***Supermarket Billing System***

**Project Overview**

A **Java-based console application** that simulates the core functionality of a supermarket billing process using **Object-Oriented Programming (OOP)** principles. The system manages product inventory, calculates bills, and processes sales transactions.

**Project Details**

| Field | Value |
| --- | --- |
| **NAME** | Vansh |
| **STUDENT ID** | BCS2023318 |
| **COURSE** | B TECH CSE ( 5TH SEM) |
| **PROJECT** | Supermarket Billing System |
| **Technologies Used** | Java Programming Language, OOP Principles |

**Project Introduction**

**Objective**

* Create a simple

**billing and inventory system** for a supermarket.

* Demonstrate core

**OOP concepts** like Encapsulation, Abstraction, and Polymorphism in a business scenario.

* Implement a

**modular and extensible** design for future feature additions.

**Class Diagram & Structure**

The system's design utilizes three main components: an Interface for basic actions, a Base Class for product properties, and a Concrete Class for a specific product type, all managed by a main system class.

| **Interface: Transactable** | **Abstract Class: Product** | **Concrete Class: GroceryItem** |
| --- | --- | --- |
| **Methods:** | **Properties:** | **Properties:** |
| + calculateTax() | - productId | - isTaxExempt (Boolean) |
| + generateReceiptEntry() | - name | **Methods:** |
|  | - unitPrice | + calculateTax() (Override) |
|  | - quantity | + generateReceiptEntry() (Override) |
|  | **Methods:** | + applyDiscount() |
|  | + getPrice() (Abstract) |  |
|  | + showDetails() |  |

**Relationship**

* **GroceryItem** extends **Product** (Inheritance).
* **GroceryItem** implements **Transactable** (Abstraction).
* **BillingSystem** class manages an ArrayList of **Product** objects (Composition/Runtime Polymorphism).

**OOP Principles Implementation**

**ABSTRACTION**

Abstraction is achieved through the use of an interface and an abstract class, hiding implementation complexity.

Java

// Interface abstraction: defines contract for transaction behavior

interface Transactable {

double calculateTax();

String generateReceiptEntry();

}

// Abstract class abstraction: base for all product types

abstract class Product {

public abstract double getPrice(); // Forces subclasses to define pricing logic

public void showDetails() { /\* Common details display logic \*/ }

}

**ENCAPSULATION**

Data members are declared as

private to restrict direct access, and controlled access is provided via public getter methods.

Java

private int productId;

private String name;

private double unitPrice;

public String getName() { return name; } // Controlled read access

public double getUnitPrice() { return unitPrice; }

**INHERITANCE**

The

Grocery Item class inherits common properties and methods from Product and implements the contract defined in Transactable.

Java

class GroceryItem extends Product implements Transactable {

// inherits product details and implements transaction methods

}

**POLYMORPHISM**

Method overriding and runtime polymorphism enable flexible handling of different product types.

Java

// Method overriding: specific tax logic for grocery items

@Override

public double calculate eTax() {

// Specific implementation for GroceryItem

}

// Runtime polymorphism: treating all product types generically

Product[] inventory = {

new GroceryItem(101, "Milk", 3.50),

new ElectronicsItem(202, "Charger", 15.00) // Assuming another subclass

};

// inventory[i].calculateTax() will call the correct subclass method at runtime

**Key Features**

Transaction Management

* **Add/Scan Item:** Products are added to a customer's bill based on ID and quantity.
* **Calculate Total:** The system calculates the subtotal, applicable taxes, and the final grand total.
* **Receipt Generation:** Creates a detailed, line-by-line breakdown of the transaction.

Inventory and Pricing

* **Product Status:** Tracks the current quantity of items available in the inventory.
* **Pricing Logic:** Different product types (e.g., groceries, electronics) can have different tax or discount rules.

Robustness

* **Input Validation:** Checks for valid product IDs and quantities.
* **Error Handling:** Manages scenarios like out-of-stock items or invalid user choices.

Menu System Flow

SUPERMARKET BILLING SYSTEM

↓

========= MAIN MENU =========

1. Start New Transaction

2. View Inventory

3. Add New Product to Inventory

4. Exit

↓

User Choice → Corresponding Action

**Sample Code**

A simplified version for the Product and Billing System to demonstrate core management.

Java

import java.util.ArrayList;

import java.util.Scanner;

// Simplified Product class (abstract for extensibility)

abstract class Product {

protected int productId;

protected String name;

protected double unitPrice;

public Product(int productId, String name, double unitPrice) {

this.productId = productId;

this.name = name;

this.unitPrice = unitPrice;

}

public int getProductId() { return productId; }

public String getName() { return name; }

public double getUnitPrice() { return unitPrice; }

// Abstract method to force specific product classes to implement

public abstract double calculateLineItemTotal(int quantity);

}

// Concrete implementation

class GroceryItem extends Product {

private static final double TAX\_RATE = 0.05; // 5% tax

public GroceryItem(int productId, String name, double unitPrice) {

super(productId, name, unitPrice);

}

// Line item total including tax

@Override

public double calculateLineItemTotal(int quantity) {

double subTotal = unitPrice \* quantity;

return subTotal \* (1 + TAX\_RATE);

}

}

class BillingSystem {

private ArrayList<Product> inventory;

// ... other methods for inventory management, transaction processing, etc.

public BillingSystem() {

inventory = new ArrayList<>();

// Pre-populate inventory for demonstration

inventory.add(new GroceryItem(101, "Apples (1kg)", 2.50));

inventory.add(new GroceryItem(102, "Bread Loaf", 1.99));

}

public Product findProduct(int id) {

for (Product p : inventory) {

if (p.getProductId() == id) {

return p;

}

}

return null;

}

public void processSale(int productId, int quantity) {

Product product = findProduct(productId);

if (product != null) {

double finalPrice = product.calculateLineItemTotal(quantity);

System.out.printf("%s x%d added. Total: $%.2f%n",

product.getName(), quantity, finalPrice);

} else {

System.out.println("Product ID " + productId + " not found!");

}

}

}

public class SupermarketBillingMain {

public static void main(String[] args) {

BillingSystem system = new BillingSystem();

Scanner sc = new Scanner(System.in);

System.out.println("\n--- Supermarket Billing System ---");

system.processSale(101, 3); // Apples x3

system.processSale(105, 1); // Invalid product

// ... Menu logic would go here

sc.close();

}

}

SAMPLE OUTPUT

Let's walk through the code execution and determine what the **output** will be when the main() method runs.

**✅ 1. Instantiating BillingSystem**

* The constructor adds **3 products** to the inventory:
  + **ID 101:** Apples (1kg), $2.50
  + **ID 102:** Bread Loaf, $1.99
  + **ID 103:** Milk (1L), $1.50

**✅ 2. system.showInventory() prints:**

Available Products:

ID: 101 | Name: Apples (1kg) | Price: $2.50

ID: 102 | Name: Bread Loaf | Price: $1.99

ID: 103 | Name: Milk (1L) | Price: $1.50

**✅ 3. system.processSale(101, 3)**

* Product ID 101 is **Apples (1kg)**, unit price = **$2.50**
* Quantity = **3**
* Subtotal = 2.50 × 3 = **7.50**
* Tax = 5%, so total = 7.50 × 1.05 = **7.875**
* Rounded to 2 decimal places → **$7.88**

Output:

Apples (1kg) x3 added. Total with tax: $7.88

**❌ 4. system.processSale(105, 1)**

* Product ID 105 does **not exist**, so output:

❌ Product ID 105 not found!

**✅ Final Combined Output:**

--- Supermarket Billing System ---

Available Products:

ID: 101 | Name: Apples (1kg) | Price: $2.50

ID: 102 | Name: Bread Loaf | Price: $1.99

ID: 103 | Name: Milk (1L) | Price: $1.50

Apples (1kg) x3 added. Total with tax: $7.88

❌ Product ID 105 not found!

**Future Enhancements**

Potential Extensions

1. **Add More Product Types:** Implement ElectronicsItem or ClothingItem subclasses with unique tax/warranty/discount rules.
2. **Discount & Loyalty:** Implement methods to apply percentage or flat-rate discounts and track customer loyalty points.
3. **Inventory Stock Management:** Track current stock levels and update them after each sale.
4. **Reporting:** Generate daily/weekly sales and tax reports.

Additional Features

* **GUI Interface:** Implement a graphical user interface instead of a console-based one.
* **Database Integration:** Use a database (like MySQL or SQLite) to persist inventory and sales data.
* **Multi-User Support:** Allow multiple cashiers to use the system simultaneously.

**Conclusion**

The Supermarket Billing System successfully applies core

**Object-Oriented Programming principles** in a practical, commercial context. The use of

**Inheritance** and **Polymorphism** makes the system **scalable** and **flexible** to accommodate new product categories and complex pricing rules. This project serves as a strong foundation for a full-featured Point-of-Sale (POS) system.

