Leetcode

Leetcode Solution 1:-

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


// Create a generic functional interface `OnceFunction` that extends Function,
// specifying its argument and return type with generics.
@FunctionalInterface
interface OnceFunction<T, R> extends Function<T, R> {
    // Override the apply method from Function to define custom behavior.
    @Override
    R apply(T t);
}
```

```
public class OnceExample {


    /**
     * Creates a function that invokes the given function once, no matter how many times it's called.
     * Subsequent calls to the created function return the result of the first invocation.
     *
     * @param func The function to restrict to a single call.
     * @param <T>  The input type of the function.
     * @param <R>  The return type of the function.
     * @return A new function that is restricted to invoking the given function only once.
     */
    public static <T, R> OnceFunction<T, R> once(Function<T, R> func) {
```

```
        // Create a new instance of `OnceFunction`.
        return new OnceFunction<>() {
            // A flag to keep track if the function has been called.
            private boolean called = false;

            // The result of the first call to remember.
            private R firstResult = null;

            @Override
            public R apply(T t) {
                // Check if the function has not been called yet.
                if (!called) {
                    // If not, invoke the function with the provided arguments and store the result.
                    firstResult = func.apply(t);
```

```
                // Update the state to prevent further
invocations.
                called = true;
                // Return the stored result.
                return firstResult;
            }
            // If the function was already called,
return the stored result.
            return firstResult;
        }
    };
}

// Example usage:
public static void main(String[] args) {
    // Define a function that takes an integer
array and returns the sum of all elements.
```

```
        Function<int[], Integer> sumFn = (int[] numbers) -> {
            int sum = 0;
            for (int n : numbers) {
                sum += n;
            }
            return sum;
        };

        // Create a once-wrappable version of the `sumFn` function.
        OnceFunction<int[], Integer> onceSumFn = OnceExample.once(sumFn);

        // Call `onceSumFn` with an integer array. It should return the sum of the numbers.
        System.out.println(onceSumFn.apply(new int[]{1, 2, 3})); // Expected output: 6
```

```
        // Attempt to call `onceSumFn` again, this time with a different integer array.
        // Since `onceSumFn` has already been called, it should return the result of the first call.
        System.out.println(onceSumFn.apply(new int[]{2, 3, 4})); // Expected output: 6
    }
}
```

Leetcode Solution 2:-

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
        // Hello World!
    }
}
```