TechShop, an electronic gadgets shop

Implement OOPs

Task 1: Classes and Their Attributes:

You are working as a software developer for TechShop, a company that sells electronic gadgets. Your task is to design and implement an application using Object-Oriented Programming (OOP) principles to manage customer information, product details, and orders. Below are the classes you need to create:

Customers Class:

Attributes: • CustomerID (int) • FirstName (string) • LastName (string) • Email (string) • Phone (string) • Address (string)

```
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
        self.totalOrders = 0
```

Methods: • CalculateTotalOrders(): Calculates the total number of orders placed by this customer. • GetCustomerDetails(): Retrieves and displays detailed information about the customer. • UpdateCustomerInfo(): Allows the customer to update their information (e.g., email, phone, or address).

```
    def calculate_total_orders(self):
        return self.totalOrders

    def get_customer_details(self):
        return (f"Customer ID: {self.CustomerID}\n"
                f"Name: {self.FirstName} {self.LastName}\n"
                f"Email: {self.Email}\nPhone: {self.Phone}\n"
                f"Address: {self.Address}")

    def update_customer_info(self, email=None, phone=None, address=None):
        if email:
            self.Email = email
        if phone:
            self.Phone = phone
        if address:
            self.Address = address
```

Products Class:

Attributes: • ProductID (int) • ProductName (string) • Description (string) • Price (decimal)

```
class Product:
    def __init__(self, productid, productname, description, price):
        self.productid = productid
        self.productname = productname
        self.description = description
        self.price = price
```

Methods: • GetProductDetails(): Retrieves and displays detailed information about the product. • UpdateProductInfo(): Allows updates to product details (e.g., price, description). • IsProductInStock(): Checks if the product is currently in stock.

```
    def get_product_details(self):
        return (f"Product ID: {self.productid}\n"
                f"Name: {self.productname}\n"
                f"Description: {self.description}\n"
                f"Price: ${self.price:.2f}")

    def update_product_info(self, price=None, description=None):
            self.price = price
            self.description = description
```

Orders Class:

Attributes: • OrderID (int) • Customer (Customer) - Use composition to reference the Customer who placed the order. • OrderDate (DateTime) • TotalAmount (decimal)

```python
from customers import Customers
class Order(Customers):
    def __init__(self, orderid, customer, orderdate, totalamount,status):
        self.orderid = orderid
        self.customer = customer
        self.orderdate = orderdate
        self.totalamount = totalamount
        self.status = status
        self.cancel = False
```

Methods:r • CalculateTotalAmount() - Calculate the total amount of the order. • GetOrderDetails(): Retrieves and displays the details of the order (e.g., product list and quantities). • UpdateOrderStatus(): Allows updating the status of the order (e.g., processing, shipped). • CancelOrder(): Cancels the order and adjusts stock levels for products.

```python
    def calculate_total_amount(self):
        return self.totalamount

    def get_order_details(self):
        return (f"Order ID: {self.orderid}\n"
                f"Customer: {self.customers.FirstName} {self.customers.LastName}\n"
                f"Order Date: {self.orderdate}\n"
                f"Total Amount: ${self.totalamount:.2f}")

    def update_order_status(self, status):
        self.status = status

    def cancel_order(self):
        self.cancel = True

```

OrderDetails Class:

Attributes: • OrderDetailID (int) • Order (Order) - Use composition to reference the Order to which this detail belongs. • Product (Product) - Use composition to reference the Product included in the order detail. • Quantity (int)

```python
from product import Product
class OrderDetail(Product):
    def __init__(self, orderdetailid, order, product, quantity):
        self.orderdetailid = orderdetailid
        self.order = order
        self.product = product
        self.quantity = quantity

```

Methods: • CalculateSubtotal() - Calculate the subtotal for this order detail. • GetOrderDetailInfo(): Retrieves and displays information about this order detail. • UpdateQuantity(): Allows updating the quantity of the product in this order detail. • AddDiscount(): Applies a discount to this order detail.

Inventory class:

Attributes: • InventoryID(int) • Product (Composition): The product associated with the inventory item. • QuantityInStock: The quantity of the product currently in stock. • LastStockUpdate

```python
from product import Product
class Inventory(Product):
    def __init__(self, inventoryid, product, quantityinstock, laststockupdate):
        self.inventoryid = inventoryid
        self.product = product
        self.quantityinstock = quantityinstock
        self.laststockupdate = laststockupdate

```

Methods:

• GetProduct(): A method to retrieve the product associated with this inventory item. • GetQuantityInStock(): A method to get the current quantity of the product in stock. • AddToInventory(int quantity): A method to add a specified quantity of the product to the inventory. • RemoveFromInventory(int quantity): A method to remove a specified quantity of the product from the inventory. • UpdateStockQuantity(int newQuantity): A method to update the stock quantity to a new value. • IsProductAvailable(int quantityToCheck): A method to check if a specified quantity of the product is available in the inventory. • GetInventoryValue(): A method to calculate the total value of the products in the inventory based on their prices and quantities. • ListLowStockProducts(int threshold): A method to list products with quantities below a specified threshold, indicating low stock. • ListOutOfStockProducts(): A method to list products that are out of stock. • ListAllProducts(): A method to list all products in the inventory, along with their quantities.

```python
    def get_product(self):
        return self.product

    def get_quantity_in_stock(self):
        return self.quantityinstock

    def add_to_inventory(self, quantity):
        self.quantityinstock += quantity

    def remove_from_inventory(self, quantity):
        self.quantityinstock -= quantity

    def update_stock_quantity(self, new_quantity):
        self.quantityinstock = new_quantity

    def is_product_available(self):
        if self.quantityinstock > 0:
            return True
        else:
            return False

    def get_inventory_value(self):
        return self.product.price * self.quantityinstock

    def list_low_stock_products(self, threshold):
        if self.quantityinstock < threshold:
            return f"{self.product.product_name}: {self.quantityinstock}"
```

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.

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).

```
    @property
    def customerID(self):
        return self._customerID

    @property
    def firstName(self):
        return self._firstName

    @firstName.setter
    def firstName(self, value):
        self._firstName = value

    @property
    def lastName(self):
        return self._lastName

    @lastName.setter
    def lastName(self, value):
        self._lastName = value

    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, value):
        self._email = value
```

```
    @property
    def phone(self):
        return self._phone

    @phone.setter
    def phone(self, value):
        self._phone = value

    @property
    def address(self):
        return self._address

    @address.setter
    def address(self, value):
        self._address = value
```

```python
    @property
    def productID(self):
        return self._productID

    @property
    def productName(self):
        return self._productName

    @productName.setter
    def productName(self, value):
        self._productName = value

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

    @description.setter
    def description(self, value):
        self._description = value

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if value >= 0:
            self._price = value
        else:
            raise ValueError("Price cannot be negative.")
```

```python
    @property
    def orderID(self):
        return self._orderID

    @property
    def customer(self):
        return self._customer

    @customer.setter
    def customer(self, value):
        self._customer = value

    @property
    def orderDate(self):
        return self._orderDate

    @orderDate.setter
    def orderDate(self, value):
        self._orderDate = value

    @property
    def totalAmount(self):
        return self._totalAmount

    @totalAmount.setter
    def totalAmount(self, value):
        if value >= 0:
            self._totalAmount = value
        else:
            raise ValueError("Total amount cannot be negative.")
```

```python
    @property
    def orderDetailID(self):
        return self._orderDetailID

    @property
    def order(self):
        return self._order

    @order.setter
    def order(self, value):
        self._order = value

    @property
    def product(self):
        return self._product

    @product.setter
    def product(self, value):
        self._product = value

    @property
    def quantity(self):
        return self._quantity

    @quantity.setter
    def quantity(self, value):
        if value > 0:
            self._quantity = value
        else:
            raise ValueError("Quantity must be a positive integer.")
```

Task 4: Composition:

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

• Orders Class with Composition:

```python
class Order:
    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

```

• OrderDetails Class with Composition:

```python
class OrderDetail:
    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
```

• Customers and Products Classes:

```python
class Customer:
    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

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
```

• Inventory Class:

```python
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
```

Task 5: Exceptions handling

• Data Validation:

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

```
class InvalidDataException(Exception):
    pass
```

```
def register_user(email):
    if "@" not in email:
        raise InvalidDataException("Invalid email address provided.")
```

• Inventory Management:

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

```
class InsufficientStockException(Exception):
    pass
```

```
def process_order(order, available_stock):
    if order.quantity > available_stock:
        raise InsufficientStockException("Insufficient stock for the order.")
```

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

```
class IncompleteOrderException(Exception):
    pass
```

```
def process_order_detail(order_detail):
    if order_detail.product is None:
        raise IncompleteOrderException("Order detail lacks a product reference.")
```

• Payment Processing: o 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.

```
class PaymentFailedException(Exception):
    pass
```

• File I/O (e.g., Logging):

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

```
class IOException(Exception):
    pass
```

```
def write_to_log(message):
    try:
        with open("text.txt", "a") as file:
            file.write(message + "\n")
    except IOError:
        raise IOException("Error occurred while writing to log.")
```

• Database Access:

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

```
class SqlException(Exception):
    pass
```

• Concurrency Control:

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

```
class ConcurrencyException(Exception):
    pass
```

• Security and Authentication:

o Challenge: Ensuring secure access and handling unauthorized access attempts. o 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.

```
class AuthenticationException(Exception):
    pass

class AuthorizationException(Exception):
    pass
```

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.

```python
import mysql.connector

class databaseconnction():
    def __init__(self,host,user,password,database):
        self.host = host
        self.user = user
        self.password = password
        self.database = database
        self.connection = None

    def connect(self):
        try:
            self.connection = mysql.connector.connect(
            host = self.host,
            user = self.user,
            password = self.password,
            database = self.database
            )
            print(f"Connected to {self.database} database")
        except:
            print("Could not connect to database")

    def disconnect(self):
        if self.connection:
            self.connection.close()
            print("Disconnected")
```

```python
if __name__ == "__main__":
    a = databaseconnction( host: "localhost", user: 'root', password: '[redacted]', database: "techshop")
    a.connect()
    a.executeQuery("show tables")
    a.disconnect()
```

if __name__ == "__main__"

un demodata ×

```
C:\Users\prash\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\prash\PycharmProjects\p
Connected to techshop database
('customers',)
('inventory',)
('orderdetails',)
('orders',)
('products',)
Disconnected
```

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.

```
42      def insertValue(self,query,value):
43          try:
44              cursor = self.connection.cursor()
45              cursor.execute(query,value)
46              cursor.close()
47              self.connection.commit()
48          except:
49              print("Error")
50
```

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.

```
50
51      def updateQuery(self,query):
52          try:
53              cursor = self.connection.cursor()
54              cursor.execute(query)
55              self.connection.commit()
56              cursor.close()
57          except:
58              print("Error")
59
60
```

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.

```
28      def executeQuery(self,query,value):
29          cursor = self.connection.cursor()
30          cursor.execute(query,value)
31          result = cursor.fetchall()
32          cursor.close()
33          for i in result:
34              print(i)
```

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.

```
42      def insertValue(self,query,value):
43          try:
44              cursor = self.connection.cursor()
45              cursor.execute(query,value)
46              cursor.close()
47              self.connection.commit()
48          except:
49              print("Error")
50
```

```
50
51      def updateQuery(self,query):
52          try:
53              cursor = self.connection.cursor()
54              cursor.execute(query)
55              self.connection.commit()
56              cursor.close()
57          except:
58              print("Error")
59
60
```

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.

```
28      def executeQuery(self,query,value):
29          cursor = self.connection.cursor()
30          cursor.execute(query,value)
31          result = cursor.fetchall()
32          cursor.close()
33          for i in result:
34              print(i)
```

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.

```
50
51      def updateQuery(self,query):
52          try:
53              cursor = self.connection.cursor()
54              cursor.execute(query)
55              self.connection.commit()
56              cursor.close()
57          except:
58              print("Error")
59
60
```

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