**Q1**

**Technical Test:**

Imagine you're working on a banking application that calculates interest on customer savings. The function calculate Interest takes in a principal amount and an interest rate and returns the calculated interest. In TypeScript, how would you write Jest tests to:

 Ensure the function correctly calculates the interest for different principal amounts and interest rates.

 Account for potential floating-point precision issues (e.g., ensuring results like 0.1  0.2 are handled correctly)?

Can you also explain any strategies or tools you’d use in your tests to handle rounding errors or precision challenges when dealing with financial calculations?

Answer-🡪 Certainly! When writing Jest tests for a banking application that calculates interest, it's important to ensure accuracy and handle potential floating-point precision issues carefully. Below is an example of how you might write the calculateInterest function and corresponding Jest tests.

**TypeScript Function**

Here’s a simple implementation of the calculateInterest function:

Typescript code-🡪

function calculateInterest(principal: number, rate: number): number {

return principal \* (rate / 100);

}

**Jest Tests**

Here’s how you could write Jest tests for this function, addressing both the correctness of calculations and potential floating-point precision issues:

typescript

import { calculateInterest } from './yourModule'; // Adjust the import based on your file structure

describe('calculateInterest', () => {

it('should correctly calculate interest for a principal of 1000 at a rate of 5%', () => {

const result = calculateInterest(1000, 5);

expect(result).toBeCloseTo(50, 5); // using toBeCloseTo for floating-point precision

});

it('should correctly calculate interest for a principal of 1500 at a rate of 3.5%', () => {

const result = calculateInterest(1500, 3.5);

expect(result).toBeCloseTo(52.5, 5); // Expected value: 1500 \* (3.5 / 100) = 52.5

});

it('should handle edge cases like 0 principal', () => {

const result = calculateInterest(0, 5);

expect(result).toBe(0);

});

it('should handle edge cases like 0 interest rate', () => {

const result = calculateInterest(1000, 0);

expect(result).toBe(0);

});

it('should handle floating-point precision correctly', () => {

const result = calculateInterest(0.1, 0.2);

expect(result).toBeCloseTo(0.0002, 10); // Expected value: 0.1 \* (0.2 / 100) = 0.0002

});

});

2. You are provided with a simplified Order management system. Your task is to write unit tests to cover key functionalities and ensure the code behaves as expected. Project: Order Management System Order.ts pre-screen question 1 // export class Order { private items: { productId: number, quantity: number }[] = []; private isProcessed: boolean = false; public addItem(productId: number, quantity: number): void { if (this.isProcessed) { throw new Error("Cannot add items to an already process ed order."); } this.items.push({ productId, quantity }); } public removeItem(productId: number): void { if (this.isProcessed) { throw new Error("Cannot remove items from an already pr ocessed order."); } this.items = this.items.filter(item => item.productId !== productId); } public processOrder(): void { if (this.items.length === 0) { throw new Error("Cannot process an order with no item s."); } this.isProcessed = true; } public getItems(): { productId: number, quantity: number } [] { return this.items; } pre-screen question 2 public isOrderProcessed(): boolean { return this.isProcessed; } }

Answer-🡪 Sure! Below are Jest unit tests for the Order class from your Order Management System. These tests cover key functionalities such as adding items, removing items, processing orders, and checking the order status.

**Jest Tests for the Order Class**

First, ensure you have Jest set up in your TypeScript project. You might need to install necessary packages if you haven't already:

**bash**

npm install --save-dev jest ts-jest @types/jest

Here’s how you can structure the tests:

**typescript**

import { Order } from './Order'; // Adjust the import based on your file structure

describe('Order', () => {

let order: Order;

beforeEach(() => {

order = new Order();

});

describe('addItem', () => {

it('should add an item to the order', () => {

order.addItem(1, 2);

expect(order.getItems()).toEqual([{ productId: 1, quantity: 2 }]);

});

it('should throw an error if the order is already processed', () => {

order.processOrder();

expect(() => order.addItem(1, 2)).toThrow("Cannot add items to an already processed order.");

});

});

describe('removeItem', () => {

it('should remove an item from the order', () => {

order.addItem(1, 2);

order.removeItem(1);

expect(order.getItems()).toEqual([]);

});

it('should throw an error if the order is already processed', () => {

order.processOrder();

expect(() => order.removeItem(1)).toThrow("Cannot remove items from an already processed order.");

});

});

describe('processOrder', () => {

it('should process the order if there are items', () => {

order.addItem(1, 2);

order.processOrder();

expect(order.isOrderProcessed()).toBe(true);

});

it('should throw an error if the order has no items', () => {

expect(() => order.processOrder()).toThrow("Cannot process an order with no items.");

});

});

describe('getItems', () => {

it('should return the current items in the order', () => {

order.addItem(1, 2);

order.addItem(2, 3);

expect(order.getItems()).toEqual([

{ productId: 1, quantity: 2 },

{ productId: 2, quantity: 3 }

]);

});

});

describe('isOrderProcessed', () => {

it('should return false if the order is not processed', () => {

expect(order.isOrderProcessed()).toBe(false);

});

it('should return true if the order is processed', () => {

order.addItem(1, 2);

order.processOrder();

expect(order.isOrderProcessed()).toBe(true);

});

});

});

**Explanation of the Tests**

1. **Setup with beforeEach:** A new instance of Order is created before each test to ensure tests are isolated and independent.
2. **Adding Items:**
   * Tests if an item is correctly added.
   * Tests if adding an item after processing throws the appropriate error.
3. **Removing Items:**
   * Tests if an item is correctly removed.
   * Tests if trying to remove an item after processing throws an error.
4. **Processing Orders:**
   * Tests if the order can be processed when it has items.
   * Tests if processing an empty order throws an error.
5. **Getting Items:**
   * Tests if the current items are correctly returned.
6. **Checking Order Status:**
   * Tests the order status before and after processing.

**Running the Tests**

To run your tests, you can use the following command:

**bash**

npx jest

This should give you feedback on the tests, ensuring that your Order class behaves as expected.

3. Write unit tests using Jest to cover the following scenarios: Adding an item to the order. Removing an item from the order. Processing the order. Preventing further modifications after the order is processed. Error handling when trying to process an empty order.  Ensure your tests cover both happy paths (correct functionality) and edge cases (e.g., removing an item that doesnʼt exist, attempting to add/remove items after processing).  Write TypeScript-friendly tests that leverage type safety and ensure your test coverage is high. Deliverables: Your test suite, including tests for all the scenarios outlined above. A brief README.md explaining how to run the tests. Evaluation Criteria: Correctness and completeness of the test cases. Usage of Jest for test automation. Integration of TypeScript features like type safety in your tests. Code readability and best practices. Error handling and edge case coverage.

Here's a comprehensive test suite using Jest for the Order class, covering the specified scenarios. Additionally, I’ll provide a brief README.md to guide you on running the tests.

### Order.test.ts

**typescript**

import { Order } from './Order'; // Adjust the import based on your file structure

describe('Order', () => {

let order: Order;

beforeEach(() => {

order = new Order();

});

describe('addItem', () => {

it('should add an item to the order', () => {

order.addItem(1, 2);

expect(order.getItems()).toEqual([{ productId: 1, quantity: 2 }]);

});

it('should allow adding multiple items', () => {

order.addItem(1, 2);

order.addItem(2, 3);

expect(order.getItems()).toEqual([

{ productId: 1, quantity: 2 },

{ productId: 2, quantity: 3 }

]);

});

it('should throw an error if trying to add an item after processing', () => {

order.processOrder();

expect(() => order.addItem(1, 2)).toThrow("Cannot add items to an already processed order.");

});

});

describe('removeItem', () => {

it('should remove an item from the order', () => {

order.addItem(1, 2);

order.removeItem(1);

expect(order.getItems()).toEqual([]);

});

it('should throw an error if trying to remove an item after processing', () => {

order.addItem(1, 2);

order.processOrder();

expect(() => order.removeItem(1)).toThrow("Cannot remove items from an already processed order.");

});

it('should do nothing if trying to remove an item that doesn\'t exist', () => {

order.addItem(1, 2);

order.removeItem(2); // Item 2 does not exist

expect(order.getItems()).toEqual([{ productId: 1, quantity: 2 }]);

});

});

describe('processOrder', () => {

it('should process the order if there are items', () => {

order.addItem(1, 2);

order.processOrder();

expect(order.isOrderProcessed()).toBe(true);

});

it('should throw an error if trying to process an empty order', () => {

expect(() => order.processOrder()).toThrow("Cannot process an order with no items.");

});

});

describe('isOrderProcessed', () => {

it('should return false if the order is not processed', () => {

expect(order.isOrderProcessed()).toBe(false);

});

it('should return true if the order is processed', () => {

order.addItem(1, 2);

order.processOrder();

expect(order.isOrderProcessed()).toBe(true);

});

});

});

### README.md

markdown

# Order Management System - Jest Tests

This repository contains unit tests for the Order Management System's `Order` class using Jest.

## Features Tested

- Adding items to an order

- Removing items from an order

- Processing an order

- Preventing modifications after the order is processed

- Error handling for edge cases

## Requirements

- Node.js

- TypeScript

- Jest

## Installation

1. Clone the repository:

```bash

git clone https://your-repo-url.git

cd your-repo-directory

1. Install the dependencies:

**bash**

npm install

To run the tests, use the following command:

**bash**

npx jest