An optimised bundle of "Solved" Interview Questions.

crack with

Interview .js

400+ pgs.

Arun M

Copyrights

Author - Arun M

Publishing Entity - Published by Arun M (Self)

Email - arun496.in@gmail.com

Legal Disclaimer

This ebook is provided for informational purposes only and does not constitute legal, financial, or professional advice. While every effort has been made to ensure the accuracy and completeness of the information provided, the author makes no warranties, express or implied, regarding the content's accuracy, adequacy, reliability, or suitability for any particular purpose.

Readers are advised to seek appropriate professional advice or conduct their own research before making any decisions based on the information contained herein. The author shall not be liable for any damages, losses, or liabilities arising from the use of or reliance on this ebook or its content.

The problems list solved and solutions added in this ebook are personally researched and referred from various online sources. I don’t hold any ownership or rights on them. 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.

Table of Content

27. Implement Custom String Tokenizer - Medium
28. Implement Custom setTimeout - Medium
29. Implement Custom setInterval - Medium
30. Implement Custom clearAllTimers - Easy
31. Implement Custom Event Emitter - Medium
32. Implement Custom Browser History - Medium
33. Implement Custom lodash _.chunk() - Medium
34. Implement Custom Deep Clone - Medium
35. Promisify the Async Callbacks - Easy
36. Implement 'N' async tasks in Series - Hard
37. Implement 'N' async tasks in Parallel - Medium
38. Implement 'N' async tasks in Race - Easy
39. Implement Custom Object.is() method - Easy
40. Implement Custom lodash _.partial() - Medium
41. Implement Custom lodash _.once() - Medium
42. Implement Custom trim() operation - Medium
43. Implement Custom reduce() method - Medium
44. Implement Custom lodash _.memoize() - Medium
45. Implement Custom memoizeLast() method - Medium
46. Implement Custom call() method - Medium
47. Implement Custom apply() method - Medium
48. Implement Custom bind() method - Medium
49. Implement Custom React "classnames" library - Medium
50. Implement Custom Redux used "Immer" library - Medium
51. Implement Custom Virtual DOM - I (Serialize) - Hard
52. Implement Custom Virtual DOM - II (Deserialize) - Medium
53. Implement Memoize/Cache identical API calls - Hard

Implement Debounce

Problem Statement

Given a function `fn` and a time `t` in milliseconds. You need to return a debounced version of that function.

Concept

First, let's try to understand what debounce means. In technical terms, Debouncing is a programming technique that helps to improve the performance of web applications by limiting the frequency of function calls. (think of this as something like a rate limiter)

In general, Debouncing is a way of delaying the execution of a function until a certain amount of time has passed since the last time it was called. This can be useful for scenarios where we want to avoid unnecessary or repeated function calls that might be expensive or time-consuming.

Example

Imagine we have a search box that shows suggestions as the user types. If we call a function to fetch suggestions on every keystroke, we might end up making too many requests to the server, which can slow down the application and waste

resources. Instead, we can use debouncing to wait until the user has stopped typing for a while before making the request.

Implementation

Let's break it down. As per the question, you will take in a mainFunction and a delay and return the debounced version of that function. Something like this,

```
const debounce = (mainFunction, delay) => {
   return function (...args) {
       // Implementation for the debounce logic goes in here
   }
}
```

Below is the full code,

```
const debounce = (mainFunction, delay) => {
   // Declare a variable called 'timer' to store the timer ID
   let timerID;

   // Return an anonymous function that takes in any number of
arguments
   return function (...args) {
       // Clear the previous timer to prevent the execution of
'mainFunction'
       clearTimeout(timerID);

       // Set a new timer that will execute 'mainFunction' after
the specified delay
       timerID = setTimeout(() => {
           mainFunction(...args);
```

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

You can read about the clearTimeout function here.

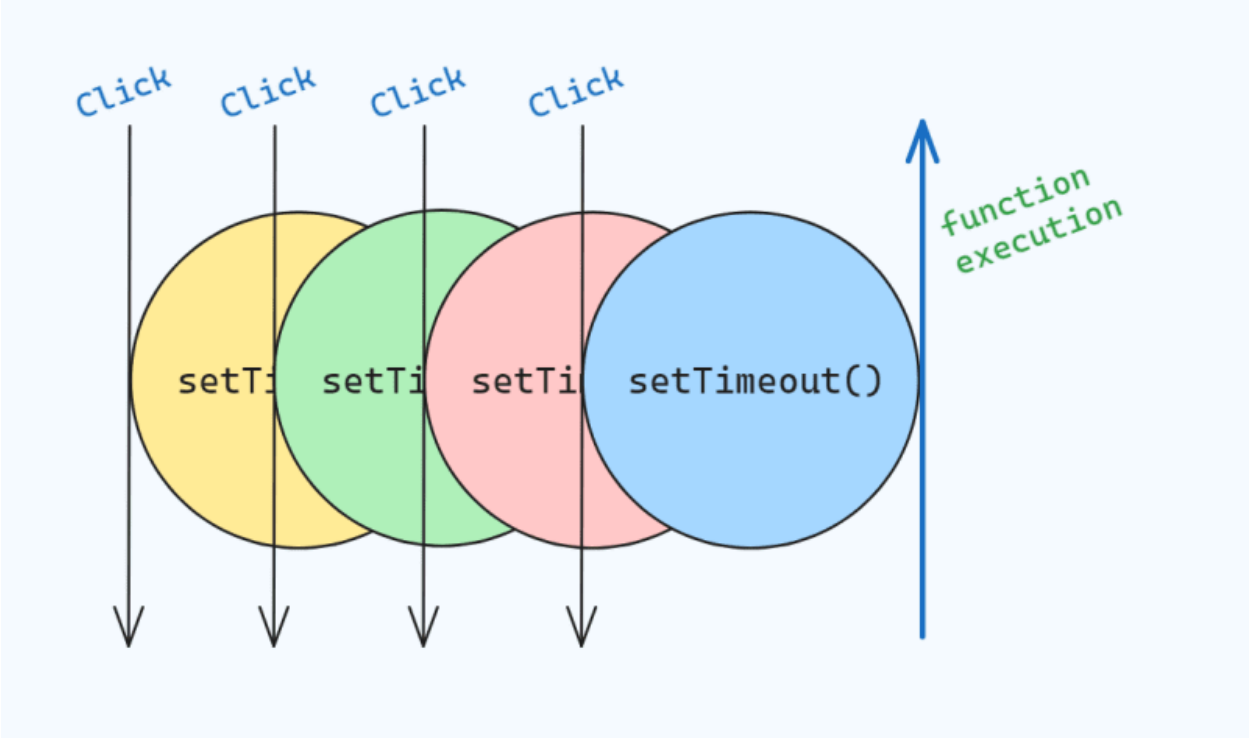
Note: We use `...args` (arguments) with rest operators while calling the mainFunction which simply means we collectively pass down any number of arguments passed while calling the debounced version of the mainFunction. So, as in below code, the mainFunction will be our `fetchData` and `debouncedFn` is the debounced version of it.

Test Case

```
// Initialize startTime. We use this to cross-check debouncedFn
output order
let startTime = Date.now();

// Define a function called 'fetchData' that logs a message to
the console
const fetchData = () => {
   console.log(`fetchData called after ${Date.now() -
startTime}ms`);
}

const debouncedFn = debounce(fetchData, 50);

------------------------------------------------

Input 1:
```

```
setTimeout(debouncedFn, 30);
setTimeout(debouncedFn, 40);

Output 1:

fetchData called at 90ms

-----------------------------------------------

Input 2:

setTimeout(debouncedFn, 30);
setTimeout(debouncedFn, 40);
setTimeout(debouncedFn, 100);
setTimeout(debouncedFn, 160);
setTimeout(debouncedFn, 170);

Output 2:

fetchData called at 90ms
fetchData called at 150ms
fetchData called at 220ms
```

Image Representation of Debounce Working


Click
Click
Click
Click
setTi
setTi
setTi
setTimeout()
function
execution

Implement Throttle

Problem Statement

Given a function `fn` and a time `t` in milliseconds. You need to return a throttled version of that function.

Concept

Let's first understand what's throttling. Throttling is a technique that limits how often a function can be called in a given period of time.

Sounds similar to Debouncing right? But there's a big difference one needs to understand. Throttling is suitable for scenarios where you want to limit how often a function can be called, but you don't want to miss any calls.

It is useful for improving the performance and responsiveness of web pages that have event listeners that trigger heavy or expensive operations, such as animations, scrolling, resizing, mousemove, fetching data, etc.

Example

Let's say you have a function that fetches some data from an API every time the user scrolls the page, you might want to throttle it so that it only makes one request every second,

instead of making hundreds of requests as the user scrolls. This way, we can avoid overloading the server with unnecessary requests.

Implementation

We can achieve throttling in javascript using a timer function such as setTimeout. The timer function can be used to enforce a delay between calls to the throttled function, allowing it to be called only once within the specified time period.

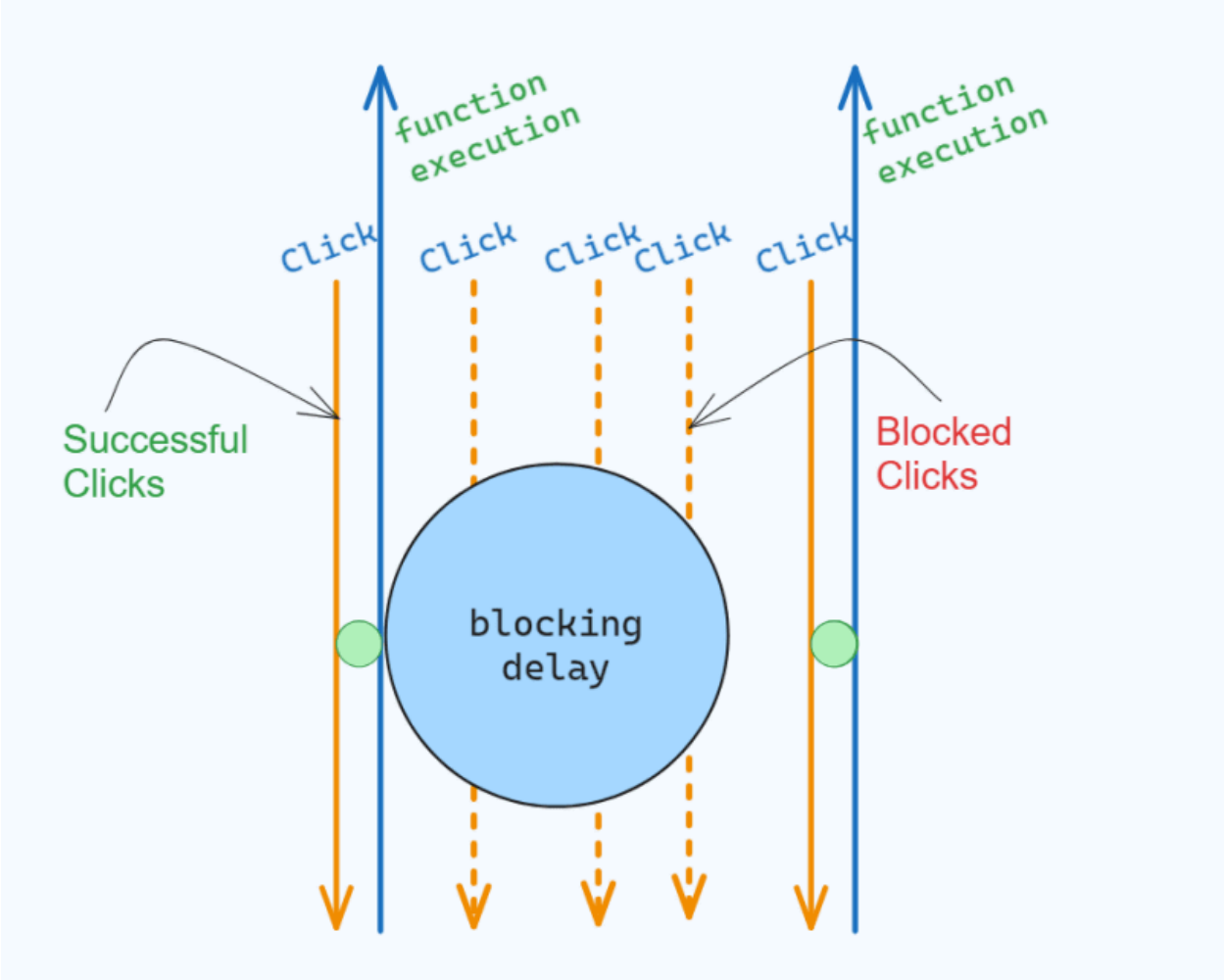
```
const throttle = (mainFunction, delay) => {
   // Initialize a Variable to keep track of the timer
   let timerFlag = null;

   // Returning a throttled version
   return (...args) => {
       // If there is no timer currently running, then execute
the mainFunction
       if (timerFlag === null) {
           mainFunction(...args);

           // Set a timer to clear the timerFlag after the
specified delay
           timerFlag = setTimeout(() => {
               // Clear the timerFlag to allow the main function
to be executed again
               timerFlag = null;
           }, delay);
       }
   }
}
```

Test Case

```
// Initialize prev which tracks last time `throttledFn` was
executed
let prev = Date.now();

// Define a function called 'fetchData' that logs a message to
the console
const fetchData = () => {
   console.log(`fetchData called after ${Date.now() - prev}ms`);
   prev = Date.now();
}

const throttledFn = throttle(fetchData, 50);

// And you will see an output printing logs after a gap of
nearly `delay` ms
document.addEventListener('mousemove', throttledFn);

-----------------------------------------------

Output:

fetchData called after 57ms
fetchData called after 55ms
fetchData called after 52ms
fetchData called after 52ms
fetchData called after 54ms
fetchData called after 53ms
```

Note: In output we don't see logs printed exactly after 50 ms delay as given. This is because your browser also does some computation behind-the-scenes which adds up to the delay.

Image Representation of Throttle Working



Implement Currying

Problem Statement

Implement a `curry()` function, which accepts a function as an input and returns a curried version of that function passed as input.

Concept

In technical terms, currying simply means evaluating functions with multiple arguments and decomposing them into a sequence of functions with some number of arguments. That said, it is a technique which transforms a callable function from f(a, b, c) into sequences of callable functions like f(a)(b)(c) or f(a,b)(c) or f(a)(b,c).

Currying doesn't call a function. It just transforms it.

In simpler terms, currying is when a function — instead of taking all arguments at one time — takes the first one and returns a new function, which takes the second one and returns a new function, which takes the third one, etc. until all arguments are completed.

Example

Below is a small example, comparing the normal function call and curried function call. The output is the same for both, but there's a difference in how they're called for execution.

```
// 1. Normal version
const add = (a, b, c)=>{
    return a+ b + c
}
console.log(add(2, 3, 5)) // 10

// 2. Curried version
const addCurry =(a) => {
    return (b)=>{
        return (c)=>{
            return a+b+c
        }
    }
}
console.log(addCurry(2)(3)(5)) // 10
```

One important doubt many get is, how does the curried version still have access to the values of arguments `a`, `b` even after the functions are returned. It is possible because of closures.

Implementation

But the problem is we don’t know the number of arguments (not fixed) which will be passed to our curried function as it can receive any number of arguments (multiple). So let’s try implementing that,

```
const curry = (mainFn) => {
   return function curried (...args) {
       // mainFn.length tells the total number of arguments it
will receive as in function declaration
```

```
        if (args.length >= mainFn.length) {
            // we wait until all arguments are collected
            return mainFn(...args);
        }
        else {
            // return new func with currently received arguments
            return curried.bind(null, ...args);
        }
    }
}
```

Test Case

```
const totalNum = (a, b, c) => {
    return a + b + c;
}

const curriedTotal = curry(totalNum);

const getLogger = () => {
    let count = 0;
    return (value) => {
        console.log(`Output for Input ${count++} is`, value);
    }
}

const log = getLogger();

-----------------------------------------------

Input:
```

```
log(curriedTotal(10)(20)(30));
log(curriedTotal(10, 20)(30));
log(curriedTotal(10)(20, 30));
log(curriedTotal(10, 20));
log(curriedTotal(10)(20, 30, 40, 50));
log(curriedTotal(10)(20, 30)(40));

Output:

Output for Input 0 is 60
Output for Input 1 is 60
Output for Input 2 is 60
Output for Input 3 is [Function: bound curried]
Output for Input 4 is 60
Error - Not a function
```

Use Case

Why currying? Is it even used?
Let's understand this with a real-life example. For example, you have a logging function `log(importance, date, message)`.

The `log` function takes in 3 arguments where,

1. `importance` can be SEVERE, WARN, ERROR, INFO.
2. `date` is the real time date and time format when we call.
3. `message` will be some relatable message we want to pass based on the log importance.

In real-world projects we use this log function to send such information to the server for debugging by making an HTTP request.

One thing to notice is that the `importance` argument has some predefined standard value formats which won't vary. Since it’s used to signify the type of a log. Keeping this in mind, below is the code on how we can write in a normal version and a curried version (which is more readable and maintainable).

```
const log = (importance, date, message) => {
    console.log(`(${importance} LOG on Day${date.getUTCDay()})
-- ${message}`);
}

-----------------------------------------------

// Normal Version - As you can see we're passing the same
argument value `SEVERE` for `importance` again and again

log('SEVERE', new Date(), 'This is severe');
log('INFO', new Date(), 'Information of code flow');
log('SEVERE', new Date(), 'Found a severe bug');
log('ERROR', new Date(), 'Error - received null');
log('SEVERE', new Date(), 'This is severe');
log('WARN', new Date(), 'This is a warning');

-----------------------------------------------

// Curried Version - Both readable and maintainable

const curryLog = (mainFn) => {
   return function curried (...args) {
       if (args.length >= mainFn.length) {
           return mainFn(...args);
       }
```

```
        else {
            return curried.bind(null, ...args);
        }
    }
}
const curriedLog = curryLog(log);

const severeLog = curriedLog('SEVERE');
const warnLog = curriedLog('WARN');
const errorLog = curriedLog('ERROR');

severeLog(new Date(), 'This is severe');
severeLog(new Date(), 'A new severe bug');
warnLog(new Date(), 'This is a warning');
```

Likewise, we can have many such use-cases and make our code modular, readable, and maintainable.

Implement Currying with Placeholders

Problem Statement

Implement a `curry()` function, which also supports placeholders. So your curried function call can also be passed placeholders as arguments along with values. Our main aim is to replace those arguments which are placeholders with values.

Note: A placeholder is some character which will be given as input. For simplicity, we'll take underscore "_" as a placeholder.

Example

```
// Function which needs to be `curried`. Hereafter, I'll refer
to it as `mainFn` for simplicity.

// You need at least mainFn.length of valid values after
replacing the placeholders.

// In this case `mainFn` requires at least 3 valid values and
also these values should not be placeholders (only then we can
execute `mainFn`)

const join = (a, b, c) => {
   return `${a}_${b}_${c}`
}
// Curried version of the `mainFn`
```

```
const curriedJoin = curry(join);

// "_" underscore is the Placeholder we assume here to solve the
problem

// Sample Test Cases:

const _ = '_'; // placeholder

curriedJoin(1, 2, 3)
// '1_2_3'

curriedJoin(_, 2)(1, 3)
// '1_2_3'

curriedJoin(_, _, _)(1)(_, 3)(2)
// '1_2_3'
```

Implementation

Here, we have 2 requirements or conditions to be satisfied:

1. We have to check the arguments length required for mainFn
2. We have to replace the placeholders "_" for the required arguments

```
function curry(mainFn) {
   return function curried(...args) {
       // 1st condition is we must satisfy the minimum argument
required by `mainFn`
       const hasRequiredArgs = args.length >= mainFn.length;
```

```
        // 2nd condition is that those min. required arguments
should not be placeholders
        const hasAnyPlaceholders = args.slice(0,
mainFn.length).includes(placeholder);

        const doesArgsSatisfy = hasRequiredArgs &&
!hasAnyPlaceholders;

        if (doesArgsSatisfy) {
            // If both conditions satisfy, we execute `mainFn`
            return mainFn(...args);
        }
    }
}
```

Note: We're slicing the `args` from 0 to `mainFn.length`. Since we only need to check arguments required by mainFn.

If all conditions satisfy, we just execute `mainFn`.

Else, we will still need to process, replacing the placeholders until we satisfy all the requirements.

Here's the complete code,

```
const placeholder = "_";

function curry(mainFn) {
    return function curried(...args) {
        const hasRequiredArgs = args.length >= mainFn.length;
```

```
        const hasAnyPlaceholders = args.slice(0,
mainFn.length).includes(placeholder);

        const doesArgsSatisfy = hasRequiredArgs &&
!hasAnyPlaceholders;

        if (doesArgsSatisfy) {
            return mainFn(...args);
        }
        else {
            // We return a function to process next arguments
passed to satisfy the conditions
            return (...nextArgs) => {
                // replace `placeholder` in `args` with the value
in `nextArgs`
                const processedArgs = args.map(arg => {
                    // Note the case where `nextArgs.length` is
less than `args.length`
                    if (arg === placeholder && nextArgs.length >
0) {
                        return nextArgs.shift();
                    }
                    else {
                        return arg;
                    }
                })

                // The main aim here was to replace placeholders
and merge the arguments.
                const mergedArgs = [...processedArgs,
...nextArgs];
                return curried(...mergedArgs);
            }
        }
    }
}
```

Test Case

```
const join = (a, b, c) => {
   return `${a}_${b}_${c}`;
}

const curriedJoin = curry(join);

const _ = '_'; // placeholder

console.log(curriedJoin(_, 2)(1, 3));
// '1_2_3'

console.log(curriedJoin(_, _, 2)(1, 3));
// '1_3_2'

console.log(curriedJoin(_, _, _, 2)(1, 3)(4));
// '1_3_4'

console.log(curriedJoin(_, _, _, _)(_, 2, _)(_, 3)(1));
// '1_2_3'

console.log(curriedJoin(_, _, 3, 4)(1, _)(2, 5));
// '1_2_3'

console.log(curriedJoin(_, _, 2)(_, 3)(_, 4)(_, _, 5)(6));
// '6_3_2'
```

Deep Flatten I

Objective

You need to flatten a deeply nested array. The original array given as input contains either nested arrays or primitive values.

Problem Statement

Implement a `flatten()` function which will flatten the original array into a new array to the `depth` of arr given in the question. `depth` is any value up to which you need to flatten the deeply nested arrays in the original array.

Note: This problem is similar to the Array.prototype.flat() method. Where now we need to build our own version of `flat()`

Example

```
// Original arr which needs to be flattened up to the given
`depth`
// `depth` is passed as second argument
// if no `depth` passed as argument, by default `depth` is 1.

const arr = [1, [2], [3, [4]]];

flatten(arr, 1);
```

```
// [1, 2, 3, [4]]

flatten(arr, 2);
// [1, 2, 3, 4]
```

The main idea of solving this problem is by collecting and concatenating the sub-arrays in an array into a single array.

So, now the core logic of the problem is below,

```
const result = [];

if (!Array.isArray(elem)) {
   // Not an array
   result.push(elem);
}
else {
   // It's an array
   // So continue flattening it and then push to result
   result.push(...flatten(elem));
}
```

Implementation

Approach 1 (Recursive):

```
function flatten(arr, depth = 1) {
   // We've flattened up to required depth, so we return arr as
it is
   if (depth === 0) {
       return arr;
   }
```

```
    const result = [];

    for (let i = 0; i < arr.length; i++) {
        if (!Array.isArray(arr[i])) {
            result.push(arr[i]);
        }
        else {
            // Flatten curr elem which is an arr before pushing
to result
            // Also, reduce `depth-1` since we need to flatten
arr[i] which is one level more deep
            const flattenedArr = flatten(arr[i], depth - 1);
            result.push(...flattenedArr);
        }
    }

    return result;
}
```

Approach 2 (Iterative):

In order, to solve the problem iteratively we need a similar behavior as in recursion. That is, in recursion, internally the `Stack` Data Structure helped to keep track of sub problems and solve it.

Exactly, recursion internally uses Stack. For our iterative solution, we'll explicitly make use of `Stack` Data Structure to simulate the recursive behavior and achieve to solve the problem.

If you carefully look at the Iterative solution, you'll see each step exactly as in our recursive solution. I have also highlighted the solution which will help you better relate.

```
function flatten(arr, depth = 1) {
   const result = [];
   // We use below arr to operate as Stack Data Structure
   const stack = [];

   // In recursive approach, we had track of `depth` because we
pass it along each function call
   // Similarly, for iterative, we need to bind the current elem
and it's `depth` together to keep track
   // [curr, depth] is exactly similar to our recursive call
flatten(curr, depth)
   const newArr = arr.map(curr => [curr, depth]);
   stack.push(...newArr);

   while (stack.length > 0) {
       const top = stack.pop();
       const [curr, depth] = top;

       // This was our base case
       if (depth === 0) {
           result.push(curr);
           continue;
       }

       if (!Array.isArray(curr)) {
           result.push(curr);
       }
       else {
           // This was similar to our recursive call where we
continue processing to flatten the arr
           const newArr = curr.map(elem => [elem, depth-1]);
```

```
            stack.push(...newArr);
        }
    }
    // Flip the `result` arr
    return result.reverse();
}
```

Note: The result is reversed while returning. We're doing this since we're explicitly using Stack (FIFO), using (push/pop) methods our result arr would be stored in reversed fashion. To maintain correct order of result we could have made use of (shift/unshift) methods, but the Time Complexity would be higher since in that case we'd be removing and adding elements at the front of the arr which is a bit costly operation.

Test Case

```
const arr1 = [1, [2], [3, [4]]];
const arr2 = [1, 2, [3, 4, [5, 6]]];

console.log(flatten(arr1, 1));
// [1, 2, 3, [4]]

console.log(flatten(arr1, 2));
// [1, 2, 3, 4]

console.log(flatten(arr1, 2));
// [1, 2, 3, 4]

console.log(flatten(arr2, 2));
// [1, 2, 3, 4, 5, 6]
```

```
console.log(flatten(arr2));
// [1, 2, 3, 4, [5, 6]]
```

Deep Flatten II

Objective

You need to flatten a deeply nested Object. The original Object given as input contains either nested Objects or primitive values.

Problem Statement

Implement a `flatten()` function which will flatten the original Object into a new Object.

Example

```
const obj1 = {
   a: 1,
   b: 2,
   c: {
       d: 3,
       e: 4
   }
};

const obj2 = {
   a: 1,
   b: 2,
   c: {
       d: 3,
       e: 4,
       f: {
```

```
            g: 5
        },
        h: null
    },
    j: 'Hi'
};

console.log(flatten(obj1));
// { a: 1, b: 2, d: 3, e: 4 }

console.log(flatten(obj2));
// { a: 1, b: 2, d: 3, e: 4, g: 5, h: null, j: 'Hi' }
```

This problem is very much similar to the previous problem (Deep Flatten I) with just the fact that here we have nested Objects instead of arrays. This can be solved with slight modification to the previous problem (Deep Flatten I).

Also, we're not flattening up to a given `depth` as in the previous question. Although if depth is given in the question, solving it would require the same approach as in the previous problem.

Implementation

The approach is similar, where we iterate through the Object's properties, then flatten those properties and create new properties for the flattened values into a new Object.

Finally we merge the resulting objects with the help of spread operator.

Recursive Approach:

```
function flatten(input) {
   // New object in which flattened values are stored
   let result = {};

   // Iterate over each keys of the 'object'
   for (let key in input) {
       // Note: typeof null is an 'object'
       if (typeof input[key] === 'object' && input[key] !==
null) {
           // merge the flattened properties into result
           result = { ...result, ...flatten(input[key]) };
       }
       else {
           result[key] = input[key];
       }
   }

   return result;
}
```

Deep Flatten III

Objective

You need to flatten a deeply nested Object. The original Object given as input contains either nested Objects or primitive values.

Problem Statement

Implement a `flattenWithPrefix()` function which will flatten the original Object into a new Object. This function will also take `prefix` as an argument which will be a string denoting the representation of keys or properties to access the current value in the deeply nested object.

We use the dots "." notation in the prefix string between keys or properties to denote the representation of accessing a value in a deeply nested object.

Example

```
const obj1 = {
   a: 1,
   b: 2,
   c: {
       d: 3,
       e: 4
   }
```

```
};


const obj2 = {
   a: 1,
   b: 2,
   c: {
       d: 3,
       e: 4,
       f: {
           g: 5
       },
       h: null
   },
   j: 'Hi'
};

console.log(flattenWithPrefix(obj1, ''));
// { a: 1, b: 2, 'c.d': 3, 'c.e': 4 }

console.log(flattenWithPrefix(obj2, ''));
// { a: 1, b: 2, 'c.d': 3, 'c.e': 4, 'c.f.g': 5, 'c.h': null, j:
'Hi' }
```

Implementation

We will be concatenating the keys to the prefix as we keep flattening the deeply nested objects within the original input object to denote the accessing the current value.

If the current property's value is not an object we assign it to the result object (flattened) where prefix string is the new key for that value.

Algorithm

1. Iterate over keys of the object
2. If the child key's value is an object,
   ➔Again call the same function
   ➔Also append it's key to the prefix string
3. Else assign value to the new prefix key

```
function flattenWithPrefix(input, prefix = '') {
   // New object in which flattened values are stored
   let result = {};

   const pre = (prefix.length > 0) ? prefix + '.' : '';

   // Iterate over each keys of the 'object'
   for (let key in input) {
       // Note: typeof null is an 'object'
       if (typeof input[key] === 'object' && input[key] !==
null) {
           // merge the flattened properties into result
           result = { ...result,
...flattenWithPrefix(input[key], pre + key) };
       }
       else {
           // we assign the prefix as the new key to denote on
how to access the current value in the original object
           result[pre + key] = input[key];
       }
   }

   return result;
}
```

Deep Flatten IV

Objective

You need to flatten the given input. The input can be either Primitive values or an Object or an Array.

Problem Statement

Implement a `flatten()` function which flattens the given input, where the input may also contain deeply nested Arrays/Objects/Primitives as well.

Implementation

The main idea is to traverse, collect and concatenate.

This simplest approach is breaking into isolated functions where each function is associated with solving a particular problem.

So, we we'll have 2 utility functions, `flattenArray()` and `flattenObject()`

If you focus, the core logic is dividing between conditions, which is common for both `flattenArray()` and `flattenObject()`

```
if (Array.isArray(elem)) {
   // Handling to Flatten the array
}
else if (typeof elem === 'object' && elem !== null) {
   // Handling to Flatten the Object
}
else {
   // Handling for the primitives
}
```

Here’s the complete code,

```
function flatten(input) {
   // If received input is itself a primitive
   if (typeof input !== 'object' || input === null) {
       return input;
   }

   if (Array.isArray(input)) {
       return flattenArray(input);
   }
   else {
       return flattenObject(input);
   }
}

function flattenArray(input) {
   let flattened = [];

   // Our conditions are divided across 3 categories
   // 1. Arrays, 2. Objects, 3. Primitives
   for (const elem of input) {
       if (Array.isArray(elem)) {
           flattened.push(...flattenArray(elem));
```

```
        }
        else if (typeof elem === 'object' && elem !== null) {
            flattened.push(flattenObject(elem));
        }
        else {
            flattened.push(elem);
        }
    }

    return flattened;
}

function flattenObject(input) {
    let flattened = {};

    // Our conditions are divided across 3 categories
    // 1. Arrays, 2. Objects, 3. Primitives
    for (const key in input) {
        const elem = input[key];

        if (Array.isArray(elem)) {
            // console.log(flattenArray(elem));
            flattened[key] = flattenArray(elem);
        }
        else if (!Array.isArray(elem) && typeof elem === 'object'
&& elem !== null) {
            flattened = { ...flattened, ...flattenObject(elem) };
        }
        else {
            flattened[key] = elem;
        }
    }

    return flattened;
}
```

I suggest don't focus on the order of the function being called and get confused on which function calls which.

Instead look at the scope of the problem each function is involved to solve.

That is,

1. flattenArray() is responsible to flatten the Array given as input.
2. flattenObject() is responsible to flatten the Object given as input.
3. Both functions ultimately are involved to handle a specific logic in cases of array or an object or a primitive value.

Test Case

```
const input1 = {
   a: 1,
   b: 2,
   c: [3, { d: 4, e: { f: null } }],
   h: {
       i: 6,
       j: {},
       k: undefined
   },
   l: 'Hi'
};


const input2 = [
   1,
```

```
    2,
    [3, { d: 4, e: undefined, nestedArr: [[5, [6]], [7]] }],
    {
        f: 8,
        g: null,
        h: {
            i: {}
        }
    }
];

console.log(flatten(input1));
// {
//     a: 1,
//     b: 2,
//     c: [ 3, { d: 4, f: null } ],
//     i: 6,
//     k: undefined,
//     l: 'Hi'
// }

console.log(flatten(input2));
// [
//     1,
//     2,
//     3,
//     { d: 4, e: undefined, nestedArr: [ 5, 6, 7 ] },
//     { f: 8, g: null }
// ]
```

Negative Indexing in Arrays (Proxies)

Problem Statement

Implement a wrapper function `wrap()` which makes negative indexing in Arrays possible.

In other programming languages, Python for instance, we can use negative indexes to access items from the end of an array.

This behavior is not available in JavaScript by default.

Example

```
// In JavaScript

const letters = ['a', 'b', 'c', 'd', 'e'];
letters[0]; // 'a'
letters[2]; // 'c'
letters[-1]; // undefined
letters[-3]; // undefined

// In Python

letters = ['a', 'b', 'c', 'd', 'e'];
letters[0]; // 'a'
letters[2]; // 'c'
letters[-1]; // 'e'
letters[-3]; // 'c'
```

Now, as per our problem statement we need to be able to support negative indexing in Arrays in JavaScript.

Thought Process

One thing is very clear, that we need to have some handling while read/update operations on the array. It could have been easier if we had simple getter and setter methods where we could have overridden the implementation.

But the major problem is, we can only read/update Array using the original way that the language supports - using the square bracket indexing notation (ofcourse, it's common across every language :))

```
const letters = ['a', 'b', 'c', 'd', 'e'];
letters[0]; // read
letters[2] = 'z'; // write
letters[-1]; // read
letters[-4] = 'x'; // write
```

Solution

Yes, the hint is already mentioned in the Problem Statement's title :)

We can make use of the Proxies to support negative indexing and also maintain the original way arrays can be accessed.

A `Proxy` object wraps another object, which can intercept and redefine fundamental operations like reading/writing for that original object.

Any interactions performed on the proxy are by default forwarded to the target, as if we're still interacting directly with the target.

(Proxy is in-built within Javascript)

Concept

Let's first understand how Proxy actually works and how we can benefit from it.

The term `Proxy` in general English means - A person who is given the power or authority to do something (such as to vote) for someone else.

```
// Syntax
const proxy = new Proxy(target, handler);
```

- ➔target - an object which will proxy to the original object (target).
- ➔handler - an object with methods that intercept operations we want to customize. These methods are more formally called "traps". Like,
  - ◆get() trap for reading a property of target

◆set() trap for writing a property into target

For most operations on objects, there's a so-called "internal method" in the specification. For example [[Get]] and [[Set]] which are internal methods to read/write properties. These methods are only used in the specification, we can't call them directly, since they are internal.

Here's a code example to understand,

```
const obj = {
   username: 'JSGuy',
   age: 20,
   password: '******'
}

// Since our `obj` has some confidential data like `password` as
property
// We don't want anyone to perform read operation directly
// So, we need to intercept before the read operation with some
custom handling

const proxyObject = new Proxy(obj, {
   // get trap will intercept the read operation
   get(target, property) {
       // Here we can add our customized handling for read
operation
       if (property === 'password') {
           return 'Sorry, confidential data!!';
       }

       // we can also forward the original read operation
```

```
directly to the object itself
        return target[property];
    }
});

console.log(proxyObject.password); // Sorry, confidential data!!
console.log(proxyObject.username); // JSGuy
console.log(proxyObject['age']); // 20
```

Similarly, we can apply different customization logic for different traps. Here’s a list of pre-defined traps we can make use of based on requirement.

Internal Method	Handler Method
[[GetPrototypeOf]]	**getPrototypeOf**
[[SetPrototypeOf]]	**setPrototypeOf**
[[IsExtensible]]	**isExtensible**
[[PreventExtensions]]	**preventExtensions**
[[GetOwnProperty]]	**getOwnPropertyDescriptor**
[[DefineOwnProperty]]	**defineProperty**
[[HasProperty]]	**has**
[[Get]]	**get**
[[Set]]	**set**
[[Delete]]	**deleteProperty**
[[OwnPropertyKeys]]	**ownKeys**
[[Call]]	**apply**
[[Construct]]	**construct**

Implementation

Here's the complete code,

```
function wrap (arr) {
   return new Proxy(arr, {
       // get trap for read operation
```

```
        get(target, property) {
            let index = Number(property);

            if (index < 0) {
                // add len to convert to positive bounds
                index += target.length;

                return target[index];
            }

            return target[index];
        },

        // set trap for write operation
        set(target, property, value) {
            let index = Number(property);

            if (index < 0) {
                index += target.length;

                // updates on -ve indexes which are out of bounds
                // is not allowed
                if (index < 0) {
                    throw new Error('Index out of bounds');
                }

                target[index] = value;

                return true;
            }

            target[index] = value;

            return true;
        }
    })
```

```
}
```

Note: We return a boolean value from the set() trap, this is because of some of the rules the Proxy needs to follow. Here the returned boolean value denotes whether the write operation was successful or not. So, `true` means write is successful, else write was unsuccessful because of some kind of error.

Test Case

```
let arr = ['a', 'b', 'c', 'd'];

// We re-assign the proxied array returned by wrap()
// Since using both versions of arr can lead to improper
behaviors
arr = wrap(arr);

console.log(arr[0]);
// 'a'

console.log(arr[-1]);
// 'd'

arr[-3] = 'z';
console.log(arr);
// [ 'a', 'z', 'c', 'd' ]

arr[-5] = 'x';
// Error: Index out of bounds

console.log(arr);
```

```
// [ 'a', 'z', 'c', 'd' ]
```

Implement a Pipe Method

Problem Statement

Implement a `pipe()` function which chains some `n` number of functions together to create a new function.

Concept

Let's first understand what we mean by pipe function. Function piping is a concept from Functional Programming that encourages use of pure functions, immutability, and the composition of functions to build more complex functionality.

In general, piping in functional programming allows you to combine multiple functions into a single pipeline. The pipe function takes an initial input, passes it through the first function, takes the output and passes it as the input to the next function, and so on. The result of the last function in the pipeline will be the final output.

Example

```
// These are some of our utility functions, the main concept is
below

const getName = (input) => input.name;
const getUppercaseName = (input) => input.toUpperCase();
```

```
const getFirstName = (input) => input.split(' ')[0];
const getReversedName = (input) =>
input.split('').reverse().join('');

// So you can see below that one function's output is passed as
argument to next function
// Just like a series (or) flow of data in pipes

const name = getName({ name: 'JSGuy Senior' });
const uppercaseName = getUppercaseName(name);
const firstName = getFirstName(uppercaseName);
const reverseName = getReversedName(firstName);

// The above code can also be written something like this
// Nesting or composing of sequence of functions
getReversedName(getFirstName(getUppercaseName(getName({ name:
'JSGuy Senior' }))));

// We have to make this sequence of steps much cleaner with pipe
function. To something like this,
pipe(
   getName,
   getUppercaseName,
   getFirstName,
   getReversedName
);
```

Implementation

The approach to implement these seems very obvious to me - which is a simple for loop and call each function in sequence.

Approach 1 (For Loop):

```
function pipe(...funcs) {
   // Accept list of functions using rest operator
   // Return a new function to accept `initialArgument` required
   return function(initialArgument) {
       // Initially our result will be the initialArgument
itself
       let result = initialArgument;

       // Loop through calling each functions passed in sequence
       for (let fn of funcs) {
           // update result, since output of one func will be
the input for other func
           result = fn(result);
       }

       return result;
   }
}
```

Approach 2 (Using Reduce):

If you carefully observe the operations out in the for-loop, it's exactly similar to the working of our `reduce()` function where we start by processing the `initialArgument` and keep accumulating.

```
function pipe(...funcs) {
   return function(initialArgument) {
       return funcs.reduce((accumulator, fn) => {
           return fn(accumulator);
       }, initialArgument)
   }
}
```

Bonus

Based on our above implementation, we assume all of our functions are synchronous. But, what if we had Async functions to execute.

The below code can now handle both the cases of synchronous and asynchronous.

```
function pipe(...funcs) {
   return function(initialArgument) {
       return funcs.reduce((chain, fn) => {
           return chain.then(result => fn(result));
       }, Promise.resolve(initialArgument))
   }
}
```

Test Case

```
const getName = (input) => input.name;
const getUppercaseName = (input) => input.toUpperCase();
const getUppercaseNameAsync = (input) => {
   return new Promise(resolve => setTimeout(() =>
resolve(input.toUpperCase()), 3000))
}
const getFirstName = (input) => input.split(' ')[0];
const getReversedName = (input) =>
input.split('').reverse().join('');
```

```
const pipeSync = pipe(
   getName,
   getUppercaseName,
   getFirstName,
   getReversedName
);

const pipeAsync = pipe(
   getName,
   getUppercaseNameAsync,
   getFirstName,
   getReversedName
);

pipeSync({ name: 'Sync JSGuy' })
.then(result => console.log(result));
// CNYS

pipeAsync({ name: 'Async JSGuy' })
.then(result => console.log(result));
// CNYSA
```

Implement Auto-retry Promises

Problem Statement

Implement a `fetchWithAutoRetry()` function which will automatically retry to fetch when error occurs, until the maximum retry limit is reached.

In Web applications, we make API calls, which is a very common use case. Now there can be many reasons for an API call to fail. We need to have a fallback when such a scenario occurs.

Thus, when an API call fails, we will auto-retry to fetch and make a new API call again. We will auto-retry until the API succeeds or until the maximum retry limit is reached.

Implementation

How can we handle the failure case? We know that making an API call is always asynchronous and it will return a promise, so when the promise gets rejected, it means there's a failure, and we need to handle the auto-retry logic in the `catch()` block.

```
// `fetchData` is our mocked function which will make API call
function fetchWithAutoRetry(fetchData, retryLimit) {
   // We need to return a promise, since fetching data is always
```

```
asynchronous
   return new Promise((resolve, reject) => {
       // IFEE function expression, since we need to make the
1st API call
       (function retryFetch() {
           // If successful, we resolve with the `data`
           fetchData()
           .then(data => resolve(data))
           .catch((error) => {
               // On error, we can retry the API call until
`retryLimit`
               if (retryLimit-- > 0) {
                   retryFetch();
               }
               else {
                   reject(error);
               }
           })
       })()
   })
}
```

Test Case

```
const fetchData = () => {
   // Mocking the API call using a simple counter logic
   let count = 0;
   return () => {
       if (count++ === 4) {
           return Promise.resolve('Data Successful');
       }
       else {
           return Promise.reject('Data Rejected');
       }
```

```
    }
}

fetchWithAutoRetry(fetchData(), 3)
.then(console.log)
.catch(console.log) // Data Rejected

fetchWithAutoRetry(fetchData(), 5)
.then(console.log) // Data Successful
.catch(console.log)
```

Implement Promise.all

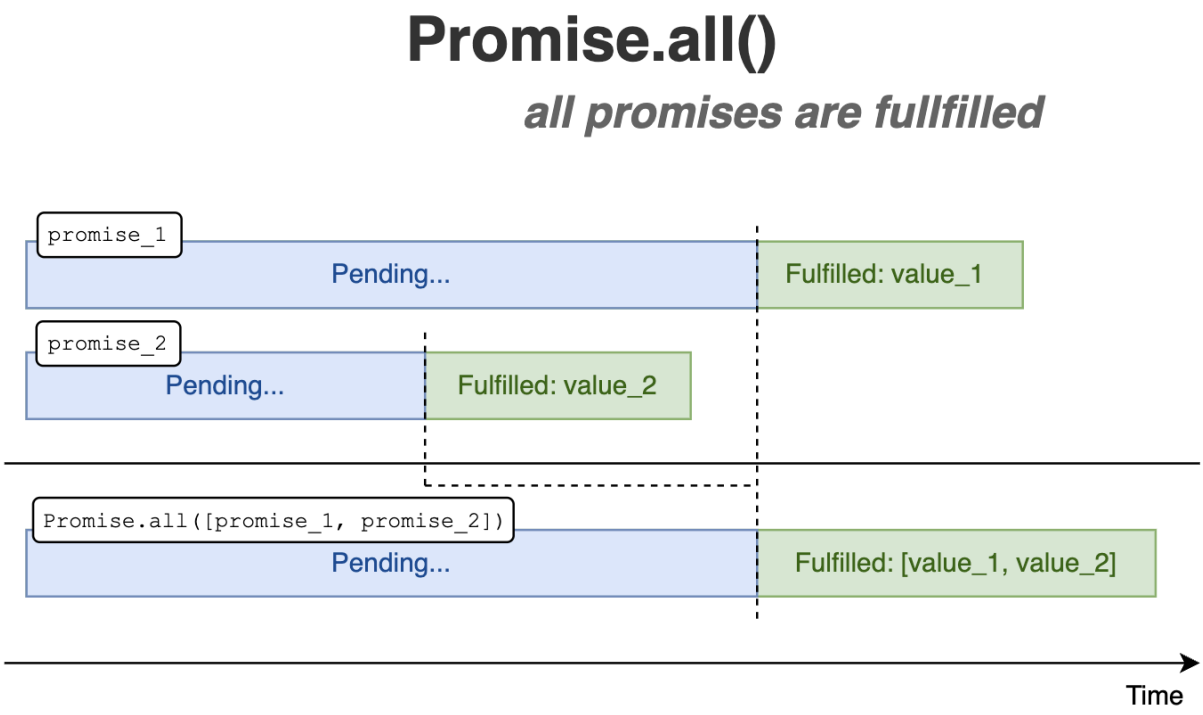
Problem Statement

Implement a polyfill of `Promise.all()` function. And you should not use the built-in function directly for the problem, instead write your own version.

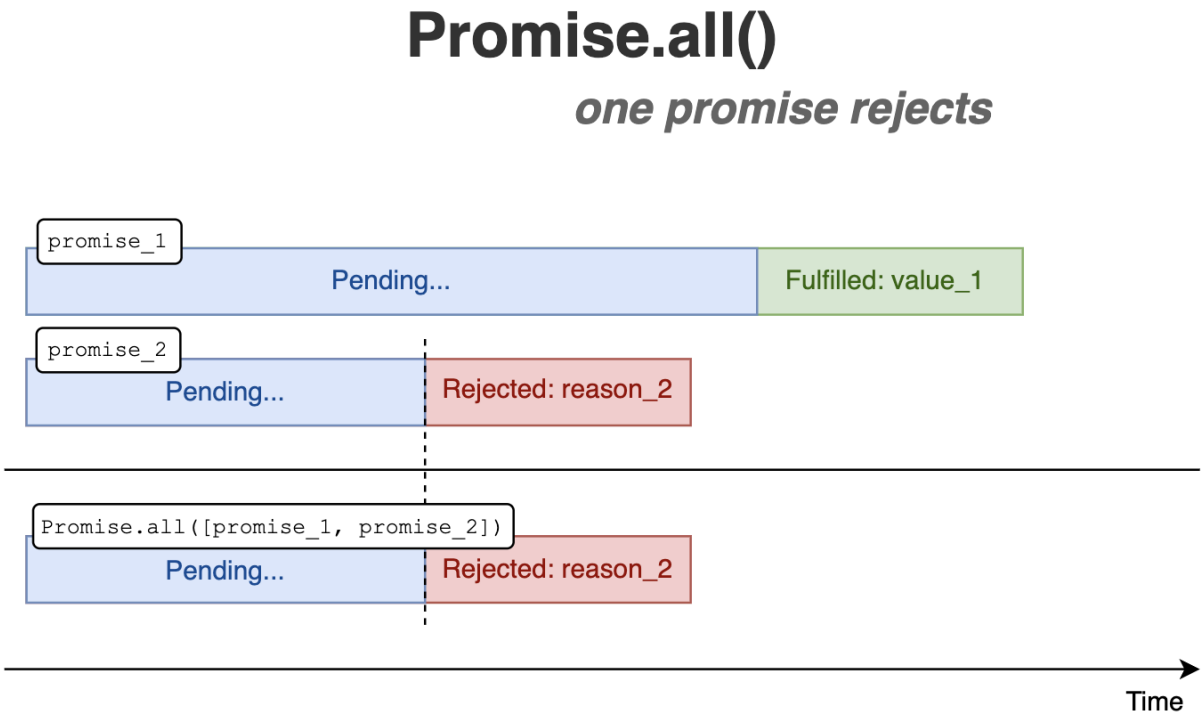
`Promise.all()` is a helper function provided by javascript to handle multiple promises at once, in parallel, and get the results in a single aggregated array.

Visual Examples

Case 1: When all promises are resolved or fulfilled.



Case 2: When either of the promises is rejected.



Example

```
// Syntax
const resultPromises = Promise.all([promise1, promise2]);

resultPromises.then((value) => {
   console.log('Resolved with', value);
})
.catch((error) => {
   console.log('Rejected with', error);
})
```

Implementation

`Promise.all()` accepts an array of promises as input and returns a Promise object.

1. The returned promise will be resolved when all the promises in the input array are resolved.
2. If any of the promises in the input array is rejected, the returned promise will be rejected with the reason for the first rejected promise.

Note:

1. Promise.all maintains the order of the results from the promises provided as input. So, the result of a promise needs to be mapped in the output array exactly to the index to that of the promise in the input array.

2. The input array passed to the function can contain both promises and non-promise values as well (like string, object, number..)

```
Promise.all = function(input) {
   // result arr to store the results of each promise
   const result = [];

   // counter to keep track if all the promises are resolved
   let totalResolved = 0;

   return new Promise((resolve, reject) => {
       // If empty arr, we can immediately resolve with []
       if (input.length === 0) {
           return resolve(result);
       }

       input.forEach((elem, index) => {
           // We need to wrap each `elem` in Promise.resolve(),
           // since `elem` can be any value other than a Promise
as well
           Promise.resolve(elem)
           .then((value) => {
               result[index] = value;
               totalResolved++;

               if (totalResolved === input.length) {
                   resolve(result);
               }
           })
           .catch((error) => reject(error))
       })
   })
}
```

Note:
A Promise can only be either resolved/rejected only once. So as per our solution, it seems if all the promises passed as input gets rejected, then the main wrapper promise will be rejected multiple times, but that's not gonna be the case, as we know the above fact now.

Test Case

```
const promise1 = Promise.all([
   Promise.resolve(1),
   new Promise((resolve) => setTimeout(() => resolve(2), 2000)),
   Promise.resolve(3),
   4
])

promise1
.then(value => console.log('Resolved with 1', value))
.catch(value => console.log('Rejected with 1', value))
// Resolved with [ 1, 2, 3, 4 ]

-------------------------------------------

const promise2 = Promise.all([
   new Promise((resolve) => setTimeout(() => resolve(3), 2000)),
   Promise.resolve(2),
   '4',
   new Promise((resolve) => setTimeout(() => resolve(1), 0)),
   Promise.resolve(5),
])

promise2
.then(value => console.log('Resolved with 2', value))
```

```
.catch(value => console.log('Rejected with 2', value))
// Resolved with [ 3, 2, '4', 1, 5 ]

-----------------------------------------------

const promise3 = Promise.all([
   Promise.resolve(1),
   new Promise((resolve, reject) => setTimeout(() => reject(2),
0)),
   new Promise((resolve) => setTimeout(() => resolve(3), 2000)),
   Promise.reject(4),
   Promise.reject(5)
])

promise3
.then(value => console.log('Resolved with 3', value))
.catch(value => console.log('Rejected with 3', value))
// Rejected with 4

-----------------------------------------------

const promise4 = Promise.all([
   null,
   undefined,
   new Promise((resolve) => setTimeout(() => resolve(2), 350)),
   {},
   'Hello'
])

promise4
.then(value => console.log('Resolved with 4', value))
.catch(value => console.log('Rejected with 4', value))
// Resolved with [ null, undefined, 2, {}, 'Hello' ]

-----------------------------------------------
```

```
const promise5 = Promise.all([])

promise5
.then(value => console.log('Resolved with 5', value))
.catch(value => console.log('Rejected with 5', value))
// Resolved with []
```

Implement Promise.allSettled

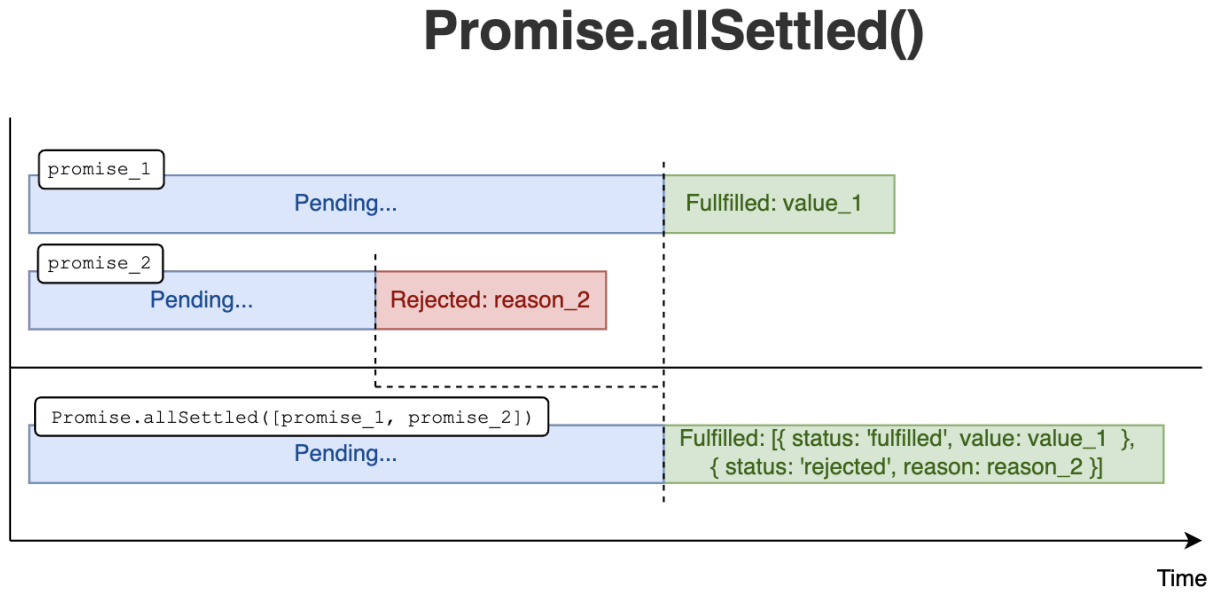
Problem Statement

Implement a polyfill of `Promise.allSettled()` function. And you should not use the built-in function directly for the problem, instead write your own version.

`Promise.allSettled()` is a helper function that runs multiple promises in parallel and aggregates the settled statuses (either fulfilled or rejected) into a result array.

Note: When a promise is settled it means it's either resolved or rejected.

Visual Examples



Example

```
// Syntax
const settledPromises = Promise.allSettled([promise1,
promise2]);

settledPromises.then((value) => {
   console.log('Resolved with', value);
})
.catch((error) => {
   console.log('Rejected with', error);
})
```

Implementation

`Promise.allSettled()` takes an array of promises as input and returns a promise object.

1. The returned promise will be resolved when all the promises in the input array are settled.
2. The returned promise will be resolved with an array of objects that each describes the status of each promise (either fulfilled or rejected) in the input array.

```
Promise.allSettled = function(input) {
   const result = [];

   // counter to keep track if all the promises are settled
   let totalSettled = 0;

   return new Promise((resolve, reject) => {
```

```
        if (input.length === 0) {
            return resolve(result);
        }

        input.forEach((elem, index) => {
            Promise.resolve(elem)
            .then((value) => {
                result[index] = { status: 'fulfilled', value };
                totalSettled++;

                if (totalSettled === input.length) {
                    resolve(result);
                }
            })
            .catch((reason) => {
                result[index] = { status: 'rejected', reason };
                totalSettled++;

                if (totalSettled === input.length) {
                    resolve(result);
                }
            })
        })
    })
}
```

Test Case

```
const promise1 = Promise.allSettled([
   Promise.resolve(1),
   new Promise((resolve) => setTimeout(() => resolve(2), 2000)),
   Promise.resolve(3),
   Promise.reject(4)
])
```

```
promise1
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Resolved with
// [
//     { status: 'fulfilled', value: 1 },
//     { status: 'fulfilled', value: 2 },
//     { status: 'fulfilled', value: 3 },
//     { status: 'rejected', reason: 4 }
// ]

-----------------------------------------------

const promise2 = Promise.allSettled([
   new Promise((resolve) => setTimeout(() => resolve(3), 2000)),
   Promise.resolve(2),
   '4',
   new Promise((resolve) => setTimeout(() => resolve(1), 0)),
   Promise.resolve(5),
])

promise2
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Resolved with
// [
//     { status: 'fulfilled', value: 3 },
//     { status: 'fulfilled', value: 2 },
//     { status: 'fulfilled', value: '4' },
//     { status: 'fulfilled', value: 1 },
//     { status: 'fulfilled', value: 5 }
// ]

-----------------------------------------------
```

```
const promise3 = Promise.allSettled([
   Promise.resolve(1),
   new Promise((resolve, reject) => setTimeout(() => reject(2),
0)),
   new Promise((resolve, reject) => setTimeout(() => reject(3),
2000)),
   Promise.reject(4),
   Promise.reject(5)
])

promise3
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Rejected with
// [
//   { status: 'fulfilled', value: 1 },
//   { status: 'rejected', reason: 2 },
//   { status: 'rejected', reason: 3 },
//   { status: 'rejected', reason: 4 },
//   { status: 'rejected', reason: 5 }
// ]

-----------------------------------------------

const promise4 = Promise.allSettled([
   null,
   undefined,
   new Promise((resolve) => setTimeout(() => resolve(2), 350)),
   {},
   'Hello'
])

promise4
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Resolved with
```

```
// [
//     { status: 'fulfilled', value: null },
//     { status: 'fulfilled', value: undefined },
//     { status: 'fulfilled', value: 2 },
//     { status: 'fulfilled', value: {} },
//     { status: 'fulfilled', value: 'Hello' }
// ]

-----------------------------------------------

const promise5 = Promise.allSettled([])

promise5
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Resolved with []
```

Implement Promise.any

Problem Statement

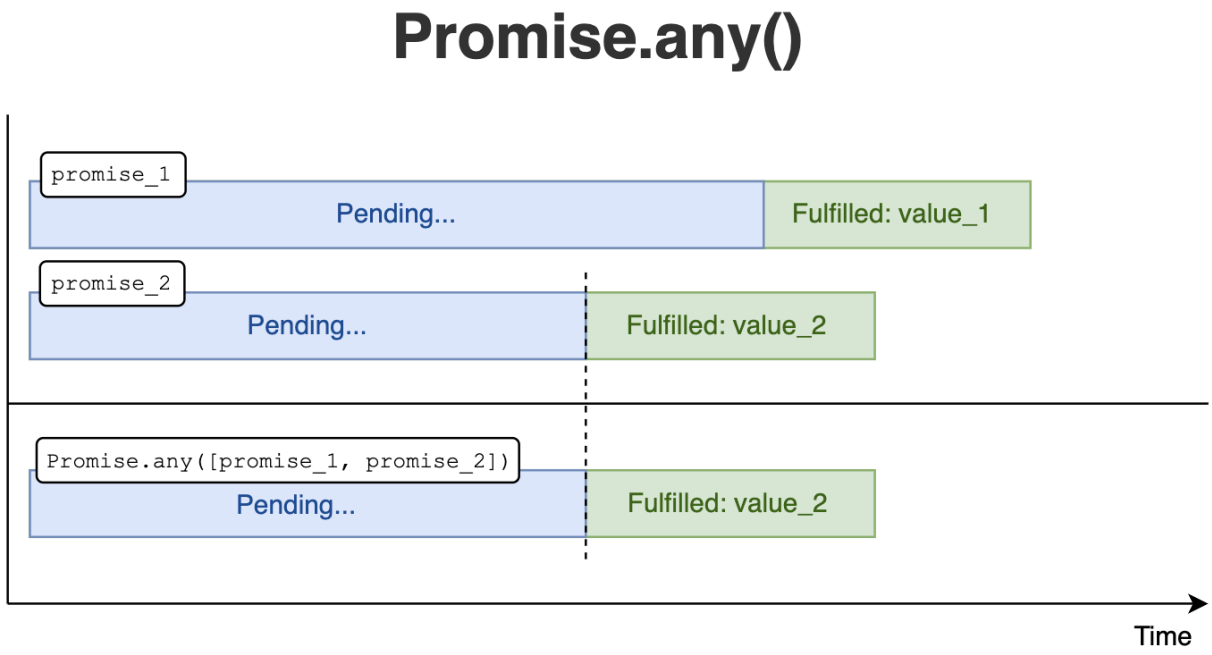
Implement a polyfill of `Promise.any()` function. And you should not use the built-in function directly for the problem, instead write your own version.

Promise.any() is a helper function that runs multiple promises in parallel and resolves to the value of the first successfully resolved promise from the input array of promises.

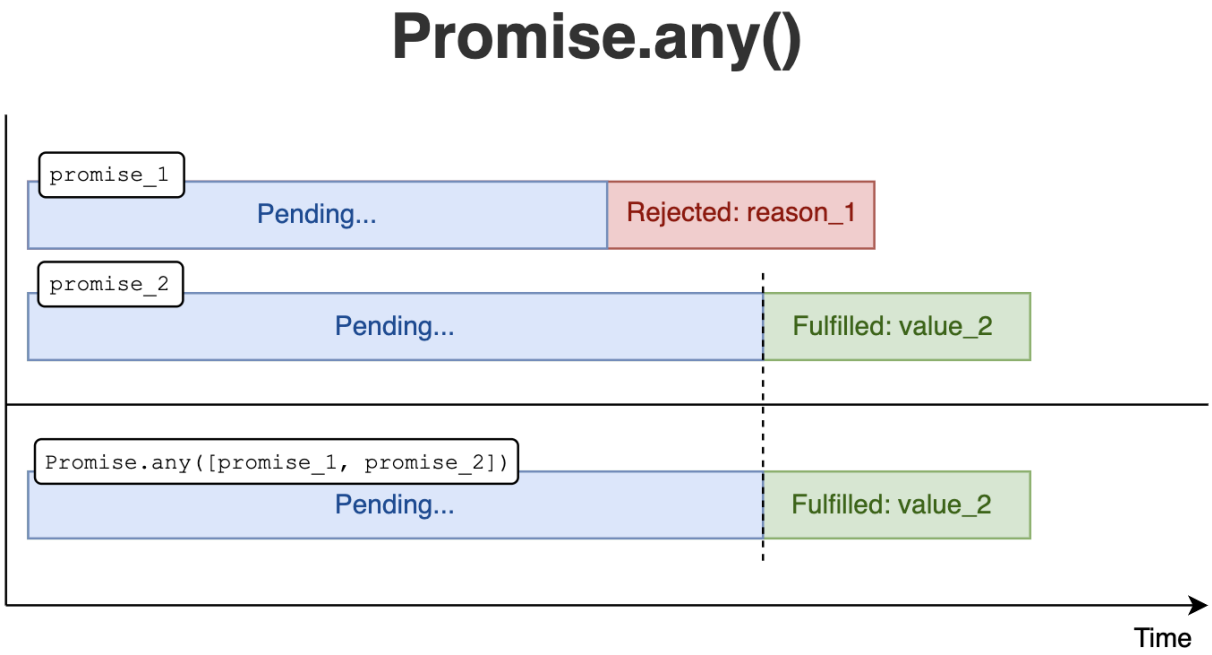
However, if all the promises in the input array are rejected or if the array is empty, then `Promise.any()` rejects with an `Aggregate Error` containing all the rejection reasons of the input promises.

Visual Examples

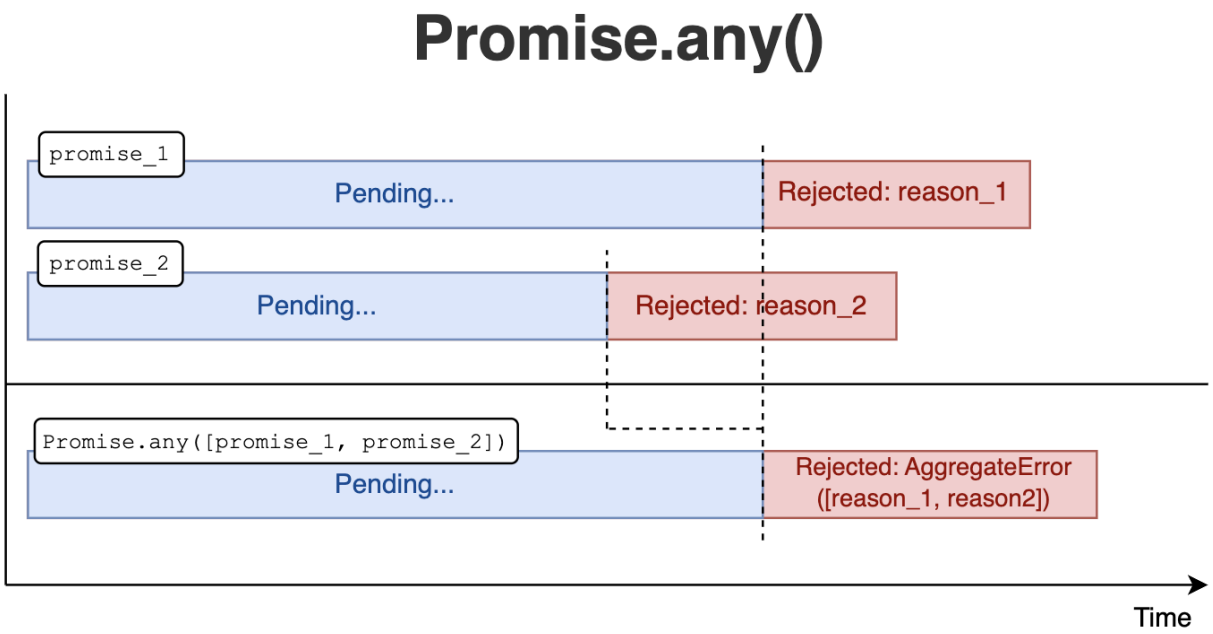
Case 1: Promise.any resolves with the value of any of the promise that resolves first from the input promises.



Case 2: Promise.any fulfills with any first fulfilled (resolved) promise. Even if some promises are rejected, these rejections are ignored.



Case 3: When all promises are rejected, then Promise.any rejects with an Aggregate Error containing all rejection reasons of input promises.



Example

```
// Syntax
const anyPromise = Promise.any([promise1, promise2]);

anyPromise.then((value) => {
   console.log('Resolved with', value);
})
.catch((error) => {
   console.log('Rejected with', error);
})
```

Implementation

`Promise.any()` takes an array of promises as input and returns a promise object.

1. The returned promise will be resolved immediately when any of the promises is resolved first from the input array.
2. If all of the promises are rejected, then the rejected promise will be rejected with an Aggregate Error.

```
Promise.any = function(input) {
   const errors = [];

   // counter to keep track if all the promises are rejected
   let totalRejected = 0;

   return new Promise((resolve, reject) => {
       if (input.length === 0) {
           return reject(new AggregateError(errors, 'Empty
Array'));
       }

       input.forEach((elem, index) => {
           Promise.resolve(elem)
           .then((value) => {
               resolve(value);
           })
           .catch((reason) => {
               errors[index] = reason;
               totalRejected++;

               if (totalRejected === input.length) {
                   reject(new AggregateError(errors, 'All
promises rejected'));
```

```
                }
            })
        })
    })
}
```

Test Case

```
const promise1 = Promise.any([
   Promise.reject(1),
   new Promise((resolve) => setTimeout(() => resolve(2), 2000)),
   Promise.reject(3),
   Promise.reject(4)
])

promise1
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Resolved with 2

-----------------------------------------------

const promise2 = Promise.any([
   new Promise((_, reject) => setTimeout(() => reject(3),
2000)),
   Promise.reject(2),
   '4',
   new Promise((_, reject) => setTimeout(() => reject(1), 0)),
   Promise.reject(5),
])

promise2
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
```

```
// Resolved with 4

-----------------------------------------------

const promise3 = Promise.any([
   Promise.reject(1),
   new Promise((_, reject) => setTimeout(() => reject(2), 0)),
   Promise.reject(3),
   Promise.reject(4)
])

promise3
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Rejected with AggregateError: All promises rejected [errors]:
[ 1, 2, 3, 4 ]

-----------------------------------------------

const promise4 = Promise.any([
   null,
   undefined,
   new Promise((resolve) => setTimeout(() => resolve(2), 350)),
   {},
   'Hello'
])

promise4
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Resolved with null

-----------------------------------------------

const promise5 = Promise.any([])
```

```
promise5
.then(value => console.log('Resolved with', value))
.catch(value => console.log('Rejected with', value))
// Rejected with AggregateError: Empty Array []
```

Implement Promise.race

Problem Statement

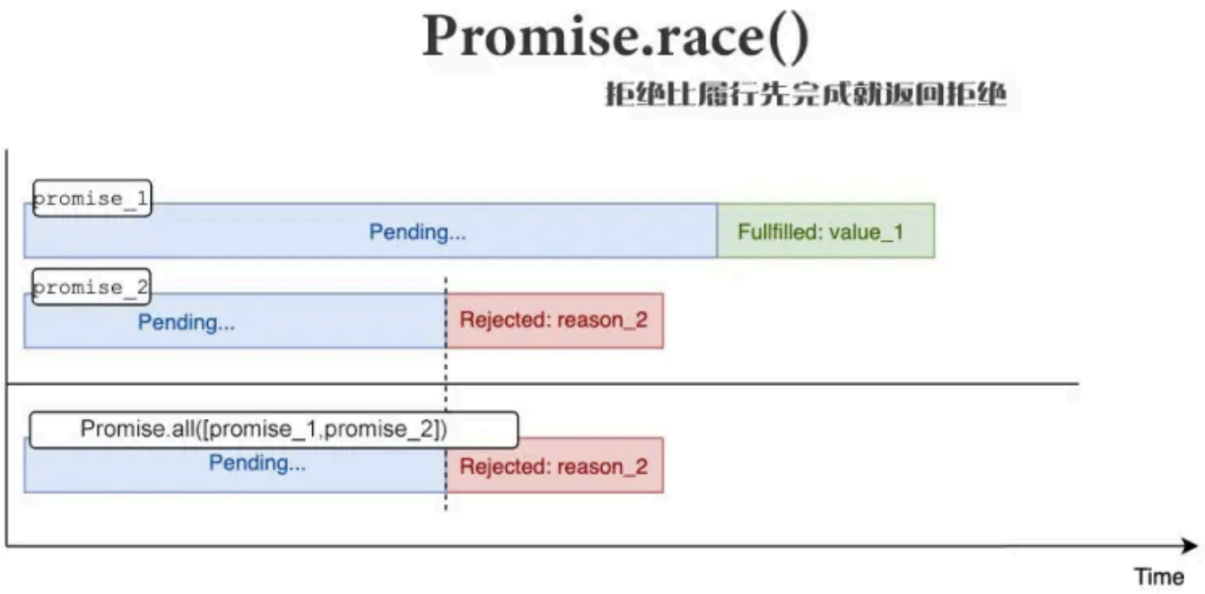
Implement a polyfill of `Promise.race()` function. And you should not use the built-in function directly for the problem, instead write your own version.

Promise.race() is a helper function that runs multiple promises in parallel and returns a promise which resolves or rejects based on whichever promise settles first.

Visual Examples

Case 1: Promise.race resolves with the value if the first settled promise is resolved.

Case 2: Promise.race also rejects with the reason if the first settled promise is rejected.



Example

```
// Syntax
const racePromise = Promise.race([promise1, promise2]);

racePromise.then((value) => {
   console.log('Resolved with', value);
})
.catch((error) => {
   console.log('Rejected with', error);
})
```

Implementation

Promise.race() takes an array of promises as input and returns a promise object.

1. The returned promise will be resolved/rejected as soon as any of the promise in the input array settles (resolves/rejects) first.

```
Promise.race = function(input) {
   return new Promise((resolve, reject) => {
       input.forEach((elem) => {
           Promise.resolve(elem)
           .then((value) => resolve(value))
           .catch((reason) => reject(reason))
       })
   })
}
```

Test Case

```
const promise1 = Promise.race([
   Promise.reject(1),
   new Promise((resolve) => setTimeout(() => resolve(2), 2000)),
   Promise.reject(3),
   Promise.reject(4)
])

promise1
.then(value => console.log('Resolved with 1', value))
.catch(value => console.log('Rejected with 1', value))
// Rejected with 1

-----------------------------------------

const promise2 = Promise.race([
   new Promise((_, reject) => setTimeout(() => reject(3),
2000)),
   Promise.reject(2),
   '4',
   new Promise((_, reject) => setTimeout(() => reject(1), 0)),
   Promise.reject(5)
])

promise2
.then(value => console.log('Resolved with 2', value))
.catch(value => console.log('Rejected with 2', value))
// Resolved with 4

-----------------------------------------

const promise3 = Promise.race([
   new Promise((_, reject) => setTimeout(() => reject(2), 0)),
   Promise.reject(3),
```

```
   Promise.reject(4)
])

promise3
.then(value => console.log('Resolved with 2', value))
.catch(value => console.log('Rejected with 3', value))
// Rejected with 3

-----------------------------------------------

const promise4 = Promise.race([
   null,
   undefined,
   new Promise((resolve) => setTimeout(() => resolve(2), 350)),
   {},
   'Hello'
])

promise4
.then(value => console.log('Resolved with 4', value))
.catch(value => console.log('Rejected with 4', value))
// Resolved with null
```

Implement Promise.finally

Problem Statement

Implement a polyfill of `.finally()` method of JavaScript Promises. And you should not use the built-in function directly for the problem, instead write your own version.

`.finally()` method of promises accepts a callback function to be called when the promise is settled (either fulfilled or rejected). The `.finally()` method also returns a new Promise object which ultimately allows you to chain calls of other promise methods like `.then()` and `.catch()`

Use case

The most important use case of this method is that you want to perform some kind of cleanup/finalized action to be performed after the promise gets settled regardless of its outcome.

So we pass in a handler which is a callback to the finally method and the callback will not be receiving arguments because it doesn't care if the promise was either fulfilled/rejected.

Example

```
const promise = new Promise((resolve, reject) => {
   // Let mock as if we make an API call which got failed
   setTimeout(() => reject('API call failed'), 2000);
})

promise
.then((value) => console.log('Resolved with', value))
.catch(((error) => console.log('Rejected since', error)))
.finally(() => console.log('Perform some cleanup (or)
handling'))
// Rejected since API call failed
// Perform some cleanup (or) handling
```

Implementation

`.finally()` method takes a callback function as input and returns a new promise object.

1. If the original promise resolves, the returned promise will resolve with the original promise's resolved value.
2. If the original promise is rejected, the returned promise will reject with the original promise's rejected value.
3. In both the above cases, we will execute the callback passed, in case the callback function returns a promise, then we need to wait for that promise to settle before we can resolve or reject the returned promise. (As per the spec)

Now, let's move on to see the full implementation of `finally()` method in code:

```
// Finally returns a promise which fulfills or rejects based on
original promise's state and value
Promise.finally = function (callback) {
   return new MyPromise((resolve, reject) => {
       // To track state and value of the current promise when
settled
       let val;
       let wasRejected;

       // we call the callback when our original promise is
settled
       this.then((value) => {
           wasRejected = false;
           val = value;
           return callback();
       }, (err) => {
           wasRejected = true;
           val = err;
           return callback();
       })
       .then(() => {
           // The callback could also return a promise, so we
should wait for it to settle before we resolve/reject
           // If the callback didn't have any error we
resolve/reject the promise based on promise state
           if (!wasRejected) {
               return resolve(val);
           }
           return reject(val);
       })
       .catch((err) => {
           // If the callback returns a rejected promise or if
the callback throws error
           // We must reject with this new error (as per the
spec, refer MDN)
           return reject(err);
```

```
        })
    })
}
```

Note:
If the callback passed to the `.finally()` method has any `throw` statements which throws an error (or) if the callback function returns a rejected promise, then we should reject the returned promise from `.finally()` with the new error reason.

```
Promise.reject(3)
.finally(() => { throw 'New Error'; })
.catch(console.log)
// New Error

Promise.reject(3)
.finally(() => Promise.reject('Rejected Error'))
.catch(console.log)
// Rejected Error
```

Test Cases

```
Promise.resolve(10)
.then((value) => console.log('Resolved with', value))
.catch(((error) => console.log('Rejected with', error)))
.finally(() => console.log('Perform some cleanup (or)
handling'))
// Resolved with 10
// Perform some cleanup (or) handling

// ---------------------------------------
```

```
Promise.reject(20)
.then((value) => console.log('Resolved with', value))
.catch(((error) => console.log('Rejected with', error)))
.finally(() => console.log('Perform some cleanup (or)
handling'))
// Rejected with 20
// Perform some cleanup (or) handling

// ---------------------------------------

Promise.resolve(30)
.then((value) => console.log('Resolved with', value))
.finally(() => {
   console.log('Perform some cleanup (or) handling')
   throw 'New Error'
})
.then((value) => console.log('Resolved with', value))
.catch(((error) => console.log('Rejected with', error)))
// Resolved with 30
// Perform some cleanup (or) handling
// Rejected with New Error

// ---------------------------------------

Promise.reject(40)
.then((value) => console.log('Resolved with', value))
.finally(() => {
   console.log('Perform some cleanup (or) handling')
   return Promise.reject(50);
})
.then((value) => console.log('Resolved with', value))
.catch(((error) => console.log('Rejected with', error)))
// Perform some cleanup (or) handling
// Rejected with 50
```

Implement Custom Javascript Promise

Problem Statement

As the title of the question states, you need to be able to emulate the Native JavaScript Promise and build your own version of Promise (say `MyPromise`) object from scratch.

Building JavaScript Promise implementation has become an increasingly popular Interview Question. It tests your core understanding and concepts of Asynchronous JS and promises at a fundamental level. It also shows you can effectively build objects and implement powerful patterns like chaining.

Concept

By definition - “A Promise object represents the eventual completion or failure of an asynchronous operation and its resulting value”.

Basically it’s a guarantee for your asynchronous task will either succeed or fail at some point of time while executing.

So, a promise can be in one of the three states:

➔PENDING, initial state when an operation is in progress.
➔FULFILLED, indicates that the operation was successful.

➔REJECTED, indicates a failure in an operation.

Note: We call a promise is settled when it is either fulfilled or rejected. Also, the terminologies `fulfilled` and `resolved` are often used interchangeably in context which ultimately indicates an operation was successful.

We will continue to implement our own promise as per the details of this specification defines us on the behavior of Promises.

Tip: Don't get overwhelmed with all the details, we will break down and simplify into easy steps and get our work done. Okay, cool!!

Example

Let's understand the skeleton of a Promise and how we use it.

It has a constructor function that accepts a callback function as input which we would call the `executorFunction`.

The executor function will be passed `resolve` and `reject` callback functions as arguments.

Also the Promise object has methods like `then`, `catch` and `finally`.

```
const promise = new Promise((resolve, reject) => {
   /*
     Your code logic goes here and you call resolve(value)
     or reject(error) to resolve or reject the promise
   */
})

promise.then((value) => {
   // Code logic on success of an operation
}).catch((error) => {
   // Code logic on failure of an operation
}).finally(() => {
   // Code logic to be executed after completion of operation
})
```

Implementation

Let's start by defining our Promise class as `MyPromise`.
So, we have some properties that are required to be defined:

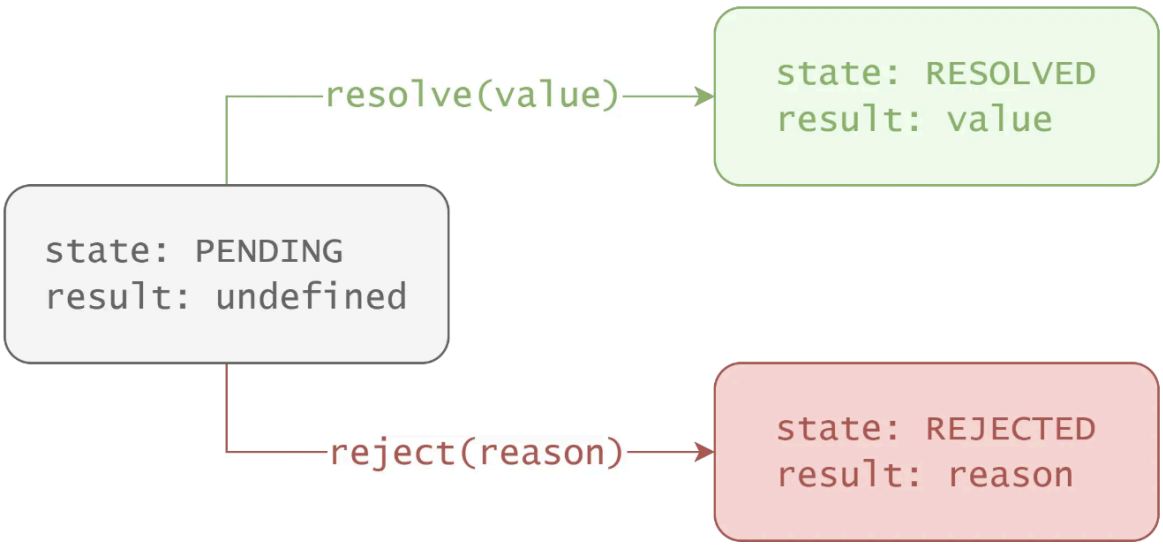
- ➔**state:** can be either of `PENDING`, `FULFILLED` or `REJECTED`.
- ➔**handlers:** stores callbacks of then, catch, finally methods (these handlers will be executed only when a promise is settled i.e. resolved or rejected).
- ➔**value:** either the value when resolved or error when rejected.

The `executorFn` passed to our constructor will be called immediately with the `resolve` and `reject` methods passed as arguments to it.

```
const STATE = {
   PENDING: 'PENDING',
   FULFILLED: 'FULFILLED',
   REJECTED: 'REJECTED',
}

class MyPromise {
   constructor(executorFn) {
       // Initial state of Promise is empty
       this.state = STATE.PENDING;
       this.value = undefined;
       this.handlers = [];

       // Invoke executorFn immediately by passing the
       // _resolve and _reject functions of our class
       try {
           executorFn(this._resolve, this._reject);
       } catch (error) {
           this._reject(error);
       }
   }

   _resolve = (value) => {}

   _reject = (error) => {}

   then(onSuccess, onFailure) {}

   catch(onFailure) {}

   finally(callback) {}
}
```

Now, let's update the implementation for the _resolve() and _reject() methods of our MyPromise class.

The aim of these two methods is to update the `state` and `value` properties and call all the handlers which were attached to the then, catch, finally methods.



We should not process resolve() or reject() calls after the promise is settled (i.e. when not in PENDING state).

```
_resolve = (value) => {
    this.updateResult(value, STATE.FULFILLED);
}

_reject = (error) => {
```

```
    this.updateResult(error, STATE.REJECTED);
}

updateResult(value, state) {
    // This is to make the processing async as per the spec
    // Since it must not be updated unless the execution context
is empty. Refer this https://promisesaplus.com/#notes
    setTimeout(() => {
        // Process the promise only if it's in pending state.
        // An already rejected or resolved promise should not be
processed
        if (this.state !== STATE.PENDING) {
            return;
        }

        // check if the `value` is also a promise
        // if yes, then wait for it to resolve/reject first
        // we pass along _resolve, _reject handlers to then()
        // so that when the other promise resolves/rejects our
promise also either resolves/rejects
        if (value instanceof MyPromise) {
            return value.then(this._resolve, this._reject);
        }

        // Update the state and value for the `promise`
        this.value = value;
        this.state = state;

        // since our promise is now settled
        // we execute all handlers already attached with then,
catch and finally methods. Will implement this function
`executeHandlers` in next phase
        this.executeHandlers();
    }, 0);
}
```

Now, coming to the most important part which is our then() method implementation.

This important part of implementation makes chaining of promises look like magic.

So, the `then()` takes 2 arguments as callback functions `onSuccess` and `onFailure`.

➔`onSuccess` handler is called if the Promise was fulfilled.
➔`onFailure` handler is called if the Promise was rejected.

Now, as said, the beauty of the `then()` method makes it possible to chain promises. How? Let's see..

The `then()` method will return a new Promise object which makes it possible to add the next `.then()` handler to that returned Promise object.

One of the typical use cases of Promises chaining is when we have a sequence of asynchronous tasks to be performed one after another.

Below is a simple code snippet to understand how Promise chaining is used:

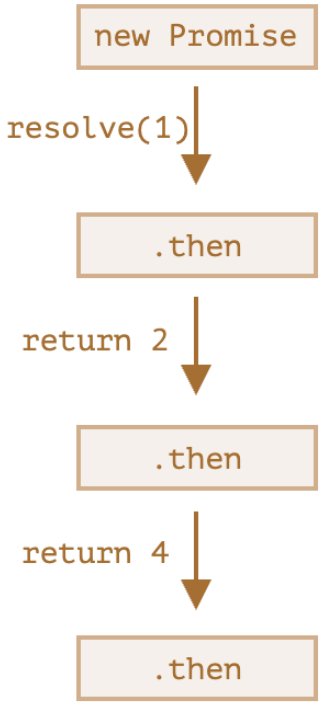
```
// Let's just assume `doubleAsync` doubles the value in
asynchronously
const doubleAsync = (value) => value * 2;
```

```
new Promise((resolve, reject) => {
   setTimeout(() => resolve(1), 1000);
})
.then((result) => {
   console.log(result); // 1
   return doubleAsync(result);
})
.then((result) => {
   console.log(result); // 2
   return doubleAsync(result);

})
.then((result) => {
   console.log(result); // 4
   return doubleAsync(result);
});
```

Facts:

1. The whole thing works, because every call to a `.then()` returns a new promise, so that we can call the next `.then()` handler on it.

2. When a handler returns a value, it becomes the result of that promise, so the next `.then()` handler's callback will be called with that result.



Now coming back to our implementation logic,

The callbacks passed to `.then()` methods are stored in `handlers` array (which is a property of our MyPromise class) using a utility function `addHandlers()` which is our utility function.

So, the `addHandlers()` method takes an object as argument `{ onSuccess, onFailure }` which will be executed when the promise is settled.

Let's walk through the implementation in code below:

```
then(onSuccess, onFailure) {
```

```
   // return a promise to make the promises chainable
   return new MyPromise((resolve, reject) => {

       // add a handler object so that when the current promise
is settled it can be called
       this.addHandlers({
           onSuccess: function (value) {
               // if no onSuccess provided, resolve the value
for the next promise chain
               if (!onSuccess) {
                   return resolve(value);
               }
               try {
                   // else resolve with the returned value from
onSuccess callback
                   return resolve(onSuccess(value));
               } catch (error) {
                   // we reject when some error is thrown
                   return reject(error);
               }
           },

           onFailure: function (value) {
               // if no onFailure provided, reject the value for
the next promise chain
               if (!onFailure) {
                   return reject(value);
               }
               try {
                   // we resolve here and not reject, since
we've handled the current promise's rejection
                   // so we resolve with the returned value from
onFailure callback for next promise chain
                   // consider this example code,
promise.then(() => {}, () => {}).then(() => {})
                   return resolve(onFailure(value));
```

```
                } catch (error) {
                    return reject(error);
                }
            }
        });
    });
}

addHandlers(handlers) {
    this.handlers.push(handlers);
    // whenever a handler is added, we need to execute them
immediately if the promise is settled
    this.executeHandlers();
}

executeHandlers() {
    // Don't execute handlers if promise is not yet fulfilled or
rejected
    if (this.state === STATE.PENDING) {
        return;
    }

    // We have multiple handlers because add them for .finally
block too
    this.handlers.forEach((handler) => {
        if (this.state === STATE.FULFILLED) {
            handler.onSuccess(this.value);
        }
        else {
            handler.onFailure(this.value);
        }
    });

    // After processing all the handlers, we reset it to empty,
since a handler should run only once when the Promise settles.
    this.handlers = [];
```

```
}
```

Now, let's implement the `.catch()` method. If you notice, we can take help of the `.then()` method itself where we have already handled the logic for `onFailure` case by passing as second argument.

```
// Since then method take the second function as onFailure,
// we can leverage it while implementing catch
catch(onFailure) {
    return this.then(null, onFailure);
}
```

Great, now coming to the final thing of our custom Promise implementation where we will implement the `.finally()` method.

Let's first understand the behavior of the `.finally()` method.

As per the MDN,
The `.finally()` method returns a new Promise object. When the promise to which this `.finally()` method is attached gets settled, the callback function which is passed as argument to `.finally()` method will be executed.

This provides a way for your code to be run whether the promise was fulfilled or rejected. The `.finally()` method is very similar to calling the `.then(onFinally, onFinally)`.

The `.finally()` method returns a Promise which will be settled with previous fulfilled or rejected value.

Example

```
Promise.resolve(2).then(() => {}, () => {})
// resolves to undefined

Promise.resolve(2).finally(() => {})
// resolves to 2

Promise.reject(3).then(() => {}, () => {})
// resolves to undefined

Promise.reject(3).finally(() => {})
// rejects to 3
```

Now let's see the implementation in code:

```
// Finally block returns a promise which fails or succeeds with
the previous promise value
finally(callback) {
   return new MyPromise((resolve, reject) => {
       // To track state and value of the current promise when
settled
       let val;
       let wasRejected;

       // we call the callback when our current promise is
settled
       this.then((value) => {
           wasRejected = false;
           val = value;
```

```
            return callback();
        }, (err) => {
            wasRejected = true;
            val = err;
            return callback();
        })
        .then(() => {
            // The callback could also return a promise, so we
should wait for it to settle before we resolve/reject
            // If the callback didn't have any error we
resolve/reject the promise based on promise state
            if (!wasRejected) {
                return resolve(val);
            }
            return reject(val);
        })
        .catch((err) => {
            // If the callback returns a rejected promise or if
the callback throws error
            // We must reject with this new error (as per the
spec, refer MDN)
            return reject(err);
        })
    })
}
```

That’s amazing, we just made it!!

Implementing all of the logical behavior to implement our own version of Promises named `MyPromise`.

Below is the complete code implementation of `MyPromise`:

```
const STATE = {
   PENDING: 'PENDING',
   FULFILLED: 'FULFILLED',
   REJECTED: 'REJECTED',
}

class MyPromise {
   constructor(executorFn) {
       this.state = STATE.PENDING;
       this.value = undefined;
       this.handlers = [];

       try {
           executorFn(this._resolve, this._reject);
       } catch (error) {
           this._reject(error);
       }
   }
   // using arrow functions for _resolve() and _reject() methods
since we don't want to lose context of "this" when these methods
are called
   _resolve = (value) => {
       this.updateResult(value, STATE.FULFILLED);
   }

   _reject = (error) => {
       this.updateResult(error, STATE.REJECTED);
   }

   updateResult(value, state) {
       setTimeout(() => {
           if (this.state !== STATE.PENDING) {
               return;
           }

           if (value instanceof MyPromise) {
```

```
            return value.then(this._resolve, this._reject);
        }

        this.value = value;
        this.state = state;

        this.executeHandlers();
    }, 0);
}

then(onSuccess, onFailure) {
    return new MyPromise((resolve, reject) => {
        this.addHandlers({
            onSuccess: function (value) {
                if (!onSuccess) {
                    return resolve(value);
                }
                try {
                    return resolve(onSuccess(value));
                } catch (error) {
                    return reject(error);
                }
            },
            onFailure: function (value) {
                if (!onFailure) {
                    return reject(value);
                }
                try {
                    return resolve(onFailure(value));
                } catch (error) {
                    return reject(error);
                }
            }
        });
    });
}
```

```
    addHandlers(handlers) {
        this.handlers.push(handlers);
        this.executeHandlers();
    }

    executeHandlers() {
        if (this.state === STATE.PENDING) {
            return;
        }

        this.handlers.forEach((handler) => {
            if (this.state === STATE.FULFILLED) {
                handler.onSuccess(this.value);
            }
            else {
                handler.onFailure(this.value);
            }
        });

        this.handlers = [];
    }

    catch(onFailure) {
        return this.then(null, onFailure);
    }

    finally(callback) {
        return new MyPromise((resolve, reject) => {
            let val;
            let wasRejected;
            this.then((value) => {
                wasRejected = false;
                val = value;
                return callback();
            }, (err) => {
```

```
                wasRejected = true;
                val = err;
                return callback();
            })
            .then(() => {
                if (!wasRejected) {
                    return resolve(val);
                }
                return reject(val);
            })
            .catch((err) => {
                return reject(err);
            })
        })
    }
}
```

Test Cases

```
new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 2000)
})
.then(
    (value) => console.log('Resolved with', value),
    (error) => console.log('Rejected with', error)
)
// Resolved with 10

// ---------------------------------------

new MyPromise((resolve, reject) => {
    setTimeout(() => {
```

```
        reject(20);
    }, 2000)
})
.then(
    (value) => console.log('Resolved with', value),
    (error) => console.log('Rejected with', error)
)
// Rejected with 20

// ---------------------------------------

new MyPromise((resolve, reject) => {
    setTimeout(() => {
        reject(30);
    }, 2000)
})
.then((value) => console.log('Resolved with', value))
.catch((error) => console.log('Rejected with', error))
// Rejected with 30

// ---------------------------------------

new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(40);
    }, 2000)
})
.then()
.then((value) => console.log('Resolved with', value))
// Resolved with 40

// ---------------------------------------

new MyPromise((resolve, reject) => {
    setTimeout(() => {
        reject(50);
```

```
    }, 2000)
})
.then()
.then((value) => console.log('Resolved with', value))
.then(null, (error) => console.log('Rejected with', error))
// Rejected with 50

// ----------------------------------------

new MyPromise((resolve, reject) => {
    setTimeout(() => {
        reject(60);
    }, 2000)
})
.then(
    (value) => console.log('Resolved with', value),
    (error) => error * 10
)
.then((value) => console.log('Resolved with', value))
// Resolved with 600

// ----------------------------------------

new MyPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(70);
    }, 2000)
})
.then((value) => value * 10)
.finally((value) => {
    console.log('Finally value is', value);
    return 'Hello' + value;
})
// Finally value is undefined

// ----------------------------------------
```

```
new MyPromise((resolve, reject) => {
   setTimeout(() => {
       resolve(80);
   }, 2000)
})
.then((value) => value * 10)
.finally((value) => {
   console.log('Finally value is', value);
   return 'Hello' + value;
})
.then((value) => console.log('Resolved with', value))
// Finally value is undefined
// Resolved with 800

// ---------------------------------------

new MyPromise((resolve, reject) => {
   setTimeout(() => {
       reject(90);
   }, 2000)
})
.then()
.then()
.then()
.finally((value) => console.log('Finally value is', value))
.then()
.then()
.then((value) => console.log('Resolved with', value))
.catch((error) => console.log('Rejected with', error))
// Finally value is undefined
// Rejected with 90

// ---------------------------------------

new MyPromise((resolve, reject) => {
```

```
    setTimeout(() => {
        reject(100);
    }, 2000)
})
.finally((value) => {
    console.log('Finally value is', value);
    throw 'New Error';
})
.then((value) => console.log('Resolved with', value))
.catch((error) => console.log('Rejected with', error))
// Finally value is undefined
// Rejected with New Error
```

Throttling Promises by Batching

Problem Statement

Implement a function `throttlePromises()` which takes an array of functions as input and each of those functions return a promise object. We also pass a number `max` which denotes the maximum promises that need to be processed concurrently.

Scenario

To understand this problem statement more clearly let's imagine this scenario first -

As said, the function `throttlePromises()` will take an array of functions which returns a promise.

Now, assume each of the function calls make an API call and return a promise which either resolves/rejects. If you have a scenario you need to make 50 API calls concurrently. It would be a bad practice since we're overloading the server by making 50 API calls instantly.

So, a better approach is to throttle it. Basically, we can make API calls in batches. The input `max` passed to our function would be our batch size.

Example

```
throttlePromises(listOfAPIsToCall, 5)
.then(data => {
   // data - If all the API calls processed in batches succeeds
})
.catch(error => {
   // error - If any of the API call fails
})
```

Implementation

`throttlePromises()` function takes an array of functions (each func returns a promise) as input and also a number `max` as input.

1. The `throttlePromises()` will also return a promise
    a. Returned Promise resolves when all of the promises of input functions are resolved
    b. Returned Promise rejects when any of the promise of input functions is rejected
2. Batch functions into a size of `max` and process concurrently using the utility of `Promise.all()`

Now, let's move on to see the full implementation in code:

```
function throttlePromises(funcsArr, max) {
   const result = [];
   let nextAPIBatch = 0;
```

```
   // We return a new promise which waits until all API calls
are batched and made
   // If any of API call fails, we reject the promise
   // If all API calls made in batches succeed, we resolve the
promise
   return new Promise((resolve, reject) => {
       // Using IIFE since the function needs to be called
immediately once after declared (Basically triggering)
       (function fetchAPICalls () {
           const start = nextAPIBatch;
           const end = nextAPIBatch + max;

           const nextAPICallsToMake = funcsArr.slice(start,
end);
           const nextAPICallsPromises =
nextAPICallsToMake.map(fn => fn());

           // We make use of Promise.all since it will parallely
execute all the batch of promises and collectively return the
results if all fulfilled, else returns error if any failure
           Promise.all(nextAPICallsPromises)
           .then(data => {
               result.push(...data);

               if (result.length === funcsArr.length) {
                   // If all API calls fulfilled, resolve
immediately
                   resolve(result);
               }
               else {
                   // Continue to batch and make nextAPICalls
                   nextAPIBatch = end;
                   fetchAPICalls();
               }
           })
           .catch(error => {
```

```
                // If any API fails, reject immediately
                reject(error);
            })
        })();
    })
}
```

Test Cases

```
// Just a utility to get random timer for setTimeout
// So that we can mock promises in async fashion
const getRandomTimer = () => Math.round(Math.random() * 1000);

const getFulfillingPromise = (value) => {
    return new Promise(resolve => {
        setTimeout(() => resolve(value), getRandomTimer())
    })
}

const getRejectingPromise = (value) => {
    return new Promise((resolve, reject) => {
        setTimeout(() => reject(value), getRandomTimer())
    })
}

const input1 = new Array(10).fill(null).map((elem, index) => ()
=> getFulfillingPromise(index));

const input2 = new Array(10).fill(null).map((elem, index) => {
    if (index === 6)
        return () => getRejectingPromise(index);
    else
        return () => getFulfillingPromise(index);
})
```

```
throttlePromises(input1, 5)
.then(data => console.log('Resolved with', data))
.catch(error => console.log('Rejected with', error))
// Resolved with [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

// ----------------------------------------

throttlePromises(input1, 3)
.then(data => console.log('Resolved with', data))
.catch(error => console.log('Rejected with', error))
// Resolved with [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

// ----------------------------------------

throttlePromises(input2, 4)
.then(data => console.log('Resolved with', data))
.catch(error => console.log('Rejected with', error))
// Rejected with 6
```

Implement Custom Deep Equal

Problem Statement

Implement a function `deepEqual()` which takes in two arguments as input which deeply compares both arguments and returns a boolean which determines if they are equal.

In the Lodash library, we have an utility function `isEqual()` which is useful when you want to compare complex data types by value and not by reference.

Our problem statement is to implement our own custom version of the similar `isEqual()` utility method.

The two arguments passed to the `deepEqual()` can be of

1. Primitives (undefined, boolean, string, number, null)
2. Object Literals
3. Arrays

Note: In JavaScript `null` is considered as a primitive value because it's behavior is seemingly primitive. Although, typeof null is returned as an `object`.

Example

```
const input1 = { a: 'Hello' };
```

```
const input2 = { a: 'Hello' };

deepEqual(input1, input2); // true

// --------------------------------

const obj1 = { name: 'Peter', stats: { points: 45, isActive:
false }};
const obj2 = { name: 'Peter', stats: { points: 45, isActive:
false }};

deepEqual(obj1, obj2); // true

// --------------------------------

const elem1 = {
   name: 'Peter',
   stats: { points: 45, isActive: false },
   order: [1, 3, 5, 6]
};
const elem2 = {
   name: 'Peter',
   stats: { points: 45, isActive: false },
   order: [1, 3, 5, 10]
};

deepEqual(elem1, elem2); // false
```

Concept

Let’s first understand what we mean by pass by value and pass by reference.

JavaScript assigns every object you create to its own place in memory. Basically, it will give you a reference memory address which will be pointing to the object you create.

```
// The variable `obj` holds the reference to the object created
const obj = { a: 'Hello' };
```

So, we can't make use of the equality operator `===` to deeply compare two objects since references of both objects will be different.

For example,

```
const obj1 = { name: 'Peter', stats: { points: 45, isActive:
false }};
const obj2 = { name: 'Peter', stats: { points: 45, isActive:
false }};

obj1 === obj2 // false (different reference)
obj1 === obj1 // true (same reference)
```

Even if both the objects have exactly the same content and structure, their references are different.

Note: JavaScript's Equality operator `===` only compares primitives by value and objects by reference.

So we have to implement our own method to compare by value at each level deeply in the objects.

One more thing we need to take care in our implementation is

the circular references. That is,

```
const a = [1, '4'];
a.push(a);
// That is, const a = [ 1, '4', [Circular *a]

const b = [1, '4'];
b.push(b);
// That is, const b = [ 1, '4', [Circular *b]

deepEqual(a, b); // true
```

Implementation

Let’s breakdown the problem into smaller requirements -

1. Handling for primitives values using the Equality operator.

2. Handling for Referenced types which is Arrays and objects
    a. If both input values are visited, it means we encountered the same value again creating a circular reference. Since both are visited it means we have already compared both the values.
    b. If not visited, we further need to compare it recursively and add to visited as we keep comparing.

Here’s the complete implementation in code:

```
// By default assigning a visited `Map` as third argument to
track circular references
function deepEqual(a, b, visited = new Map()) {
   if (Number.isNaN(a) && Number.isNaN(b)) {
       // As per spec comparing two NaN values returns false
       // Since we're comparing based on value we'll return true
       return true;
   }

   // Handles primitives like number, string, undefined,
boolean, null
   // Also handles objects with same references
   if (a === b) {
       return true;
   }

   // We need to process for equality only if `a` and `b` are
object values
   // Like Arrays [], Object Literals {}
   if (typeof a !== 'object' || typeof b !== 'object') {
       return false;
   }

   // Handling for circular references
   // Means we've already compared these two values
   if (visited.has(a) && visited.get(a) === b) {
       return true;
   }

   // we map `a -> b` in visited
   // This will indicate that of both these values will have
been compared already
   // This will be useful when we encounter same values again in
recursion
   visited.set(a, b);
```

```
    // Object.keys method works in both cases of Arrays and
Objects values
    // For Objects it gives the keys of Objects
    // For Arrays it gives indices as the keys
    const keysA = Object.keys(a);
    const keysB = Object.keys(b);

    // If both are of unequal length, obviously the values can't
be equal any further
    if (keysA.length !== keysB.length) {
        return false;
    }

    for (let i = 0; i < keysA.length; i++) {
        // We recursively call the same logic
        const keyA = keysA[i], keyB = keysB[i];
        if (!deepEqual(a[keyA], b[keyB], visited)) {
            return false;
        }
    }

    // We've processed at all levels deeply, so we can say both
values are deeply equal now
    return true;
}
```

Test Cases

```
let obj1, obj2;

console.log(deepEqual(1, 'hello')); // false
```

```
// --------------------------------

console.log(deepEqual(1, '1')); // false

// --------------------------------

console.log(deepEqual(1, 1)); // true

// --------------------------------

console.log(deepEqual(NaN, NaN)); // true

// --------------------------------

console.log(deepEqual(NaN, null)); // false

// --------------------------------

console.log(deepEqual([], [])); // true

// --------------------------------

obj1 = { a: 1, b: 2, c: 3 };
obj2 = { a: 1, b: 2, c: 3 };

console.log(deepEqual(obj1, obj2)); // true

// --------------------------------

obj1 = { a: 1, b: 2, c: 4 };
obj2 = { a: 1, b: 2, c: 3 };

console.log(deepEqual(obj1, obj2)); // false

// --------------------------------
```

```
obj1 = { a: 1, b: [1, [2, [3, { c: 4 }]]], c: 4 };
obj2 = { a: 1, b: [1, [2, [3, { c: 4 }]]], c: 4 };

console.log(deepEqual(obj1, obj2)); // true

// --------------------------------

const arr1 = [1, '4'];
arr1.push(arr1);

const arr2 = [1, '4'];
arr2.push(arr2);

console.log(deepEqual(arr1, arr2)); // true
```

Implement Custom Object.assign

Problem Statement

Implement a function `objectAssign()` which is a polyfill of the built-in `Object.assign()` function. And you should not use the built-in function directly for the problem, instead write your own version.

What does `Object.assign()` do?

By definition, it copies all enumerable own properties from one or more source objects to a target object and returns the target object.

So, the `Object.assign()` will be passed a target object and any number of source objects as inputs.

Note:

- ➔ **Enumerable -** Enumerability refers to whether a property can be iterated over or accessed using iteration methods like Object.keys or for..in loop
- ➔ **Own (Ownership) -** Ownership determines whether a property belongs to the object directly or is inherited from its prototype chain.

We'll talk about these in great depth below.

Example

```
const target = { a: 1, b: 2 };
const source = { c: 4, d: 5 };

objectAssign(target, source);

console.log(target);
// { a: 1, b: 2, c: 4, d: 5 }

// ---------------------------------------

const target = { a: 1 };
const source1 = { b: 2 };
const source2 = { c: 30 };
const source3 = { d: 40 };

objectAssign(target, source1, source2, source3);

console.log(target);
// { a: 1, b: 2, c: 30, d: 40 }

// ---------------------------------------

// When target and the source has the same property name, the
value of source's property will override the value in target

const target = { a: 1, b: 20 };
const source1 = { b: 2, c: 3 };
const source2 = { c: 30, d: 4 };

objectAssign(target, source1, source2);

console.log(target);
```

```
// { a: 1, b: 2, c: 30, d: 4 }

// ---------------------------------------

const target = { a: 1 };
const source = { b: 5 };
const returnedTarget = objectAssign(target, source);

// The `returnedTarget` will be the same `target` obj which was
passed as input
console.log(returnedTarget === target); // true
```

Concepts

Before jumping straight away to the solution or implementation we need to understand a few concepts which are important for the problem.

1. Objects - Property Flags and descriptors :

On a general note we know, objects can store properties, where each property is a “key-value” pair. But there’s more to object properties.

Property Flags :

We know the fact that in an object a key has a value. But the fact is the value which we have mapped to a `key` in the object is also an object which has an attribute of value in it which ultimately stores our value. But, when we try to access/write the property we don’t see it as an object (actually it’s an internal implementation of the language).

We usually don’t see them while doing console.log for an object because generally they do not show up. Although JavaScript provides us some utility methods to get those. Refer below example -

```
const user = {
   name: "Peter"
}

// The below method helps us to get the descriptor object (full information about a property) for the property 'name' from the object 'user'
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

console.log(descriptor);
/* This below object is basically called as 'property descriptor'
{
   value: 'Peter', // This what we were talking about
   writable: true,
   enumerable: true,
   configurable: true
}
*/
```

Also note from the above example, the property descriptor object mapped to the `name` key in our `user` object has 3 more special attributes (so-called "flags").

➔**writable -** if `true`, the value can be changed/modified, else it’s read-only.

➔**enumerable -** if `true`, then listed in loops (or) the property can be iterated, otherwise not listed.

➔**configurable -** if `true`, the property can be deleted and these 3 attributes can be modified, otherwise not.

```
// We can use Object.defineProperty() method to set these
attributes for a property for the give object

Object.defineProperty(object, propertyName, descriptor);
```

Let’s see some example in code:

```
const user = {
   age: 20
}

// `writable` set to false
Object.defineProperty(user, 'name', {
   value: 'Peter',
   writable: false,
   enumerable: true,
   configurable: true
})

console.log(user.name); // Peter
user.name = 'Parker';
console.log(user.name); // Peter

// Re-defining the descriptor of the property 'name'
// `writable` set to true
Object.defineProperty(user, 'name', {
   value: 'Parker',
   writable: true,
```

```
   enumerable: true,
   configurable: true
})

console.log(user.name); // Parker
user.name = 'Parker';
console.log(user.name); // Parker

// -----------------------------------------

const user = {
   age: 20,
   dept: 'cse'
}

// `enumerable` set to false
Object.defineProperty(user, 'name', {
   value: 'Peter',
   writable: true,
   enumerable: false,
   configurable: true
})

for (let key in user) {
   console.log(key);
}
// 'age', 'dept'

console.log(Object.keys(user));
// ['age', 'dept'] - 'name' property will be ignored


// Re-defining the descriptor of the property 'name'
// `enumerable` set to true
Object.defineProperty(user, 'name', {
   value: 'Parker',
```

```
    writable: true,
    enumerable: true,
    configurable: true
})

for (let key in user) {
    console.log(key);
}
// 'age', 'dept', 'name'

console.log(Object.keys(user));
// ['age', 'dept', 'name']

// -----------------------------------------

const user = {
    age: 20,
}

// `configurable` set to false
Object.defineProperty(user, 'name', {
    value: 'Peter',
    writable: true,
    enumerable: true,
    configurable: false
})


// Let's try redefining the descriptors of the property 'name'

// trying `writable` set to false
Object.defineProperty(user, 'name', {
    value: 'Parker',
    writable: false,
    enumerable: true,
    configurable: true
```

```
})
// throws [Error]: Cannot redefine property: name


// trying `enumerable` set to false
Object.defineProperty(user, 'name', {
   value: 'Parker',
   writable: true,
   enumerable: false,
   configurable: true
})
// throws [Error]: Cannot redefine property: name


// trying `configurable` set to false
Object.defineProperty(user, 'name', {
   value: 'Parker',
   writable: true,
   enumerable: true,
   configurable: true
})
// throws [Error]: Cannot redefine property: name
// Yes, we can't also change the `configured` flag back to true
```

2. Ownership:

Ownership determines whether a property belongs to an object directly or is inherited from its prototype chain. It is important to differentiate between own and inherited properties.

Own properties are defined directly on the object itself.

Ever wondered about methods like `.toUpperCase()` on strings and `.reverse()` on Arrays and properties like `.length` on Arrays? We never defined them or implemented them on our objects but they still exist.

The answer is that these methods come built-in within each type of data structure thanks to something called **prototype inheritance.**

In JavaScript, an object can inherit properties of another object. The object from where the properties are inherited is called the prototype. In short, objects can inherit properties from other objects — the prototypes.

```
// Use hasOwnProperty() and hasOwn() methods to check if own
property or inherited property
const obj = { name: 'Peter' };
console.log(obj.hasOwnProperty('name')); // true
console.log(Object.hasOwn(obj, 'name')); // true



const newObj = { age: 20 };
const parent = { name: 'Peter' };

Object.setPrototypeOf(newObj, parent);

console.log(newObj.hasOwnProperty('name')); // false
console.log(Object.hasOwn(newObj, 'name')); // false
console.log(newObj.hasOwnProperty('age')); // true
console.log(newObj.name); // 'Peter' (property accessed from
parent in prototype chain)
```

Note: When you iterate using Object.keys() methods or normal for..in loop only own and enumerable properties will be iterated on.

3. Symbols :

(Although the concept of how `Symbol` works is not that necessary to be known to solve this problem, but it's definitely a good-to-learn)

Only two primitive types are allowed to be used as keys in objects.

➔string type
➔symbol type

Note: When you try to set a key of type number it's automatically converted to string. So, obj[1] is the same as obj["1"] and obj[true] is the same as obj["true"].

Symbols basically are "always" a unique value returned. Symbols can be used to be used as a key in an object.

```
// You can't see what the value is, since it's meant to be
private as per spec
// Instead it just prints `Symbol()`
let sym = Symbol();
console.log(sym); // Symbol()

// Since Symbols always will be unique, that's why we can use
these for keys in objects
Symbol() === Symbol(); // false
```

```
// A symbol can also be passed an optional description as an
argument
const symWithDesc = Symbol('id');
console.log(symWithDesc); // Symbol(id)

// Since same symbol reference
console.log(symWithDesc === symWithDesc); // true

// Everytime the constructor of Symbol is called it returns a
unique value even if the description for both are same - 'id'
console.log(Symbol('id') === Symbol('id')); // false

// 1. Global Symbols: You can have the same Symbol with
description declared globally to access anywhere in your
codebase
// 2. Use Symbol with `.for()` method. This method creates a
global Symbol value for the passed description 'world' (as in
below case) and can be accessed anywhere across your codebase
// 3. `.for()` creates a new Symbol with the description with
doesn't exists and returns,
// if Symbol with given description already exists then returns
the same Symbol value
console.log(Symbol.for('world')); // Symbol(world)
console.log(Symbol.for('world') === Symbol.for('world')); //
true

// Note: We have to use the square-bracket notation to declare a
symbol-type property
const obj = {
   name: 'Peter', // string-type property
   [Symbol('age')]: 19 // symbol-type property
}

console.log(obj);
// { name: 'Peter', [Symbol(age)]: 19 }
```

```
// Note: String-type and Symbol-type are completely different
console.log(typeof 'Hello'); // string
console.log(typeof Symbol()); // symbol
console.log(typeof Symbol('Hello')); // symbol
```

Note: Any string type property used as key for an object will be `enumerable` as `true` by default (i.e. while looping the key will be shown). But the Symbol type property used as key on the other hand will have the enumerable as `false` by default (i.e. while looping the key will not be shown). As per the spec, Symbol type is non-enumerable since it indicates private information.

Implementation

`objectAssign()` function takes a target object as input and also any number of source objects.

1. If the target is not a valid object like `null` or `undefined` it should throw an exception.
2. Loop over all the source objects individually and collect all properties (keys) of the source object both enumerable and non-enumerable. Assign only the enumerable properties.
3. If the target and both source objects have the same property (key) and if the property in the target object has

the flag writable set to `false` then we should throw an exception.

Below is the full implementation in code,

```
// We collect all the source objects using the rest operator
function objectAssign(target, ...sources) {
   if (target === null || target === undefined) {
       throw new Error('target should not be null or
undefined');
   }

   // Using the Object constructor to wrap and turn primitives
like number, boolean, string, BigInt, NaN into its wrapper
Object form
   target = Object(target);

   // loop through all source objects and assign properties to
target object
   for (let source of sources) {
       if (source === null || source === undefined) {
           continue; // skip if the source is not a valid object
       }

       // Using the Object constructor to wrap and turn
primitives like number, boolean, string, BigInt, NaN into its
wrapper Object form
       source = Object(source);

       // Reflect.ownKeys returns an array of own property keys
including string and symbol (both enumerable and non-enumerable)
       const keys = Reflect.ownKeys(source);
       const descriptors =
Object.getOwnPropertyDescriptors(source);
```

```
        keys.forEach(key => {
            const targetDescriptor =
Object.getOwnPropertyDescriptor(target, key);

            if (targetDescriptor && targetDescriptor.writable ===
false) {
                throw new Error(`Property ${key} is not writable
to target`);
            }

            // Remember the definition of Object.assign() method
            // We should assign only enumerable properties of the
source. So if the property on the source is enumerable, assign
it to target.
            if (descriptors[key].enumerable) {
                target[key] = source[key];
            }
        })
    }

    return target;
}
```

Test Cases

```
const target = null;
const source = { name: 'Hello' };

objectAssign(target, source);
// Error: target should not be null or undefined

// ----------------------------------------
```

```
const target = 'world';
const source = { name: 'Hello' };

console.log(objectAssign(target, source));
// { name: 'Hello' }

// ------------------------------------------

const target = Infinity;
const source = { name: 'Hello' };

console.log(objectAssign(target, source));
// { name: 'Hello' }

// ------------------------------------------

const target = {};
const source = { name: 'Hello' };

console.log(objectAssign(target, source));
// { name: 'Hello' }

// ------------------------------------------

const target = { a: 1, b: 2 };
const source = { c: 30, d: -1 };

console.log(objectAssign(target, source));
// { a: 1, b: 2, c: 30, d: -1 }

// ------------------------------------------

const target = { a: 1, b: 2 };
const source = { c: 30, b: -1 };

console.log(objectAssign(target, source));
```

```
// { a: 1, b: -1, c: 30 }

// ------------------------------------------

const target = { a: 1, b: 2 };
const source1 = { c: 40, d: -1 };
const source2 = { e: 50, a: 0 };
const source3 = { f: 3, f: -1 };

console.log(objectAssign(target, source1, source2, source3));
// { a: 0, b: 2, c: 40, d: -1, e: 50, f: -1 }

// ------------------------------------------

const source = { name: 'Peter' };
Object.defineProperty(source, 'age', {
   value: '19',
   writable: true,
   enumerable: false,
   configurable: true
});

const target = {};

console.log(objectAssign(target, source));
// { name: 'Peter' } - 'age' doesn't appear since non-enumerable

// ------------------------------------------

const target = { name: 'Peter', age: '19' };
Object.defineProperty(target, 'name', {
   value: 'Parker',
   writable: false,
   enumerable: true,
   configurable: true
});
```

```
console.log(target);
// { name: 'Parker', age: '19' }

const source = { name: 'Spiderman', age: '19' };

console.log(objectAssign(target, source));
// Error: Property name is not writable to target

// ----------------------------------------

const target = { a: 1, b: 2 };
const source1 = { c: 30, b: -1 };
const source2 = null;
const source3 = NaN;
const source4 = { d: 100 };

console.log(objectAssign(target, source1, source2, source3,
source4));
// { a: 1, b: -1, c: 30, d: 100 }

// ----------------------------------------

const target = { name: 'Spiderman', age: '19' };

const source = { age: '29' };
Object.defineProperty(source, Symbol('name'), {
  value: 'Peter',
  writable: true,
  enumerable: true,
  configurable: true
});

console.log(source);
// { age: '29', [Symbol(name)]: 'Peter' }
```

```
console.log(objectAssign(target, source));
// { name: 'Spiderman', age: '29', [Symbol(name)]: 'Peter' }

// ----------------------------------------

const target = { name: 'Spiderman', age: '19' };

// Symbol(name) property set to false
const source = { age: '29' };
Object.defineProperty(source, Symbol('name'), {
  value: 'Peter',
  writable: true,
  enumerable: false,
  configurable: true
});

console.log(source);
// { age: '29' }

console.log(objectAssign(target, source));
// { name: 'Spiderman', age: '29' }
```

Implement Custom JSON.stringify

Problem Statement

Implement a function `stringify()` which is a polyfill of the built-in `JSON.stringify()`. And you should not use the built-in function directly for the problem, instead write your own version.

`JSON.stringify()` method converts a Javascript value or Object to a JSON string version which can then later also be parsed back to a Javascript value or object using the `JSON.parse()` method.

The idea of JSON.stringify is to serialize the data and JSON.parse to de-serialize it.

Use Case

We might have seen a lot of use cases of using it in our day-to-day development work, where we use it for Deep cloning objects, or using the stringified version to send as data over the network.

Syntax

```
const obj = { name: 'Peter', age: 29 };
```

```
console.log(JSON.stringify(obj));
// {"name":"Peter","age":29}
```

Example

```
const obj = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
   address: {
       city: 'New york',
       state: 'NY'
   }
};

console.log(stringify(obj));
//{"name":"Peter","age":29,"spiderman":true,"movies":["Spiderman
","Amazing Spiderman","Far From Home"]}
```

Implementation

Let's get lookout at the implementation details now,

We just need to understand the representation of different data types.

That is, JSON.stringify supports some following data types:

- ➔ Objects { ... }
- ➔ Arrays [...]
- ➔ Primitives - Strings (""), numbers, booleans

For primitives we return the value as it is.

For Objects / Arrays, since it can be nested we need to process each property or value recursively to get the stringified version for it.

The stringify method accepts a JavaScript value as input and returns the stringified / serialized JSON version as output.

```
function stringify(value) {
   // Handling for string values
   if (typeof value === 'string') {
       // We need to explicitly wrap in double-quotes as per the
standard
       return `"${value}"`;
   }

   // Handling for valid primitives
   if (typeof value === 'number' || typeof value === 'boolean')
{
       // return value as it is in the string representation
       return `${value}`;
   }

   // Handling for Arrays
   if (typeof value === 'object' && Array.isArray(value)) {
       // Since array can be deeply nested with values, process
each element in array recursively
       const stringifiedResult = [];

       value.forEach(val => {
           stringifiedResult.push(stringify(val));
```

```
        })

        // return stringified result in array representation
using square brackets `[]`
        return '[' + stringifiedResult.join(',') + ']';
    }

    // Handling for objects
    if (typeof value === 'object' && value !== null &&
!Array.isArray(value)) {
        // Since object can be deeply nested with values, process
each property (enumerable) of object recursively
        const stringifiedResult = [];

        const keys = Object.keys(value);

        for (const key of keys) {
            const result = stringify(value[key]);
            const stringifiedFormat = `"${key}":${result}`;
            stringifiedResult.push(stringifiedFormat);
        }

        // return stringified result in object literal
representation using curly braces `{}`
        return '{' + stringifiedResult.join(',') + '}';
    }
}
```

Great, so till now we have been able to support the basic data types.

The next step is to add support for more range of data types which the original JSON.stringify supports like,

➔ Support for Nullable values like null, Infinity, NaN.
➔ Support for Ignorable values like function definitions, undefined, Symbols values in Objects.
➔ Support for Date Objects to represent in the ISOString format.
➔ Handle nullable and ignorable values in arrays.
➔ Since, JSON.stringify doesn't support BigInt type, we also need to throw an Error.

So we can create utility functions to collect these nullable types and ignorable types.

```
const isTypeNullCategory = (value) => {
   if (value === null && typeof value === 'object') return true;
   if (typeof value === 'number' && Number.isNaN(value)) return
true;
   if (typeof value === 'number' && !Number.isFinite(value))
return true;
}

const isTypeIgnorableCategory = (value) => {
   if (typeof value === 'symbol') return true;
   if (value === undefined || typeof value === 'undefined')
return true;
   if (typeof value === 'function') return true;
}
```

Now we're all set to move to the final implementation in code and have all our cases handled.

```
function stringify(value) {
```

```
    // Create utility funcs to categorize types
    // 1. Nullable types
    // 2. Ignorable types

    const isTypeNullCategory = (value) => {
        if (value === null && typeof value === 'object') return
true;
        if (typeof value === 'number' && Number.isNaN(value))
return true;
        if (typeof value === 'number' && !Number.isFinite(value))
return true;
    }

    const isTypeIgnorableCategory = (value) => {
        if (typeof value === 'symbol') return true;
        if (value === undefined || typeof value === 'undefined')
return true;
        if (typeof value === 'function') return true;
    }

    if (isTypeNullCategory(value)) {
        // return stringified version of `null`
        return `${null}`;
    }

    if (isTypeIgnorableCategory(value)) {
        // In this category we return `undefined` directly
        return undefined;
    }

    // Handling for Date objects
    if (typeof value === 'object' && value !== null && value
instanceof Date) {
        // return Date in the form of ISO string and wrap in
double quotes
        return `"${value.toISOString()}"`;
```

```
    }

    // Handling for BigInt
    if (typeof value === 'bigint') {
        // BigInt type has no handling so we should throw an
Error
        throw new Error('Do not know how to serialize a BigInt in
stringified format');
    }

    // Handling for string values
    if (typeof value === 'string') {
        // We need to explicitly wrap in double-quotes as per the
standard
        return `"${value}"`;
    }

    // Handling for valid primitives
    if (typeof value === 'number' || typeof value === 'boolean')
{
        // return value as it is in the string representation
        return `${value}`;
    }

    // Handling for Arrays
    if (typeof value === 'object' && Array.isArray(value)) {
        // Since array can be deeply nested with values, process
each element in array recursively
        const stringifiedResult = [];

        value.forEach(val => {
            // if val falls under nullable category type, we just
process it as a `null` value
            if (isTypeNullCategory(val) ||
isTypeIgnorableCategory(val)) {
                stringifiedResult.push(stringify(null));
```

```
            }
            else {
                stringifiedResult.push(stringify(val));
            }
        })

        // return stringified result in array representation
using square brackets `[]`
        return '[' + stringifiedResult.join(',') + ']';
    }

    // Handling for objects
    if (typeof value === 'object' && value !== null &&
!Array.isArray(value)) {
        // Since object can be deeply nested with values, process
each property (enumerable) of object recursively
        const stringifiedResult = [];

        const keys = Object.keys(value);

        for (const key of keys) {
            const val = value[key];
            // We'll ignore to process the properties whose
values fall under ignorable category
            if (!isTypeIgnorableCategory(val)) {
                const result = stringify(val);
                const stringifiedFormat = `"${key}":${result}`;
                stringifiedResult.push(stringifiedFormat);
            }
        }

        // return stringified result in object literal
representation using curly braces `{}`
        return '{' + stringifiedResult.join(',') + '}';
    }
}
```

Test Case

Now that our implementation is done we will add test cases to also compare the results of our custom `stringify()` with the original `JSON.stringify()`.

```
// Number
const testcase1 = 2024;

console.log(stringify(testcase1));
// 2024

console.log(stringify(testcase1) === JSON.stringify(testcase1));
// true

// ----------------------------------------

// String
const testcase2 = 'Test';

console.log(stringify(testcase2));
// "Test"

console.log(stringify(testcase2) === JSON.stringify(testcase2));
// true

// ----------------------------------------

// String
const testcase3 = false;

console.log(stringify(testcase3));
// false
```

```
console.log(stringify(testcase3) === JSON.stringify(testcase3));
// true

// ----------------------------------------

// Object
const testcase4 = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
};

console.log(stringify(testcase4));
//
{"name":"Peter","age":29,"spiderman":true,"movies":["Spiderman",
"Amazing Spiderman","Far From Home"]}

console.log(stringify(testcase4) === JSON.stringify(testcase4));
// true


// ----------------------------------------

// Nested Objects
const testcase5 = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
   address: {
       city: 'New york',
       state: 'NY'
   }
};
```

```
console.log(stringify(testcase5));
//
{"name":"Peter","age":29,"spiderman":true,"movies":["Spiderman",
"Amazing Spiderman","Far From Home"],"address":{"city":"New
york","state":"NY"}}

console.log(stringify(testcase5) === JSON.stringify(testcase5));
// true

// ----------------------------------------

// Arrays
const testcase6 = [1, 2, 3, null];

console.log(stringify(testcase6));
// [1,2,3,null]

console.log(stringify(testcase6) === JSON.stringify(testcase6));
// true

// ----------------------------------------

// Nested Arrays
const testcase7 = [1, 2, Symbol(), 4, [undefined, 8, 9, [10, ()
=> {}]]];

console.log(stringify(testcase7));
// [1,2,null,4,[null,8,9,[10,null]]]

console.log(stringify(testcase7) === JSON.stringify(testcase7));
// true

// ----------------------------------------

// Nested Arrays + Objects + Date type + Nullable types
```

```
const testcase8 = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   arr: [1, 2, Symbol(), 4, [undefined, 8, 9, [10, () => {}, new
Date()]]],
   address: {
       city: 'New york',
       state: 'NY'
   }
};

console.log(stringify(testcase8));
//
{"name":"Peter","age":29,"spiderman":true,"arr":[1,2,null,4,[nul
l,8,9,[10,null,"2024-01-07T12:18:51.802Z"]]],"address":{"city":"
New york","state":"NY"}}

console.log(stringify(testcase8) === JSON.stringify(testcase8));
// true

// ----------------------------------------

// Nested Arrays + Objects + Date type + Nullable types +
Ignorable types

const testcase9 = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   arr: [1, 2, Symbol(), 4, [undefined, 8, 9, [10, () => {},
Infinity, NaN]]],
   address: {
       city: 'New york',
       state: 'NY',
       landmark: undefined,
```

```
        getArea() {}
    },
    birthDate: new Date()
};

console.log(stringify(testcase9));
//
{"name":"Peter","age":29,"spiderman":true,"arr":[1,2,null,4,[nul
l,8,9,[10,null,null,null]]],"address":{"city":"New
york","state":"NY"},"birthDate":"2024-01-07T12:23:54.764Z"}

console.log(stringify(testcase9) === JSON.stringify(testcase9));
// true

// ----------------------------------------

const testcase10 = () => {};

console.log(stringify(testcase10));
// undefined

console.log(stringify(testcase10) ===
JSON.stringify(testcase10));
// true

// ----------------------------------------

const testcase11 = undefined;

console.log(stringify(testcase11));
// undefined

console.log(stringify(testcase11) ===
JSON.stringify(testcase11));
// true
```

```
// ----------------------------------------

const testcase12 = Infinity;

console.log(stringify(testcase12));
// null

console.log(stringify(testcase12) ===
JSON.stringify(testcase12));
// true

// ----------------------------------------

const testcase13 = BigInt(100000);

console.log(stringify(testcase13));
// [Error]: Do not know how to serialize a BigInt in stringified
format
```

Implement Custom JSON.parse

Problem Statement

Implement a function `parseJSON()` which is a polyfill of the built-in `JSON.parse()`. And you should not use the built-in function directly for the problem, instead write your own version.

`JSON.parse()` method takes a valid JSON string (serialized or stringified version) as input and converts it to a valid Javascript value.

The idea of JSON.parse is to deserialize the serialized or stringified data back to the Javascript value as represented in the JSON string.

Example

```
const obj = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
   address: {
       city: 'New york',
       state: 'NY'
   }
}
```

```
const stringifiedObj = JSON.stringify(obj);
//
{"name":"Peter","age":29,"spiderman":true,"movies":["Spiderman",
"Amazing Spiderman","Far From Home"],"address":{"city":"New
york","state":"NY"}}

console.log(parseJSON(stringifiedObj));
/*
{
 name: 'Peter',
 age: 29,
 spiderman: true,
 movies: [ 'Spiderman', 'Amazing Spiderman', 'Far From Home' ],
 address: { city: 'New york', state: 'NY' }
}
*/
```

Approach

Before we approach solving the problem we need to understand some semantics of a JSON.

Basically the structure of how a valid JSON value would appear.

Semantics? What do I mean by those?
You remember in our previous question of JSON.stringify we defined a structure for each javascript value.

That is, let's say we have an object whose structure was defined to be something like this {...} within curly braces. To elaborate more it'd be something like this,

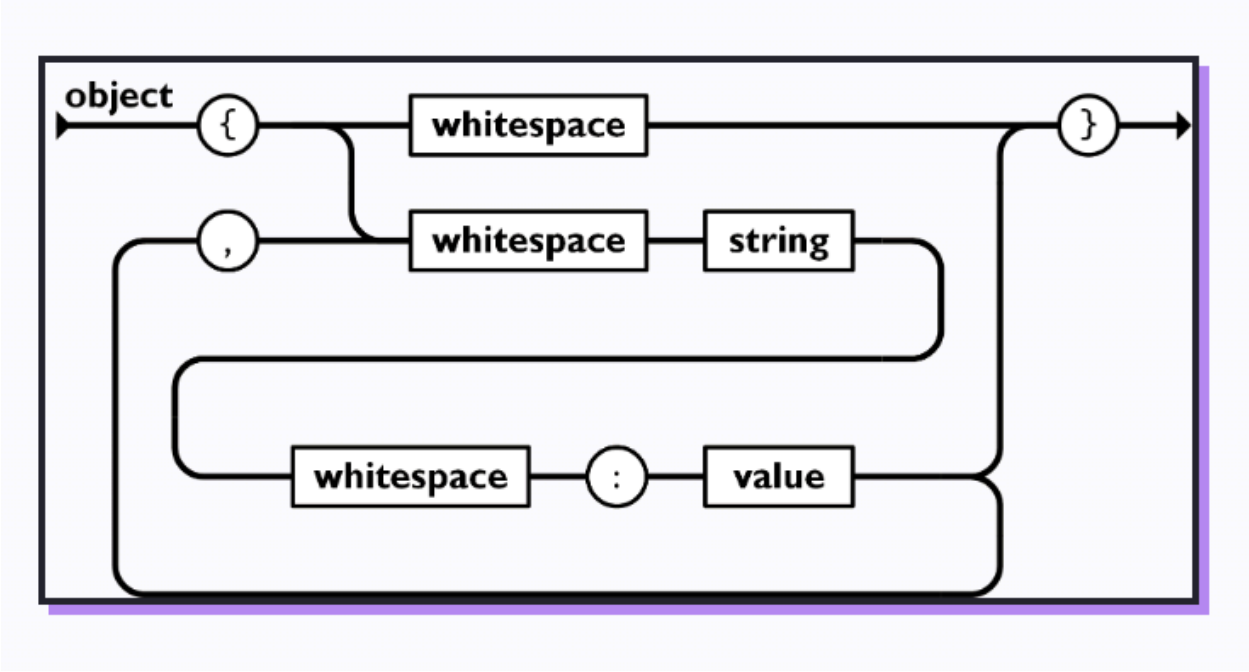
`{ key1 : value1, key2: value2 }`

Let's say we had an object whose structure was defined to be something like this [...] within square brackets. To elaborate more it'd be something like this,

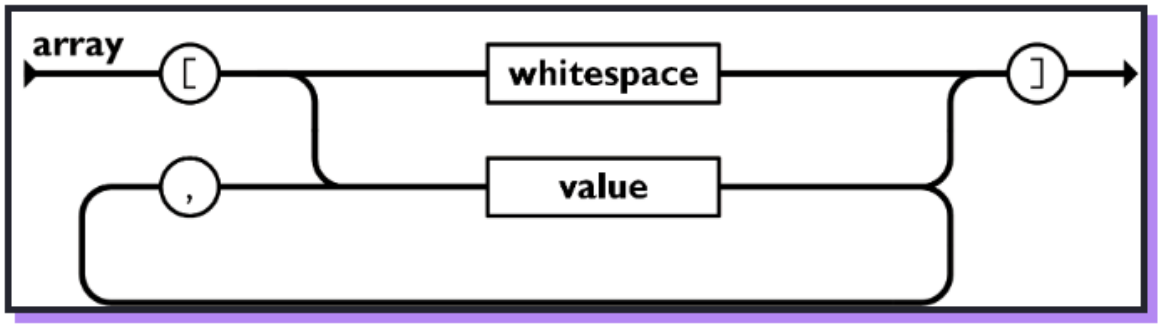
`[ value1, value2 ]`

Let's now understand this same semantics in a diagrammatic or visual fashion,

Object Semantics:



Array Semantics:



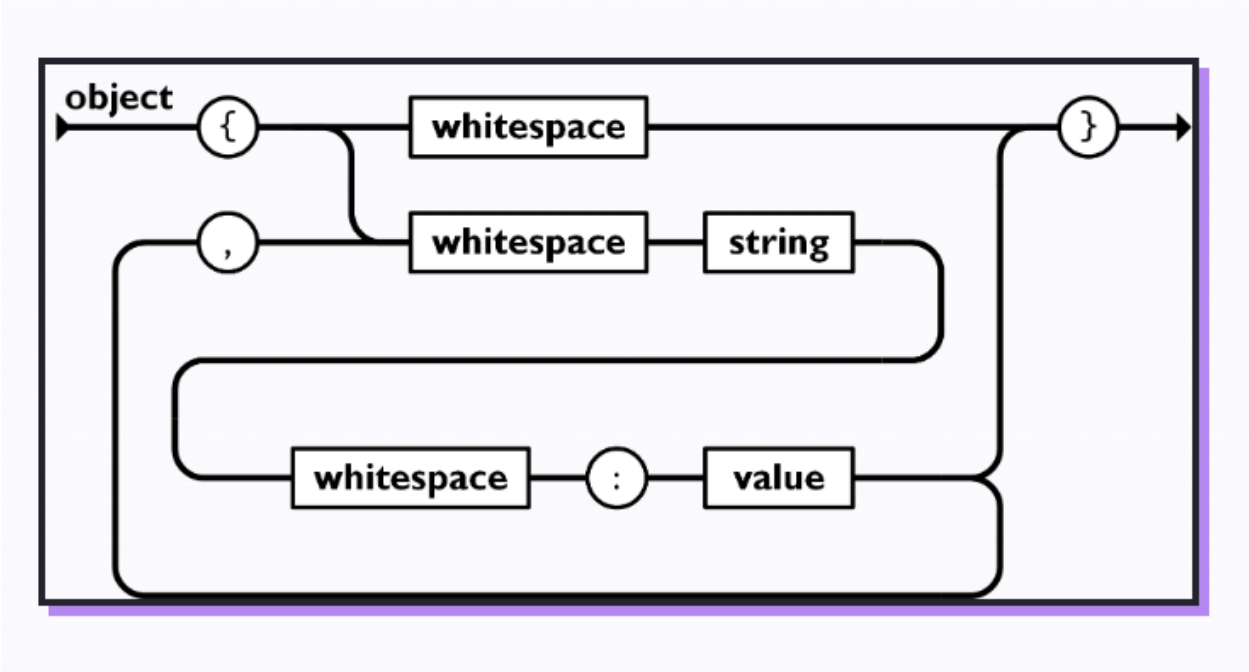
Regarding other types like primitive data types we can comparatively handle it easily while implementing the code.

Implementation

Now that having the semantics in mind, you'll be easily able to relate how we can simply break down our implementation logic by just following those lines in above the diagram.

You can also consider understanding more about the semantics from the specifications as well. But in any case, we'll still cover things end-to-end in a simplified fashion below.

Cool, so let's begin with the implementation. Let's start by understanding the semantics of an **"Object"** first.



So, following the lines of the diagram, starting from the left we have an open curly braces `{` and then we have two options to follow from here:

➔`whitespace` → `}`
➔`whitespace` → `string` → `whitespace` → `:` → `value` → `}`

Also, when we reach a "value", we'd again have two options:

➔`value` → `}`
➔`value` → `,` → `whitespace` → … → `value`

Note: `whitespace`are nothing but the characters we would usually add when we want some indentation to improve readability. Characters like empty spaces ` `, new lines `\n`, tab spaces `\t` fall under the `whitespace` category.

We declare a function `parseJSON()` which will take a JSON string `str` as input.

```
function parseJSON(str) {
   // We initialize this `i=0` to keep track of the current char we are on while parsing
   // We will end parsing as soon as `i` comes to end of the input string `str`
   let i = 0;
}
```

Now we look at implementing the logic for just parsing an Object exactly based on the semantics as defined above :

```
function parseJSON(str) {
   let i = 0;

   function parseObject() {
       // if ith char starts with `{`, it's an object and continue processing until we find closing `}`
       if (str[i] === '{') {
           // move to the next character of `{` to continue parsing
           i++;

           // whitespaces are very common, and all we need to do is ignore or skip them
           skipWhitespace();

           // If str[i] is not `}` then we take the path of `whitespace` -> `string` -> `whitespace` -> ':' -> `value` -> ...
           while (str[i] !== '}') {
```

```
                // Now we're looking for a 'key' of the object as
per the semantics
                // A key would be a string, so we parse string
                const key = parseString();
                skipWhitespace();

                // We expect a colon `:` now as per the
semantics, so accumulate it
                eatColon();
                skipWhitespace();

                // Now we're looking for 'value' of the object as
per the semantics
                // But the value can be anything - boolean,
string, object, array, null
                // So we re-use parseValue and call it
recursively to the parse value deeply
                const value = parseValue();
            }
        }
    }
}
```

Note: Don’t worry about the unimplemented functions used in above code like `skipWhitespace()` or `eatColon()` or `parseString()` or `parseValue()`. We’ll implement it soon, for now just understand that the functions does its work. Focus on understanding the semantics of parsing the Object from the above code.

Also we have some naming conventions for those functions which means :

➔parseSomething() - which will parse something based on

the value and return the parsed value.

➔ eatSomething() - when we expect some characters to be there, but we won't actually use those characters.

➔ skipSomething() - when we expect to just skip over some characters.

Now, an object can have `n` number of key-value pairs which would be separated by commas `,`

And the fact is we'll encounter a comma only after parsing the first key-value pair, that is only in the second loop.

Also we would need to accumulate all of those key-value pairs in an object and then return that new object.

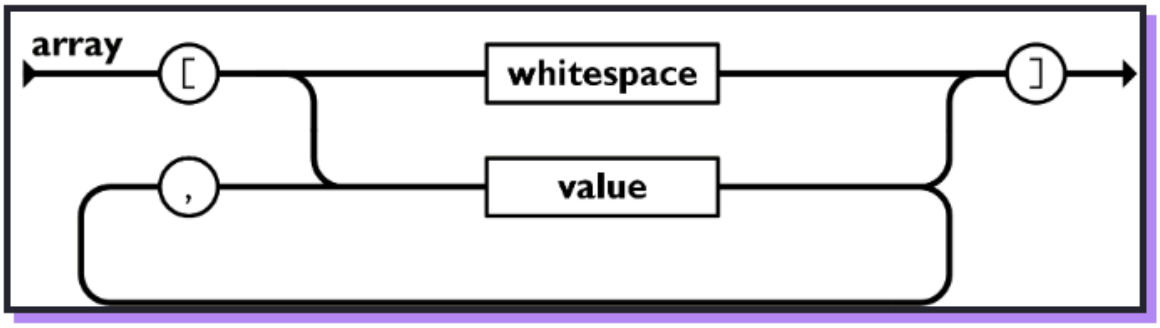
Let's look on how do we handle that :

```
function parseJSON(str) {
   let i = 0;

   function parseObject() {
       if (str[i] === '{') {
           i++;
           skipWhitespace();

           // `result` object to which we'll add the parsed
`key:value`
           const result = {};

           // We use `initial` to start accumulating for commas
`,` from second loop
```

```
            let initial = true;

            while (str[i] !== '}') {
                // Except for the 1st or initial time, we expect
a comma `,` as per the semantics, so accumulate it
                // Example Object - { "a" : "Hello", "b" :
"World", "c" : true }
                if (!initial) {
                    // There can be whitespaces before and after
commas `,` - so skip them
                    skipWhitespace();
                    eatComma();
                    skipWhitespace();
                }

                const key = parseString();
                skipWhitespace();

                eatColon();
                skipWhitespace();

                const value = parseValue();

                result[key] = value;

                // toggle `initial` to false so that we can
accumulate upcoming commas after each property of object
                initial = false;
            }

            // move to the next character of `}` to continue
parsing
            i++;

            // Return the parsed object
            return result;
```

```
        }
    }
}
```

Now that we're done with parsing an Object, let's move further on how we can parse an **"Array"** following the same semantics from the below diagram.

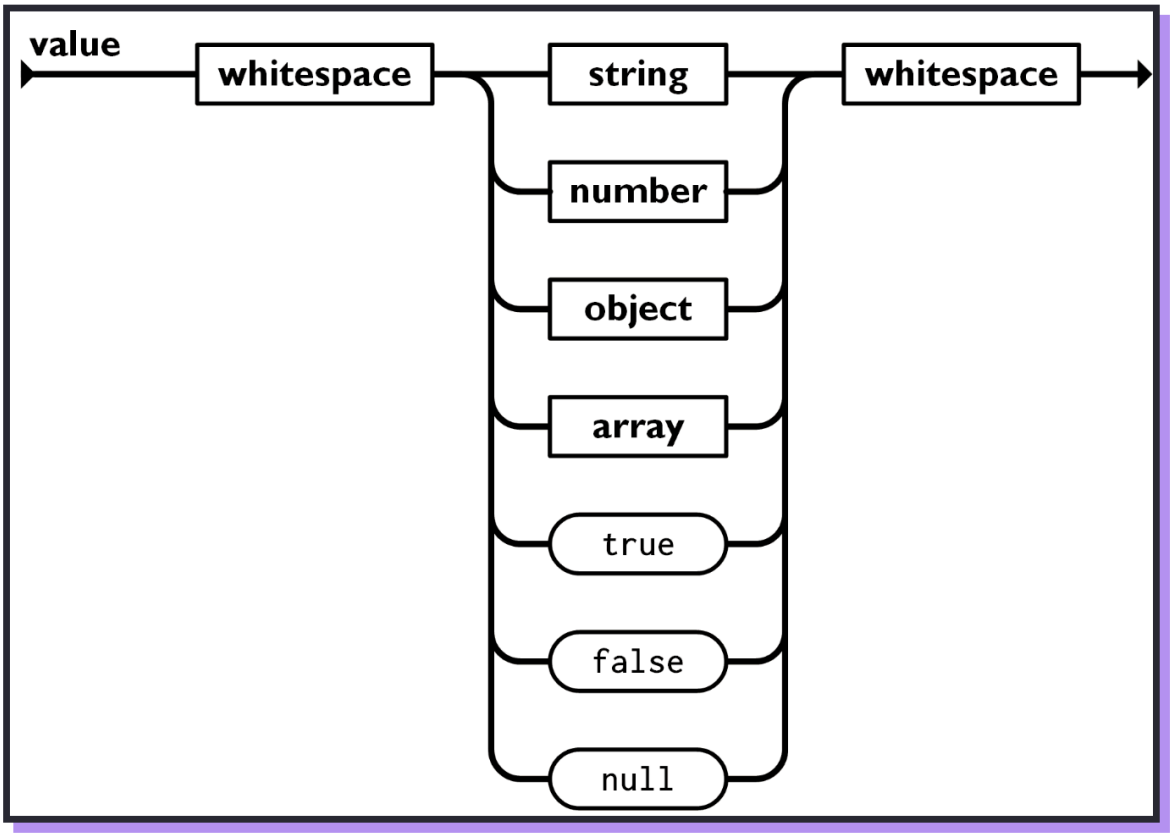



```
function parseJSON(str) {
    // ...
    function parseArray() {
        if (str[i] === '[') {
            // move to the next character of `[` to continue
parsing
            i++;
            skipWhitespace();

            const result = [];

            let initial = true;

            while(str[i] !== ']') {
                // Except for the 1st or initial time, we expect
a comma `,` as per the semantics, so accumulate it
```

```
                // Example Array - [ "Hello", "World", 100 ]
                if (!initial) {
                    // There can be whitespaces before and after
commas `,` - so skip them
                    skipWhitespace();
                    eatComma();
                    skipWhitespace();
                }

                const value = parseValue();
                result.push(value);

                initial = false;
            }

            // move to the next character of `]` to continue
parsing
            i++;

            return result;
        }
    }
}
```

Now comes the most important part of our implementation that is, parsing a **"value".**

Basically we need to know that a value can be anything like strings, numbers, objects, arrays, null, boolean (true/false). Let’s understand this from the below diagram :




```
function parseJSON(str) {
   // ...
   function parseValue() {
       skipWhitespace();

       // We try out all possibilities of data types to parse
the correct type of value
       const value = (
           parseString() ??
           parseNumber() ??
           parseObject() ??
           parseArray() ??
           parseOtherPrimitives() // parses null, true, false
```

```
        );

        skipWhitespace();

        return value;
    }
}
```

We have again declared a few utility functions which does it's work in returning the respective parsed value.

The `??` symbol used is the nullish coalescing operator.

Now let's move on to implement the fully functional code where we have defined all of the utility functions :

```
function parseJSON(str) {
    // We initialize this `i=0` to keep track of the current char we are on while parsing
    let i = 0;

    // Just call `parseValue` it will take care of parsing the JSON deeply at all levels for the right data type
    return parseValue();

    function parseValue() {
        skipWhitespace();

        // We try out all possibilities of data types to parse the correct type of value
        const value = (
            parseString() ??
            parseNumber() ??
```

```
            parseObject() ??
            parseArray() ??
            parseOtherPrimitives()
        );

        skipWhitespace();

        return value;
    }

    function parseObject() {
        // if ith char starts with `{`, it's an object and
continue processing until we find closing `}`
        if (str[i] === '{') {
            // move to the next character of `{` to continue
parsing
            i++;

            // whitespaces are very common, and all we need to do
is ignore or skip them
            skipWhitespace();

            // `result` object to which we'll add the parsed
`key:value`
            const result = {};

            // We use `initial` to start accumulating for commas
`,` from second loop
            let initial = true;

            // If str[i] is not `}` then we take the path of
`whitespace` -> `string` -> `whitespace` -> ':' -> `value` ->
...
            while (str[i] !== '}') {
                // Except for the 1st or initial time, we expect
a comma `,` as per the semantics, so accumulate it
```

```
                // Example Object - { "a" : "Hello", "b" :
"World", "c" : true }
                if (!initial) {
                    // There can be whitespaces before and after
commas `,` - so skip them
                    skipWhitespace();
                    eatComma();
                    skipWhitespace();
                }

                // Now we're looking for a 'key' of the object as
per the semantics
                // A key would be a string, so we parse string
                const key = parseString();
                skipWhitespace();

                // We expect a colon `:` now as per the
semantics, so accumulate it
                eatColon();
                skipWhitespace();

                // Now we're looking for 'value' of the object as
per the semantics
                // But the value can be anything - boolean,
string, object, array, null
                // So we re-use parseValue and call it
recursively to the parse value deeply
                const value = parseValue();

                result[key] = value;

                // toggle `initial` to false so that we can
accumulate upcoming commas after each property of object
                initial = false;
            }
```

```
            // move to the next character of `}` to continue
parsing
            i++;

            // Return the parsed object
            return result;
        }
    }

    function parseArray() {
        // if ith char starts with `[`, it's an Array and
continue processing until we find closing `]`
        if (str[i] === '[') {
            // move to the next character of `[` to continue
parsing
            i++;

            // whitespaces are very common, and all we need to do
is ignore or skip them
            skipWhitespace();

            // `result` array to which we'll need to add the
parsed `value`
            const result = [];

            // We use `initial` to start accumulating for commas
`,` from second loop
            let initial = true;

            while (str[i] !== ']') {
                // Except for the 1st or initial time, we expect
a comma `,` as per the semantics, so accumulate it
                // Example Array - [ "Hello", "World", 100 ]
                if (!initial) {
                    // There can be whitespaces before and after
commas `,` - so skip them
```

```
                skipWhitespace();
                eatComma();
                skipWhitespace();
            }

            const value = parseValue();
            result.push(value);

            // toggle `initial` to false so that we can
accumulate upcoming commas after each value in array
            initial = false;
        }

        // move to the next character of `]` to continue
parsing
        i++;

        // Return the parsed array
        return result;
    }
}

function parseString() {
    // if ith char starts with `"`, it's an string and
continue processing until we find closing `"`
    if (str[i] === '"') {
        // move to the next character of `"` to continue
parsing
        i++;

        // `result` string to which we'll need to add each
chars
        let result = "";

        while (str[i] !== '"') {
            result += str[i];
```

```
                i++;
            }

            // move to the next character of `"` to continue
parsing
            i++;

            // Return the parsed string
            return result;
        }
    }

    function parseNumber() {
        let start = i;

        //  We traverse until we keep finding valid number values
        while (str[i] >= '0' && str[i] <= '9') {
            i++;
        }

        if (i > start) {
            return Number(str.slice(start, i));
        }
    }

    function parseOtherPrimitives() {
        let result;

        // We try to match exactly with other primitive data
types
        if (str.slice(i, i + 4) === 'true') {
            result = true;
            i += 4;
        }
        else if (str.slice(i, i + 5) === 'false') {
            result = false;
```

```
            i += 5;
        }
        else if (str.slice(i, i + 4) === 'null') {
            result = null;
            i += 4;
        }

        return result;
    }

    function eatComma() {
        if (str[i] !== ',') {
            throw new Error('Expected a comma `,` but got
something else');
        }

        i++;
    }

    function eatColon() {
        if (str[i] !== ':') {
            throw new Error('Expected a colon `:` but got
something else');
        }

        i++;
    }

    function skipWhitespace() {
        // We can often see these spaces, and we should just skip
over it, to parse the original value further
        // " " - empty space
        // "\n" - new line
        // "\t" - tab space
        while (str[i] === " " || str[i] === "\n" || str[i] ===
"\t") i++;
```

```
    }
}
```

And that's it we're done with our custom JSON.parse which parseJSON.

Test Cases

```
const input1 = {
    name: 'Peter',
    age: 29,
    spiderman: true,
    movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
    address: {
        city: 'New york',
        state: 'NY'
    }
}

const stringifiedJson1 = JSON.stringify(input1);
//
'{"name":"Peter","age":29,"spiderman":true,"movies":["Spiderman"
,"Amazing Spiderman","Far From Home"],"address":{"city":"New
york","state":"NY"}}'

console.log(parseJSON(stringifiedJson1));
/*
{
 name: 'Peter',
 age: 29,
 spiderman: true,
 movies: [ 'Spiderman', 'Amazing Spiderman', 'Far From Home' ],
 address: { city: 'New york', state: 'NY' }
}
```

```
*/

// ---------------------------------------

const input2 = {
   a: 'Hello',
   b: true,
   c: ['Yo', 20, null, { name: 'Kuch bhi', age: 30 }],
   def: { ghi: 'jkl', mno: null, pqrs: { tuv: [7, null, false] }
},
   g: { h: { i: { j: { k: 'Got here!!' } } } }
}

const stringifiedJson2 = JSON.stringify(input2);
// '{"a":"Hello","b":true,"c":["Yo",20,null,{"name":"Kuch
bhi","age":30}],"def":{"ghi":"jkl","mno":null,"pqrs":{"tuv":[7,n
ull,false]}},"g":{"h":{"i":{"j":{"k":"Got here!!"}}}}}'

console.log(parseJSON(stringifiedJson2));
/*
{
 a: 'Hello',
 b: true,
 c: [ 'Yo', 20, null, { name: 'Kuch bhi', age: 30 } ],
 def: { ghi: 'jkl', mno: null, pqrs: { tuv: [ 7, null, false ] }
},
 g: { h: { i: { j: { k: 'Got here!!' } } } }
}
*/

// ---------------------------------------

const input3 = [
   null,
   'Hello World',
   {},
```

```
    [
        {
            a: 10,
            b: [1, 2, 3],
            c: false,
            d: ''
        }
    ]
]

const stringifiedJson3 = JSON.stringify(input3);
// '[null,"Hello
World",{},[{"a":10,"b":[1,2,3],"c":false,"d":""}]]'

console.log(parseJSON(stringifiedJson3));
// [ null, 'Hello World', {}, [ { a: 10, b: [ 1, 2, 3 ], c:
false, d: '' } ] ]

// ----------------------------------------

const input4 = {};

const stringifiedJson4 = JSON.stringify(input4);
// '{}'

console.log(parseJSON(stringifiedJson4));
// {}

// ----------------------------------------

const input5 = null;

const stringifiedJson5 = JSON.stringify(input5);
// 'null'

console.log(parseJSON(stringifiedJson5));
```

```
// null

// ----------------------------------------

const input6 = 100;

const stringifiedJson6 = JSON.stringify(input6);
// '100'

console.log(parseJSON(stringifiedJson6));
// 100

// ----------------------------------------

const input7 = {
   a: {
       b: {
           c: {
               d: {
                   e: {
                       f: {

                       }
                   }
               }
           }
       }
   },
};

const stringifiedJson7 = JSON.stringify(input7);
// '{"a":{"b":{"c":{"d":{"e":{"f":{}}}}}}}'

console.log(parseJSON(stringifiedJson7));
// { a: { b: { c: { d: { e: { f: {} } } } } } }
```

```
// ----------------------------------------

// Tescase with a lot of white spaces like empty space, tab
spaces, new lines
const stringifiedJson8 = `{
   "a"     :   {"b":   "Hello World"     },
"c":[  1,2  , 3,4    ]
}`
// The actual representation of above input looks like -
// '{  \n    "a"     :   {"b":  "Hello World"     },\n"c":[  1,2
, 3,4    ]   \n}'

console.log(parseJSON(stringifiedJson8));
// { a: { b: 'Hello World' }, c: [ 1, 2, 3, 4 ] }
```

Implement Custom typeof operator

Problem Statement

Implement a function `getTypeOf()` which is a polyfill for the built-in `typeof` operator. And you should not use the built-in function directly for the problem, instead write your own version.

The thing is Javascript's `typeof`operator is not used as a function call instead follows the below syntax :

```
// Original syntax
typeof data;
```

But in our case we would define a custom function and make it behave exactly like the `typeof` operator.

Now here, we have two challenges that need to be solved.

1. We should return the correct datatype of whatever data we get.
2. We should not use the built-in `typeof` operator of Javascript in our custom function.

Concept

Cool, first things first, let's understand what's wrong with the `typeof` operator. What do I mean in the 1st point by returning the correct datatype?

Javascript's `typeof` operator is a bit like the old car which gets the work done by getting you the type of the data but you have to do some workarounds in some cases.

In general,

```
typeof 100; // number
typeof 'Hello World'; // string
typeof false; // boolean
typeof { name: 'Peter' }; // object
```

It works fine right?

But checkout the below cases,

```
typeof { name: 'Peter' }; // object
typeof [1, 2, 3]; // object
typeof Math; // object
typeof JSON; // object
typeof new FileReader(); // object
typeof new Number(200); // object
typeof new Boolean(true); // object
typeof new String('Hello World'); // object
```

That's where the problem starts. Javascript's `typeof` operator is not able to distinguish between generic objects like {} and

the other built-in types like Array, Date, JSON, Math, Map, Set, RegExp, Arguments, Error and primitive wrapper objects (Number, Boolean, String).

Happens that we have a solution to detect the right datatype.

Every Javascript object has an internal property known as [[class]] - The double square bracket notation represents internal properties used to specify the behavior of Javascript engines.

So, as per the standard, [[class]] is a string value which indicates a specification defined classification of objects.

In simple words, it means each built-in object type has a unique, non-editable, standard-defined value for its [[class]] property for the classification of different types of objects.

But we cannot access the Internal property [[class]] directly, since it cannot be accessed directly. There's a reason for it to be internal.

But we have a workaround which helps us to get the value from it.

`Object.prototype.toString()`

The `toString()` method of Object instances returns a string

representing this object in the following format:

[object [[class]]]

..where [[class]] is the class property of the object.

Means for Arrays we should get [object Array], for Dates we should get [object Date], for JSON we should get [object JSON]..

Let's try by calling the `toString()` directly on the objects see the output :

```
const arr = [1,2,3];
console.log(arr.toString()); // 1,2,3

const date = new Date();
console.log(date.toString()); // Sun Jan 14 2024 16:54:27
GMT+0530 (India Standard Time)

const regex = new RegExp();
console.log(regex.toString()); // /(?:)/
```

But, that’s not what we expected right?

We expected it to be like [object Array], [object Date]..

We need to know one more fact here. That is, the `toString()` is meant to be overridden by the derived objects. And we know that all Objects like Arrays or Date or JSON are derived

from the main Object instance and each of those derived objects has their own way of representing the value from the `toString()` method by overriding it.

Here's an example below to override :

```
function Spiderman(name) {
   this.name = name;
}

const spidey = new Spiderman('Peter Parker');

Spiderman.prototype.toString = function () {
   return `${this.name}`;
};

console.log(spidey.toString());
// Expected output: "Peter Parker"
```

Similarly, each built-in object has its own way of representing by overriding the `toString()` method.

Fortunately, we can use the `call` function to force the generic `toString()` method of the main Object instance upon them..

```
const arr = [1,2,3];
Object.prototype.toString.call(arr);
// '[object Array]'

const date = new Date();
Object.prototype.toString.call(date);
// '[object Date]'
```

```
const regex = new RegExp();
Object.prototype.toString.call(regex);
// '[object RegExp]'

const num = new Number(10);
Object.prototype.toString.call(num);
// '[object Number]'

const str = new String('Hello World');
Object.prototype.toString.call(str);
// '[object String]'

const bool = new Boolean(false);
Object.prototype.toString.call(bool);
// '[object Boolean]'

const map = new Map();
Object.prototype.toString.call(map);
// '[object Map]'

const set = new Set();
Object.prototype.toString.call(set);
// '[object Set]'

const fileReader = new FileReader();
Object.prototype.toString.call(fileReader);
// '[object FileReader]'

const arraybuffer = new ArrayBuffer();
Object.prototype.toString.call(arraybuffer);
// '[object ArrayBuffer]'

const dataView = new DataView(new ArrayBuffer());
Object.prototype.toString.call(dataView);
// '[object DataView]'
```

```
const uint8Arr = new Uint8Array();
Object.prototype.toString.call(uint8Arr);
// '[object Uint8Array]'

const err = new Error();
Object.prototype.toString.call(err);
// '[object Error]'

// What about Objects which don't support constructors to create
new instances like JSON, Math?
// It manages to get the [[class]] property from the passed
instance directly
// Since we know that JSON, Math are already defined Objects
which cannot be changes or instantiated
Object.prototype.toString.call(JSON);
// '[object JSON]'

Object.prototype.toString.call(Math);
// '[object Math]'

// What about primitive types directly instead of using
Primitive Wrapper Objects
// It happens that these primitive will be automatically be
converted to Wrapper Objects first
// That's why you'll see the same exact result
Object.prototype.toString.call(20);
// '[object Number]'

Object.prototype.toString.call('Hello World');
// '[object String]'

Object.prototype.toString.call(null);
// '[object Null]'

Object.prototype.toString.call(undefined);
```

```
// '[object Undefined]'

Object.prototype.toString.call(NaN);
// '[object Number]'
// As per the old standards, it's clearly stated that NaN is
also falls under the `Number` type
```

Cool, now things are working as expected and returning us the exact datatype of the built-in object.

Now all we have to do is extract or strip out only the data type which is Number, Array, Map..

Implementation

```
function getTypeOf(data) {
   const type = Object.prototype.toString.call(data); //
'[object type]'

   // 1. Strip the square brackets []
   // 2. Split by empty space character and get the type alone
at the 1th index
   // 3. Convert it to lowercase, to make it work exactly like
the built-in `typeof` operator
   // 4. Finally return the value
   const datatype = type.slice(1, -1).split('
')[1].toLowerCase();
   return datatype;
}
```

And that’s it, we're done with the implementation.

Test Case

```
const arr = [1,2,3];
getTypeOf(arr);
// 'array'

const date = new Date();
getTypeOf(date);
// 'date'

const regex = new RegExp();
getTypeOf(regex);
// 'regexp'

const bool = new Boolean(false);
getTypeOf(bool);
// 'boolean'

getTypeOf(JSON);
// 'json'

const map = new Map();
getTypeOf(map);
// 'map'

const err = new Error();
getTypeOf(err);
// 'error'

getTypeOf(null);
// 'null'

const arraybuffer = new ArrayBuffer();
getTypeOf(arraybuffer);
// 'arraybuffer'
```

Implement Custom lodash _.get()

Problem Statement

Implement a function `get()` which is your own version of the lodash's `_.get()` method. This method is used to get the value from the object based on the path given as input.

The `get()` method accepts 3 parameters as input - object, path, defaultValue.

- ➔object - the actual object from which the value has to be retrieved
- ➔path - (Array/String) denoting the path to access the property (key) from the object
- ➔defaultValue - An optional value used to be returned from the `get()` method when no property is found in the object based on the path

```
// Syntax
function get(obj, path, defaultValue) {}
```

Example

```
const obj = {
   a: {
       b: 'Hello',
       c: null,
```

```
        d: [1, 2, 'World'],
        e: [
            { name: 'Peter Parker' },
            { work: 'Spiderman' }
        ],
        h: {
            i: {
                j: 'Iron Man',
                k: 'Batman'
            }
        }
    },
    f: {
        g: undefined
    }
}

get(obj, 'a.h.i.j', 'Key Not Found');
// Iron Man

get(obj, 'a.c', 'Key Not Found');
// null

get(obj, 'f.g.h.i', 'Key Not Found');
// Key Not Found

get(obj, ['a', 'e', '1', 'work'], 'Key Not Found');
// Spiderman

get(obj, 'a.d[2]', 'Key Not Found');
// World

get(obj, 'f[g]', 'Key Not Found');
// undefined
```

Note: The path is either a string with dot notations or square bracket notations (applicable for both arrays and objects) as well. Or the path can also be an array of keys passed in array positions.

Implementation

Let's go ahead with implementing the solution :

- ➔ The first thing you have to do is parse the path making it a valid array of keys.
- ➔ Then evaluate the current key for its existence in the object.
- ➔ If all the keys are traversed or processed in the object, return the value, else recursively continue processing to get the value.

The first thing we need to do is to parse the input to a common notation. That is, we don't want square brackets, so replace them with dot notation.

```
function get(obj, path, defaultValue) {
   // if the input is not an array and instead a string
   if (!Array.isArray(path)) {
       // Below 2 lines replaces all the square bracket notation
`[]` with dot notation `.`
       // This makes our job of parsing easy
       // Example : 'a.b[c]' -> 'a.b.c', 'a.b.c[0][1]' ->
'a.b.c.0.1', 'a[1].b.c' -> a.1.b.c
       path = path.replaceAll('[', '.');
```

```
        path = path.replaceAll(']', '');
    }
}
```

Now let’s move to the full implementation :

```
function get(obj, path, defaultValue) {
    // The `obj` passed should be a valid object
    // Note: Array is also an object
    if (obj === null || typeof obj !== 'object') {
        return defaultValue;
    }

    // First step is to replace all of the square bracket
notation [] with dot notation
    // This will work for accessing values from both Objects and
arrays
    let keys = [];
    if (!Array.isArray(path)) {
        path = path.replaceAll('[', '.');
        path = path.replaceAll(']', '');
        keys = path.split('.');
    }
    else {
        keys = path;
    }

    const currKey = keys[0];

    // Means we have processed all the keys in the path, so just
return the value for the key
    if (keys.length === 1) {
        // We use `hasOwnProperty` method to check if a key
exists on the object
```

```
        // Using `obj[currKey]` is not good, since there can be a
falsy value as well like null, undefined, '' (which are
completely valid)
        // So the aim should be to check if the property was
defined on the object or not
        return obj.hasOwnProperty(currKey) ? obj[currKey] :
defaultValue;
    }
    else {
        // Recursively continue traversing the path on the object
to get the value
        if (obj.hasOwnProperty(currKey)) {
            return get(obj[currKey], keys.slice(1),
defaultValue);
        }

        return defaultValue;
    }
}
```

Test Cases

```
const obj = {
    a: {
        b: 'Hello',
        c: null,
        d: [1, 2, 'World'],
        e: [
            { name: 'Peter Parker' },
            { work: 'Spiderman' }
        ],
        h: {
            i: {
                j: 'Iron Man',
```

```
                k: 'Batman'
            }
        }
    },
    f: {
        g: undefined
    }
}

console.log(get(obj, 'a.b', 'Key Not Found'));
// Hello

console.log(get(obj, ['a', 'h', 'i', 'k'], 'Key Not Found'));
// Batman

console.log(get(obj, 'a[b]', 'Key Not Found'));
// Hello

console.log(get(obj, ['a', 'e', '1', 'work'], 'Key Not Found'));
// Spiderman

console.log(get(obj, 'a[d].1', 'Key Not Found'));
// 2

console.log(get(obj, 'a.d.2', 'Key Not Found'));
// World

console.log(get(obj, 'a.d.3', 'Key Not Found'));
// Key Not Found

console.log(get(obj, 'a[d][0]', 'Key Not Found'));
// 1

console.log(get(obj, 'a.e.0.name', 'Key Not Found'));
// Peter Parker
```

```
console.log(get(obj, 'f.g', 'Key Not Found'));
// undefined

console.log(get(obj, 'f.g.h.i.j.k', 'Key Not Found'));
// Key Not Found
```

Implement Custom lodash _.set()

Problem Statement

Implement a function `set()` which is your own version of the lodash's `_.set()` method. This method is used to set the value on the object based on the path passed as input.

The `set()` method accepts 3 parameters as input - object, path, value.

- ➔object - the actual object which needs to be modified and new value has to be set
- ➔path - (Array/String) denoting the path of keys on which value needs to be set
- ➔value - the actual value that needs to be set on the object, by traversing the path of keys

```
// Syntax
function set(obj, path, value) {}
```

Example

```
const obj = {
   a: {
       b: 'Hello',
       e: [{ name: 'Peter Parker' }],
```

```
        h: {
            i: {
                j: 'Iron Man'
            }
        }
    },
    f: {
        g: undefined
    }
}

set(obj, 'a.c', null);
/*
{
 a: { b: 'Hello', e: [ { name: 'Peter Parker' } ], h: { i: { j:
'Iron Man' } } }, c: null },
 f: { g: undefined }
}
*/

set(obj, 'a.d[0]', 1);
/*
{
 a: { b: 'Hello', e: [ { name: 'Peter Parker' } ], h: { i: { j:
'Iron Man' } } }, d: [ 1 ] },
 f: { g: undefined }
}
*/

set(obj, ['a', 'e', '1'], { work: 'Spiderman' });
/*
{
   a: { b: 'Hello', e: [ { name: 'Peter Parker' }, { work:
'Spiderman' } ], h: { i: { j: 'Iron Man' } } },
   f: { g: undefined }
}
```

```
*/
```

Implementation

Let’s go ahead with implementing the solution :

- ➔ The first thing you have to do is parse the path making it a valid array of keys.
- ➔ If processed all the keys and now if only 1 key left, then set the value passed on the object
- ➔ Else if all keys of the path are not traversed yet then check,
  - ◆ If the key is not defined on the input object, then we need to define the currKey on the input object either as an array or object based on the constraints (refer code for example)
  - ◆ If the key is already defined on the input object, then continue traversing the keys of the path recursively until the last key remains and set the value on the input object.

Cool, so now the first thing we need to do is to parse the input to a common notation. That is, we don’t want square brackets, so replace them with dot notation.

```
function set(obj, path, defaultValue) {
   // if the input is not an array and instead a string
```

```
    if (!Array.isArray(path)) {
        // Below 2 lines replaces all the square bracket notation
`[]` with dot notation `.`
        // This makes our job of parsing easy
        // Example : 'a.b[c]' -> 'a.b.c', 'a.b.c[0][1]' ->
'a.b.c.0.1', 'a[1].b.c' -> a.1.b.c
        path = path.replaceAll('[', '.');
        path = path.replaceAll(']', '');
    }
}
```

Now let's move to the full implementation :

```
function set(obj, path, value) {
    // The `obj` passed should be a valid object
    // Note: Array is also an object
    if (obj === null || typeof obj !== 'object') {
        return;
    }

    // First step is to replace all of the square bracket
notation [] with dot notation
    // This will work for accessing values from both Objects and
arrays
    let keys = [];
    if (!Array.isArray(path)) {
        path = path.replaceAll('[', '.');
        path = path.replaceAll(']', '');
        keys = path.split('.');
    }
    else {
        keys = path;
    }
```

```
    const currKey = keys[0];

    // Means we have processed all the keys in the path, so just
set the value for the key
    if (keys.length === 1) {
        // Set the value, since we have traversed full path
        // Note : The below assignment operation works fine for
both arrays and objects
        // Since internally an array is also an object
        obj[currKey] = value;
    }
    else {
        // If `currKey` is not defined yet on `obj`, we need to
define it
        if (!obj.hasOwnProperty(currKey)) {
            // --> There can be 2 possibilities that the currKey
can be either an object (or) array
            // How to decide that? - This depends on the type of
the key for the `nextKey` which we would need to assign
            // If nextKey is a number format, then array, else an
object
            // Example : 'a[1]' -> 'a.1' -> currKey=a, nextKey=1
-> So define array
            // Example : 'a.b' -> currKey=a, nextKey=b -> So
define object

            // Verify if `nextKey` is not a number
            const nextKey = keys[1];
            const num = Number(nextKey);
            // check if is Not a Number
            const isNotANumber = isNaN(num);
            obj[currKey] = (isNotANumber) ? {} : [];
        }

        // Continue processing the path recursively
        set(obj[currKey], keys.slice(1), value);
```

```
    }
}
```

Test Cases

```
const obj = {
    a: {
        b: 'Hello',
        e: [{ name: 'Peter Parker' }],
        h: {
            i: {
                j: 'Iron Man'
            }
        }
    },
    f: {
        g: undefined
    }
}


set(obj, 'a.c', null);
/*
{
 a: { b: 'Hello', e: [ { name: 'Peter Parker' } ], h: { i: { j:
'Iron Man' } } }, c: null },
 f: { g: undefined }
}
*/

set(obj, 'a.d[0]', 1);
/*
{
 a: { b: 'Hello', e: [ { name: 'Peter Parker' } ], h: { i: { j:
```

```
'Iron Man' } }, d: [ 1 ] },
 f: { g: undefined }
}
*/

set(obj, 'a.e.1', { work: 'Spiderman' });
/*
{
   a: { b: 'Hello', e: [ { name: 'Peter Parker' }, { work:
'Spiderman' } ], h: { i: { j: 'Iron Man' } } },
   f: { g: undefined }
}
*/

set(obj, ['a', 'e', '1'], { work: 'Spiderman' });
/*
{
   a: { b: 'Hello', e: [ { name: 'Peter Parker' }, { work:
'Spiderman' } ], h: { i: { j: 'Iron Man' } } },
   f: { g: undefined }
}
*/

set(obj, 'a[e].1', { work: 'Spiderman' });
/*
{
   a: { b: 'Hello', e: [ { name: 'Peter Parker' }, { work:
'Spiderman' } ], h: { i: { j: 'Iron Man' } } },
   f: { g: undefined }
}
*/

set(obj, 'a[h].i[k]', 'Batman');
/*
{
 a: { b: 'Hello', e: [ { name: 'Peter Parker' } ], h: { i: { j:
```

```
'Iron Man', k: 'Batman' } } },
 f: { g: undefined }
}
*/

set(obj, 'f[g]', 'Javascript');
/*
{
 a: { b: 'Hello', e: [ { name: 'Peter Parker' } ], h: { i: { j:
'Iron Man' } } },
 f: { g: 'Javascript' }
}
*/

set(obj, 'f[l][m].n.p[0]', 100);
/*
{
   a: { b: 'Hello', e: [ { name: 'Peter Parker' } ], h: { i: {
j: 'Iron Man' } } },
   f: { g: undefined, l: { m: { n: { p: [ 100 ] } } } } }
}
*/

set(obj, 'f.g[0]', 200);
// Nothing sets since 'f.g' in the object is undefined and we
can't set anything on that
```

Implement Custom lodash _.omit()

Problem Statement

Implement a function `omit()` which is your own version of the lodash's `_.omit()` method. This method is used to omit/delete the last key on the object based on the path passed as input.

The `omit()` method accepts 2 parameters as input - object, path.

- ➔ object - the actual source object on which we need to process the omit logic
- ➔ path - the string denoting the path which holds the property that needs to be omitted

```
// Syntax
function set(obj, path, value) {}
```

Example

```
const obj = {
   a: {
       b: 'Hello',
       c: null,
       d: [1, 2, 'World'],
       e: [
           { name: 'Peter Parker' },
```

```
            { work: 'Spiderman' }
        ],
        h: {
            i: {
                j: 'Iron Man',
                k: 'Batman'
            }
        }
    },
    f: {
        g: undefined
    }
}

omit(obj, 'a.b');
/*
{
 a: {
   c: null,
   d: [ 1, 2, 'World' ],
   e: [ { name: 'Peter Parker' }, { work: 'Spiderman' } ],
   h: { i: { j: 'Iron Man', k: 'Batman' } }
 },
 f: { g: undefined }
}
*/

omit(obj, 'a[d][1]');
/*
{
 a: {
   b: 'Hello',
   c: null,
   d: [ 1, <empty>, 'World' ],
   e: [ { name: 'Peter Parker' }, { work: 'Spiderman' } ],
   h: { i: { j: 'Iron Man', k: 'Batman' } }
```

```
 },
 f: { g: undefined }
}
*/
```

Implementation

Let's go ahead with implementing the solution :

- ➔ The first thing you have to do is parse the path making it a valid array of keys.
- ➔ If processed all the keys and now if only 1 key left, then omit/delete the element.
- ➔ Else if all keys of the path are not traversed yet then continue processing recursively until the key which needs to be omitted is found.

Cool, so now the first thing we need to do is to parse the input to a common notation. That is, we don't want square brackets, so replace them with dot notation.

```
function omit(obj, path, defaultValue) {
   // if the input is not an array and instead a string
   if (!Array.isArray(path)) {
       // Below 2 lines replaces all the square bracket notation
`[]` with dot notation `.`
       // This makes our job of parsing easy
       // Example : 'a.b[c]' -> 'a.b.c', 'a.b.c[0][1]' ->
'a.b.c.0.1', 'a[1].b.c' -> a.1.b.c
       path = path.replaceAll('[', '.');
```

```
        path = path.replaceAll(']', '');
    }
}
```

Now let’s move to the full implementation :

```
function omit(obj, path) {
    // The `obj` passed should be a valid object
    // Note: Array is also an object
    if (obj === null || typeof obj !== 'object') {
        return;
    }

    // First step is to replace all of the square bracket
notation [] with dot notation
    // This will work for accessing values from both Objects and
arrays
    let keys = []
    if (!Array.isArray(path)) {
        path = path.replaceAll('[', '.');
        path = path.replaceAll(']', '');
        keys = path.split('.');
    }
    else {
        keys = path;
    }

    const currKey = keys[0];

    // Means we have processed all the keys in the path, so just
set the value for the key
    if (keys.length === 1) {
        if (Array.isArray(obj[currKey])) {
            // For arrays, we need to delete the element at index
```

```
using the splice method
            obj[currKey].splice(currKey, 1);
        }
        else {
            // For objects, we can use the delete keyword to
remove the property
            delete obj[currKey];
        }
    }
    else {
        // Else we continue traversing the path recursively until
we find the last key which needs to be omitted
        omit(obj[currKey], keys.slice(1));
    }
}
```

Test Cases

```
const obj = {
    a: {
        b: 'Hello',
        c: null,
        d: [1, 2, 'World'],
        e: [
            { name: 'Peter Parker' },
            { work: 'Spiderman' }
        ],
        h: {
            i: {
                j: 'Iron Man',
                k: 'Batman'
            }
        }
    },
```

```
   f: {
       g: undefined
   }
}

omit(obj, 'a.b');
/*
{
 a: {
   c: null,
   d: [ 1, 2, 'World' ],
   e: [ { name: 'Peter Parker' }, { work: 'Spiderman' } ],
   h: { i: { j: 'Iron Man', k: 'Batman' } }
 },
 f: { g: undefined }
}
*/

omit(obj, 'a[d][1]');
/*
{
 a: {
   b: 'Hello',
   c: null,
   d: [ 1, <empty>, 'World' ],
   e: [ { name: 'Peter Parker' }, { work: 'Spiderman' } ],
   h: { i: { j: 'Iron Man', k: 'Batman' } }
 },
 f: { g: undefined }
}
*/

omit(obj, 'a.h.i.k');
/*
{
 a: {
```

```
   b: 'Hello',
   c: null,
   d: [ 1, 2, 'World' ],
   e: [ { name: 'Peter Parker' }, { work: 'Spiderman' } ],
   h: { i: { j: 'Iron Man' } }
 },
 f: { g: undefined }
}
*/

omit(obj, 'f[g].l.m.n');
/*
Since `f.g` in the object is `undefined` which is not valid so
we won't process any further any return right away
{
 a: {
   b: 'Hello',
   c: null,
   d: [ 1, 2, 'World' ],
   e: [ { name: 'Peter Parker' }, { work: 'Spiderman' } ],
   h: { i: { j: 'Iron Man', k: 'Batman' } }
 },
 f: { g: undefined }
}
*/
```

Implement Custom String Tokenizer

Problem Statement

Implement a custom functionality of `tokenizer()` which processes your input string (basically a math expression in our case) into multiple tokens and gets those individual tokens.

By definition -

When given a character sequence (input string), tokenization is the task of chopping it up into pieces called tokens.

In general, Tokenizers are very important in parsing because it lets us group together characters into meaningful symbols which we refer to as tokens.

A more relatable example would be when building compilers and interpreters for programming languages, it's important to use tokenizers to create tokens for common keywords such as `if`, `else`, `var`, etc

As for our problem statement, we need to tokenize math expressions.

For example, let’s say we have a math expression `10 + 20`

Where each of these 10, +, 20 happens to be your tokens omitting the whitespaces.

Example

```
const tokenizer = new Tokenizer('10 + 20 * 30');
// `getNextToken` method gets us individual tokens
let token = tokenizer.getNextToken();
while (token) {
   // Print each token
   console.log(token);
   token = tokenizer.getNextToken();
}

/*
{ type: 'operand', token: 10 }
{ type: 'operator', token: '+' }
{ type: 'operand', token: 20 }
{ type: 'operator', token: '*' }
{ type: 'operand', token: 30 }
*/
```

Implementation

Let's first construct our `Tokenizer` class which we will use to process tokens.

The `constructor` of the class also accepts an `input` string.

We have an `index` property to track our input string position while tokenizing.

Along with that, we have an

```
class Tokenizer {
   // These are some predefined tokens (math expressions) which
we'll be encountering in the `input` string
   operators = ['+', '-', '/', '*', '(', ')', '^'];

   constructor(input) {
       // The input string which we need to tokenize
       this.input = input;
       // The index variable will keep track of where we're
while tokenizing `input` string
       this.index = 0;
   }
}
```

Now, let's add a new method `hasMoreTokens` to determine whether the entire `input` string has been processed or tokenized.

```
class Tokenizer {
   // These are some predefined tokens (math expressions) which
we'll be encountering in the `input` string
   operators = ['+', '-', '/', '*', '(', ')', '^'];

   constructor(input) {
       this.input = input;
       this.index = 0;
   }

   // To determine if the entire input string is processed (or)
tokenized or if we still have tokens to process
   hasMoreTokens() {
```

```
        return this.index < this.input.length;
    }
}
```

Now all we have are two core logical methods to implement - `matchToken` and `getNextToken`.

`getNextToken` method is used to retrieve the next token from the input string.

`matchToken` method will be used to test against predefined operator tokens, whitespaces, numbers, and finally process and get us a token.

Along with that let's look at the full implementation as well :

```
class Tokenizer {
    operators = ['+', '-', '/', '*', '(', ')', '^'];

    constructor(input) {
        this.input = input;
        this.index = 0;
    }

    hasMoreTokens() {
        return this.index < this.input.length;
    }

    // Used to test the curr char in our `input` string with the
`charToBeMatched`
    // If there's a successful match, it returns the match and
moves the index position, so that next set of tokens can be
```

```
processed
   matchToken() {
       // Skip whitespaces
       while (this.hasMoreTokens() &&
this.input.charAt(this.index) === ' ') {
           this.index++;
       }

       // Match pre-defined math expression tokens
       if (this.hasMoreTokens() &&
this.operators.includes(this.input.charAt(this.index))) {
           const token = this.input.charAt(this.index);
           this.index += 1;
           return { type: 'operator', token };
       }

       // Match numbers from 0-9
       // Using `buffer` so that we can collect all the sequence
of a number (basically grouping the individual chars of number)
       let buffer = '';
       while (this.hasMoreTokens() &&
this.input.charAt(this.index) >= '0' &&
this.input.charAt(this.index) <= '9') {
           buffer += this.input.charAt(this.index);
           this.index++;
       }

       // If the content in the buffer, matches as a valid
number
       if (buffer.length > 0 && !isNaN(Number(buffer))) {
           const token = Number(buffer);
           return { type: 'operand', token };
       }

       return null;
   }
```

```
    // Used to retrieve the next token from the `input` string
    getNextToken() {
        // If no tokens left to process, we just skip it
        if (!this.hasMoreTokens()) {
            return null;
        }

        const tokenValue = this.matchToken();
        return tokenValue;
    }
}
```

Test Cases

```
const printAllTokens = (tokenizer) => {
    let token = tokenizer.getNextToken();
    while (token) {
        console.log(token);
        token = tokenizer.getNextToken();
    }
}

printAllTokens(new Tokenizer('10 + 20'));
/*
{ type: 'operand', token: 10 }
{ type: 'operator', token: '+' }
{ type: 'operand', token: 20 }
*/

printAllTokens(new Tokenizer('10 + 20 * 30 - 40'));
/*
{ type: 'operand', token: 10 }
{ type: 'operator', token: '+' }
```

```
{ type: 'operand', token: 20 }
{ type: 'operator', token: '*' }
{ type: 'operand', token: 30 }
{ type: 'operator', token: '-' }
{ type: 'operand', token: 40 }
*/

printAllTokens(new Tokenizer('100 + 20 * (301 - 40)'));
/*
{ type: 'operand', token: 10 }
{ type: 'operator', token: '+' }
{ type: 'operand', token: 20 }
{ type: 'operator', token: '*' }
{ type: 'operator', token: '(' }
{ type: 'operand', token: 30 }
{ type: 'operator', token: '-' }
{ type: 'operand', token: 40 }
{ type: 'operator', token: ')' }
*/

printAllTokens(new Tokenizer('      10 ^   20 * 30-40   '));
/*
{ type: 'operand', token: 10 }
{ type: 'operator', token: '^' }
{ type: 'operand', token: 20 }
{ type: 'operator', token: '*' }
{ type: 'operand', token: 30 }
{ type: 'operator', token: '-' }
{ type: 'operand', token: 40 }
*/
```

Implement Custom setTimeout

Problem Statement

Implement a custom function `mySetTimeout()` and `myClearTimeout()` which are polyfills for the in-built `setTimeout` and `clearTimeout` functions.

And you should not use the built-in function directly for the problem, instead write your own version.

The `setTimeout` function is utilized to introduce a delay to execute a function call after a specified amount of time has passed.

The `setTimeout` function returns a distinct identifier that can be used to pass as input to the `clearTimeout` function, where the `clearTimeout` function stops the execution of that function call which was scheduled to be called after the specified delay.

At last we can pass any number of parameters to our `setInterval` function which needs to be passed to the callback function.

```
// Syntax

// Schedule a timer for the function `fn` to be executed after
```

```
the `delay` (delay in milliseconds)
const id = setTimeout(fn, delay, param1, param2, ..., paramN);

// clearTimeout(id) accepts the unique identifier `id` of the
scheduled timer which needs to be cleared before executing
clearTimeout(id);
```

Example

```
const startTime = Date.now();

const id = mySetTimeout(() => console.log(`Timer executed after
${Date.now() - startTime} ms`), 4000)
// Timer executed after 4001 ms

myClearTimeout(id);
// Note : The time `4001 ms` is not equal to the delay=4000 we
passed. This is because by design, setTimeout only guarantees to
execute "after" specified delay, but doesn't guarantee that it
will execute exactly at delay=4000. So don't expect timer to be
executed exactly at the specified `delay`
```

Implementation

Let’s breakdown on how we can implement the solution:

➔ We would need to maintain a map `timerMap`, where we can store the unique identifiers `id` for the timers scheduled. Later we can use this to check if the timer was cleared by `clearTimeout`.

➔The `mySetTimeout` implements the timer logic to delay the execution of the callback function call and returns the unique identifier `id`.
➔The `clearTimeout` is passed `id` of the registered timer, where we just delete the `id` from the `timerMap`.

Note: `requestIdleCallback` is a method in JavaScript that allows to schedule a function to be executed during the browser's idle periods. The browser's idle periods occur when the browser is not busy performing other tasks, such as handling user input or rendering a web page. During these idle periods, the browser can execute any functions that have been scheduled using `requestIdleCallback`

Below is our full implementation of `setTimeout` in code:

```
function createMySetTimeout() {
   // this  unique `timerID` helps to identify the
registered/scheduled timer
   let timerID = 0;
   // `timerMap` which maintains all the unique identifiers `id`
for the timer scheduled/registered
   const timerMap = {};

   function mySetTimeout(callback, delay, ...args) {
       // helper func to register/schedule a timer
       const id = scheduleTimer();

       const startTime = Date.now();
       function check() {
           // If the scheduled/registered callback is deleted,
```

```
just skip away
            if (!timerMap[id]) return;

            if (Date.now() - startTime >= delay) {
                // If the `delay` threshold is crossed, means we
can now execute the callback
                callback(...args);
            }
            else {
                // Since `delay` threshold is not crossed yet, we
wait until next idle period
                requestIdleCallback(() => check());
            }
        }

        // We know the fact that native `setTimeout` funcs are
asynchronous in nature
        // Also the fact that, the callback won't be executed
unless the Javascript call stack is free/idle
        // So we use, `requestIdleCallback` which makes sure that
your callback will be called when there is some idle time
        requestIdleCallback(() => check());

        // returns the unique identifier `id` for the registered
timer
        return id;
    }

    function myClearTimeout(id) {
        // If a timer is registered/scheduled for the `id` -
delete it
        if (timerMap[id]) delete timerMap[id];
    }

    function scheduleTimer() {
        // create unique id
```

```
        const id = ++timerID;
        // register the callback the unique `id` in our
`timerMap`
        timerMap[id] = true;

        return id;
    }

    return { mySetTimeout, myClearTimeout }
}
```

Test Cases

```
const { mySetTimeout, myClearTimeout } = createMySetTimeout();

const print = () => console.log(`Timer executed after
${Date.now() - startTime} ms`);

const startTime = Date.now();
const id1 = mySetTimeout(print, 4000)
// Timer executed after 4001 ms

const id2 = mySetTimeout(print, 1000)
// Timer executed after 1001 ms

const id3 = mySetTimeout(print, 1000)
// Timer executed after 1008 ms

// Case : Clear timers before execution
const id4 = mySetTimeout(print, 3000)
const id5 = mySetTimeout(print, 1000)
const id6 = mySetTimeout(print, 2000)

// We use `mySetTimeout` to call `myClearTimeout` just before
```

```
the scheduled timers `id4, id5, id6` are executed
mySetTimeout(() => myClearTimeout(id4), 2750)
mySetTimeout(() => myClearTimeout(id6), 1900)
// Timer executed after 1001 ms
// Note : id4, id6 timers are cleared, so only id5 timer is
executed
```

Implement Custom setInterval

Problem Statement

Implement a custom function `mySetInterval()` and `myClearInterval()` which are polyfills for the in-built `setInterval` and `clearInterval` functions.

And you should not use the built-in function directly for the problem, instead write your own version.

The `setInterval` function is utilized to execute a function call at specified intervals of delay repeatedly.

The `setInterval` function returns a distinct identifier that can be used to pass as input to the `clearInterval` function, where the `clearInterval` function stops the execution of that function call which was scheduled to be called after the specified intervals of delay repeatedly.

At last we can pass any number of parameters to our `setInterval` function which needs to be passed to the callback function.

```
// Syntax

// Schedule an interval for the function `fn` to be executed
repeatedly for every `delay` amount of intervals (delay in
```

```
milliseconds)
const id = setInterval(fn, delay, param1, param2, ..., paramN);

// `clearInterval(id)` accepts the unique identifier `id` of the
scheduled interval which needs to be cleared before executing
clearInterval(id);
```

Example

```
const startTime = Date.now();
const id = mySetInterval(() => console.log(`Interval executed at
${Date.now() - startTime} ms`), 4000)
// Interval executed at 4003 ms
// Interval executed at 8013 ms
// Interval executed at 12015 ms
// Interval executed at 16017 ms

// Note : The times `4003 ms, 8013 ms, 12015 ms, 16017 ms` is
not equal to the delay=4000 times the interval executed passed
as input. This is because by design, setInterval only guarantees
to execute "after" every specified delay, but doesn't guarantee
that it will execute exactly at delay=4000. So don't expect
interval to be executed exactly at the specified `delay` of
intervals

setTimeout(() => myClearInterval(id), 19000)
// We clear the interval after 19000 ms as a result, the
callback function passed doesn't execute anymore. Also if you
notice that our interval executes only after the minimum delay
time passed as input.
```

Implementation

Let’s breakdown on how we can implement the solution:

- ➔The implementation for our custom `setInterval` becomes comparatively easy once we have understood the implementation of our custom `setTimeout` as in our previous question.
- ➔The main idea is to execute the same `callback` function repeatedly at specified intervals of delay.
- ➔Here we can take help of our `setTimeout` function to introduce the delay before the callback execution.

Below is our full implementation of `setInterval` in code:

```
function createMySetInterval() {
   // this unique `intervalID` helps to identify the
registered/scheduled interval
   let intervalID = 0;
   // `intervalMap` which maintains all the unique identifiers
`id` for the interval scheduled/registered
   const intervalMap = {};

   function mySetInterval(callback, delay, ...args) {
       // register the unique identifier `id` in our
`intervalMap`
       const id = ++intervalID;

       function scheduleInterval() {
           // assign the identifier returned by `setTimeout` in
our intervalMap so that we can use it to `clearTimeout` when the
`clearInterval` will be called
           intervalMap[id] = setTimeout(() => {
               // Call the callback func passing the args
```

```
                callback(...args);

                // Schedule the next execution only if our
identifier is not cleared out from our `intervalMap`
                if (intervalMap[id]) scheduleInterval();
            }, delay)
        }

        // trigger the interval initially
        scheduleInterval();

        // returns the unique identifier `id` for the registered
interval
        return id;
    }

    function myClearInterval(id) {
        // If an interval is registered/scheduled for the
identifier `id` - delete it, and also clear the respective
timeout as well
        if (intervalMap[id]) {
            clearTimeout(intervalMap[id]);
            delete intervalMap[id];
        }
    }

    return { mySetInterval, myClearInterval }
}
```

Test Cases

```
const { mySetInterval, myClearInterval } =
createMySetInterval();
```

```
const print = () => console.log(`Interval executed at
${Date.now() - startTime} ms`);

const startTime = Date.now();

const id1 = mySetInterval(print, 1000);
// Interval executed at 1002 ms
// Interval executed at 2005 ms
// Interval executed at 3007 ms
// Interval executed at 4011 ms
// Interval executed at 5017 ms
// Interval executed at 6019 ms
// ... (this goes on)

const id2 = mySetInterval(print, 2000);
// Interval executed at 2003 ms
// Interval executed at 4010 ms
// Interval executed at 6011 ms

setTimeout(() => myClearInterval(id2), 7000);

const id3 = mySetInterval(print, 4000);
// Interval executed at 4003 ms
// Interval executed at 8013 ms
// Interval executed at 12015 ms
// Interval executed at 16017 ms

setTimeout(() => myClearInterval(id3), 19000);
```

Implement Custom clearAllTimers

Problem Statement

Implement a function `clearAllTimers()` which clears out all of the timeouts scheduled by `setTimeout`

This is useful as we need not maintain unique identifiers `id` to clear each timer individually. With this utility we can clear all of the timers in one shot.

Example

```
setTimeout(callback, 750);
setTimeout(callback, 1000);
setTimeout(callback, 1250);
setTimeout(callback, 1500);

// This method is highly useful since we need not remember all
the timers, and can clear all at once
clearAllTimers();
// Output: Nothing prints out since all the timers are cleared
```

Implementation

Let’s get started with implementation details here:

➔We create a IIFE function so that we can initialize and create wrappers for `setTimeout`, `clearTimeout`, `clearAllTimers`.
➔Maintain a `timerMap` to track each of those unique identifiers `id` scheduled by `setTimeout`. This is important to track so that we can clear all the timers using these unique `ids` of the timers scheduled.
➔In our wrapper logic of `setTimeout` we add the `id` to our `timerMap` and delete it when `clearTimeout` is called.
➔Now, that we are tracking all the ids, the logic to clear all the timers is pretty straightforward. We just need to iterate over all the `ids` in our `timerMap` and clear them individually.

Below is the full implementation in code:

```
// Create an IIFE which sets the wrapper for the setTimeout,
clearTimeout, which will help us in implementing the logic for
our clearAllTimers
(function createSetTimeout() {
   // We need to keep track of all the timers, so that we can
clear all the timers when required to
   const timerMap = {};

   // Store the native methods
   const nativeSetTimeout = window.setTimeout;
   const nativeClearTimeout = window.clearTimeout;

   // Create a wrapper around `setTimeout`
   window.setTimeout = (callback, delay, ...args) => {
```

```
        const id = nativeSetTimeout(callback, delay, ...args);
        // add the new uniquer timer identifier `id` to the
`timerMap` to track it
        timerMap[id] = true;
    }

    // Create a wrapper around `clearTimeout`
    window.clearTimeout = (id) => {
        nativeClearTimeout(id);
        // When `clearTimeout` is called we also need to clear
the timer from our tracker `timerMap` as well
        delete timerMap[id];
    }

    // Our original solution to the problem of clearing all
timers
    window.clearAllTimers = () => {
        // Just iterate over all the timers and clear them
one-by-one
        for (let id in timerMap) {
            clearTimeout(id);
        }
    }
})()
```

Test Cases

```
const startTime = Date.now();
const print = () => console.log(`Timer executed after
${Date.now() - startTime} ms`);

setTimeout(print, 750)
setTimeout(print, 1000)
setTimeout(print, 1250)
```

```
setTimeout(print, 1500)

// This method is highly useful since we need not remember all
the timers, and can clear all at once
clearAllTimers();
// Output: Nothing prints out since all the timers are cleared

// ----------------------------------------

setTimeout(print, 750)
setTimeout(print, 1000)
setTimeout(print, 1250)
setTimeout(print, 1500)

// Since we're clearing all the timers after 1200 ms, the timers
scheduled before 1200ms will be executed
setTimeout(() => clearAllTimers(), 1200);
// Output:
// Timer executed after 753 ms
// Timer executed after 1001 ms
```

Implement Custom Event Emitter

Problem Statement

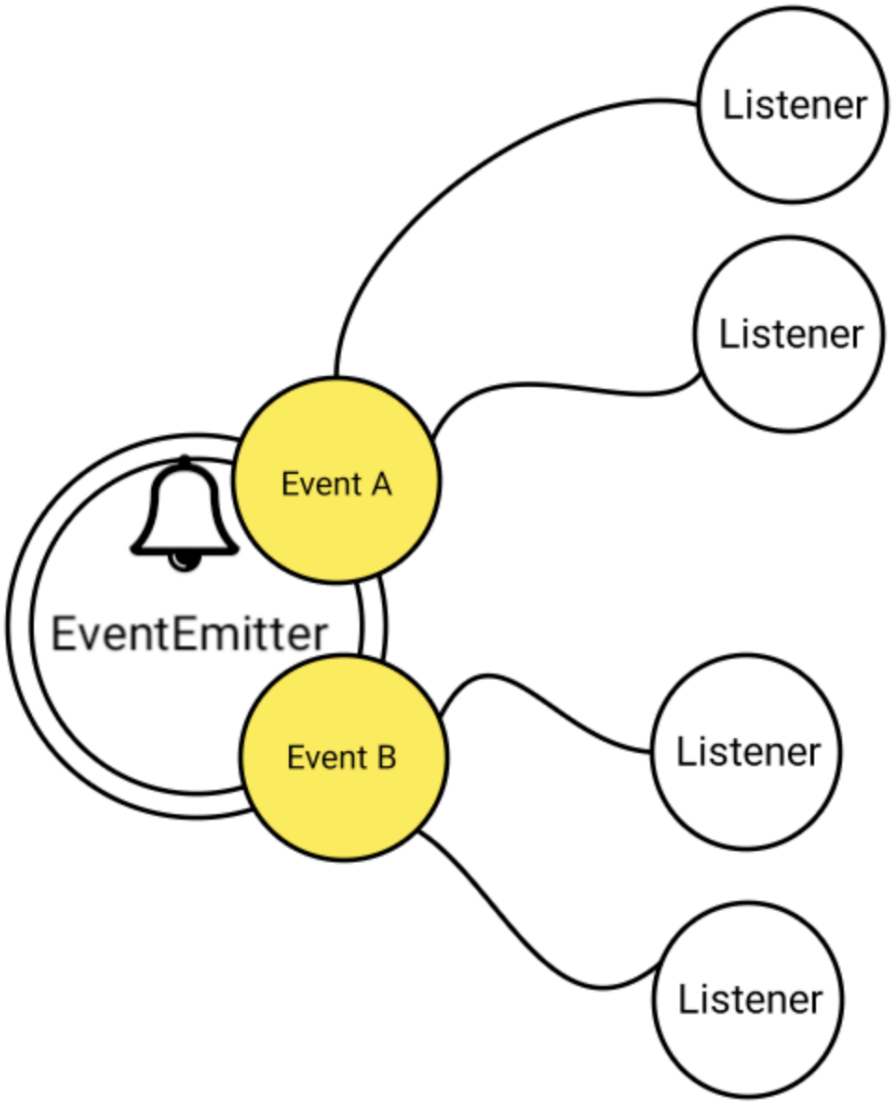
Implement a custom `Event Emitter` which allows us to create, listen to events and also allows you to manage those events.

In simple words, as the name says `Event Emitter` is something which triggers an event and anyone who is listening can execute their work when only that specific type of event occurs or triggers.

The idea of `Event Emitter` is majorly similar to the Pub-Sub Pattern (Publisher-Subscriber) or the Observer pattern at high level.

```
// Syntax

// Creates an `event` of specific type and attach an callback
which listens for the event to be triggered
emitter.addEventListener(event, callback);

// Triggering/emitting the specific event, which will execute
all the added callback listeners attached
emitter.emitEvent(event, arg1, arg2, ..., argN);

// Removes that `callback` which is listening for that `event`
emitter.removeEventListener(event, callback);
```

➔addEventListener - is used to add a callback function that’s going to be executed when the event is triggered
➔emitEvent - is used to trigger the event
➔removeEventListener - removes the callback which is listening for that event



Example

```
// Create an emitter instance
const emitter = new EventEmitter();

// Callback func which will listen & execute when the event is
triggered
const greet = (message) => console.log(`Greet: ${message}`);

// Create an event `hello` and attach a listener to it
emitter.addEventListener('hello', greet);

// Trigger the event, which will also execute all the listeners
emitter.emitEvent('hello', 'Hello, World!');
// Greet: Hello, World!
```

Implementation

We start by creating a Javascript class named `EventEmitter`. `eventsMap` is our events registry which stores the eventNames as the keys and an array as a value which contains all the listener callbacks.

```
class EventEmitter {
   constructor() {
       // Define an `eventsMap` object which maps <eventName,
[callbacks]> eventNames to an array of callbacks which will be
executed when the event emits/triggers
       this.eventsMap = {};
   }
}
```

We define a method `addEventListener` in the class, which will register an event. If the event is new, we first register it in our `eventsMap` with an empty array and then add the listener callback to the array. Else we directly push the listener to the array.

```
class EventEmitter {
   // ...

   // adds a callback listener for the `eventName`
   addEventListener(eventName, callback) {
       // If the `eventName` is new, which was never set
earlier, we create an array to store the listener callbacks for
the eventName
       if (!this.eventsMap[eventName]) {
           this.eventsMap[eventName] = [];
       }

       // Add the callback in the array to the specified
`eventName`
       this.eventsMap[eventName].push(callback);
   }
}
```

We define a method `removeEventListener` in the class, which removes only the callback passed as input from the list of listener callbacks for that event.

```
class EventEmitter {
   // ...

   // removes the same callback listener for the `eventName`
```

```
    removeEventListener(eventName, callback) {
        // Get all the callbacks for the specified `eventName`
        const allCallbacks = this.eventsMap[eventName] || [];

        // Loop through all the callbacks, and remove the input
callback by using the `filter` method
        this.eventsMap[eventName] = allCallbacks.filter(cb =>
callback !== cb);
    }
}
```

Finally, we just need to implement the `emit` method which will emit/trigger the events and then execute all the listener callbacks in an async fashion.

```
class EventEmitter {
    // ...

    // emits event for the specified `eventName` and executes all
the listener callbacks asynchronously
    emitEvent(eventName, ...args) {
        // Get all the callbacks for the specified `eventName`
        const allCallbacks = this.eventsMap[eventName] || [];

        // Loop through all the callbacks and execute them one by
one in an async fashion
        allCallbacks.forEach(callback => {
            // We make use of `requestIdleCallback` so that the
callback will be called when the main thread is idle/free
            requestIdleCallback(() => callback(...args));

            // Alternatively: we can also use the `setTimeout` as
a simple hack to execute asynchronously. Also useful for Node.js
environment compatibility
```

```
            // setTimeout(() => callback(...args), 0);
        });
    }
}
```

Note: `requestIdleCallback` queues up the callback following the first-in first-out pattern, which means the callbacks will be executed in the order they were registered (basically the order is synchronous, but the execution is not).

Now let's see the full implementation in code:

```
class EventEmitter {
    constructor() {
        this.eventsMap = {};
    }

    addEventListener(eventName, callback) {
        if (!this.eventsMap[eventName]) {
            this.eventsMap[eventName] = [];
        }

        this.eventsMap[eventName].push(callback);
    }

    removeEventListener(eventName, callback) {
        const allCallbacks = this.eventsMap[eventName] || [];
        this.eventsMap[eventName] = allCallbacks.filter(cb =>
callback !== cb);
    }

    emitEvent(eventName, ...args) {
        const allCallbacks = this.eventsMap[eventName] || [];
```

```
        // Loop through all the callbacks and execute them one by
one in an async fashion
        allCallbacks.forEach(callback => {
            // We make use of `requestIdleCallback` so that the
callback will be called when the main thread is idle/free
            requestIdleCallback(() => callback(...args));

            // Alternatively: we can also use the `setTimeout` as
a simple hack to execute asynchronously
            // setTimeout(() => callback(...args), 0);
        });
    }
}
```

Test Cases

```
const emitter = new EventEmitter();

const greet = (message) => console.log(`Greet: ${message}`);
const greetAgain = (message) => console.log(`Greet again:
${message}`);
const farewell = (message) => console.log(`Farewell:
${message}`);

emitter.addEventListener('hello', greet);
emitter.addEventListener('hello', greet);
emitter.addEventListener('hello', greetAgain);
emitter.addEventListener('goodbye', farewell);

emitter.emitEvent('hello', 'Hello, World!');
// Greet: Hello, World!
// Greet: Hello, World!
// Greet again: Hello, World!
```

```
emitter.emitEvent('goodbye', 'Goodbye, World!');
// Farewell: Goodbye, World!

emitter.removeEventListener('hello', greet);
emitter.emitEvent('hello', 'Hello, World!');
// Greet again: Hello, World!
```

Implement Custom Browser History

Problem Statement

Implement a custom `Browser History` tracker that will keep track of the order of visited URLs and will implement a simplified version for visiting the previous, current and forward URLs.

```
// Let's understand the working a bit visually. Say we start a
new tab, this is the empty history.

// [ ]

// Then we visit url A, B, C one after the other.

// [ A, B, C ]
//         ↑

// We are currently at C, we could now go back to B, then to A

// [ A, B, C ]
//   ↑

// We are currently at A, now we can move forward, and go to B

// [ A, B, C ]
//      ↑

// Now if we visit a new url D, since we are currently at B, C
is cleared (basically all the forward URLs will be cleared)
```

```
// [ A, B, D ]
//         ↑
```

Implementation

We start by creating a Javascript class `BrowserHistory` which will track all the visited URLs.

➔history - To keep track of all the visited URLs
➔current - Number which points the current URL in the history array (in the form of index)

```
class BrowserHistory {
   constructor(url) {
       // Define an array `history` to keep track of all the
visited URLs
       this.history = [];

       // `current` indicates the current URL we're viewing
       this.current = -1;

       if (url) {
           // If the url is provided while calling the
constructor, we need to visit this url by adding it to the
history and incrementing `current`
           this.history.push(url);
           this.current++;
       }
   }
}
```

Now, we got 3 important methods,

1. visit() - when a new URL has to be visited, we add the new URL to the browser history and also clear all the forward URLs using the `slice()`

```
class BrowserHistory {
   // ...

   // when a new url is visited
   visit(url) {
       // we slice all the urls up until current, this is
because when a new URL is visited, all the other `forward` urls
should be cleared
       this.history = this.history.slice(0, this.current+1);

       // Now just add the new url to history and increment
current
       this.history.push(url);
       this.current++;
   }
}
```

2. back() - visit the previous URL from the `history` by decrementing `current`
3. forward() - visit the next URL from the `history` by incrementing `current`

```
class BrowserHistory {
   // ...

   // When previous url needs to be visited
   back() {
       // We just decrement the current, but also ensure that we
```

```
don't go below 0 using Math.max()
        this.current = Math.max(0, this.current-1);
        return this.history[this.current];
    }

    // When next url needs to be visited
    forward() {
        // We just increment the current, but also ensure that we
don't go beyond the length of history using Math.min()
        this.current = Math.min(this.history.length - 1,
this.current+1);
        return this.history[this.current];
    }
}
```

Now let's move on to the full implementation in code:

```
class BrowserHistory {
    constructor(url) {
        this.history = [];
        this.current = -1;

        if (url) {
            this.history.push(url);
            this.current++;
        }
    }

    visit(url) {
        this.history = this.history.slice(0, this.current+1);
        this.history.push(url);
        this.current++;
    }
```

```
    back() {
        this.current = Math.max(0, this.current-1);
        return this.history[this.current];
    }

    forward() {
        this.current = Math.min(this.history.length - 1,
this.current+1);
        return this.history[this.current];
    }
}
```

Test Cases

```
const browserHistory = new BrowserHistory("https://google.com");

// Visiting new URLs
browserHistory.visit("https://youtube.com");
browserHistory.visit("https://facebook.com");

// Navigating back and forth
console.log(browserHistory.back());
// https://youtube.com

console.log(browserHistory.back());
// https://google.com

console.log(browserHistory.forward());
// https://youtube.com

// Visiting a new URL from a past page
browserHistory.visit("https://twitter.com");

console.log(browserHistory.back());
```

```
// https://youtube.com

console.log(browserHistory.forward());
// https://twitter.com
```

Implement Custom lodash _.chunk()

Problem Statement

Implement a function `chunk()` which is used to break the `array` which is passed as input into small chunks. The size of the chunk is also passed as input (the default size being 1, if no size is passed in input)

```
// Syntax
chunk(arr, chunkSize);
```

➔arr - The original array which needs to be chunked
➔chunkSize - describes of what size the arr needs to be chunked

Example

```
chunk([1,2,3,4,5], 1)
// [[1], [2], [3], [4], [5]]

chunk([1,2,3,4,5], 2)
// [[1, 2], [3, 4], [5]]
```

Implementation

Now, let's start with the implementation details.

Approach 1:

The idea is simple, we loop through all the elements of the array. We also maintain a `chunkedArr` and keep pushing items to the `chunkedArr`.

Once the `chunkedArr` reaches the `chunkSize` we push it to the `result`. Then, we start with the new `chunkedArr` and repeat the same process.

```
function chunk(arr, chunkSize = 1) {
   const result = [];
   // If chunkSize is 0 or less, it means no items of arr needs
to be chunked
   if (chunkSize <= 0 || arr.length === 0) {
       return result;
   }

   // `chunkedArr` stores the chunked items of array of
`chunkSize`
   let chunkedArr = [];
   for (let i = 0; i < arr.length; i++) {
       chunkedArr.push(arr[i]);

       // When chunkSize is reached, push the `chunkedArr` to
the `result`
       if (chunkedArr.length === chunkSize) {
           result.push(chunkedArr);
           chunkedArr = [];
       }
   }

   // Some items might be remained in `chunkedArr` as the
`chunkSize` might not have been reached, so we push `chunkedArr`
```

```
of remaining items to `result` as it is
    if (chunkedArr.length > 0) {
        result.push(chunkedArr);
    }

    return result;
}
```

Approach 2: (Using Reduce Method)

The main idea to chunk the arr is the same, just that we always maintain the `chunkedArr` within the result, then access and add items to it. If `chunkSize` is reached we add in a new `chunkedArr`.

Note: The initialValue `[ [ ] ]` is a 2D which will accumulate the chunkedArrs and then finally form the result. At 0th index is an empty arr which is the first chunkedArr. Likewise, we keep adding those chunks as we keep processing.

```
function chunk(arr, chunkSize = 1) {
    // If chunkSize is 0 or less, it means no items of arr needs
to be chunked
    if (chunkSize <= 0 || arr.length === 0) {
        return [];
    }

    const initialValue = [[]];
    return arr.reduce((result, elem) => {
        // Get the last arr which will be our `chunkedArr`
        let chunkedArr = result[result.length-1];
```

```
        // If chunkedArr has reached chunkSize, we push a new
empty array which will be the new `chunkedArr`
        if (chunkedArr.length === chunkSize) {
            result.push([]);
        }

        // So, now the `chunkedArr` will be the latest empty arr
we just pushed to the `result`
        chunkedArr = result[result.length-1];

        chunkedArr.push(elem);

        return result;
    }, initialValue)
}
```

Test Cases

```
chunk([1,2,3,4,5], 1);
// [[1], [2], [3], [4], [5]]

chunk([1,2,3,4,5], 2);
// [[1, 2], [3, 4], [5]]

chunk([1,2,3,4,5], 0);
// []

chunk([1,2,3,4,5], 3);
// [[1, 2, 3], [4, 5]]

chunk([1,2,3,4,5], 4);
// [[1, 2, 3, 4], [5]]

chunk([1,2,3,4,5], 5);
```

```
// [[1, 2, 3, 4, 5]]

chunk([1,2,3,4,5], 6);
// [[1, 2, 3, 4, 5]]

chunk([1,2,3,4,5]);
// [[1], [2], [3], [4], [5]]
```

Implement Custom Deep Clone

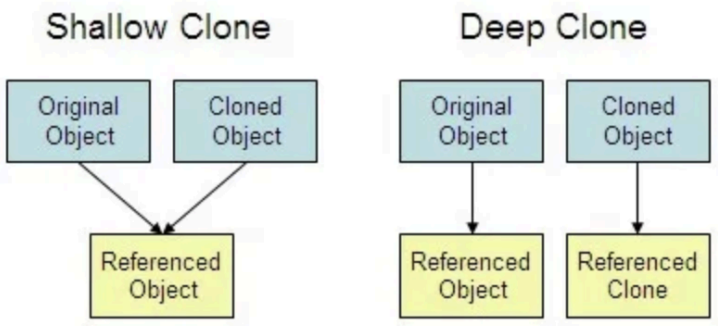
Problem Statement

Implement a function `deepClone()` which is used to create a deep copy of the value and it recursively clones the value. The newly created object has the same value as the original value but they are not the same object in the memory.

The method `deepClone()` will be passed an input `value` which needs to be cloned recursively.

```
// Syntax
deepClone(value);
```

I recommend solving this problem of Implementing Custom Object.assign() which implements shallow copy, and there I have explained all the technical concepts which will be easy for you to relate while implementing this problem.



Example

```
const obj = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
   address: {
       city: 'New york',
       state: 'NY',
       temp: ['California', 'Alabama', 'Florida']
   }
}

const clonedObj = deepClone(obj);

console.log(clonedObj);
/*
{
   name: 'Peter',
   age: 29,
   spiderman: true,
   movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
   address: {
       city: 'New york',
       state: 'NY',
       temp: ['California', 'Alabama', 'Florida']
   }
}
*/

console.log(clonedObj === obj);
// false
```

```
console.log(clonedObj.movies === obj.movies);
// false

console.log(clonedObj.address === obj.address);
// false

console.log(clonedObj.address.temp === obj.address.temp);
// false
```

Implementation

Let's understand the implementation details we need to apply:

- ➔ Create a function `deepClone()` which takes an input `value` which we need to clone.
- ➔ For primitives like number, string, booleans, null, undefined we return the value as it is.
- ➔ For reference data types like Arrays/Objects, we need to traverse them recursively each element or property of the Array/Object and call `deepClone()` on them. This process shall go on and return us the cloned value.
- ➔ All we need to do is just add the value to the result Array/Object and finally return it.

```
function deepClone(value) {
   if (value === undefined || value === null) {
       return value;
   }

   // Handling for primitive data types like number, boolean,
string
```

```
    if (typeof value !== 'object') {
        // For primitive values we return `value` since those are
referenced by values and not by memory reference
        return value;
    }

    // Based on the type of value (Arrays or Objects), we use the
required datatype
    const result = Array.isArray(value) ? [] : {};

    // Note: for...in loop helps us iterate on all the enumerable
properties of both Arrays/Objects
    // For Arrays, `key` will be the index values - 0, 1, 2, 3...
    // For Objects, `key` will be the individual enumerable keys
defined on the object
    for (let key in value) {
        // This works for both Arrays/Objects since we're aware
of the fact that Arrays are also objects
        result[key] = deepClone(value[key]);
    }

    // Note: The above for...in loop retrieves only the
enumerable properties
    // But `Symbol()` datatypes are non-enumerable properties of
the object, and we need to clone Symbol based properties as well
    // So we make use of `getOwnPropertySymbols` which gets us an
array of `Symbol` properties on the array

    const symbolProps = Object.getOwnPropertySymbols(value);
    for (let key of symbolProps) {
        result[key] = deepClone(value[key]);
    }

    return result;
}
```

One more major case we are missing out is the circular references. Refer this example to understand the circular references,

```
const a = [1, '4'];
a.push(a);
// That is, const a = [ 1, '4', [Circular *a]

const b = [1, '4'];
b.push(b);
// That is, const b = [ 1, '4', [Circular *b]
```

Basically when we clone an Object/Array which has any circular references we need to handle it in order to avoid Call Stack Limit Exceeded error (StackOverflow).

The idea is simple to use a Javascript `Map` named as `visited` which will store the <value, clonedValue>.

In our recursion logic, if a value is already visited, just return the mapped clonedValue (as we have already processed it).

Else add a new entry for the value and clonedValue (Object/Array).

Let’s checkout the full implementation of this below:

```
// We take `visited` map as an optional parameter which if not
passed while calling `deepClone()` we ourself assign a `new
Map()` object to `visited`
```

```
function deepClone(value, visited = new Map()) {
    if (value === undefined || value === null) {
        return value;
    }

    if (typeof value !== 'object') {
        return value;
    }

    const result = Array.isArray(value) ? [] : {};

    // Check if the `value` is already visited
    if (visited.has(value)) {
        // If yes then just return the value (to avoid circular
reference)
        return visited.get(value);
    }
    else {
        // Else the set the `result` as the value, since this
result will have all the deepCloned values once the below lines
of code is processed
        visited.set(value, result);
    }

    for (let key in value) {
        result[key] = deepClone(value[key], visited);
    }

    const symbolProps = Object.getOwnPropertySymbols(value);
    for (let key of symbolProps) {
        result[key] = deepClone(value[key], visited);
    }

    return result;
}
```

Test Cases

```
deepClone(arr1);
/*
[
 { name: 'Spiderman', [Symbol(greet)]: 'Hello World!!' },
 {
   a: null,
   b: undefined,
   c: { [Symbol(farewell)]: 'Goodbye world!!' }
 }
]
*/

const obj1 = {
   a: 1,
   b: [1, [2, [3, { c: 4 }]]],
   c: 4
};

deepClone(obj1);
// { a: 1, b: [ 1, [ 2, [ 3, { c: 4 } ] ] ], c: 4 }

const obj2 = {
   name: 'Peter',
   age: 29,
   spiderman: true,
   movies: ['Spiderman', 'Amazing Spiderman', 'Far From Home'],
   address: {
       city: 'New york',
       state: 'NY',
       temp: ['California', 'Alabama', 'Florida']
   }
}
```

```
deepClone(obj2);
/*
{
 name: 'Peter',
 age: 29,
 spiderman: true,
 movies: [ 'Spiderman', 'Amazing Spiderman', 'Far From Home' ],
 address: {
   city: 'New york',
   state: 'NY',
   temp: [ 'California', 'Alabama', 'Florida' ]
 }
}
*/
```

Promisify the Async callbacks

Problem Statement

Implement a function `promisify()` which converts callback based async functions to a promisified version.

Let's get some basic idea first to understand why we need callbacks and promises at all?

Callback and promise are used to handle asynchronous operations. An asynchronous operation is an operation that is not executed immediately. It is executed after some time.

For example, if you are making a request to the server to get some data, then it will take some time to get a response from the server. So, we need to wait for the response. If we are making multiple requests to the server, then we need to wait for all the responses. So, we need to handle asynchronous operations.

Note: Okay, what exactly I mean by async callbacks are the standard error first callback function. In the standard error first callback function, the first parameter is an `error`, and the second parameter will be the `result`.

```
const someAsyncFunc = (callback, data) => {
   // ...
}

const input = 10;
someAsyncFunc((error, result) => {}, input);
```

Now, let's understand the reason why we would need to **convert callback based async functions to promisified** version.

We need to convert the callback to a promisified version because the promise is more readable and easy to handle. We can handle multiple asynchronous operations using promise. We can also handle asynchronous operations which have a dependency on each other using promise.

Let's understand this with an example. Suppose we have a simple `multiply` function that will take two numbers and return the product of those two numbers in a callback function.

We have also added the `setTimeout` function to simulate the asynchronous behavior of the function.

```
function multiply(a, b, callback) {
   setTimeout(() => callback(null, a * b), 1000);
}

multiply(2, 3, (error, value) => {
```

```
    if (error instanceof Error) {
        console.log(error);
    }
    else {
        console.log(value);
    }
});
// Output: 6 (Prints out after 1000ms)
```

Now, let's say we want to multiply another number to the result. We can do this by passing the result to another function.

```
multiply(1, 2, (error, firstResult) => {
    console.log(firstResult);
    // Output for firstResult : 2

    multiply(firstResult, 3, (error, secondResult) => {
        console.log(secondResult);
        // Output for secondResult : 6

        multiply(secondResult, 4, (error, finalResult) => {
            console.log(finalResult);
            // Output for finalResult : 24
        })
    })
})
```

This can get more complex if we have 'n' async tasks to execute and ultimately leading to callback hell, which is not more readable as well.

Example

```
const multiplyPromise = promisify(multiply);

multiplyPromise(1, 2)
.then(firstResult => multiplyPromise(firstResult, 3))
.then(finalResult => console.log(`Final result is
${finalResult}`))
.catch(error => console.log('Found error', error))

// Final result is 6
```

Implementation

Let's checkout the implementation details:

- ➔Define a function `promisify` which takes in any async function, and returns a new function which is a promisified version of that async function.
- ➔Basically the returned function wraps the async function in a new Promise object. Then we call the async function, in the same error-first callback style.
- ➔When the callback is called we settle the promise with either resolved/rejected based on the callback values.

```
const promisify = (asyncFunc) => {
   // We return a function which will promisify the `asyncFunc`
   return function(...args) {
       // wrap the `asyncFunc` in a promise object and return
the promise
       return new Promise((resolve, reject) => {
```

```
            const errorFirstCallback = (error, result) => {
                if (error instanceof Error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            }

            // Now call the async function and pass the
`errorFirstCallback`
            asyncFunc(errorFirstCallback, ...args);
        })
    }
}
```

Test Cases

```
const multiplyPromise = promisify(multiply);

multiplyPromise(1, 2)
.then(firstResult => multiplyPromise(firstResult, 3))
.then(secondResult => multiplyPromise(secondResult, 4))
.then(finalResult => console.log(`Final result is
${finalResult}`))
.catch(error => console.log('Found error', error))
// Final result is 24

multiplyPromise(1, 2)
.then(firstResult => multiplyPromise(firstResult, 3))
.then(finalResult => console.log(`Final result is
${finalResult}`))
.catch(error => console.log('Found error', error))
// Final result is 6
```

Implement 'N' async tasks in Series

Problem Statement

Implement a function `series()` which takes in an array of async functions as input. We need to process each async task one-by-one in series or sequence.

Initially we pass some initial input data which is required by that first async function to process on. That async function will return an output `result` after processing it. Then, we need to pass that `result` value to the second async function as the new input data to it. After processing the input, the second async function also returns an output `result` again. Now, we again need to pass this `result` value to the next async function and the process goes on… until we process all the async functions.

You can imagine this as a pipe.

```
// asyncFunc1(input) -> output1 -> asyncFunc1(output1) ->
output2 -> asyncFunc3(output2) -> output3 -> ...
```

Now, the problem is since we need to process asynchronous tasks, we cannot simply make a for loop, because we have no idea at what point of time the asyncFunc() might return us the output result.

So, we need to implement a custom logic so that we can execute the async tasks one after the another.

Note: Each async function takes in 2 parameters as input, first is the error-first callback and then the input data on which it processes.

```
const someAsyncFunc = (callback, input) => {
   // your async work goes in here

   // processing the `input` data
   // ...
   // Got the `result` as an output

   if (result instanceof Error) {
       callback(result, undefined); // (error, no data)
   }
   else {
       callback(undefined, result); // (no error, result data)
   }
}

// Note: The callback which will be passed to the
`someAsyncFunc` will be the standard "error-first" callback
type, where 1st parameter is error, and 2nd parameter being the
data (or result from that `someAsyncFunc`)

const callback = (error, result) => {
   // ...
};

// You can understand the importance of this callback as this
term `callbackMeWhenCompleted`, which indicates that this
callback will be called after the `someAsyncFunc` completes it's
```

```
work and passes either an error or the result, which indicates
the status of the `someAsyncFunc` (successful or failure)
```

Example

```
const multiplyBy2Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 2), 3000);
}

const multiplyBy3Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 3), 1000);
}

// `series` func takes in the array of async tasks, and returns
an new function `asyncSeriesExecutor`,
const asyncSeriesExecutor = series([
   multiplyBy2Async,
   multiplyBy3Async,
])

// This function `asyncSeriesExecutor` takes 2 parameters -

// 1st is the error-first callback
// 2nd is the initial `input` data on which the asyncFuncs will
process in series, and passes the `result` of the asyncFunc to
next set of asyncFuncs as next `input` data in series and the
process goes on until all the asyncFuncs are completed

asyncSeriesExecutor((error, result) => {
   if (error instanceof Error) {
       console.log(`[Caught Error] ${error}`);
   }
   else {
       console.log(`Completed async tasks with result :
```

```
${result}`);
   }
}, 1)

// Output - Completed async tasks with result : 6
```

Implementation

Approach 1: (Using recursion)

Let's go through the implementation details:

- ➔Pass the array of async functions to the `series` function which returns another function which handles the async tasks in series.
- ➔This returned function takes two parameters, a callback `(error, result) => {}` and the initial input data `initData` to start processing the first async task.
- ➔Start by executing the first async function and passing the `initData`. The async function also takes a callback `callbackMeWhenCompleted` which will be called when the async task completes.

```
function series(asyncFuncsArr) {

   // `mainCallback` is passed to know the end-result after
executing all the asyncFuncs from asyncFuncsArr
   return function(mainCallback, initData) {
       // `idx` will help us track and get asyncFuncs one-by-one
in "series"
```

```
        let idx = 0;

        const callbackMeWhenCompleted = (error, result) => {
            // ...
        }

        // Get the first async func
        const firstAsyncFunc = asyncFuncsArr[idx];

        // Trigger the first asyncFunc, and pass the
`callbackMeWhenCompleted` and `initData`
firstAsyncFunc(callbackMeWhenCompleted, initData);
    }
}
```

- ➔When the async task completes, the callback `callbackMeWhenCompleted` gets an error or the result of that async task.
- ➔If it's an error, return away by calling the `mainCallback` with the same error.
- ➔Else check if we have `nextAsyncFunc`,
  - ◆If yes then pass the `result` to that `nextAsyncFunc` as input so that it processes and this process continues until we have async funcs in the array,
  - ◆Else return away by calling the `mainCallback`by passing the `result`.

```
function series(asyncFuncsArr) {
    return function(mainCallback, initData) {
        let idx = 0;
        const callbackMeWhenCompleted = (error, result) => {
            // Now, when this func `callbackMeWhenCompleted` is
```

```
called, means the async task was completed - either with an
`error` or with `result`

            if (error instanceof Error) {
                // If the asyncFunc had an error, just return
away by calling the `mainCallback` with the same `error`
mainCallback(error, undefined);
            }
            else {
                // Increment `idx` to get the `nextAsyncFunc`
                idx++;

                // If not an error, pass current `result` to the
nextAsyncFunc
                const nextAsyncFunc = asyncFuncsArr[idx];

                // If we have `nextAyncFunc`, we continue execute
in "series"
                if (nextAsyncFunc) {
                    nextAsyncFunc(callbackMeWhenCompleted,
result);
                }
                else {
                    mainCallback(undefined, result);
                }
            }
        }

        const firstAsyncFunc = asyncFuncsArr[idx];
        firstAsyncFunc(callbackMeWhenCompleted, initData);
    }
}
```

Approach 2: (Using Promisify and Reduce)

To understand promisifying, refer to our previous problem - 35. Promisify the Async callbacks to better understand the concept of promisification.

Basic idea of promisifying is to convert error-first callback based async functions to promise based async functions.

- ➔Create an initial promise using the `initData`.
- ➔Then pass the `initialPromise` as initial value to the reduce method.
- ➔In the reduce method iterate over all the async funcs -
  - ◆Now, if the `resultPromise` resolves promisify the nextAsyncFunc and pass the resolved `result` as input to the `promisifiedasyncFunc`.
  - ◆Else if `resultPromise` is rejected, return the rejected promise
- ➔The above reduce method also forms the promise chain which leads to our `finalPromise`. Based on the `finalPromise` settled status call the `mainCallback`.

```
// Refer to our previous question to understand this logic in
our previous problem number - 35. Promisify the async callbacks
const promisify = (asyncFunc) => {
   // We return a function which will promisify the `asyncFunc`
   return function(...args) {
       // wrap the `asyncFunc` in a promise object and return
the promise
       return new Promise((resolve, reject) => {
```

```
            const errorFirstCallback = (error, result) => {
                if (error instanceof Error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            }

            // Now call the async function and pass the
`errorFirstCallback`
            asyncFunc(errorFirstCallback, ...args);
        })
    }
}

function series(asyncFuncsArr) {
    return function(mainCallback, initData) {
        // Create your `initialPromise` to be passed to the
`reduce` method as the initialValue
        const initialPromise = Promise.resolve(initData);
        // Iterate over all the `asyncFuncs` and reduces to a
`finalPromise`
        // Ultimately this forms chains all of the promises
promise, making the asyncFuncs to execute in "series"

        const finalPromise = asyncFuncsArr.reduce((resultPromise,
nextAsyncFn) => {
            return resultPromise.then(result => {
                // When resolved with result, promisify
`nextAsyncFn`
                const promisifiedAsyncFunc =
promisify(nextAsyncFn);

                // Execute `promisifiedAsyncFunc` and return it
to make it part of the promise chain
                return promisifiedAsyncFunc(result);
```

```
            })
            .catch(error => Promise.reject(error))

        }, initialPromise)


        // Finally when the `finalPromise` resolves/rejects, call
the `mainCallback` based on the settled state of the promise
        finalPromise
        .then(finalResult => mainCallback(undefined,
finalResult))
        .catch(error => mainCallback(error, undefined))
    }
}
```

Test Cases

```
const multiplyBy2Async = (callback, input) => {
    setTimeout(() => callback(undefined, input * 2), 3000);
}

const multiplyBy3Async = (callback, input) => {
    setTimeout(() => callback(undefined, input * 3), 1000);
}

const asyncError = (callback, input) => {
    setTimeout(() => callback(new Error('Testing error')), 2000);
}

const callbackMeWhenCompleted = (error, result) => {
    if (error instanceof Error) {
        console.log(`[Caught Error] ${error}`);
    }
```

```
    else {
        console.log(`Completed async tasks with result :
${result}`);
    }
}

const asyncSeriesExecutor1 = series([
    multiplyBy2Async,
    multiplyBy3Async,
    multiplyBy3Async,
    multiplyBy2Async,
])

asyncSeriesExecutor1(callbackMeWhenCompleted, 10)

// Completed async tasks with result : 360

const asyncSeriesExecutor2 = series([
    multiplyBy2Async,
    asyncError,
    multiplyBy2Async,
])
asyncSeriesExecutor2(callbackMeWhenCompleted, 1)
// [Caught Error] Error: Testing error
```

Implement 'N' async tasks in Parallel

Problem Statement

Implement a function `parallel()` which takes in an array of async functions as input. We need to process each of these async tasks in parallel.

It is different from the previous problem Implement 'N' async tasks in Series since here we need to wait until each async function to get completed so that we can then execute the next async function.

Instead, we execute all of them in parallel, and wait only until all of those async functions get completed and then finally return the collected results of those async functions together.

You can imagine this is similar to the behavior of Promise.all()

Note: Each async function takes in 2 parameters as input, first is the error-first callback and then the input data on which it processes.

```
const someAsyncFunc = (callback, input) => {
   // your async work goes in here

   // processing the `input` data
   // ...
```

```
    // Got the `result` as an output

    if (result instanceof Error) {
        callback(result, undefined); // (error, no data)
    }
    else {
        callback(undefined, result); // (no error, result data)
    }
}

// Note: The callback which will be passed to the
`someAsyncFunc` will be the standard "error-first" callback
type, where 1st parameter is error, and 2nd parameter being the
data (or result from that `someAsyncFunc`)

const callback = (error, result) => {
    // ...
};

// You can understand the importance of this callback as this
term `callbackMeWhenCompleted`, which indicates that this
callback will be called after the `someAsyncFunc` completes its
work and passes either an error or the result, which indicates
the status of the `someAsyncFunc` (successful or failure)
```

The function `parallel()` takes in an array of async functions as input and returns a new function `parallelExecutor` which takes in an error-first callback and the initial `initData`.

Example

```
const multiplyBy2Async = (callback, input) => {
    setTimeout(() => callback(undefined, input * 2), 3000);
```

```
}

const multiplyBy3Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 3), 1000);
}

const callbackMeWhenCompleted = (error, result) => {
   if (error instanceof Error) {
       console.log(`[Caught Error] ${error}`);
   }
   else {
       console.log(`Completed async tasks with result :
${result}`);
   }
}

const asyncParallelExecutor1 = parallel([
   multiplyBy2Async,
   multiplyBy3Async
])


asyncParallelExecutor1(callbackMeWhenCompleted, 10)
// Completed async tasks with result : [20,30]
```

Implementation

Let’s go through the implementation details now:

Approach 1: (Using recursion)

➔create a parallel function and pass in an array of async functions and return a new function which will be the parallelExecutor for the async functions.
➔Iterate over each async function and pass the error-first callback and initial `initData` as input parameters to them.
➔If any error is found, immediately call the mainCallback passing the error value.
➔Else keep collecting the `result` of the async functions in the `resultsArr` maintaining the index of the result as same as that `asyncFunc` index.

```
function parallel(asyncFuncsArr) {

   // return a helper function which would help us parallelize
the asyncFuncs
   return (mainCallback, initData) => {
       // tracks if any error has occurred in the asyncFuncs
       let errorFound = false;

       // tracks the count of the number of async functions that
completed with result
       let resultsCount = 0;

       // stores the result of all the asyncFuncs maintaining
the index of the results
       const resultsArr = [];

       // iterate on each async funcs and execute them in
parallel
       asyncFuncsArr.forEach((asyncFunc, idx) => {
           const callbackMeWhenCompleted = (error, result) => {
               // If an error was already found, just skip
```

```
further
                if (errorFound) {
                    return;
                }

                // If the error occurs, call the `mainCallback`
right way passing the error value
                if (error instanceof Error) {
                    errorFound = true;
                    mainCallback(error, undefined);
                }
                else {
                    // Else store the result in the `resultsArr`
                    resultsArr[idx] = result;
                    resultsCount++;
                }

                // When all of the asyncFuncs are completed, call
the `mainCallback` passing the resultsArr as well
                if (resultsCount === asyncFuncsArr.length) {
                    mainCallback(undefined, resultsArr);
                }
            }

            // Call all the asyncFunc with the same `initData`
            asyncFunc(callbackMeWhenCompleted, initData);
        })
    }
}
```

Approach 2: (Using Promisify)

To understand promisifying, refer to our previous problem - 35. Promisify the Async callbacks to better understand the concept of promisification.

Basic idea of promisifying is to convert error-first callback based async functions to promise based async functions.

- ➔Create a function `parallel()` and pass in an array of async functions as input and return a new function which will be the parallelExecutor which takes in an error-first callback and initial value `initData`.
- ➔Iterate over all the async functions, promisify the async function and call the `promisifiedAsyncFunc` it by passing the `initData`
- ➔If the promise resolves with result, add the result to the `resultsArr`. Once all the async functions completes and gets resolved with results, then finally return the collected `resultsArr`.
- ➔Else if the promise is rejected with an error call the `mainCallback` right away passing the error value.

```
// Refer our previous question to understand this logic in our
previous problem number 35 - Promisify the async callbacks
const promisify = (asyncFunc) => {
   // We return a function which will promisify the `asyncFunc`
   return function(...args) {
       // wrap the `asyncFunc` in a promise object and return
the promise
       return new Promise((resolve, reject) => {
           const errorFirstCallback = (error, result) => {
```

```
                if (error instanceof Error) {
                    reject(error);
                } else {
                    resolve(result);
                }
            }

            // Now call the async function and pass the
`errorFirstCallback`
            asyncFunc(errorFirstCallback, ...args);
        })
    }
}

function parallel(asyncFuncsArr) {

    // return a helper function which would help us parallelize
the asyncFuncs
    return (mainCallback, initData) => {
        // tracks if any error has occured in the asyncFuncs
        let errorFound = false;

        // tracks the count of the number of async functions that
completed with result
        let resultsCount = 0;

        // stores the result of all the asyncFuncs maintaining
the index of the results
        const resultsArr = [];

        // iterate on each async funcs and execute them in
parallel
        asyncFuncsArr.forEach((asyncFunc, idx) => {
            // Promisify the async function
            const promisifiedAsyncFunc = promisify(asyncFunc);
```

```
            promisifiedAsyncFunc(initData)
            .then(result => {
                if (errorFound) {
                    return;
                }

                // store the result in the `resultsArr`
                resultsArr[idx] = result;
                resultsCount++;

                // When all of the asyncFuncs are completed, call
the `mainCallback` passing the resultsArr as well
                if (resultsCount === asyncFuncsArr.length) {
                    mainCallback(undefined, resultsArr);
                }
            })
            .catch(error => {
                if (errorFound) {
                    return;
                }

                // If the error occurs, call the `mainCallback`
right way passing the error value
                errorFound = true;
                mainCallback(error, undefined);
            })
        })
    }
}
```

Test Cases

```
const multiplyBy2Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 2), 3000);
```

```
}

const multiplyBy3Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 3), 1000);
}

const asyncError = (callback, input) => {
   setTimeout(() => callback(new Error('Testing error')), 2000);
}

const callbackMeWhenCompleted = (error, result) => {
   if (error instanceof Error) {
       console.log(`[Caught Error] ${error}`);
   }
   else {
       console.log(`Completed async tasks with result :
${result}`);
   }
}

const asyncParallelExecutor1 = parallel([
   multiplyBy2Async,
   multiplyBy3Async,
   multiplyBy3Async,
   multiplyBy2Async
])


asyncParallelExecutor1(callbackMeWhenCompleted, 10)
// Completed async tasks with result : [20, 30, 30, 20]

const asyncParallelExecutor2 = parallel([
   multiplyBy2Async,
   asyncError,
   multiplyBy2Async,
])
```

```
asyncParallelExecutor2(callbackMeWhenCompleted, 1)
// [Caught Error] Error: Testing error
```

Implement 'N' async tasks in Race

Problem Statement

Implement a function `race()` which takes in an array of async functions as input and returns the first result (or) first error as soon as any of the async functions get completed first.

You can imagine this is similar to the behavior of Promise.race()

Note: Each async function takes in 2 parameters as input, first is the error-first callback and then the input data on which it processes.

```
const someAsyncFunc = (callback, input) => {
   // your async work goes in here

   // processing the `input` data
   // ...
   // Got the `result` as an output

   if (result instanceof Error) {
       callback(result, undefined); // (error, no data)
   }
   else {
       callback(undefined, result); // (no error, result data)
   }
}

// Note: The callback which will be passed to the
```

```
`someAsyncFunc` will be the standard "error-first" callback
type, where 1st parameter is error, and 2nd parameter being the
data (or result from that `someAsyncFunc`)

const callback = (error, result) => {
   // ...
};

// You can understand the importance of this callback as this
term `callbackMeWhenCompleted`, which indicates that this
callback will be called after the `someAsyncFunc` completes its
work and passes either an error or the result, which indicates
the status of the `someAsyncFunc` (successful or failure)
```

The function `race()` takes in an array of async functions as input and returns a new function `raceExecutor` which takes in an error-first callback and the initial `initData`.

Example

```
const multiplyBy2Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 2), 3000);
}

const multiplyBy3Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 3), 1000);
}

const callbackMeWhenCompleted = (error, result) => {
   if (error instanceof Error) {
       console.log(`[Caught Error] ${error}`);
   }
   else {
```

```
        console.log(`Completed async tasks with result :
${result}`);
    }
}

const asyncRaceExecutor1 = race([
    multiplyBy2Async,
    multiplyBy3Async,

])

asyncRaceExecutor1(callbackMeWhenCompleted, 10)
// Completed async tasks with result : [30]
```

Implementation

Let's go through the implementation details now:

Approach 1: (Using recursion)

- ➔ create a parallel function and pass in an array of async functions and return a new function which will be the parallelExecutor for the async functions.
- ➔ Iterate over each async function and pass the error-first callback and initial `initData` as input parameters to them.
- ➔ When any of the async functions get completed first we immediately call the `mainCallback` right away passing the result / error whatever it be.

```
function race(asyncFuncsArr) {
   return function(mainCallback, initData) {
       // `completed` tracks if any of this async functions was
completed or not
       let completed = false;

       asyncFuncsArr.forEach(asyncFunc => {
           const callbackMeWhenCompleted = (error, result) => {
               // If any of the async functions is completed,
return away
               if (completed) {
                   return;
               }

               // Mark `completed` as true, which indicates the
race among the async functions is completed
               completed = true;

               if (error instanceof Error) {
                   // If error, call `mainCallback` with the
same error
                   mainCallback(error, undefined);
               }
               else {
                   // Else call `mainCallback` with the result
value
                   mainCallback(undefined, result);
               }
           }

           asyncFunc(callbackMeWhenCompleted, initData);
       })
   }
}
```

Approach 2: (Using Promisify)

To understand promisifying, refer to our previous problem - 35. Promisify the Async callbacks to better understand the concept of promisification.

Basic idea of promisifying is to convert error-first callback based async functions to promise based async functions.

- ➔Create a function `parallel()` and pass in an array of async functions as input and return a new function which will be the parallelExecutor which takes in an error-first callback and initial value `initData`.
- ➔Iterate over all the async functions, promisify the async function and call the `promisifiedAsyncFunc` it by passing the `initData`
- ➔Wait for any of the async functions to get completed first, once completed mark it as completed, then call the `mainCallback` passing either the first error / result.

```
// Refer our previous question to understand this logic in our
previous problem number 35 - Promisify the async callbacks
const promisify = (asyncFunc) => {
   // We return a function which will promisify the `asyncFunc`
   return function(...args) {
       // wrap the `asyncFunc` in a promise object and return
the promise
       return new Promise((resolve, reject) => {
           const errorFirstCallback = (error, result) => {
               if (error instanceof Error) {
                   reject(error);
               } else {
                   resolve(result);
```

```
            }
        }

        // Now call the async function and pass the
`errorFirstCallback`
        asyncFunc(errorFirstCallback, ...args);
    })
  }
}

function race(asyncFuncsArr) {
  return function(mainCallback, initData) {
    // `completed` tracks if any of this async functions was
completed or not
    let completed = false;

    asyncFuncsArr.forEach(asyncFunc => {
        // Promisify the async function
        const promisifiedAsyncFunc = promisify(asyncFunc);

        promisifiedAsyncFunc(initData)
        .then(result => {
            if (completed) {
                return;
            }

            // Mark `completed` as true, which indicates the
race among the async functions is completed
            completed = true;

            // Call `mainCallback` with the result value
            mainCallback(undefined, result);
        })
        .catch(error => {
            if (completed) {
                return;
```

```
                }

                completed = true;

                // Call `mainCallback` with the same error value
                mainCallback(error, undefined);
            })
        })
    }
}
```

Test Cases

```
const multiplyBy2Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 2), 3000);
}

const multiplyBy3Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 3), 1000);
}

const multiplyBy4Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 4), 500);
}

const multiplyBy5Async = (callback, input) => {
   setTimeout(() => callback(undefined, input * 5), 250);
}

const asyncError = (callback, input) => {
   setTimeout(() => callback(new Error('Testing error')), 2000);
}

const callbackMeWhenCompleted = (error, result) => {
```

```
    if (error instanceof Error) {
        console.log(`[Caught Error] ${error}`);
    }
    else {
        console.log(`Completed async tasks with result :
[${result}]`);
    }
}

const asyncRaceExecutor1 = race([
    multiplyBy2Async,
    multiplyBy3Async,
    multiplyBy4Async,
    multiplyBy5Async
])


asyncRaceExecutor1(callbackMeWhenCompleted, 10)
// Completed async tasks with result : [50]

const asyncRaceExecutor2 = race([
    multiplyBy2Async,
    asyncError,
    multiplyBy2Async,
])
asyncRaceExecutor2(callbackMeWhenCompleted, 1)
// [Caught Error] Error: Testing error
```

Implement Custom Object.is() method

Problem Statement

Implement a function `is()` which takes in two values as input. It returns a boolean value indicating whether the two arguments are the same or not.

```
// Syntax
is(value1, value2);
```

➔value1 - It is the first value to be compared
➔value2 - It is the second value to be compared

Example

```
console.log(is(10, 10));
// true

console.log(is('Hello', 'Hello'));
// true

console.log(is({ a: 1 }, { a: 1 }));
// false

console.log(is(null, undefined));
// false

console.log(is(null, 'Hello'));
// false
```

Now, don’t you think, triple equals does the same thing `===`

```
console.log(10 === 10);
// true

console.log('Hello' === 'Hello');
// true

console.log({ a: 1 } === { a: 1 });
// false

console.log(null === undefined);
// false

console.log(null === 'Hello');
// false
```

So, yeah the result seems the same for both. And, we have a few concepts to understand, what different do these 3 equality operators actually do.

- ➔Double equals `==`
- ➔Triple equals `===`
- ➔Object.is() method
- ➔SameValueZero (this used in most built-in operations of the language, now we will focus only on the first 3 equality operators)

Double Equals: It performs type coercions (implicit conversion of values from one data type to another) when

comparing two values, and will also handle special cases for Numbers like NaN and zero-based values like +0 and -0 as well.

```
console.log(10 == '10');
// true

console.log('1' == 1);
// true

console.log(10 == 10);
// true

console.log({ a: 1 } == { a: 1 });
// false

const obj = { b: 2 };
console.log(obj == obj);
// true

console.log(NaN == NaN);
// false

const val = NaN;
console.log(val == val);
// false

console.log(-0 == +0);
// true

console.log(-0 == 0);
// true

console.log(-0 === -0);
// true
```

Note: A `NaN` is not equal to another `NaN`, it's not because of the different references, but that's how the language defines it.

Triple Equals: This is a more strict form of value comparison since it will not perform implicit type coercions on the values. It also handles special cases for Numbers like NaN and zero-based values like +0 and -0 as well.

```
console.log(10 === '10');
// false

console.log('1' === 1);
// false

console.log(10 === 10);
// true

console.log({ a: 1 } === { a: 1 });
// false

const obj = { b: 2 };
console.log(obj === obj);
// true

console.log(NaN === NaN);
// false

const val = NaN;
console.log(val === val);
// false
```

```
console.log(-0 === +0);
// true

console.log(-0 === 0);
// true

console.log(-0 === -0);
// true
```

Object.is(): It is a superset of the triple equals operator, that is Object.is() also does not do implicit type coercions, and does not do any kind of special handling for NaN and zero-based values like -0 and +0.

```
console.log(Object.is(10, '10'));
// false

console.log(Object.is('1', 1));
// false

console.log(Object.is(10, 10));
// true

console.log(Object.is({ a: 1 }, { a: 1 }));
// false

const obj = { b: 2 };
console.log(Object.is(obj, obj));
// true

console.log(Object.is(NaN, NaN));
// true

const val = NaN;
```

```
console.log(Object.is(val, val));
// true

console.log(Object.is(-0, +0));
// false

console.log(Object.is(-0, 0));
// false

console.log(Object.is(-0, -0));
// true
```

The `==` and `===` operator treats the number values +0 and -0 as equal and also comparing two NaN values as different whereas the `Object.is()` method treats them differently.

If you want to check more such examples of different value comparisons, checkout this table here

Implementation

Let's note down the points we need to take care of while implementing comparison logic for our custom `Object.is()` method:

1. If both the values are undefined.
2. If both the values are null.
3. If both the values are true or false.
4. If both the strings are of the same length with the same characters and in the same order.

5. For reference based values like object or arrays, if both the values point to the same reference in memory, only then both are the same value.
6. If both the values are numbers and both are “+0” or both are ‘-0’.
7. If both the values are numbers and both are “NaN” or both non-zero and both not NaN and both have the same value.

Points 1-5 can be handled using the triple equals operator `===` itself. We only need to additionally handle for points 6 and 7.

Let’s checkout the implementation in code:

```
function is(a, b) {
   // Handle special cases in numbers like NaN and zero-based
value (+0 or -0)
   if (typeof a === 'number' && typeof b === 'number') {
       if (Number.isNaN(a) && Number.isNaN(b)) {
           return true;
       }

       // we first need to individually check if both `a` and
`b` are zero-based value `0` (be it -0 or +0, so for that we use
triple equals `===`)
       if (a === 0 && b === 0) {
           // We know that, 1/0 equals `Infinity` and 1/-0
equals `-Infinity`. The fact that both (Infinity and -Infinity)
will not be equal
           // For example, let's take the value for
Infinity=100, therefore -Infinity will be -100, and (100 ===
```

```
-100) is false
            return (1/a === 1/b);
        }
    }


    // If not a NaN or zero-based value (+0, or -0), we can use
the same triple equals `===`
    return a === b;
}
```

Bonus: We can also implement our simple `Number.isNaN()` method as well. Based on the general fact that two `NaN` values when compared are not equal.

```
function isNaN(val) {
    return typeof val === 'number' && val !== val;
}
```

Test Cases

```
console.log(is(10, '10'));
// false

console.log(is('1', 1));
// false

console.log(is(10, 10));
// true

console.log(is({ a: 1 }, { a: 1 }));
// false
```

```
const obj = { b: 2 };
console.log(is(obj, obj));
// true

console.log(is(NaN, NaN));
// true

const val = NaN;
console.log(is(val, val));
// true

console.log(is(-0, +0));
// false

console.log(is(-0, 0));
// false

console.log(is(-0, -0));
// true

console.log(is(0, +0));
// true
```

Implement Custom lodash _.partial()

Problem Statement

Implement a function `partial()` which is your own version of the lodash's `_.partial()` method.

Let's first understand what does partial function mean?

In simple terms -

Calling a function with fewer arguments than it expects and then it returns a new function that takes the remaining of the arguments. This is called Partial Application of Function.

```
// Syntax
const partialFn = partial(func, partialArgs);
partialFn(remainingArgs);
```

Example

Note the underscore, "_" passed as argument to the `partial()` function, this is a placeholder, which needs to be filled up with valid values.

In the returned partial function `partialSum` we pass in the remaining arguments to replace the placeholders, in this case '2'.

```
const sumOfThree = (a, b, c) => a + b + c;
const productOfFour = (a, b, c, d) => a * b * c * d;

const partialSum = partial(sumOfThree, 1, '_', 3);
console.log(partialSum(2));
// 6
```

Implementation

Let's get started with the implementation details for the problem:

- ➔Create a function `partial` which takes in the `mainFn` and the `partialArgs`. The `partialArgs` may contain placeholders "_".
- ➔Return a new function which takes in the `nextArgs`, whose values need to be replaced with the placeholders if it exists.
- ➔Then, take the remaining extra args from `nextArgs` and merge them all together as `args`.
- ➔Finally call the `mainFn` passing all the `args`.

```
// We know that functions are also objects, so adding the
placeholder property directly on the function object
partial.placeholder = '_';

function partial(mainFn, ...partialArgs) {
   return function(...nextArgs) {
       // tracks the current element's index in nextArgs
```

```
        let i = 0;

        const args = [...partialArgs];

        // Replace placeholders with values from `nextArgs`
        args.forEach((arg, index) => {
            if (arg === partial.placeholder) {
                args[index] = nextArgs[i++];
            }
        })

        // Add the `remainingArgs` to the `args`
        const remainingArgs = nextArgs.slice(i);
        const finalArgs = [...args, ...remainingArgs];
        console.log(finalArgs);

        // Call the `mainFn` with all the `finalArgs`
        return mainFn(...finalArgs);
    }
}
```

Test Cases

```
const sumOfThree = (a, b, c) => a + b + c;
const productOfFour = (a, b, c, d) => a * b * c * d;

const partialSum = partial(sumOfThree, 1, '_', 3);
console.log(partialSum(2));
// 6

const partialProduct = partial(productOfFour, '_', '_', 3, '_');
console.log(partialProduct(1, 2, 4));
// 24
```

Implement Custom lodash _.once()

Problem Statement

Implement a function `once()` which is your own version of the lodash's `_.once()` method. This method forces your function to be called only once. However, repeated calls to that function will return the same value returned in the first call.

```
// Syntax
const oncedFn = once(mainFn);
oncedFn(args);
```

Example

```
const sum = (a, b, c) => a + b + c;

const oncedSum = once(sum);
oncedSum(1, 2, 3);
// 6

oncedSum(3, 4, 5);
// 6
```

Implementation

Let's get started with the implementation details:

➔Create a function `once` which takes in the `mainFn` as input and returns a new function which is the onced version.

➔Use two variables, `result` to store the result of the first call and `isCalledOnce` to track if it's called once.

➔If not called even once, then call `mainFn` using the `call` method to maintain the "this" context and then cache the result for next repeated calls.

```
function once(mainFn) {
   // stores the `result` of mainFn when it was called for 1st
time
   let result;

   // Tracks if the `mainFn` was called once or not
   let isCalledOnce = false;

   return function (...args) {
       // If not called once, call it and store the result
       if (!isCalledOnce) {
           // using the `call` method of functions, so persist
the context of "this" which the function is called
           result = mainFn.call(this, ...args);
           isCalledOnce = true;
       }

       return result;
   }
}
```

Test Cases

```
const sum = (a, b, c) => a + b + c;

const oncedSum = once(sum);
oncedSum(1, 2, 3);
// 6

// ----------------------------------------

oncedSum(3, 4, 5);
// 6

// ----------------------------------------

oncedSum(10, 20, 30);
// 6

// ----------------------------------------

function greetWithFullName(lastName) {
   return `${this.firstName} ${lastName}`;
}

const oncedName = once(greetWithFullName);

const obj = { firstName: 'Peter', oncedName };
obj.oncedName('Parker');
// Peter Parker
```

Implement Custom trim() operation

Problem Statement

Implement a function `trim()` which is your own version of the `String.prototype.trim()` method. You should not use the in-built `trim()` method.

The `trim()` function will be passed a string value as input and we need to remove the whitespace characters from both ends of this string and return a new string, without modifying the original string.

```
// Syntax
trim(str);
```

Example

```
trim('   hello  world');
// hello  world

trim('\t   hello  world       ');
// hello  world
```

Implementation

Let's get started with the implementation details:

➔Create a function named `trim()` which takes a string value `str` as input.
➔We create a helper function `isWhiteSpaceCharacter` which tells us if a character is a whitespace character or not.
➔Then we keep iterating from the start of the string until we find any non-whitespace character.
➔Then we keep iterating from the end of the string until we find any non-whitespace character.
➔Finally, we just slice the string from start to end which contains no whitespace characters at the start or at the end of the string and just return it.

```
function trim(str) {
   const isWhiteSpaceCharacter = (ch) => {
       // these are some predefined white space characters which
needs to be trimmed off - empty character, empty space, tab
spaces, newlines, or any other kind of whitespace characters
       return ch === '' || ch === ' ' || ch === '\t' || ch ===
'\n' || ch === '\s';
   }

   let start = 0;
   let end = str.length-1;

   while (start < str.length &&
isWhiteSpaceCharacter(str.charAt(start))) {
       start++;
   }

   while (end > 0 && isWhiteSpaceCharacter(str.charAt(end))) {
       end--;
```

```
    }

    return str.slice(start, end+1);
}
```

Test Cases

```
trim('   hello  world');
// hello  world

trim('   hello  world       ');
// hello  world

trim('   hello  world\t ');
// hello  world

trim(`   hello

world       `);
/*
hello

world
*/

trim(`\n   hello\tworld   \t\t`);
// hello   world

trim(`\t   hello\nworld   \s`);
/*
hello
world
*/
```

```
trim('    \n \t   ');
// No output prints here (since the input contains only
whitespace character)
```

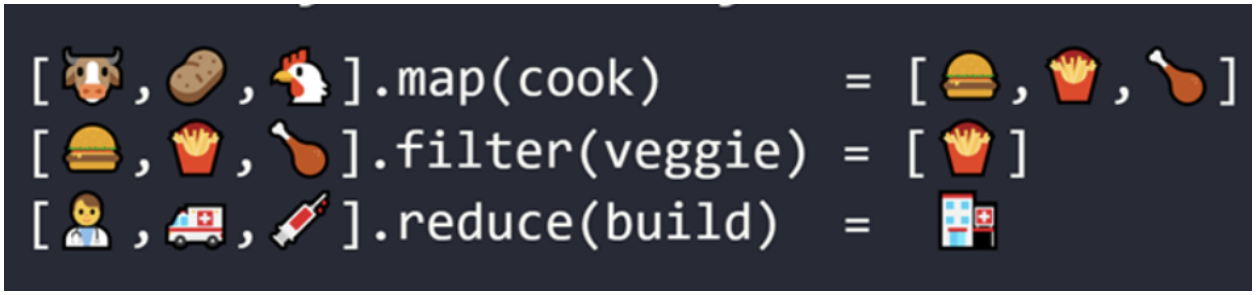
Implement Custom reduce() method

Problem Statement

Implement a function `reduce()` which is your own version of the `Array.prototype.reduce()` method. You should not use the built-in reduce method.

By definition -

The `reduce()` method is used to reduce the array values to a single value by executing the provided callback function for each value of the array (left to right) and the return value of that callback is stored in an accumulator.




```
// Syntax
reduce(callback, initialValue);

// The callback accepts 4 parameters as input
const callback = (accumulator, currentValue, index, array) => {}
```

- ➔initialValue - It is an optional parameter and is used to specify the value to be passed to the function as the initial value. If not passed, the first element of the array will be taken as the initialValue.
- ➔callback - It is a required parameter and is used to run for each element of the array, the callback also accepts 4 more parameters,
  - ◆accumulator - On the first call, its value is the initialValue (if specified), otherwise its value is array[0]. For subsequent calls, it will be the value resulting from the previous call of the `callback`.
  - ◆currentValue - It specifies the currentValue or current element from the array.
  - ◆index - It is the index of the currentValue from the array.
  - ◆array - It is the actual array on which the reduce method was called on.
- ➔In most cases, we would only be working with the `accumulator` and `currentValue` of the callback parameters.

Use Cases

Some of the most common use cases of using the `reduce()` method is summing an array into single value, promise sequencing, and function piping,..

Example

```
const arr = [1, 2, 3, 4, 5];
const initialValue = 10;

const sum = (accumulator, currentValue) => {
   const sum = accumulator + currentValue;
   return sum;
}

arr.reduce(sum, initialValue);
// 25
```

Implementation

Let’s get started with the implementation details:

- ➔Create a function `reduce()` and attach it to the `Array.prototype.reduce` directly, so that we can call the `reduce()` method directly over the array instance.
- ➔Store the `accumulator` initially with initialValue (if present, else select the first element of array)
- ➔Iterate on each element of the array and call the callback (with the help of `call()` method to maintain the context of "this") passing all the required parameters.
- ➔Finally, return away the `accumulator` which is the final result after processing each element of the arr.

```
Array.prototype.reduce = function reduce(callback, initialValue)
{
   // Variable that accumulates the result after performing
```

```
operation by executing the callback one-by-one on the array
elements
   let accumulator = initialValue;

   // Note: `this` will be the actual array on which the
`reduce()` method was called
   const arr = this;

   // Iterate on each element of arr
   for (let i = 0; i < arr.length; i++) {
       if (!accumulator) {
           // If initialValue was not passed, then accumulator
will be undefined, so we initialize it with first element of arr
           accumulator = arr[i];
       }
       else {
           // Now, process by executing the callback by passing
the each element to the array, and accumulating the result in
`accumulator`
           accumulator = callback.call(this, accumulator,
arr[i], i, arr);
       }
   }

   // Return the final accumulated result value
   return accumulator;
}
```

Test Cases

```
const arr1 = [1, 2, 3, 4, 5];
const initialValue1 = 10;

const sum = (accumulator, currentValue) => {
```

```
    const sum = accumulator + currentValue;
    return sum;
}

arr1.reduce(sum, initialValue1);
// 25

// ------------------------------------

// When `initialValue` is not passed

const arr2 = [1, 2, 3, 4, 5];
const initialValue2 = 10;

arr1.reduce((accumulator, currentValue) => {
    const sum = accumulator + currentValue;
    return sum;
});
// 15

// ------------------------------------

// Function piping using reduce

function increment(input) {
    return input + 1;
}

function decrement(input) {
    return input - 1;
}

function double(input) {
    return input * 2;
}
```

```
// Let's say we're give a number, which we need to increment
first, then decrement the result of that, then double the last
result
// So we just use simple arr, and add our methods
const arr3 = [increment, decrement, double];

const initialValue3 = 100;
const applyOperation = (accumulator, currentValue) => {
   // `currentValue` will the functions added in the array
   const result = currentValue(accumulator);
   return result;
}

arr3.reduce(applyOperation, initialValue3);
// 200
```

Implement Custom memoize() method

Problem Statement

Implement a function `memoize()` which is your own version of the lodash's `_.memoize()` method.

The `memoize()` function is used to memoize (or cache) the result of a function call which is passed as input.

The basic idea of memoization is for the same arguments passed, the function will always return the same result.

So, to cache we will use these arguments as keys. And we will map the key with the result value.

```
// Syntax
const memoizedFn = memoize(func, resolver);
memoizedFn(args);

const resolver = (...args) => args.join('_');
```

➔ func - the actual func whose results need to be memoized.

➔ resolver - an optional argument, which generates the key for the result to be cached, passing it as a function we can have more control on how we desire to generate the

key (I have kept it simple for demonstration where we just join all the arguments using underscore "_").

Note: If `resolver` function is not passed by default the first argument is taken as the key.

Example

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

const resolver = (...args) => args.join('_');

const memoizedSum = memoize(sum, resolver);
memoizedSum(1, 2, 3);
// 6

memoizedSum(1, 2, 3);
// 6 (value retrieved from cache)

memoizedSum(1, 2, 3);
// 6 (value retrieved from cache)

memoizedSum(2, 4, 3);
// 9 (since new args passed, new result is generated, and this
will be cached from now on)

memoizedSum(2, 4, 3);
// 9 (value retrieved from cache)
```

Implementation

Let’s get started with the implementation details:

- ➔Create a `memoize` function which takes in `mainFn` whose results need to be memoized and a resolver to generate key representation.
- ➔Return a function which is passed an arbitrary number of arguments.
- ➔If the `resolver` function exists, get key representation from it by passing all the `args`.
- ➔If the key is already present in cache means, the result was memoized, so return away.
- ➔Else call the `mainFn` and store the newly generated `result` in the cache and return away.

```
function memoize(mainFn, resolver) {
   // Initialise a plain object, which will be used as cache to
store the `result` of the `mainFn` call based on the arguments
passed to it
   const cache = {};

   return function (...args) {
       // Get the key representation

       // If resolver exists, it's generated based on args
passed, else assign the first arg as key
       const key = resolver ? resolver(...args) : args[0];

       if (cache.hasOwnProperty(key)) {
           // If `result` already exists in cache
           return cache[key];
       }
       else {
```

```
            // Else, calculate the `result` and store in the
cache
            // Using `call` method to maintain the context of
"this"
            const result = mainFn.call(this, ...args);
            cache[key] = result;
            return result;
        }
    }
}
```

Implement Custom memoizeLast() method

Problem Statement

Implement a function `memoizeLast()` which is used to memoize (or cache) only the latest or last calculated result based on the arguments.

Recommend solving the previous problem first for some basic understanding, 44. Implement Custom lodash _.memoize()

The reason we only cache the last result is that when we use a cache map or object, which stores the keys of all memoized results, it can get bloated.

As a result we choose to cache only the last results.

```
// Syntax
const memoizedLastFn = memoizeLast(func, isArgsEqual);
memoizedLastFn(args);
```

- ➔func - the actual func whose results need to be memoized.
- ➔isArgsEqual - an optional parameter, which is a function used to compare currently passed `args` and the last cached `lastArgs` and returns a boolean value indicating

if both are equal or not. If `isArgsEqual` is not passed while calling the `memoizeLast` function, it by default uses a comparison method (checkout this in the implementation section).

Earlier we had a `resolver` function which generates the key representation of the arguments. This `key` was used to check if the result exists in the cache or not.

Now, since we don't have the cache map or object to store all the key representations, we store the entire list of arguments array itself as a cache.

Next time, when the memoized function is called we use this `isArgsEqual` function to compare the values of current `args` with that of `lastArgs` to check if they are equal or not. If equal, return away the `lastResult` cached, else calculate the new result.

Example

```
const memoizedSum = memoizeLast(sum);
memoizedSum(1, 2, 3);
// 6 (since new args passed, new result is generated)

memoizedSum(1, 2, 3);
// 6 (value retrieved from cache)

memoizedSum(1, 2, 3);
// 6 (value retrieved from cache)
```

```
memoizedSum(2, 4, 3);
// 9 (since new args passed, new result is generated)

memoizedSum(2, 4, 3);
// 9 (value retrieved from cache)
```

Implementation

Let's get started with the implementation details:

- ➔Create a `memoizeLast` function which takes in `mainFn` whose results need to be memoized and a `isArgsEqual` function (by default we pass our `defaultIsArgsEqual` function).
- ➔`memoizeLast` returns a function which is passed an arbitrary number of arguments.
- ➔Declare two variables, `lastArgs` and `lastResult` which acts as our cache of storing the latest result and the respective latest args.
- ➔`isArgsEqual` is used to compare `lastArgs` and `args`.
  - ◆If the values of both args are equal, means the result is already memoized (cached)
  - ◆Else, calculate the new result by calling the `mainFn` and also store the current `args` and `result` and finally return the result.

```
function memoizeLast(mainFn, isArgsEqual = defaultIsArgsEqual) {
  // Store the `lastArgs` and `lastResult` when a latest
```

```
`mainFn` call was made
    let lastArgs = [];
    let lastResult;

    return function (...args) {
        // Check if the lastArgs and current args are equal
        if (isArgsEqual(args, lastArgs)) {
            // If equal, return memoized result `lastResult`
            return lastResult;
        }
        else {
            // Else, calculate the `result`
            // Store current `result`, `args` for `lastArgs`,
`lastResult`
            const result = mainFn.call(this, ...args);
            lastResult = result;
            lastArgs = args;

            return result;
        }
    }
}

const defaultIsArgsEqual = (args1, args2) => {
    // Since this is used as default, we just shallow compare
    // And check if all elements of both the args array are same
or not
    let isEqual = true;

    if (args1.length !== args2.length) {
        isEqual = false;
    }
    else {
        args1.forEach((value, index) => {
            if (isEqual && value !== args2[index]) {
                isEqual = false;
```

```
            }
        })
    }

    return isEqual;
}

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

Implement Custom call() method

Problem Statement

Implement a function `call()` which is your own version of the `Function.prototype.call` method. You should not use the in-built `call()` method.

The `call()` method takes an object as the first argument and sets it as the "this" context for the function which needs to be invoked.

The `call()` method then optionally takes any arbitrary number of arguments that needs to be passed to the function that needs to be invoked.

```
// Syntax
func.call(thisArg, arg1, arg2, ..., argN);
```

Concept

First, let's get some idea around why we would need this method, which is crucial before implementing this problem.

```
const spidermanObj = {
   firstname: "Peter",
   lastname: "Parker",
   printName() {
```

```
        return this.firstname + " " + this.lastname;
    }
}

console.log(spidermanObj.printName());
// Peter Parker

const ironmanObj = {
    firstname: "Tony",
    lastname: "Stark",
    printName() {
        return this.firstname + " " + this.lastname;
    }
}

console.log(ironmanObj.printName());
// Tony Stark
```

You notice that both the objects in the above code snippet have a similar structuring with common properties and common methods. Only change is in the value of the properties of those 2 objects.

The method `printName()` is doing the same job and has duplicated code, only the values would change when executed.

It doesn't seem to be a good practice. So, for that reason, I will define the `printName()` method as a separate method as in the below code snippet.

```
function printName() {
   // The "this" context is based on which objects calls the
method
   // For spidermanObj.printName(), "this" will be
`spidermanObj`
   // For ironmanObj.printName(), "this" will be `ironmanObj`
   return this.firstname + " " + this.lastname;
}

const spidermanObj = {
   firstname: "Peter",
   lastname: "Parker",
   printName
}

console.log(spidermanObj.printName());
// Peter Parker

const ironmanObj = {
   firstname: "Tony",
   lastname: "Stark",
   printName
}

console.log(ironmanObj.printName());
// Tony Stark
```

The results are the same, because the "this" context is defined by the object which calls the method.

Here’s where call comes into the picture, where we can directly pass the "this" context to the method `printName` or any other function with the help of `call()` method.

```
function printName(role) {
   // `this` context is defined by the `object` passed to the
call method
   return this.firstname + " " + this.lastname + " - " + role;
}

const spidermanObj = {
   firstname: "Peter",
   lastname: "Parker",
   printName
}

// Passing an argument `role` as well to the printName method
console.log(printName.call(spidermanObj, 'spiderman'));
// Peter Parker - spiderman

const ironmanObj = {
   firstname: "Tony",
   lastname: "Stark",
   printName
}

console.log(printName.call(ironmanObj, 'ironman'));
// Tony Stark - ironman
```

Implementation

```
Function.prototype.call = function myCall(thisArg, ...args) {
   // Get the actual function that needs to be invoked and on
which we need to set the "this" context
   const fn = this;

   // We know the fact that the "this" context for a method will
be set based on the object which calls it
```

```
    // `thisArg` will be our object or context we need to set
    const context = Object(thisArg);

    // Note: We wrap `thisArg` in an Object constructor, to
handle primitive values as well like null, undefined, number,
string, which returns their wrapper objects

    // Generate a unique key using `Symbol()` to avoid object
property conflicts
    const key = Symbol();

    // Set the invoking function `fn` as a value for the unique
`key` as a property on the object `context`
    context[key] = fn;

    // Now all we need to do is invoke `fn` via `context` object
and pass the additional args as well
    const result = context[key](...args);

    // Finally once we're done calling the function `fn`
    // We delete the property from the `context` obj to avoid
memory leaks
    delete context[key];

    // Return away the result generated by invoking the function
`fn`
    return result;
}
```

Test Cases

```
function printName(role) {
    return this.firstname + " " + this.lastname + " - " + role;
}
```

```
const spidermanObj = {
   firstname: "Peter",
   lastname: "Parker",
   printName
}

printName.call(spidermanObj, 'spiderman');
// Peter Parker - spiderman

const ironmanObj = {
   firstname: "Tony",
   lastname: "Stark",
   printName
}

printName.call(ironmanObj, 'ironman');
// Tony Stark - ironman
```

Implement Custom apply() method

Problem Statement

Implement a function `apply()` which is your own version of the `Function.prototype.apply` method. You should not use the in-built `apply()` method.

This problem is exactly the same as that of the previous problem, 46. Implement Custom call() method

Recommend you to go through that first and understand the concepts discussed.

The `apply()` method takes an object as the first argument and sets it as the "this" context for the function which needs to be invoked.

The only difference of `apply()` method to that of the `call()` method is -

The `apply()` method then optionally takes any arbitrary number of arguments in an array-like format that needs to be passed to the function that needs to be invoked.

```
// Syntax
func.call(thisArg, [arg1, arg2, ..., argN]);
```

We won't be discussing all the important concepts again, as we have discussed it already in the previous problem here, 46. Implement Custom call() method (Recommended to read out that first)

Implementation

```
// The only change in code is in the 1st line where we take `args` as it is and not in the rest operator format `...args` as in the call() method
Function.prototype.apply = function myApply(thisArg, args) {
   // Get the actual function that needs to be invoked and on which we need to set the "this" context
   const fn = this;

   // We know the fact that the "this" context for a method will be set based on the object which calls it
   // `thisArg` will be our object or context we need to set
   const context = Object(thisArg);

   // Note: We wrap `thisArg` in an Object constructor, to handle primitive values as well like null, undefined, number, string, which returns their wrapper objects

   // Generate a unique key using `Symbol()` to avoid object property conflicts
   const key = Symbol();

   // Set the invoking function `fn` as a value for the unique `key` as a property on the object `context`
   context[key] = fn;

   // Now all we need to do is invoke `fn` via `context` object and pass the additional args as well
```

```
    const result = context[key](...args);

    // Finally once we're done calling the function `fn`
    // We delete the property from the `context` obj to avoid
memory leaks
    delete context[key];

    // Return away the result generated by invoking the function
`fn`
    return result;
}
```

Test Cases

```
function printName(role, city) {
    return `${this.firstname} ${this.lastname} - ${role} -
${city}`;
}

const spidermanObj = {
    firstname: "Peter",
    lastname: "Parker",
    printName
}

printName.apply(spidermanObj, ['spiderman', 'NY']);
// Peter Parker - spiderman - NY

const ironmanObj = {
    firstname: "Tony",
    lastname: "Stark",
    printName
}
```

```
printName.apply(ironmanObj, ['ironman', 'NJ']);
// Tony Stark - ironman - NJ
```

Implement Custom bind() method

Problem Statement

Implement a function `bind()` which is your own version of the `Function.prototype.bind` method. You should not use the in-built `bind()` method.

This problem is exactly the same as those of the previous problems, 46. Implement Custom call() method and 47. Implement Custom apply() method

Recommend you to go through those first and understand the concepts discussed.

The `bind()` method takes an object as the first argument and sets it as the "this" context (kind of permanently) for the function which needs to be invoked.

The only difference of `bind()` method to that of the `call()` method is -

Basically it will return a newly bound function with the "this" context passed, so whenever the bound function is invoked, automatically the "this" context set on it will be applied.

The `bind()` method then optionally takes any arbitrary number of arguments that needs to be passed to the function that needs to be invoked.

Example

```
function printName(role, city) {
   return `${this.firstname} ${this.lastname} - ${role} -
${city}`;
}

const spidermanObj = {
   firstname: "Peter",
   lastname: "Parker",
   printName
}

const printSpiderman = printName.bind(spidermanObj, 'spiderman',
'NY');

printSpiderman();
// Peter Parker - spiderman - NY

printSpiderman();
// Peter Parker - spiderman - NY

const ironmanObj = {
   firstname: "Tony",
   lastname: "Stark",
   printName
}

const printIronman = printName.bind(ironmanObj, 'ironman',
'NJ');
```

```
console.log(printIronman());
// Tony Stark - ironman - NJ

console.log(printIronman());
// Tony Stark - ironman - NJ
```

We won't be discussing all the important concepts again, as we have discussed it already in the previous problem here, 46. Implement Custom call() method (Recommended to read out that first)

Implementation

```
Function.prototype.bind = function myBind(thisArg, ...args) {

   // Get the actual function that needs to be invoked and on which we need to set the "this" context
   const fn = this;

   // We know the fact that the "this" context for a method will be set based on the object which calls it
   // `thisArg` will be our object or context we need to set
   const context = Object(thisArg);

   // Note: We wrap `thisArg` in an Object constructor, to handle primitive values as well like null, undefined, number, string, which returns their wrapper objects

   // Generate a unique key using `Symbol()` to avoid object property conflicts
   const key = Symbol();
```

```
    // Set the invoking function `fn` as a value for the unique
`key` as a property on the object `context`
    context[key] = fn;

    // This is the only major change we did, by wrapping it in a
new function
    return function() {
        // Now all we need to do is invoke `fn` via `context`
object and pass the additional args as well
        const result = context[key](...args);

        // We won't be deleting `context[key]` since this
returned function can be invoked many times, so we need to
persist it

        // Return away the result generated by invoking the
function `fn`
        return result;
    }
}
```

Implement Custom React classnames library

Problem Statement

Implement a function `classnames()` which is a utility method used to conditionally join classNames together separated by space characters.

This method is available as a famous package named classnames which is used in most React projects.

Concept

In React, we use the `className` prop to set the class names for the elements.

```
<div className="class1 class2 class3">
  Some demo content
</div>
```

But, what in case class names are added conditionally or generated dynamically? Then, we would do something like this, which is kinda verbose..

```
<div className={`class1  ${shouldAddClass2 ? "class2" : ""}`}>
  Some demo content
```

```
</div>
```

classnames function simplifies all of that for us.

The classnames() function accepts any arbitrary number of javascript arguments as input, then filters out the falsy values, and generates the final string for className.

Example

```
classnames('foo', 'bar');
// 'foo bar'

classnames('foo', { bar: true });
// => 'foo bar'

classnames({ 'foo-bar': true });
// 'foo-bar'

classnames({ 'foo-bar': false });
// ''

classnames({ foo: true }, { bar: true });
// 'foo bar'

classnames({ foo: true, bar: true });
// 'foo bar'
```

Implementation

Let’s get started with the implementation details:

First we need to know some facts based on which we must implement.

- ➔Declare an array `results` to which we push all the valid individual class names.
- ➔String and Number arguments are taken as valid class names - `classnames('foo', 'bar', 10)`.
- ➔Other primitive arguments are ignored like null, undefined, Symbol(), BigInt, booleans.
- ➔For Objects, we only take the string-based keys as valid classnames, if and only if the value for that property is a truthy value.
- ➔For Arrays, we loop through and recursively evaluate each element of the array.

```
function classnames(...args) {
   // accumulate all of the individual classnames tokens in
`results`. Finally we can just join all individual classname
tokens with empty space
   const results = [];

   args.forEach(arg => {
       // If the `arg` is a falsy value just return away (null,
false, '', undefined)
       if (!arg) return;

       // Handling for string or number types - push directly to
results
       if (typeof arg === 'string' || typeof arg === 'number') {
           results.push(arg);
           return;
       }
```

```
        // Handling for Arrays
        if (Array.isArray(arg)) {
            // If arg is an array, we loop through each value
recursively calling `classnames`
            for (let val of arg) {
                if (val) results.push(classnames(val));
            }
            return;
        }

        // Handling for Objects
        // If arg is an Object, we loop through each property
        // Note : for...in loop only loops through "string" based
keys on the object
        for (let key in arg) {
            const val = arg[key];
            // If `val` is a truthy value like true, non-empty
strings, object/array reference, then push the `key` to results
            if (val) results.push(key);
        }
    })

    // Finally join the classnames token with space separated
    return results.join(' ');
}
```

Test Cases

```
classnames('hide', 'minimize', 100);
// 'hide minimize 100'

// ---------------------------------
```

```
classnames('hide', 'maximize', false);
// 'hide maximize'

// --------------------------------

classnames('show', 'maximize', false, 200);
// 'show maximize 200'

// --------------------------------

const input1 = ['container', { visible: true, 'width-300': 300
}, { height: null }];
classnames(input1);
// 'container visible width-300'

// --------------------------------

const input2 = ['child-container', { visible: false,
'width-300': 300 }, [ { isMobile: 'yes', responsive: undefined
}, { 'margin-5': true } ]];

classnames(input2);
// 'child-container width-300 isMobile margin-5'

// --------------------------------

const input3 = ['child-container', { visible: false,
'width-300': 300 }, [ { isMobile: 'yes', responsive: undefined
}, { 'margin-5': true } ]];

classnames('toggled', input3, [ { isWrapped: true } ], {
'color-warning': undefined });
// 'toggled child-container width-300 isMobile margin-5
isWrapped'
```

Implement Custom Redux used "Immer" library

Problem Statement

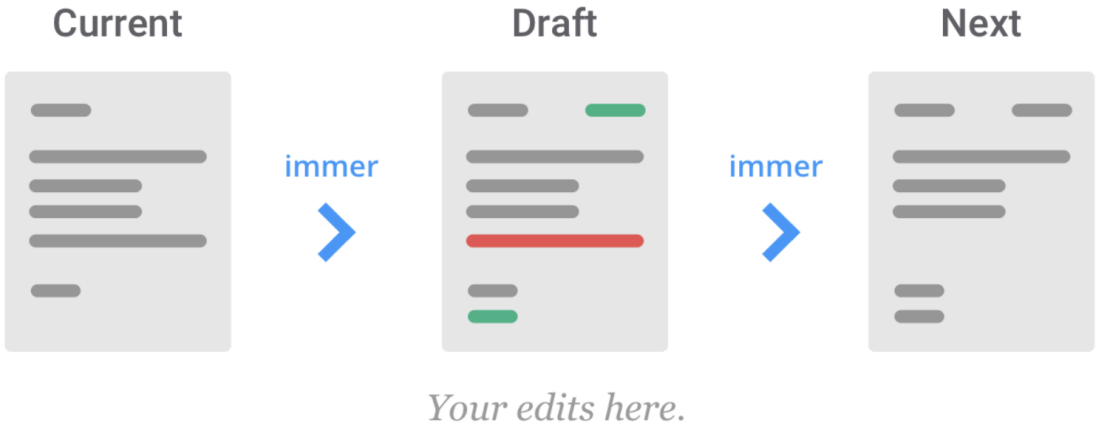
Implement a function `produce()` which is a utility method provided by the Immer Library.

Let's first understand what "Immer" does. In general context, Immer is a library which basically allows you to work with immutable states in a convenient way.

Redux depends on this library heavily, and is used in every React based project which uses Redux.

It’s highly encouraged to read this single page before proceeding on to implementing the solution - Understand in simple terms what Immer really does.

The basic idea is that with Immer you will apply all your changes to a temporary draft, which is a proxy of the currentState. Once all your mutations are completed, Immer will produce the nextState based on the mutations to the draft state. This means that you can interact with your data by simply modifying it while keeping all the benefits of immutable data.



The `produce()` takes a base state, and a `recipe` function which will receive an exact copy of the baseState called `draft` as an argument on which the desired mutations can be performed.

Finally, the `produce()` method also returns a `newState` on which all the modifications / changes / mutations will reflect. The interesting thing about Immer is that the `baseState` will be untouched, but the `nextState` will reflect all changes made to draftState.

```
// Syntax
const nextState = produce(baseState, (draft) => {
   // Your mutation logic on the draft goes here
});
```

➔baseState - the initial immutable state passed to `produce` method.
➔recipe - the second argument of produce, that captures

how the base state should be "mutated".

➔ draft - the first argument of any recipe, which is a proxy to the original base state that can be safely mutated.

➔ produce - function that records the mutations performed on the `draft` and applies the modification on the `nextState` and returns it.

Example

```
const baseState = {
   a: 1,
   b: {
       c: 3,
       d: 4,
       e: 5,
   },
   g: {
       h: 7,
       i: {
           j: [1, 2, 3],
           k: 10,
       },
   },
};

const nextState = produce(baseState, (draft) => {
   draft.b.c = 'Hello'
});


// newState modified
console.log(nextState.b);
// { c: 'Hello', d: 4, e: 5 }
```

```
// baseState will be as it is "unmodified"
console.log(baseState.b);
// { c: 3, d: 4, e: 5 }

// Modified property `b` will have new reference
console.log(baseState.b === nextState.b);
// false

// Since child is modified `b` parent will also have new
reference
console.log(baseState === nextState);
// false

// Unmodified property `g` remains as it is with same reference
console.log(baseState.g === nextState.g);
// true
```

Implementation

Let's get started with the implementation details:

- ➔ Create a `produce()` function which takes in the baseState, and a function recipe.
- ➔ In the recipe function, first create a clone copy of `baseState`.
- ➔ Compare for any modifications / mutations,
  - ◆ If no mutations then return the same `baseState`.
  - ◆ Else validate the mutations, recursively iterate on each property of the `clone`
    - If a property has no mutations replace it with the same `key` reference of the `baseState`.

- Else, keep validating for modifications for each property recursively.

➔Finally, return the cloned state, which is validated and has all the mutations reflected.

```
function produce(base, recipe) {
   // First we need to create a clone of the `base` state so
that the modifications are performed only on the clone and not
on the original `base` state
   const clone = JSON.parse(JSON.stringify(base));

   // Pass the cloned state to the `recipe` callback on which
the modifications will be made
   recipe(clone);

   // Next we need to apply our logic to find if any
modifications were made or not.


   // We use JSON.stringify to serialize the data and do string
comparison, else can also use `deepEqual` utility as in (Ques.
19)
   if (JSON.stringify(base) === JSON.stringify(clone)) {
       // If both are same means no modifications made, return
`base` state as it is
       return base;
   }
   else {
       // If modifications are performed, instead of returning
the entire `clone` as it is, we need to validate the
modifications
       validateModifications(base, clone);
   }
```

```
    return clone;
}

function validateModifications(base, clone) {
    if (typeof base !== 'object' || typeof clone !== 'object') {
        // Handling for primitive types
        return (base === clone) ? base : clone;
    }

    // We need to specially handle `null` values since typeof
null is also an "object"
    if (base === null || clone === null) {
        return (base === clone) ? base : clone;
    }

    // Iterate on each of object/Array properties
    const keys = Object.keys(clone);

    for (let key of keys) {
        // Means the value for the `key` property is "unmodified"
        if (JSON.stringify(clone[key]) ===
JSON.stringify(base[key])) {
            // Just replace the cloned property `clone[key]`,
with the original unmodified property itself `base[key]`
            clone[key] = base[key];
        }
        else {
            // For modified properties, we recursively validate,
and apply modifications only where required
            validateModifications(base[key], clone[key]);
        }
    }
}
```

Test Cases

```
const baseState = {
   a: 1,
   b: {
       c: 3,
       d: 4,
       e: 5,
   },
   g: {
       h: 7,
       i: {
           j: [1, 2, 3],
           k: 10,
       },
   },
};

const nextState1 = produce(baseState, (draft) => {
   draft.a = 'Hello'
   draft.g.i.j[2] = 100;
})
console.log(nextState1);
/*
{
 a: 'Hello',
 b: { c: 3, d: 4, e: 5 },
 g: { h: 7, i: { j: [ 1, 2, 100 ], k: 10 } }
}
*/


const nextState2 = produce(baseState, (draft) => {
   draft.b = null
})
console.log(nextState2);
/*
```

```
{
 a: 'Hello',
 b: null,
 g: { h: 7, i: { j: [ 1, 2, 100 ], k: 10 } }
}
*/
```

Implement Custom Virtual DOM - I (Serialize)

Problem Statement

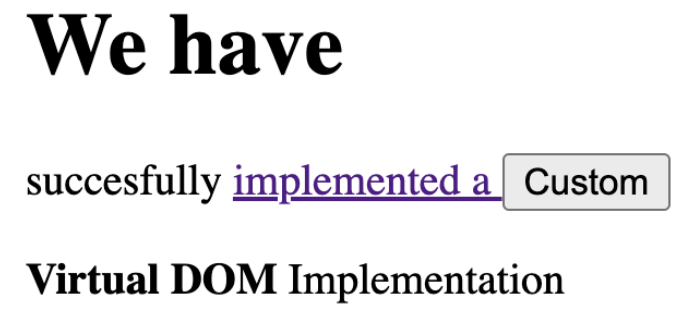
Implement a function `virtualDOM()` which serializes the real DOM structure to a Javascript based representation of Virtual DOM just as React does.

Example

You will be given a real DOM structure something like this:

```
<div id="root">
   <h1>We have</h1>
   <div class="container">
       successfully
       <a href="https://google.com" target="_blank">
           implemented a
       </a>
       <button>
           Custom
       </button>
       <p>
           <strong class="bold">Virtual DOM</strong>
           <span>Implementation</span>
       </p>
   </div>
</div>
```

The DOM structure appears to be something like this on the Browser:



For which we need to serialize it to a Javascript based representation of Virtual DOM something like this:

```
{
   type: "div",
   props: {
       id: "root",
       children: [
           {
               type: "h1",
               props: {
                   children: "We have"
               }
           },
           {
               type: "div",
               props: {
                   className: "container",
                   children: [
                       "successfully",
```

```
                {
                    type: "a",
                    props: {
                        href: "https://google.com",
                        target: "_blank",
                        children: "implemented a"
                    }
                },
                {
                    type: "button",
                    props: {
                        children: "Custom"
                    }
                }
            ]
        }
    },
    {
        type: "p",
        props: {
            children: [
                {
                    type: "strong",
                    props: {
                        className: "bold",
                        children: "Virtual DOM"
                    }
                },
                {
                    type: "span",
                    props: {
                        children: "Implementation"
                    }
                }
            ]
        }
```

```
            }
        ]
    }
}
```

The basic idea of the above representation boils down to:

```
{
   type: 'div', // signifies the type of DOM Node
   props: {
       // Lists all the properties on the DOM Node (attributes,
children nodes)

       // ...
   }
}
```

Implementation

Let's get started with the implementation details:

Creating the virtualDOM function which takes in the root node as input and returns a Javascript value which will contain the Virtual DOM representation.

```
const root = document.getElementById('root');

function virtualDOM(root) {
   // Create a `virtualTree` which will contain the entire
serialized representation of the DOM tree
   // The two important properties of this `virtualTree` are
```

```
`type` and `props`
   // `type` - signifies the type of DOM node
   // `props` - signifies all properties (attributes) attached
on DOM node

   const virtualTree = {};

   // Logic to serialize DOM Tree

   return virtualTree;
}
```

Add the `type` of the node on the `virtualDOM` using the `tagName`property of the node.

```
const root = document.getElementById('root');

function virtualDOM(root) {
   // Now let's define the type of DOM node
   // `tagName` property of the DOM node tells the type - div,
span, a, p..
   const virtualTree = {
       type: root.tagName.toLowerCase()
   };

   // Logic to serialize DOM Tree

   return virtualTree;
}
```

Now, the first step is to process all the attribute based properties on the node. Iterate on each individual attribute and attach it to the props object.

```
const root = document.getElementById('root');

function virtualDOM(root) {
   const virtualTree = {
       type: root.tagName.toLowerCase(),
   };

   // Now let's check for the properties on the DOM node
   const props = {};

   // 1. First we check for attribute based props - class, href,
id, name..

   // If the root node has any attributes attached on it
   if (root.hasAttributes()) {
       const attributes = root.attributes;

       // Iterate over each attribute
       for (let i = 0; i < attributes.length; i++) {
           // Get name and value for the attribute
           const name = attributes[i].name;
           const value = attributes[i].value;

           // Assign the attribute (so called property on the
props object)
           // A special check only for `class` attribute
           if (name === 'class') props['className'] = value;
           else props[name] = value;
       }
   }

   // Assign the props to the `virtualTree`
   virtualTree.props = props;

   return virtualTree;
```

```
}
```

Now, the last step is to process all the child nodes on the node. Iterate on each individual child node. We differentiate between the text nodes and Element nodes.

Incase of text nodes we can directly push to `children` array.

Incase of element nodes, we must recursively process to get its own virtual DOM representation and then we can push that result directly to the `children` array.

Finally, just attach the children property in the props object and attach it to the virtualDOM object itself.

```
const root = document.getElementById('root');

function virtualDOM(root) {
   const virtualTree = {
       type: root.tagName.toLowerCase(),
   };

   const props = {};

   if (root.hasAttributes()) {
       const attributes = root.attributes;

       for (let i = 0; i < attributes.length; i++) {
           const name = attributes[i].name;
           const value = attributes[i].value;
```

```
            if (name === 'class') props['className'] = value;
            else props[name] = value;
        }
    }

    // 2. Second we check for children based props - text nodes
or other HTML elements

    const children = [];

    // If the root node has child nodes
    if (root.childNodes.length > 0) {
        for (let i = 0; i < root.childNodes.length; i++) {
            const node = root.childNodes[i];

            // If the node is of type `text`
            if (node.nodeType === Node.TEXT_NODE) {
                // Trim the whitespaces and replace newline
characters with empty string
                const text =
node.textContent.trim().replace('\n', '');
                if (text.length > 0) children.push(text);
            }
            else if (node.nodeType === Node.ELEMENT_NODE) {
                // Else if a DOM Element node, then recursively
process and push the final result to the children array
                children.push(virtualDOM(node));
            }
        }
    }

    // Assign the children in the props
    if (children.length > 0) {
        // For a single child node, just attach the 1st elem
instead of the whole arr
        props.children = (children.length === 1) ? children[0] :
```

```
children;
    }

    // Assign the props to the `virtualTree`
    virtualTree.props = props;

    return virtualTree;
}
```

And we're done with the implementation of serializing a DOM structure to a Virtual DOM representation.

Implement Custom Virtual DOM - II (Deserialize)

Problem Statement

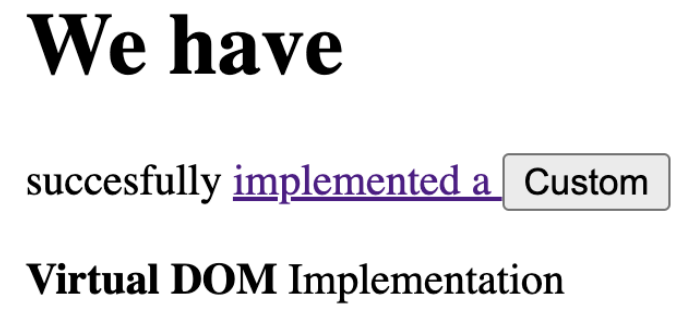
Implement a function `render()` which deserializes the VirtualDOM structure to a real DOM structure which contains rendered elements and nodes.

Example

Earlier we had a DOM structure something like this:

```
<div id="root">
   <h1>We have</h1>
   <div class="container">
       successfully
       <a href="https://google.com" target="_blank">
           implemented a
       </a>
       <button>
           Custom
       </button>
       <p>
           <strong class="bold">Virtual DOM</strong>
           <span>Implementation</span>
       </p>
   </div>
</div>
```

The DOM structure appears to be something like this on the Browser:



For which the Virtual DOM structure appears to be something like this:

```
{
   type: "div",
   props: {
       id: "root",
       children: [
           {
               type: "h1",
               props: {
                   children: "We have"
               }
           },
           {
               type: "div",
               props: {
                   className: "container",
                   children: [
                       "successfully",
```

```
                {
                    type: "a",
                    props: {
                        href: "https://google.com",
                        target: "_blank",
                        children: "implemented a"
                    }
                },
                {
                    type: "button",
                    props: {
                        children: "Custom"
                    }
                }
            ]
        }
    },
    {
        type: "p",
        props: {
            children: [
                {
                    type: "strong",
                    props: {
                        className: "bold",
                        children: "Virtual DOM"
                    }
                },
                {
                    type: "span",
                    props: {
                        children: "Implementation"
                    }
                }
            ]
        }
```

```
            }
        ]
    }
}
```

This we need to actually deserialize the VirtualDOM and convert it back to HTML elements and nodes and finally render it.

Implementation

Let's get started with the implementation details:

Create a function `render()` which implements the logic to render the vDOM to real DOM elements.

For the implementation to get the virtual DOM representation, check out the previous problem 51. Implement Custom Virtual DOM - I (Serialize) and copy-paste the same method here.

```
function render(vDOM) {
    // ...

    // Your Logic to render the elements from vDOM
}

const root = document.getElementById('root');
const vDOM = virtualDOM(root);

// Pass the vDOM to the `render()` method and get the rendered
element
```

```
const renderedElement = render(vDOM);

// Append the rendered element to the real DOM in the `body` as
child
document.body.appendChild(renderedElement);
```

Destructure `type` and `props` for current element and create the element based on the `type` named root and finally return it to be rendered.

```
function render(vDOM) {
   // Destructure the `type` and `props` for the current element
which needs to be rendered
   const { type, props } = vDOM;

   // Create the `root` node element of the specific `type`
   const root = document.createElement(type);

   // You logic to process the props goes here

   return root;
}

const root = document.getElementById('root');
const vDOM = virtualDOM(root);
const renderedElement = render(vDOM);
document.body.appendChild(renderedElement);
```

Now, the first step is to process the attributes based props. We iterate on each attribute and set it on the root element.

```
function render(vDOM) {
```

```
    const { type, props } = vDOM;

    const root = document.createElement(type);

    // 1. First we will process the attribute based props

    // Destructure the `children` prop and `restProps`
    const { children, ...restProps } = props;

    // `restProps` are basically the attribute based props on the
element
    const attrProps = restProps;

    // Iterate on each of the attributes and set it on the root
element which needs to be rendered
    for (let attr of attrProps) {
        const value = attrProps[attr];
        root.setAttribute(attr, value);
    }

    return root;
}

const root = document.getElementById('root');
const vDOM = virtualDOM(root);
const renderedElement = render(vDOM);
document.body.appendChild(renderedElement);
```

The second step is to process the children node, there can be more than 1 child or just a single child as well.

A child can also be a text node or an element representation. Text nodes are handled as the base case itself.

For more than 1 child, we recursively process each child iterating one-by-one and generate the rendered element which will be appended as child to the root.

```
function render(vDOM) {
   const { type, props } = vDOM;

   // If the current vDOM is a text node, we return it right
away creating a text node
   if (isTextNode(vDOM)) {
       const element = document.createTextNode(vDOM);
       return element;
   }

   const root = document.createElement(type);

   const { children, ...restProps } = props;

   const attrProps = restProps;
   for (let attr in attrProps) {
       const value = attrProps[attr];
       root.setAttribute(attr, value);
   }

   // 2. Second we will process the children nodes

   // If we have more than 1 child node, iterate on each child
   if (Array.isArray(children)) {
       for (let child of children) {
           // Recursively process and render the child elements
           const element = render(child);
           root.appendChild(element);
       }
   }
   else {
```

```
        // Recursively process and render the child elements
        const element = render(children);
        root.appendChild(element);
    }

    return root;
}

function isTextNode(vDOM) {
    // We can differentiate a text node if the child doesn't have
the `type` key on it
    return typeof vDOM !== 'object';
}

// const root = document.getElementById('root');
const vDOM = virtualDOM(root);
const renderedElement = render(vDOM);
document.body.appendChild(renderedElement);
```

And we're done with the implementation of deserializing a Virtual DOM representation to a real DOM structure which can be rendered.

The final output on your browser screen should be the same as shown below.

We have

succesfully implemented a Custom

Virtual DOM Implementation

We have

succesfullyimplemented a Custom

Virtual DOM Implementation

Implement Memoize/Cache identical API calls

Problem Statement

Implement a function `createAPICallCaching()` that caches the identical API calls.

In a more complicated frontend application, there is a high possibility multiple API calls will be made, and also the API calls can be identical with the same API path or configs, which may lead to duplicated API requests being made.

You want to avoid the unnecessary API calls, based on the following assumption that `GET` API call response hardly changes within 1000ms.

So identical GET API calls should return the same response within 1000ms. By identical, it means path and config are deeply equal.

```
// Syntax
const getAPIWithCaching = createAPICallCaching();
const responsePromise = getAPIWithCaching('/search', { key1:
'youtube', key2: 'videos' });
responsePromise
.then(() => {})
.catch(() => {})
```

`createAPICallCaching` returns a new function `getAPIWithCaching` which makes the API call on behalf of us and implements all of the caching logic.

This returned function `getAPIWithCaching` again takes in two parameters, `path` and `config`.

- ➔path - is basically the path of the API Endpoint or the URL.
- ➔config - is the API or URL search params.

Example

```
const getAPIWithCaching = createAPICallCaching();

getAPIWithCaching('/search', { keyword: 'abc'});
// 1st call, this will call the API

getAPIWithCaching('/search', { keyword: 'abc'});
// 2nd call is identical to 1st call, API won't be called again,
instead returns the result of the 1st call

getAPIWithCaching('/search', { keyword: 'abc'});
// 3rd call is not identical, this will call the API

// after 1000ms
setTimeout(() => getAPIWithCaching('/search', {keyword: 'abc'}),
1000);
// 4th call is identical to 1st call,
// but since after 1000ms, the API call will be made again
```

Understanding the caching mechanism:

```
const getAPIWithCaching = createAPICallCaching();

getAPIWithCaching('/search1', { keyword: 'abc'});
// 1st call, call callAPI(), add a cache entry

getAPIWithCaching('/search2', { keyword: 'abc'});
// 2nd call, call callAPI(), add a cache entry

getAPIWithCaching('/search3', { keyword: 'abc'});
// 3rd call, call callAPI(), add a cache entry

getAPIWithCaching('/search4', { keyword: 'abc'});
// 4th call, call callAPI(), add a cache entry

getAPIWithCaching('/search5', { keyword: 'abc'});
// 5th call, call callAPI(), add a cache entry

getAPIWithCaching('/search6', { keyword: 'abc'});
// 6th call, call callAPI(), add a cache entry
// cache of 1st call is removed

getAPIWithCaching('/search1', { keyword: 'abc'});
// identical with 1st call, but cache of 1st call is removed
// new cache of entry is added
```

Implementation

Let’s get started with the implementation details:

We define a function `createAPICallCaching()` which returns us a new function `getAPIWithCaching()` which handles all of

the logic to make API calls and returns the API response, alongside applying the caching logic for us.

```
// Create a function `createAPICallCaching` which returns a function `getAPIWithCaching` which API calls with all of the caching logic
function createAPICallCaching() {
   // We make use of Javascript `Map` to cache our APIs
   const cache = new Map();

   // Maximum time limit upto which an API call can be cached
   const CACHE_TIME_LIMIT = 1000;

   // Maximum APIs that can be cached
   const CACHE_SIZE_LIMIT = 5;

   // return this function which performs the API calls for you with all the benefits of managing a cache
   return function getAPIWithCaching(path, config) {
       // Your logic to cache identical APIs goes here

       // ...
   }
}
```

Now the simplest step is to make the API call, for simplicity we have used a mock function `fetchAPIMock()` which mocks the API call with some delay.

```
function createAPICallCaching() {
   const cache = new Map();
   const CACHE_TIME_LIMIT = 1000;
   const CACHE_SIZE_LIMIT = 5;
```

```
    return function getAPIWithCaching(path, config) {
        // Make the API call
        const responsePromise = fetchAPIMock(path, config);

        // Return away the current API response
        return responsePromise;
    }

    // This function is just to mock API calls with some delay
    function fetchAPIMock(path, config) {
        return new Promise((resolve) => {
            setTimeout(() => resolve(Math.random() * 100), 1200)
        });
    }
}
```

Now comes the most important step which is implementing the caching logic.

The steps are pretty simple:

- ➔ Create a unique hashing logic which helps in identifying the identical API.
- ➔ After making an API call we create an entry in the Map for that API, mapping the hash as the key and the API details as the value.
- ➔ While calling for any next API call, we first check if it exists in cache,
  - ◆ If yes, and API start time is less than the time limit to keep in cache, then we return the same response,

else we delete it from the cache assuming it to be expired and so next make a new API call.

- Also at the end, we check that if the Cache size limit exceeds, we remove the first made out of cache to avoid over storage or bloating of our cache size.

```
function createAPICallCaching() {
   const cache = new Map();
   const CACHE_TIME_LIMIT = 1000;
   const CACHE_SIZE_LIMIT = 5;

   return function getAPIWithCaching(path, config) {
       // Get the hash using both path & config,
       const hash = getHashForAPI(path, config);

       // If API is already in `cache`
       if (cache.has(hash)) {
           const apiEntry = cache.get(hash);

           // If the API call is made within cache expiry time
limit, which is 1000ms
           if (Date.now() - apiEntry.startTime <=
CACHE_TIME_LIMIT) {
               // Return the same API result
               return apiEntry.promise;
           }

           // Since this API already is identical and already
present in cache, we delete this entry, and set a new entry
below
           cache.delete(hash);
       }

       const responsePromise = fetchAPIMock(path, config);
```

```
        // Create an entry in the `cache` map for the current API
call made <hash, apiDetails>
        cache.set(
            hash, // key
            { responsePromise, startTime: Date.now() } // value
        );

        // If cache size limit exceeds we delete an entry from
the start
        if (cache.size > CACHE_SIZE_LIMIT) {
            // Gets us the firstAPI's hash and delete this entry
from cache
            const firstAPIHash = cache.keys().next().value;
            cache.delete(firstAPIHash);
        }

        // Return away the current API response
        return responsePromise;
    }

    function getHashForAPI(path, config) {
        const keys = Object.keys(config);
        // We will construct the entire URL and use it as our
unique hashkey, to identify identical APIs

        // For path='/search', config={ key1: 'youtube', key2:
'videos' }
        // The hash is : '/search?key1=youtube&?key2=videos'
        const hash = path + keys.map(key =>
`?${key}=${config[key]}`).join(`&`);
        return hash;
    }

    function fetchAPIMock(path, config) {
        return new Promise((resolve) => {
```

```
            setTimeout(() => resolve(Math.random() * 100), 1200)
        });
    }
}
```

TestCases

You can just trace out the behavior of the code and check if our logic works the same as intended below.

```
const getAPIWithCaching = createAPICallCaching();

getAPIWithCaching('/search1', { keyword: 'abc'});
// 1st call, call callAPI(), add a cache entry

getAPIWithCaching('/search2', { keyword: 'abc'});
// 2nd call, call callAPI(), add a cache entry

getAPIWithCaching('/search3', { keyword: 'abc'});
// 3rd call, call callAPI(), add a cache entry

getAPIWithCaching('/search4', { keyword: 'abc'});
// 4th call, call callAPI(), add a cache entry

getAPIWithCaching('/search5', { keyword: 'abc'});
// 5th call, call callAPI(), add a cache entry

getAPIWithCaching('/search6', { keyword: 'abc'});
// 6th call, call callAPI(), add a cache entry
// cache of 1st call is removed

getAPIWithCaching('/search1', { keyword: 'abc'});
```

```
// identical with 1st call, but cache of 1st call is removed
// new cache of entry is added
```