c": "HelloWorld",
"d": 2,
"e": {
"f": {
"g": -4
}
},
"h": "GoodNightMoon"
}
}
*/
```

```
console
```

```
.log(
```

```
JSON
```

```
.parse(
```

```
'[{"x": 5,"y": 6}]'
```

```
));
```

```
/*
```

```
[
```

```
{
```

```
"x": 5,
```

```
"y": 6
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
290
```

```
}
```

```
]
```

```
*/
```

```
console
```

```
.log(
```

```
JSON
```

```
.parse(
```

```
'[3,"false",false,null]'
```

```
));
```

```
/*
```

```
[
```

```
3,
```

```
"false",
```

```
false,
```

```
null
```

```
]
```

```
*/
```

```
console
```

```
.log(
```

```
JSON
```

```
.parse(
```

```
'{"x": [10,null,null,null]}'
```

```
));
```

```
/*
```

```
{
```

```
"x": [
```

```
10,
```

```
null,  
null,  
null  
]  
}  
*/  
  
console  
  
.log(  
JSON  
  
.parse(  
'['  
));  
  
/*  
  
[]  
  
*/
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

291

HTML encoding of a string.

Problem Statement -

Given a string and an array representing the HTML encoding of the

string from and to with the given tag. Return the HTML encoded string.

Example

Input:

```
const
```

```
str =
```

```
'Hello, world'
```

```
;
```

```
const
```

```
styleArr = [[
```

```
0
```

```
,
```

```
2
```

```
,
```

```
'i'
```

```
], [
```

```
4
```

```
,
```

```
9
```

```
,
```

'b'

], [

7

,

10

,

'u'

]];

Output:

'<i>Hel</i>l<b>o, w<u>orl</u></b><u>d</u>'

Note – <u> tag gets placed before the <b> tag and after it as the insertion index overlaps it.

[It is one of the most common features that is implemented in the](#)

[WYSIWYG](#) editors, where you write normal text and apply styles to it and it will be converted to HTML encoding at the end.

There are two ways to solve this question.

- First one is extremely simple as most of the work is done by an

inbuilt method from the JavaScript.

- In the second one, we write the complete logic for encoding.

Let us first see the second method as most of the times in the interview you will be asked to implement this way only.

Custom function for HTML encoding of the string

The styles can be overlapping thus one tag can overlap the other and in such scenarios we have to close the overlapping tags and re-open it again.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

292

Input:

```
const
```

```
str =
```

```
"Hello, World"
```

```
;
```

```
const
```

```
style = [
```

```
0
```

```
,
```

```
2
```

```
,
```

```
'i'
```

```
], [
```

```
1
```

```
,
```

```
3
```

,

'b'

];

Output:

<i>

H

<b>

el

</b></i><b>

l

</b>

o, World

// b is starting from 1 and ending at 3, i is in between b.

As you can see in the above example b is overlapping thus it is closed at 2nd character of the string and re-opened at 3.

We can solve this with the help of a [priority\\_queue](#) and a [stack](#) .

- Aggregate the styles on the starting index in the order of priority of the range difference between them. (endIndex – startIndex),

this way the longest tag is created first.

- Now using a stack, track the opening and closing of tags.



- If the current style's ending index is greater than the previous one, then create a new tag as it is overlapping and push this new

entry back to the stack and in the next iteration create this tag.

- At the end return the string of the top most stack entry.

## Priority queue

We are not going to use the complete implementation of Priority

Queue, rather a helper function that adds a new item in the array and sorts it in the ascending or descending order depending upon the

order.

```
// helper function works as Priority queue
```

```
// to add tags and sort them in descending order
```

```
// based on the difference between the start and end
```

```
function
```

```
addAndSort
```

```
(
```

```
track, index, data
```

```
) {
```

```
if
```

```
(!track[index]) track[index] = [];
```

```
track[index] = [...track[index], data];
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

293

```
track[index].sort((  
  a, b  
  ) => a.getRange() > b.getRange());  
};
```

Stack

A simple implementation of the Stack data structure with the required methods.

function

Stack

() {

let

items = [];

let

top =

0

;

//Push an item in the Stack

this

.push =

function

```
(  
  element  
) {  
  items[top++] = element;  
};  
  
//Pop an item from the Stack  
  
this  
  .pop =  
  function  
  () {  
  return  
  items[--top];  
};  
  
//Peek top item from the Stack  
  
this  
  .peek =  
  function  
  () {  
  return  
  items[top -
```

```
1
```

```
];
```

```
};
```

```
};
```

Tag method

A helper function to create a new tag for each entry of styles and its corresponding, also helps to get the range for easier processing in the priority queue. We push this in the priority queue and sort it.

```
// helper function to form a tag
```

```
// and trace the string
```

```
function
```

Tag

```
(
```

```
start, end, tag
```

```
) {
```

```
this
```

```
.start = start;
```

```
this
```

```
.end = end;
```

```
this
```

```
.tag = tag;
```

```
this
```

```
.text =
```

```
""
```

```
;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

294

```
this
```

```
.getRange =
```

```
()
```

```
=> {
```

```
return
```

```
this
```

```
.end -
```

```
this
```

```
.start;
```

```
};
```

```
};
```

Encoder (main) method

In this method we perform all the encoding step by step.

- Create an empty array trace of the size of the input string.

- Iterate the styles and form a new tag for each entry.
- On the start index of the styles aggregate the common tags in the priority queue based on the difference of their range in descending order.
- Add these priority queues at the start indexes of the styles in the trace .
- Create a new stack and add an initial Tag with maximum range  
i.e Tag(start = 0, end = Number.MAX\_VALUE, tag = "") .
- Iterate till input string length, get the tags from the trace at each index, iterate the prioritized tags if they are present.
- Create the opening HTML tag for each tag in the queue and push it in the stack. Check if the current end Index of the tag is greater than the previous one in the stack, then create a new tag and push it in the priority queue.
- At the end close all the HTML tags at the given index in the loop.

function

parse

(

str, markups

) {

// create an empty array for all the indexes of

the string

```
const
track =
new
Array
(str.length).fill(
null
);
// add the tag at the starting point
// of each text mentioned in the markups
for
(
let
markup
of
markups) {
const
[start, end, tag] = markup;
addAndSort(track, start,
new
Tag(start, end, tag));
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

295

```
// create a new stack
```

```
const
```

```
html =
```

```
new
```

```
Stack();
```

```
// initialize with a new Tag that has max range
```

```
and empty string
```

```
html.push(
```

```
new
```

```
Tag(
```

```
0
```

```
,
```

```
Number
```

```
.MAX_VALUE,
```

```
""
```

```
));
```

```
// iterate each character of the string
```



```
for
(
let
i =
0
; i < str.length; i++) {
// check for opening tags and add them
while
(track[i] && track[i].length >
0
) {
const
cur = track[i].shift();
cur.text =
`<${cur.tag}>`
;
// for example in [0, 2, 'i'] , [1, 3, 'b']
// b is starting from 1 and ending at 3, i is
in between b.
// <i> <b> </b> </i> <b> </b>
```

```
// if the end of the nested tag is larger than
the parent,
// split the tag
// and insert the remaining split to the bucket
after its parent
if
(cur.end > html.peek().end) {
  const
  split =
  new
  Tag(html.peek().end +
  1
  ,
  cur.end, cur.tag);
  cur.end = html.peek().end;
  addAndSort(track, html.peek().end +
  1
  , split);
}
// push the new tag
```

```
html.push(cur);

}

// add the current character to the currently
topmost tag

html.peek().text += str[i];

// Check for closing tags and close them.

while

(html.peek().end === i) {

html.peek().text +=

`</${html.peek().tag}>`

;

const

temp = html.pop().text;

html.peek().text += temp;

}

}
```

// return the topmost

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

return

```
html.pop().text;
```

```
};
```

Test Case

Input:

```
const
```

```
encoded = parse(
```

```
'Hello, World'
```

```
,
```

```
[[
```

```
0
```

```
,
```

```
2
```

```
,
```

```
"i"
```

```
],
```

```
[
```

```
7
```

```
,
```

```
10
```

```
,
```

"u"

],

[

4

,

9

,

"b"

],

[

2

,

7

,

"i"

],

[

7

,

9

,

"u"

ll);

console

.log(encoded);

Output:

"<i>He<i>l</i></i><i>l<b>o,

<u><u>W</u></u></b></i><b><u><u>or</u></u></b><u>l</u>d"

"

***Hello,***

**W**

[OceanofPDF.com](http://OceanofPDF.com)

**or**

**l**

d"

Using [DOMParser\(\)](#)

DOMParser() parses the HTML source code from the string and the good thing is it appropriately places the opening and closing tags if it is missing.

Thus all we have to do is traverse the styles and add tags at the mentioned opening and closing positions in the input string.

Then pass this string to the DOMParser() and it will generate the HTML from the string.

function

parse

(

string, markups

) {

// place the opening and closing tags at the appropriate

indexes

const

```
fragments = markups.reduce((  
  chars, [start,  
  end, tag]  
) => {  
  chars[start] =  
    `<${tag}>`  
  + chars[start];  
  chars[end] +=  
    ``  
  ;  
  ;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

297

```
return  
chars;  
}, [...string]);  
  
// pass this string to DOMParser()  
  
// to convert it to HTML  
  
return  
  
new  
  
DOMParser()
```



```
.parseFromString(fragments.join(
"
),
'text/html'
)
.body.innerHTML;
}
```

Test Case

Input:

```
const
encoded = parse(
'Hello, World'
,
[[
0
,
2
,
'i'
],
```

[

7

,

10

,

"u"

],

[

4

,

9

,

"b"

],

[

2

,

7

,

"i"

],

[

7

,

9

,

"u"

]);

console

.log(encoded);

Output:

"<i>He<i>l</i>l<b>o,

<u><u>W</u></u></b></i><b><u><u>or</u></u></b><u>l</u>d"

"

***Hello,***

**W**

**or**

**l**

d"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

298

CSS selector generator

Problem Statement -

Given a root node and target node, generate a CSS selector from the root to the target. Provide an exact selection.

Example

Input:

```
<div id=
```

```
"root"
```

```
>
```

```
<article>
```

Prepare for interview

```
</article>
```

```
<section>
```

on

<p>

<span>

Learnersbucket

<button>

click me!

</button>

<button id=

"target"

>

click me!

</button>

</span>

</p>

</section>

</div>

Output:

"div[id='root'] > section:nth-child(2) > p:nth-child(1) > span:nth-child(1)

> button:nth-child(2)"

To generate the CSS selector, all we have to do is start from the target and trace back to the root (parent).

- Use a while loop and keep iterating till we have found the root of the target.
- In each iteration get the index of the current child in its immediate parent to decide the nth-child position.
- At the end, add the roots tag name. The selector will begin from this.

© JavaScript Interview Guide | learnersbucket.com

299

const

generateSelector = (

root, target

) => {

// trace the selector from target to root

const

selectors = [];

// iterate till root parent is found

while

(target !== root) {

// get the position of the current element as

its parent child

// add 1 to it as CSS nth-child is not like an

array, it starts from 1.

```
const
nthChild =
Array
.from(target.parentNode.children).indexOf(target)
+
1
;

const
selector =
`${target.tagName.toLowerCase()}:nth-child(${nthChild})`
;

// add the selector at the front
selectors.unshift(selector);

// move to the parent
target = target.parentNode;
}

// add the root's tag name at the beginning
// with your preferred selector
// id is used here
selectors.unshift(
```

```
`${target.tagName.toLowerCase()}[id="${target.id}"]`
```

```
);
```

```
// join the path of the selector and return them
```

```
return
```

```
selectors.join(
```

```
' > '
```

```
);
```

```
}
```

Test Case

Input:

```
<div id=
```

```
"root"
```

```
>
```

```
<article>
```

Prepare for interview

```
</article>
```

```
<section>
```

```
on
```

```
<p>
```

```
<span>
```



Learnersbucket

```
<button>
```

click me!

```
</button>
```

```
<button id=
```

```
"target"
```

```
>
```

click me!

```
</button>
```

```
</span>
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

300

```
</p>
```

```
</section>
```

```
</div>
```

```
const
```

```
root =
```

```
document
```

```
.getElementById(
```

```
"root"
```

```
);  
  
const  
  
target =  
  
document  
  
.getElementById(  
  
"target"  
  
);  
  
console  
  
.log(generateSelector(root, target));
```

Output:

```
"div[id='root'] > section:nth-child(2) > p:nth-child(1) > span:nth-child(1)  
> button:nth-child(2)"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

301

## Aggregate the Input values

### Problem Statement -

Given multiple input elements with different names and values inside a wrapper. Aggregate the values of the input on the names.

### Example

Input:

```
<form id="parent">
```

```
<input type="text" name="a.c" value="1"/>
<input type="text" name="a.b.d" value="2"/>
<input type="text" name="a.b.e" value="3"/>
</form>
```

Output:

```
{
  "a": {
    "c": "1",
    "b": {
      "d": "2",
      "e": "3"
    }
  }
}
```

The approach is simple to solve this, create a function that will accept the parent's id as input and get all the input elements under that parent.

Now using [Array.reduce\(\)](#) we can aggregate the values of each input on its key. All we have to do is,

- Split the name on the dot and get all the keys as an array.

- Check if the key already exists or not, if it does not create an empty object and assign it to it.
- If the key is the last from the array then assign the value to it.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

302

- At the end update the reference of this new object to the current object.

```
const
```

```
aggregateValues = (
```

```
id
```

```
) => {
```

```
// get the element
```

```
const
```

```
element =
```

```
document
```

```
.querySelector(
```

```
`#${id}`
```

```
);
```

```
// get all the input elements under it
```

```
// change the type depending on the requirement
```

```
// can take it as a argument
```

```
const
```

```
inputs = element.querySelectorAll(
  'input[type="text"]'
);

// aggregate the input values

return
Array
.from(inputs).reduce((
  prev, current
)
=> {
  // split the name and from an array of keys

  const
  names = current.name.split(
    "."
  );

  // store the previous object

  // for traversing the object

  // parent -> child -> grandchild

  let

  temp = prev;
```

```
// iterate each key in the name
names.forEach((
  name, index
) => {
  // if the key is not already present,
  // create an empty object
  if
    (!(name
      in
        temp)) {
    temp[name] = {};
  }
  // if the current key is the last one
  // assign the value to it
  if
    (index == names.length -
      1
    ) {
    temp[name] = current.value;
  }
}
```

```
// reference the next value to current
```

```
temp = temp[name];
```

```
});
```

```
// return the formed object
```

```
return
```

```
prev;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

303

```
}, {});
```

```
}
```

Test Case

Input:

```
<form id="parent">
```

```
<input type="text" name="a.c" value="1"/>
```

```
<input type="text" name="a.b.d" value="2"/>
```

```
<input type="text" name="a.b.e" value="3"/>
```

```
</form>
```

```
console.log(aggregateValues('parent'));
```

Output:

```
{
```

```
"a": {  
  "c": "1",  
  "b": {  
    "d": "2",  
    "e": "3"  
  }  
}  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

304

Fetch request and response Interceptors

Problem Statement -

Add a request and response interceptor method to fetch that can be used to monitor each request and response.

Example

```
const  
  
requestInterceptor = (  
  requestArguments  
) => {  
  
  console
```



```
.log(  
  "Before request"  
);  
  
}  
  
const  
responseInterceptor = (  
  response  
) => {  
  console  
    .log(  
      "After response"  
    );  
}  
  
fetch(  
  'https://jsonplaceholder.typicode.com/todos/1'  
)  
  .then(  
    response  
  ) => response.json()  
  .then(  

```

```
json
=>
console
.log(json))
// "Before request"
// "After response"
/*
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
*/
```

Axios, one of the most popular libraries for making network calls, comes with [interceptors](#) `axios.interceptor.request` and `axios.interceptor.response` which can be used to monitor and perform actions before any request is made and after every response is received.

© JavaScript Interview Guide | learnersbucket.com

For example, you can add a response interceptor and monitor if the server is giving 401 unauthorized , then log out the user from the application and reset the state.

We can implement the same by overriding the existing fetch method.

All we have to do is store the original fetch method in a variable and override it.

Create two methods on the window object (so that it is globally available), requestInterceptor and responseInterceptor .

Before each request, pass the arguments to requestInterceptor and get the updated value from it, passing it further to the original fetch method.

Similarly, after each response, pass the response to the responseInterceptor , and return the updated value from it.

```
//store the original fetch
```

```
const
```

```
originalFetch =
```

```
window
```

```
.fetch;
```

```
// request interceptor
```

```
// perform all the pre-request actions
```

```
window
```

```
.requestInterceptor = (  
  args  
) => {  
  // your action goes here  
  return  
  args;  
}  
  
// response interceptor  
// perform all the post-response actions  
window  
  .responseInterceptor = (  
    response  
  ) => {  
    // your actions goes here  
    return  
    response;  
  }  
  
// override the original fetch  
window  
  .fetch =
```

async

(...args) => {

© JavaScript Interview Guide | learnersbucket.com

306

// request interceptor

// pass the args to request interceptor

args = requestInterceptor(args);

// pass the updated args to fetch

let

response =

await

originalFetch(...args);

// response interceptor

// pass the response to response interceptor

response = responseInterceptor(response);

// return the updated response

return

response;

};

Test Case

As you can see in the requestInterceptor we are manually setting the page number and in the responseInterceptor we are returning the parsed JSON value.

Input:

```
// request interceptor

// perform all the pre-request actions

window

.requestInterceptor = (

args

) => {

// original request does not contains page info

// assign the pagination in the interceptor

args[

0

] = args[

0

] +

"2"

;

return

args;
```

```
}  
  
// response interceptor  
  
// perform all the post-response actions  
  
window  
  
.responseInterceptor = (  
  
response  
  
) => {  
  
// convert the value to json  
  
// to avoid parsing every time  
  
return  
  
response.json();  
  
}  
  
fetch(  
  
'https://jsonplaceholder.typicode.com/todos/'  
  
)  
  
.then(  
  
json  
  
=>  
  
console  
  
.log(json));
```

307

Output:

```
{  
  "userId"  
:  
  1  
,  
  "id"  
:  
  2  
,  
  "title"  
:  
  "quis ut nam facilis et officia qui"  
,  
  "completed"  
:  
  false  
}
```



308

Cached api call with expiry time

Problem Statement -

Implement a function in JavaScript that caches the API response for the given amount of time. If a new call is made between that time, the response from the cache will be returned, else a fresh API call will be made.

Example

```
const
```

```
call = cachedApiCall(
```

```
1500
```

```
);
```

```
// first call
```

```
// an API call will be made and its response will be cached
```

```
call(
```

```
'https://jsonplaceholder.typicode.com/todos/1'
```

```
,
```

```
{}).then((
```

```
a
```

```
) =>
```

```
console
```

```
.log(a));

// "making new api call"

/*

{

  "userId": 1,

  "id": 1,

  "title": "delectus aut autem",

  "completed": false

}

*/

// cached response will be returned

// it will be quick

setTimeout(

  ()

  => {

    call(

      'https://jsonplaceholder.typicode.com/todos/1'

      ,

      {}).then((

        a
```

```
) =>
```

```
console
```

```
.log(a));
```

```
},
```

```
700
```

```
);
```

```
/*
```

```
{
```

```
"userId": 1,
```

```
"id": 1,
```

```
"title": "delectus aut autem",
```

```
"completed": false
```

```
}
```

```
*/
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
309
```

```
// a fresh API call is made
```

```
// as time for cached entry is expired
```

```
setTimeout(
```

```
()
```

```
=> {  
  call(  
    'https://jsonplaceholder.typicode.com/todos/1'  
    ,  
    {}).then((  
      a  
    ) =>  
      console  
        .log(a));  
    },  
    2000  
  );  
  // "making new api call"  
  /*  
  {  
    "userId": 1,  
    "id": 1,  
    "title": "delectus aut autem",  
    "completed": false  
  }  
}
```

\*/

We can implement this function by forming a [closure](#). The outer

function will accept the time and return an async inner function that will accept the arguments to make the API call.

In the inner function, we will create a new unique key from the arguments to cache value.

Using this key, get the entry from the cache. If there is no entry present or the time of the entry is expired, make a new API call. Else return the value of the entry.

To generate the key and make the API call we will be using two helper functions.

Generating unique key

```
// helper function to create a key from the input
```

```
const
```

```
generateKey = (
```

```
path, config
```

```
) => {
```

```
const
```

```
key =
```

```
Object
```

```
.keys(config)
```

```
.sort((  
  a, b  
) => a.localeCompare(b))  
.map((  
  k  
) => k +  
  ":"  
  + config[k].toString())
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

310

```
.join(  
  "&"  
);  
return  
path + key;  
};
```

Make API call

// helper function to make api call

const

makeApiCall =

```
async
(path, config) => {
  try
  {
    let
    response =
    await
    fetch(path, config);
    response =
    await
    response.json();
    return
    response;
  }
  catch
  (e){
    console
    .log(
    "error "
    + e);
```

```
}
```

```
return
```

```
null
```

```
;
```

```
};
```

Main function to cache API call

```
const
```

```
cachedApiCall = (
```

```
time
```

```
) => {
```

```
// to cache data
```

```
const
```

```
cache = {};
```

```
// return a new function
```

```
return
```

```
async
```

```
function
```

```
(
```

```
path, config = {}
```

```
) {
```



```
// get the key

const

key = generateKey(path, config);

// get the value of the key

let

entry = cache[key];

// if there is no cached data

// or the value is expired

// make a new API call

if

(!entry ||

Date

.now() > entry.expiryTime){

console

.log(

"making new api call"

);

// store the new value in the cache

try

{
```

```
const
value =
await
makeApiCall(path, config)
cache[key] = { value, expiryTime:
Date
.now()
+ time };
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

311

```
}
```

```
catch
```

```
(e){
```

```
console
```

```
.log(error);
```

```
}
```

```
}
```

```
//return the cache
```

```
return
```

```
cache[key].value;
```

```
}
```

```
};
```

Test Case

```
const
```

```
call = cachedApiCall(
```

```
1500
```

```
);
```

```
// first call
```

```
// an API call will be made and its response will be cached
```

```
call(
```

```
'https://jsonplaceholder.typicode.com/todos/1'
```

```
,
```

```
{}).then((
```

```
a
```

```
) =>
```

```
console
```

```
.log(a));
```

```
/*making new api call"
```

```
/*
```

```
{
```

```
"userId": 1,  
"id": 1,  
"title": "delectus aut autem",  
"completed": false  
}  
*/  
  
// cached response will be returned  
// it will be quick  
setTimeout(  
  ()  
=> {  
  call(  
    'https://jsonplaceholder.typicode.com/todos/1'  
    ,  
    {}).then((  
      a  
    ) =>  
      console  
        .log(a));  
  },
```

700

);

/\*

{

"userId": 1,

"id": 1,

"title": "delectus aut autem",

"completed": false

}

\*/

© JavaScript Interview Guide | learnersbucket.com

312

// a fresh API call is made

// as time for cached entry is expired

setTimeout(

()

=> {

call(

'https://jsonplaceholder.typicode.com/todos/1'

,

```
{}).then((  
  a  
) =>  
  console  
    .log(a));  
},  
2000  
);  
// "making new api call"  
/*  
{  
  "userId": 1,  
  "id": 1,  
  "title": "delectus aut autem",  
  "completed": false  
}  
*/
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

313

Polyfill for `getElementByClassName()`

## Problem Statement -

Write a custom function to find all the elements with the given class in the DOM.

### Example

Input:

```
<div class=
```

```
'a'
```

```
id=
```

```
"root"
```

```
>
```

```
<div class=
```

```
'b'
```

```
id=
```

```
'b-1'
```

```
>
```

```
<div class=
```

```
'a'
```

```
id=
```

```
'a-2'
```

```
>
```

```
<div class=
```

'd'

id=

'd-1'

></div>

</div>

<div class=

'c'

id=

'c-1'

>

<div class=

'a'

id=

'a-3'

>

<div class=

'd'

id=

'd-2'

></div>



</div>

</div>

</div>

</div>

findByClass(

'a'

);

Output:

[

<div class=

"a"

id=

"root"

>

<div class=

"b"

id=

"b-1"

>

<div class=

"a"

id=

"a-2"

>

<div class=

"d"

id=

"d-1"

></div>

</div>

<div class=

"c"

id=

"c-1"

>

<div class=

"a"

id=

"a-3"

>

<div class=

"d"

id=

"d-2"

></div>

</div>

</div>

</div>

</div>

,

<div

class

=

"a"

id=

"a-2"

>

<div class=

"d"

id=

```
"d-1"
```

```
></div>
```

```
<
```

```
/div>,
```

```
<div class="a" id="a-3">
```

© JavaScript Interview Guide | learnersbucket.com

314

```
<div class="d" id="d-2"></
```

```
div>
```

```
</div>
```

```
]
```

This can be one of the approaches that we can follow, as a DOM

element can have multiple classes, all we have to do is start from the body or the root element and check if the current node's classlist has the class or not.

If it has it, then push the node in the result. After that traverse all the children of the node and for each child check if they have the class in their classList or not.

We can recursively call the same function for each child and it will call their children and so on and perform the test and return the result.

This traversing pattern is known as DFS ( [Depth First Search](#) ).

function

```
findByClass  
  
(  
  class  
) {  
  // get the root element,  
  // you can start from the body  
  
  const  
  
  root =  
  
  document  
  
  .body;  
  
  // helper function to perform a search using dfs  
  
  function  
  
  search  
  
  (  
  
  node  
  
  ) {  
  
  // store the result  
  
  let  
  
  result = [];  
  
  // if the class name is present in the class list
```

of the element

// add the element in the result

if

(node.classList.contains(

class

)) {

result.push(node);

}

// for all the children of the element

// recursively search and check if the class is

present

for

(

const

element

of

node.children) {

// recursively search

const

res = search(element);

315

```
// add the result from the recursive
```

```
// search to the actual result
```

```
result = result.concat(res);
```

```
}
```

```
// return the result
```

```
return
```

```
result;
```

```
}
```

```
// initiate the search and return the result
```

```
return
```

```
search(root);
```

```
}
```

Test Case

Input:

```
<div class=
```

```
'a'
```

```
id=
```

```
"root"
```

>

<div class=

'b'

id=

'b-1'

>

<div class=

'a'

id=

'a-2'

>

<div class=

'd'

id=

'd-1'

></div>

</div>

<div class=

'c'

id=



'c-1'

>

<div class=

'a'

id=

'a-3'

>

<div class=

'd'

id=

'd-2'

></div>

</div>

</div>

</div>

</div>

console

.log(findByClass(

'a'

));

Output:

[

<div class=

"a"

id=

"root"

>

<div class=

"b"

id=

"b-1"

>

<div class=

"a"

id=

"a-2"

>

<div class=

"d"

id=

"d-1"

></div>

</div>

<div class=

"c"

id=

"c-1"

>

<div class=

"a"

id=

"a-3"

>

<div class=

"d"

id=

"d-2"

></div>

</div>

316

</div>

</div>

</div>

,

<div

class

=

"a"

id=

"a-2"

>

<div class=

"d"

id=

"d-1"

></div>

<

/div>,

<div class="a" id="a-3">

```
<div class="d" id="d-2"></
```

```
div>
```

```
</div>
```

```
]
```

© JavaScript Interview Guide | learnersbucket.com

317

Implement getByClassNameHierarchy()

Problem Statement -

Write a function getByClassNameHierarchy() that takes a path of class names as input and returns an array of the last elements of that path.

Example

Input:

```
<div class="a" id="a-1">
```

```
<div class="b" id="b-1">
```

```
<div class="c" id="c-1"/>
```

```
<div class="c" id="c-2"/>
```

```
</div>
```

```
<div class="c" id="c-3"/>
```

```
</div>
```

```
getByClassNameHierarchy("a>b>c");
```

Output:

```
[  
  
<div class="c" id="c-1"></div>,  
  
<div class="c" id="c-2"></div>  
  
]
```

[This function is the extended version of](#) `getElementByClassName()` .

Using the following approach we can solve this.

- Split the input path in an array of classes.
- Use an index tracker to track the current class at different levels.
- Recursively traverse the DOM from the root and in each

functional call add the following checks,

- If the element is null then return (terminate further calls).
- If the current class is the last of the path and it is present in the element, add it to the result as we have reached the required path

and return (terminate further calls).

© JavaScript Interview Guide | learnersbucket.com

318

- In the end, traverse all the children of the current element and if the element has the current class, recursively traverse its child

with the next class (increase the index tracker by one), else

traverse the child with the first class (index tracker as zero) and

keep on finding the path.

For the DOM traversal, we will be using a helper function.

```
function  
getByClassNameHierarchy  
(  
  element, classNames  
)  
{  
  // get all the classnames  
  const  
  classList = classNames.split(  
    '>  
  );  
  // pass the array as a reference  
  const  
  result = [];  
  // traverse the dom from the root  
  traverseDOM(element, classList,  
    0  
  , result);  
  // return the result
```

```
return  
result;  
}  
  
// helper function  
  
function  
traverseDOM  
(  
element, classNames, index, result  
) {  
    // if the element is not present  
    if  
    (!element) {  
        return  
        ;  
    }  
  
    // get the current class name  
    const  
    targetClass = classNames[index];  
  
    // if the last class of the classNames  
  
    // and the element contains the class.
```



```
// add the element to the result  
  
if  
  
(index === classNames.length -  
  
1  
  
&&  
  
element.classList.contains(targetClass)) {  
result.push(element);  
  
return  
  
;  
  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

319

```
// iterate each children of the element  
  
for  
  
(  
  
const  
  
child  
  
of  
  
element.children) {  
  
// if the child has the class
```

```
// recursively traverse and search for the next
class in the child

// thus increase the index by one

if
(element.classList.contains(targetClass)) {
traverseDOM(child, classNames, index +
1
,
result);
}

// else start the search from the scratch in the
child

else
{
traverseDOM(child, classNames,
0
, result);
}
}
}
```

Test Case

Input:

<div class=

"a"

id=

"root"

>

<div class=

"b"

id=

"b-1"

>

<div class=

"c"

id=

"c-1"

>

</div>

<div class=

"c"

id=

"c-2"

>

</div>

</div>

<div class=

"b"

id=

"b-2"

>

<div class=

"c"

id=

"c-3"

>

</div>

</div>

</div>

console

.log(getByClassNameHierarchy(

```
document
.getElementById(
'root'
),
'a>b>c'
));
```

Output:

```
[
<div class=
"c"
id=
"c-1"
></div>
,
<div
class
=
"c"
id=
"c-2"
```

>

</div>

,

<div

class

=

"c"

id=

"c-3"

>

</div>

]

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

320

Find element with the given color property

Problem Statement -

Write a function to find all the elements with the given color. Here the color will be provided in any format like, plain text ( white ), HEXA value ( #fff or #ffffff ), or RGB value ( RGB(255, 255, 255) ).

Example

Input:

<div id=

"root"

>

<span style=

"color:#fff;"

>

1

</span>

<span style=

"color:#eee;"

>

2

</span>

<span style=

"color:white;"

>

3

</span>

<span style=

"color:rgb(255, 255, 255);"

>

4

</span>

</div>

findElementByColor(

document

.getElementById(

'root'

),

'rgb(255, 255, 255)'

);

Output:

[

<span style=

"color:#fff;"

>

1

</span>

,

<span style=



```
"color:white;"
```

```
>
```

```
3
```

```
<
```

```
/span>,
```

```
<span style="color:rgb(255, 255, 255);">4</
```

```
span>
```

```
]
```

The most challenging part of this problem is to convert the color

values to a single format that can be used for the comparison.

For this, we can create a function that will temporarily create a new element and apply the input color as its style, then using the

[getComputedStyle\(\) method](#), [we can get the RGB value of the color](#)

[and convert it to HEXA code](#) for comparison.

```
function
```

```
getHexColor
```

```
(
```

```
color
```

```
){
```

```
// create a new element
```

```
const
```

```
div =  
document  
.createElement(  
'div'  
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

321

```
// apply the color to the element  
div.style.color = color;  
  
// get the computed style of the div  
let  
colors =  
window  
.getComputedStyle(  
document  
.body.appendChild(div));  
  
// get the RGB value of the color  
colors = colors.color.match(  
/d+/g  
)
```

```
function
(
a
){
return
parseInt
(a,
10
); });

// remove the div

document

.body.removeChild(div);

// convert the RGB value to HEXA and return it.

return

(colors.length >=

3

)?

'#'

+ (((

1
```

<<

24

)

+ (colors[

0

] <<

16

) +

(colors[

1

] <<

8

) + colors[

2

]).toString(

16

).substr(

1

))

:

```
false
```

```
;
```

```
}
```

For finding the elements with the given color value, we will first

convert the input color value to the HEXA using the above function.

Then recursively traverse all the DOM elements and get their color values and convert them to HEXA and compare it with the input color value's HEXA.

If they match, push the element to the result array and return it.

```
function
```

```
findElementByColor
```

```
(
```

```
element, colorValue
```

```
){
```

```
// convert input color to HEXA
```

```
const
```

```
inputColorHexa = getHexColor(colorValue);
```

```
// to store the result
```

```
const
```

```
result = [];
```

```
// helper function to traverse the DOM
```

```
const
```

```
search = (
```

```
element, attrValue
```

```
) => {
```

```
// get the color value of the element
```

```
let
```

```
value = element.style[
```

```
'color'
```

```
];
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
322
```

```
// convert the value to the HEXA
```

```
value = getHexColor(value);
```

```
// if both the HEXA value matches
```

```
// store the result
```

```
if
```

```
(value === inputColorHexa){
```

```
result.push(element);
```

```
}
```

```
// recursively search for each child of the element
for
(
const
child
of
element.children) {
search(child, attrValue);
}
};

// begin the search
search(element, colorValue);

// return the result
return
result;
}
```

Test Case

Input:

<div id=

"root"

>

<span style=

"color:#fff;"

>

1

</span>

<span style=

"color:#eee;"

>

2

</span>

<span style=

"color:white;"

>

3

</span>

<span style=

"color:rgb(255, 255, 255);"

>

4



```
</span>
```

```
</div>
```

```
console
```

```
.log(findElementByColor(
```

```
document
```

```
.getElementById(
```

```
'root'
```

```
),
```

```
'white'
```

```
));
```

```
Output:
```

```
[
```

```
<span style=
```

```
"color:#fff;"
```

```
>
```

```
1
```

```
</span>
```

```
,
```

```
<span style=
```

```
"color:white;"
```

>

3

<

/span> ,

<span style="color:rgb(255, 255, 255);">4</

span>

]

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

323

Throttle an array of task

Problem Statement -

Implement a throttler that executes an array of tasks. When the

throttler is passed a number, only executes that number of the tasks and passes the other tasks into a queue.

Example

Input:

const

task = [

1

,

2

,

3

,

4

,

5

,

6

,

7

,

8

,

9

,

10

];

const

count =

5

```
;
throttle(task, count,
2000
);
// [1, 2, 3, 4, 5] //
immediately
throttle(task, count,
2000
);
// [6, 7, 8, 9, 10] //
after 2 seconds
throttle(task, count,
2000
);
// [1, 2, 3, 4, 5] //
after 2 seconds
```

In each call, 10 new tasks are pushed and only 5 are executed, remaining are stored in the queue.

What is throttling?

Throttling is a way/technique to restrict the number of function

execution/calls.

Throttling an array of tasks

[For this implementation, we will modify the original throttle function](#)

to accept an array of tasks and a count and run the number of tasks the same as the count.

In the original implementation, we will add a queue array that will

store the task. In each throttle call, copy all the tasks to the queue and then run only the count of tasks from the queue and so on in the

consecutive call.

By default, the count will be of the same size as the array of tasks.

© JavaScript Interview Guide | learnersbucket.com

324

```
const
```

```
throttle = (
```

```
task, count = task.length, callback,
```

```
delay = 1000
```

```
) => {
```

```
// track the throttle
```

```
let
```

```
lastFunc;
```

```
let
```

```
lastRan;

// track the task

let

queue = [];

return

function

() {

// store the context to pass it to the callback

function

const

context =

this

;

const

args =

arguments

;

// if the throttle is executed the first time

// run it immediately

if
```

```
(!lastRan) {  
  // copy all the tasks to the queue  
  queue = [...queue, ...task];  
  // get the amount of task to run  
  const  
  execute = queue.splice(  
    0  
    , count);  
  // pass those tasks to the callback  
  callback(execute);  
  // update the last ran time  
  // to run it after the delay  
  lastRan =  
    Date  
    .now();  
}  
else  
{  
  // clear the timer  
  clearTimeout(lastFunc);
```

```
// start a new timer

// run the function after the delay

lastFunc = setTimeout(

function

() {

// calc the difference between

// the last ran and current time

// if it is greater than the delay

// invoke it

if

((

Date

.now() - lastRan) >= delay) {

// copy all the tasks to the queue

queue = [...queue, ...task];

© JavaScript Interview Guide | learnersbucket.com

325

// get the amount of task to run

const

execute = queue.splice(
```



0

, count);

// pass those tasks to the callback

callback(execute);

// update the last ran time

// to run it after the delay

lastRan =

Date

.now();

}

}, delay - (

Date

.now() - lastRan));

}

}

};

Test Case

Input:

// this will add these tasks at each call

btn.addEventListener(

'click'

, throttle([

1

,

2

,

3

,

4

,

5

,

6

,

7

,

8

,

9

,

10

],

2

,

(task) => {

console

.log(task);

},

2000

));

Output:

// [object Array] (2)

[

1

,

2

]

// 1st call

// [object Array] (2)

[

3

,

4

]

// 2nd call after 2 seconds

// [object Array] (2)

[

5

,

6

]

// 3rd call after 2 seconds

// [object Array] (2)

[

7

,

8

]

// 4th call after 2 seconds

// [object Array] (2)

```
[
```

```
9
```

```
,
```

```
10
```

```
]
```

```
// 5th call after 2 seconds
```

```
// [object Array] (2)
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

326

```
[
```

```
1
```

```
,
```

```
2
```

```
]
```

```
// 6th call after 2 seconds
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

327

Decode a string

Problem Statement -

Given a string, write a program to decode the string which is encoded in a pattern where a substring is wrapped in square brackets led by a number.

Example

Input:

"2[a2[b]]"

"3[b2[ca]]"

output:

"abbabb"

"bcacabcacabcaca"

This problem could be solved by using the two [stacks](#) .

- We will use two different stacks, one to store the count of numbers numStack and second to store the substring charStack .
- We will iterate the whole string on each character and check if the current char is number then push it to the numStack .
- Else if the character is opening square bracket '[' then check if there is a count number assigned to it or not. If it is then it may have already been pushed in the first condition so just add the character to charStack else add the current character to charStack and 1 to the numStack as it will be called only once.
- If the character is closing square bracket ']' then get the (top element from stack) count from numStack and substring from charStack and decode them, then again add the decoded string back to the charStack so that in the next iteration the decoded

substring will be repeated along with the parent substring.

- Else the character is alphabet so add it to the charStack .

© JavaScript Interview Guide | learnersbucket.com

328

- In the end create a string from the charStack (which will have decoded substring) and return it.

```
let
```

```
decodeString = (
```

```
str
```

```
) => {
```

```
let
```

```
numStack =
```

```
new
```

```
Stack();
```

```
let
```

```
charStack =
```

```
new
```

```
Stack();
```

```
let
```

```
decoded =
```

```
""
```

```
, temp =  
""  
  
;  
  
for  
  
(  
  
let  
  
i =  
  
0  
  
; i < str.length; i++){  
  
let  
  
count =  
  
0  
  
;  
  
let  
  
val = str[i];  
  
// if char is number then  
  
// push to numStack  
  
if  
  
(  
  
/^\d+$/
```



```
.test(val)){  
numStack.push(val);  
}  
else  
if  
(val ===  
'['  
) {  
// else if open bracket and previous character  
is number  
// then it will already added to numStack in  
the above (if condition)  
// just add the char to charStack  
if  
(  
/^\d+$/  
.test(str[i  
-1  
])) {  
charStack.push(str[i]);
```

```
}  
  
else  
  
{  
  
    // else add 1 to numstack  
  
    // and char to charStack  
    charStack.push(str[i]);  
    numStack.push(  
        1  
    );  
}  
}  
  
else  
  
if  
  
(val ===  
  
'I'  
  
) {  
  
    // if close bracket  
  
    // reset temp and count  
    temp =  
    ""
```

```
;
```

```
count =
```

```
0
```

```
;
```

```
// get the count from numStack
```

© JavaScript Interview Guide | learnersbucket.com

329

```
count = !numStack.isEmpty() && numStack.pop();
```

```
// get the subStr from charStack
```

```
while
```

```
(!charStack.isEmpty() && charStack.peek()
```

```
!==
```

```
'['
```

```
){
```

```
temp = charStack.pop() + temp;
```

```
}
```

```
// remove the '[' char from charStack
```

```
if
```

```
(!charStack.isEmpty() && charStack.peek()
```

```
===
```

```
'['  
) {  
    charStack.pop();  
}  
  
// create the repeat subStr  
  
decoded = temp.repeat(count);  
  
// push the newlyCreated subStr to charStack  
again  
for  
  
(  
  
let  
  
j =  
  
0  
  
; j < decoded.length; j++) {  
    charStack.push(decoded[j]);  
}  
  
// reset the string  
  
decoded =  
  
""  
  
;
```

```
}  
  
else  
  
{  
  
    // if alpha character then add to charStack  
  
    charStack.push(val);  
  
}  
  
}  
  
// form the decoded string from charStack  
  
while  
  
(!charStack.isEmpty()){  
  
    decoded = charStack.pop() + decoded;  
  
}  
  
// return the decoded str  
  
return  
  
decoded;  
  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

330

Test Case

Input:

```
console  
  
.log(decodeString(  
  
"2[a2[b]]"  
  
));
```

```
console  
  
.log(decodeString(  
  
"3[b2[ca]]"  
  
));
```

Output:

"abbabb"

"bcacabcacabcaca"

© JavaScript Interview Guide | learnersbucket.com

331

## Trie data structure

A Trie, also known as a digital tree or prefix tree, is a kind of search tree — an ordered [tree data structure](#) used to store a dynamic set or associative array where the keys are usually strings.

### What is Trie data structure?

Trie data structure was described by René de la Briandais in 1959

solely to solve the very problem of representing a set of words.

The term “Trie” comes from the word retrieval and is usually

pronounced “try”, to separate it from other “tree” structures.

However, it is basically a tree data structure with certain rules to follow in terms of how it is created and used. It is a tree-like data structure wherein the nodes of the tree store the entire alphabet and strings/words can be retrieved by traversing down a branch path.

According to Donald Knuth’s research in [The Art of Computer](#)

[Programming](#) : Trie memory for computer searching was first recommended by René

de la Briandais. He pointed out that we can save memory space at the expense of running time if we use a linked list for each node vector since most of the entries in the vectors tend to be empty.

The main idea behind using Tries as a data structure was that they could be a nice compromise between running time and memory.

List of operations performed on Trie

- *insert(word)* : Adds a new word.
- *remove(word)* : Removes the given word.
- *contains(word)* : Checks if Trie has the given word.
- *find(prefix)* : Returns all the words with the given prefix.

© JavaScript Interview Guide | learnersbucket.com

332

Implementing Trie data structure

Each Trie has an empty root node, with links (or references) to other nodes — one for each possible alphabetic value. The shape and the

structure of a trie is always a set of linked nodes, connecting back to an empty root node.

Thus, the size of a trie is directly correlated to the size of all the possible values that the Trie could represent.

Base structure

We will need a TrieNode object to insert a new word in the Trie, which would represent an entirely new Trie.

```
// we start with the TrieNode

const

TrieNode =

function

(

key

) {

// the "key" value will be the character in sequence

this

.key = key;

// we keep a reference to parent

this

.parent =

null
```



```
;

// we have hash of children

this

.children = {};

// check to see if the node is at the end

this

.end =

false

;

this

.getWord =

function

() {

let

output = [];

let

node =

this

;

while
```

```
(node !==  
null  
) {  
  output.unshift(node.key);  
  node = node.parent;  
}  
return  
output.join(  
"  
");  
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

333

}

const

Trie =

function

() {

this

.root =

```
new
```

```
TrieNode(
```

```
  null
```

```
);
```

```
//Other methods will go here...
```

```
}
```

Inserting a word in Trie

To insert a new word in the Trie we have to check for two things.

- Check that the word that needs to be added doesn't already exist in this trie.

- Next, if we've traversed down the branch where this word ought to live and the words don't exist yet, we'd insert a value into the node's reference where the word should go.

```
// inserts a word into the trie.
```

```
this
```

```
.insert =
```

```
function
```

```
(
```

```
  word
```

```
) {
```

```
  let
```

```
node =  
this  
.root;  
  
// we start at the root  
  
// for every character in the word  
for  
(  
let  
i =  
0  
; i < word.length; i++) {  
    // check to see if the character node exists  
    in children.  
  
    if  
  
    (!node.children[word[i]]) {  
        // if it doesn't exist, we then create it.  
  
        node.children[word[i]] =  
  
        new  
  
        TrieNode(word[i]);  
  
        // we also assign the parent to the child
```

```
node.  
  
node.children[word[i]].parent = node;  
  
}  
  
// proceed to the next depth in the trie.  
  
node = node.children[word[i]];  
  
// finally, we check to see if it's the last  
  
word.  
  
if  
  
(i == word.length  
  
-1  
  
) {  
  
// if it is, we set the end flag to true.
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

334

```
node.end =  
  
true  
  
;  
  
}  
  
}  
  
};
```

## Searching a word in the Trie

To check if the trie contains the given word or not.

- For every character in the word. Check to see if character nodes exist in children.
- If it exists, proceed to the next depth of the trie.
- Else return false, since it's not a valid word.
- At the end return the word.

// check if it contains a whole word.

this

.contains =

function

(

word

) {

let

node =

this

.root;

// for every character in the word

for

(

```
let
i =
0
; i < word.length; i++) {
// check to see if the character node exists
in children.
if
(node.children[word[i]]) {
// if it exists, proceed to the next depth
of the trie.
node = node.children[word[i]];
}
else
{
// doesn't exist, return false since it's
not a valid word.
return
false
;
}
```

```
}  
  
// we finished going through all the words, but  
  
is it a whole word?  
  
return  
  
node.end;  
  
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

335

Find the word starting with given prefix in the Trie To find all the words with a given prefix, we need to perform two

operations.

- First, make sure the prefix actually has words.
- Second, find all the words with the given prefix.

```
// returns every word with given prefix
```

```
this  
  
.find =  
  
function  
  
(  
  
prefix  
  
) {  
  
let
```



```
node =  
this  
.root;  
let  
output = [];  
// for every character in the prefix  
for  
(  
let  
i =  
0  
; i < prefix.length; i++) {  
// make sure prefix actually has words  
if  
(node.children[prefix[i]]) {  
node = node.children[prefix[i]];  
}  
else  
{  
// there's none. just return it.
```

```

return
output;
}
}

// recursively find all words in the node
findAllWords(node, output);

return
output;
};

// recursive function to find all words in the given
node.

const
findAllWords = (
node, arr
) => {

// base case, if node is at a word, push to output
if
(node.end) {
arr.unshift(node.getWord());
}

```

```
// iterate through each children, call recursive
```

```
findAllWords
```

```
for
```

```
(
```

```
let
```

```
child
```

```
in
```

```
node.children) {
```

```
findAllWords(node.children[child], arr);
```

```
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

336

```
}
```

### Removing a word from the Trie

To delete a key, we do not delete the node corresponding to the key as it might have some children which still contain a key. Instead, we

simply have to search for it and set its value to null.

However, to improve efficiency, if the node corresponding to the key has no children or all its children have null values, we might also

delete the entire node.

```
// removes the given word
```

```
this

.remove =

function

(

word

) {

let

root =

this

.root;

if

(!word)

return

;

// recursively finds and removes a word

const

removeWord = (

node, word

) => {

// check if current node contains the word
```

```
if
(node.end && node.getWord() === word)
{
// check and see if node has children

let
hasChildren =
Object
.keys(node.children).length
>
0
;

// if we have children we only want
to un-flag the end node
that marks the end of a word.

// this way we do not remove words that
contain/include
supplied word

if
(hasChildren) {
node.end =
```

false

;

}

else

{

// remove word by getting parent

and setting children to

empty dictionary

node.parent.children = {};

}

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

337

return

true

;

}

// recursively remove word from all children

for

(

let

```
key
in
node.children) {
removeWord(node.children[key], word)
}
return
false
};
// call remove word on root node
removeWord(root, word);
};
```

Complete code of Trie data structure

```
// we start with the TrieNode
const
TrieNode =
function
(
key
) {
// the "key" value will be the character in sequence
```

```
this

.key = key;

// we keep a reference to parent

this

.parent =

null

;

// we have hash of children

this

.children = {};

// check to see if the node is at the end

this

.end =

false

;

this

.getWord =

function

() {

let
```



```
output = [];  
let  
node =  
this  
;  
while  
(node !==  
null  
) {  
output.unshift(node.key);  
node = node.parent;  
}  
return  
output.join(  
"  
");  
};  
  
© JavaScript Interview Guide | learnersbucket.com  
338  
}
```

```
const
Trie =
function
() {
  this
  .root =
  new
  TrieNode(
  null
  );
  // inserts a word into the trie.
  this
  .insert =
  function
  (
  word
  ) {
    let
    node =
    this
```

```
.root;

// we start at the root

// for every character in the word

for

(

let

i =

0

; i < word.length; i++) {

// check to see if the character node exists

in children.

if

(!node.children[word[i]]) {

// if it doesn't exist, we then create it.

node.children[word[i]] =

new

TrieNode(word[i]);

// we also assign the parent to the child

node.

node.children[word[i]].parent = node;
```

```
}  
  
// proceed to the next depth in the trie.  
node = node.children[word[i]];  
  
// finally, we check to see if it's the last  
word.  
  
if  
(i == word.length  
-1  
) {  
    // if it is, we set the end flag to true.  
    node.end =  
    true  
;  
}  
}  
};  
  
// check if it contains a whole word.  
this  
.contains =  
function
```

```
(  
word  
) {  
  let  
  node =  
  this  
  .root;  
  
  // for every character in the word  
  for  
  (  
  let  
  i =  
  0  
  ; i < word.length; i++) {  
    // check to see if the character node exists  
    in children.  
  
    if  
  
    (node.children[word[i]]) {  
      // if it exists, proceed to the next depth  
      of the trie.
```

```
node = node.children[word[i]];
```

```
}
```

```
else
```

```
{
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
339
```

```
// doesn't exist, return false since it's
```

```
not a valid word.
```

```
return
```

```
false
```

```
;
```

```
}
```

```
}
```

```
// we finished going through all the words, but
```

```
is it a whole word?
```

```
return
```

```
node.end;
```

```
};
```

```
// returns every word with given prefix
```

```
this
```

```
.find =  
function  
(  
  prefix  
) {  
  let  
    node =  
    this  
    .root;  
  let  
    output = [];  
  // for every character in the prefix  
  for  
  (  
    let  
      i =  
      0  
      ; i < prefix.length; i++) {  
    // make sure prefix actually has words  
    if
```

```
(node.children[prefix[i]]) {  
  node = node.children[prefix[i]];  
}  
else  
{  
  // there's none. just return it.  
  return  
  output;  
}  
}  
  
// recursively find all words in the node  
findAllWords(node, output);  
  
return  
output;  
};  
  
// recursive function to find all words in the given  
node.  
  
const  
findAllWords = (  
  node, arr
```



```

) => {

// base case, if node is at a word, push to output
if
(node.end) {
arr.unshift(node.getWord());
}

// iterate through each children, call recursive
findAllWords
for
(
let
child
in
node.children) {
findAllWords(node.children[child], arr);
}

```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

340

```

}
```

// removes a word from the trie.

```
this

.remove =

function

(

word

) {

let

root =

this

.root;

if

(!word)

return

;

// recursively finds and removes a word

const

removeWord = (

node, word

) => {

// check if current node contains the word
```

```
if
(node.end && node.getWord() === word)
{
// check and see if node has children

let
hasChildren =
Object
.keys(node.children).length
>
0
;

// if we have children we only want
to un-flag the end node
that marks the end of a word.

// this way we do not remove words that
contain/include
supplied word

if
(hasChildren) {
node.end =
```

```
false  
;  
}  
else  
{  
    // remove word by getting parent  
    and setting children to  
    empty dictionary  
    node.parent.children = {};  
}  
return  
true  
;  
}  
// recursively remove word from all children  
for  
(  
    let  
    key  
    in
```

```
node.children) {  
  removeWord(node.children[key], word)  
}  
  
return  
  
false  
  
};  
  
// call remove word on root node  
  
removeWord(root, word);  
  
© JavaScript Interview Guide | learnersbucket.com  
  
341  
  
};  
  
}
```

Test Case

Input:

```
const  
  
trie =  
  
new  
  
Trie();  
  
// insert few values  
  
trie.insert(
```

```
"peter"  
);  
trie.insert(  
  "piper"  
);  
trie.insert(  
  "picked"  
);  
trie.insert(  
  "pickled"  
);  
trie.insert(  
  "pepper"  
);  
// check contains method  
console  
.log(trie.contains(  
  "picked"  
));  
console
```

```
.log(trie.contains(
    "pepper"
));
trie.remove(
    "pepper"
);
// check find method
console
.log(trie.find(
    "pi"
));
console
.log(trie.find(
    "pe"
));
```

Output:

true

true

[

"pickled"

```
,  
"picked"
```

```
,  
"piper"
```

```
]
```

```
[
```

```
"peter"
```

```
]
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

342

Find first or last occurrence of a given  
number in a sorted array

Problem Statement -

Given a sorted array with duplicate values we have to create two  
different algorithms which will find the first and last occurrence of the  
given element.

Example

Input:

```
[
```

```
1
```

```
,
```



2

,

3

,

4

,

5

,

5

,

6

,

7

,

8

,

9

]

5

Output:

4

//Index with first occurrence

5

//Index with second occurrence

The best way to find anything in a sorted array is by using [binary search](#) .

We will be using a modified version of binary search to find the first or last occurrence of the element in the array.

First Occurrence

As binary search returns the index of the element as soon as it finds it , to find the first occurrence, even after finding the element we will keep looking in the lower range to check if the element has occurred before it or not, if it is then return the lowest index.

const

first = (

arr, value

) => {

let

low =

0

;

let

```
high = arr.length -
```

```
1
```

```
;
```

```
let
```

```
result =
```

```
-1
```

```
;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
343
```

```
//keep looking for all the elements in the array
```

```
while
```

```
(low <= high){
```

```
//Get the mid
```

```
const
```

```
mid =
```

```
Math
```

```
.ceil((low + high) /
```

```
2
```

```
);
```

```
//If element found
```

```
//Then store the index and look in the lower
range for first
occurrence
if
(arr[mid] === value){
result = mid;
high = mid -
1
;
}
else
if
(value < arr[mid]){
//If value is less than the mid element then
looking the lower range
high = mid -
1
;
}
else
```

```
{  
    //If value is greater than the mid element  
    then look in the upper  
    range  
    low = mid +  
    1  
    ;  
}  
}
```

//Return the index

```
return  
result;  
}
```

Test Case

Input:

```
const
```

```
arr = [
```

```
1
```

```
,
```

```
2
```

,

3

,

4

,

5

,

5

,

6

,

6

,

7

,

8

,

9

,

10

```
];  
  
console  
.log(first(arr,  
5  
));  
  
console  
.log(first(arr,  
6  
));
```

Output:

```
4  
  
6
```

### Last Occurrence

Just like finding the first occurrence to find the last occurrence we will have to keep looking for the element in the upper range after it is

found to make sure we get the last index of the element.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

344

```
const
```

```
last = (
```

```
arr, value
```

```
) => {  
  let  
  low =  
  0  
  ;  
  let  
  high = arr.length -  
  1  
  ;  
  let  
  result =  
  -1  
  ;  
  //Search till the array exists  
  while  
  (low <= high){  
  //Get the mid  
  const  
  mid =  
  Math
```



```
.ceil((low + high) /  
2  
);  
  
//If element found  
  
//Then keep looking for the last element in the  
upper range  
  
if  
  
(arr[mid] === value){  
    result = mid;  
  
    low = mid +  
  
    1  
  
    ;  
  
    }  
  
    else  
  
    if  
  
(value < arr[mid]){  
  
    //Else if value is less than mid element then  
looking in the lower  
range  
  
    high = mid -
```

```
1
;
}
else
{
//Else if value is greater than mid element
then look in the upper
range
low = mid +
1
;
}
}

//Return the result
return
result;
}
```

Test Case

Input:

const

```
arr = [
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
,
```

```
5
```

```
,
```

```
5
```

```
,
```

```
6
```

```
,
```

```
6
```

```
,
```

```
7
```

```
,
```

```
8
```

```
,  
9  
  
,  
10  
  
];  
  
console  
.log(last(arr,  
5  
));  
  
console  
.log(last(arr,  
6  
));
```

Output:

```
5  
7
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

345

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

346

## Piping function in JavaScript – Part 1

### Problem Statement -

Given an object which can have a function as a value at a nested level, create a function that will accept arguments as input and pass it through all the functions in the input object and return the computed value.

### Example

Input:

```
{  
  a : {  
    b : (  
      a,b,c  
    ) => a+b+c,  
    c : (  
      a,b,c  
    ) => a+b-c,  
  },  
  d : (  
    a,b,c  
  ) => a-b-c  
}
```

Fn(obj)(

1

,

1

,

1

);

Output:

{

a : {

b :

3

,

c :

1

},

d:

-1

}

We can solve this by forming a [closure](#) , creating a function that will accept the object as input, and from this function returning another function that will accept the arguments.

Inside the inner function. Iterate all the keys of the object, if the value of the key is a function, pass the arguments to it and store its

computed value on that key. Else recursively call the same function for nested processing.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

347

At the end return the original input object as we are doing the processing in-place.

```
const
```

```
pipe = (
```

```
  obj
```

```
) => {
```

```
  // return another function that will accept all
```

```
  the args
```

```
  return
```

```
  function
```

```
  (
```

```
    ...args
```

```
  ){
```

```
    // iterate the keys of the object
```

```
    for
```

```
(  
  let  
  key  
  in  
  obj) {  
  // get the value  
  let  
  val = obj[key];  
  // if the value is a function  
  if  
  (  
    typeof  
    val ===  
    'function'  
  ) {  
    // pass the args to the function  
    // store the result on the same key  
    obj[key] = val(...args);  
  }  
  else
```



```
{  
  // else recursively call the same function  
  // if it is nested object it will be further  
  processed  
  obj[key] = pipe(val)(...args);  
}  
}  
  
// return the input after processing  
return  
obj;  
}  
};
```

Test Case

Input:

```
let  
test = {  
  a: {  
    b: (  
      a, b, c  
    ) => a + b + c,
```

```

c: (
  a, b, c
) => a + b - c,
},
d: (
  a, b, c
) => a - b - c,
e:
1
,
f:
true
};

```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

348

console

.log(pipe(test)(

1

,

1

,

1

));

Output:

{

"a"

: {

"b"

:

3

,

"c"

:

1

},

"d"

:

-1

,

"e"

:

1

,

"f"

:

true

}

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

349

## Piping function in JavaScript – Part 2

### Problem Statement -

Create a function that accepts multiple functions as an argument and a value and run this value through each function and return the final

output.

### Example

Input:

```
const
```

```
val = { salary:
```

```
10000
```

```
};
```

```
const
```

```
getSalary = (  
  person  
) => person.salary  
  
const  
addBonus = (  
  netSalary  
) => netSalary +  
  1000  
;  
  
const  
deductTax = (  
  grossSalary  
) => grossSalary - (grossSalary  
  *  
  .3  
);  
  
const  
result = pipe(  
  getSalary,  
  addBonus,
```

```
deductTax
```

```
)(val);
```

Output:

7700

One of the way to solve this is by forming a [closure](#).

Create two functions, in the first function accept all the functions as arguments and inside it create another function and return it. The

inner function will accept the value and it will pass the value through the functions that are received in the outer function in the sequence.

At the end return the final output from the last function.

```
// accept functions as arguments
```

```
// using rest ... operator convert then to array
```

```
const
```

```
pipe =
```

```
function
```

```
(
```

```
...fns
```

```
){
```

© JavaScript Interview Guide | learnersbucket.com

350

```
// form a closure with inner function
```

```
return  
  
function  
  
(  
  val  
  
) {  
  // run the value through all the functions  
  
  for  
  
  (  
    let  
  
    f  
  
    of  
  
    fns) {  
    val = f(val);  
  
  }  
  
  // return the value after last processing  
  
  return  
  
  val;  
  
};  
  
};
```

Test Case

Input:

```
const
```

```
getSalary = (
```

```
  person
```

```
) => person.salary
```

```
const
```

```
addBonus = (
```

```
  netSalary
```

```
) => netSalary +
```

```
  1000
```

```
;
```

```
const
```

```
deductTax = (
```

```
  grossSalary
```

```
) => grossSalary - (grossSalary
```

```
  *
```

```
  .3
```

```
);
```

```
const
```

```
val = { salary:
```



```
10000  
  
};  
  
const  
result = pipe(  
  getSalary,  
  addBonus,  
  deductTax  
)({ salary:  
  10000  
});  
  
console  
.log(result);
```

Output:

7700

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

351

## Create analytics SDK in JavaScript

### Problem Statement -

Implement an analytics SDK that exposes log events, it takes in events and queues them and then starts sending the events.

- Send each event after a delay of 1 second and this logging fails every  $n \% 5$  times.
- Send the next event only after the previous one resolves.
- When the failure occurs, attempt a retry.

Example

Input:

```
const
```

```
sdk =
```

```
new
```

```
SDK();
```

```
sdk.logEvent(
```

```
"event 1"
```

```
);
```

```
sdk.logEvent(
```

```
"event 2"
```

```
);
```

```
sdk.logEvent(
```

```
"event 3"
```

```
);
```

```
sdk.logEvent(
```

```
"event 4"
```

```
);  
sdk.logEvent(  
    "event 5"  
);  
sdk.logEvent(  
    "event 6"  
);  
sdk.logEvent(  
    "event 7"  
);  
sdk.logEvent(  
    "event 8"  
);  
sdk.logEvent(  
    "event 9"  
);  
sdk.logEvent(  
    "event 10"  
);  
sdk.send();
```

Output:

"Analytics sent event 1"

"Analytics sent event 2"

"Analytics sent event 3"

"Analytics sent event 4"

"-----"

"Failed to send event 5"

"Retrying sending event 5"

"-----"

[OceanofPDF.com](http://OceanofPDF.com)

"Analytics sent event 5"

© JavaScript Interview Guide | learnersbucket.com

352

"Analytics sent event 6"

"Analytics sent event 7"

"Analytics sent event 8"

"-----"

"Failed to send event 9"

"Retrying sending event 9"

"-----"

"Analytics sent event 9"

"Analytics sent event 10"

Breaking the problem statement into subproblems we can create this

SDK in three steps.

Delay function

The most important part of this function is that the events will be sent after a delay of 1 second and fails  $n\%5$  times.

We can do the same by extending the [sleep function](#) .

Create a new Promise and inside that run a setTimeout that will

resolve the promise after the delay.

To the same, add one extra condition that will check if the current execution is  $n\%5$  then reject, else resolve.

```
// function to delay the execution
```

```
wait =
```

```
()
```

```
=>
```

```
new
```

```
Promise
```

```
((
```

```
resolve, reject
```

```
) => {
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
// reject every  $n \% 5$  time
```

```
if
```

```
(
```

```
this
```

```
.count %
```

```
5
```

===

```
0
){
reject();
}
else
{
resolve();
}
},
1000
);
});
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

353

Queue the events

We have to store the events so that they can be sent one by one. We also need a tracker so that we can reject for each  $n\%5$  th call .

We can create a class and initialize these in the constructor.

class

SDK

```
{  
  constructor  
()  
  // hold the events  
  this  
  .queue = [];  
  // track the count  
  this  
  .count =  
  1  
  ;  
}  
// push event in the queue  
logEvent(ev) {  
  this  
  .queue.push(ev);  
}  
}
```

Sending the events



The final part is sending the events, for this, we can create a helper function that will recursively call itself and keep on sending one-one events every time.

This will be an async function and in each call, get the first element from the queue and try the wait() , if it resolves then print the log or perform any other operation, else if it fails, push the event back in the queue for retry. Finally, recursively call the same function for the next operation.

Add a base case to stop the execution if there are no more events in the queue. Also, track the count in each call.

```
// to send analytics
```

```
// recursively send the events
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

354

```
sendAnalytics =
```

```
async
```

```
function
```

```
() {
```

```
// if there are no events in the queue
```

```
// stop execution
```

```
if
```

```
(
```

```
this
```

```
.queue.length ===
```

```
0
){
return
;
}
// get the first element from the queue
const
current =
this
.queue.shift();
try
{
// delay
await
this
.wait();
// print the event
// can perform any other operations as well
like making api call
console
```

```
.log(  
  "Analytics sent "  
  + current);  
  
// increase the count  
  
this  
  
.count++;  
  
}  
  
catch  
  
(e){  
  
  // if execution fails  
  
  console  
  
    .log(  
  
      "-----"  
  
    );  
  
  console  
  
    .log(  
  
      "Failed to send "  
  
      + current);  
  
  console  
  
    .log(  

```

```
"Retrying sending "  
+ current);  
  
console  
  
.log(  
  
"-----"  
  
);  
  
// reset the count  
  
this  
  
.count =  
  
1  
  
;  
  
// push the event back into the queue  
  
this  
  
.queue.unshift(current);  
  
}  
  
finally  
  
{  
  
// recursively call the same function  
  
// to send the remaining  
  
this
```

```
.sendAnalytics();
```

```
}
```

```
}
```

```
// start the execution
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

355

```
send =
```

```
async
```

```
function
```

```
() {
```

```
  this
```

```
    .sendAnalytics();
```

```
}
```

Putting everything together

```
class
```

```
  SDK
```

```
 {
```

```
  constructor
```

```
  () {
```

```
    // hold the events
```

this

.queue = [];

// track the count

this

.count =

1

;

}

// push event in the queue

logEvent(ev) {

this

.queue.push(ev);

}

// function to delay the execution

wait =

()

=>

new

Promise

((

resolve, reject

) => {

setTimeout(

()

=> {

// reject every n % 5 time

if

(

this

.count %

5

===

0

){

reject();

}

else

{

resolve();

}

```
},  
1000  
);  
});  
  
// to send analytics  
  
// recursively send the events  
  
sendAnalytics =  
  
async  
  
function  
  
(){  
  
// if there are no events in the queue  
  
// stop execution  
  
if  
  
(  
  
this  
  
.queue.length ===  
  
0  
  
) {  
  
return  
  
;
```



```
}
```

```
// get the first element from the queue
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

356

```
const
```

```
current =
```

```
this
```

```
.queue.shift();
```

```
try
```

```
{
```

```
// delay
```

```
await
```

```
this
```

```
.wait();
```

```
// print the event
```

```
// can perform any other operations as well
```

```
like making api call
```

```
console
```

```
.log(
```

```
"Analytics sent "
```

```
+ current);  
  
// increase the count  
  
this  
  
.count++;  
  
}  
  
catch  
  
(e){  
  
// if execution fails  
  
console  
  
.log(  
  
"-----"  
  
);  
  
console  
  
.log(  
  
"Failed to send "  
  
+ current);  
  
console  
  
.log(  
  
"Retrying sending "  
  
+ current);
```

```
console
.log(
  "-----"
);
// reset the count
this
.count =
1
;
// push the event back into the queue
this
.queue.unshift(current);
}
finally
{
// recursively call the same function
// to send the remaining
this
.sendAnalytics();
}
```

```
}  
  
// start the execution  
  
send =  
  
async  
  
function  
  
(){  
  
this  
  
.sendAnalytics();  
  
}  
  
}
```

Test Case

Input:

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

357

const

sdk =

new

SDK();

sdk.logEvent(  
"event 1"

```
);  
sdk.logEvent(  
"event 2"  
);  
sdk.logEvent(  
"event 3"  
);  
sdk.logEvent(  
"event 4"  
);  
sdk.logEvent(  
"event 5"  
);  
sdk.logEvent(  
"event 6"  
);  
sdk.logEvent(  
"event 7"  
);  
sdk.logEvent(  

```

"event 8"

);

sdk.logEvent(

"event 9"

);

sdk.logEvent(

"event 10"

);

sdk.logEvent(

"event 11"

);

sdk.logEvent(

"event 12"

);

sdk.logEvent(

"event 13"

);

sdk.logEvent(

"event 14"

);

```
sdk.logEvent(  
  "event 15"  
);
```

```
sdk.logEvent(  
  "event 16"  
);
```

```
sdk.logEvent(  
  "event 17"  
);
```

```
sdk.logEvent(  
  "event 18"  
);
```

```
sdk.logEvent(  
  "event 19"  
);
```

```
sdk.logEvent(  
  "event 20"  
);
```

```
sdk.send();
```

Output:

"Analytics sent event 1"

"Analytics sent event 2"

"Analytics sent event 3"

"Analytics sent event 4"

"-----"

"Failed to send event 5"

"Retrying sending event 5"

"-----"

"Analytics sent event 5"

"Analytics sent event 6"

"Analytics sent event 7"

"Analytics sent event 8"

"-----"

"Failed to send event 9"

"Retrying sending event 9"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

358

"-----"

"Analytics sent event 9"

"Analytics sent event 10"



"Analytics sent event 11"

"Analytics sent event 12"

"-----"

"Failed to send event 13"

"Retrying sending event 13"

"-----"

"Analytics sent event 13"

"Analytics sent event 14"

"Analytics sent event 15"

"Analytics sent event 16"

"-----"

"Failed to send event 17"

"Retrying sending event 17"

"-----"

"Analytics sent event 17"

"Analytics sent event 18"

"Analytics sent event 19"

"Analytics sent event 20"

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

Check if given binary tree is full

Problem Statement -

Given a binary tree, check whether it is a full binary tree or not.

A full B-tree is defined as a binary tree in which all nodes have either zero or two child nodes.

Alternatively we can say there is no node in a full B-tree, which has only one child node.

Example

Input :

1

/ \

2

3

/ \

4

5

Output : Yes

Input :

1

/ \

2

3

/

4

Output :No

Recursive Approach

- Create a function which recursively calls itself to check if each node has 0 or 2 children or not.
- If the given node is empty then it's a full binary tree.
- If a given node has no child then also it is full.
- If it has left and right both children then it is full.
- Otherwise it is not a full binary tree.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

360

const

isFullTreeRecursive = (

root

) => {

// if empty tree

if

(root ==

null

```

){
return
true
;
}
// if leaf node
if
(root.left ===
null
&& root.right ===
null
) {
return
true
;
}
// if both left and right subtrees are not null
// the are full
if
((root.left !==

```

```
    null
  ) && (root.right !==
    null
  ))
{
  return
    (isFullTreeRecursive(root.left) &&
      isFullTreeRecursive(root.right));
}

// if none work
return
  false
;
}
```

Test Case

Input:

function

Node

(

val

```
) {  
  this  
    .val = val;  
  this  
    .left =  
      null  
    ;  
  this  
    .right =  
      null  
    ;  
}  
let  
tree =  
new  
Node(  
  10  
)  
tree.left =  
new
```

```
Node(  
  20  
);  
tree.right =  
new  
Node(  
  30  
);  
tree.left.right =  
new  
Node(  
  40  
);  
console  
.log(isFullTreeRecursive(tree));
```

Output:

false

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

361

Iterative Approach

We can implement the above recursive solution iteratively using a queue.

- Add the root to the queue.
- Keep iterating while the queue is not empty.
- In each iteration get the first value from the queue and check if it has 0 or 2 children then continue. Otherwise return false.
- Add those child's back to the queue to check their children and so on.

```
const
```

```
isFullTreeIterative = (
```

```
  root
```

```
) => {
```

```
  if
```

```
    (root ===
```

```
      null
```

```
    ){
```

```
      return
```

```
      true
```

```
    ;
```

```
  }
```

```
const
```

```
q = [];
```



```
//Add the root to the queue initially  
q.unshift(root);  
  
// traverse all the nodes of the binary tree  
// level by level until queue is empty  
while  
(q.length){  
  const  
  node = q.pop();  
  
  //If it is a leaf then continue  
  if  
  (node.left ===  
  null  
  && node.right ===  
  null  
  ){  
    continue  
  }  
  
  // if either of the child is not null and the  
  // other one is null, then binary tree is not
```

```
// a full binary tree
```

```
if
```

```
(node.left ===
```

```
null
```

```
|| node.right ===
```

```
null
```

```
)
```

```
{
```

```
return
```

```
false
```

```
;
```

```
}
```

```
// push left and right childs of 'node'
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

362

```
// on to the queue 'q'
```

```
q.unshift(node.left);
```

```
q.unshift(node.right);
```

```
}
```

```
// binary tree is a full binary tree
```

```
return
```

```
true
```

```
;
```

```
}
```

Test Case

Input:

```
function
```

```
Node
```

```
(
```

```
val
```

```
) {
```

```
this
```

```
.val = val;
```

```
this
```

```
.left =
```

```
null
```

```
;
```

```
this
```

```
.right =
```

```
null
```

```
;
}
let
tree =
new
Node(
10
);
tree.left =
new
Node(
20
);
tree.right =
new
Node(
30
);
tree.left.right =
new
```

```
Node(  
  40  
);  
tree.left.left =  
new  
Node(  
  50  
);  
console  
.log(isFullTreeIterative(tree));
```

Output:

true

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

363

Find height and width of binary tree

Problem Statement -

Given a binary tree, write two programs to find each height and width of it.

Example

Input :

1

/ \

2

3

/ \ / \

4

5 6 7

Output :

Height = 3

Width = 4

Height of a binary tree

Height of a binary tree is the maximum depth of the tree.

To find out the depth of the tree we will have to perform the following operations.

We will recursively find out the depth of the left subtree and right subtree. Then get the max of both depths and return it.

const

btHeight = (

node

) => {

// if null return 0

if

```
(node ===  
null  
) {  
  return  
  0  
;  
}  
  
// get left subtree height  
  
const  
leftHeight = btHeight(node.left);  
  
© JavaScript Interview Guide | learnersbucket.com  
  
364  
  
// get right subtree height  
  
const  
rightHeight = btHeight(node.right);  
  
// return the max of them  
  
return  
leftHeight > rightHeight ? leftHeight +  
1  
: rightHeight +
```

1

;

}

Test Case

Input:

function

Node

(

val

) {

this

.val = val;

this

.left =

null

;

this

.right =

null

;



```
}  
  
let  
  
tree =  
  
new  
  
Node(  
  
10  
  
);  
  
tree.left =  
  
new  
  
Node(  
  
20  
  
);  
  
tree.right =  
  
new  
  
Node(  
  
30  
  
);  
  
tree.left.right =  
  
new  
  
Node(  

```

```
40
);
tree.left.left =
new
Node(
50
);
tree.right.right =
new
Node(
70
);
tree.right.left =
new
Node(
60
);
console
.log(btHeight(tree));
```

Output:

3

Width of a binary tree

The width of a binary tree is the number of nodes present at the given level.

We will first find the height of the tree and then find the width by recursively checking the number of nodes at every level. Then we will return the maximum width of them.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

365

```
const
```

```
helper = (
```

```
node, level
```

```
) => {
```

```
// if null return 0
```

```
if
```

```
(node ===
```

```
null
```

```
){
```

```
return
```

```
0
```

```
;
```

```
}
```

```
// if at root level return 1

if
(level ===
1
)
return
1
;

// else recursively find the width at each level
if
(level >
1
){
return
helper(node.left, level -
1
) + helper(node.right,
level -
1
);
```

```
};  
  
return  
  
0  
  
;  
  
};  
  
const  
  
btWidth = (  
  
node  
  
) => {  
  
let  
  
maxWidth =  
  
0  
  
;  
  
let  
  
width, height = btHeight(node);  
  
/* get width of each level and compare  
the width with maximum width so far */  
  
for  
  
(  
  
let
```

```
i =  
1  
; i <= height; i++)  
{  
width = helper(node, i);  
if  
(width > maxWidth) {  
maxWidth = width;  
}  
}  
return  
maxWidth;  
};
```

Test Case

Input:

function

Node

```
(  
val  
) {
```

```
this
.val = val;

this
.left =
null
;

this
.right =
null
;

}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

366

```
let
tree =
new
Node(
10
);

tree.left =
```

```
new
Node(
20
);
tree.right =
new
Node(
30
);
tree.left.right =
new
Node(
40
);
tree.left.left =
new
Node(
50
);
tree.right.right =
```



```
new
Node(
70
);
tree.right.left =
new
Node(
60
);
console
.log(btWidth(tree));
```

Output:

4

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

367

Polyfill for extend

Problem Statement -

Write a simple polyfill for the [extends](#) method. Create a simple function that will accept the two functions parent and child and extend the child function to have all parent properties.

Example

```
function
```

```
Parent
```

```
() {
```

```
this
```

```
.name =
```

```
"abc"
```

```
;
```

```
};
```

```
Parent.prototype.walk =
```

```
function
```

```
() {
```

```
console
```

```
.log (
```

```
this
```

```
.name +
```

```
', I am walking!'
```

```
);
```

```
};
```

```
function
```

```
Child
```

```
() {  
  this  
    .name =  
      "pqr"  
    ;  
  };  
  
  Child.prototype.sayHello =  
    function  
      () {  
        console  
          .log(  
            'hi, I am a student'  
          );  
      };  
  
    // function to extend  
    extend(Parent, Child);  
  
    const  
      child =  
        new  
          Child();
```

```
child.sayHello();
```

```
child.walk();
```

```
console
```

```
.log(child
```

```
instanceof
```

```
Parent);
```

```
console
```

```
.log(child
```

```
instanceof
```

```
Child);
```

Output:

```
"hi, I am a student"
```

```
"pqr, I am walking!"
```

```
true
```

```
true
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

368

To extend a child from a parent, we must copy all the static and non-static methods of the complete prototype chain.

And for this, we can either update the reference of the child's

prototype to point to the parent or use the

[Object.setPrototypeOf\(child, parent\)](#) to create the link.

Here I have used one method for extending and commented on the second. You can choose between either based on your preference.

```
const
```

```
myExtends = (
```

```
SuperType, SubType
```

```
) => {
```

```
// extend the child to point to the parent
```

```
// all the nonstatic methods
```

```
SubType.prototype.__proto__ = SuperType.prototype;
```

```
//ES5: Object.setPrototypeOf(SubType.prototype,
```

```
SuperType.prototype);
```

```
// static methods;
```

```
SubType.__proto__ = SuperType;
```

```
//ES5: Object.setPrototypeOf(SubType, SuperType);
```

```
// as the child is pointing to the parent
```

```
// after it, update the child's constructor to point
```

```
itself.
```

```
SubType.prototype.constructor = SubType;
```

```
//ES5: Object.setPrototypeOf(SubType.prototype,
```

```
SubType.prototype);
```

```
}
```

Test Case

Input:

```
// Parent
```

```
function
```

```
Person
```

```
() {
```

```
this
```

```
.name =
```

```
"abc"
```

```
;
```

```
}
```

```
// non static methods
```

```
Person.prototype.walk =
```

```
function
```

```
() {
```

```
console
```

```
.log (
```

```
this
```

```
.name +  
' , I am walking!'  
);  
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

369

```
Person.prototype.sayHello =  
function  
{  
  console  
    .log (  
      'hello'  
    );  
};  
  
// static methods
```

```
Person.staticSuper =  
function  
{  
  console  
    .log(  
      'hello'  
    );  
};
```

```
'static'  
  
);  
  
};  
  
// child  
  
function  
  
Student  
  
() {  
  
  this  
  
    .name =  
  
    "pqr"  
  
  ;  
  
}  
  
// sayHello  
  
// this will replace the parent after extending  
  
Student.prototype.sayHello =  
  
function  
  
(){  
  
  console  
  
    .log(  
  
    'hi, I am a student'
```



```
);  
}  
  
// add sayGoodBye method  
  
Student.prototype.sayGoodBye =  
  
function  
  
(){  
  
console  
  
.log(  
  
'goodBye'  
  
);  
  
}  
  
const  
  
Extend = myExtends(Person, Student);  
  
const  
  
student1 =  
  
new  
  
Student();  
  
student1.sayHello();  
  
student1.walk();  
  
student1.sayGoodBye();
```

```
Student.staticSuper();
```

```
console
```

```
.log(student1.name);
```

```
// check inheritance
```

```
console
```

```
.log(student1
```

```
instanceof
```

```
Person);
```

```
console
```

```
.log(student1
```

```
instanceof
```

```
Student);
```

Output:

```
"hi, I am a student"
```

```
"pqr, I am walking!"
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

370

```
"goodBye"
```

```
"static"
```

```
"pqr"
```

true

true

© JavaScript Interview Guide | learnersbucket.com

371

Animate elements in a sequence

Problem Statement -

- Implement a loading bar that animates from 0 to 100% in 3 seconds.
- Start loading bar animation upon a button click.
- Queue multiple loading bars if the button is clicked more than once. Loading bar N starts animating with loading bar N-1 is done animating.
- Follow up Start loading N bar after N-1 is half done (50%).

Let's get started, we can individually tackle the sub-problems.

A loading bar that animates

Create a div dynamically through JavaScript.

const

loadingBar =

document

.createElement(

```
"div"
```

```
);
```

Apply dynamic animation keyframes to the element.

```
let
```

```
styleSheet =
```

```
null
```

```
;
```

```
const
```

```
dynamicAnimation = (
```

```
name, styles
```

```
) => {
```

```
//create a stylesheet
```

```
if
```

```
(!styleSheet) {
```

```
styleSheet =
```

```
document
```

```
.createElement(
```

```
"style"
```

```
);
```

```
styleSheet.type =
```

```

"text/css"

;

document

.head.appendChild(styleSheet);

}

//insert the new key frames

styleSheet.sheet.insertRule(

`@keyframes ${name} ${styles}`

,

styleSheet.length

);

};

```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

372

Using this function we can dynamically add the keyframes of the animation and then apply these animations to any element.

```

dynamicAnimation(

"loadingBar"

,

`

0%{

```

```
width: 0%;  
  
}  
  
100%{  
width: 100%;  
}`  
);  
  
loadingBar.style.height =  
"10px"  
;  
  
loadingBar.style.backgroundColor =  
"Red"  
;  
  
loadingBar.style.width =  
"0"  
;  
  
loadingBar.style.animation =  
"loadingBar 3s forwards"  
;
```

We are done creating the loading bar, just need to add it to the DOM to animate, for which we will get an entry element and append this

into that.

```
const
```

```
entry =
```

```
document
```

```
.getElementById(
```

```
"entry"
```

```
);
```

```
entry.appendChild(loadingBar);
```

Wrap everything inside a function and invoke it to generate a loading bar.

You can also pass the duration to this function (how long the

animation should run) as well as the keyframes itself.

```
const
```

```
generateLoadingBar =
```

```
()
```

```
=> {
```

```
//create a div
```

```
const
```

```
loadingBar =
```

```
document
```

```
.createElement(
```

```
"div"
```

```
);
```

```
//apply styles
```

```
dynamicAnimation(
```

```
"loadingBar"
```

```
,
```

```
`
```

```
0%{
```

```
width: 0%;
```

```
}
```

```
100%{
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

373

ADD ANIMATION

```
width: 100%;
```

```
`
```

```
);
```

```
loadingBar.style.height =
```

```
"10px"
```

```
;
```

```
loadingBar.style.backgroundColor =
```



```
"Red"

;

loadingBar.style.width =

"0"

;

loadingBar.style.marginBottom =

"10px"

;

loadingBar.style.animation =

"loadingBar 3s forwards"

;

//append the div

const

entry =

document

.getElementById(

"entry"

);

entry.appendChild(loadingBar);

};
```

Start loading bar animation upon a button click.

Create a button and on its click invoke the above function so that it will generate the loading bar and will be animated once added to the DOM.

```
//on btn click, generate the loading bar
```

```
document
```

```
.getElementById(
```

```
"btn"
```

```
).addEventListener(
```

```
"click"
```

```
,
```

```
(e) => {
```

```
  generateLoadingBar();
```

```
});
```

Queue multiple loading bars if the button is clicked more than once and load the next one once the previous animation is finished.

The last part of this question is to queue the loading bars when the button is clicked multiple times and animate them sequentially one

after another.

Create a global variable count and increment its value by one every

time the button is clicked, vice-versa decrease its value by one, when a loading bar is animated.

374

```
//global variable to track the count of loading bars
```

```
let
```

```
count =
```

```
0
```

```
;
```

```
//function to update the count
```

```
const
```

```
updateCount = (
```

```
val
```

```
) => {
```

```
count += val;
```

```
document
```

```
.getElementById(
```

```
"queueCount"
```

```
).innerText
```

```
= count;
```

```
};
```

For generating the next loading bar from the queue, we will have to

recursively call the same function (which generates the loading bar) when the animation of the previous loading bar is done.

Thankfully we have an event for that, `animationend` which is fired

every time an animation ends on the element, this is also a reason why I choose CSS animation over JavaScript timers to animate elements.

When the `animationend` is triggered, recursively call the same function to generate the loading bar and update the queue count.

```
//on animation end
```

```
loadingBar.addEventListener(
```

```
"animationend"
```

```
, () =>
```

```
{
```

```
//decrease the count
```

```
updateCount(
```

```
-1
```

```
);
```

```
if
```

```
(count >
```

```
0
```

```
) {
```

```
//generate the loading bar
```

```
generateLoadingBar();
```

```
}  
  
});
```

We will also need to update the code on the button click, invoke the `generateLoadingBar` function only when the count is zero as all other subsequent calls will be invoked recursively.

Also, update the count on each click.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

375

```
//on btn click, generate the loading bar
```

```
document
```

```
.getElementById(
```

```
"btn"
```

```
).addEventListener(
```

```
"click"
```

```
,
```

```
(e) => {
```

```
//trigger animation
```

```
if
```

```
(count ===
```

```
0
```

```
) {
```

```
generateLoadingBar();
```

```
}
```

```
//update count
```

```
updateCount(
```

```
1
```

```
);
```

```
});
```

Putting everything together

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
<head>
```

```
<meta charset="UTF-8" />
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0"
```

```
/>
```

```
<title>Animate elements in sequence</title>
```

```
</head>
```

```
<body>
```

```
<div id="entry"></div>
```

```
<p><span>In Queue:</span><span id="queueCount">0</span></p>
```

```
<button id="btn">ADD ANIMATION</button>
```

```
<script>

// function to add keyframes dynamically

let styleSheet = null;

const dynamicAnimation = (name, styles) => {

//create a stylesheet

if (!styleSheet) {

styleSheet = document.createElement("style");

styleSheet.type = "text/css";

document.head.appendChild(styleSheet);

}

//insert the new key frames

styleSheet.sheet.insertRule(

`@keyframes ${name} ${styles}`,

styleSheet.length

);

© JavaScript Interview Guide | learnersbucket.com

376

};

//global variable to track the count of loading bars

let count = 0;
```

```
//function to update the count

const updateCount = (val) => {

count += val;

document.getElementById("queueCount").innerText = count;

};

//generate loading bars

const generateLoadingBar = () => {

//create a div elm

const loadingBar = document.createElement("div");

//apply styles

//animation keyframes

dynamicAnimation(

"loadingBar",

、

0%{

width: 0%;

}

100%{

width: 100%;

}

}
```



```

);

loadingBar.style.height = "10px";

loadingBar.style.backgroundColor = "Red";

loadingBar.style.width = "0";

loadingBar.style.marginBottom = "10px";

loadingBar.style.animation = "loadingBar 3s forwards";

//append the loading bar

const entry = document.getElementById("entry");

entry.appendChild(loadingBar);

//on animation end

loadingBar.addEventListener("animationend", () => {

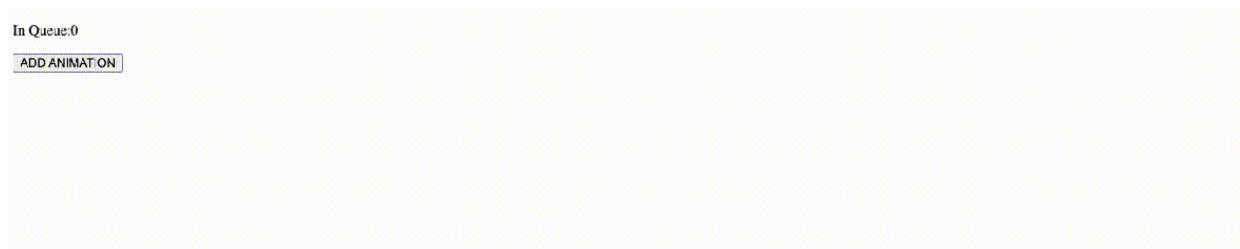
//decrease the count

updateCount(-1);

```

© JavaScript Interview Guide | learnersbucket.com

377



```

if (count > 0) {

//generate the loading bar

```

```
generateLoadingBar();

}

});

//remove listener

loadingBar.removeEventListener("animationend", () => {});

};

//on btn click, generate the loading bar

document.getElementById("btn").addEventListener("click", (e) => {

//trigger animation

if (count === 0) {

generateLoadingBar();

}

//update count

updateCount(1);

});

</script>

</body>

</html>
```

Follow-up:- Loading bar N starts animating with loading bar  
N-1 is done animating 50%.

In the follow-up, we have to start animating the Nth bar when the N-1th bar is half done.

Unfortunately, there are only four events associated with animations.

© JavaScript Interview Guide | learnersbucket.com

378

- *animationstart* :- When animation starts.
- *animationend* :- When animation ends.
- *animationcancel* :- When animation unexpectedly aborts without triggering animationend event.
- *animationiteration* :- When an iteration of animation ends and next one begins. This event is not triggered at the same time as the animationend event.

There is no way to determine how much animation has been completed.

To solve this problem, we use a hack, a workaround, we animate two elements simultaneously, one which runs on normal duration and the other which runs for the duration when the next animation has to be triggered.

For example, the next animation should trigger when the first loading bar is 50% done, thus let's say our original loading bar is going to complete 100% animation in 3 seconds, which means the next

animation should be triggered when it is 50% done in 1.5 seconds (half time).

We will parallelly animate another element for that duration and on its animationend trigger the next rendering.

```
//generate loading bars

const

generateLoadingBar =

()

=> {

//fragment

const

fragment =

document

.createDocumentFragment();

//create a div elm

const

loadingBar =

document

.createElement(

"div"

);

//apply styles
```

379

```
//animation keyframes
```

```
dynamicAnimation(
```

```
"loadingBar"
```

```
,
```

```
`
```

```
0%{
```

```
width: 0%;
```

```
}
```

```
100%{
```

```
width: 100%;
```

```
`
```

```
);
```

```
loadingBar.style.height =
```

```
"10px"
```

```
;
```

```
loadingBar.style.backgroundColor =
```

```
"Red"
```

```
;
```

```
loadingBar.style.width =
```

"0"

;

loadingBar.style.marginBottom =

"10px"

;

loadingBar.style.animation =

"loadingBar 3s forwards"

;

//create shadow loading bar

const

shadowLoadingBar =

document

.createElement(

"div"

);

//apply styles

//animation keyframes

dynamicAnimation(

"shadowLoadingBar"

,

,

```
0%{
```

```
width: 0%;
```

```
}
```

```
100%{
```

```
width: 50%;
```

```
}`
```

```
);
```

```
//it will be hidden
```

```
shadowLoadingBar.style.height =
```

```
"5px"
```

```
;
```

```
shadowLoadingBar.style.backgroundColor =
```

```
"green"
```

```
;
```

```
shadowLoadingBar.style.width =
```

```
"0"
```

```
;
```

```
shadowLoadingBar.style.marginBottom =
```

```
"10px"
```

```
;
```

```
shadowLoadingBar.style.animation =
```

```
"shadowLoadingBar
```

```
1.5s forwards"
```

```
;
```

```
//add the both the bars to the fragment
```

```
fragment.appendChild(loadingBar);
```

```
fragment.appendChild(shadowLoadingBar);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

380



```
//append the loading bar
```

```
const
```

```
entry =
```

```
document
```

```
.getElementById(
```

```
"entry"
```

```
);
```



```
entry.appendChild(fragment);

//on animation end on shadow bar
shadowLoadingBar.addEventListener(
  "animationend"
  ,
  () => {
    //decrease the count
    updateCount(
      -1
    );
    if
    (count >
      0
    ) {
      //generate the loading bar
      generateLoadingBar();
    }
  });

//remove listener
shadowLoadingBar.removeEventListener(
```

```
"animationend"
```

```
,
```

```
() => {});
```

```
};
```

If you notice, we are creating two loading bars and adding them in fragments, and hard-coded the duration of the shadow bar 1.5 and width to 50%. You can make it dynamic by using a simple mathematical calculation.

Also, we have kept the background of the shadow bar green and it is visible currently (just to show the working), but you can hide it.

Everything else remains the same.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

381

### LocalStorage with expiry

#### Problem Statement -

Extend the local storage to accept an expiry time and expire the entry after that time.

#### Example

```
// set 'bar' on 'foo' that will expiry after 1000 milliseconds
```

```
myLocalStorage.setItem(
```

```
'foo'
```

```
,  
'bar'  
  
,  
1000  
  
);  
  
// after 2 seconds  
  
console  
  
.log(myLocalStorage.getItem(  
  
'foo'  
  
));  
  
// null
```

To implement this we will override the existing local storage method.

While adding the entry we will accept the expiry date in milliseconds (30 days by default). Set the expiry date to time from the current date along with the value and store it in the original local storage.

```
// add an entry  
  
// default expiry is 30 days in milliseconds  
  
setItem(key, value, maxAge =  
  
30  
  
*  
  
60
```

\*

60

\*

1000

)

{

// store the value as the object

// along with the expiry date

let

result = {

data : value

}

if

(maxAge){

// set the expiry

// from the current date

result.expireTime =

Date

.now() + maxAge;

}

```
// stringify the result
```

© JavaScript Interview Guide | learnersbucket.com

382

```
// and the data in original storage
```

```
window
```

```
.localStorage.setItem(key,
```

```
JSON
```

```
.stringify(result));
```

```
}
```

Likewise, while getting the value for the given key, check if there is a value associated with the key, if it exists and is not expired then return the value, else remove the entry and return null.

```
getItem(key) {
```

```
// get the parsed value of the given key
```

```
let
```

```
result =
```

```
JSON
```

```
.parse(
```

```
window
```

```
.localStorage.getItem(key));
```

```
// if the key has value
```

```
if
(
    result
){
    // if the entry is expired

    // remove the entry and return null

    if
    (
        result.expireTime <=
        Date
        .now()
    ){
        window
        .localStorage.removeItem(key);

        return
        null
    ;
    }

    // else return the value

    return
    result.data;
}

// if the key does not have value

return
```

```
null
```

```
;
```

```
}
```

Putting both together

```
window
```

```
.myLocalStorage = {
```

```
getItem(key) {
```

```
// get the parsed value of the given key
```

```
let
```

```
result =
```

```
JSON
```

```
.parse(
```

```
window
```

```
.localStorage.getItem(key));
```

```
// if the key has value
```

```
if
```

```
(result){
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

383

```
// if the entry is expired
```

```
// remove the entry and return null

if
(result.expireTime <=
Date
.now()){
window
.localStorage.removeItem(key);
return
null
;
}

// else return the value
return
result.data;
}

// if the key does not have value
return
null
;
},
```



```
// add an entry

// default expiry is 30 days in milliseconds

setItem(key, value, maxAge =

30

*

60

*

60

*

1000

)

{

// store the value as object

// along with expiry date

let

result = {

data : value

}

if

(maxAge){
```

```
// set the expiry

// from the current date

result.expireTime =

Date

.now() + maxAge;

}

// stringify the result

// and the data in original storage

window

.localStorage.setItem(key,

JSON

.stringify(result));

},

// remove the entry with the given key

removeItem(key) {

window

.localStorage.removeItem(key);

},

// clear the storage
```

384

```
clear() {  
  window  
    .localStorage.clear();  
}  
};
```

Test Case

Input:

```
myLocalStorage.setItem(  
  'foo'  
  ,  
  'bar'  
  ,  
  1000  
);  
setTimeout(  
  ()  
=> {  
  console  
    .log(myLocalStorage.getItem(  
      'foo'    ));  
});
```

```
'foo'
```

```
));
```

```
},
```

```
1500
```

```
);
```

Output:

```
null
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

385

Create your custom cookie

Problem Statement -

Create your custom cookie that is available on the [document](#) object with the only max-age option.

Example

```
document
```

```
.myCookie =
```

```
"blog=learnersbucket"
```

```
;
```

```
document
```

```
.myCookie =
```

```
"name=prashant;max-age=1"
```

```
;
//
this will expire after 1
second
console
.log(
document
.myCookie);
// "blog=learnersbucket; name=prashant"
setTimeout(
()
=> {
console
.log(
document
.myCookie);
},
1500
);
// "blog=learnersbucket"
```

```
// "name=prashant" is expired after 1 second
```

As our custom cookie is available on the document object, we will

extend the document using `Object.defineProperty()` and add the

custom property `"myCookie"`, this will give us the addition methods like `get()` and `set()` that could be used behind the scene to make the cookie entry and return it.

To implement this, we will create a function with a [map](#) to persist the entry of each cookie.

- As the cookie string is colon separated values with data and

options, in the `set()` method we will extract the data and separate it on the key and value and also the options (max-age) and store

them in the map.

- While getting the cookie, get all the entries of the map and for each entry check if it has expired or not. If it is expired, delete the

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

386

entry. Send the remaining entry as a string with a colon-separated.

To parse the input and separate the data and options, we will be using a helper function.

```
// helper function to parse the
```

```
// key-value pairs
```

```
function
```

```
parseCookieString
```

```
(  
  str  
) {  
  // split the string on ;  
  // separate the data and the options  
  const  
  [nameValue, ...rest] = str.split(  
    ';' )  
  );  
  // get the key value separated from the data  
  const  
  [key, value] = separateKeyValue(nameValue);  
  // parse all the options and store it  
  // like max-age  
  const  
  options = {};  
  for  
  (  
    const  
    option
```

```
of
rest) {
  const
  [key, value] = separateKeyValue(option);
  options[key] = value;
}
return
{
  key,
  value,
  options,
}
}
// helper function
// to separate key and value
function
separateKeyValue
(
  str
) {
```



```
return  
  
str.split(  
  
'='  
  
).map(  
  
s  
  
=> s.trim());  
  
}
```

For the main logic we can extend the document object with  
`Object.defineProperty()` .

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

387

```
// enable myCookie  
  
function  
  
useCustomCookie  
  
() {  
  
// to store the key and value  
  
// of each cookie  
  
const  
  
store =  
  
new
```

```
Map
();
Object
.defineProperty(
document
,
'myCookie'
, {
configurable:
true
,
get() {
const
cookies = [];
const
time =
Date
.now());
// get all the entries from the store
for
```

```
(  
  const  
  [name, { value, expires }]  
  of  
  store)  
  {  
    // if the entry is expired  
    // remove it from the store  
    if  
      (expires <= time) {  
        store.delete(name);  
      }  
    // else push the key-value pair in the cookies  
    array  
    else  
    {  
      cookies.push(  
        `${name}=${value}`  
      );  
    }  
  }
```

```
}  
  
// return all the key-value pairs as a string  
  
return  
  
cookies.join(  
  
'; '  
  
);  
  
},  
  
set(val) {  
  
// get the key value of the data  
  
// and option from the string  
  
const  
  
{ key, value, options } = parseCookieString(val);  
  
// if option has max-age  
  
// set the expiry date  
  
let  
  
expires =  
  
Infinity  
  
;  
  
if  
  
(options[
```

```
"max-age"
```

```
) {
```

```
  expires =
```

```
    Date
```

```
      .now() +
```

```
      Number
```

```
        (options[
```

```
          "max-age"
```

```
        ])
```

```
      * 
```

```
      1000
```

```
    ;
```

```
  }
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

388

```
// add the entry in the store
```

```
store.set(key, { value, expires });
```

```
}
```

```
})
```

```
};
```

Test Case

Input:

```
useCustomCookie();
```

```
document
```

```
.myCookie =
```

```
"blog=learnersbucket"
```

```
;
```

```
// this will expire after 1 second
```

```
document
```

```
.myCookie =
```

```
"name=prashant;max-age=1"
```

```
;
```

```
console
```

```
.log(
```

```
document
```

```
.myCookie);
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
console
```

```
.log(  
document  
.myCookie);  
  
},  
  
1500  
  
);
```

Output:

```
"blog=learnersbucket; name=prashant"
```

```
"blog=learnersbucket"
```

© JavaScript Interview Guide | learnersbucket.com

389

## Create an Immutability Helper - Part 1

### Problem Statement -

Implement a simple immutability helper that allows a certain set of actions to update the [frozen input object](#). The input object can only be updated through this function and the returned value is also frozen.

Note – For simplicity, only one operation is allowed at a time.

### Actions

`_push_` : Array – Pushes the destination array in the input array.

```
const
```

```
inputArr = [
```

1

,

2

,

3

,

4

]

const

outputArr = update(

inputArr, {\_push\_: [

5

,

6

,

7

]}

);

console

.log(outputArr);



```
// [1,2,3,4,5,6,7]
```

`_replace_` : Object | Array – Replaces the destination value in the input object.

```
---
```

Object

```
---
```

```
const
```

```
state = {
```

```
  a: {
```

```
    b: {
```

```
      c:
```

```
        1
```

```
      }
```

```
    },
```

```
    d:
```

```
      2
```

```
    };
```

```
const
```

```
newState = update(
```

```
state,
```

```
{a: {b: { c: {_replace_:
```

3

}}}}

);

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

390

console

.log(newState);

/\*

{

"a": {

"b": {

"c": 3

}

},

"d": 2

}

\*/

---

Array

---

```
const
```

```
inputArr = [
```

```
  1
```

```
,
```

```
  2
```

```
,
```

```
  3
```

```
,
```

```
  4
```

```
]
```

```
const
```

```
outputArr = update(
```

```
  inputArr,
```

```
  {
```

```
    1
```

```
    : { _replace_:
```

```
      10
```

```
    } }
```

```
  );
```

```
console
```

```
.log(outputArr);
```

```
// [1,10,3,4]
```

`_merge_` : Object – Merges the destination value in the input object.

```
const
```

```
state = {
```

```
  a: {
```

```
    b: {
```

```
      c:
```

```
        1
```

```
      }  
    },
```

```
    d:
```

```
      2
```

```
    };
```

```
const
```

```
newState = update(  
  state,
```

```
  {a: {b: { _merge_ : {e:
```

```
    5
```

```
  }}}}
```

```
);  
  
console  
  
.log(newState);  
  
/*
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

391

```
{  
  
  "a": {  
  
    "b": {  
  
      "c": 1,  
  
      "e": 5  
  
    }  
  
  },  
  
  "d": 2  
  
}  
  
*/
```

*\_transform\_* : Object | Array – Transforms the destination value by passing it through this function.

```
const
```

```
inputArr = {a: { b:
```

2

```

});

const
outputArr = update(inputArr, {a: { b: {_transform_:
(
item
) => item *
2
}}});

console
.log(outputArr);

/*
{
"a": {
"b": 4
}
}
*/

```

We can create a helper function that will accept two arguments (data, action), the action is always an Object.

We can recursively deep traverse the object and in each call check if the current key is any of the actions then perform the action

accordingly.

Otherwise depending if the input an array or object recursively calls the same function with the current value to update.

Wrap this helper function inside another parent function. As the input object will be frozen, we cannot directly update it, thus we will create a

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

392

clone of it. Pass the clone for update through the helper function and in the end freeze the output before returning it back.

```
// function to deepfreeze

function
deepFreeze
(
  object
) {
  // get the property names defined on object
  let
    propNames =
      Object
        .getOwnPropertyNames(object);
  // recursively freeze all the properties
  for
```

```
(  
  let  
    name  
  of  
    propName) {  
  let  
    value = object[name];  
    object[name] = value &&  
    typeof  
    value ===  
    "object"  
  ?  
    deepFreeze(value) : value;  
  }  
  return  
    Object  
    .freeze(object);  
};  
  
// function to perform the action  
function
```



```
update
(
inputObj, action
){
const
clone =
JSON
.parse(
JSON
.stringify(inputObj));
function
helper
(
target, action
) {
// iterate the entries of the action
for
(
const
[key, value]
```

of

Object

.entries(action))

{

// if the key is of action type

// perform the action

switch

(key) {

// add a new value

case

'\_push\_'

:

return

[...target, ...value];

// replace the entry

case

'\_replace\_'

:

return

value;

```
// merge the values
```

```
case
```

```
'_merge_'
```

```
:
```

```
if
```

```
(!(target
```

```
instanceof
```

```
Object
```

```
)) {
```

```
throw
```

```
Error
```

```
(
```

```
"bad merge"
```

```
);
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
393
```

```
}
```

```
return
```

```
{...target, ...value};
```

```
// add the transformed value
```

```
case
'_transform_'
:
return
value(target);
// for normal values
default
:
// if it is an array
if
(target
instanceof
Array
) {
// create a copy
const
res = [...target];
// update the value
res[key] = update(target[key], value);
// return after update
```

```
return  
  
res;  
  
}  
  
// if it is an object  
  
else  
  
{  
  
// recursively call the same function  
  
// and update the value  
  
return  
  
{  
  
...target,  
  
[key]: update(target[key], value)  
  
}  
  
}  
  
}  
  
};  
  
};  
  
// perform the operation  
  
const  
  
output = helper(clone, action);
```

```
// freeze the output
```

```
deepFreeze(output);
```

```
//return it
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

394

```
return
```

```
output;
```

```
};
```

Test Case 1: `_push_`

```
const
```

```
inputArr = [
```

```
1
```

```
,
```

```
2
```

```
,
```

```
3
```

```
,
```

```
4
```

```
];
```

```
// deep freeze object
```

```
deepFreeze(inputArr);  
  
const  
outputArr = update(  
inputArr, {_push_: [  
5  
  
,  
6  
  
,  
7  
  
]}  
);  
  
// won't update as output is deep freezed  
  
outputArr[  
0  
  
] =  
10  
  
;  
  
console  
.log(outputArr);  
  
// [1,2,3,4,5,6,7]
```

Test Case 2: `_replace_`

```
const
```

```
state = {
```

```
  a: {
```

```
    b: {
```

```
      c:
```

```
        1
```

```
      }
```

```
    },
```

```
    d:
```

```
      2
```

```
  };
```

```
// freeze the object
```

```
deepFreeze(state);
```

```
const
```

```
newState = update(
```

```
  state,
```

```
  {a: {b: { c: {_replace_:
```

```
    3
```

```
  }}}}
```



```
);
```

```
// does not updates
```

```
// as output is frozen
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
395
```

```
newState.a.b.c =
```

```
10
```

```
;
```

```
console
```

```
.log(newState);
```

```
/*
```

```
{
```

```
"a": {
```

```
"b": {
```

```
"c": 3
```

```
}
```

```
},
```

```
"d": 2
```

```
}
```

```
*/
```

Test Case 3: \_merge\_

```
const
```

```
state = {
```

```
  a: {
```

```
    b: {
```

```
      c:
```

```
        1
```

```
      }
```

```
    },
```

```
    d:
```

```
      2
```

```
  };
```

```
// freeze the object
```

```
deepFreeze(state);
```

```
const
```

```
newState = update(
```

```
  state,
```

```
  {a: {b: { _merge_ : {e:
```

```
    5
```

```
  }}}}
```

```
);  
  
// does not updates  
  
// as output is frozen  
  
newState.a.b.e =  
  
10  
  
;  
  
console  
  
.log(newState);  
  
/*  
  
{
```

```
"a": {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
396
```

```
"b": {
```

```
"c": 1,
```

```
"e": 5
```

```
}
```

```
},
```

```
"d": 2
```

```
}
```

```
*/
```

Test Case 4: `_transform_`

```
const
```

```
state = {a: { b:
```

```
2
```

```
}};
```

```
// freeze the object
```

```
deepFreeze(state);
```

```
const
```

```
newState = update(state, {a: { b: {_transform_:
```

```
(
```

```
item
```

```
) => item *
```

```
2
```

```
}}});
```

```
// does not updates
```

```
// as output is frozen
```

```
newState.a.b =
```

```
10
```

```
;
```

console

.log(newState);

/\*

{

"a": {

"b": 4

}

}

\*/

© JavaScript Interview Guide | learnersbucket.com

397

Create an immutability helper – part 2

Problem Statement -

Create an immutability helper like [Immer produce\(\)](#) that allows modifications of the restricted objects.

Example

const

obj = {

a: {

b: {

c:

```
2
}
}
};

// object is frozen

// its properties cannot be updated

deepFreeze(obj);

// obj can only be updated through the produce function

const

newState = produce(obj, draft => {

draft.a.b.c =

3

;

draft.a.b.d =

4

;

});

console

.log(newState);

/*
```

```
{  
  "a": {  
    "b": {  
      "c": 3,  
      "d": 4  
    }  
  }  
}  
*/  
  
// newState will also be frozen  
// it cannot be updated  
  
delete  
  
newState.a.b.c;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

398

```
console  
  
.log(newState);  
  
/*  
  
{  
  "a": {
```

```
"b": {
```

```
"c": 3,
```

```
"d": 4
```

```
}
```

```
}
```

```
}
```

```
*/
```

To implement this we will first create a clone of the input obj and then pass this input object to the callback function of produce, for

processing.

After processing, perform a [deep comparison](#) to check if there is any change or not. If there is no change then return the original input, else return the new updated one.

[Deep freeze](#) the object before returning it so that it cannot be updated directly.

```
// function to deep freeze object
```

```
function
```

```
deepFreeze
```

```
(
```

```
object
```

```
) {
```

```
// get the property names defined on object
```



```
let
propNames =
Object
.getOwnPropertyNames(object);
// recursively freeze all the properties
for
(
let
name
of
propNames) {
let
value = object[name];
object[name] = value &&
typeof
value ===
"object"
?
deepFreeze(value) : value;
}
```

```
return
```

```
Object
```

```
.freeze(object);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

399

```
};
```

```
// function to deep check two objects
```

```
const
```

```
deepEqual = (
```

```
object1, object2
```

```
) => {
```

```
// get object keys
```

```
const
```

```
keys1 =
```

```
Object
```

```
.keys(object1);
```

```
const
```

```
keys2 =
```

```
Object
```

```
.keys(object2);
```

```
// if mismatched keys

if
(keys1.length !== keys2.length) {
return
false
;
}

for
(
const
key
of
keys1) {
// get the values
const
val1 = object1[key];
const
val2 = object2[key];
// if both values are objects
const
```

```
areObjects = val1 &&
typeof
val1 ===
"object"
&& val2 &&
typeof
val2 ===
"object"
;
// if are objects
if
(areObjects){
// deep check again
if
(!deepEqual(val1, val2)){
return
false
;
}
}
```

```
// if are not objects

// compare the values

else

if

(!areObjects && val1 !== val2){

return

false

;

}

}

return

true

;

};

// main function to update the value

function

produce

(

base, recipe

) {
```

```
// clone the frozen object
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
400
```

```
let
```

```
clone =
```

```
JSON
```

```
.parse(
```

```
JSON
```

```
.stringify(base));
```

```
// pass the clone to the recipe
```

```
// get the updated value
```

```
recipe(clone);
```

```
// if both are different
```

```
// update the value
```

```
if
```

```
(deepEqual(base, clone)) {
```

```
clone = base;
```

```
}
```

```
// deep freeze
```

```
deepFreeze(clone);
```

```
// return the clone
```

```
return
```

```
clone;
```

```
};
```

Test Case

```
const
```

```
obj = {
```

```
  a: {
```

```
    b: {
```

```
      c:
```

```
        2
```

```
    }
```

```
  }
```

```
};
```

```
// object is frozen
```

```
// its properties cannot be updated
```

```
deepFreeze(obj);
```

```
// obj can only be updated through the produce function
```

```
const
```

```
newState = produce(obj, draft => {
```

```
draft.a.b.c =
```

```
3
```

```
;
```

```
draft.a.b.d =
```

```
4
```

```
;
```

```
});
```

```
console
```

```
.log(newState);
```

```
/*
```

```
{
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
401
```

```
"a": {
```

```
"b": {
```

```
"c": 3,
```

```
"d": 4
```

```
}
```

```
}
```

```
}
```



```
*/  
  
// newState will also be frozen  
  
// it cannot be updated  
  
delete  
  
newState.a.b.c;  
  
console  
  
.log(newState);  
  
/*  
  
{  
  
"a": {  
  
"b": {  
  
"c": 3,  
  
"d": 4  
  
}  
  
}  
  
}  
  
*/
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

402

Make high priority Api call

## Problem Statement -

While you're throttling your requests, suppose a high-priority API call needs to be made, how would you manage it? aka. How to make

high-priority API calls in JavaScript.

There are two ways in which we can prioritize the API calls.

Using Request.Priority

The fetch method in the browser comes with an additional option to

[prioritize the API requests](#) .

It can be one of the following values: low, high, and auto. By default browsers fetch requests on high priority.

```
// articles list (high by default)
```

```
let
```

```
articles =
```

```
await
```

```
fetch(
```

```
'/articles'
```

```
);
```

```
// articles recommendation list (suggested low)
```

```
let
```

```
recommendation =
```

```
await
```

```
fetch(  
  '/articles/recommendation'  
  ,  
  {priority:  
    'low'  
  });
```

Using microtask queue

According to MDN –

A microtask is a short function which is executed after the function or program which created it exits and only if the [JavaScript execution stack](#) is empty, but before returning control to the event loop being used by the [user agent](#) to drive the script's execution environment.

Thus while [throttling](#) , the calls will be made after some delay using the timer functions.

© JavaScript Interview Guide | learnersbucket.com

403

We can make a high-priority call between the consecutive timer functions.

let

callback =

()

=> {

fetch(

```
'https://jsonplaceholder.typicode.com/todos/1'
```

```
)
```

```
.then(
```

```
response
```

```
=> response.json())
```

```
.then(
```

```
json
```

```
=>
```

```
console
```

```
.log(json))
```

```
}
```

```
let
```

```
callback2 =
```

```
()
```

```
=> {
```

```
fetch(
```

```
'https://jsonplaceholder.typicode.com/todos/2'
```

```
)
```

```
.then(
```

```
response
```

```
=> response.json()

.then(
  json
=>
  console
  .log(json))
}

let
urgentCallback =
()
=> {
  fetch(
    'https://jsonplaceholder.typicode.com/todos/3'
  )
  .then(
    response
    => response.json())
  .then(
    json
    =>
```

```
console
.log(json))
}
console
.log(
"Main program started"
);
setTimeout(callback,
0
);
setTimeout(callback2,
10
);
queueMicrotask(urgentCallback);
console
.log(
"Main program exiting"
);
Output:
"Main program started"
```

"Main program exiting"

{

"userId"

:

1

,

"id"

:

3

,

"title"

:

"fugiat veniam minus"

,

"completed"

:

false

}

{

"userId"

:

1

,

"id"

:

1

,

"title"

:

"delectus aut autem"

,

"completed"

:

false

}

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

404

{

"userId"

:



1

,

"id"

:

2

,

"title"

:

"quis ut nam facilis et officia qui"

,

"completed"

:

false

}

Note – The calls are made in priority, but the time at which they get resolved can vary, thus we can see random outputs.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

405

Convert JSON to HTML

Problem Statement -

Given a JSON as an object with type, children, and property of the

DOM element, write a function to convert the object to the actual DOM.

Example

Input:

```
const
json = {
  type:
  'div'
,
  props: { id:
  'hello'
, class:
  "foo"
},
  children: [
    {type:
    'h1'
, children:
    'HELLO'
},
```

```
{type:
'p'
, children: [{type:
'span'
, props: {class:
"bar"
}, children:
'World'
}] }
]
};
JSONtoHTML(json);
```

Output:

```
<div id=
"hello"
class=
"foo"
>
<h1>
HELLO
```

```
</h1>

<p>

<span class=
"bar"

>

World

</span>

</p>

</div>
```

The input JSON can be an object or an array of objects, thus it has to be handled appropriately.

The logic to implement this function is straightforward.

- Create a fragment to store the array of DOM elements.
- If the JSON element is an array of elements, iterate the array, for each entry in the array, create a new element of the given type,

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

406

assign all the properties, and if the children are an array of elements, recursively call the same function to generate the

HTML of the same, else set children as the innerText.

- Else if the JSON element is an object, convert it to an array and recursively call the same function so that it will form the DOM

out of it.

```
const
```

```
JSONtoHTML = (
```

```
  json
```

```
) => {
```

```
  // create a fragment
```

```
  const
```

```
  fragment =
```

```
  document
```

```
  .createDocumentFragment();
```

```
  if
```

```
  (
```

```
    Array
```

```
    .isArray(json)){
```

```
    // convert each entry of array to DOM element
```

```
    for
```

```
    (
```

```
      let
```

```
      entry
```

```
      of
```

```
json){  
  // create the element  
  
  const  
  
  element =  
  
  document  
  
  .createElement(entry.type);  
  
  // if props available  
  
  // set them  
  
  if  
  
  (entry.props){  
    for  
  
    (  
  
    let  
  
    key  
  
    in  
  
    entry.props){  
      element.setAttribute(key, entry.props[key]);  
    }  
  }  
  
  // if array of children
```

if

[OceanofPDF.com](http://OceanofPDF.com)

```
(  
  Array  
  .isArray(entry.children)){  
  // recursively convert the children to DOM  
  // and assign them  
  for  
  (  
    let  
    child  
    of  
    entry.children){  
    element.appendChild(JSONtoHTML(child));  
  }  
}  
  
// if children is string / text  
else  
{  
  element.innerText = entry.children;  
}
```



407

```
// add the element back to the fragment
fragment.appendChild(element);
}
}

// if not array recursively call the same function
// pass the entry as an array.
else
{
return
JSONtoHTML([json]);
}

return
fragment;
};
```

Test Case

Input:

const

JSON

= [

```
{  
  type:  
    'div'  
  ,  
  props: { id:  
    'hello'  
  , class:  
    'foo'  
  },  
  children: [  
    {type:  
      'h1'  
    , children:  
      'HELLO'  
    },  
    {type:  
      'p'  
    , children: [{type:  
        'span'  
      , props: {class:
```

"bar"

}, children:

'World'

}] }

]

},

{

type:

'section'

,

props: { id:

'hello-2'

, class:

"foo-2"

},

children: [

{type:

'h1'

, children:

'HELLO-2'

```
},  
{type:  
'p'  
, children: [{type:  
'span'  
, props: {class:  
"bar-2"  
}, children:  
'World'  
}] }  
]  
}];  
console  
.log(JSONtoHTML(json));
```

Output:

```
<div id=  
"hello"  
class=  
"foo"  
>
```

<h1>

HELLO

</h1>

<p>

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

408

<span class=

"bar"

>

World

</span>

</p>

</div>

<section id=

"hello-2"

class

=

"foo-2"

>

<h1>

HELLO-2

</h1>

<p>

<span class=

"bar-2"

>

World

</span>

<

/p>

</

section>

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

409

Convert HTML to JSON

Problem Statement -

Write a function that takes a DOM node as input and converts it to the JavaScript object. The object should have the type of node, its

attributes, and all the children.

Example

Input:

```
<div id=
```

```
"foo"
```

```
>
```

```
<h1>
```

```
Hello
```

```
</h1>
```

```
<p class=
```

```
"bar"
```

```
>
```

```
<span>
```

```
World!
```

```
</span>
```

```
</p>
```

```
</div>
```

```
Output:
```

```
{
```

```
"props"
```

```
: {
```

```
"id"
```

```
:
```

"foo"

},

"children"

:[

{

"children"

:

"Hello"

,

"type"

:

"h1"

},

{

"props"

: {

"class"

:

"bar"

},



"children"

: [

{

"children"

:

"World!"

,

"type"

:

"span"

}

],

"type"

:

"p"

}

],

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

410

"type"

:

"div"

}

There are three things that we need to generally take care of to create this function.

- Get the name of the HTML element of the node.
- Get all the properties/attributes of the node.
- If the node does not have any children then get its content, otherwise iterates each child, and calls the same function recursively with them to generate the object for each.

We will be using a helper function that will give us all the attributes of the node.

// helper function to get all the attributes of node

const

getAllAttributes = (

node

) => {

let

obj = {};

for

(

let

```
att, i =  
0  
  
, atts = node.attributes, n =  
atts.length; i < n;  
i++){  
    att = atts[i];  
    obj[att.nodeName] = att.nodeValue;  
}  
  
return  
obj;  
};  
  
const  
HTMLtoJSON = (  
node  
) => {  
    // to store the output  
    const  
    output = {};  
    // get the node name  
    const
```

```
type = node.localName;

// set the children to innerText by default

let

children = node.innerText;

// if the node has children

© JavaScript Interview Guide | learnersbucket.com

411

if

(node.children.length >

0

){

// recursively compute all the children

// and return an array of them

children = [];

for

(

let

child

of

node.children){
```

```
children.push(HTMLtoJSON(child));
```

```
};
```

```
};
```

```
// get all the properties of the node
```

```
const
```

```
props = getAllAttributes(node);
```

```
// if properties exist store them
```

```
if
```

```
(
```

```
Object
```

```
.keys(props).length){
```

```
output[
```

```
'props'
```

```
] = props;
```

```
}
```

```
// store the type and children
```

```
output[
```

```
'children'
```

```
] = children;
```

```
output[
```

```
'type'
```

```
] = type;
```

```
return
```

```
output;
```

```
};
```

Test Case

Input:

```
<div id=
```

```
"foo"
```

```
>
```

```
<h1>
```

Hello

```
</h1>
```

```
<p class=
```

```
"bar"
```

```
>
```

```
<span>
```

World!

```
</span>
```

```
</p>
```

</div>

const

node =

document

.getElementById(

"foo"

);

console

.log(HTMLtoJSON(node));

Output:

{

"props"

: {

"id"

:

"foo"

},

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

412

"children"

```
: [  
  {  
    "children"  
    :  
    "Hello"  
    ,  
    "type"  
    :  
    "h1"  
    },  
    {  
      "props"  
      : {  
        "class"  
        :  
        "bar"  
        },  
      "children"  
      : [  
        {
```



"children"

:

"World!"

,

"type"

:

"span"

}

],

"type"

:

"p"

}

],

"type"

:

"div"

}

## Concurrent history tracking system

### Problem Statement -

Design a concurrent history tracking system where an entity and its different services can be registered and changes being made to them can be tracked.

### Example

```
const
```

```
historyTracking = HistoryTracking();
```

```
historyTracking.registerEntity(
```

```
"document"
```

```
);
```

```
historyTracking.registerService(
```

```
"document"
```

```
,
```

```
'JavaScript
```

```
Ultimate Guide'
```

```
);
```

```
historyTracking.track(
```

```
"document"
```

```
,
```

'JavaScript Ultimate  
Guide'

,

"Problem

1"

);

historyTracking.track(

"document"

,

'JavaScript Ultimate  
Guide'

,

"Problem 1,

Problem 2"

);

historyTracking.track(

"document"

,

'JavaScript Ultimate  
Guide'

```
,  
"Problem  
3"  
);  
console  
.log(historyTracking.getHistory(  
"document"  
,  
'JavaScript Ultimate  
Guide'  
));  
// ["Problem 1","Problem 1, Problem 2","Problem 3"]
```

This tracking system works similarly to Google drive tracking, where we can have an entity like Google Docs and multiple services (docs)

inside that and each service will have its own historical record.

To implement this system.

We will use a [map](#) to store the unique entities. Each entity can have multiple services and its history stored as an object and this object will be stored as a value next to the entity in the map.

Whenever a new history entry is requested, we will compare it with the last history, if data is changed, push it into the history.

414

We will be using the [singleton design pattern](#) to make sure that one instance is used for tracking.

class

HistoryTrackingHelper

{

constructor

() {

// to track unique entries

this

.entities =

new

Map

();

}

// register a new entity

registerEntity(entity) {

this

.entities.set(entity, {});

};

// register a new service to the entity

```
registerService(entity, service) {  
  
  const  
  
  existingServices =  
  
  this  
  
  .entities.get(entity);  
  
  // if the entity is not present  
  
  // create a new entity with the service  
  
  if  
  
  (!existingServices){  
  
    this  
  
    .entities.set(entity, {[service]: []});  
  
    }  
  
    // add the service to the existing entity  
  
    else  
  
    {  
  
      const  
  
      merged = {...existingServices, [service]:  
  
      []};  
  
      this  
  
      .entities.set(entity, merged);  
    }  
  }  
}
```

```
}
```

```
}
```

```
// track the history of the entity and its service
```

```
track(entity, service, newData) {
```

```
// get the last entry of the service
```

```
const
```

```
services =
```

```
this
```

```
.entities.get(entity);
```

```
const
```

```
history = services[service];
```

```
const
```

```
last = history[history.length -
```

```
1
```

```
];
```

```
// there is no previous entry
```

```
// add the current as latest
```

```
if
```

```
(!last){
```

```
const
```

```
serviceWithNewHistory = {...services,  
[service]: [newData]};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

415

this

```
.entities.set(entity, serviceWithNewHistory);  
}
```

```
// else compare the new one with the last one
```

```
// if both are different then make the new entry
```

```
else
```

```
{
```

```
const
```

```
lastStr =
```

```
JSON
```

```
.stringify(last);
```

```
const
```

```
newDataStr =
```

```
JSON
```

```
.stringify(newData);
```

```
if
```



```
(lastStr !== newDataStr){  
  
  const  
  
  serviceWithNewHistory = {...services,  
  
    [service]: [...history,  
  
      newData]};  
  
  this  
  
    .entities.set(entity, serviceWithNewHistory);  
  
  }  
  
  }  
  
  }  
  
  // get the complete history  
  getHistory(entity, service) {  
  
    const  
  
    services =  
  
    this  
  
      .entities.get(entity);  
  
    return  
  
    services[service];  
  
  }  
  
}
```

```
// create a single instance of the tracking
```

```
// using single-ton design pattern
```

```
const
```

```
HistoryTracking = (
```

```
function
```

```
() {
```

```
let
```

```
instance;
```

```
return
```

```
function
```

```
() {
```

```
if
```

```
(!instance) {
```

```
instance =
```

```
new
```

```
HistoryTrackingHelper();
```

```
}
```

```
return
```

```
instance;
```

```
};
```

```
})();
```

Test Case

Input:

```
const
```

```
historyTracking = HistoryTracking();
```

```
historyTracking.registerEntity(
```

```
"document"
```

```
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

416

```
historyTracking.registerService(
```

```
"document"
```

```
,
```

```
'JavaScript Ultimate Guide'
```

```
);
```

```
historyTracking.track(
```

```
"document"
```

```
,
```

```
'JavaScript Ultimate
```

```
Guide'
```

,

"Problem

1"

);

historyTracking.track(

"document"

,

'JavaScript Ultimate

Guide'

,

"Problem 1,

Problem 2"

);

historyTracking.track(

"document"

,

'JavaScript Ultimate

Guide'

,

"Problem

```
3"  
);  
console  
.log(historyTracking.getHistory(  
"document"  
,  
'JavaScript Ultimate  
Guide'  
));
```

Output:

```
[  
"Problem 1"  
,  
"Problem 1, Problem 2"  
,  
"Problem 3"  
]
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

417

Implement an in-memory search engine

## Problem Statement -

Implement an in-memory search engine where multiple documents could be stored under a particular namespace and search on them and sort the search results by passing the orderBy parameter.

## Example

```
const
searchEngine =
new
InMemorySearch();
searchEngine.addDocuments(
'Movies'
,
{name:
'Avenger'
, rating:
8.5
,
year:
2017
},
```

{name:

'Black Adam'

, rating:

8.7

,

year:

2022

},

{name:

'Jhon Wick 4'

, rating:

8.2

, year:

2023

},

{name:

'Black Panther'

, rating:

9.0

, year:

```
2022
}
);
console
.log(searchEngine.search(
'Movies'
, (e) => e.rating
>
8.5
, {key:
'rating'
, asc:
false
}));
/*
[
{
"name": "Black Panther",
"rating": 9,
"year": 2022
```



```
},  
  
{  
  "name": "Black Adam",  
  "rating": 8.7,  
  "year": 2022  
}  
]  
*/
```

To implement this, we can use a [map](#) to register and track different namespaces and their associated documents.

For searching, we will take the namespace and a callback function that will filter the value based on the output of this callback function.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

418

This function will also accept an option `orderBy` parameter, that will have the key and the order on which the search result will be ordered.

class

InMemorySearch

```
{
```

```
  constructor
```

```
() {
```

```
// to track namespace and its document

this

.entities =

new

Map

();

}

// register the new namespace

registerNameSpace(name){

this

.entities.set(name, []);

}

// add document to the namespace

addDocuments(nameSpace, ...documents){

const

existing =

this

.entities.get(nameSpace);

// if the namespace does not exists

// create one and add the documents
```

```
if
(!existing){
this
.entities.set(nameSpace, [...documents]);
}
// else add the merge the documents
else
{
this
.entities.set(nameSpace, [...existing, ...documents]);
}
}

// search the documents of the given namespace
search(nameSpace, filterFN, orderByFN){
// get the namespace
const
docs =
this
.entities.get(nameSpace);
// get it filtered
```

```
const
filtered = docs.filter((
e
) => filterFN(e));
// if orderby is requestd
if
(orderByFN){
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

419

```
const
{key, asc} = orderByFN;
// orderby the searched result
// based on the key and order requested
return
filtered.sort((
a, b
) => {
if
(asc){
return
```

```
a[key] - b[key];
```

```
}
```

```
else
```

```
{
```

```
return
```

```
b[key] - a[key];
```

```
}
```

```
});
```

```
}
```

```
return
```

```
filtered;
```

```
}
```

```
};
```

Test Case

Input:

```
const
```

```
searchEngine =
```

```
new
```

```
InMemorySearch();
```

```
searchEngine.addDocuments(
```

'Movies'

,

{name:

'Avenger'

, rating:

8.5

,

year:

2017

},

{name:

'Black Adam'

, rating:

8.7

,

year:

2022

},

{name:

'Jhon Wick 4'

```
, rating:
```

```
8.2
```

```
, year:
```

```
2023
```

```
},
```

```
{name:
```

```
'Black Panther'
```

```
, rating:
```

```
9.0
```

```
, year:
```

```
2022
```

```
}
```

```
);
```

```
console
```

```
.log(searchEngine.search(
```

```
'Movies'
```

```
, (e) => e.rating
```

```
>
```

```
8.5
```

```
, {key:
```

```
'rating'
```

```
, asc:
```

```
false
```

```
));
```

Output:

```
/*
```

```
[
```

```
{
```

```
"name": "Black Panther",
```

```
"rating": 9,
```

```
"year": 2022
```

```
},
```

```
{
```

```
"name": "Black Adam",
```

```
"rating": 8.7,
```

```
"year": 2022
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
420
```

```
}
```

```
]
```



\*/

© JavaScript Interview Guide | learnersbucket.com

421

Implement a fuzzy search function

Problem Statement -

Implement a function in JavaScript that performs fuzzy string matching, it accepts an array of strings and a query as input and returns the list of strings that matches.

Example

```
const
```

```
strArr = [
```

```
'Doomsayer'
```

```
,
```

```
'Doomguard'
```

```
,
```

```
'Doomhamer'
```

```
,
```

```
'Bane of Doom'
```

```
,
```

```
'Fearsome Doomguard'
```

```
,  
'Dr. Boom'  
  
,  
'Majordomo'  
  
,  
'Shadowbomber'  
  
,  
'Shadowform'  
  
,  
'Goldshire footman'  
  
];  
  
const  
  
query =  
  
'an'  
  
;  
  
fuzzySearch(strArr, query);  
  
// ["Bane of Doom", "Goldshire footman"]
```

A string is considered matching if all characters from the search string are found in the string in the same order.

The most complex part of this function is creating the fuzzy search algorithm.

From the problem statement, we understand that the order of characters matters most in the search. Thus we can use the [sliding window technique](#) to search in each string.

© JavaScript Interview Guide | learnersbucket.com

422

- Use two variables, one to track the characters of the query and another to track the last searched position in the string.

- Pick the first character from the string, and get its first index in the string from the last searched position, if the character is

present, update the last searched position to the current position

and keep on check-in further.

- If any of the characters are not present return false, else return true.

```
const
```

```
fuzzySearch =
```

```
function
```

```
(
```

```
str, query
```

```
) {
```

```
// convert the query and str
```

```
// for case-insensitive search
```

```
str = str.toLowerCase();
```

```
query = query.toLowerCase();

// use two variables to track the

// current character

// and last searched position in the string

let

i =

0

, lastSearched =

-1

, current = query[i];

while

(current){

// if the character is not present

// return false

if

(!~(lastSearched = str.indexOf(current, lastSearched

+

1

))){

return
```

```
false  
  
;  
  
};  
  
current = query[++i];  
  
};  
  
// if the search completes  
  
// return true  
  
return  
  
true  
  
;  
  
};
```

Filter all the strings of the array through this function and return the list with the passed ones.

```
const  
  
search =  
  
function  
  
(  
  
arr, query  
  
) {  
  
return  
  
arr.filter((
```

e

```
) => fuzzySearch(e, query));
```

```
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

423

Test Case

Input:

```
const
```

```
arr = [
```

```
'Doomsayer'
```

```
,
```

```
'Doomguard'
```

```
,
```

```
'Doomhamer'
```

```
,
```

```
'Bane of Doom'
```

```
,
```

```
'Fearsome Doomguard'
```

```
,
```

```
'Dr. Boom'
```

,

'Majordomo'

,

'Shadowbomber'

,

'Shadowform'

,

'Goldshire footman'

];

console

.log(search(arr,

'sh'

));

Output:

[

"Shadowbomber"

,

"Shadowform"

,

"Goldshire footman"

]

© JavaScript Interview Guide | learnersbucket.com

424

Cancel an API request

Problem Statement -

Explain a way to cancel the ongoing API request.

Explanation

JavaScript web API's have a method called [AbortController](#) . This AbortController has a property called [signal](#) that allows us to create an AbortSignal that can be associated with the Fetch API which provides an option to abort the API request.

The signal is passed as a parameter to the fetch API.

To make this work we will create a constructor of the

*AbortController()* and get its signal property. Pass this signal property as a parameter to the fetch API and invoke the abort method of the

constructor whenever we want to cancel the API request.

When *abort()* is called, the *fetch()* promise rejects with an *AbortError* In the below example, we have created two buttons, *download* and *abort* . On the *download* click, an API request will be made and on the *abort* click the request will be aborted.

HTML:

<div>

<button class=



```
"download"
```

```
>
```

```
Download
```

```
</button>
```

```
<button class=
```

```
"abort"
```

```
>
```

```
Abort
```

```
</button>
```

```
</div>
```

```
JavaScript:
```

```
// create abort controller
```

```
const
```

```
controller =
```

```
new
```

```
AbortController();
```

```
const
```

```
signal = controller.signal;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
// get the buttons

const
downloadBtn =
document
.querySelector(
".download"
);

const
abortBtn =
document
.querySelector(
".abort"
);

// download event
downloadBtn.addEventListener(
"click"
, makeCall);

// abort event
abortBtn.addEventListener(
"click"
```

```
, () => {  
  controller.abort();  
  
  console  
  
  .log(  
  
    "Download aborted"  
  
  );  
  
});  
  
// helper method to make api call  
  
function  
  
makeCall  
  
() {  
  
  fetch(  
  
    'https://jsonplaceholder.typicode.com/photos'  
  
    ,  
  
    { signal })  
  
  .then((  
  
    response  
  
  ) => {  
  
    console  
  
    .log(  

```

```
"complete"  
  
    , response);  
  
    })  
  
    .catch((  
  
    err  
  
    ) => {  
  
    console  
  
    .error(  
  
    `error: ${err.message}`  
  
    );  
  
    });  
  
    };
```

Throttle on slow 2G in the developer tools and hit on the download button, later click on abort, the API call will terminate with the abort error.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

426

Highlight the words in the string

Problem Statement -

Given a string and array of keywords, highlight the words in the string that are part of the array of keywords.

Example

```
const
str =
"Ultimate JavaScript / FrontEnd Guide"
;

const
words = [
'Front'
,
'End'
,
'JavaScript'
];

highlight(str, words);

// "Ultimate <strong>JavaScript</strong> / <strong>FrontEnd</strong>
Guide"
```

If two words are overlapping or adjacent, combine them together. As

you can see in the above example there are two different words Front and End but they are highlighted together.

To implement this function we have to handle two different cases.

1. Check if any of the words in the string is present in the keywords array then highlight it.

2. Else, break each word into two sub-words and check for each

sub-word if they are present or not, if any part is present highlight them separately if both the parts are present highlight them together.

```
function
```

```
highlight
```

```
(
```

```
str, keywords
```

```
) {
```

```
// unique set of keywords
```

```
const
```

```
uniqueKeywords =
```

```
new
```

```
Set
```

```
(keywords);
```

```
// split the str into words
```

```
let
```

```
words = str.split(
```

```
" "
```

```
);
```

```
// traverse each word
```

```
const
```

```
result = words.map(  
  word  
  => {  
    // to track the embedding  
    © JavaScript Interview Guide | learnersbucket.com  
    427  
    let  
    output =  
    "  
    ;  
    // if the word is found in the keywords set  
    // highLight it  
    if  
    (uniqueKeywords.has(word)) {  
    output =  
    `<strong>${word}</strong>`  
    ;  
    }  
    // else check if the substring of the word is  
    in the keywords set
```

```
else
{
for
(
let
i =
0
; i < word.length; i++) {
// break the word into two parts
const
prefix = word.slice(
0
, i +
1
);
const
suffix = word.slice(i +
1
);
// if both the parts are present in keywords
```



```
// embed them together

if
(uniqueKeywords.has(prefix) && uniqueKeywords.has(suffix))
{
output =
`<strong>${prefix}${suffix}</strong>`
;
break
;
}

// else if the only prefix is present
// highlight it
else
if
(uniqueKeywords.has(prefix) && !uniqueKeywords.has(suffix))
{
output =
`<strong>${prefix}</strong>${suffix}`
;
}
```

```
// else if the only suffix is present

// highlight it

else

if

(!uniqueKeywords.has(prefix) && uniqueKeywords.has(suffix))

{

output =

`${prefix}<strong>${suffix}</strong>`

;

}

}

}

// if no embedding has happened

// return the original word

return

output !==

"

? output : word;

});

// from the string back
```

```
return
```

```
result.join(
```

```
" "
```

```
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

428

```
}
```

Test Case

Input:

```
const
```

```
str =
```

```
"Ultimate JavaScript / FrontEnd Guide"
```

```
;
```

```
const
```

```
words = [
```

```
'Front'
```

```
,
```

```
'End'
```

```
,
```

```
'JavaScript'
```

```
];
```

```
console
```

```
.log(highlight(str, words));
```

Output:

```
// "Ultimate <strong>JavaScript</strong> / <strong>FrontEnd</strong>  
Guide"
```

© JavaScript Interview Guide | learnersbucket.com

429

Reactjs Questions

© JavaScript Interview Guide | learnersbucket.com

430

usePrevious() hook

Create a hook in React that will remember the previous value.

usePrevious() hook will take the current value as input and hold it and will return it whenever it will get a new value. For the initial render, it will return undefined as there will not be any previous value for it.

To create the usePrevious() hook we will need to use the useRef() and useEffect() hook together.

useRef()

Between renderings, you can maintain values using the useRef() Hook which means the value won't change or be lost when the React

components re-render. This will help us to persist the previous value.

useEffect()

With the useEffect() hook, we can manage the side effects in the components during the lifecycle events.

Thus we can create a new reference using useRef() and update its value inside the useEffect() whenever a new value is provided, at the end return the reference value.

function

usePrevious

(

value

) {

// create a new reference

const

ref = useRef();

// store current value in ref

useEffect(

()

=> {

ref.current = value;

}, [value]);

// only re-run if value changes

```
// return previous value (happens before update  
in useEffect above)
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

431

```
return
```

```
ref.current;
```

```
}
```

current is the default object available on each reference that can be used to store any value.

Because the ref.current is returned before the useEffect() update, it returns the previous value, by default there is no value for ref.current thus it returns undefined .

Test case

```
import
```

```
{ useState, useEffect, useRef }
```

```
from
```

```
"react"
```

```
;
```

```
const
```

```
usePrevious = (
```

```
value
```

```
) => {  
  
  // create a new reference  
  
  const  
  
  ref = useRef();  
  
  // store current value in ref  
  
  useEffect(  
  
    ()  
  
    => {  
  
      ref.current = value;  
  
    }, [value]);  
  
  // only re-run if value changes  
  
  // return previous value (happens before update  
  in useEffect above)  
  
  return  
  
  ref.current;  
  
};  
  
const  
  
Example =  
  
()  
  
=> {
```

```
const  
[count, setCount] = useState(  
0  
);  
// get the previous value passed into the hook on  
the last render
```

```
const  
prevCount = usePrevious(count);  
// show both current and previous value  
return  
(  
<div>  
<h1>  
Now: {count}, before: {prevCount}  
</h1>  
<button onClick=  
{()  
=>  
setCount(count - 1)}>Decrement  
</button>
```



432

</div>

);

};

export

default

Example;

433

useIdle() hook

Create an useIdle() hook in React that will return the boolean value depending upon the active or inactive state of the user after a defined amount of time.

An user is considered to be inactive or idle if he is not performing any sort of action using interaction hardware like a mouse, or keyboard for desktops and laptops and touch on mobile and tablets.

For this, there are a set of events that we can listen to like mousemove , mousedown , keypress , DOMMouseScroll , mousewheel , touchmove , MSPointerMove .

Also, we need to handle edge cases where the window or tab is out of focus, for which we will listen to the focus and blur events.

If any of these events are triggered then set the user to be Active , else if none of them have happened for a given amount of time then set the

user to be Idle or Inactive .

We will take duration as input for useIdle(delay) for which if the user is not performing any action then he will be considered as Idle.

The logic to implement is straightforward, we will use a useState() to monitor the user's active status and useEffect() to assign the event listeners on the window object as well as document and later remove

the listeners during cleanup.

Using useRef() we will track a setTimeout that will change status if the user has not performed any action for the duration received as input, else clear the timer and start a fresh timeout.

© JavaScript Interview Guide | learnersbucket.com

434

```
import
```

```
{ useState, useEffect, useRef }
```

```
from
```

```
"react"
```

```
;
```

```
const
```

```
useIdle = (
```

```
  delay
```

```
) => {
```

```
  const
```

```
[isIdle, setIsIdle] = useState(
  false
);
// create a new reference to track timer
const
timeoutId = useRef();
// assign and remove the listeners
useEffect(
  ()
  => {
    setup();
    return
    ()
    => {
      cleanUp();
    };
  });
const
startTimer =
  ()
```

```
=> {  
  
  // wait till delay time before calling goInactive  
  
  timeoutId.current = setTimeout(goInactive, delay);  
  
};  
  
const  
  
resetTimer =  
  
()  
  
=> {  
  
  //reset the timer and make user active  
  
  clearTimeout(timeoutId.current);  
  
  goActive();  
  
};  
  
const  
  
goInactive =  
  
()  
  
=> {  
  
  setIsIdle(  
  
    true  
  
  );  
  
};
```

```
const
goActive =
()
=> {
  setIsIdle(
    false
  );
  // start the timer to track Inactiveness
  startTimer();
};

const
setup =
()
=> {
  document
    .addEventListener(
      "mousemove"
      , resetTimer,
      false
    );
}
```

```
document  
.  
addEventListener(  
"mousedown"  
, resetTimer,  
false  
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

435

```
document  
.  
addEventListener(  
"keypress"  
, resetTimer,  
false  
);
```

```
document  
.  
addEventListener(  
"DOMMouseScroll"  
, resetTimer,  
false  
);
```

```
document
```

```
.addEventListener(
```

```
"mousewheel"
```

```
, resetTimer,
```

```
false
```

```
);
```

```
document
```

```
.addEventListener(
```

```
"touchmove"
```

```
, resetTimer,
```

```
false
```

```
);
```

```
document
```

```
.addEventListener(
```

```
"MSPointerMove"
```

```
, resetTimer,
```

```
false
```

```
);
```

```
//edge case
```

```
//if tab is changed or is out of focus
```

```
window
  .addEventListener(
    "blur"
    , startTimer,
    false
  );
window
  .addEventListener(
    "focus"
    , resetTimer,
    false
  );
};

const
cleanUp =
()
=> {
  document
    .removeEventListener(
      "mousemove"
```



```
, resetTimer);  
  
document  
.  
removeEventListener(  
  
"mousedown"  
  
, resetTimer);  
  
document  
.  
removeEventListener(  
  
"keypress"  
  
, resetTimer);  
  
document  
.  
removeEventListener(  
  
"DOMMouseScroll"  
  
,  
  
resetTimer);  
  
document  
.  
removeEventListener(  
  
"mousewheel"  
  
, resetTimer);  
  
document  
.  
removeEventListener(  
  

```

```
"touchmove"
, resetTimer);

document
.removeEventListener(
"MSPointerMove"
,
resetTimer);

//edge case
//if tab is changed or is out of focus

window
.removeEventListener(
"blur"
, startTimer);

window
.removeEventListener(
"focus"
, resetTimer);

// memory leak

clearTimeout(timeoutId.current);

};
```

```
// return previous value (happens before update
```

```
in useEffect above)
```

```
return
```

```
isIdle;
```

```
};
```

Test Case

Input:

```
const
```

Example =

```
()
```

```
=> {
```

```
const
```

```
isIdle = useIdle(
```

```
2000
```

```
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

436

```
return
```

```
(
```

```
<div>
```

```
<h1>
```

```
IsIdle: {isIdle ? "true" : "false"}
```

```
</h1>
```

```
</div>
```

```
);
```

```
};
```

Output:

IsIdle:

false

IsIdle:

true

// after 2 seconds

© JavaScript Interview Guide | learnersbucket.com

437

useAsync() hook

useAsync(asyncFn, immediate) takes an async function and an

immediate flag as input and it will provide an abstraction for complete async operation (API calls) in React, in return it will give the status , value , error , refetch .

- *state* : It will have one of the four values ["idle", "pending", "success", "error"] depending upon the current state of the asyncFn.

- *value* : If the state is successful then this will have the value returned from the `asyncFn` .
- *error* : If the state is error then this will have the error returned from the `asyncFn` .
- *refetch()* : This function can be used to invoke the function again and refetch data.

Based on the input and output, let's implement the `useAsync()` hook.

We will be using `useState()` to monitor the status , value , & error .

and [useCallback\(\)](#) hook to create a memoized `refetch()` function.

A memoized version of the callback that only changes if one of the dependencies has changed is what `useCallback()` returns. To avoid needless renderings, this is helpful when delivering callbacks to optimized child components that rely on reference equality.

At the end, we will pass the immediate flag that we took as input to the `useEffect()` hook that will trigger the refetch if the value of the immediate flag changes and is true .

```
const
```

```
useAsync = (
```

```
  asyncFn, immediate = false
```

```
) => {
```

```
  // four status to choose ["idle", "pending", "success",
```

```
  "error"]
```

```
  const
```

```
[state, setState] = useState({
```

```
  status:
```

```
    "idle"
```

```
,
```

```
  value:
```

```
    null
```

```
,
```

```
  © JavaScript Interview Guide | learnersbucket.com
```

```
  438
```

```
  error:
```

```
    null
```

```
,
```

```
});
```

```
// return the memoized function
```

```
// useCallback ensures the below useEffect is not
```

```
called
```

```
// on every render, but only if asyncFunction changes.
```

```
const
```

```
  refetch = useCallback(
```

```
    ()
```

```
=> {  
  // reset the state before call  
  setState({  
    status:  
    "pending"  
  ,  
    value:  
    null  
  ,  
    error:  
    null  
  ,  
  });  
  return  
  asyncFn()  
  .then((  
    response  
  ) => {  
    setState({  
      status:
```

"success"

,

value: response,

error:

null

,

});

})

.catch((

error

) => {

setState({

status:

"error"

,

value:

null

,

error: error,

});



```
});  
  
}, [asyncFn]);  
  
// execute the function  
  
// if asked for immediate  
  
useEffect(  
  
()  
  
=> {  
  
  if  
  
    (immediate) {  
  
      refetch();  
  
    }  
  
  }, [refetch, immediate]);  
  
// state values  
  
const  
  
  { status, value, error } = state;  
  
© JavaScript Interview Guide | learnersbucket.com  
  
439  
  
return  
  
  { refetch, status, value, error };  
  
};
```

Test Case

Input:

```
//dummy api call
```

```
const
```

```
fakeApiCall =
```

```
()
```

```
=> {
```

```
return
```

```
new
```

```
Promise
```

```
((
```

```
resolve, reject
```

```
) => {
```

```
setTimeout(
```

```
()
```

```
=> {
```

```
const
```

```
rnd =
```

```
Math
```

```
.random() *
```

10

;

rnd <=

5

? resolve(

"Success"

) : reject(

"Error"

);

},

1000

);

});

};

const

Example =

()

=> {

const

{ status, value, error } = useAsync(fakeApiCall,

true

);

return

(

<div>

<div>

Status: {status}

</div>

<div>

Value: {value}

</div>

<div>

error: {error}

</div>

</div>

);

};

Output:

Status: success

Value: Success

error:

© JavaScript Interview Guide | learnersbucket.com

440

useDebounce() hook

Debouncing is a method or a way to execute a function when it is made sure that no further repeated event will be triggered in a given frame of time.

We have already seen how to implement [normal debounce](#) and [debounce with an immediate flag](#).

Create a useDebounce() hook in React with the immediate flag as it will behave normally as well depending upon the flag.

We will be using useRef() to track the timerId of setTimeout so that we can reset it if a subsequent full call is made within the defined time.

Also, we will wrap the logic inside the useCallback() to avoid needless re-renderings as the callback function returns a memoized function

that only changes when one of the dependencies changes.

```
const
```

```
useDebounce = (
```

```
  fn, delay, immediate = false
```

```
)
```

```
=> {
```

```
  // ref the timer
```

```
  const
```

```
timerId = useRef();

// create a memoized debounce

const

debounce = useCallback(

function

() {

// reference the context and args for the setTimeout

function

let

context =

this

,

args =

arguments

;

// should the function be called now? If immediate

is true

// and not already in a timeout then the answer

is: Yes

const
```

```
callNow = immediate && !timerId.current;
```

```
// base case
```

```
// clear the timeout to assign the new timeout  
to it.
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

441

```
// when event is fired repeatedly then this helps to reset
```

```
clearTimeout(timerId.current);
```

```
// set the new timeout
```

```
timerId.current = setTimeout(  
function
```

```
() {
```

```
// Inside the timeout function, clear the
```

```
timeout variable
```

```
// which will let the next execution run when
```

```
in 'immediate' mode
```

```
timerId.current =
```

```
null
```

```
;
```

```
// check if the function already ran with
```

```
the immediate flag

if
(!immediate) {
// call the original function with apply
fn.apply(context, args);
}
}, delay);

// immediate mode and no wait timer? Execute
the function immediately

if
(callNow) fn.apply(context, args);
},
[fn, delay, immediate]
);

return
debounce;
};
```

Test Case 1: Without an immediate flag

Input:

```
const
```



Example =

()

=> {

const

print =

()

=> {

console

.log(

"hello"

);

};

const

debounced = useDebounce(print,

500

);

useEffect(

()

=> {

window

```
.addEventListener(  
  "mousemove"  
  , debounced,  
  false  
);  
  
return  
  
()  
=> {  
  window  
  .removeEventListener(  
    "mousemove"  
    , debounced,  
    false  
  );  
};  
});
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

442

return

<></>

```
;
```

```
};
```

Output:

"hello" //after 500 millisecond delay when user stops moving mouse Test  
Case 2: With an immediate flag

Input:

```
const
```

```
Example =
```

```
()
```

```
=> {
```

```
const
```

```
print =
```

```
()
```

```
=> {
```

```
console
```

```
.log(
```

```
"hello"
```

```
);
```

```
};
```

```
// immediate
```

```
const
```

```
debounced = useDebounce(print,
```

```
500
```

```
,
```

```
true
```

```
);
```

```
useEffect(
```

```
()
```

```
=> {
```

```
window
```

```
.addEventListener(
```

```
"mousemove"
```

```
, debounced,
```

```
false
```

```
);
```

```
return
```

```
()
```

```
=> {
```

```
window
```

```
.removeEventListener(
```

```
"mousemove"
```

```
,debounced,
```

```
false
```

```
);
```

```
};
```

```
});
```

```
return
```

```
<></>
```

```
;
```

```
};
```

Output:

"hello" //immediately only once till the mouse moving is not stopped

"hello" //immediately again once till the mouse moving is not stopped

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

443

useThrottle() hook

Throttling is a way/technique to restrict the number of function

execution/call. For example, consider a lucky draw number generator, we want to get a number only after a particular time.

Excessive function invocations in javascript applications hamper the performance drastically. To optimize an app we need to handle this

correctly.

There are scenarios where we may invoke functions when it isn't necessary. For example, consider a scenario where we want to make an API call to the server on a button click.

If the user spam the click then this will make an API call on each click.

This is not what we want, we want to restrict the number of API calls that can be made. The other call will be made only after a specified interval of time.

We have already seen how to [implement the throttle function](#) .

Create a useThrottle() hook in React with the leading and trailing flag When leading is enabled the first function will invoke right away and then after the specified delay, while when trailing is enabled the first function will invoke after the delay and so on.

We will be using useRef() to track the timerId of setTimeout so that we can reset it as and when required and previous arguments.

Also, we will wrap the logic inside the useCallback() to avoid needless re-renderings as the callback function returns a memoized function

that only changes when one of the dependencies changes.

© JavaScript Interview Guide | learnersbucket.com

444

const

useThrottle = (

fn, wait, option = { leading: true, trailing: true }

)

=> {

```
const
timerId = useRef();
// track the timer
const
lastArgs = useRef();
// track the args
// create a memoized debounce
const
throttle = useCallback(
function
(
...args
) {
const
{ trailing, leading } = option;
// function for delayed call
const
waitFunc =
()
=> {
```

```
// if trailing invoke the function and start
the timer again
if
(trailing && lastArgs.current) {
fn.apply(
this
, lastArgs.current);
lastArgs.current =
null
;
timerId.current = setTimeout(waitFunc, wait);
}
else
{
// else reset the timer
timerId.current =
null
;
}
};
```



```
// if leading run it right away
if
(!timerId.current && leading) {
  fn.apply(
    this
    , args);
}
// else store the args
else
{
  lastArgs.current = args;
}
// run the delayed call
if
(!timerId.current) {
  timerId.current = setTimeout(waitFunc, wait);
}
},
[fn, wait, option]
);
```

```
return
```

```
throttle;
```

```
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

445

Test Case 1: With leading flag

Input:

```
const
```

Example =

```
()
```

```
=> {
```

```
const
```

```
print =
```

```
()
```

```
=> {
```

```
console
```

```
.log(
```

```
"hello"
```

```
);
```

```
};
```

```
const
throttled = useThrottle(print,
2500
, { leading:
true
, trailing:
false
});
return
<button onClick=
{throttled}
>
click me
</button>
;
};
```

Output:

```
"hello"
// immediately
"hello"
```

```
// after 2500 milliseconds of last call
```

```
"hello"
```

```
// after 2500 milliseconds of last call
```

Test Case 2: With trailing flag

Input:

```
const
```

```
Example =
```

```
()
```

```
=> {
```

```
const
```

```
print =
```

```
()
```

```
=> {
```

```
console
```

```
.log(
```

```
"hello"
```

```
);
```

```
};
```

```
const
```

```
throttled = useThrottle(print,
```

2500

, { leading:

false

, trailing:

true

});

return

<button onClick=

{throttled}

>

click me

</button>

;

};

Output:

"hello"

// after 2500 milliseconds

"hello"

// after 2500 milliseconds of last call

"hello"

// after 2500 milliseconds of last call

© JavaScript Interview Guide | learnersbucket.com

446

useResponsive() hook

Create useResponsive() hook in React that will return the device type (isMobile, isTablet, isDesktop) depending upon the window width.

Many times we require to conditionally render components depending upon the device, rather than hiding and showing through CSS we can use this hook.

For this, we will assign an event listener to the [window](#) object and listen to the resize event on the function onResizeHandler() that will update the state whenever the user resizes the screen.

Assigning the event listener and removing it are abstracted inside two functions Setup() and Cleanup() and it is called inside the useEffect() hook, we have also called onResizeHandler() to initialize the value.

const

useResponsive =

()

=> {

// screen resolutions

const

[state, setState] = useState({

isMobile:

false

,

isTablet:

false

,

isDesktop:

false

,

});

useEffect(

()

=> {

// update the state on the initial load

onResizeHandler();

// assign the event

Setup();

return

()

=> {

```
// remove the event
```

```
Cleanup();
```

```
};
```

```
}, []);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

447

```
// update the state on window resize
```

```
const
```

```
onResizeHandler =
```

```
()
```

```
=> {
```

```
const
```

```
isMobile =
```

```
window
```

```
.innerWidth <=
```

```
768
```

```
;
```

```
const
```

```
isTablet =
```

```
window
```



```
.innerWidth >=
768
&&
window
.innerWidth
<=
990
;
const
isDesktop =
window
.innerWidth >
990
;
setState({ isMobile, isTablet, isDesktop });
};
// debounce the resize call
const
debouncedCall = useDebounce(onResizeHandler,
500
```

```
);  
  
// add event listener  
  
const  
  
Setup =  
  
()  
  
=> {  
  
window  
  
.addEventListener(  
  
"resize"  
  
, debouncedCall,  
  
false  
  
);  
  
};  
  
// remove the listener  
  
const  
  
Cleanup =  
  
()  
  
=> {  
  
window  
  
.removeEventListener(  

```

```
"resize"  
  
  , debouncedCall,  
  
  false  
  
);  
  
};  
  
return  
  
state;  
  
};
```

The function onResizeHandler() is debounced using [useDebounce\(\)](#) hook as we won't to avoid the continuous state updates as the user keeps resizing, rather update once when the user is done resizing.

Test Case

Input:

const

Example =

()

=> {

const

{ isMobile, isTablet, isDesktop } = useResponsive();

console

```
.log(isMobile, isTablet, isDesktop);
```

```
return
```

```
<></>
```

```
;
```

```
};
```

© JavaScript Interview Guide | learnersbucket.com

448

Output:

false, false, true

© JavaScript Interview Guide | learnersbucket.com

449

useWhyDidYouUpdate() hook

Performance optimization is the key to making any web app resilient especially this time around when the front end is becoming more and more complex.

One way to achieve this in React is by avoiding unnecessary re-renders and to track this we need to monitor what has changed in the props or states within the component.

With the useWhyDidYouUpdate() hook we can determine what has changed that has triggered the re-rendering. Let us see how we can create this in React.

The idea is simple: we will extend the [usePrevious\(\)](#) hook and compare the previous values with new values and see if it has changed.

```
function
```

```
useWhyDidYouUpdate
```

```
(
```

```
name, props
```

```
) {
```

```
// create a reference to track the previous data
```

```
const
```

```
previousProps = useRef();
```

```
useEffect(
```

```
()
```

```
=> {
```

```
if
```

```
(previousProps.current) {
```

```
// merge the keys of previous and current data
```

```
const
```

```
keys =
```

```
Object
```

```
.keys({ ...previousProps.current,
```

```
...props });
```

```
// to store what has change

const

changesObj = {};

// check what values have changed between the

previous and current

keys.forEach((

key

) => {

// if both are object

if

(

typeof

props[key] ===

"object"

&&

typeof

previousProps.current[key] ===

"object"

) {

if
```

```
(  
  JSON  
    .stringify(previousProps.current[key])  
    !==  
    JSON  
      .stringify(props[key])) {  
  // add to changesObj  
  © JavaScript Interview Guide | learnersbucket.com  
  450  
  changesObj[key] = {  
    from: previousProps.current[key],  
    to: props[key],  
  };  
}  
}  
else  
{  
  // if both are non-object  
  if  
    (previousProps.current[key] !== props[key])
```

```
{  
  // add to changesObj  
  changesObj[key] = {  
    from: previousProps.current[key],  
    to: props[key],  
  };  
}  
}  
});  
  
// if changesObj not empty, print the cause  
if  
  
(  
  Object  
    .keys(changesObj).length) {  
  console  
    .log(  
      "This is causing re-renders"  
      ,  
      name, changesObj);  
}
```



```
}  
  
// update the previous props with the current  
previousProps.current = props;  
  
});  
  
}
```

Test Case

Input:

```
import  
  
React, { useEffect, useRef, useState }  
  
from  
  
"react"  
  
;  
  
const  
  
Counter = React.memo((  
  
props  
  
) => {  
  
useWhyDidYouUpdate(  
  
"Counter"  
  
, props);  
  
return
```

```
<div style=  
  {props.style}  
>  
  {props.count}  
</div>  
  
;  
});  
  
export  
default  
function  
App  
( ) {  
  const  
    [count, setCount] = useState(  
      0  
    );  
  const  
    [testCase, setTestCase] = useState(  
      null  
    );
```

451

const

counterStyle = {

fontSize:

"3rem"

,

color:

"red"

,

};

return

(

<div>

<div className=

"counter"

>

<Counter

count=

{count}

```
style=
{counterStyle}
testCaseWithArray=
{testCase}
function=
{()
=>
console.log(count)}
/>
<button
onClick=
{()
=>
{
setCount(count + 1);
setTestCase([count + 1]);
}}
>
Increment
</button>
```

```
</div>
```

```
</div>
```

```
);
```

```
}
```

Output:

This is causing re-renders Counter

```
{count: {...}, testCaseWithArray: {...}, function: {...}}
```

count:

from: 0

to: 1

function:

from: () => console.log(count) // 0

to: () => console.log(count) // 1

testCaseWithArray:

from: null

to: [1]

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

452

useOnScreen() hook

Tracking components visibility can be really handy in multiple cases, especially for performance, when you want to load to the media like, image,

video, audio, etc only when the component is in the view port or is visible.

Another case is when you want to track the user activity like when a user is starring a product (product is in the viewport) so that you can use this data for recommendations.

For this we can create a `useOnScreen()` hook that will return boolean value if the component is the view port or not.

There are two ways to implement it.

1. Using [Intersection Observer](#).
2. Using [getBoundingClientRect\(\)](#).

Using Intersection Observer

With `useRef()` we will create reference to the DOM element which we want to track, and then pass this to the `useOnScreen()` hook.

`useOnScreen()` hook will set up the observation for the ref when the component will be mounted. This will be done in the `useEffect()` and then create an instance of `IntersectionObserver` and if the entry is

interacting, update the state to true which means it is visible,  
otherwise false .

Disconnect the observation when the component is about to unmount  
inside the `useEffect()` .

function

`useOnScreen`

(

ref

```
) {
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

453

```
const
```

```
[isIntersecting, setIntersecting] = useState(
```

```
false
```

```
);
```

```
// monitor the interaction
```

```
const
```

```
observer =
```

```
new
```

```
IntersectionObserver(
```

```
(
```

```
[entry]
```

```
) => {
```

```
// update the state on interaction change
```

```
setIntersecting(entry.isIntersecting);
```

```
}
```

```
);
```

```
useEffect(
```

```
()
```

```
=> {
```

```
// assign the observer
```

```
observer.observe(ref.current);
```

```
// remove the observer as soon as the component
```

```
is unmounted
```

```
return
```

```
()
```

```
=> {
```

```
observer.disconnect();
```

```
};
```

```
}, []);
```

```
return
```

```
isIntersecting;
```

```
}
```

```
Test Case
```

```
const
```

```
Element = (
```

```
{ number }
```

```
) => {
```



```
const
ref = useRef();

const
isVisible = useOnScreen(ref);

return
(
<div ref=
{ref}
className=
"box"
>
{number}
{isVisible ? `I am on screen` : `I am invisible`}
</div>
);
};

const
DummyComponent =
()
=> {
```

```
const
arr = [];
for
(
let
i =
0
; i <
20
; i++) {
arr.push(
<Element key=
{i}
number=
{i}
/>
);
}
```

```
return arr;
```

```
};
```

```
export default DummyComponent;
```

Using `getBoundingClientRect()`

Unlike Intersection Observer, here we will have to perform a simple calculation to determine if the element is in the viewport or not.

If the top of the element is greater than zero but less than the

`window.innerHeight` then it is in the viewport. We can also add some offset in case we want a buffer.

Assign a scroll event on the window and inside the listener get the

`getBoundingClientRect()` of the element. Perform the calculation and

update the state accordingly.

```
function
```

```
useOnScreen2
```

```
(
```

```
ref
```

```
) {
```

```
const
```

```
[isIntersecting, setIntersecting] = useState(
```

```
false
```

```
);
```

```
// determine if the element is visible

const
observer =
function
() {
const
offset =
50
;
const
top = ref.current.getBoundingClientRect().top;
setIntersecting(top + offset >=
0
&& top - offset
<=
window
.innerHeight);
};
useEffect(
()
```

```
=> {  
  // first check  
  observer();  
  // assign the listener  
  window  
  .addEventListener(  
    "scroll"  
    , observer);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

455

```
// remove the listener  
return  
()  
=> {  
  window  
  .removeEventListener(  
    "scroll"  
    , observer);  
};  
}, []);
```

return

isIntersecting;

}

Test Case

const

Element = (

{ number }

) => {

const

ref = useRef();

const

isVisible = useOnScreen2(ref);

return

(

<div ref=

{ref}

className=

"box"

>

{number}

```
{isVisible ? `I am on screen` : `I am invisible`}
```

```
</div>
```

```
);
```

```
};
```

```
const
```

```
DummyComponent =
```

```
()
```

```
=> {
```

```
const
```

```
arr = [];
```

```
for
```

```
(
```

```
let
```

```
i =
```

```
0
```

```
; i <
```

```
20
```

```
; i++) {
```

```
arr.push(
```

```
<Element key=
```

```
{i}  
number=  
{i}  
/>  
);  
}  
return arr;  
};  
export default DummyComponent;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

456

useScript() hook

There are scripts that we don't require on the initial load of our app, rather than in certain components or places.

For example, Google AdSense script, we can load it after once the application is ready and the component that will display the ads is mounted.

In such scenarios, we can use the useScript() hook to asynchronously inject scripts.

The idea is simple, pass the script source to the useScript() hook and it will check if any script with this source is already injected or not, if it is present,



return 'ready' state. Else create a new script with the source and inject it at the end of the body.

Assign event listeners on this script tag, which will update the statuses.

on successful load 'ready' and on error 'error' .

```
function
```

```
useScript
```

```
(
```

```
src
```

```
) {
```

```
// keep track of script status ("idle", "loading",
```

```
"ready", "error")
```

```
const
```

```
[status, setStatus] = useState(src ?
```

```
"loading"
```

```
:
```

```
"idle"
```

```
);
```

```
useEffect(
```

```
()
```

```
=> {
```

```
// if no url provided, set the state to be idle
```

```
if
(!src) {
setStatus(
"idle"
);
return
;
}

// get the script to check if it is already sourced
or not

let
script =
document
.querySelector(
`script[src="${src}"]`
);

if
(script) {
// if the script is already loaded, get its
status and update.
```

457

```
setStatus(script.getAttribute(  
  "data-status"  
));  
  
}  
  
else  
  
{  
  
  // create script  
  
  script =  
  
  document  
  
  .createElement(  
  
    "script"  
  
  );  
  
  script.src = src;  
  
  script.async =  
  
  true  
  
  ;  
  
  script.setAttribute(  
  
    "data-status"
```

```
,  
"loading"  
);  
  
// inject the script at the end of the body  
document  
    .body.appendChild(script);  
  
// set the script status in a custom attribute  
const  
    setAttributeFromEvent = (  
        event  
    ) => {  
        script.setAttribute(  
            "data-status"  
        , event.type  
        );  
    };  
  
    "load"  
    ?  
    "ready"  
    :  
    "error"
```

```
);  
  
};  
  
// assign the event listeners to monitor if  
script is loaded properly  
script.addEventListener(  
  "load"  
  , setAttributeFromEvent);  
script.addEventListener(  
  "error"  
  , setAttributeFromEvent);  
}  
  
// helper function to update the script status  
const  
setStateFromEvent = (  
  event  
) => {  
  setStatus(event.type ===  
    "load"  
    ?  
    "ready"
```

```
:  
"error"  
);  
};  
  
// setup  
script.addEventListener(  
"load"  
, setStateFromEvent);  
script.addEventListener(  
"error"  
, setStateFromEvent);  
  
// clean up  
return  
()  
=> {  
if  
(script) {  
script.removeEventListener(  
"load"  
, setStateFromEvent);
```

```
script.removeEventListener(  
  "error"  
  , setStateFromEvent);  
}  
};  
}, [src]);  
return  
status;  
}
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

458

Test Case

```
const
```

```
Dummy =
```

```
()
```

```
=> {
```

```
const
```

```
status = useScript(  
  "https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.  
  js"
```

```
);  
  
console  
  
.log(status);  
  
return  
  
<></>  
  
;  
  
};  
  
export default Dummy;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

459

useOnClickOutside() hook

Check if the user has clicked outside the component or not. Create the useOnClickOutside() hook in React that will help us to detect it.

useOnClickOutside(ref, callback) will accept the component/element reference and the callback function and will invoke the callback function if clicked outside the reference.

All we have to do is listen to the mouse and touch event like

mousedown , touchstart and on the event fire check if the event.target is not the descendant of the reference then invoke the callback.

Wrap this logic inside the useEffect() hook so that we can assign and remove listeners.



```
function
useOnClickOutside
(
ref, callback
) {
useEffect(
()
=> {
const
listener = (
event
) => {
// if the reference is not present
// or the target is descendant of the reference
// return
if
(!ref.current || ref.current.contains(event.target))
{
return
;
}
```

```
}  
  
// invoke the callback  
callback(event);  
  
};  
  
document  
  .addEventListener(  
    "mousedown"  
    , listener);  
  
document  
  .addEventListener(  
    "touchstart"  
    , listener);  
  
return  
  
()  
  
=> {  
  
  document  
    .removeEventListener(  
      "mousedown"  
      ,  
      listener);
```

460

document

.removeEventListener(

"touchstart"

, listener);

};

},

// add ref and callback to effect dependencies

[ref, callback]

);

}

Test Case

Input:

function

Example

() {

const

ref = useRef();

useOnClickOutside(ref, () => {

console

.log(

"Clicked"

);

});

return

(

<div>

<p>

Outside Click me!

</p>

<p ref=

{ref}

>

Click me!

</p>

</div>

);

}

Output:

"Clicked"

// when clicked on Outside Click me!

© JavaScript Interview Guide | learnersbucket.com

461

useHasFocus() hook

Implement a hook in react that helps to determine if the application is in focus or not. This will help stop the background processing when

the user is not focused or on the tab.

To create the useHasFocus() we will have to listen to the focus and blur events on the window.

Use useState() to persist the state and useEffect() to assign and remove the event listeners.

Whenever the window is blurred, set the focus state to false , else whenever the window is focused, update the state to true .

Use the [document.hasFocus\(\)](#) to get the initial state.

import

{ useState, useEffect }

from

"react"

;

const

useHasFocus =

```
()  
  
=> {  
  
  // get the initial state  
  
  const  
  
  [focus, setFocus] = useState(  
  
    document  
  
    .hasFocus());  
  
  useEffect(  
  
    ()  
  
    => {  
  
      // helper functions to update the status  
  
      const  
  
      onFocus =  
  
      ()  
  
      => setFocus(  
  
        true  
  
      );  
  
      const  
  
      onBlur =  
  
      ()
```

```
=> setFocus(
false
);
// assign the listener
// update the status on the event
window
.addEventListener(
"focus"
, onFocus);
window
.addEventListener(
"blur"
, onBlur);
// remove the listener
return
()
=> {
window
.removeEventListener(
"focus"
```

```
, onFocus);
```

```
window
```

```
.removeEventListener(
```

```
"blur"
```

```
, onBlur);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

462

```
};
```

```
}, []);
```

```
// return the status
```

```
return
```

```
focus;
```

```
};
```

Test Case

Input:

```
const
```

Example =

```
()
```

```
=> {
```

```
const
```



```
focus = useHasFocus();
```

```
console
```

```
.log(focus);
```

```
return
```

```
<></>
```

```
;
```

```
};
```

Output:

```
true
```

```
false // change the tab
```

```
true // back to the tab
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

463

useToggle() hook

Implement the useToggle() hook in React that accepts an array of

values and the start index and toggles the next value in the array. If the index reaches to the last index of the array, reset the index.

This hook will return the current value and the toggle() method using which we can toggle the values.

To create this hook all we need to do is track the indexes and we can use the index to return the value from the array thus, use the

useState() hook to persist and track indexes and then use the

useCallback() hook to create the toggle function.

By using the useCallback() hook, we can memorize the toggle method so that it can be passed down through components.

```
import
```

```
{ useCallback, useState }
```

```
from
```

```
"react"
```

```
;
```

```
const
```

```
useToggle = (
```

```
values, startIndex = 0
```

```
) => {
```

```
// to track the indexes
```

```
const
```

```
[index, setIndex] = useState(startIndex);
```

```
// define and memorize the toggler function in case
```

```
we pass down the
```

```
component,
```

```
// this will move the index to the next level and
```

```
reset it if it goes
```

beyond the limit.

```
const
```

```
toggle = useCallback(
```

```
()
```

```
=> setIndex((
```

```
prevIndex
```

```
) => (prevIndex >= values.length
```

```
-
```

```
1
```

```
?
```

```
0
```

```
:
```

```
prevIndex +
```

```
1
```

```
)),
```

```
[values]
```

```
);
```

```
// return value and toggle function
```

```
return
```

```
[values[index], toggle];
```

```
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

464

Test Case

Input:

function

Example

```
() {
```

```
// call the hook which returns, the current value
```

```
and the toggled
```

```
function
```

```
const
```

```
[currentValue, toggleValue] = useToggle([
```

```
"a"
```

```
,
```

```
"b"
```

```
,
```

```
"c"
```

```
,
```

```
"d"
```

```
],  
2  
);  
  
return  
  
<button onClick=  
  
{toggleValue}  
  
>  
  
"currentValue"  
  
:  
  
{currentValue}  
  
</button>  
  
;  
  
}
```

export

default

Example;

Output:

currentValue: c

// initially

currentValue: d

```
// onClick
```

```
currentValue: a
```

```
// onClick
```

```
currentValue: b
```

```
// onClick
```

```
currentValue: c
```

```
// onClick
```

© JavaScript Interview Guide | learnersbucket.com

465

useCopy() hook

Implement an useCopy() hook in React that copies the given text to the clipboard.

The useCopy() method returns a method copy(text) which accepts the text as input and copies that text and the copied text.

To implement this function we will use the browser's inbuilt method

that allows copying things [navigator.clipboard](#).

navigator.clipboard has two methods.

- *writeText(text)* – Used to copy any given text.
- *readText()* – Used to read the copied text.

Both operations are asynchronous and return promises. For our use,

we will use the writeText(text) and wrap this inside the try...catch block.

We will also use the `useState()` hook to persist the copied text. If the promise is fulfilled then update the state with the text else set the state to null.

This operation will take place inside the `copy()` function that accepts the text as input and tries to copy that.

```
import
{ useState }
from
"react"
;
const
useCopy =
()
=> {
const
[copiedText, setCopiedText] = useState();
const
copy =
async
(text) => {
if
```

```
(!navigator?.clipboard) {  
  console  
  .warn(  
    "Clipboard not supported"  
  );
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

466

```
  return  
  false  
  ;  
}  
  
// try to save to clipboard then save it in the  
state if worked  
  
try  
{  
  await  
  navigator.clipboard.writeText(text);  
  setCopiedText(text);  
}  
  
catch
```



```
(error) {  
  console  
    .error(  
      `Failed copying the text ${text}`  
    ,  
      error);  
  setCopiedText(  
    null  
  );  
}  
};  
  
return  
[copiedText, copy];
```

[OceanofPDF.com](https://oceanofpdf.com)

```
};
```

Test Case

Input:

function

Example

```
() {
```

```
// call the hook which returns, copied text and
```

```
the copy function
```

```
const
```

```
[copiedText, copy] = useCopy();
```

```
return
```

```
<button onClick=
```

```
{()
```

```
=>
```

```
copy("Hello World!")}>
```

```
"copiedText" :
```

```
{copiedText}
```

```
</button>
```

```
;
```

```
}
```

export

default

Example;

Output:

copiedText:

// initially

copiedText: Hello World!

// after click

© JavaScript Interview Guide | learnersbucket.com

467

useLockedBody() hook

Implement the useLockedBody() hook in React that will lock the body from further scrolling.

The useLockedBody() hook will take the reference of the parent and return the lock state and the method that will toggle the lock state.

To lock the body, we will have to remove the overflow from the body so that everything inside it is prevented from scrolling and hide the scrollbar.

To hide the scroll bar, get the [scrollWidth](#) of the referenced element and add the same size right padding to the body to cover the gap.

This complete processing will take inside the [useLayoutEffect\(\)](#) hook as the side effect is with the DOM.

Use a state to monitor the toggling and depending upon the toggle

state, lock or unlock the body.

```
const
```

```
useLockedBody = (
```

```
ref, initiallyLocked = false
```

```
)
```

```
=> {
```

```
const
```

```
[locked, setLocked] = useState(initiallyLocked);
```

```
// handle side effects before render
```

```
useLayoutEffect(
```

```
()
```

```
=> {
```

```
if
```

```
(!locked) {
```

```
return
```

```
;
```

```
}
```

```
// save original body style
```

```
const
```

```
originalOverflow =
```

```
document
.body.style.overflow;

const
originalPaddingRight =
document
.body.style.paddingRight;

// lock body scroll

document
.body.style.overflow =
"hidden"

;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

468

```
// get the scrollBar width

const

root = ref.current;

// or root

const

scrollBarWidth = root ? root.offsetWidth

- root.scrollWidth :
```

```
0
;
// prevent width reflow
if
(scrollBarWidth) {
document
.body.style.paddingRight =
`${scrollBarWidth}px`
;
}
// clean up
return
()
=> {
document
.body.style.overflow = originalOverflow;
if
(scrollBarWidth) {
document
.body.style.paddingRight = originalPaddingRight;
```

```
}  
};  
}, [locked]);  
  
// update state when dependency changes  
useEffect(  
  ()  
  => {  
    if  
      (locked !== initiallyLocked) {  
        setLocked(initiallyLocked);  
      }  
    }, [initiallyLocked]);  
return  
[locked, setLocked];  
};
```

Test Case

Input:

const

Example =

()

```
=> {  
  
  const  
  
  ref = useRef();  
  
  // call the hook which returns, current value and  
  the toggler function  
  
  const  
  
  [locked, setLocked] = useLockedBody(ref);  
  
  return  
  
  (  
  
    <div style=  
  
      {{  
  
        height: "200vh" }} id=  
  
        "abc"  
  
        ref=  
  
        {ref}  
  
      >  
  
      <button onClick=  
  
        {}  
  
      >  
  
        setLocked(!locked)}>{locked
```



```
? "unlock scroll"
```

© JavaScript Interview Guide | learnersbucket.com

469

```
: "lock scroll"}
```

```
</button>
```

```
</div>
```

```
);
```

```
};
```

```
// click on the button to lock and unlock body locking
```

© JavaScript Interview Guide | learnersbucket.com

470

Number increment counter

Problem Statement -

Create a web app in Javascript or any framework of your choice (React, Angular, Vue), which takes a number and a duration as an

input and prints the number starting from 0 incrementing it by 1 in the given duration.

Functional component that will take input.

```
//App.js
```

```
import
```

```
React, { useState }
```

```
from
"react"
;
const
App =
()
=> {
const
[number, setNumber] = useState(
0
);
const
[duration, setDuration] = useState(
0
);
const
[start, setStart] = useState(
false
);
//If any input changes reset
```

```
const
basicReset =
()
=> {
  setStart(
    false
  );
};

//store number
const
numberChangeHandler = (
  e
) => {
  const
  { value } = e.target;
  setNumber(value);
  basicReset();
};

//store duration
const
```

```
durationChangeHandler = (
```

```
e
```

```
) => {
```

```
const
```

```
{ value } = e.target;
```

```
setDuration(value);
```

```
basicReset();
```

```
};
```

```
const
```

```
startHandler =
```

```
()
```

```
=> {
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
471
```

```
// trigger the animation
```

```
};
```

```
const
```

```
resetHandler =
```

```
()
```

```
=> {
```

window

.location.reload();

};

return

(

<main style={{ width: "500px", margin: "50px auto" }}>

<section className="input-area">

<div>

<div>

<label htmlFor="number">Number:</label>{ " " }

<input

id="number"

type="number"

value={number}

onChange={numberChangeHandler}

/>

</div>

<div>

<label htmlFor="duration">Duration:</label>{ " " }

<input

```
id="duration"

type="number"

value={duration}

onChange={durationChangeHandler}

/>

</div>

</div>

<br />

<div>

<button onClick={startHandler}>start</button>{" "}

<button onClick={resetHandler}>reset</button>

</div>

</section>

</main>

);

};

export default App;
```

© JavaScript Interview Guide | learnersbucket.com

Number:

Duration:

### Solution 1 : Using setInterval

If you are a straight forward developer like me who likes to try the familiar approaches, then the first thing that comes to your mind is using a setInterval function.

It is extremely simple to come up with a solution using setInterval, all we have to do is,

Calculate the time interval at which the setInterval should be called in order to increment the number.

We can do that with this simple formula  $(\text{duration} / \text{number}) * 1000$  , for example,  $(2 / 1000) * 1000 = 2$  , which means we have to increment the counter every 2 milliseconds to reach from 0 to 1000 in 2 seconds.

Now there are two ways in which you can implement this in react,

1. Ref to the DOM element and increment the count directly in each interval call.
2. Update the state and let react update the count.

Both of these approaches do not affect the time because all we are

doing is updating a single DOM element, if we had to update multiple nested DOM elements then we should be using the second approach.

```
//CountMethods.js
```

```
import
```

```
React, { useEffect, useState, useRef }
```

```
from
```

```
"react"
```

```
;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

473

```
//setInterval
```

```
const
```

```
CountSetInterval = (
```

```
props
```

```
) => {
```

```
const
```

```
intervalRef = useRef();
```

```
const
```

```
countRef = useRef();
```

```
// label of counter
```

```
// number to increment to
```

```
// duration of count in seconds
```

```
const
```

```
{ number, duration } = props;
```



```
// number displayed by component

const

[count, setCount] = useState(

"0"

);

// calc time taken for computation

const

[timeTaken, setTimeTaken] = useState(

Date

.now());

useEffect(

()

=> {

let

start =

0

;

// first three numbers from props

const

end =
```

```
parseInt
(number);
// if zero, return
if
(start === end)
return
;
// find duration per increment
let
totalMilSecDur =
parseInt
(duration);
let
incrementTime = (totalMilSecDur / end) *
1000
;
// timer increments start counter
// then updates count
// ends if start reaches end
let
```

```
timer = setInterval(  
    ()  
    => {  
        start +=  
        1  
        ;  
        //update using state  
        setCount(  
            String  
            (start));  
        //update using ref  
        // countRef.current.innerHTML = start;  
        if  
        (start === end) {  
            clearInterval(timer);  
            const  
            diff =  
            Date  
            .now() - timeTaken;
```

474

```
setTimeTaken(diff /  
1000  
);  
//uncomment this when using ref  
// setCount(String(start));  
}  
}, incrementTime);  
// dependency array  
}, [number, duration]);  
return  
(  
<>  
<span ref=  
{countRef}  
className=  
"Count"  
>  
{count}  
</span>
```

```
{ " "
```

```
{ " "
```

```
{number === count && (
```

```
<span>
```

```
| Took :
```

```
<b>
```

```
{timeTaken}
```

```
</b>
```

```
seconds to complete
```

```
</span>
```

```
)}
```

```
</>
```

```
);
```

```
};
```

I have also added a time log to determine exactly how much time it

takes to increment the count in order to make sure we are progressing in the right direction.

Let's call this function on the click of the start button inside the input function.

```
// app.js
```

```
import
```

```
React, { useState }  
  
from  
  
"react"  
  
;  
  
import  
  
{ CountSetInterval }  
  
from  
  
"./CountMethods"  
  
;  
  
const  
  
App =  
  
()  
  
=> {  
  
  const  
  
  [number, setNumber] = useState(  
  
    0  
  
  );  
  
  const  
  
  [duration, setDuration] = useState(  
  
    0
```

```
);
```

```
const
```

```
[start, setStart] = useState(
```

```
false
```

```
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

475

```
//If any input changes reset
```

```
const
```

```
basicReset =
```

```
()
```

```
=> {
```

```
setStart(
```

```
false
```

```
);
```

```
};
```

```
//store number
```

```
const
```

```
numberChangeHandler = (
```

```
e
```

```
) => {  
  
  const  
  
    { value } = e.target;  
  
  setNumber(value);  
  
  basicReset();  
  
};  
  
//store duration  
  
const  
  
durationChangeHandler = (  
  
  e  
  
  ) => {  
  
    const  
  
      { value } = e.target;  
  
    setDuration(value);  
  
    basicReset();  
  
    };  
  
    const  
  
    startHandler =  
  
    ()  
  
    => {
```



```
// trigger the animation

setStart(

true

);

};

const

resetHandler =

()

=> {

window

.location.reload();

};

return

(

<main style={{ width: "500px", margin: "50px auto" }}>

<section className="input-area">

<div>

<div>

<label>Number:</label>{" "}

<input
```

```
type="number"
value={inputValue}
onChange={inputChangeHandler}
/>
```

```
</div>
```

```
<div>
```

© JavaScript Interview Guide | learnersbucket.com

476

```
<label>Duration:</label>{" "}
```

```
<input
```

```
type="number"
```

```
value={duration}
```

```
onChange={durationChangeHandler}
```

```
/>
```

```
</div>
```

```
</div>
```

```
<br />
```

```
<div>
```

```
<button onClick={startHandler}>start</button>{" "}
```

```
<button onClick={resetHandler}>reset</button>
```

```
</div>
```

```
</section>
```

```
<br />
```

```
<section className="result-area">
```

```
<div>
```

```
SetInterval:{" "
```

```
{(start && (
```

```
<CountSetInterval
```

```
label={"count"}
```

```
number={inputValue}
```

```
duration={parseInt(duration)}
```

```
/>
```

```
)) ||
```

```
0}
```

```
</div>
```

```
</section>
```

```
</main>
```

```
);
```

```
};
```

```
export default App;
```

477



Number: 1000

Duration: 2

start reset

SetInterval: 0

## Output

Weird!, right?

It is taking longer than we expected to increment the count even though we are doing everything properly.

Well, it turns out that the `setInterval` function is not behaving as we have thought it should.

## Solution 2: Using `setTimeout`

Let's change the approach and try to implement the same logic using `setTimeout`.

We can mimic the `setInterval` function using `setTimeout` by recursively calling the same function.

Using the same calculation, let's implement this.

```
//setTimeout
```

```
const  
CountSetTimeout = (  
  props  
) => {  
  const  
  intervalRef = useRef();  
  const  
  countRef = useRef();
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

478

```
// label of counter  
  
// number to increment to  
  
// duration of count in seconds  
  
const  
  
{ number, duration } = props;  
  
// number displayed by component  
  
const  
  
[count, setCount] = useState(  
  "0"  
);
```

```
// calc time taken for computation

const

[timeTaken, setTimeTaken] = useState(

Date

.now());

useEffect(

()

=> {

let

start =

0

;

// first three numbers from props

const

end =

parseInt

(number);

// if zero, return

if

(start === end)
```

```
return

;

// find duration per increment

let

totalMilSecDur =

parseInt

(duration);

let

incrementTime = (totalMilSecDur / end) *

1000

;

// timer increments start counter

// then updates count

// ends if start reaches end

let

counter =

()

=> {

intervalRef.current = setTimeout(

()
```

```
=> {  
  
start +=  
  
1  
  
;  
  
//update using state  
  
setCount(  
  
String  
  
(start));  
  
//update using ref  
  
// countRef.current.innerHTML = start;  
  
counter();  
  
if  
  
(start === end) {  
  
clearTimeout(intervalRef.current);  
  
const  
  
diff =  
  
Date  
  
.now() - timeTaken;  
  
//uncomment this when using ref  
  
// setCount(String(start));
```



479

```
setTimeTaken(diff /
```

```
1000
```

```
);
```

```
}
```

```
}, incrementTime);
```

```
};
```

```
//invoke
```

```
counter();
```

```
// dependency array
```

```
}, [number, duration]);
```

```
return
```

```
(
```

```
<>
```

```
<span ref=
```

```
{countRef}
```

```
className=
```

```
"Count"
```

```
>
```

```

    {count}
  </span>

  { " " }

  { " " }

  {number === count && (
    <span>
      | Took :
      <b>
        {timeTaken}
      </b>
      seconds to complete
    </span>
  )}
</>

);

};

```

Let us see what happens when we invoke this function on the click of the start button.

```

// app.js

import

React, { useState }

```

```
from
"react"
;
import
{ CountSetTimeout }
from
"./CountMethods"
;
const
App =
()
=> {
const
[number, setNumber] = useState(
0
);
const
[duration, setDuration] = useState(
0
);
```

```
const  
[start, setStart] = useState(  
false  
);  
//If any input changes reset
```

```
const  
basicReset =  
()  
=> {  
setStart(  
false  
);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
480
```

```
};
```

```
//store number
```

```
const  
numberChangeHandler = (  
e  
) => {
```

```
const
{ value } = e.target;
setNumber(value);
basicReset();
};

//store duration

const
durationChangeHandler = (
e
) => {
const
{ value } = e.target;
setDuration(value);
basicReset();
};

const
startHandler =
()
=> {
// trigger the animation
```

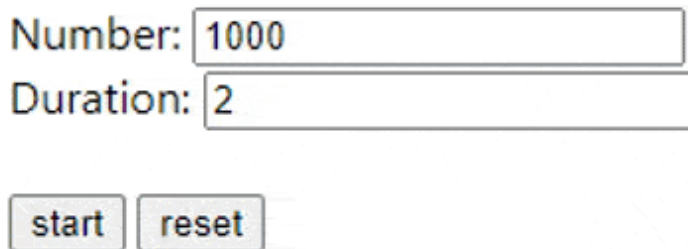
```
setStart(  
  true  
);  
};  
  
const  
resetHandler =  
()  
=> {  
  window  
  .location.reload();  
};  
  
return  
(  
  <main style={{ width: "500px", margin: "50px auto" }}>  
    <section className="input-area">  
      <div>  
        <div>  
          <label>Number:</label>{" "  
          <input  
            type="number"
```

```
value={inputValue}  
onChange={inputChangeHandler}  
/>  
  
</div>
```

```
<div>  
  
<label>Duration:</label>{" "  
  
<input  
type="number"  
value={duration}
```

© JavaScript Interview Guide | learnersbucket.com

481



The screenshot shows a web form with two input fields. The first field is labeled "Number:" and contains the value "1000". The second field is labeled "Duration:" and contains the value "2". Below the input fields are two buttons: "start" and "reset".

SetTimeout: 0

```
onChange={durationChangeHandler}  
/>  
  
</div>  
  
</div>
```

<br />

<div>

<button onClick={startHandler}>start</button>{" "}

<button onClick={resetHandler}>reset</button>

</div>

</section>

<br />

<section className="result-area">

<div>

SetInterval:{" "}

{(start && (

<CountSetTimeout

label={"count"}

number={inputValue}

duration={parseInt(duration)}

/>

)) ||

0}

</div>

</section>



```
</main>
```

```
);
```

```
};
```

```
export default App;
```

Output

© JavaScript Interview Guide | learnersbucket.com

482

This is taking more time than the `setInterval`.

Why is this happening?.

If you read the definition of each of these methods you will realize that.

- *setTimeout* : This method sets a timer which executes a function or specified piece of code once the timer expires.
- *setInterval* : This method repeatedly calls a function or executes a code snippet, with a fixed time delay between each call.

Which means the time specified for either of these functions is the minimum time, but it can take longer than that.

After some research on MDN, I found out that there are two major reasons which are causing the delay.

### 1. Clamping.

In modern browsers, `setTimeout()/setInterval()` calls are throttled to a minimum of once every 4 ms when successive calls are triggered due

to callback nesting (where the nesting level is at least a certain depth), or after certain number of successive intervals.

## 2.Execution context

The timer can also fire later when the page (or the OS/browser itself) is busy with other tasks. One important case to note is that the

function or code snippet cannot be executed until the thread that

called `setTimeout()` has terminated.

© JavaScript Interview Guide | learnersbucket.com

483

It turns out that, `setTimeout` or `setInterval` function won't function properly when

1. Delay is less than 4 ms.
2. There is execution happening inside timer functions which is blocking the next execution.

If we can somehow avoid using the timer functions, we should be able to solve this problem. But is there a way without using them?.

Turns out there is a way in which it can be implemented.

Solution 3: Using [requestAnimationFrame](#) method .

Using this method we can come up with a solution which would

increment the count from 0 to the specified number in given duration.

First, let us understand what this method is.

According to MDN –

The `window.requestAnimationFrame()` method tells the browser that you wish to perform an animation and requests that the browser calls a

specified function to update an animation before the next repaint. The method takes a callback as an argument to be invoked before the repaint.

In simple terms what this method does is ask the browser to perform animation, which in turn refreshes the screen very fast. At-least 60 frames per second to perform animations.

Now how is this useful in creating an increment counter?.

Read this text from MDN for better understanding.

© JavaScript Interview Guide | learnersbucket.com

484

You should call this method whenever you're ready to update your animation on screen. This will request that your animation function be called before the browser performs the next repaint. The number of

callbacks is usually 60 times per second, but will generally match the display refresh rate in most web browsers as per W3C recommendation.

This function takes a callback function and passes the current timestamp to that callback function as an argument.

Now using a `startTime` variable which stores the time before invoking the function and this current timestamp which we receive in the

callback every time, we can recursively invoke this function for the given duration and using a good calculation we can increment the count.

Considering at-least 60 frames are refreshed in one second, which

means if we have to count from 0 to 1000 in 1 second we should be incrementing the number by  $60/1000 = \sim 16.667$  using which we can come with clever calc which will increment the number based on much time has passed of animation.

Depending upon how often this function is invoked we will see an increment animation happening on the screen.

For bigger numbers this is not incrementing the count by 1 but still the animation is happening so fast that human eyes will not be able to differentiate.

Note:- In gif you should be able to see this.

```
//Animation  
  
const  
  
countAnimate = (  
  obj, initVal, lastVal, duration  
)  
=> {  
  let  
  
  startTime =  
  
  null  
  
  ;
```

485

```
//get the current timestamp and assign it to the  
currentTime variable  
  
let  
  
currentTime =  
  
Date  
  
.now();  
  
//pass the current timestamp to the step function  
  
const  
  
step = (  
  
currentTime  
  
) => {  
  
//if the start time is null, assign the current  
time to startTime  
  
if  
  
(!startTime) {  
  
startTime = currentTime;  
  
}  
  
//calculate the value to be used in calculating  
the number to be
```

```
displayed
const
progress =
Math
.min((currentTime - startTime)
/ duration,
1
);
//calculate what to be displayed using the value
gotten above
obj.innerHTML =
Math
.floor(progress * (lastVal
- initVal) + initVal);
//checking to make sure the counter does not exceed
the last value
(lastVal)
if
(progress <
1
```

```
) {  
  window  
    .requestAnimationFrame(step);  
}  
else  
{  
  window  
    .cancelAnimationFrame(  
      window  
        .requestAnimationFrame(step));  
  // add time diff  
  const  
    diff = currentTime - startTime;  
  const  
    elm =  
      document  
        .createElement(  
          "SPAN"  
        );  
  elm.innerHTML =
```

```
` | Took : <b>${diff
```

```
/ 1000}</
```

```
b>
```

```
seconds to
```

```
complete;
```

```
obj.appendChild(elm);
```

```
}
```

```
};
```

```
//start animating
```

```
window.requestAnimationFrame(step);
```

```
};
```

Let's test it out.

```
// app.js
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

486

```
import
```

```
React, { useState, useRef }
```

```
from
```

```
"react"
```

```
;
```



```
import
{ countAnimate }
from
"./CountMethods"
;
const
App =
()
=> {
const
[number, setNumber] = useState(
0
);
const
[duration, setDuration] = useState(
0
);
const
[start, setStart] = useState(
false
```

```
);  
  
const  
  
countRef = useRef();  
  
//If any input changes reset  
  
const  
  
basicReset =  
  
()  
  
=> {  
  
  setStart(  
  
    false  
  
  );  
  
  countRef.current.innerHTML =  
  
    "0"  
  
  ;  
  
  };  
  
  //store number  
  
  const  
  
  numberChangeHandler = (  
  
    e  
  
  ) => {
```

```
const
{ value } = e.target;
setNumber(value);
basicReset();
};

//store duration

const
durationChangeHandler = (
e
) => {
const
{ value } = e.target;
setDuration(value);
basicReset();
};

const
startHandler =
()
=> {
// trigger the animation
```

```
setStart(  
  true  
);  
countAnimate(  
  countRef.current,  
  0  
  ,  
  parseInt  
    (inputValue),  
  parseInt  
    (duration) *  
    1000  
);  
};  
  
const  
resetHandler =  
()  
=> {  
  window  
    .location.reload();
```

487

};

return

(

<main style={{ width: "500px", margin: "50px auto" }}>

<section className="input-area">

<div>

<div>

<label>Number:</label>{" "}

<input

type="number"

value={inputValue}

onChange={inputChangeHandler}

/>

</div>

<div>

<label>Duration:</label>{" "}

<input

type="number"

```
value={duration}
onChange={durationChangeHandler}
/>

</div>

</div>

<br />

<div>

<button onClick={startHandler}>start</button>{" "}

<button onClick={resetHandler}>reset</button>

</div>

</section>

<br />

<section className="result-area">

<div>

Animate: <span ref={countRef}>0</span>

</div>

</section>

</main>

);

};
```

export default App;

© JavaScript Interview Guide | learnersbucket.com

488

Number:

Duration:

Animate: 0

Number:

Duration:

SetTimeout: 0  
SetInterval: 0  
Animate: 0

Output

Comparison of all the three solutions

Watch this video to get a better [understanding of animations in the browser.](#)

Please try it out yourself, code is available on [github repo.](#)

489

Capture product visible on viewport

Problem Statement -

This question was asked to a friend in a real-estate property site's interview.

The question was quoted as “If a user scrolls and sees any property and stays there for more than 5 sec then call API and store that property”.

Apart from the online real-estate platform, this can be also applied on other platforms as well, such as social media like Facebook where if users read a post for a few seconds, store and use it to provide

recommendations of new posts. The same can be used on E-commerce platforms or other platforms where products are listed.

Let us see how we should approach such problems and then solve them with an example. I have created a dummy HTML template that contains different blocks, which can be used for testing.

To solve this, we will use the methodology we have used in the

[useOnScreen\(\)](#) hook that will determine if the current element is inside the viewport or not. If it is under the viewport, make an api call and log it.

For the demo purpose, let's create a block.

const

Elements = (



```
{ index }
```

```
) => {
```

```
// element style
```

```
const
```

```
style = {
```

```
flex:
```

```
"1 300px"
```

```
,
```

```
height:
```

```
"300px"
```

```
,
```

```
display:
```

```
"inline-flex"
```

```
,
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
490
```

```
alignItems:
```

```
"center"
```

```
,
```

```
justifyContent:
```

"center"

,

margin:

"5px"

,

background:

"red"

,

fontSize:

"40px"

,

color:

"#fff"

,

};

return

(

<div style=

{style}

>

```
<div>
```

```
{index}
```

```
</div>
```

```
</div>
```

```
);
```

```
};
```

Form an array of blocks from this element and render this inside a

wrapper so that the screen becomes scrollable and we will be able to determine the elements that are in viewport or are visible.

Create a reference to this wrapper element so that we can get all its children and then check which of them are visible.

```
const
```

```
Example =
```

```
()
```

```
=> {
```

```
// create a reference for the block
```

```
const
```

```
ref = useRef([]);
```

```
// render 20 blocks of elements
```

```
const
```

```
elementsList = [];
```

```
for
(
let
i =
1
; i <=
20
; i++) {
elementsList.push(
<Elements index=
{i}
/>
);
}

// element style
const wrapperStyle = {
display: "flex",
alignItems: "center",
justifyContent: "center",
flexWrap: "wrap",
```

```
};
```

© JavaScript Interview Guide | learnersbucket.com

491

```
return (
```

```
<div style=
```

```
{wrapperStyle}
```

```
ref=
```

```
{ref}
```

```
>
```

```
{elementsList}
```

```
</div>
```

```
);
```

```
};
```

Create a helper function using [getBoundingClientRect\(\)](#) rather than using [IntersectionObserver API](#) as using the simple calculation we can determine the position of the elements in the viewport to check if it is visible rather than keep on observing it.

```
// Helper function to check if element is in viewport
```

```
const
```

```
isInViewPort =
```

```
function
```

```
(
```

```
elem
) {
const
bounding = elem.getBoundingClientRect();
return
(
bounding.top >=
0
&&
bounding.left >=
0
&&
bounding.bottom <= (
window
.innerHeight ||
document
.documentElement.clientHeight) &&
bounding.right <= (
window
.innerWidth ||
```

```
document  
  
.documentElement.clientWidth)  
  
);  
  
};
```

Finally listen to the [debounced](#) scroll event after a delay and check what all elements are visible when the user stops scrolling.

```
// make api call  
  
const  
  
makeApiCall =  
  
async  
  
() => {  
  
  // array of child elements  
  
  const  
  
  htmlArray = [...ref.current.children];  
  
  // check if the child element is in viewport  
  
  // perform further actions  
  
  htmlArray.forEach((  
  
    e  
  
  ) => {  
  
    console  
  
.log(e.innerText, isInViewport(e));
```

492

```
});
```

```
};
```

```
// debounced api call after 5 sec
```

```
const
```

```
debouncedApiCall = useDebounce(makeApiCall,
```

```
5000
```

```
);
```

```
useEffect(
```

```
()
```

```
=> {
```

```
window
```

```
.addEventListener(
```

```
"scroll"
```

```
, debouncedApiCall);
```

```
return
```

```
()
```

```
=> {
```

```
window
```



```
.removeEventListener(  
  "scroll"  
  , debouncedApiCall);  
};  
}, []);
```

Putting everything together.

```
import  
  { useState, useEffect, useRef, useCallback  
  }  
from  
  "react"  
;  
  
const  
  useDebounce = (  
    fn, delay, immediate = false  
  )  
=> {  
  // ref the timer  
  
  const  
  
    timerId = useRef();
```

```
// create a memoized debounce

const

debounce = useCallback(

function

() {

// reference the context and args for the setTimeout

function

let

context =

this

,

args =

arguments

;

// should the function be called now? If immediate

is true

// and not already in a timeout then the answer

is: Yes

const

callNow = immediate && !timerId.current;
```

```
// base case  
  
// clear the timeout to assign the new timeout  
to it.
```

```
// when event is fired repeatedly then this  
helps to reset
```

```
clearTimeout(timerId.current);
```

```
// set the new timeout
```

```
timerId.current = setTimeout(  
function
```

```
() {
```

```
// Inside the timeout function, clear the  
timeout variable
```

```
// which will let the next execution run when  
in 'immediate' mode
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
493
```

```
timerId.current =
```

```
null
```

```
;
```

```
// check if the function already ran with
```

```
the immediate flag

if
(!immediate) {
// call the original function with apply
fn.apply(context, args);
}
}, delay);

// immediate mode and no wait timer? Execute
the function immediately

if
(callNow) fn.apply(context, args);
},
[fn, delay, immediate]
);

return
debounce;
};

function
useOnScreen
(
```

```
ref
) {
  const
  [isIntersecting, setIntersecting] = useState(
    false
  );
  // monitor the interaction
  const
  observer =
  new
  IntersectionObserver((
    [entry]
  )
  => {
    // update the state on interaction change
    setIntersecting(entry.isIntersecting);
  });
  useEffect(
    ()
    => {
```

```
// assign the observer

observer.observe(ref.current);

// remove the observer as soon as the component
is unmounted

return

()

=> {

observer.disconnect();

};

}, []);

return

isIntersecting;

}

const

Elements = (

{ index }

) => {

// element style
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
const
style = {
flex:
"1 300px"
,
height:
"300px"
,
display:
"inline-flex"
,
alignItems:
"center"
,
justifyContent:
"center"
,
margin:
"5px"
,
```

background:

"red"

,

fontSize:

"40px"

,

color:

"#fff"

,

};

return

(

<div style=

{style}

>

<div>

{index}

</div>

</div>

);



```
};
```

```
const
```

```
Example =
```

```
()
```

```
=> {
```

```
// create a reference for the block
```

```
const
```

```
ref = useRef([]);
```

```
// Helper function to check if element is in viewport
```

```
const
```

```
isInViewPort =
```

```
function
```

```
(
```

```
elem
```

```
) {
```

```
const
```

```
bounding = elem.getBoundingClientRect();
```

```
return
```

```
(
```

```
bounding.top >=
```

0

&&

bounding.left >=

0

&&

bounding.bottom <= (

window

.innerHeight ||

document

.documentElement.clientHeight) &&

bounding.right <= (

window

.innerWidth ||

document

.documentElement.clientWidth)

);

};

// make api call

const

makeApiCall =

```
async
```

```
() => {
```

```
// array of child elements
```

```
const
```

```
htmlArray = [...ref.current.children];
```

```
// check if the child element is in viewport
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

495

```
// perform further actions
```

```
htmlArray.forEach((
```

```
e
```

```
) => {
```

```
if(isInViewPort(e)){
```

```
console
```

```
.log(e.innerText);
```

```
}
```

```
});
```

```
};
```

```
// debounced api call after 5 sec
```

```
const
```

```
debouncedApiCall = useDebounce(makeApiCall,
5000
);
useEffect(
()
=> {
window
.addEventListener(
"scroll"
, debouncedApiCall);
return
()
=> {
window
.removeEventListener(
"scroll"
, debouncedApiCall);
};
}, []);
// render 20 blocks of elements
```

```
const
elementsList = [];

for
(
let
i =
1
; i <=
20
; i++) {
elementsList.push(
<Elements index=
{i}
/>
);
}

// element style
const wrapperStyle = {
display: "flex",
alignItems: "center",
```

```
justifyContent: "center",
```

```
flexWrap: "wrap",
```

```
};
```

```
return (
```

```
<div style=
```

```
{wrapperStyle}
```

```
ref=
```

```
{ref}
```

```
>
```

```
{elementsList}
```

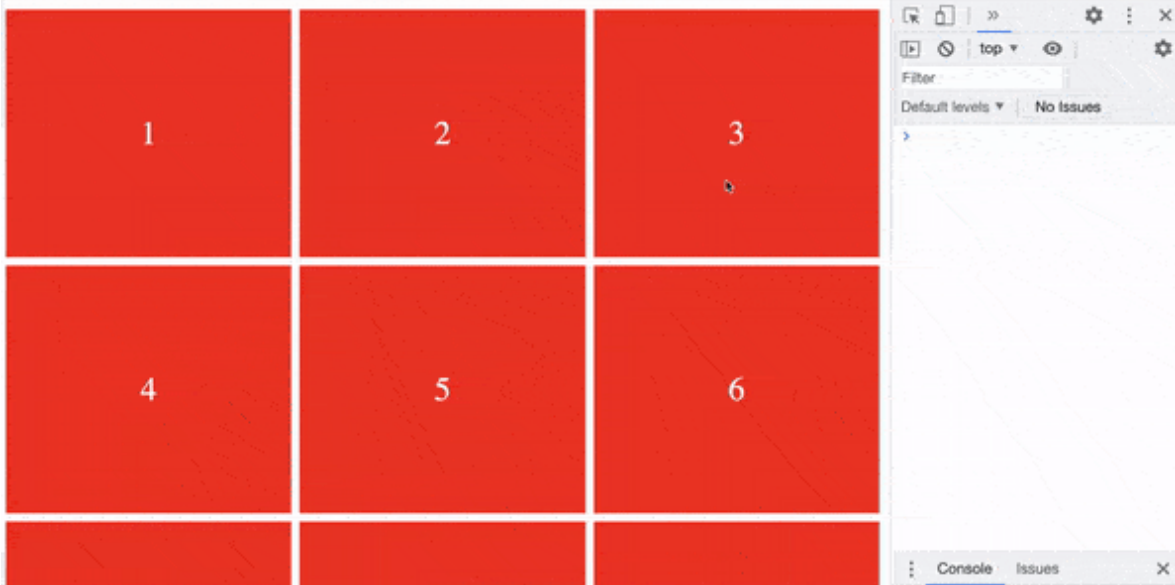
```
</div>
```

```
);
```

```
};
```

```
export default Example;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)



## Output

© JavaScript Interview Guide | learnersbucket.com

497

## Highlight text on selection

Implement a hook in React that will return the selected text on the web page and the coordinates of the selection so that a popup can be shown to Tweet the selected text just like Medium.

To implement this we will listen to the mouseup event and on the event get the window selection to get the highlighted text and the current node of the text. The text can be overlapping thus we can get the start and the end node.

We can pass the element reference to the hook just to restrict the selection of particular elements. Check if the selected text's start and end node is part of the element then only get the selection and its

coordinates.

Store the value in a state and return the data.

```
const
```

```
useSelectionText = (
```

```
  ref
```

```
) => {
```

```
  // to store the selected text
```

```
  // and tools
```

```
  const
```

```
  [data, setData] = useState({ showTools:
```

```
    false
```

```
  });
```

```
  // handle the mouseup event
```

```
  const
```

```
  onMouseup =
```

```
  ()
```

```
  => {
```

```
    // get the window selection
```

```
    const
```

```
    selection =
```



```
window
.getSelection();
// get the parent node
const
startNode = selection.getRangeAt(
0
).startContainer.parentNode;
// get the end node
const
endNode = selection.getRangeAt(
0
).endContainer.parentNode;
// if the current element is is not part of the
selection node
// do not show tools
© JavaScript Interview Guide | learnersbucket.com
498
if
(!startNode.isSameNode(ref.current) ||
!startNode.isSameNode(endNode)) {
```

```
setData({
  showTools:
    false
  ,
});
return
;
}

// get the coordinates of the selection
const
{ x, y, width } =
selection.getRangeAt(
  0
).getBoundingClientRect();
// if not much is selected
// do not show tools
if
(!width) {
  setData({
    showTools:
```

```
false  
  
,  
  
});  
  
return  
  
;  
  
}  
  
// if text is selected  
  
// update the selection  
  
// and the co-ordinates  
  
// the y position is adjusted to show bar above  
the selection  
  
if  
  
(selection.toString()) {  
  
setData({  
  
x: x,  
  
y: y +  
  
window  
  
.scrollY -  
  
25  
  
,
```

```
showTools:
true
,
selectedText: selection.toString(),
width,
});
}
};

// handle selection
// on the mouseup event
useEffect(
()
=> {
// add the event
document
.addEventListner(
"mouseup"
, onMouseup);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

```
// remove the listener

return

()

=> {

document

.removeEventListener(

"mouseup"

, onMouseup);

};

}, []);

// return the data

return

data;

};

Usage

const

Example =

()

=> {

const
```

```
ref = useRef();

const

data = useSelectionText(ref);

return

(

<div>

{data.showTools && (

// position the popover according to the need

<span

style={{

position: "absolute",

left: `${data.x + data.width / 4}px`,

top: `${data.y}px`,

width: data.width / 2,

display: "inline-block",

height: "20px",

textAlign: "center",

}}

>

{/* twitter icon */}
```

```
<svg style={{ width: "24px", height: "24px" }} viewBox="0 0 24
24">

<path
fill="#000000"
d="M22.46,6C21.69,6.35 20.86,6.58 20,6.69C20.88,6.16
21.56,5.32 21.88,4.31C21.05,4.81 20.13,5.16 19.16,5.36C18.37,4.5 17.26,4
16,4C13.65,4 11.73,5.92 11.73,8.29C11.73,8.63 11.77,8.96
11.84,9.27C8.28,9.09 5.11,7.38 3,4.79C2.63,5.42 2.42,6.16
2.42,6.94C2.42,8.43 3.17,9.75 4.33,10.5C3.62,10.5 2.96,10.3
2.38,10C2.38,10
2.38,10 2.38,10.03C2.38,12.11 3.86,13.85 5.82,14.24C5.46,14.34 5.08,14.39
© JavaScript Interview Guide | learnersbucket.com
500
4.69,14.39C4.42,14.39 4.15,14.36 3.89,14.31C4.43,16 6,17.26
7.89,17.29C6.43,18.45 4.58,19.13 2.56,19.13C2.22,19.13 1.88,19.11
1.54,19.07C3.44,20.29 5.7,21 8.12,21C16,21 20.33,14.46
20.33,8.79C20.33,8.6
20.33,8.42 20.32,8.23C21.16,7.63 21.88,6.87 22.46,6Z"
/>

</svg>

{/* edit icon */}
```

```
<svg style={{ width: "24px", height: "24px" }} viewBox="0 0 24
24">

<path
fill="#000000"
d="M18.5,1.15C17.97,1.15 17.46,1.34
17.07,1.73L11.26,7.55L16.91,13.2L22.73,7.39C23.5,6.61 23.5,5.35
22.73,4.56L19.89,1.73C19.5,1.34 19,1.15
18.5,1.15M10.3,8.5L4.34,14.46C3.56,15.24 3.56,16.5
4.36,17.31C3.14,18.54
1.9,19.77 0.67,21H6.33L7.19,20.14C7.97,20.9 9.22,20.89
10,20.12L15.95,14.16"
/>

</svg>

</span>

)}
```

```
<div ref={ref}>
```

There are many variations of passages of Lorem Ipsum available, but the majority have suffered alteration in some form, by injected humour, or randomised words which don't look even slightly believable. If you are going to use a passage of



Lorem Ipsum, you need to be sure there isn't anything embarrassing hidden in the middle of text. All the Lorem Ipsum generators on the Internet tend to repeat predefined chunks as necessary, making this the first true generator on the Internet. It uses a dictionary of over 200 Latin words, combined with a handful of model sentence structures, to generate Lorem Ipsum which looks reasonable. The generated Lorem Ipsum is therefore always free from repetition, injected humour, or non-characteristic words etc. Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

501

Ipsum available, but the majority have suffered alteration in some form, by injected humour, or randomised words which don't look even slightly believable. If you are unsure there isn't anything embarrassing hidden in the middle of text. All the Lorem Ipsum generators on the Internet tend to repeat predefined chunks as necessary, making this the first true generator on the Internet. It uses a dictionary of over 200 Latin words, combined with a handful of model sentence structures, to generate Lorem Ipsum which looks reasonable. The generated Lorem Ipsum is therefore always free from repetition, injected humour, or non-characteristic words etc. Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Not only that, but the Lorem Ipsum text is also a very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a well-known book by Cicero, "de Finibus Bonorum et Malorum".

years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

</div>

</div>

);

};

Output

© JavaScript Interview Guide | learnersbucket.com

## Batch api calls in sequence

### Problem Statement -

You are given an info-graphic component where you have to batch call APIs in sequence. Let's say you have 20 APIs to call, batch call 5 APIs together, and the next 5 after the previous one is done, and so on. The first call will take after a delay of 5 seconds and once all the APIs are executed, reset and start from the beginning.

### Example

Input:

// API calls

[

1

,

2

,

3

,

4

,

5

,

6

,

7

,

8

,

9

,

10

,

11

,

12

,

13

,

14

,

15

,

16

```
,
17
,
18
,
19
,
20
]

// [1, 2, 3, 4, 5] // first call after 5 seconds
// [6, 7, 8, 9, 10] // second call
// [11, 12, 13, 14, 15] // third call
// [16, 17, 18, 19, 20] // fourth call
```

We will use a dummy promise that will resolve after 1 second to mimic the API call.

- Then break the array of promises into [chunks](#) of 5.
- Use a state to track the indexes of these subarrays, inside the `useEffect` hook, check if the current index is first then make the call after the 5 seconds, and once all APIs are executed update the state and increment the index. If the index is greater than the subarray size then reset it.
- After the index will update, the `useEffect` hook will be invoked and the subsequent calls will be made, thus all the APIs will be executed

recursively.

- To make the API calls we will use a helper function that will execute all the [promises in parallel](#) and increment the index after the operation.

© JavaScript Interview Guide | learnersbucket.com

503

```
import
```

```
{ useState, useEffect }
```

```
from
```

```
"react"
```

```
;
```

```
// helper function to create promise task
```

```
// that resolves randomly after some time
```

```
const
```

```
asyncTask =
```

```
function
```

```
(
```

```
i
```

```
) {
```

```
  return
```

```
    new
```

```
    Promise
```

```
((
  resolve, reject
) => {
  setTimeout(
    ()
    => resolve(
      `Completing ${i}`
    ),
    1000
  );
});

// helper function to create subarrays of given size
const
chop = (
  arr, size = arr.length
) => {
  //temp array
  const
  temp = [...arr];
```

```
//output

const

output = [];

let

i =

0

;

//iterate the array

while

(i < temp.length) {

//slice the sub-array of given size

//and push them in output array

output.push(temp.slice(i, i + size));

i = i + size;

}

return

output;

};

const

Example =
```



```
()  
  
=> {  
  
  // array of promises  
  
  // 20  
  
  const  
  
  promises = [  
  
    asyncTask(  
  
      1  
  
    ),  
  
    asyncTask(  
  
      2  
  
    ),  
  
    asyncTask(  
  
      3  
  
    ),  
  
    asyncTask(  
  
      4  
  
    ),  
  
    asyncTask(  
  
      5
```

),

asyncTask(

6

),

asyncTask(

7

),

asyncTask(

8

),

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

504

asyncTask(

9

),

asyncTask(

10

),

asyncTask(

11

),

asyncTask(

12

),

asyncTask(

13

),

asyncTask(

14

),

asyncTask(

15

),

asyncTask(

16

),

asyncTask(

17

),

asyncTask(

18

),

asyncTask(

19

),

asyncTask(

20

),

];

// sub array of promises of size 5

// 4 sub arrays in total

const

subArrays = chop(promises,

5

);

// to track the indexes of subarrays

const

[index, setIndex] = useState(

0

);

```
// helper function to perform the async operations

const

asyncOperations =

async

(promises) => {

  try

  {

    // execute all the promises of sub-array together

    const

    resp =

    await

    Promise

    .all(promises);

    // print the output of the current sub array

    console

    .log(index, resp);

  }

  catch

  (e) {

    console
```

```
.log(e);  
  
}  
  
finally  
  
{  
  
  // update the index after the operation  
  
  setIndex(index < subArrays.length -  
  
    1  
  
    ? index  
  
    +  
  
    1  
  
    :  
  
    0  
  
  );  
  
}  
  
};  
  
useEffect(  
  
  ()  
  
  => {  
  
    // run first promise after 5 second  
  
    if
```

```
(index ===
```

```
0
```

```
) {
```

```
  setTimeout(
```

```
    ()
```

```
    => {
```

```
      asyncOperations(subArrays[index]);
```

```
    }  
    © JavaScript Interview Guide | learnersbucket.com
```

```
    505
```

```
  },
```

```
    5000
```

```
);
```

```
}
```

```
// and the remaining promises after the previous
```

```
one is done
```

```
else
```

```
{
```

```
  asyncOperations(subArrays[index]);
```

```
}
```

```
}, [index]);
```

```
return
```

```
<></>
```

```
;
```

```
};
```

```
export default Example;
```

Output

```
// after 5 seconds
```

```
0
```

```
(
```

```
5
```

```
) [
```

```
'Completing 1'
```

```
,
```

```
'Completing 2'
```

```
,
```

```
'Completing
```

```
3'
```

```
,
```

```
'Completing 4'
```

```
,
```



'Completing 5'

]

1

(

5

) [

'Completing 6'

,

'Completing 7'

,

'Completing

8'

,

'Completing 9'

,

'Completing 10'

]

2

(

5

) [

'Completing 11'

,

'Completing 12'

,

'Completing

13'

,

'Completing 14'

,

'Completing 15'

]

3

(

5

) [

'Completing 16'

,

'Completing 17'

,

'Completing

18'

,

'Completing 19'

,

'Completing 20'

]

// after 5 seconds

0

(

5

) [

'Completing 1'

,

'Completing 2'

,

'Completing

3'

,

'Completing 4'

,

'Completing 5'

]

1

(

5

) [

'Completing 6'

,

'Completing 7'

,

'Completing

8'

,

'Completing 9'

,

'Completing 10'

]

2

(

5

) [

'Completing 11'

,

'Completing 12'

,

'Completing

13'

,

'Completing 14'

,

'Completing 15'

]

3

(

5

) [

'Completing 16'

,

'Completing 17'

,

'Completing

18'

,

'Completing 19'

,

'Completing 20'

]

© JavaScript Interview Guide | learnersbucket.com

506

Time in human readable format

Problem Statement -

Create a component in React that takes time as input and returns a

string representing the time in the human-readable format like “just now”, “a few secs ago”, “a minute ago”, “10 mins ago”, etc.

Example

Input:

<FormattedTime time=

{new

Date("Sun Nov 20 2022 14:20:59")}

/>

Output:

3 hours ago

This component requires calculating the difference between two times efficiently and then showing the appropriate message.

Get a time as input and get the difference between it with the current time in seconds.

```
// get the current time in milliseconds
```

```
const
```

```
current = +
```

```
Date
```

```
.now();
```

```
// get the date in milliseconds
```

```
const
```

```
lastTime = +lastDate;
```

```
// get the difference in milliseconds
```

```
let
```

```
diff =
```

```
Math
```

```
.abs(current - lastTime);
```

```
// convert the time to seconds
```

```
diff = diff /
```

1000

;

Create an enum of times in seconds and the messages.

// messages

© JavaScript Interview Guide | learnersbucket.com

507

const

messages = {

NOW:

"just now"

,

LESS\_THAN\_A\_MINUTE:

"a few secs ago"

,

LESS\_THAN\_5\_MINUTES:

"a minute ago"

,

MINUTES:

"mins ago"

,



HOURS:

"hours ago"

,

DAYS:

"days ago"

,

MONTHS:

"months ago"

,

YEARS:

"years ago"

,

};

// time in seconds

const

timeInSecond = {

MINUTE:

60

,

HOUR:

60

\*

60

,

DAY:

24

\*

60

\*

60

,

MONTH:

30

\*

24

\*

60

\*

60

,

YEAR:

365

\*

24

\*

60

\*

60

,

};

Check, in which range do times fit and return the message accordingly.

// convert the time to the human-readable format

switch

(diff) {

case

diff <

10

:

return

messages.NOW;

```
case
diff >
10
&& diff < timeInSeconds.MINUTE:
return
messages.LESS_THAN_A_MINUTE;
case
diff > timeInSeconds.MINUTE && diff < timeInSeconds.MINUTE
*
5
:
return
messages.LESS_THAN_5_MINUTES;
default
:
if
(diff < timeInSeconds.HOUR) {
return
`${getFormatted(diff
/ timeInSeconds.MINUTE)}
```

```

    ${messages.MINUTES}`;

    } else if (diff > timeInSeconds.HOUR && diff < timeInSeconds.DAY) {

    return `${getFormatted(diff /

    timeInSeconds.HOUR)}

    ${messages.HOURS}`

    ;

    }

    else

    if

    (diff > timeInSeconds.DAY && diff <

    timeInSeconds.MONTH) {

    return

    `${getFormatted(diff

    / timeInSeconds.DAY)}

    ${messages.DAYS}`;

    } else if (diff > timeInSeconds.MONTH && diff < timeInSeconds.YEAR) {

    return `${getFormatted(diff /

    timeInSeconds.MONTH)}

```

```
    ${messages.MONTHS}`  
    ;  
  }  
  else  
  if  
  (diff > timeInSeconds.YEAR) {  
  return  
  `${getFormatted(diff  
  / timeInSeconds.YEAR)}`  
  ${messages.YEARS}`;  
  }  
}
```

Encapsulate this logic inside a function that will format and return the value.

```
// messages
```

```
const
```

```
messages = {
```

```
  NOW:
```

```
    "just now"
```

```
,
```

```
  LESS_THAN_A_MINUTE:
```

"a few secs ago"

,

LESS\_THAN\_5\_MINUTES:

"a minute ago"

,

MINUTES:

"mins ago"

,

HOURS:

"hours ago"

,

DAYS:

"days ago"

,

MONTHS:

"months ago"

,

YEARS:

"years ago"

,

```
};
```

```
// time in seconds
```

```
const
```

```
timeInSeconds = {
```

```
  MINUTE:
```

```
    60
```

```
,
```

```
  HOUR:
```

```
    60
```

```
    *
```

```
    60
```

```
,
```

```
  DAY:
```

```
    24
```

```
    *
```

```
    60
```

```
    *
```

```
    60
```

```
,
```

```
  MONTH:
```



30

\*

24

\*

60

\*

60

,

YEAR:

365

\*

24

\*

60

\*

60

,

};

// get the floor value

const

```
getFormatted = (  
  time  
) => {  
  return  
    Math  
      .floor(time);  
};  
  
// helper function to calculate  
const  
calculate = (  
  lastDate  
) => {  
  
  // get the current time in milliseconds  
  const  
  current = +  
    Date  
      .now();  
  
  © JavaScript Interview Guide | learnersbucket.com  
  
  509  
  
  // get the date in milliseconds
```

```
const
lastTime = +lastDate;

// get the difference in milliseconds

let
diff =

Math

.abs(current - lastTime);

// convert the time to seconds

diff = diff /

1000

;

// convert the time to the human-readable format

switch
(diff) {

case
diff <

10

:

return
messages.NOW;
```

```
case
diff >
10
&& diff < timeInSeconds.MINUTE:
return
messages.LESS_THAN_A_MINUTE;
case
diff > timeInSeconds.MINUTE && diff < timeInSeconds.MINUTE
*
5
:
return
messages.LESS_THAN_5_MINUTES;
default
:
if
(diff < timeInSeconds.HOUR) {
return
`${getFormatted(diff
/ timeInSeconds.MINUTE)}
```

```
    ${messages.MINUTES}`;  
    } else if (diff > timeInSeconds.HOUR && diff < timeInSeconds.DAY) {  
    return `${getFormatted(diff /  
    timeInSeconds.HOUR)}  
    ${messages.HOURS}`  
    ;  
    }  
    else  
    if  
    (diff > timeInSeconds.DAY && diff <  
    timeInSeconds.MONTH) {  
    return  
    `${getFormatted(diff  
    / timeInSeconds.DAY)}  
    ${messages.DAYS}`;  
    } else if (diff > timeInSeconds.MONTH && diff < timeInSeconds.YEAR) {  
    return `${getFormatted(diff /  
    timeInSeconds.MONTH)}  
    ${messages.MONTHS}`  
    ;
```

```

    }

    else

    if

    (diff > timeInSeconds.YEAR) {

    return

    `${getFormatted(diff

    / timeInSeconds.YEAR)}

    ${messages.YEARS}`;

    }

    }

    };

    const FormattedTime = ({ time }) => {

    // calculate the time

    const convertedTime = calculate(time);

    return <p>{convertedTime}</

    p>;

    };

```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

510

```
export default FormattedTime;
```

Test Case

Input:

```
<FormattedTime time=  
{new  
Date("Sun Nov 20 2022 14:20:59")}  
>
```

Output:

3 hours ago

© JavaScript Interview Guide | learnersbucket.com

511

Detect overlapping circles

Problem Statement -

Draw circles on the screen on the click and whenever two circles overlap change the color of the second circle.

- When a user clicks anywhere on the DOM, create a circle around it of a radius of 100px with a red background.
- If two or more circles overlap, change the background of the later circle.

Let us understand the logic of creating the circle first.

As we have to create the circle with a radius of 100px (200px diameter), rather than generating the circle on the click, we will store the coordinates of the position where the circle should be generated when the user clicks and then create circles out of these coordinates.

As all the circles will be in absolute position so that they can be freely placed on the screen, we will calculate the top , bottom , left , and right positions that will help in placement as well as detecting if two circles are colliding.

Get the [clientX](#) and [clientY](#) coordinates when the user clicks and align the circle around with a simple calculation so that it is placed in the center. Also before updating the state check if the current circle is overlapping with the existing circles then update the background color of the current.

```
// helper function to gather configuration when user clicks
```

```
const
```

```
draw = (
```

```
e
```

```
) => {
```

```
// get the coordinates where user has clicked
```

```
const
```

```
{ clientX, clientY } = e;
```

```
// decide the position where circle will be created
```

```
and placed
```

```
© JavaScript Interview Guide | learnersbucket.com
```

```
512
```

```
// as the circle is of 100 radius (200 diameter), we are subtracting the values
```

```
// so that circle is placed in the center
```

```
// set the initial background color to red
```



```
setElementsCoordinates((
```

```
  prevState
```

```
) => {
```

```
  const
```

```
  current = {
```

```
    top: clientY -
```

```
    100
```

```
    ,
```

```
    left: clientX -
```

```
    100
```

```
    ,
```

```
    right: clientX -
```

```
    100
```

```
    +
```

```
    200
```

```
    ,
```

```
    bottom: clientY -
```

```
    100
```

```
    +
```

```
    200
```

```
,  
background:  
"red"  
  
,  
};  
  
// before making the new entry  
  
// check with the existing circles  
  
for  
  
(  
  
let  
  
i =  
  
0  
  
; i < prevState.length; i++) {  
  
// if the current circle is colliding with any  
  
existing  
  
// update the background color of the current  
  
if  
  
(elementsOverlap(current, prevState[i]))  
  
{  
  
current.background = getRandomColor();
```

```
break
```

```
;
```

```
}
```

```
}
```

```
return
```

```
[...prevState, current];
```

```
});
```

```
};
```

Assign the event listener and draw the circle on the click.

```
// assign the click event
```

```
useEffect(
```

```
()
```

```
=> {
```

```
document
```

```
.addEventListener(
```

```
"click"
```

```
, draw);
```

```
return
```

```
()
```

```
=> {
```

```
document  
  
.removeEventListener(  
  
"click"  
  
, draw);  
  
};  
  
}, []);
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

513

Helper function to detect collision and generate random colors.

```
// helper function to generate a random color
```

```
const  
  
getRandomColor =  
  
()  
  
=> {  
  
  const  
  
  letters =  
  
  "0123456789ABCDEF"  
  
  ;  
  
  let  
  
  color =
```

```
"#"
;
for
(
let
i =
0
; i <
6
; i++) {
color += letters[
Math
.floor(
Math
.random() *
16
)];
}
return
color;
```

```

};

// helper function to detect if two elements are overlapping

const
elementsOverlap = (
rect1, rect2
) => {
const
collide = !(
rect1.top > rect2.bottom ||
rect1.right < rect2.left ||
rect1.bottom < rect2.top ||
rect1.left > rect2.right
);
return
collide;
};

```

Generate the circles from the coordinates which we have stored after the user has clicked. As the detection is done before making entry into the state, the circles are generated with different colors if they collide.

```

// circle element

const

```

```
Circle = (  
  { top, left, background }  
) => {  
  return  
  (  
    <div  
    style=  
    {{  
    position: "absolute",  
    width: "200px",  
    height: "200px",  
    borderRadius: "50%",  
    opacity: "0.5",  
    background,  
    top,  
    © JavaScript Interview Guide | learnersbucket.com  
    514  
    left,  
    }}  
    ></div>
```

```
);  
};  
  
return  
  
(  
  
<div>  
  
  { /* render each circle */}  
  
  {elementsCoordinates.map((e) => (  
  
    <Circle {...e} key=  
  
      {e.top  
  
        + e.left + e.right}  
  
    />  
  
  ))}  
  
  </div>  
  
);
```

Putting everything together.

```
import  
  
{ useEffect, useState }  
  
from  
  
"react"  
  
;
```



```
// helper function to generate a random color
```

```
const
```

```
getRandomColor =
```

```
()
```

```
=> {
```

```
const
```

```
letters =
```

```
"0123456789ABCDEF"
```

```
;
```

```
let
```

```
color =
```

```
"#"
```

```
;
```

```
for
```

```
(
```

```
let
```

```
i =
```

```
0
```

```
; i <
```

```
6
```

```
; i++) {  
  color += letters[  
    Math  
      .floor(  
        Math  
          .random() *  
            16  
          )];  
    }  
  return  
    color;  
  };  
  
// helper function to detect if two elements are overlapping  
  
const  
elementsOverlap = (  
  rect1, rect2  
) => {  
  const  
  collide = !(  
    rect1.top > rect2.bottom ||
```

```
rect1.right < rect2.left ||  
rect1.bottom < rect2.top ||  
rect1.left > rect2.right  
);  
return  
collide;  
};
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

515

```
const
```

```
Example =
```

```
()
```

```
=> {
```

```
// store the configuration of each circle
```

```
const
```

```
[elementsCoordinates, setElementsCoordinates]
```

```
= useState([]);
```

```
// helper function to gather configuration when
```

```
user clicks
```

```
const
```

```
draw = (  
  e  
) => {  
  // get the coordinates where user has clicked  
  
  const  
  
    { clientX, clientY } = e;  
  
  // decide the position where circle will be created  
  and placed  
  
  // as the circle is of 100 radius (200 diameter),  
  we are subtracting  
  the values  
  
  // so that circle is placed in the center  
  
  // set the initial background color to red  
  
  setElementsCoordinates((  
    prevState  
  ) => {  
    const  
  
    current = {  
      top: clientY -  
      100
```

```
,  
left: clientX -  
100  
  
,  
right: clientX -  
100  
  
+  
200  
  
,  
bottom: clientY -  
100  
  
+  
200  
  
,  
background:  
"red"  
  
,  
};  
  
// before making the new entry  
// check with the existing circles
```

```
for
(
let
i =
0
; i < prevState.length; i++) {
// if the current circle is colliding with
any existing
// update the background color of the current
if
(elementsOverlap(current, prevState[i]))
{
current.background = getRandomColor();
break
;
}
}
return
[...prevState, current];
});
```

```
};  
  
// assign the click event  
useEffect(  
  ()  
  => {  
    document  
      .addEventListener(  
        "click"  
        , draw);  
    return  
      ()  
      => {  
        © JavaScript Interview Guide | learnersbucket.com  
        516  
        document  
          .removeEventListener(  
            "click"  
            , draw);  
      };  
    }, []);
```

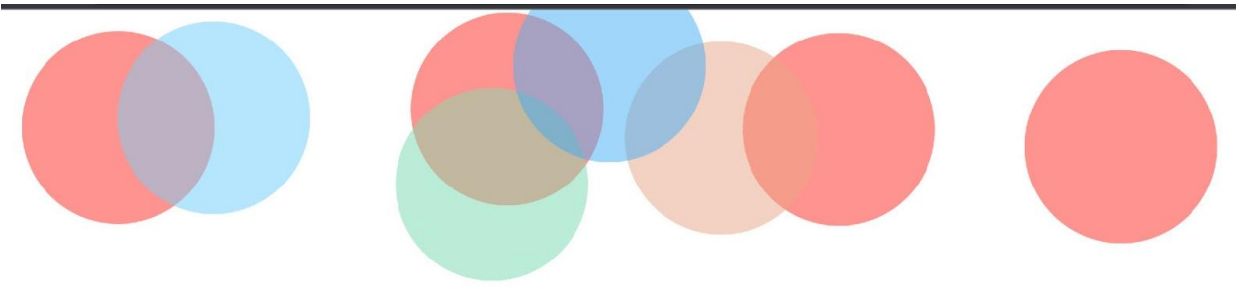
```
// circle element

const
Circle = (
  { top, left, background }
) => {
  return
  (
    <div
      style=
      {{
        position: "absolute",
        width: "200px",
        height: "200px",
        borderRadius: "50%",
        opacity: "0.5",
        background,
        top,
        left,
      }}
    ></div>
  )
}
```



```
);  
};  
  
return  
  
(  
  
<div>  
  
  { /* render each circle */}  
  
  {elementsCoordinates.map((e) => (  
  
    <Circle {...e} key=  
  
      {e.top  
  
        + e.left + e.right}  
  
    />  
  
  ))}  
  
  </div>  
  
);  
  
};  
  
export default Example;
```

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)



Output

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

518

System Design

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

519

Overview

There is no specific answer to a system design question, each answer is subjective.

The best way to answer is to discuss things on points, having a clear understanding of the basics and concepts really helps.

Make a point -> Validate the point -> And convince that the approach you are explaining would be suited to the problem that you are dealing with.

Ask clarifying questions at the beginning, don't assume anything,

make sure you get all the features set listed and then have your

discussion around these points only.

Only explain things that you know thoroughly, unnecessary speaking

will do more harm than good. If you are not certain about anything,

clearly convey that to the interviewers, they may provide hints.

In the next part we will see two system design questions. This book was not meant to cover system design and I am little in-experienced in it. Maybe in the future I will write a book around it too.

These are the two questions I have worked on in my professional life and the same I am discussing next.

© JavaScript Interview Guide | [learnersbucket.com](https://learnersbucket.com)

520

Maker checker flow

Problem Statement -

Imagine a corporate hierarchy where users with higher authority will have privileges over the role with lower authority.

Based on their privileges and role, the features of the applications will be available to the users.

Consider these 4 roles.

Senior manager > manager > software engineer > intern.

In this order, the intern will have the least privilege and the senior manager will have the most privilege.

A higher authority can update the role of the lower authority to any level lower than it, for example, a Senior manager can promote an

intern to the manager directly.

If the lower authority makes any change that requires approval, the

task should be in a pending state and anyone above its role should be able to approve it.

Based on this, design a web application for role-based users.

Discuss the following features

- How do you dynamically show the web application features based on the user's role?.
- How do you handle the privileges and authority?

© JavaScript Interview Guide | learnersbucket.com

521

- How do you extend the application so that the role names can be dynamic?
- Things that you will handle in the frontend and things that you expect from the backend.

How do you dynamically show the web application features based on the user's role.?

There are two levels at which we can add the restrictions.

- *Route level restriction* – where users with a certain role can access certain routes.
- *Feature level restriction* – where users with certain roles and privileges can perform the actions.

For example, an intern can view the employee hierarchy in the web application but cannot perform any actions whereas a senior manager can view and also do certain actions like promoting or demoting.

Thus after discussing with the product managers and getting clarity on the roles and their privileges, we can design a complete hierarchy

chart with the features set and convert this to configuration to

dynamically add restrictions.

Based on the configurations add the restrictions at the route level and the features level. Call this configuration API before login in and cache it.

Once the user is logged in, based on their role, use the configuration details and dynamically render the view.

For route-level restrictions, [create a middleware that will check if the user has the authority to access the route and then allow it .](#)

© JavaScript Interview Guide | learnersbucket.com

522

Feature-level restrictions will answer the second point “How do you handle the privileges and authority.”?

How do you handle the privileges and authority?

This is going to be the most complex part of the application. It requires careful implementation at the component level, module level, and

sub-page level.

For example, The Pending request module will be visible to everyone

except for Interns,

A software engineer can only view the intern’s requests, a manager can view both the software engineer’s and the intern’s requests, and so on.

To make it completely dynamic, we will have to rely on the

configuration and one way we can do it is to create the components for different roles and abstract the logic, this will minimize the chances of error with the multiple conditions and every user role will have its own component level implementation.

Note – “Expect a follow-up question, where you have to create a component to show the working. (Low-level design)”.

How do you extend the application so that the role names can be dynamic?

We can have alias mapping and can use the alias labels for the same level. This way roles and privileges will remain the same but the labels could be different.

© JavaScript Interview Guide | learnersbucket.com

523

Things that you will handle on the front end and things that you expect from the back end.

The session management of the user will rely on the backend and the configuration as well. The configuration will be constructed as a JSON and will be returned from the backend using which the frontend will dynamically render the views.

A token-based authorization mechanism would be the best suited as whenever a user's role is changed, its session/token will be expired, if that user is currently logged in the system, on its next API call, 401 unauthorized will be returned and we can have the API interceptor that will log out the user.

There should be checks to validate the privileges both at the backend and the frontend to be extra sure.

The configuration JSON should be the single source of truth for both the parts to minimize the chances of error.

© JavaScript Interview Guide | learnersbucket.com

524

Design an online judge

Problem Statement -

Design an online judge like Leetcode. The web applications should have the following sections

- Problem listing with an option to sort.
- Problem solving with the solutions.
- Discussion section to discuss interview questions & experience, seek career guidance, etc with an option to filter.

Keeping these in mind, design a coding challenge web app for an authenticated user. Please be mindful of the performance of the application.

Let us tackle each point individually and then at the end we will discuss the overall application performance.

Problem listing

The problem listing page will have all the metadata related to the problem like how many users have tried to solve the problem, what was the acceptance percentage, difficulty, and upvoting.

We can list the problems in a tabular format and display the important details like difficulties and provide an option to the user to filter the problem based on the metadata.

They can also search the problem using a search bar with the fuzzy

search in the problem title and its description. All other options will be available for filter thus it would not be necessary to search on that.

© JavaScript Interview Guide | learnersbucket.com

525

There would also be an option to show which problem you have successfully solved so that you can filter them out.

Because a user won't solve all the problems at once, we can add

pagination to show only a specific number of problems at a time on the page and for the rest, the user will have to navigate, this will be a good performance measure.

Note – “Expect a follow-up question where you are asked to

implement the problem listing page with search and filter options. The filters can be dynamic or static, aka [DATA-TABLE](#) “.

Problem solving with solution

There is no point in repeating the wheel, thus we can use an

[open-source online judge](#) or paid one depending upon the requirements for the core part of executing the coding and providing the result.

This page will be a two section layout, where in one part questions

along with sample test cases will be described and in the other section, the IDE will be integrated.



The question section can also have the solutions and the past results tabs in it.

There will be a pagination option to navigate to the next and previous questions.

© JavaScript Interview Guide | learnersbucket.com

526

Discussion section to discuss interview questions, and experiences, seek career guidance, etc with an option to filter.

This could be a multi-tab public forum, where registered users can ask and answer queries posted by them. There will be an option to tag the questions while asking and then filter based on these tags or using the search.

For posting the question we can have a WYSIWYG editor like

[CKEditor](#) as the user may want to post the code samples.

For answering, we can have one level reply for simplicity. Where only a reply to the question will be given rather than another user's reply.

Note – “Expect a follow-up question to code a simple question-and-answer web app”.

Architecture

[Micro-frontend architecture](#) could be really helpful in boosting the overall performance of this application.

Each page/module of this application can be developed independently by a different team in any framework of choice and they can be bundled and hosted through the CDN individually.

We can then load each of them in the main application and ship it.

Each bundle will be lazy-loaded.

This way the code will be split and being small bundles they will be faster to load. The state between each can be shared using the local storage.

© JavaScript Interview Guide | learnersbucket.com

527

As SEO would be an important factor, we can use the SSR frameworks using a [hybrid rendering technique](#), where SEO-related stuff will be

rendered on the server side and remain on the client side.

© JavaScript Interview Guide | learnersbucket.com

528

[OceanofPDF.com](#)