Class 28 - Polyfills of Higher Order method

Agenda
Polyfills of higher Order Functions (Map ,Filter , reduce)
Interview Problem solving on Funtional Programming

title: Polyfills of Higher order Functions

polyfill refers to a piece of code (usually JavaScript) that implements features on web browsers that do not support those features natively. Polyfills allow web developers to use modern features of the JavaScript language and APIs, ensuring that their website or application works consistently across different browsers and environments.

title: Polyfill of Map method

Quick Revision of map Method

The map method in JavaScript creates a new array populated with the results of calling a provided function on every element in the calling array. It doesn't change the original array and is often used for transforming data.

Writing a Polyfill for the map

First let us the original map method

```
const arr = [1,2,4]
   const result = arr.map((item) => item * 2)
```

Hover over the map and see the callback function definition as below

```
2      const result = arr.map((item) => item * 2)
(method) Array<number>.map<number>(callbackfn: (value: number,
index: number, array: number[]) => number, thisArg?: any):
number[]
```

Recall sparse arrays from DSA

```
// Polyfill for Array.prototype.map
if (!Array.prototype.myMap) {
   Array.prototype.myMap = function(callback, thisArg) {
     // Step 1: Throw a TypeError if 'callback' is not a function
     if (typeof callback !== 'function') {
       throw new TypeError(callback + ' is not a function');
     }
      // Step 2: Create a new empty array for the results
     const val = new Array(this.length);
      // Step 3: Iterate over the array
     for (let i = 0; i < this.length; i++) {
       // Check if the index exists in the array to handle sparse arrays
       if (i in this) {
         // Step 4: Execute 'callback' for each element, considering 'thisArg'
         // Use a ternary operator to check if 'thisArg' is provided
         var context = thisArg ? thisArg : this;
         var mappedValue = callback.call(context, this[i], i, this);
          // Step 5: Push the result of the callback into the 'result' array
         val[i] = mappedValue;
       }
     }
      // Step 6: Return the new array
```

```
    return val;
  };
}

const arr = [1,2,4]
delete arr[1]
  const result = arr.map((item) => item * 2)
  const data = arr.myMap((item) => item * 2)
  console.log(result)
  console.log(data)
```

Explanation:

Feature Detection: Start by checking if Array.prototype.myMap doesn't already exist to avoid overwriting any native implementations.

Type Checking: If the provided callback is not a function, throw a TypeError to enforce correct usage.

Result Array: Initialize an empty array (result) to hold the new values produced by the callback function.

Iteration: Loop through each element of the array with a for loop.

Index Check: Use if (i in this) to account for sparse arrays, ensuring that only existing indexes are processed.

Callback Execution: Directly execute the callback function for each element using call. If thisArg is provided, it temporarily becomes the context (this) for the callback; otherwise, the current array (this)

serves as the context. The callback function receives three arguments:

The current element (this[i])
The current index (i)
The entire array (this)

Push to Result Array: Add the value returned by the callback to the result array.

Return New Array: After completing the iteration, return the result array, which now contains the transformed elements.

This polyfill replicates the functionality of the map method for environments where it's not natively supported,

// EXTRA - CASE WHERE THIS IS REQUIRED FOR MAP

```
const thisArg = { multiplier: 2 };
const ans = [1, 2, 3].map(function (element) {
 return element * this.multiplier;
}, thisArg); // `this` in the callback is set to `thisArg`

console.log(ans); // [2, 4, 6]
```

title: Polyfill of Filter method

Quick Revision of filter Method

The filter method in JavaScript creates a new array with all elements that pass the test implemented by the provided function. It applies a given function to each element of the array, and if the function returns true for an element, that element is included in the new array. The original array is not modified.

Writing a Polyfill for the filter

Here's how to implement a polyfill for Array.prototype.filter without using the call method, along with a detailed explanation for each line:

```
// Polyfill for Array.prototype.filter without using 'call'
if (!Array.prototype.myFilter) {
 Array.prototype.myFilter = function(callback, thisArg) {
   // Step 1: Throw a TypeError if 'callback' is not a function
   if (typeof callback !== 'function') {
     throw new TypeError(callback + ' is not a function');
   }

   // Step 2: Create a new empty array for the results
   var result = [];

   // Step 3: Iterate over the array
   for (var i = 0; i < this.length; i++) {
     // Check if the index exists in the array to handle sparse
arrays
     if (i in this) {
```

```
      // Define the context for the callback
      var context = thisArg ? thisArg : this;

      // Step 4: Execute 'callback' for each element, considering
'thisArg'
      // If 'callback' returns true, push the current element to
'result'
      if (callback.call(context, this[i], i, this)) {
        result.push(this[i]);
      }
    }
  }

  // Step 5: Return the new array
  return result;
 };
}
```

USAGE

```
// Usage
var numbers = [1, 2, 3, 4, 5];
var evenNumbers = numbers.myFilter(function(number) {
 return number % 2 === 0;
});
console.log(evenNumbers); // Output: [2, 4]
```

Explanation:

Feature Detection: Checks if Array.prototype.myFilter doesn't exist to avoid overwriting any native implementation.

Type Checking: Throws a TypeError if the provided callback is not a function to ensure proper usage.

Result Array: An empty array (result) is initialized to store elements that pass the callback function's test.

Iteration: The method iterates over each element of the array with a for loop.

Index Check: Using if (i in this) ensures that the callback is only applied to existing elements, which is particularly important for sparse arrays.

Callback Execution: The callback function is executed directly for each element. If thisArg is provided, it's used to set the context (this) for the callback; otherwise, the current array (this) serves as the context. The callback receives three arguments:

The current element (this[i])
The current index (i)
The entire array (this)

If the callback function returns true, indicating the current element passes the test, that element is added to the result array.

Return New Array: After iterating through all elements, the result array, now containing only elements that passed the test, is returned.

This polyfill allows for the functionality of the filter method to be used in environments where it is not natively supported, ensuring broader compatibility of web applications.

title: Polyfill of Reduce method

Quick Revision of reduce Method

The reduce method in JavaScript applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value. This method is often used for summing up all elements of an array, but it's versatile enough to handle tasks like building up a single object from an array of objects.

```
const arr = [1,2,3,4,5]
const sum = arr.reduce((acc, curr) => acc + curr, 0)
console.log(sum) // 15
```

Recall different cases of reduce like empty array with no initial value Non empty array without initial value

Implementation of the polyfill

```
// Polyfill for Array.prototype.reduce
if (!Array.prototype.myReduce) {
 Array.prototype.myReduce = function(callback, initialValue) {
```

```
  // Step 1: Throw a TypeError if 'callback' is not a function
  if (typeof callback !== 'function') {
    throw new TypeError(callback + ' is not a function');
  }

  // Step 2: Handle empty array with no initial value case
  if (this.length === 0 && arguments.length === 1) {
    throw new TypeError('Reduce of empty array with no initial
value');
  }

  // Step 3: Set the initial index and accumulator
  var accumulator = arguments.length >= 2 ? initialValue : this[0];
  var startIndex = arguments.length >= 2 ? 0 : 1;

  // Step 4: Iterate over the array
  for (let i = startIndex; i < this.length; i++) {
    // Check if the index exists in the array to handle sparse
arrays
    if (i in this) {
      // Step 5: Update the accumulator
      accumulator = callback(accumulator, this[i], i, this);
    }
  }

  // Step 6: Return the accumulated value
  return accumulator;
 };
}
```

Example for the second case check

```
const empty = []
const ans = empty.reduce((acc, curr) => acc + curr)
console.log(ans) // TypeError: Reduce of empty array with no initial value
```

Explanation:

Feature Detection: First, it checks if Array.prototype.myReduce does not already exist to avoid overriding any existing method.

Type Checking: If the callback argument is not a function, it throws a TypeError, ensuring that the method is called with appropriate arguments.

Handling Empty Arrays: If the method is called on an empty array without an initial value, it throws a TypeError, because in such cases, there's no value to return or to start the reduction with.

Initial Values: The accumulator starts with initialValue if provided; otherwise, it uses the first element of the array. The iteration starts from the first or second element accordingly.

Iteration and Accumulation: The method iterates over the array, updating the accumulator with the result of the callback function. The callback takes four arguments:

accumulator: The accumulator accumulates the callback's return values.
currentValue: The current element being processed in the array.
currentIndex: The index of the current element being processed.

array: The array reduce was called upon.
This step carefully considers only the elements present in the array, which handles cases of sparse arrays gracefully.

Return Value: After iterating through the array, the method returns the accumulated value.

This polyfill allows for the use of the reduce functionality in environments where it might not be natively supported, ensuring broader compatibility across different JavaScript environments.

title: Interview Problem 1

Problem 1

Flattening an Array

Problem Statement: Flatten Nested Array

You are given a nested array containing array elements. Your task is to write a JavaScript function to flatten this nested array into a single-dimensional array, where all the nested arrays' elements are moved to the top-level array.

For example, given the following nested array:

const nestedArray = [1, [2, 3], [4, [5, 6]]];
The flattened array should be:

[1, 2, 3, 4, 5, 6]

Your task is to implement a function flattenArray that takes the nested array as input and returns the flattened array.

Note:

Ensure that your solution works for arrays of any depth and can handle arrays with mixed data types.

Solution

1. Define the flattenArray function: First, we define a function named flattenArray which takes an array arr as its parameter.

```
function flattenArray(arr) {
 // Function implementation will go here
}
```

2. What array method can be useful to iterate over the input array and reduce it to single output array
3. Use the reduce method to iterate over the array: We'll use the Array.prototype.reduce() method to iterate over each element of the array.

```
function flattenArray(arr) {
 return arr.reduce((flatArray, item) => {
     // Logic will be implemented inside this callback function
 }, []);
}
```

4. Check if the current element is an array or not: Inside the callback function of reduce(), we check whether the current item is an array or not.

```
function flattenArray(arr) {
 return arr.reduce((flatArray, item) => {
     if (Array.isArray(item)) {
         // If it's an array, flatten it
     } else {
         // If it's not an array, push it to the flatArray
     }
 }, []);
}
```

5. What should be done if the array element is array again
6. Recursively flatten nested arrays: If the item is an array, we'll recursively call the flattenArray function on that nested array.

```
function flattenArray(arr) {
 return arr.reduce((flatArray, item) => {
     if (Array.isArray(item)) {
         flatArray.push(...flattenArray(item)); // Recursively flatten nested
arrays
     } else {
         flatArray.push(item); // If it's not an array, push it to the
flatArray
     }
     return flatArray;
 }, []);
}
```

7. Return the flattened array: Finally, we return the flatArray which contains all the elements of the nested array flattened into a single-dimensional array.
8.

```
function flattenArray(arr) {
 return arr.reduce((flatArray, item) => {
   if (Array.isArray(item)) {
     flatArray.push(...flattenArray(item)); // Recursively flatten nested
arrays
   } else {
     flatArray.push(item); // If it's not an array, push it to the flatArray
   }
   return flatArray;
 }, []);
}

// Example nested array
const nestedArray = [1, [2, 3], [4, [5, 6]]];

// Flatten the nested array
const flattenedArray = flattenArray(nestedArray);
console.log(flattenedArray); // Output: [1, 2, 3, 4, 5, 6]
```

title: Interview Problem 2

Problem Statement:
You are given an array of objects representing transactions made by customers. Each object contains the following properties:

customerId: Number, representing the unique ID of the customer.
amount: Number, representing the amount of the transaction.

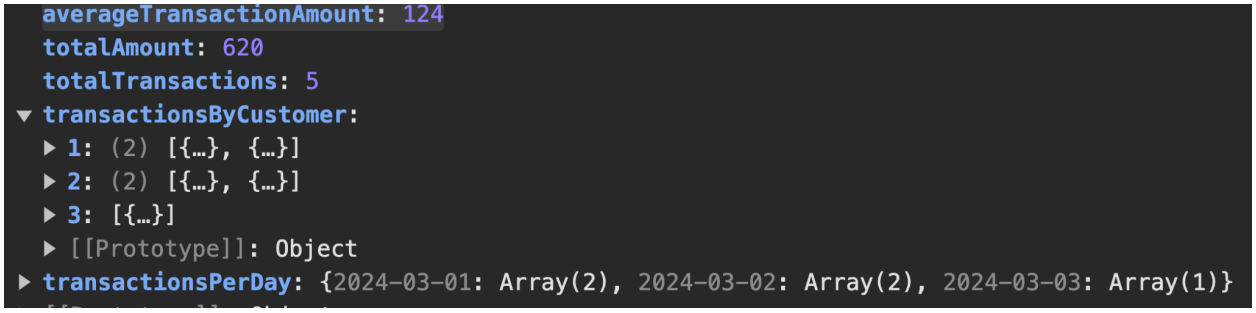
date: String, representing the date of the transaction (in the format "YYYY-MM-DD").
Your task is to write a JavaScript function using functional programming techniques that takes this array of transaction objects and returns an object containing the following information:

totalTransactions: Total number of transactions.
totalAmount: Total amount of all transactions.
averageTransactionAmount: Average amount of transactions.
transactionsPerDay: An object where keys are dates and values are arrays containing transactions made on that date.
transactionsByCustomer: An object where keys are customer IDs and values are arrays containing transactions made by that customer.

INPUT

```
const transactions = [
 { customerId: 1, amount: 100, date: '2024-03-01' },
 { customerId: 2, amount: 150, date: '2024-03-01' },
 { customerId: 1, amount: 200, date: '2024-03-02' },
 { customerId: 3, amount: 50, date: '2024-03-02' },
 { customerId: 2, amount: 120, date: '2024-03-03' }
];
```

OUTPUT


averageTransactionAmount: 124
totalAmount: 620
totalTransactions: 5
transactionsByCustomer:
1: (2) [{…}, {…}]
2: (2) [{…}, {…}]
3: [{…}]
[[Prototype]]: Object
transactionsPerDay: {2024-03-01: Array(2), 2024-03-02: Array(2), 2024-03-03: Array(1)}