Implement OOPs

Assignment – 1

By-Kunal Sudhir Mishra

Task 1: Classes and Their Attributes

Task 2: Class Creation:

• Create the classes (Customers, Products, Orders, OrderDetails and Inventory) with the specified attributes.

• Implement the constructor for each class to initialize its attributes.

• Implement methods as specified

Source code

- **Customers**

```
class Customers:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address


    def get_customer_details(self):
        print(f"Customer ID: {self.customer_id}")
        print(f"Name: {self.first_name} {self.last_name}")
        print(f"Email: {self.email}")
        print(f"Phone: {self.phone}")
        print(f"Address: {self.address}")
        pass

    def update_customer_info(self, new_email=None, new_phone=None, new_address=None):
        if new_email:
            self.email = new_email
        if new_phone:
```

```
    def update_customer_info(self, new_email=None, new_phone=None, new_address=None):
        if new_email:
            self.email = new_email
        if new_phone:
            self.phone = new_phone
        if new_address:
            self.address = new_address
        print("Customer information updated successfully.")
        pass
```

- **Products**

```
1 usage
class Products:
    def __init__(self, product_id, product_name, description, price):
        self.product_id = product_id
        self.product_name = product_name
        self.description = description
        self.price = price

    def get_product_details(self):
        print(f"Product ID: {self.product_id}")
        print(f"Product Name: {self.product_name}")
        print(f"Description: {self.description}")
        print(f"Price: {self.price}")
        pass

    def update_product_info(self, new_price=None, new_description=None):

        if new_price:
            self.price = new_price
        if new_description:
            self.description = new_description
        print("Product information updated successfully.")
        pass
```

- **Orders**

```
class Orders:
    def __init__(self, order_id, customer, order_date, total_amount):
        self.order_id = order_id
        self.customer = customer
        self.order_date = order_date
        self.total_amount = total_amount

    1 usage
    def calculate_total_amount(self, order_details_list):
        # Calculate the total amount of the order.
        total_amount = 0
        for order_detail in order_details_list:
            total_amount += order_detail.calculate_subtotal()
        return total_amount
        pass
```

```
    def get_order_details(self, order_details_list):
        # Retrieves and displays the details of the order (e.g., product list and quant
        print(f"Order ID: {self.order_id}")
        print(f"Order Date: {self.order_date}")
        print(f"Total Amount: ${self.calculate_total_amount()}")
        print("Order Details:")
        for order_detail in order_details_list:
            order_detail.get_order_detail_info()
        pass

    def update_order_status(self, new_status):
        if new_status in ["Processing", "Shipped", "Delivered", "Cancelled"]:
            self.order_status = new_status
            print(f"Order status updated to: {new_status}")
        else:
            print("Invalid order status.")

        pass
```

- **OrderDetails**

```
2 usages
class OrderDetails:
    def __init__(self, order_detail_id, order, product, quantity):
        self.order_detail_id = order_detail_id
        self.order = order
        self.product = product
        self.quantity = quantity

    4 usages (1 dynamic)
    def calculate_subtotal(self):
        return self.quantity * self.product.price

    1 usage (1 dynamic)
    def get_order_detail_info(self):
        print(f"Order Detail ID: {self.order_detail_id}")
        print(f"Product: {self.product.product_name}")
        print(f"Quantity: {self.quantity}")
        print(f"Subtotal: ${self.calculate_subtotal()}")
```

```
    def update_quantity(self, new_quantity):

        if new_quantity > 0:
            self.quantity = new_quantity
            print(f"Quantity updated to: {new_quantity}")
        else:
            print("Invalid quantity.")

    def add_discount(self, discount_percentage):
        if 0 <= discount_percentage <= 100:
            discount_amount = (discount_percentage / 100) * self.calculate_subtotal()
            discounted_subtotal = self.calculate_subtotal() - discount_amount
            print(f"Discount applied: {discount_percentage}%")
            print(f"Discounted Subtotal: ${discounted_subtotal}")
        else:
            print("Invalid discount percentage.")
```

- **Inventory**

```
1 usage
class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock, last_stock_update):
        self.inventory_id = inventory_id
        self.product = product
        self.quantity_in_stock = quantity_in_stock
        self.last_stock_update = last_stock_update

    def get_product(self):
        # A method to retrieve the product associated with this inventory item.
        return self.product

    def get_quantity_in_stock(self):
        # A method to get the current quantity of the product in stock.
        return self.quantity_in_stock
```

```python
def add_to_inventory(self, quantity):
    if quantity > 0:
        self.quantity_in_stock += quantity
        print(f"{quantity} units of {self.product.product_name} added to inventory.")
    else:
        print("Invalid quantity.")

def remove_from_inventory(self, quantity):
    # A method to remove a specified quantity of the product from the inventory.
    if 0 < quantity <= self.quantity_in_stock:
        self.quantity_in_stock -= quantity
        print(f"{quantity} units of {self.product.product_name} removed from inventory.")
    else:
        print("Invalid quantity or insufficient stock.")

def update_stock_quantity(self, new_quantity):
    # A method to update the stock quantity to a new value.
    if new_quantity >= 0:
        self.quantity_in_stock = new_quantity
        print(f"Stock quantity updated to: {new_quantity}")
    else:
        print("Invalid stock quantity.")
```

Driver Code

```python
from Customers import Customers
from Products import Products
from Orders import Orders
from OrderDetails import OrderDetails
from Inventory import Inventory
from DatabaseConnector import DatabaseConnector

1 usage
def update_customer_info():
    customer_id = int(input("Enter CustomerID: "))
    new_email = input("Enter new email: ")
    new_phone = input("Enter new phone: ")
    new_address = input("Enter new address: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Customer with ID: {customer_id} to Email: {new_email}, Phone: {new_phone}, Address: {new_address}")

        cursor.execute("""
            UPDATE Customers
            SET Email = %s, Phone = %s, Address = %s
            WHERE CustomerID = %s
        """, (new_email, new_phone, new_address, customer_id))

        db_connector.connection.commit()
```

```
        db_connector.connection.commit()

        print("Customer information updated successfully.")
    except Exception as e:
        print(f"Error updating customer information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()

1 usage
def update_product_info():
    product_id = int(input("Enter ProductID: "))
    new_price = float(input("Enter new price: "))
    new_description = input("Enter new description: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Product with ID: {product_id} to Price: {new_price}, Description: {new_description}")

        cursor.execute("""
            UPDATE Products
            SET Price = %s, Description = %s
            WHERE ProductID = %s
        """, (new_price, new_description, product_id))

        db_connector.connection.commit()
```

```
1 usage
def update_order_info():
    order_id = int(input("Enter OrderID: "))
    new_status = input("Enter new order status: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Order with ID: {order_id} to Status: {new_status}")

        cursor.execute("""
            UPDATE Orders
            SET OrderStatus = %s
            WHERE OrderID = %s
        """, (new_status, order_id))

        db_connector.connection.commit()

        print("Order information updated successfully.")
    except Exception as e:
        print(f"Error updating order information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

```
1 usage
def update_product_info():
    product_id = int(input("Enter ProductID: "))
    new_price = float(input("Enter new price: "))
    new_description = input("Enter new description: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Product with ID: {product_id} to Price: {new_price}, Description: {new_description}")

        cursor.execute("""
            UPDATE Products
            SET Price = %s, Description = %s
            WHERE ProductID = %s
        """, (new_price, new_description, product_id))

        db_connector.connection.commit()

        print("Product information updated successfully.")
    except Exception as e:
        print(f"Error updating product information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

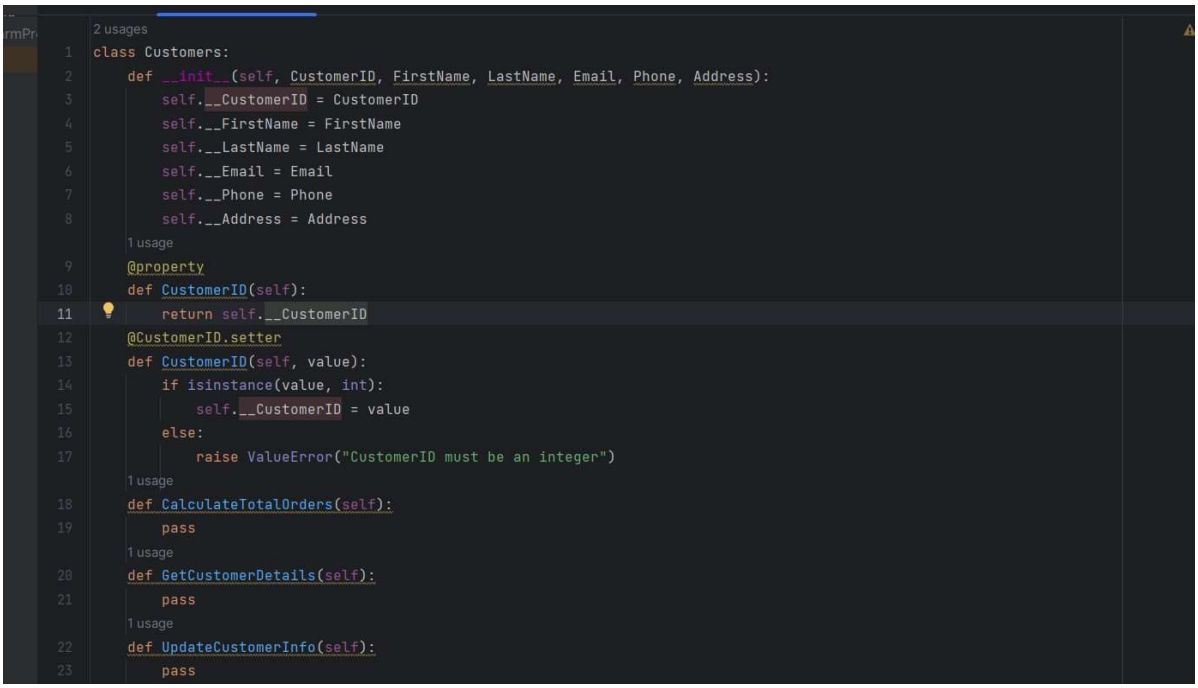
```
    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Inventory with ID: {inventory_id} to Quantity: {new_quantity}")

        cursor.execute("""
            UPDATE Inventory
            SET QuantityInStock = %s
            WHERE InventoryID = %s
        """, (new_quantity, inventory_id))

        db_connector.connection.commit()

        print("Inventory information updated successfully.")
    except Exception as e:
        print(f"Error updating Inventory information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()

if __name__ == "__main__":
    update_customer_info()
    update_product_info()
    update_order_info()
    update_order_details_info()
    update_inventory_info()
```

Task 3: Encapsulation:

• Implement encapsulation by making the attributes private and providing public properties (getters and setters) for each attribute.

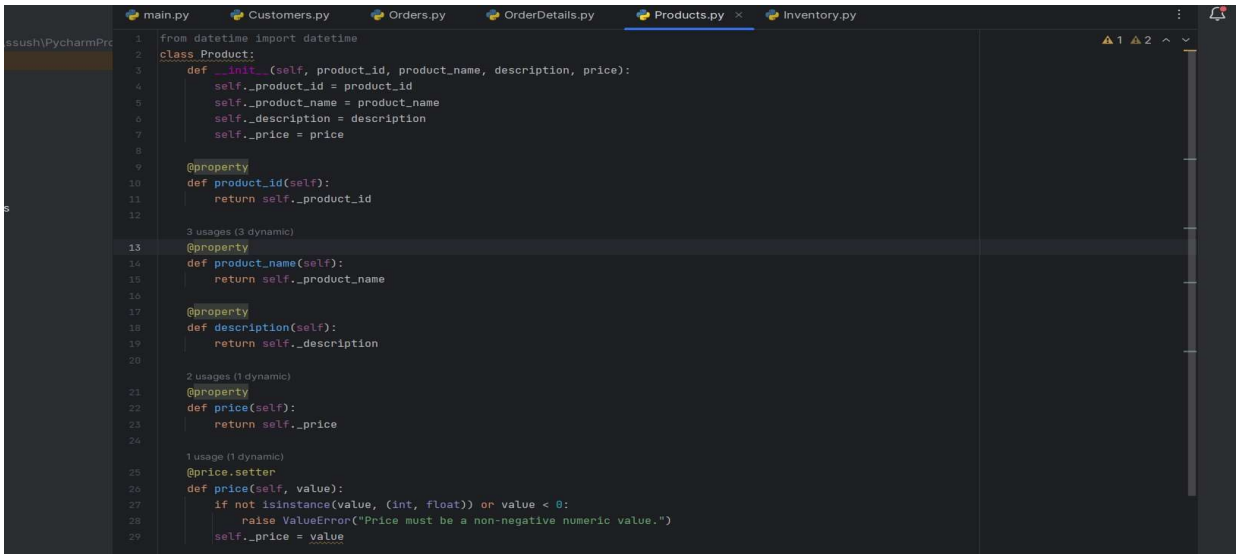
• Add data validation logic to setter methods (e.g., ensure that prices are non-negative, quantities are positive integers).

- **Customers.py**



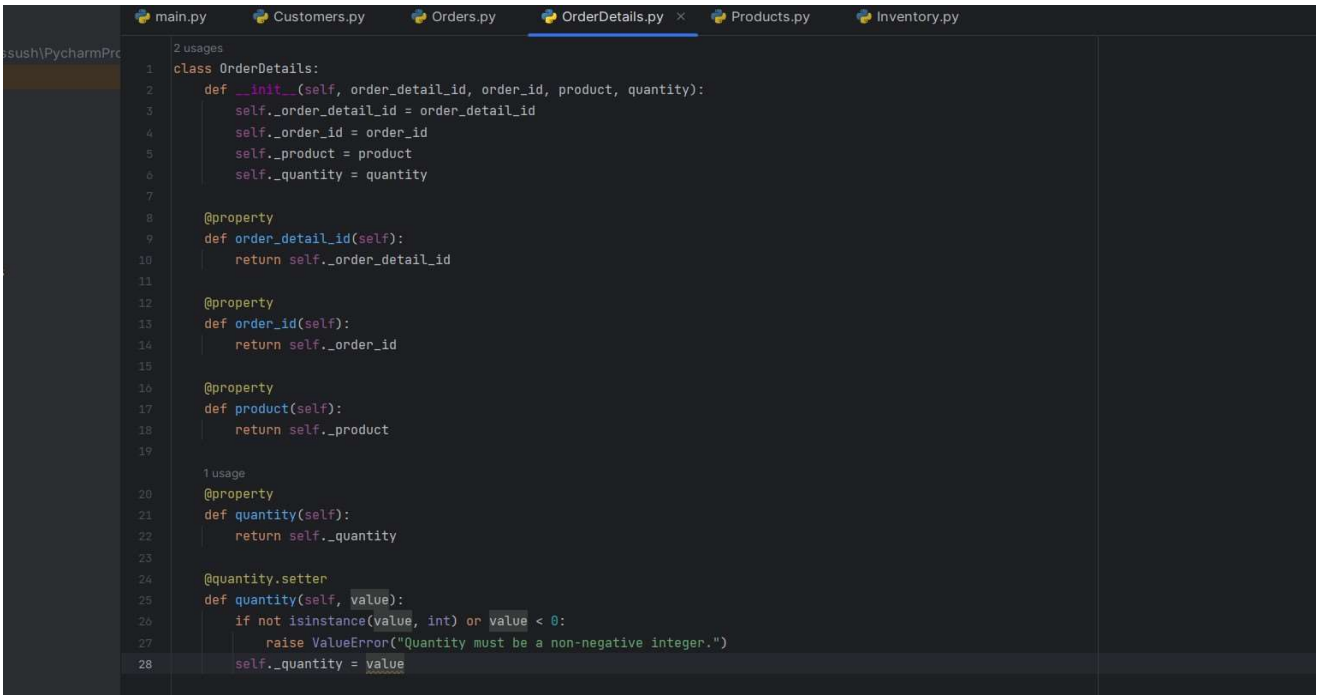
```
2 usages
class Customers:
    def __init__(self, CustomerID, FirstName, LastName, Email, Phone, Address):
        self.__CustomerID = CustomerID
        self.__FirstName = FirstName
        self.__LastName = LastName
        self.__Email = Email
        self.__Phone = Phone
        self.__Address = Address
    1 usage
    @property
    def CustomerID(self):
        return self.__CustomerID
    @CustomerID.setter
    def CustomerID(self, value):
        if isinstance(value, int):
            self.__CustomerID = value
        else:
            raise ValueError("CustomerID must be an integer")
    1 usage
    def CalculateTotalOrders(self):
        pass
    1 usage
    def GetCustomerDetails(self):
        pass
    1 usage
    def UpdateCustomerInfo(self):
        pass
```

- **Products.py**

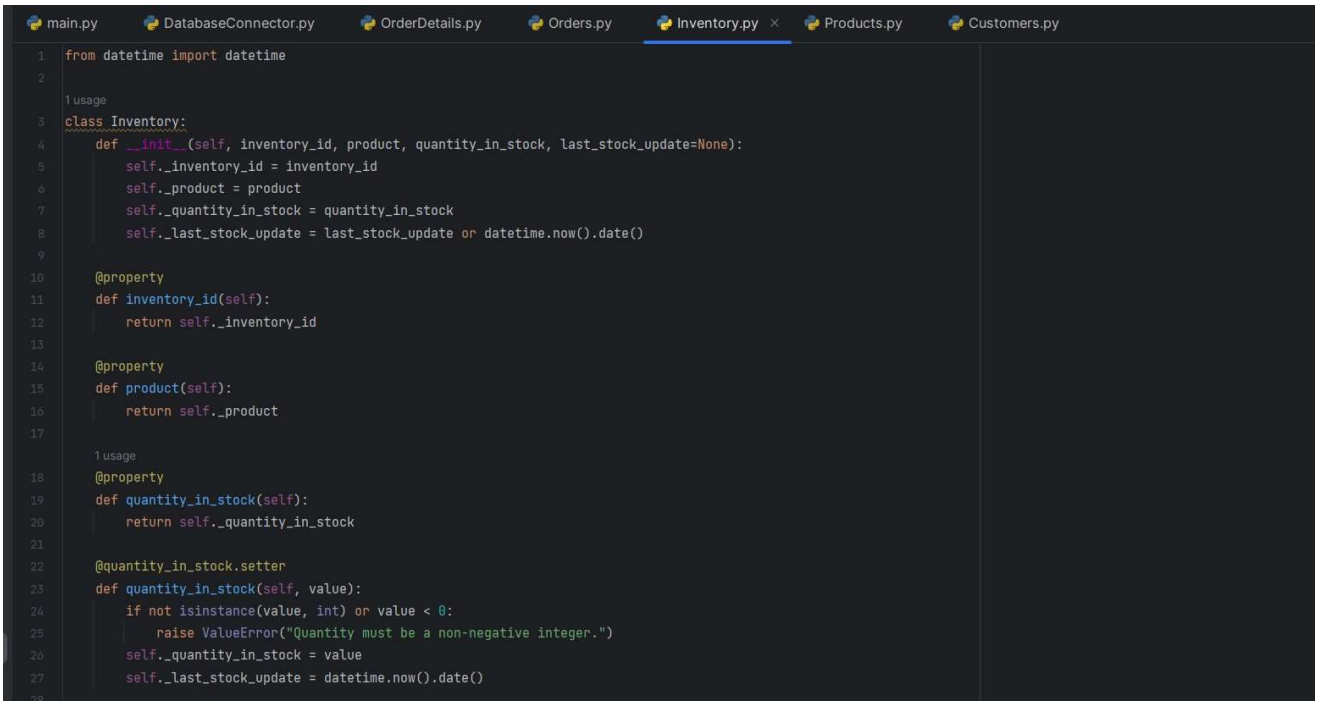


```
from datetime import datetime
class Product:
    def __init__(self, product_id, product_name, description, price):
        self._product_id = product_id
        self._product_name = product_name
        self._description = description
        self._price = price

    @property
    def product_id(self):
        return self._product_id

    3 usages (3 dynamic)
    @property
    def product_name(self):
        return self._product_name

    @property
    def description(self):
        return self._description

    2 usages (1 dynamic)
    @property
    def price(self):
        return self._price

    1 usage (1 dynamic)
    @price.setter
    def price(self, value):
        if not isinstance(value, (int, float)) or value < 0:
            raise ValueError("Price must be a non-negative numeric value.")
        self._price = value
```

- **OrderDetails.py**



- **Inventory.py**



Task 4: Composition:

Ensure that the Order and OrderDetail classes correctly use composition to reference Customer and Product objects.

• **OrderDetails Class with Composition:**

Similarly, in the OrderDetails class, we want to establish composition relationships with both the Orders and Products classes to represent the details of each order, including the product being ordered. In the OrderDetails class, we've added two private attributes, order and product, of types Orders and Products, respectively, establishing composition relationships. The Order property provides access to the Orders object associated with the order detail, and the Product property provides access to the Products object representing the product in the order detail.

```
3 usages
class OrderDetails:
    def __init__(self, OrderDetailID, OrderID, ProductID, Quantity):
        self.OrderDetailID = OrderDetailID
        self.OrderID = OrderID
        self.ProductID = ProductID
        self.Quantity = Quantity

    def GetOrderDetailInfo(self):
        print(f'OrderDetailID: {self.OrderDetailID}, OrderID: {self.OrderID}, ProductID: {self.ProductID}, Quantity: {self.Quantity}')

```

```
mysql> select * from orderdetails;
+----------------+---------+-----------+----------+
| OrderDetailID | OrderID | ProductID | Quantity |
+----------------+---------+-----------+----------+
|              1 |       1 |         1 |       50 |
|              2 |       1 |         1 |        2 |
|              3 |       1 |         1 |       50 |
+----------------+---------+-----------+----------+
3 rows in set (0.00 sec)

mysql>
```

- **Customers and Products Classes:**
  The Customers and Products classes themselves may not have direct composition relationships with other classes in this scenario. However, they serve as the basis for composition relationships in the Orders and OrderDetails classes, respectively.

```
class Products:
    def __init__(self, ProductID, ProductName, Description, Price):
        self.ProductID = ProductID
        self.ProductName = ProductName
        self.Description = Description
        self.Price = Price

    def GetProductDetails(self):
        print(f'ProductID: {self.ProductID}, ProductName: {self.ProductName}, Description: {self.Description}, Price: {self.Price}')
```

```
mysql> select * from products;
+-----------+-------------+---------------+----------+
| ProductID | ProductName | Description   | Price    |
+-----------+-------------+---------------+----------+
|         1 | Gadget      | cool laptop   | 50000.00 |
|         2 | Gadget      | A cool gadget |    99.99 |
|         3 | Gadget      | A cool gadget |    99.99 |
|         4 | Gadget      | cool tv       |  5000.00 |
|         5 | Gadget      | A cool gadget |    99.99 |
|         6 | Gadget      | coool         | 52489.00 |
|         7 | Gadget      | A cool gadget |    99.99 |
|         8 | Gadget      | cool tablet   |  7000.00 |
|         9 | Gadget      | A cool gadget |    99.99 |
|        10 | Gadget      | A cool gadget |    99.99 |
|        11 | Gadget      | A cool gadget |    99.99 |
|        12 | Gadget      | A cool gadget |    99.99 |
+-----------+-------------+---------------+----------+
12 rows in set (0.01 sec)
```

- **Inventory Class:**

The Inventory class represents the inventory of products available for sale. It can have composition relationships with the Products class to indicate which products are in the inventory.

```python
class Inventory:
    def __init__(self, InventoryID, Product, QuantityInStock, LastStockUpdate):
        self.InventoryID = InventoryID
        self.Product = Product
        self.QuantityInStock = QuantityInStock
        self.LastStockUpdate = LastStockUpdate

    def GetInventoryDetails(self):
        print(f'InventoryID: {self.InventoryID}, ProductID: {self.ProductID}, QuantityInStock: {self.QuantityInStock}, LastStockUpdate: {self.LastStockUpdate}')
```

```
mysql> select * from inventorys;
ERROR 1146 (42S02): Table 'techshopdb.inventorys' doesn't exist
mysql> select * from inventory;
+-------------+-----------+-----------------+-----------------+
| InventoryID | ProductID | QuantityInStock | LastStockUpdate |
+-------------+-----------+-----------------+-----------------+
|           1 |         1 |             500 | 2024-01-31      |
|           2 |         1 |              50 | 2024-01-31      |
|           3 |         1 |              50 | 2024-01-31      |
|           4 |         1 |             100 | 2024-01-30      |
|           5 |         1 |             500 | 2024-01-30      |
|           6 |         1 |             100 | 2024-01-30      |
+-------------+-----------+-----------------+-----------------+
6 rows in set (0.00 sec)

mysql> use techshopdb;;
Database changed
```

Task 5: Exceptions handling

• Data Validation: Challenge: Validate user inputs and data from external sources (e.g., user registration, order placement). Scenario: When a user enters an invalid email address during registration. o Exception Handling: Throw a custom InvalidDataException with a clear error message.

• Inventory Management: Challenge: Handling inventory-related issues, such as selling more products than are in stock. Scenario: When processing an order with a quantity that exceeds the available stock. Exception Handling: Throw an InsufficientStockException and update the order status accordingly.

• Order Processing: Challenge: Ensuring the order details are consistent and complete before processing. Scenario: When an order detail lacks a product reference. o Exception Handling: Throw an IncompleteOrderException with a message explaining the issue.

• Payment Processing: Challenge: Handling payment failures or declined transactions. o Scenario: When processing a payment for an order and the payment is declined. o Exception Handling: Handle payment-specific exceptions (e.g., PaymentFailedException) and initiate retry or cancellation processes.

• File I/O (e.g., Logging): Challenge: Logging errors and events to files or databases. o Scenario: When an error occurs during data persistence (e.g., writing a log entry). Exception Handling: Handle file I/O exceptions (e.g., IOException) and log them appropriately.

• Database Access: Challenge: Managing database connections and queries. o Scenario: When executing a SQL query and the database is offline. Exception Handling: Handle database-specific exceptions (e.g., SqlException) and implement connection retries or failover mechanisms.

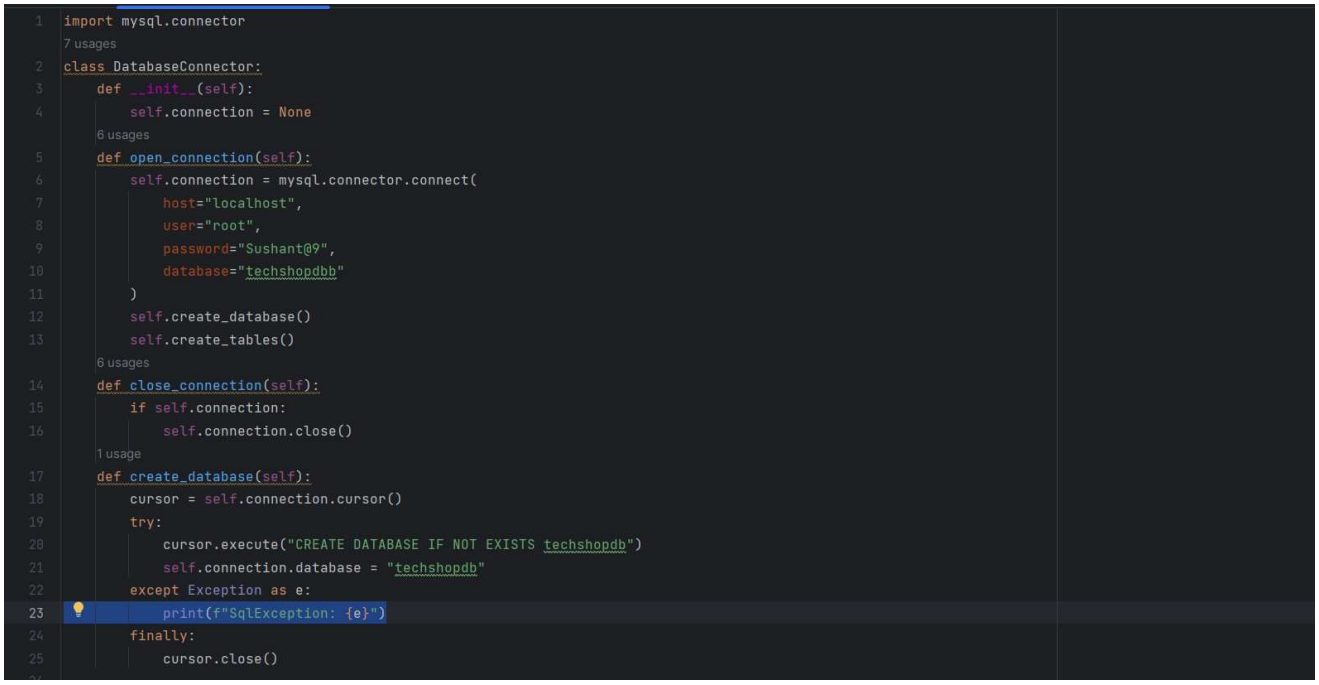
• Concurrency Control: Challenge: Preventing data corruption in multi-user scenarios. o Scenario: When two users simultaneously attempt to update the same order. Exception Handling: Implement optimistic concurrency control and handle ConcurrencyException by notifying users to retry.

• Security and Authentication: Challenge: Ensuring secure access and handling unauthorized access attempts. Scenario: When a user tries to access sensitive information without proper authentication. o Exception Handling: Implement custom AuthenticationException and AuthorizationException to handle security-related issues.

```
def insert_customer(self, first_name, last_name, email, phone, address):
    cursor = self.connection.cursor()
    try:
        query = "INSERT INTO Customers (FirstName, LastName, Email, Phone, Address) VALUES (%s, %s, %s, %s, %s)"
        values = (first_name, last_name, email, phone, address)
        cursor.execute(query, values)
        self.connection.commit()
        print("Customer inserted successfully.")
    except Exception as e:
        print(f"InvalidDataException: {e}")
    finally:
        cursor.close()
```

```
140
141     def insert_inventory(self, product_id, quantity_in_stock, last_stock_update):
142         cursor = self.connection.cursor()
143         try:
144             query = "INSERT INTO Inventory (ProductID, QuantityInStock, LastStockUpdate) VALUES (%s, %s, %s)"
145             values = (product_id, quantity_in_stock, last_stock_update)
146             cursor.execute(query, values)
147             self.connection.commit()
148             print("Inventory inserted successfully.")
149         except Exception as e:
150             print(f"InsufficientStockException: {e}")
151         finally:
152             cursor.close()
```

```
127
128     def insert_order_detail(self, order_id, product_id, quantity):
129         cursor = self.connection.cursor()
130         try:
131             query = "INSERT INTO OrderDetails (OrderID, ProductID, Quantity) VALUES (%s, %s, %s)"
132             values = (order_id, product_id, quantity)
133             cursor.execute(query, values)
134             self.connection.commit()
135             print("Order detail inserted successfully.")
136         except Exception as e:
137             print(f"IncompleteOrderException: {e}")
138         finally:
139             cursor.close()
```



```python
import mysql.connector
7 usages
class DatabaseConnector:
    def __init__(self):
        self.connection = None
    6 usages
    def open_connection(self):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="Sushant@9",
            database="techshopdbb"
        )
        self.create_database()
        self.create_tables()
    6 usages
    def close_connection(self):
        if self.connection:
            self.connection.close()
    1 usage
    def create_database(self):
        cursor = self.connection.cursor()
        try:
            cursor.execute("CREATE DATABASE IF NOT EXISTS techshopdb")
            self.connection.database = "techshopdb"
        except Exception as e:
            print(f"SqlException: {e}")
        finally:
            cursor.close()
```

Task 6: Collections

• **Managing Products List:**

Challenge: Maintaining a list of products available for sale (List). o Scenario: Adding, updating, and removing products from the list. o Solution: Implement methods to add, update, and remove products. Handle exceptions for duplicate products, invalid updates, or removal of products with existing orders.

• **Managing Orders List:**

Challenge: Maintaining a list of customer orders (List). o Scenario: Adding new orders, updating order statuses, and removing canceled orders. o Solution: Implement methods to add new orders, update order statuses, and remove canceled orders. Ensure that updates are synchronized with inventory and payment records.

• **Sorting Orders by Date:** Challenge: Sorting orders by order date in ascending or descending order. o Scenario: Retrieving and displaying orders based on specific date ranges. o Solution: Use the List collection and provide custom sorting methods for order date. Consider implementing SortedList if you need frequent sorting operations.

• **Inventory Management with SortedList:** Challenge: Managing product inventory with a SortedList based on product IDs. Scenario: Tracking the quantity in stock for each product and quickly retrieving inventory information. oSolution: Implement a SortedList where keys are product IDs. Ensure that inventory updates are synchronized with product additions and removals.

• **Handling Inventory Updates:**

Challenge: Ensuring that inventory is updated correctly when processing orders. o Scenario: Decrementing product quantities in stock when orders are placed. Solution: Implement a method to update inventory quantities when orders are processed. Handle exceptions for insufficient stock. • Product Search and Retrieval: Challenge: Implementing a search functionality to find products based on various criteria (e.g., name, category). Scenario: Allowing customers to search for products. o Solution: Implement custom search methods using LINQ queries on the List collection. Handle exceptions for invalid search criteria.

• **Duplicate Product Handling:**

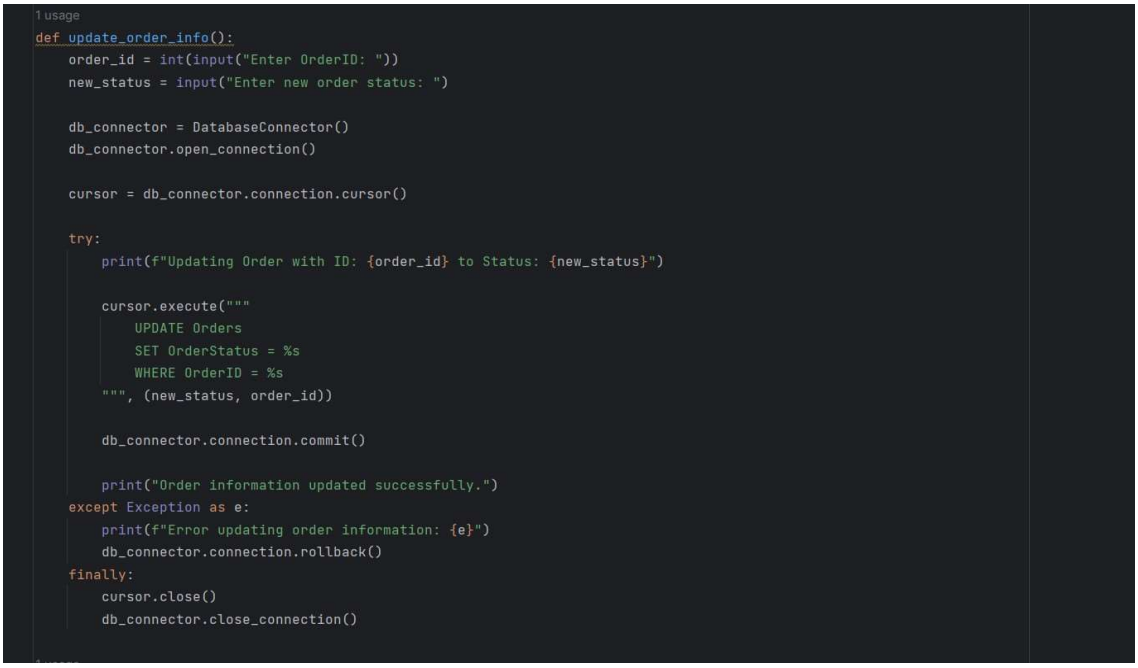
Challenge: Preventing duplicate products from being added to the list. o Scenario: When a product with the same name or SKU is added.

Solution: Implement logic to check for duplicates before adding a product to the list. Raise exceptions or return error messages for duplicates.

• **Payment Records List:** o Challenge: Managing a list of payment records for orders (List). Scenario: Recording and updating payment information for each order. Solution: Implement methods to record payments, update payment statuses, and handle payment errors. Ensure that payment records are consistent with order records.

• **OrderDetails and Products Relationship:** Challenge: Managing the relationship between OrderDetails and Products. Scenario: Ensuring that order details accurately reflect the products available in the inventory. Solution: Implement methods to validate product availability in the inventory before adding order details. Handle exceptions for unavailable products.

```
1 usage
def update_product_info():
    product_id = int(input("Enter ProductID: "))
    new_price = float(input("Enter new price: "))
    new_description = input("Enter new description: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Product with ID: {product_id} to Price: {new_price}, Description: {new_description}")

        cursor.execute("""
            UPDATE Products
            SET Price = %s, Description = %s
            WHERE ProductID = %s
        """, (new_price, new_description, product_id))

        db_connector.connection.commit()

        print("Product information updated successfully.")
    except Exception as e:
        print(f"Error updating product information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```



```
1 usage
def update_order_info():
    order_id = int(input("Enter OrderID: "))
    new_status = input("Enter new order status: ")

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Order with ID: {order_id} to Status: {new_status}")

        cursor.execute("""
            UPDATE Orders
            SET OrderStatus = %s
            WHERE OrderID = %s
        """, (new_status, order_id))

        db_connector.connection.commit()

        print("Order information updated successfully.")
    except Exception as e:
        print(f"Error updating order information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

main.py | DatabaseConnector.py | Customers.py | Products.py | Orders.py | OrderDetails.py | Inventory.py

```python
def update_order_details_info():
    order_detail_id = int(input("Enter OrderDetailID: "))
    new_quantity = int(input("Enter new quantity: "))

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating OrderDetails with ID: {order_detail_id} to Quantity: {new_quantity}")

        cursor.execute("""
            UPDATE OrderDetails
            SET Quantity = %s
            WHERE OrderDetailID = %s
        """, (new_quantity, order_detail_id))

        db_connector.connection.commit()

        print("OrderDetails information updated successfully.")
    except Exception as e:
        print(f"Error updating OrderDetails information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

1 usage

```python
def update_inventory_info():
    inventory_id = int(input("Enter InventoryID: "))
    new_quantity = int(input("Enter new quantity: "))

    db_connector = DatabaseConnector()
    db_connector.open_connection()

    cursor = db_connector.connection.cursor()

    try:
        print(f"Updating Inventory with ID: {inventory_id} to Quantity: {new_quantity}")

        cursor.execute("""
            UPDATE Inventory
            SET QuantityInStock = %s
            WHERE InventoryID = %s
        """, (new_quantity, inventory_id))

        db_connector.connection.commit()

        print("Inventory information updated successfully.")
    except Exception as e:
        print(f"Error updating Inventory information: {e}")
        db_connector.connection.rollback()
    finally:
        cursor.close()
        db_connector.close_connection()
```

Task 7: Database Connectivity

• Implement a DatabaseConnector class responsible for establishing a connection to the "TechShopDB" database. This class should include methods for opening, closing, and managing database connections. • Implement classes for Customers, Products, Orders, OrderDetails, Inventory with properties, constructors, and methods for CRUD (Create, Read, Update, Delete) operations.

1: Customer Registration Description: When a new customer registers on the TechShop website, their information (e.g., name, email, phone) needs to be stored in the database. Task: Implement a registration form and database connectivity to insert new customer records. Ensure proper data validation and error handling for duplicate email addresses.

2: Product Catalog Management Description: TechShop regularly updates its product catalog with new items and changes in product details (e.g., price, description). These changes need to be reflected in the database. Task: Create an interface to manage the product catalog. Implement database connectivity to update product information. Handle changes in product details and ensure data consistency.

3: Placing Customer Orders Description: Customers browse the product catalog and place orders for products they want to purchase. The orders need to be stored in the database. Task: Implement an order processing system. Use database connectivity to record customer orders, update product quantities in inventory, and calculate order totals.

4: Tracking Order Status Description: Customers and employees need to track the status of their orders. The order status information is stored in the database. Task: Develop a feature that allows users to view the status of their orders. Implement database connectivity to retrieve and display order status information.

5: Inventory Management Description: TechShop needs to manage product inventory, including adding new products, updating stock levels, and removing discontinued items. Task: Create an inventory management system with database connectivity. Implement features for adding new products, updating quantities, and handling discontinued products.

6: Sales Reporting Description: TechShop management requires sales reports for business analysis. The sales data is stored in the database. Task: Design and implement a reporting system that retrieves sales data from the database and generates reports based on specified criteria.

7: Customer Account Updates Description: Customers may need to update their account information, such as changing their email address or phone number. Task: Implement a user profile management feature with database connectivity to allow customers to update their account details. Ensure data validation and integrity.

8: Payment Processing Description: When customers make payments for their orders, the payment details (e.g., payment method, amount) must be recorded in the database. Task: Develop a payment processing system that interacts with the database to record payment transactions, validate payment information, and handle errors.

9: Product Search and Recommendations Description: Customers should be able to search for products based on various criteria (e.g., name, category) and receive product recommendations. Task: Implement a product search and recommendation engine that uses database connectivity to retrieve relevant product information.

```python
import mysql.connector

class DatabaseConnector:
    def __init__(self, host, username, password, database):
        self.host = host
        self.username = username
        self.password = password
        self.database = database
        self.connection = None

    def open_connection(self):
        try:
            self.connection = mysql.connector.connect(
                host=self.host,
                user=self.username,
                password=self.password,
                database=self.database
            )
            print("Connected to the database.")
        except mysql.connector.Error as err:
            print(f"Error: {err}")

    def close_connection(self):
        if self.connection:
            self.connection.close()
            print("Connection closed.")

    def execute_query(self, query, values=None):
        cursor = self.connection.cursor()
        try:
            cursor.execute(query, values)
            self.connection.commit()
            print("Query executed successfully.")
        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            cursor.close()
```

```python
# main.py
from datetime import datetime
from Customers import Customers
from Products import Products
from Orders import Orders
from OrderDetails import OrderDetails
from Inventory import Inventory
from DatabaseConnector import DatabaseConnector

# Set your MySQL database credentials
db_host = "localhost"
db_user = "root"
db_password = "765795"
db_name = "TechShopDB"

# Create a DatabaseConnector instance
db_connector = DatabaseConnector(host=db_host, username=db_user,
password=db_password, database=db_name)

# Open the database connection
db_connector.open_connection()

def insert_data():
    # Create a sample instance for Karthika Mam
    karthika = Customers(customer_id=2, first_name="Karthika", last_name="Mam",
email="karthika@example.com",
                         phone="9876543210", address="123 Main Street")

    # Insert data into the Customers table
    db_connector.execute_query(
        "INSERT INTO Customers (CustomerID, FirstName, LastName, Email, Phone,
Address) VALUES (%s, %s, %s, %s, %s, %s)",
        (karthika.CustomerID, karthika.FirstName, karthika.LastName,
karthika.Email, karthika.Phone, karthika.Address)
    )

    # Create a sample instance for MacBook
    macbook = Products(product_id=2, product_name="MacBook", description="Apple
MacBook Pro", price=2000)

    # Insert data into the Products table for MacBook
    db_connector.execute_query(
        "INSERT INTO Products (ProductID, ProductName, Description, Price)
VALUES (%s, %s, %s, %s)",
        (macbook.ProductID, macbook.ProductName, macbook.Description,
macbook.Price)
    )

    # Create a sample instance for iPhone
    iphone = Products(product_id=3, product_name="iPhone", description="Apple
iPhone 13 Pro", price=1200)

    # Insert data into the Products table for iPhone
    db_connector.execute_query(
        "INSERT INTO Products (ProductID, ProductName, Description, Price)
VALUES (%s, %s, %s, %s)",
        (iphone.ProductID, iphone.ProductName, iphone.Description,
iphone.Price)
```

Output:

```
mysql> select*from customers;
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------+
| CustomerID | FirstName | LastName | Email                 | Phone      | Address                                       |
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------+
|          1 | Shivam    | Singh    | shivasingh414@gmail.com | 8340508631 | Street 12 Sector 9/B Bokaro Steel City Jharkhand |
|          2 | Karthika  | Mam      | karthika@example.com  | 9876543210 | 123 Main Street                               |
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------+
2 rows in set (0.00 sec)

mysql> select*from products;
+-----------+-------------+--------------------+---------+
| ProductID | ProductName | Description        | Price   |
+-----------+-------------+--------------------+---------+
|         2 | MacBook     | Apple MacBook Pro  | 2000.00 |
|         3 | iPhone      | Apple iPhone 13 Pro | 1200.00 |
+-----------+-------------+--------------------+---------+
2 rows in set (0.01 sec)

mysql> select*from inventory;
+-------------+-----------+----------------+---------------------+
| InventoryID | ProductID | QuantityInStock | LastStockUpdate     |
+-------------+-----------+----------------+---------------------+
|           2 |         2 |              5 | 2024-02-01 16:09:55 |
+-------------+-----------+----------------+---------------------+
1 row in set (0.00 sec)

mysql> select*from orders;
+---------+------------+---------------------+-------------+
| OrderID | CustomerID | OrderDate           | TotalAmount |
+---------+------------+---------------------+-------------+
|       2 |          2 | 2024-02-01 16:09:55 |     3200.00 |
+---------+------------+---------------------+-------------+
1 row in set (0.00 sec)

mysql> select*from orderdetails;
+--------------+---------+-----------+----------+
| OrderDetailID | OrderID | ProductID | Quantity |
+--------------+---------+-----------+----------+
|            2 |       2 |         3 |        3 |
+--------------+---------+-----------+----------+
1 row in set (0.00 sec)
```