2704. To Be Or Not To Be

Easy

Leetcode Link

Problem Description

The problem requires us to write a function called `expect`. This function is designed to help developers perform unit tests on their code by checking the equality or inequality of values. The `expect` function takes in a value (referred to as `val`) and returns an object with two methods: `toBe` and `notToBe`.

- The method `toBe(val)` takes another value and checks if this value is strictly equal to the initially provided value (`val`) in the `expect` function, using the strict equality operator `===`. If the two values are strictly equal, it returns `true`. If not, it throws an error with the message "Not Equal".
- The method `notToBe(val)` takes another value and checks if this value is strictly not equal to the initially provided value (`val`) in the `expect` function, using the strict inequality operator `!==`. If the two values are not strictly equal, it returns `true`. If they are equal, it throws an error with the message "Equal".

This functionality is inspired by testing libraries in which assertions are made to validate the expected outcome of code execution against a specific value.

Intuition

The intuition behind the provided solution is to create a simple testing utility that allows developers to verify the outcome of certain operations. The typical use case for such utility is unit testing, where functions or methods are expected to produce certain results given predefined inputs.

To implement the `expect` function, we:

1. Accept a value `val` that represents the expected result of a test case.
2. Return an object containing two closure functions, `toBe` and `notToBe`. Each function accepts one parameter for comparison with the original `val`.

For `toBe`:

- We compare the passed value (`toBeVal`) with the original `val` using the strict equality operator (`===`).
- If the values match, we return `true`, indicating the test passed.
- If the values do not match, we throw an Error with the message "Not Equal", indicating the test failed.

For `notToBe`:

- We also compare the passed value (`notToBeVal`) with the original `val`, but this time we check for strict inequality (`!==`).
- If the values do not match, we return `true`, meaning the test passed.
- If the values are equal, we throw an Error with the message "Equal", indicating the test failed.

This solution is efficient and straightforward because each method performs a single logical evaluation and is contained within its own function scope, providing a clear and isolated outcome for each assertion.

Solution Approach

The solution approach for the `expect` function implementation leverages the concept of closures in JavaScript (TypeScript in this case). Closures allow a function to remember the environment in which it was created even when it is executed elsewhere. We use closures to keep a reference to the original `val` which is to be compared against in both `toBe` and `notToBe` methods.

Here's the step-by-step breakdown of the algorithm and patterns used:

1. Define the function `expect(val: any)`, which accepts a parameter `val`. This is the value against which other values will be tested. It's a generic function that can accept any JavaScript data type.
2. Within `expect`, we return an object with two methods: `toBe` and `notToBe`. Each method is a closure that retains access to `val`.
3. The `toBe` method:
    - Accepts a parameter (`toBeVal: any`) to compare with the original `val`.
    - Checks if `toBeVal` is strictly equal to `val` using the `===` operator.
    - If they are equal, it returns `true`.
    - If not, it throws an Error with a message "Not Equal".
4. The `notToBe` method:
    - Accepts a parameter (`notToBeVal: any`) to be compared against `val`.
    - Checks if `notToBeVal` is strictly not equal to `val` using the `!==` operator.
    - If they are not equal, it returns `true`.
    - If they are equal, it throws an Error with the message "Equal".

The `toBe` method tests for value and type equality, which is crucial because JavaScript has type coercion. By using `===`, we ensure that no type conversion happens, providing a more reliable test for equality.

The `notToBe` method does the opposite, testing for inequality without type coercion by using `!==`, ensuring a rigorous check that the values are not equal, again without automatic type conversion.

The returned object from `expect` function contains both `toBe` and `notToBe` methods, enabling chained calls such as `expect(value).toBe(otherValue)` in the testing code. The clear separation of methods allows for easy extension if more comparison methods are needed in the future.

The use of closures and simple boolean checks makes the implementation straightforward and efficient, avoiding the need for more complex data structures or algorithms.

Example Walkthrough

Let's apply the `expect` function to a simple scenario to illustrate how it works. Imagine you have a function `add(a, b)` that returns the sum of two numbers, and you want to test if your `add` function is correctly adding numbers.

Here's a sample `add` function:

```
function add(a, b) {
    return a + b;
}
```

Now, you want to test that `add(1, 2)` returns `3`. Here's how you can do it using the `expect` function:

```
// The expected result of add(1, 2) is 3.
const result = add(1, 2);

// We call expect with the result of the add function.
// Then, we chain the toBe method with the value we are expecting - 3 in this case.
expect(result).toBe(3); // This should pass as 1 + 2 does indeed equal 3.

// If the test passes, nothing happens. If the test fails, an Error will be thrown with the message "Not Equal".
```

In this example, since `add(1, 2)` is equal to `3`, calling `expect(result).toBe(3)` will return `true` because `result === 3`. No error is thrown, which in a test environment would mean the test passed.

Alternatively, if you want to test if the `add` function doesn't return a wrong value, you can use the `notToBe` method:

```
// Let's say we want to ensure that add(1, 2) is not returning 4.
expect(result).notToBe(4); // This should pass because result is 3, and 3 !== 4.

// If the test passes, nothing happens. If the test fails, an Error will be thrown with the message "Equal".
```

In this case, since `result` is not equal to `4`, the call to `expect(result).notToBe(4)` will return `true`. No error is thrown, so the test is considered to have passed.

This walkthrough demonstrates how the `expect` function can be used to verify both the presence of an expected value and the absence of an incorrect value with a sleek and straightforward syntax.

Python Solution

```
# Define a class representing two methods that assess equality or inequality.
class EqualityChecker:
    # Initialize the checker with the value to compare.
    def __init__(self, value):
        self.value = value

    # Method to check if the provided value equals the expected value.
    def to_be(self, expected_value):
        # Standardized method name to Python convention
        # If the values are not strictly equal, raise an error with the message 'Not Equal'.
        if self.value != expected_value:
            raise ValueError("Not Equal")
        # If the values are strictly equal, return True.
        return True

    # Method to check if the provided value does not equal the specified value.
    def not_to_be(self, expected_value):
        # Standardized method name to Python convention
        # If the values are strictly equal, raise an error with the message 'Equal'.
        if self.value == expected_value:
            raise ValueError("Equal")
        # If the values are not strictly equal, return True.
        return True

# A global function that takes a value and returns an instance with two methods for equality checking.
def expect(value):
    # Return a new instance of EqualityChecker
    return EqualityChecker(value)

# Usage examples:
# expect(5).to_be(5) # returns True
# expect(5).not_to_be(5) # raises an error with the message "Equal"
```

Java Solution

```
// Interface representing two methods that assess equality or inequality.
interface EqualityChecker {
    // Method to check if the provided value equals the expected value.
    boolean toBe(Object expectedValue);
    // Method to check if the provided value does not equal the specified value.
    boolean notToBe(Object expectedValue);
}

// A public final class that encapsulates the functionality to perform equality checks.
public final class Expect {

    // Private constructor to prevent instantiation
    private Expect() {}

    // Static method that takes a value and returns an EqualityChecker with two methods for equality checking.
    public static EqualityChecker expect(final Object value) {
        return new EqualityChecker() {
            // The `toBe` method compares the provided value with `expectedValue` for strict equality.
            @Override
            public boolean toBe(Object expectedValue) {
                // If the values are not strictly equal, throw an error with the message 'Not Equal'.
                if (!value.equals(expectedValue)) {
                    throw new AssertionError("Not Equal");
                }
                // If the values are strictly equal, return true.
                return true;
            }

            // The `notToBe` method checks that the provided value is not strictly equal to `expectedValue`.
            @Override
            public boolean notToBe(Object expectedValue) {
                // If the values are strictly equal, throw an error with the message 'Equal'.
                if (value.equals(expectedValue)) {
                    throw new AssertionError("Equal");
                }
                // If the values are not strictly equal, return true.
                return true;
            }
        };
    }
}

// Usage examples:
// Expect.expect(5).toBe(5); // returns true
// Expect.expect(5).notToBe(5); // throws an error with the message "Equal"
```

C++ Solution

```
#include <stdexcept> // Required for std::runtime_error
#include <iostream> // Required for std::cout (if needed for demonstration)

// Define a struct representing two methods that assess equality or inequality.
struct EqualityChecker {
    // Stored value for comparison
    const auto& value;

    // Constructor to initialize the struct with a value for comparison
    EqualityChecker(const auto& val) : value(val) {}

    // Method to check if the stored value equals the expected value.
    bool toBe(const auto& expectedValue) const {
        // If the values are not strictly equal, throw an error with the message 'Not Equal'.
        if (value != expectedValue) {
            throw std::runtime_error("Not Equal");
        }
        // If the values are strictly equal, return true.
        return true;
    }

    // Method to check if the stored value does not equal the specified value.
    bool notToBe(const auto& expectedValue) const {
        // If the values are strictly equal, throw an error with the message 'Equal'.
        if (value == expectedValue) {
            throw std::runtime_error("Equal");
        }
        // If the values are not strictly equal, return true.
        return true;
    }
};

// A global function that takes a value and returns an EqualityChecker object
// with two methods for equality checking.
EqualityChecker expect(const auto& value) {
    return EqualityChecker(value);
}

// Usage examples:
// This can be uncommented to test the functionality in a main function.
/*
int main() {
    try {
        // This should return true as the values are equal.
        std::cout << std::boolalpha << expect(5).toBe(5) << std::endl;

        // This should throw an error as the values are equal.
        std::cout << expect(5).notToBe(5) << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
    return 0;
}
*/
```

Typescript Solution

```
// Define a type representing two methods that assess equality or inequality.
type EqualityChecker = {
    // Method to check if the provided value equals the expected value.
    toBe: (expectedValue: any) => boolean;
    // Method to check if the provided value does not equal the specified value.
    notToBe: (expectedValue: any) => boolean;
};

// A global function that takes a value and returns an object with two methods for equality checking.
function expect(value: any): EqualityChecker {
    return {
        // The `toBe` method compares the provided value with `expectedValue` for strict equality.
        toBe: (expectedValue: any) => {
            // If the values are not strictly equal, throw an error with the message 'Not Equal'.
            if (value !== expectedValue) {
                throw new Error('Not Equal');
            }
            // If the values are strictly equal, return true.
            return true;
        },
        // The `notToBe` method checks that the provided value is not strictly equal to `expectedValue`.
        notToBe: (expectedValue: any) => {
            // If the values are strictly equal, throw an error with the message 'Equal'.
            if (value === expectedValue) {
                throw new Error('Equal');
            }
            // If the values are not strictly equal, return true.
            return true;
        },
    };
}

// Usage examples:
// expect(5).toBe(5); // returns true
// expect(5).notToBe(5); // throws an error with the message "Equal"
```

Time and Space Complexity

The functions `toBe` and `notToBe` are simple comparison operations that check for equality and inequality respectively. Their execution time does not depend on the size of the input, but rather they execute in constant time.

Time Complexity

Each of these functions (`toBe` and `notToBe`) within the returned object has a time complexity of $O(1)$ since they perform a single comparison operation regardless of the input size.

Space Complexity

The space complexity of the function `expect` is also $O(1)$. It does not allocate additional space that is dependent on the input size; it simply returns an object containing two methods. These function closures capture the provided value `val`, but this does not create a space complexity that scales with input size.