2804. Array Prototype ForEach

Easy

Leetcode Link

Problem Description

The problem requires the creation of a custom `forEach` method that can be applied to all arrays. In JavaScript, the `forEach` method is a built-in function that executes a provided function once for each array element. However, this custom method should be written without using the native `.forEach` or other built-in array methods.

The custom `forEach` method needs to accept two parameters:

- `callback`: a function that you want to execute for each element in the array.
- `context`: an object that can be referred to as `this` within the scope of the `callback` function.

The `callback` function itself should have access to three arguments:

1. `currentValue`: the value of the array element in the current iteration.
2. `index`: the index of the current array element.
3. `array`: the entire array that the `forEach` is being executed upon.

The essence of this problem is to enhance the array prototype inside JavaScript, ensuring that this functionality is available on all arrays within that context.

Intuition

The intuition is to replicate the behavior of the built-in `forEach` functionality manually. We want to achieve three key tasks:

1. Iterate through each element of the array – we can do this via a for loop that starts at the beginning of the array (index 0) and continues to the end (length of the array).
2. During each iteration, execute the `callback` function with the proper arguments – we can call the `callback` function with `currentValue`, `index`, and `array`.
3. Respect the `context` in which `callback` is executed if provided – we can use the `call` method on the `callback` function to set the `this` value explicitly to the `context` passed as a parameter.

The solution doesn't return anything because the purpose of `forEach` is to execute side effects rather than compute and return a value. The `forEach` method is more about doing something with each array element, like modifying the array or using the elements for some other side effectful operations. This implementation fits the requirement which is to execute the provided function for each of the array's elements with the given context.

Solution Approach

The implementation of the custom `forEach` method makes use of JavaScript's prototypal inheritance. By appending a new function to `Array.prototype`, we ensure that all arrays inherit this method.

Here is the step-by-step breakdown of the approach:

1. We start by adding a new function to `Array.prototype` named `forEach`. This means that every array created in this JavaScript environment will now have access to this custom `forEach` method.

2. Our `forEach` function takes in two parameters: `callback` and `context`. The `callback` is a function that we want to call with each element of the array. The `context` is optional and is used to specify the value of `this` inside the callback function when it is called.

3. We make use of a simple for loop to iterate over the array. The initial index is set to 0, and we loop through until we reach the end of the array, denoted by `this.length`, because within this function, `this` refers to the array upon which the `forEach` method was called.

4. For each iteration, we use the `call` method of the `callback` function to execute it. The `call` method is a built-in JavaScript method that allows us to call a function with an explicitly set `this` value – in this case, the `context` parameter. If `context` is `undefined`, the value of `this` inside the callback will default to the global object, which is the default behavior in a browser, or to `undefined` in strict mode.

5. The `callback` function is called with three arguments: the current element value (`this[i]`), the current index (`i`), and the array itself (`this`). This matches the standard `forEach` method's signature.

The code does not include any complex algorithms, data structures, or patterns—it is a straightforward iteration using a for loop and function calls, sticking closely to the requirements of how the built-in `forEach` is expected to function.

The beauty of this approach is in its simplicity and direct manipulation of the `Array.prototype` to achieve the desired effect across all arrays within the scope of execution.

Example Walkthrough

Let's consider an example where we have an array of names and we want to print each name to the console with a greeting. The array is `['Alice', 'Bob', 'Charlie']`.

Here's a breakdown of how we could use our custom `forEach` method to accomplish this:

1. First, we'd set up our custom `forEach` method by extending the `Array.prototype` as described in the solution approach. This method would then be available on every array.

2. We would define a `callback` function that takes `currentValue` (the name in this context), `index`, and `array` as arguments. Our callback function would simply print the greeting to the console using the current name:

```
function greeting(name, index) {
  console.log(`Hello, ${name}! You are at position ${index + 1}.`);
}
```

3. We would then call our custom `forEach` method on our array of names, passing the `greeting` function as the `callback`.

```
const names = ['Alice', 'Bob', 'Charlie'];
names.forEach(greeting);
```

4. When we execute this code, our custom `forEach` method iterates over the array. For each element:

   a. It calls the `greeting` function using `call`, which applies the given `context` (if any). In this case, we haven't provided a context, so it defaults to the global context.

   b. This results in our `greeting` function being called with each name and its corresponding index. The `greeting` function then logs the greeting to the console.

5. As the loop progresses, `greeting` would be called with 'Alice' at index 0, 'Bob' at index 1, and 'Charlie' at index 2.

6. The final output to the console would be:

```
Hello, Alice! You are at position 1.
Hello, Bob! You are at position 2.
Hello, Charlie! You are at position 3.
```

This small example illustrates how the custom `forEach` method behaves similarly to the built-in `forEach`, allowing us to execute a function for each element in an array without relying on the native method.

Python Solution

```
class Array(list):
    # The custom forEach function added to the Array class (inherits from list)
    def forEach(self, callback, context=None):
        # Iterate over each element in the list
        for index, value in enumerate(self):
            # If context is provided, apply it to the callback
            if context:
                callback(value, index, self, context)
            else:
                # Call the callback with the current element, its index, and the list itself
                callback(value, index, self)

# Example usage of the custom forEach function

# Create an instance of Array with numbers
arr = Array([1, 2, 3])

# Define a callback function that doubles the value of each list element
def double_values_callback(value, index, array, context=None):
    # Update the list element to its doubled value
    array[index] = value * 2

# Define an optional context object (unused in this example)
context = {"context": True}

# Apply the custom forEach function to the arr Array using the doubleValuesCallback and context
arr.forEach(double_values_callback, context)

# Print the updated list to the console, which should show doubled values
print(arr)  # Output will be: [2, 4, 6]
```

Java Solution

```
import java.util.function.BiConsumer;

// Custom interface extending the functionality of BiConsumer interface
interface ForEach<T> extends BiConsumer<T, Integer> {
}

/**
 * A utility class to work with arrays
 */
class ArrayUtils {

    /**
     * A custom implementation of the forEach function that operates on arrays.
     *
     * @param arr      The array on which the operation is performed.
     * @param callback The callback function to execute for each element.
     * @param <T>      The type of the elements in the array.
     */
    public static <T> void forEach(T[] arr, ForEach<T> callback) {
        // Iterate over each element in the array
        for (int i = 0; i < arr.length; i++) {
            // Execute the callback function with the current element and its index
            callback.accept(arr[i], i);
        }
    }
}

/**
 * Example usage of the custom forEach function.
 */
public class Main {
    public static void main(String[] args) {
        // Create an array of numbers
        Integer[] arr = {1, 2, 3};

        // Define a callback function that doubles the value of each array element
        ForEach<Integer> doubleValuesCallback = new ForEach<>() {
            public void accept(Integer value, Integer index) {
                arr[index] = value * 2;
            }
        };

        // Apply the custom forEach function to the arr array using the doubleValuesCallback
        ArrayUtils.forEach(arr, doubleValuesCallback);

        // Print the updated array to the console, which should show doubled values
        for (Integer value : arr) {
            System.out.println(value);  // Output will be: 2, 4, 6
        }
    }
}
```

C++ Solution

```
#include <iostream>
#include <vector>
#include <functional>

// Declare a template for a custom forEach function
template <typename T>
void forEach(std::vector<T>& vec, std::function<void(T&, size_t, std::vector<T>&)> callback, void* context = nullptr) {
    // Iterate over each element in the vector
    for (size_t i = 0; i < vec.size(); ++i){
        // Call the callback function, passing the current element by reference, its index, and the vector itself
        callback(vec[i], i, vec);
    }
}

int main() {
    // Example usage of the custom forEach function

    // Create a vector of numbers
    std::vector<int> arr = {1, 2, 3};

    // Define a callback function that doubles the value of each element of the vector
    auto doubleValuesCallback = [](int& value, size_t index, std::vector<int>& array){
        // Double the passed element's value
        value *= 2;
    };

    // Define an optional context object (not utilized in this example)
    // In this C++ version, it's not used but provided for compatibility with the original interface
    void* context = nullptr;

    // Apply the custom forEach function to the arr vector using the doubleValuesCallback
    forEach(arr, doubleValuesCallback, context);

    // Print the updated vector to the console, which should show doubled values
    for (const int& value : arr) {
        std::cout << value << " ";  // Output will be: 2 4 6
    }

    return 0;
}
```

Typescript Solution

```
// Extending the Array prototype interface to include the custom forEach function
interface Array<T> {
  forEach(callback: (value: T, index: number, array: T[]) => void, context?: any): void;
}

// Custom implementation of the forEach function that adheres to TypeScript's syntax and type safety
Array.prototype.forEach = function<T>(this: T[], callback: (value: T, index: number, array: T[]) => void, context?: any): void {
  // Iterate over each element in the array
  for (let i = 0; i < this.length; i++) {
    // Call the callback function with the specified context, passing the current element, its index, and the array itself
    callback.call(context, this[i], i, this);
  }
};

// Example usage of the custom forEach function

// Create an array of numbers
const arr: number[] = [1, 2, 3];

// Define a callback function that doubles the value of each array element
const doubleValuesCallback = (value: number, index: number, array: number[]): void => {
  array[index] = value * 2;
};

// Apply the custom forEach function to the arr array using the doubleValuesCallback and context
arr.forEach(doubleValuesCallback, context);

// Print the updated array to the console, which should show doubled values
console.log(arr); // Output will be: [2, 4, 6]
```

Time and Space Complexity

The time complexity of the provided custom `forEach` function is $O(n)$, where n is the number of elements in the array upon which `forEach` is called. This is due to the for loop iterating through each element of the array exactly once.

The space complexity of `forEach` is $O(1)$ (constant space complexity), assuming that the callback function's space complexity is also constant. The space used by the `forEach` method itself does not grow with the size of the input array because it uses a fixed amount of additional space (a single loop counter `i`).