2667. Create Hello World Function

Easy

Leetcode Link

Problem Description

The task is to implement a function named `createHelloWorld`. The function doesn't take any parameters and should return a new function. The returned function, when called, should always return the string "Hello World", regardless of any arguments it might be passed.

Intuition

The solution approach for this problem is quite straightforward. We need to define a function that when invoked, returns another function. The inner function is the one that should be designed to return the string "Hello World". In JavaScript and TypeScript, which is a superset of JavaScript, functions are first-class citizens, meaning they can be returned by other functions.

To solve this problem, we create the outer function `createHelloWorld` without parameters since none are specified in the requirements. Inside this function, we define another function that takes a variable number of arguments, indicated by the `...args` syntax. The `args` notation is called the rest parameter syntax and is used to handle an indefinite number of arguments. However, the inner function does not need to use these arguments at all, it simply returns the fixed string "Hello World". The outer function then returns the inner function.

This approach makes use of closure, where the inner function retains access to the outer function's scope even after the outer function has finished execution. In this case, the scope doesn't contain any variables, but it's a principle that could be useful if the problem were extended to include such requirements.

Solution Approach

The solution to this problem involves the concept of higher-order functions and closures in TypeScript:

- **Higher-order functions**: In TypeScript, functions can be treated as first-class citizens, meaning they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables. In this scenario, we're creating a higher-order function `createHelloWorld` that returns another function.
- **Closures**: A closure is a function that remembers the environment in which it was created. This inner function, even though it does not use any environment variables, still demonstrates the concept of closures. It has the potential to access and use variables from its outer scope if needed.
- **Function definition**: The `createHelloWorld` function is defined without parameters, which aligns with the requirements.
- **Inner function**: Inside `createHelloWorld`, we define an anonymous inner function. This inner function uses the rest parameter syntax `...args` to accept any number of arguments. It ignores these arguments and simply returns the string `"Hello World"`.
- **Returning the inner function**: Finally, we return the inner anonymous function from `createHelloWorld`. When the `createHelloWorld` function is called, it will return this inner function, which can then be invoked separately to always return `"Hello World"`.

To reiterate, the key characteristics of the solution are the use of higher-order functions and the concept of closures, even though the closure's capability to access the outer environment is not directly utilized here. This pattern allows us to create a factory function `createHelloWorld` that produces a simple function with a consistent behavior.

Here's a step-by-step breakdown of the inner function's behavior:

1. The anonymous inner function is declared within `createHelloWorld`, which takes `...args` but doesn't use them.
2. This inner function is set up to always return the constant string `"Hello World"`, regardless of the input arguments.
3. The aforementioned inner function, with its set behavior, is returned by the `createHelloWorld` function.

This completes our solution, here is the code snippet for a better understanding:

```
function createHelloWorld() {
    // This is the higher-order function
    return function (...args): string {
        // This inner function is a closure, it always returns "Hello World"
        return 'Hello World';
    };
}

// Usage example:
// const f = createHelloWorld();
// f(); // "Hello World"
```

Each time `createHelloWorld` is called, a new instance of the inner function is created and returned, all of which will uniformly return the string `Hello World` when executed.

Example Walkthrough

To illustrate the solution approach, let's consider an example of how the `createHelloWorld` function would be used in practice:

1. The outer function `createHelloWorld` is defined in our TypeScript code. This function doesn't take any arguments and will return an inner function.
2. We then call `createHelloWorld()` from somewhere in our codebase to get the inner function. Each call to `createHelloWorld` will create a new instance of this inner function.
3. We store the returned inner function in a variable for use. For example:

```
const helloFunction = createHelloWorld();
```

4. With our variable `helloFunction` holding the inner function, we can call it to get the string `"Hello World"`.
5. When we invoke `helloFunction()` even with arguments, like `helloFunction(1, 'a', true)`, it will still ignore those inputs and simply return `"Hello World"`.

Here's how the code looks in action:

```
// Step 1:
// Declare the createHelloWorld function
function createHelloWorld() {
    // Step 2:
    // This is the inner function that the createHelloWorld function will return
    return function (...args): string {
        // Step 3:
        // No matter what arguments this inner function receives, it always returns "Hello World"
        return 'Hello World';
    };
}

// Usage example:
// Invoke `createHelloWorld` to obtain the inner function and store it in `f`
const helloFunction = createHelloWorld();

// Call the inner function `helloFunction` with no arguments
console.log(helloFunction()); // Outputs: "Hello World"

// Call the inner function `helloFunction` with some arbitrary arguments
console.log(helloFunction(1, 'a', true)); // Outputs: "Hello World", ignoring the arguments
```

In the above example, we see that no matter how `helloFunction` is called, the output remains consistent, returning the string `"Hello World"`. This demonstrates the concept of higher-order functions and closures within TypeScript, where the `createHelloWorld` function serves as a factory for creating new functions that adhere to a specific behavior.

Python Solution

```
# This function `create_hello_world` creates and returns a new function.
# The returned function takes any number of arguments (none of which are used) and returns a string.
def create_hello_world():
    # Here we define and return the inner function.
    # It ignores any incoming arguments and when called, it simply returns the string "Hello World".
    def inner_function(*args):
        return 'Hello World'
    return inner_function

# Example usage:
# We create a new function 'hello_world_function' by calling 'create_hello_world'.
hello_world_function = create_hello_world()
# When we call 'hello_world_function', it will return the string "Hello World".
greeting = hello_world_function()  # greeting will be "Hello World"


```

Java Solution

```
import java.util.function.Function;

public class HelloWorldCreator {

    // This method `createHelloWorld` creates and returns a new function.
    public static Function<Object[], String> createHelloWorld() {
        // Here we define and return the inner function.
        // It is a lambda function that ignores any incoming arguments and when called,
        // it simply returns the string "Hello World".
        return (Object[] args) -> "Hello World";
    }

    // Example usage:
    public static void main(String[] args) {
        // We create a new function 'helloWorldFunction' by calling 'createHelloWorld'.
        Function<Object[], String> helloWorldFunction = createHelloWorld();

        // When we call 'helloWorldFunction', it will return the string "Hello World".
        // Since the function ignores its arguments, we provide an empty Object array.
        String greeting = helloWorldFunction.apply(new Object[]{});

        // The variable 'greeting' will contain "Hello World"
        System.out.println(greeting); // This will print "Hello World"
    }
}


```

C++ Solution

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

// This function `createHelloWorld` creates and returns a new function.
// The returned function can accept any number of arguments (none of which are used) and returns a string ("Hello World").
std::function<std::string()> createHelloWorld() {
    // Here we define and return the inner lambda function.
    // It ignores any incoming arguments and when called, it simply returns the string "Hello World".
    return []() -> std::string {
        // Returning the greeting message
        return "Hello World";
    };
}

// Example usage:
int main() {
    // We create a new function 'helloWorldFunction' by calling 'createHelloWorld'.
    auto helloWorldFunction = createHelloWorld();

    // When we call 'helloWorldFunction', it will return the string "Hello World".
    std::string greeting = helloWorldFunction(); // greeting will hold the string "Hello World"

    // Output the greeting to the console.
    std::cout << greeting << std::endl;

    return 0; // Return success
}


```

Typescript Solution

```
// This function `createHelloWorld` creates and returns a new function.
// The returned function takes any number of arguments (none of which are used) and returns a string.
function createHelloWorld(): (...args: any[]) => string {
    // Here we define and return the inner function.
    // It ignores any incoming arguments and when called, it simply returns the string "Hello World".
    return function(...args: any[]): string {
        return 'Hello World';
    };
}

// Example usage:
// We create a new function 'helloWorldFunction' by calling 'createHelloWorld'.
const helloWorldFunction = createHelloWorld();
// When we call 'helloWorldFunction', it will return the string "Hello World".
const greeting = helloWorldFunction(); // greeting will be "Hello World"


```

Time and Space Complexity

Time Complexity

The time complexity of the `createHelloWorld` function is `O(1)`, which means it runs in constant time. This is because it only involves creating and returning a simple function that, when called, returns a hard-coded string `'Hello World'`.

Once the inner function returned by `createHelloWorld` is called, it also executes in `O(1)` time for the same reason: it performs no computations or iterations and simply returns the string `'Hello World'`.

Space Complexity

The space complexity of `createHelloWorld` is also `O(1)`. It does not utilize any additional space that scales with input size since it simply returns a function. There are no dynamically-allocated data structures or variables that depend on input parameters.

When the returned function is called, the space complexity remains `O(1)` because it only provides a hard-coded string, with no additional space used regardless of the arguments (`...args`) passed to it. The `...args` parameter accepts an arbitrary number of arguments, but since it is not used within the function, it does not affect space complexity.