# 2796. Repeat String

Easy

Leetcode Link

## Problem Description

The problem requires us to enhance the `String` prototype in TypeScript by adding a new method called `replicate`. This new method should take an integer `x` and return a new string where the original string is repeated `x` number of times. A key point in the description is that we should implement this functionality without using the built-in `string.repeat` method provided in JavaScript/TypeScript.

## Intuition

To arrive at a solution for replicating a string `x` times without using the `string.repeat` method, we can utilize an array as an intermediary. The intuition is based on two steps:

1. Create an array with `x` number of elements, with each element being the original string that we want to replicate. This is done using the `new Array(times)` constructor which creates an array of the specified length, and `fill(this)` which fills it with the current string context (`this` refers to the string instance where the method is called).
2. Join all the array elements into a single string. The `join('')` method is used to concatenate all the elements of the array into a single string, with the empty string `''` serving as the separator (which means there will be no characters between the repetitions of the original string).

When we put these two steps together in the `replicate` function, we get a new string consisting of the original string repeated `x` times without using the `string.repeat` method.

## Solution Approach

The implementation of the `replicate` method in TypeScript involves extending the `String` prototype, which is a standard JavaScript object prototype pattern that allows us to add new properties or methods to all instances of the `String` class.

Here are the steps taken in the implementation:

1. **Extending the `String` Interface:** To add a new method to the `String` prototype in TypeScript, we first need to extend the `String` interface with our new method signature. This is done through declaration merging by declaring an expanded interface named `String` globally which includes our custom method `replicate(times: number): string;`.

```
declare global {
    interface String {
        replicate(times: number): string;
    }
}
```

   This tells TypeScript that every `String` object will now have a method called `replicate` that takes a number and returns a string.

2. **Implementing the `replicate` Method:** The `replicate` method is then added to the `String` prototype.

```
String.prototype.replicate = function (times: number) {
    return new Array(times).fill(this).join('');
};
```

   - **Algorithm:** The algorithm employed here is straightforward and involves the creation of an array followed by joining its elements. No complex patterns or data structures are required.
   - **Creating an Array of Repetitions:** We use the `new Array(times)` constructor to create an array of length equal to the number of times the string needs to be replicated. Then, the `fill` method fills up the array with the original string (`this`) in all positions. This effectively creates an array where each element is the string to be replicated.
   - **Joining the Array:** Once the array is filled, we use the `join` method with an empty string as a separator to concatenate all the elements of the array, effectively repeating the original string `times` number of times.

This solution is efficient because the heavy lifting is done by native array operations, which are highly optimized in JavaScript engines. Since we are avoiding recursion and repeated string concatenation, we minimize the risk of performance issues and potential call stack overflows for large values of `times`. It is a simple and elegant way to add custom repeat functionality to strings in TypeScript.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach for enhancing the `String` prototype with a `replicate` method in TypeScript.

Suppose we have the string `"hello"` and we want to replicate it 3 times. To achieve this using our custom `replicate` method, we would start by calling `"hello".replicate(3)`.

Here's what happens step by step when we call the `replicate` method:

1. An array with 3 elements is created using `new Array(3)`.
2. This array is then filled with the string `"hello"` in all positions, making the array look like this: `["hello", "hello", "hello"]`.
3. Finally, we join all the array elements with no separator using `join('')`, which results in the string `"hellohellohello"`.

So, if we execute the following TypeScript code:

```
"hello".replicate(3)
```

The output we expect to see would be:

```
"hellohellohello"
```

With this example, we have shown how the `replicate` method uses the concept of arrays to create multiple copies of a string and concatenate them to produce a final string with the original string repeated the specified number of times without using the built-in `string.repeat` method.

## Python Solution

```
# Extends the built-in string class to add a new method called 'replicate'
class ExtendedString(str):
    # The 'replicate' method takes a number and returns a string where the original
    # string is repeated that many times.
    def replicate(self, times):
        # Return the current string instance multiplied by 'times', which effectively
        # repeats the string 'times' times.
        return self * times

# To ensure the 'replicate' method is available to all string instances,
# extend the global 'str' type by subclassing ExtendedString.
# This allows the usage of 'replicate' on any string object created after this point.
str = ExtendedString

# Example of using the newly added 'replicate' method
example_string = str("Hello ")
replicated_string = example_string.replicate(3)
print(replicated_string)  # Output: Hello Hello Hello
```

Please note, in Python, monkey patching built-in types (like adding a method to the built-in `str` type) is not a common or recommended practice because it can lead to unexpected behaviors, especially in larger projects or with third-party libraries. The example above demonstrates how to add the method by subclassing, but whenever you create a string, you would now have to explicitly use `str()` to ensure it has the `replicate` method, which is not the case in default strings directly created with quotes.

For a simpler, more Pythonic solution that doesn't attempt to modify the built-in `str` type:

```
# Define a function called 'replicate' that replicates a string.
def replicate(string, times):
    # Return the string repeated 'times' times.
    return string * times

# Example usage of the 'replicate' function
example_string = "Hello "
replicated_string = replicate(example_string, 3)
print(replicated_string)  # Output: Hello Hello Hello
```

## Java Solution

```
public final class StringUtils {

    /**
     * Replicates a given String a specified number of times.
     *
     * @param input The string to be replicated.
     * @param times The number of times to replicate the string.
     * @return A new String that is a replication of the input String 'times' times.
     */
    public static String replicate(String input, int times) {
        // Use a StringBuilder to efficiently replicate the string.
        StringBuilder builder = new StringBuilder();

        // Loop the number of times required, appending the input each time.
        for (int i = 0; i < times; i++) {
            builder.append(input);
        }

        // Return the result as a string.
        return builder.toString();
    }

    // Private constructor to prevent instantiation.
    // Utility classes should not be instantiated.
    private StringUtils() {
    }

    // main method for example usage
    public static void main(String[] args) {
        // Example usage of the replicate method
        String originalString = "abc";
        String replicatedString = StringUtils.replicate(originalString, 3);
        System.out.println(replicatedString); // Expected output: "abcabcabc"
    }
}
```

## C++ Solution

```
#include <string> // Include the standard string library
#include <iostream> // Include the IO stream library for demonstration purposes

// Define the 'replicate' function that takes a string and an int
// It returns a new string where the original string is repeated 'times' times
std::string replicate(const std::string& str, int times) {
    std::string result; // Initialize the result string
    // Reserve enough space in result to avoid repeatedly reallocating
    result.reserve(str.size() * times);
    for(int i = 0; i < times; ++i) {
        result += str; // Append the original string 'times' times
    }
    return result; // Return the result string
}

// Demonstration of the 'replicate' function
int main() {
    std::string original = "Hello";
    int times = 3;
    std::string replicated = replicate(original, times); // Use the 'replicate' function
    std::cout << replicated; // Outputs: HelloHelloHello
    return 0; // Return successful exit code
}
```

## Typescript Solution

```
// Extend the global String interface to add a new method called 'replicate'
declare global {
    // The 'replicate' method is expected to be present on all string instances
    // It takes a number and returns a string where the original string is repeated that many times
    interface String {
        replicate(times: number): string;
    }
}

// Define the replicate function on the String prototype, making it available to all strings
String.prototype.replicate = function (times: number): string {
    // Initialize an array of 'times' length, fill it with the current string instance (this),
    // and then join all elements into a single string where the original string is replicated
    return new Array(times + 1).join(this);
};

// Ensure the extended String interface is globally available
export {};
```

## Time and Space Complexity

### Time Complexity

The time complexity of the `replicate` method can be considered as $O(n * m)$, where $n$ is the number of times the string is replicated, and $m$ is the length of the source string.

This is because the `.fill` method instantiates an array of size $n$ and fills it with the source string. Then, the `.join` method iterates over this array to concatenate the strings together. If the array `.fill` method is implemented in a way that performs a deep copy, the time to fill the array is $O(n)$, but as it copies references in this case, we can consider this part $O(n)$ for the references. However, the `.join` operation requires concatenating the string $n$ times, with each concatenation operation being proportional to the length of the string, which is $m$. Therefore, the `.join` operation dominates and results in a time complexity of $O(n * m)$.

### Space Complexity

The space complexity of the `replicate` function is $O(n * m)$ as well. This is because it creates an array of $n$ elements, each of which is a reference to the source string of length $m$. When concatenated, the resulting string occupies space proportional to $n * m$.

One could argue that since `.fill` is using the same reference, the array does not require additional space proportional to the length of the string. However, the resulting string from `.join` still requires $n * m$ space. Therefore, the overall space complexity remains $O(n * m)$.