2794. Create Object from Two Arrays

Easy

Leetcode Link

Problem Description

The task is to create an object based on two provided arrays: `keysArr` and `valuesArr`. The `keysArr` contains the potential keys for the object, and `valuesArr` contains the corresponding values. As you iterate through both arrays, you are supposed to use the elements at the same index in each array to construct a key-value pair in the new object `obj`. However, there are a couple of rules to follow:

1. If there are duplicate keys, you should only keep the first occurrence of the key-value pair—any subsequent duplicate key should not be included in the object.
2. If a key in `keysArr` is not a string, you need to convert it into a string. This can be done using the `String()` function.

The problem tests your understanding of objects in TypeScript, as well as array manipulation and the consideration of edge cases such as duplicates and type conversion.

Intuition

To approach this problem, we need to think methodically about the process of building an object from two arrays, ensuring we adhere to the stated rules. Here's an intuitive step-by-step approach:

1. Initialize an empty object to store our key-value pairs.
2. Iterate over the `keysArr` array. For each key, we should:
   - Convert the key to a string, which allows any type of values in `keysArr` to be used as valid object keys.
   - Check if the key is already present in our object. Since object keys in JavaScript are unique, if the key already exists, it means we have a duplicate and should not add it again.
   - If the key does not exist in our object, add the key-value pair to the object using the key from `keysArr` and value from `valuesArr`.
   - It is important to note that we don't need to check for the same index in `valuesArr` as the assumption is that both arrays have a one-to-one mapping.

This solution is straightforward and efficient since it performs a single traversal over the input arrays and maintains a constant lookup time for keys in the object by benefiting from the properties of the object data structure in JavaScript (or TypeScript).

Solution Approach

The solution approach for creating an object from two arrays involves the following steps, demonstrating the use of algorithms, data structures, and patterns:

1. **Initialization**: We begin by initializing an empty object `ans` of type `Record<string, any>`. In TypeScript, `Record` is a utility type that constructs an object type with a set of known property keys (in this case, `string`) and corresponding value types (`any` in this case, which represents any value type).

   ```
   const ans: Record<string, any> = {};
   ```

2. **Iteration**: The next step is to loop through the `keysArr` using a standard `for` loop. This loop will iterate over all elements of the `keysArr` array and by extension, through the `valuesArr` since they are matched by index.

   ```
   for (let i = 0; i < keysArr.length; ++i)
   ```

3. **String Conversion**: Inside the loop, each key is converted to a string to ensure consistency of the object keys. This conversion is done using the `String` function.

   ```
   const k = String(keysArr[i]);
   ```

4. **Key Uniqueness Check**: We must ensure that each key included in the object is unique. If the key `k` is `undefined` in `ans`, it means the key is not yet added to the object, and it's safe to insert the key-value pair into it.

   ```
   if (ans[k] === undefined)
   ```

5. **Creating Key-Value Pair**: Once we confirm the key is unique, we assign the value from `valuesArr` at the same index to the key we've just processed.

   ```
   ans[k] = valuesArr[i];
   ```

6. **Returning the Result**: After the loop has processed all key-value pairs, the fully constructed object `ans` is returned.

   ```
   return ans;
   ```

This solution uses a `for` loop for iteration, object data structure for key-value pairing, and string conversion. Altogether, this results in a time complexity of O(n), where n is the number of elements in `keysArr`. The space complexity is also O(n) to store the resulting key-value pairs in the object. The use of JavaScript object properties ensures that no duplicate keys are present in the final output.

Example Walkthrough

Let's say we are given the following two arrays:

```
const keysArr = ['foo', 'bar', 'foo', 123];
const valuesArr = [1, 2, 3, 4];
```

The goal is to create an object that maps each string key from `keysArr` to the corresponding value in `valuesArr`.

Following are the steps from the solution approach applied to this example:

1. **Initialization**: We start by creating an empty object:

   ```
   const ans: Record<string, any> = {};
   ```

   `ans` is now an empty object `{}`.

2. **Iteration**: We iterate through each element of `keysArr`:

   ```
   for (let i = 0; i < keysArr.length; ++i)
   ```

   This loops from `i = 0` to `i = 3` since there are 4 elements in `keysArr`.

3. **String Conversion**: Convert each key to a string:

   ```
   const k = String(keysArr[i]);
   ```

   This will convert the keys to `'foo'`, `'bar'`, `'foo'`, `'123'`.

4. **Key Uniqueness Check**: We check if the key already exists in the `ans` object. For the first-time key `'foo'`, `'bar'`, and `'123'`, the condition will be `true`, but it's false for the second occurrence of `'foo'`.

   ```
   if (ans[k] === undefined)
   ```

   For the first time, we encounter `'foo'`, `'bar'`, and `'123'`, they don't exist in `ans`, so we proceed to add them.

5. **Creating Key-Value Pair**: Add the key-value pair to the `ans` object if the key is first-time seen:

   ```
   ans[k] = valuesArr[i];
   ```

   After processing each key-value pair, `ans` will look like:

   ```
   { 'foo': 1, 'bar': 2, '123': 4 }
   ```

   Note that the second `'foo'` was not added because it was already present.

6. **Returning the Result**: After looping through the arrays, we return the `ans` object:

   ```
   return ans;
   ```

At the end of execution, the resulting object will be:

```
{
  'foo': 1,
  'bar': 2,
  '123': 4
}
```

This matches our rules where we avoid duplicates and convert non-string keys to strings. The returned object now correctly maps each key of `keysArr` to its corresponding value from `valuesArr` following the mentioned transformation and restrictions.

Python Solution

```
def create_object(keys_arr, values_arr):
    # Initialize an empty dictionary to store the key-value pairs
    record = {}

    # Iterate over all elements in the keys array
    for i in range(len(keys_arr)):
        # Convert the key at index i to a string
        key = str(keys_arr[i])

        # If the key doesn't exist in the record, add it with its matching value
        if key not in record:
            # Use the corresponding index in the values array for the value
            record[key] = values_arr[i]
        # If the key already exists, the loop continues to the next iteration without modification

    # Return the constructed dictionary
    return record
```

Java Solution

```
import java.util.HashMap;
import java.util.Map;

public class ObjectCreator {

    /**
     * This method takes two arrays, one for keys and one for values, and creates a map
     * with each key mapped to its corresponding value. If there are duplicate keys, only
     * the first occurrence is considered.
     *
     * @param keysArr  the array containing keys
     * @param valuesArr the array containing values
     * @return a map constructed with key-value pairs from the given arrays
     */
    public Map<String, Object> createObject(Object[] keysArr, Object[] valuesArr) {
        // Initialize an empty HashMap to store the key-value pairs
        Map<String, Object> record = new HashMap<>();

        // Iterate over all elements in the keys array
        for (int i = 0; i < keysArr.length; ++i) {
            // Convert the key at index i to a string
            String key = keysArr[i].toString();

            // If the key doesn't exist in the record, add it with its matching value
            if (!record.containsKey(key)) {
                // Check if the key has a corresponding value before adding it
                Object value = i < valuesArr.length ? valuesArr[i] : null;

                record.put(key, value);
            }
            // If the key already exists, it is ignored due to the check above
        }

        // Return the constructed map
        return record;
    }
}
```

C++ Solution

```
#include <unordered_map>
#include <string>
#include <vector>

// This function takes two vectors: one for keys (keys) and one for values (values)
// It creates an unordered_map (objectMap) with each key mapped to its corresponding value.
// If there are duplicate keys, only the first occurrence is considered.
std::unordered_map<std::string, std::string> CreateObject(const std::vector<std::string>& keys, const std::vector<std::string>& values) {
    // Initialize an empty unordered_map to store the key-value pairs
    std::unordered_map<std::string, std::string> objectMap;

    // Iterate over all elements in the keys vector
    for (size_t i = 0; i < keys.size(); ++i) {
        // Convert the key at index 'i' to a string (it's already a string, but kept for consistency)
        std::string key = keys[i];

        // If the key doesn't exist in the objectMap, add it with its matching value
        if (objectMap.find(key) == objectMap.end()) {
            // Check if we have a corresponding value for the key, if not set an empty string
            std::string value = (i < values.size()) ? values[i] : std::string{};
            objectMap[key] = value;
        }
        // If the key already exists, it is ignored due to the check above
    }

    // Return the constructed objectMap
    return objectMap;
}
```

Typescript Solution

```
// This function takes two arrays: one for keys (keysArr) and one for values (valuesArr)
// It creates an object (record) with each key mapped to its corresponding value.
// If there are duplicate keys, only the first occurrence is considered.

function createObject(keysArr: any[], valuesArr: any[]): Record<string, any> {
    // Initialize an empty object to store the key-value pairs
    const record: Record<string, any> = {};

    // Iterate over all elements in the keys array
    for (let i = 0; i < keysArr.length; ++i) {
        // Convert the key at index i to a string
        const key = String(keysArr[i]);

        // If the key doesn't exist in the record, add it with its matching value
        if (record[key] === undefined) {
            record[key] = valuesArr[i];
        }
        // If the key already exists, it is ignored due to the check above
    }

    // Return the constructed record
    return record;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given function is primarily determined by the for loop that iterates through the `keysArr` array. For each iteration, it performs a constant time operation of converting the key to a string and checking/assigning it in the `ans` object.

The main operations within the loop are:

- Converting the key to a string: `O(1)`
- Checking if the key exists in the object: `O(1)` on average
- Assigning the value to the object: `O(1)`

Considering the loop runs `n` times, where `n` is the length of `keysArr`, the overall time complexity is `O(n)`.

Space Complexity

The space complexity is determined by the space required to store the `ans` object, which has as many properties as there are unique keys provided in the `keysArr`.

- `ans` object storage: up to `O(n)`

The total space complexity of the function is `O(n)`, where `n` is the length of `keysArr`.

Note: In the worst-case scenario, if the keys are all unique, the space complexity will indeed be `O(n)`. However, if keys are repeated, the number of actual unique keys stored could be less than `n`, providing a best-case space complexity of `O(u)`, where `u` is the number of unique keys.