# IMPLEMENT OF OOPS

## 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.

**Customers Class:**

**Attributes:**

• CustomerID (int)

• FirstName (string)

• LastName (string)

• Email (string)

• Phone (string)

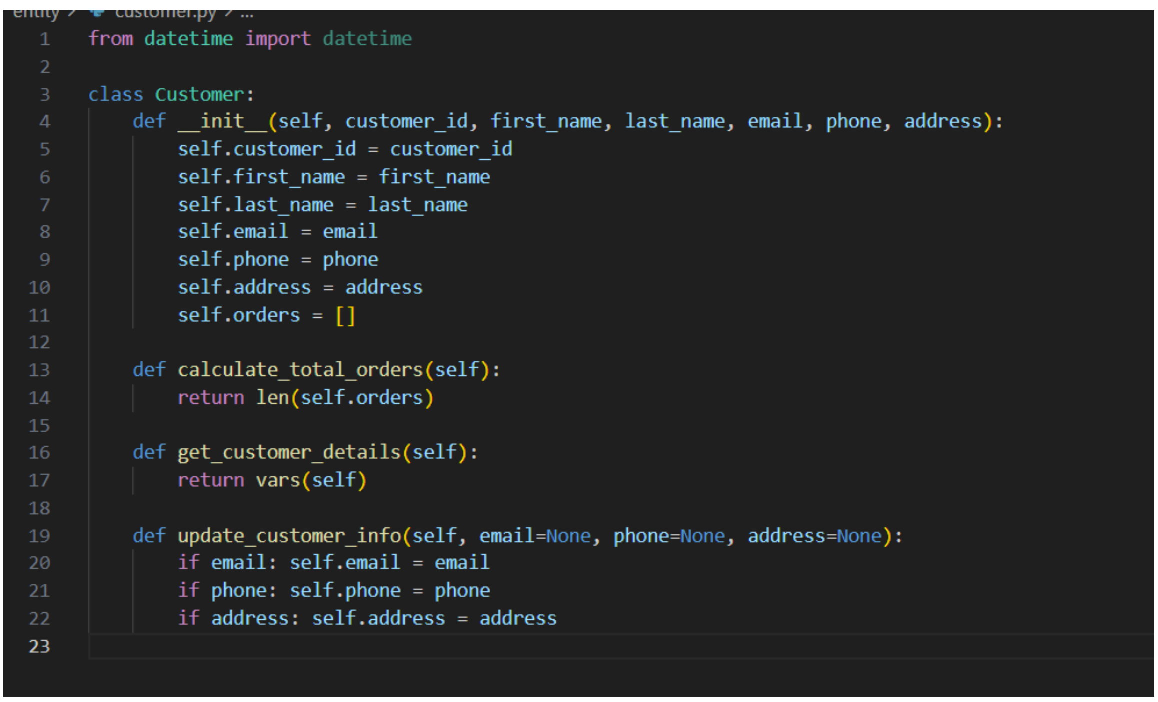
• Address (string)

**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).



```python
from datetime import datetime

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
        self.orders = []

    def calculate_total_orders(self):
        return len(self.orders)

    def get_customer_details(self):
        return vars(self)

    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)

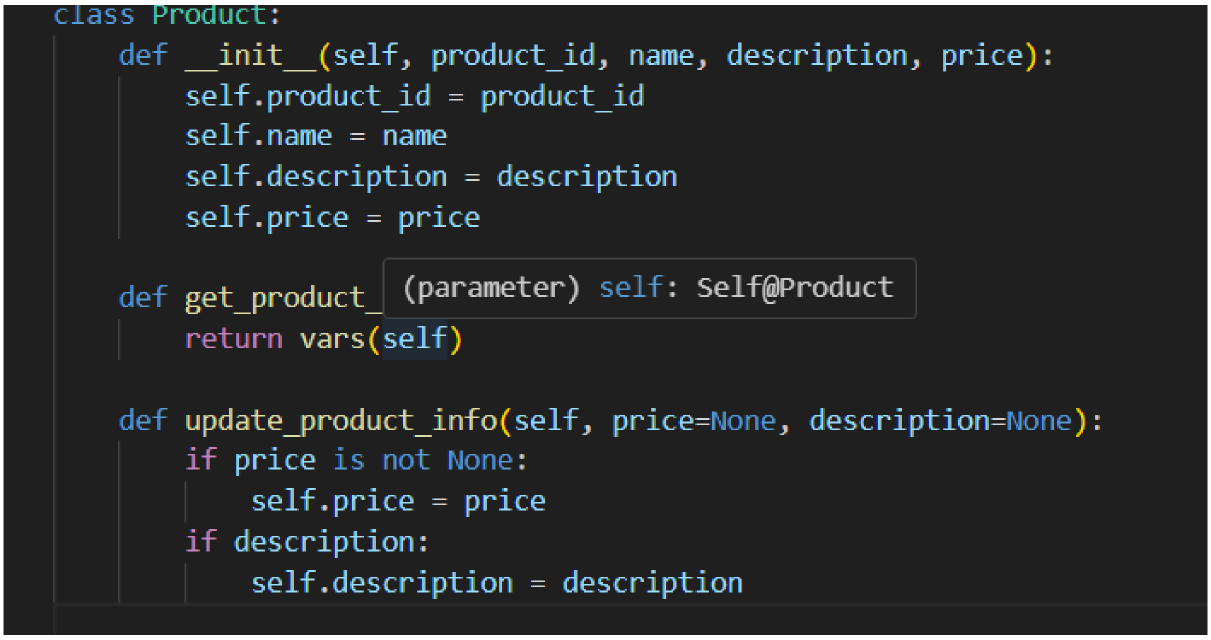
• Price (decimal)

**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.



```
class Product:
    def __init__(self, product_id, name, description, price):
        self.product_id = product_id
        self.name = name
        self.description = description
        self.price = price

    def get_product_ (parameter) self: Self@Product
        return vars(self)

    def update_product_info(self, price=None, description=None):
        if price is not None:
            self.price = price
        if description:
            self.description = description
```

**Orders Class:**

**Attributes:**

• OrderID (int)

• Customer (Customer) - Use composition to reference the Customer who placed the order.

• OrderDate (DateTime)

• TotalAmount (decimal)

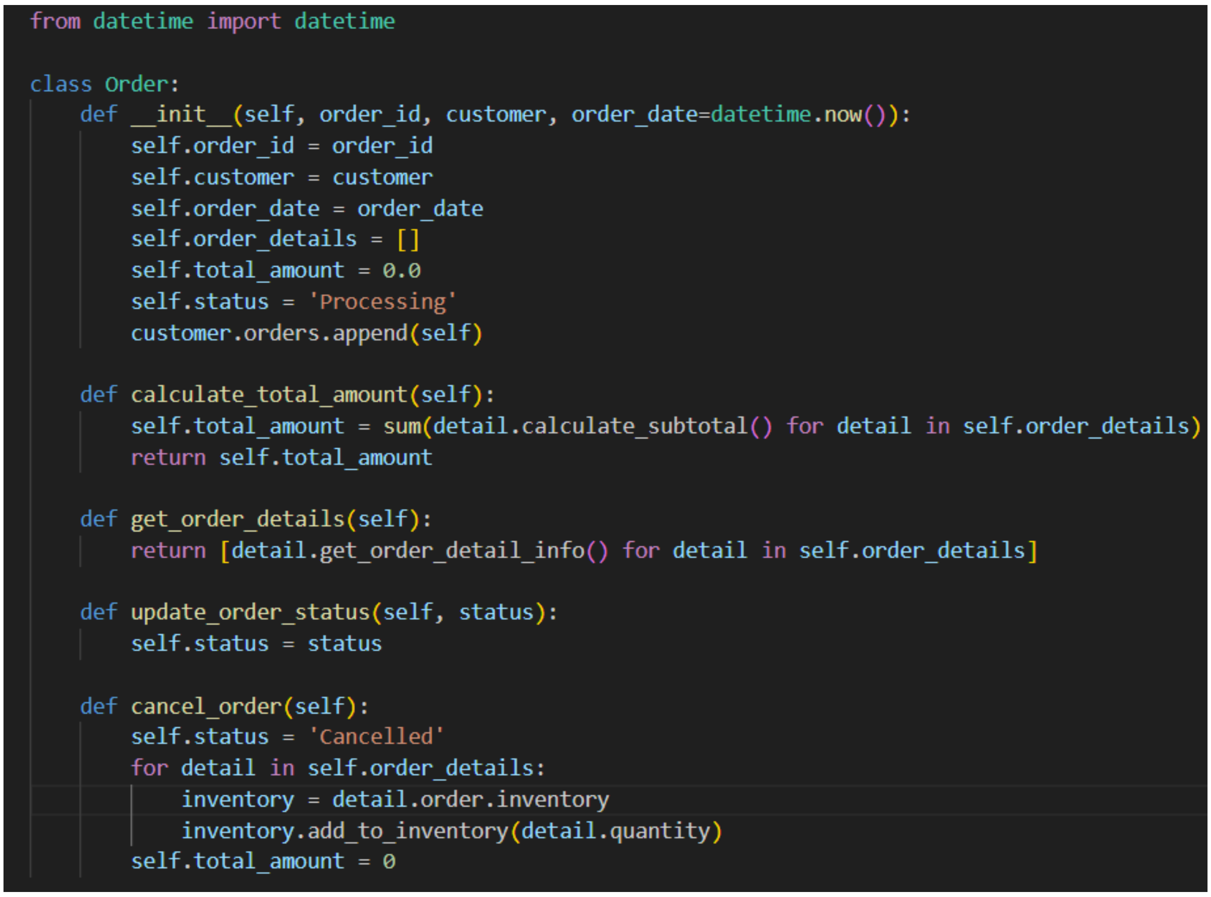
**Methods:**

• 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
from datetime import datetime

class Order:
    def __init__(self, order_id, customer, order_date=datetime.now()):
        self.order_id = order_id
        self.customer = customer
        self.order_date = order_date
        self.order_details = []
        self.total_amount = 0.0
        self.status = 'Processing'
        customer.orders.append(self)

    def calculate_total_amount(self):
        self.total_amount = sum(detail.calculate_subtotal() for detail in self.order_details)
        return self.total_amount

    def get_order_details(self):
        return [detail.get_order_detail_info() for detail in self.order_details]

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

    def cancel_order(self):
        self.status = 'Cancelled'
        for detail in self.order_details:
            inventory = detail.order.inventory
            inventory.add_to_inventory(detail.quantity)
        self.total_amount = 0
```

**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)

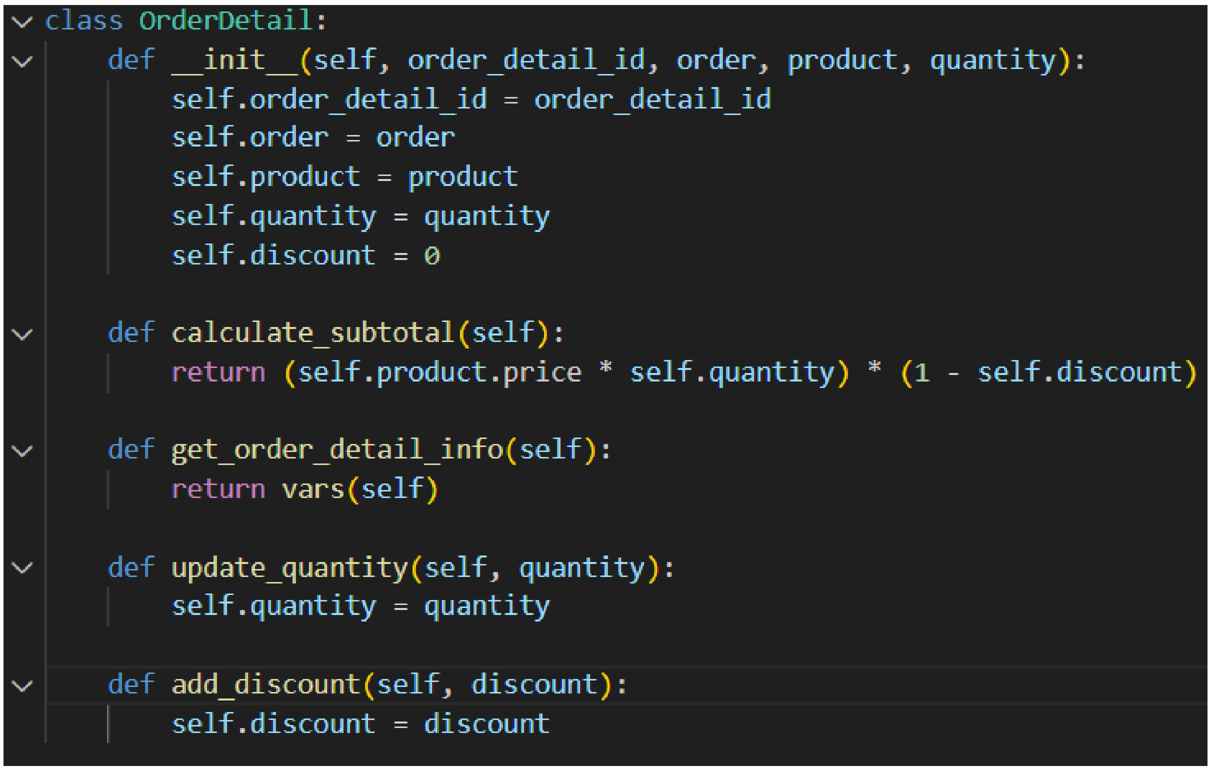
**Methods:**

• 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.



```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
        self.discount = 0

    def calculate_subtotal(self):
        return (self.product.price * self.quantity) * (1 - self.discount)

    def get_order_detail_info(self):
        return vars(self)

    def update_quantity(self, quantity):
        self.quantity = quantity

    def add_discount(self, discount):
        self.discount = discount
```

Inventory class: Attributes:

• InventoryID(int)

• Product (Composition): The product associated with the inventory item.

• QuantityInStock: The quantity of the product currently in stock.

• 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.

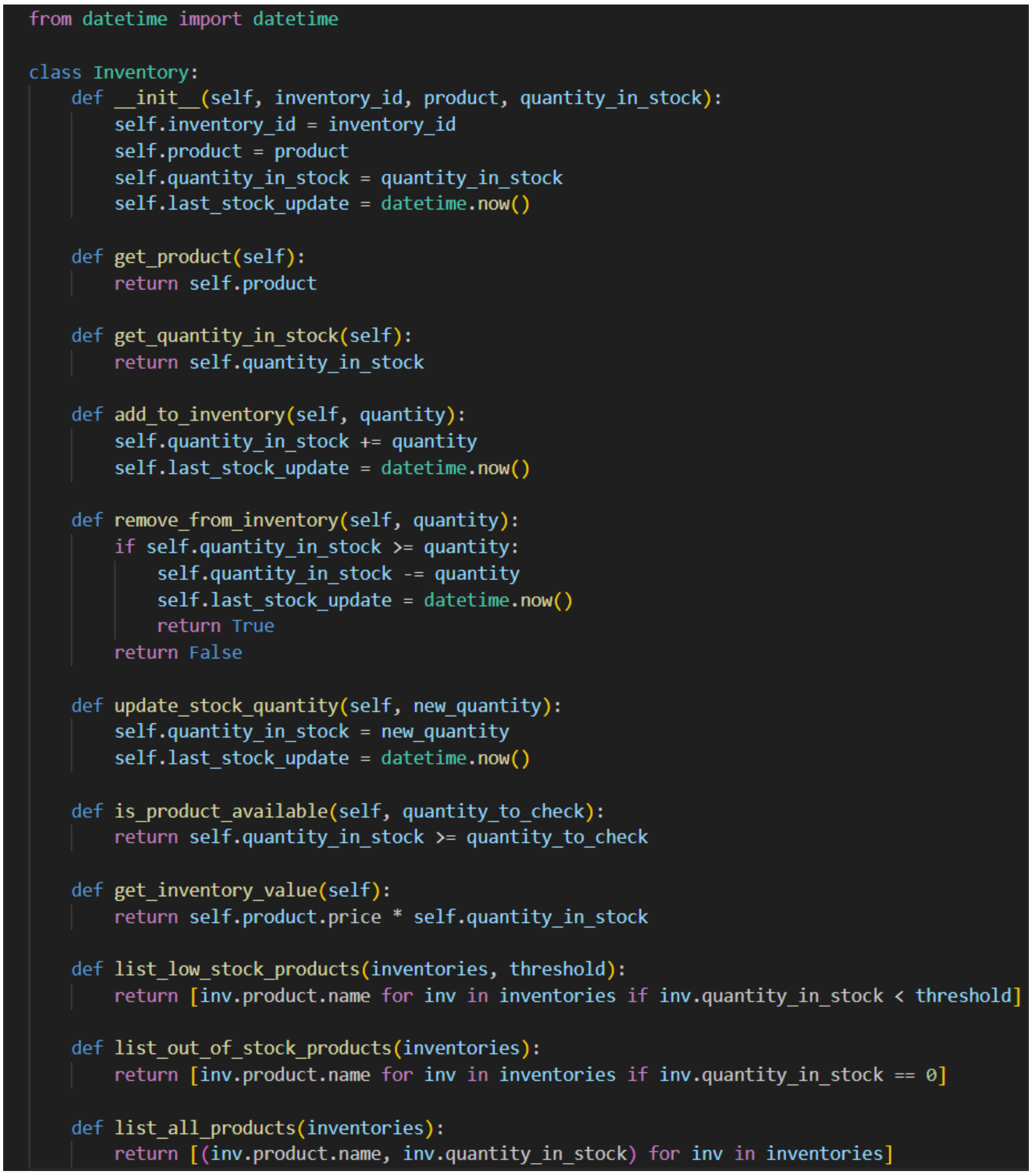
• IsProductAvailable(int quantityToCheck): A method to check if a specified quantity of the product is available in the inventory.

•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.



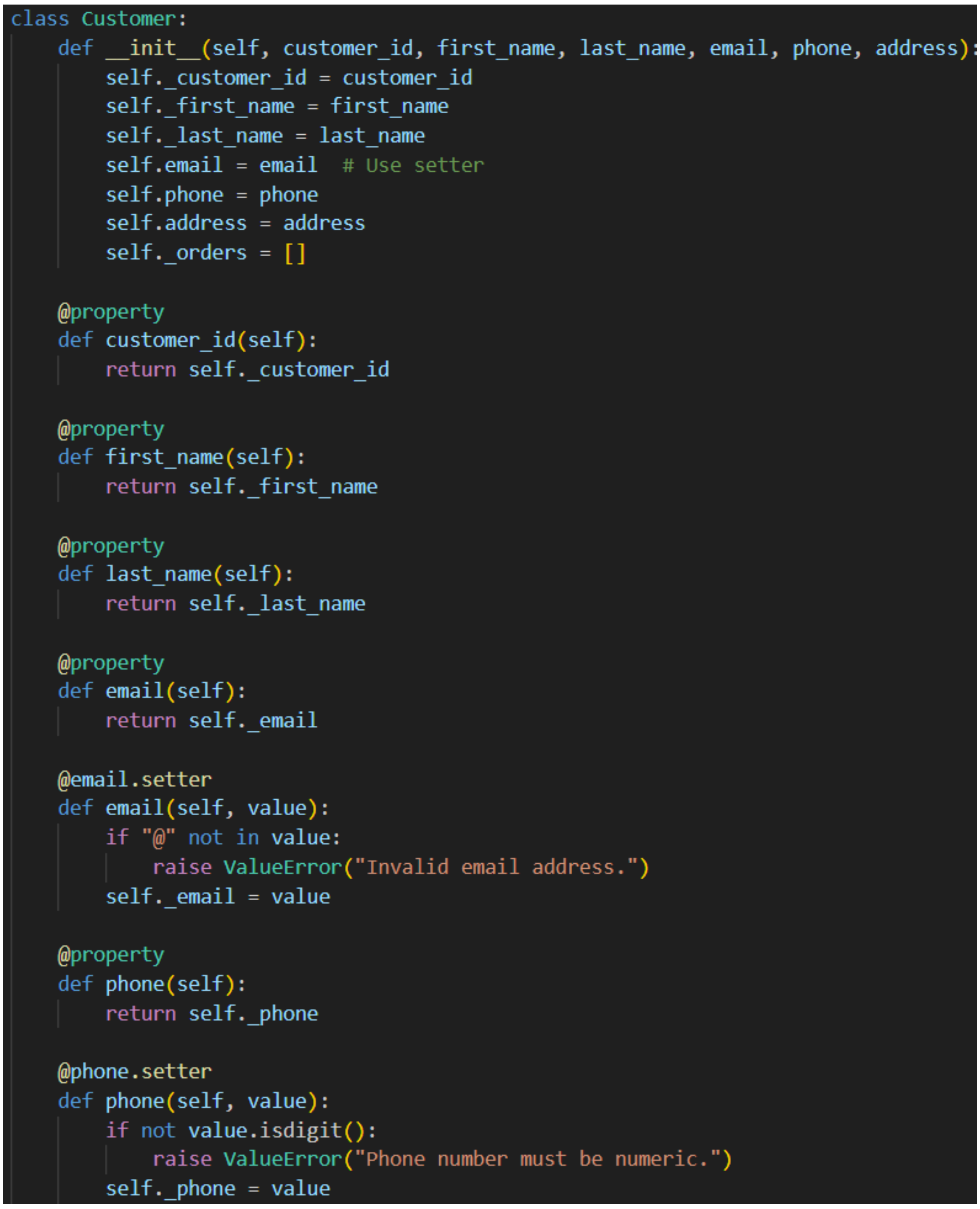
```python
from datetime import datetime

class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock):
        self.inventory_id = inventory_id
        self.product = product
        self.quantity_in_stock = quantity_in_stock
        self.last_stock_update = datetime.now()

    def get_product(self):
        return self.product

    def get_quantity_in_stock(self):
        return self.quantity_in_stock

    def add_to_inventory(self, quantity):
        self.quantity_in_stock += quantity
        self.last_stock_update = datetime.now()

    def remove_from_inventory(self, quantity):
        if self.quantity_in_stock >= quantity:
            self.quantity_in_stock -= quantity
            self.last_stock_update = datetime.now()
            return True
        return False

    def update_stock_quantity(self, new_quantity):
        self.quantity_in_stock = new_quantity
        self.last_stock_update = datetime.now()

    def is_product_available(self, quantity_to_check):
        return self.quantity_in_stock >= quantity_to_check

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

    def list_low_stock_products(inventories, threshold):
        return [inv.product.name for inv in inventories if inv.quantity_in_stock < threshold]

    def list_out_of_stock_products(inventories):
        return [inv.product.name for inv in inventories if inv.quantity_in_stock == 0]

    def list_all_products(inventories):
        return [(inv.product.name, inv.quantity_in_stock) for inv in inventories]
```

## Task 3: Encapsulation:

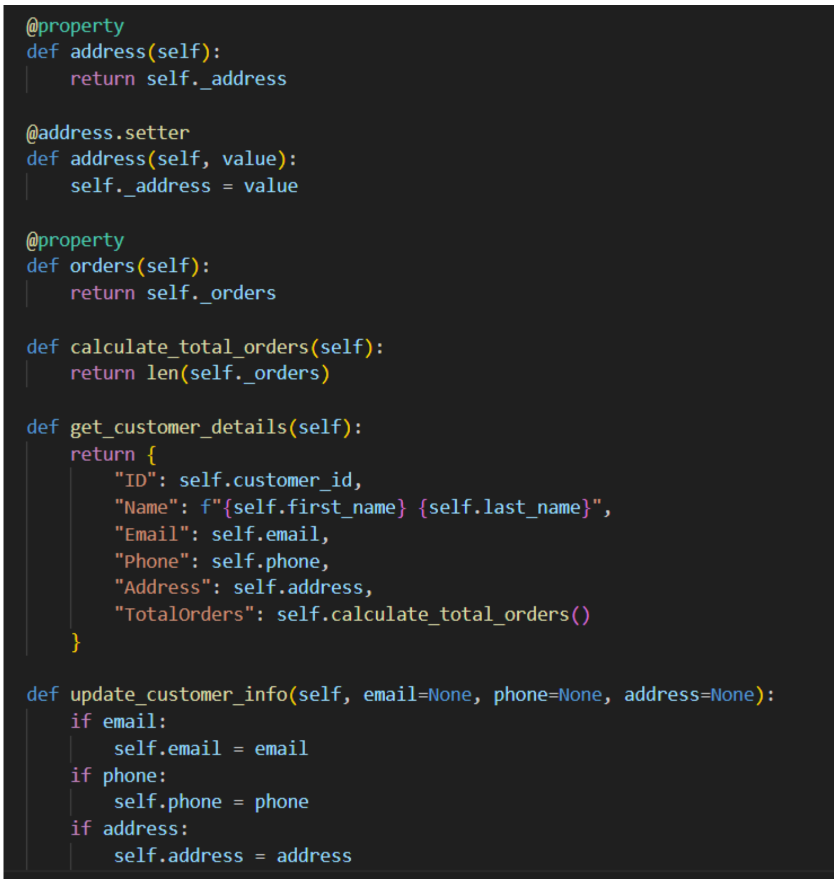
• Implement encapsulation by making the attributes private and providing public properties (getters and setters) for each attribute.

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

**CUSTOMER:**

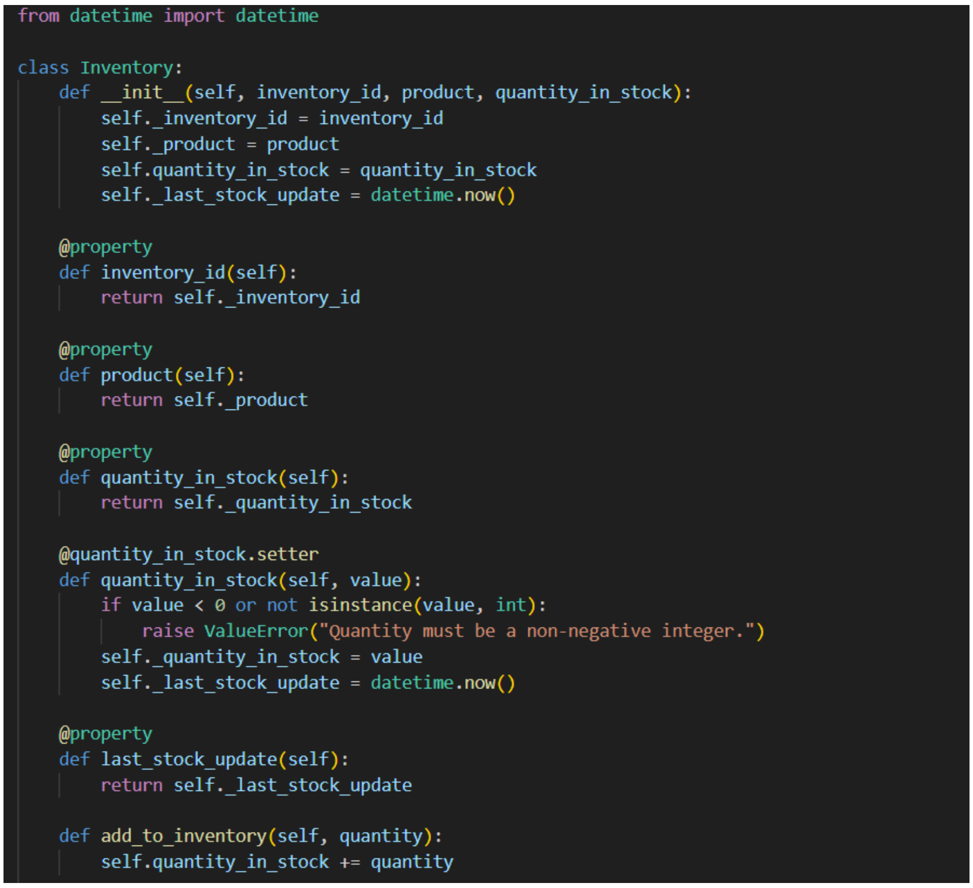


```python
class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        self._customer_id = customer_id
        self._first_name = first_name
        self._last_name = last_name
        self.email = email  # Use setter
        self.phone = phone
        self.address = address
        self._orders = []

    @property
    def customer_id(self):
        return self._customer_id

    @property
    def first_name(self):
        return self._first_name

    @property
    def last_name(self):
        return self._last_name

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

    @email.setter
    def email(self, value):
        if "@" not in value:
            raise ValueError("Invalid email address.")
        self._email = value

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

    @phone.setter
    def phone(self, value):
        if not value.isdigit():
            raise ValueError("Phone number must be numeric.")
        self._phone = value
```

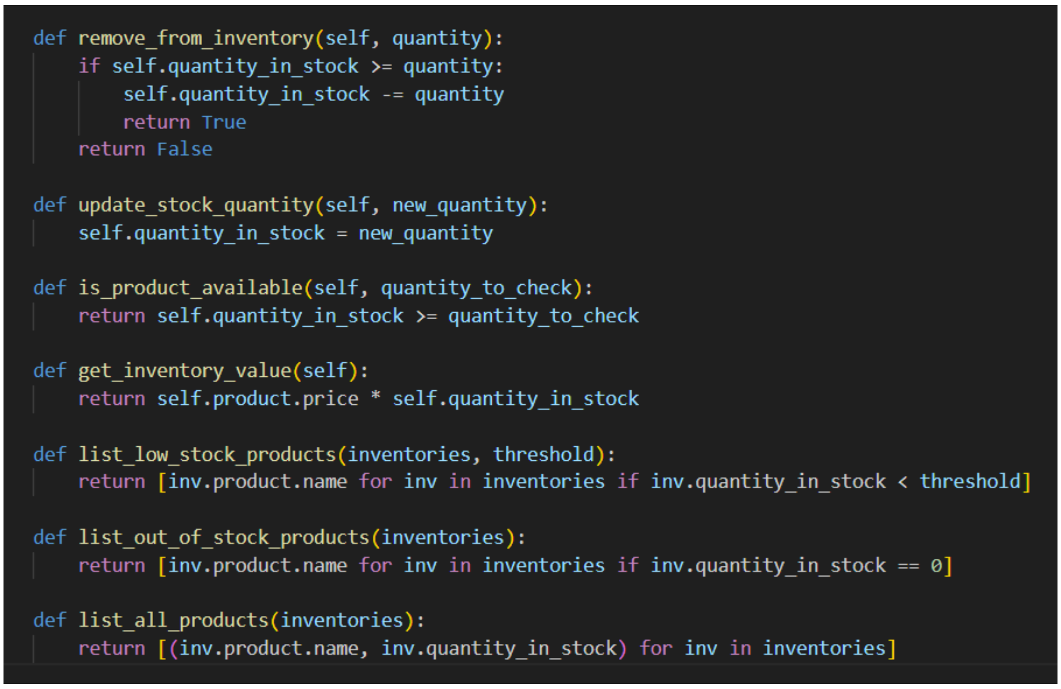


```python
    @property
    def address(self):
        return self._address

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

    @property
    def orders(self):
        return self._orders

    def calculate_total_orders(self):
        return len(self._orders)

    def get_customer_details(self):
        return {
            "ID": self.customer_id,
            "Name": f"{self.first_name} {self.last_name}",
            "Email": self.email,
            "Phone": self.phone,
            "Address": self.address,
            "TotalOrders": self.calculate_total_orders()
        }

    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
```

## INVENTORY:

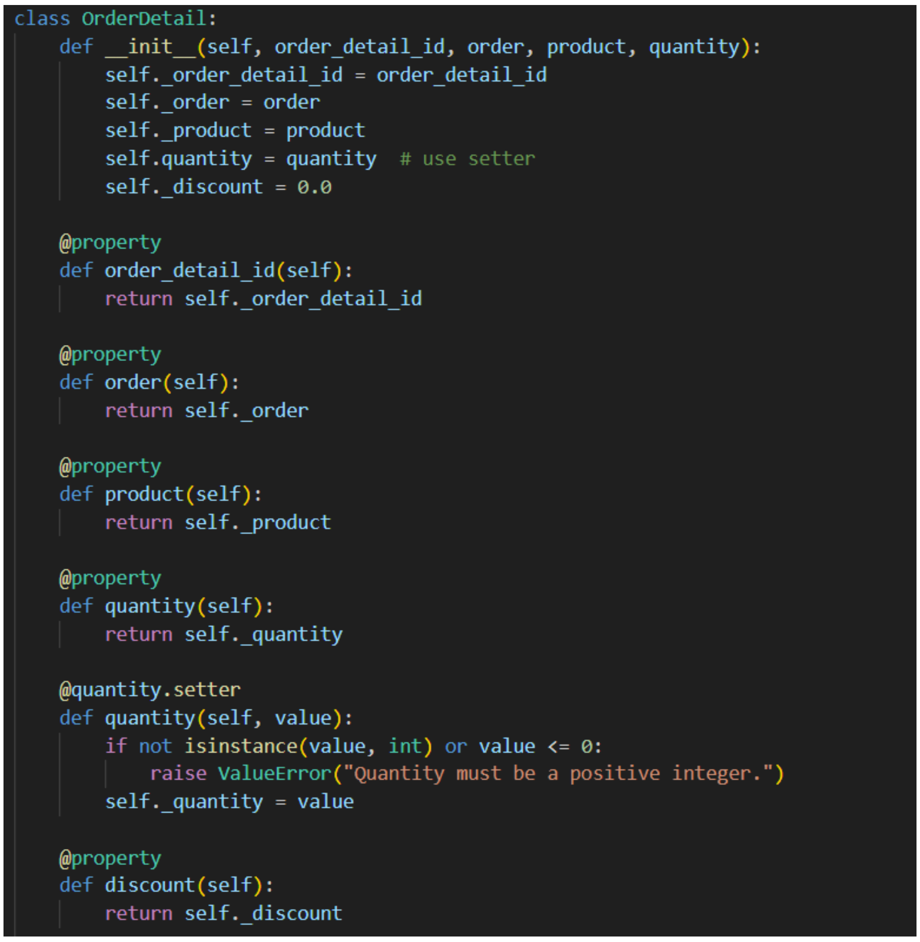


```python
from datetime import datetime

class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock):
        self._inventory_id = inventory_id
        self._product = product
        self.quantity_in_stock = quantity_in_stock
        self._last_stock_update = datetime.now()

    @property
    def inventory_id(self):
        return self._inventory_id

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

    @property
    def quantity_in_stock(self):
        return self._quantity_in_stock

    @quantity_in_stock.setter
    def quantity_in_stock(self, value):
        if value < 0 or not isinstance(value, int):
            raise ValueError("Quantity must be a non-negative integer.")
        self._quantity_in_stock = value
        self._last_stock_update = datetime.now()

    @property
    def last_stock_update(self):
        return self._last_stock_update

    def add_to_inventory(self, quantity):
        self.quantity_in_stock += quantity
```

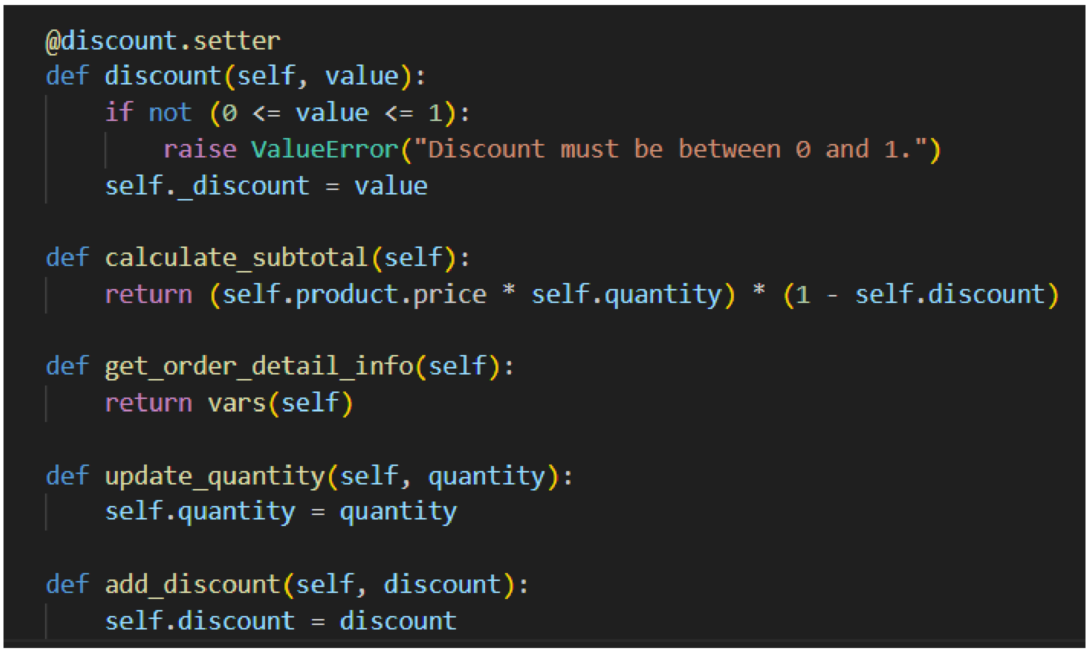


```python
def remove_from_inventory(self, quantity):
    if self.quantity_in_stock >= quantity:
        self.quantity_in_stock -= quantity
        return True
    return False

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

def is_product_available(self, quantity_to_check):
    return self.quantity_in_stock >= quantity_to_check

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

def list_low_stock_products(inventories, threshold):
    return [inv.product.name for inv in inventories if inv.quantity_in_stock < threshold]

def list_out_of_stock_products(inventories):
    return [inv.product.name for inv in inventories if inv.quantity_in_stock == 0]

def list_all_products(inventories):
    return [(inv.product.name, inv.quantity_in_stock) for inv in inventories]
```

## ORDER DETAIL:

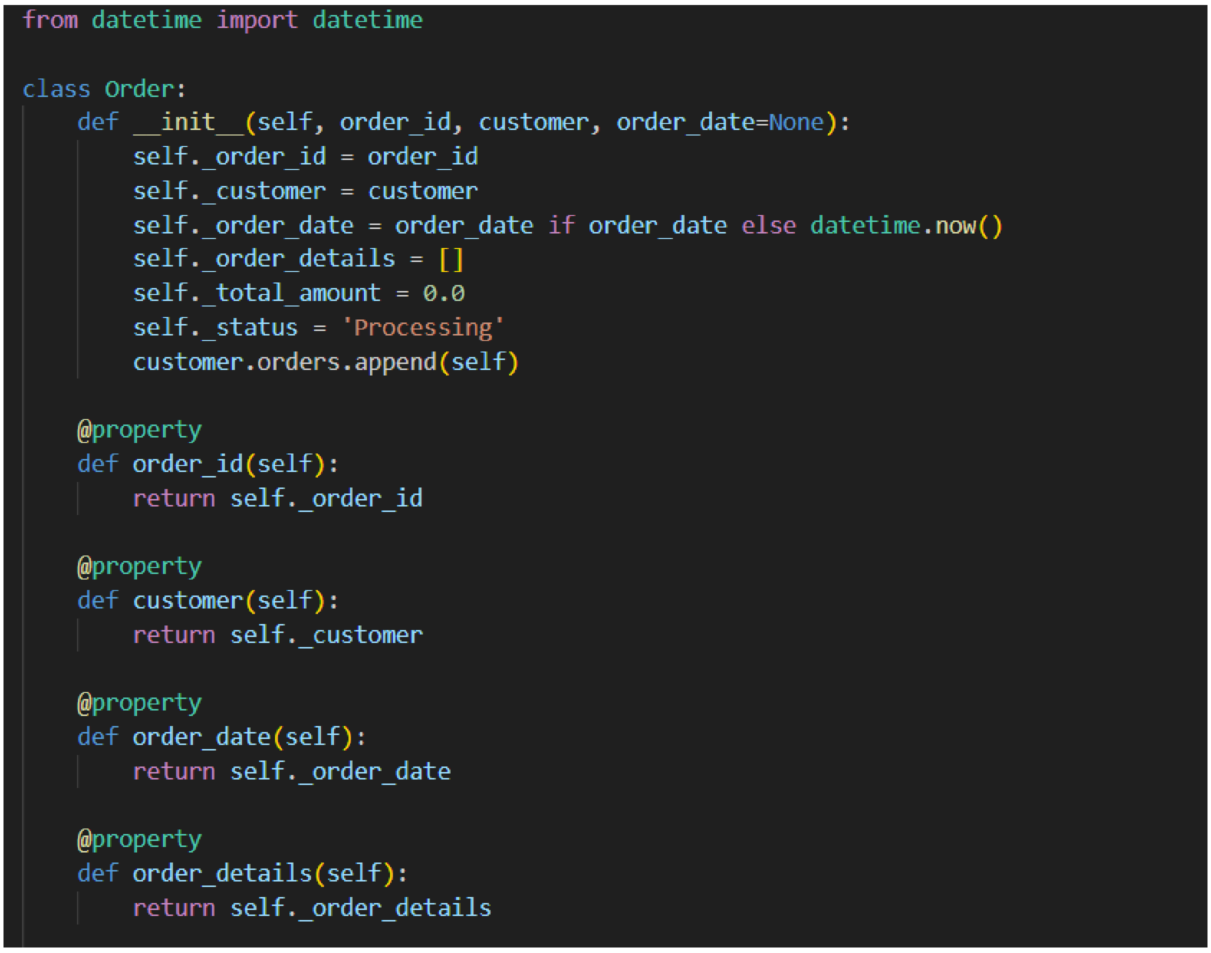


```python
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity):
        self._order_detail_id = order_detail_id
        self._order = order
        self._product = product
        self.quantity = quantity  # use setter
        self._discount = 0.0

    @property
    def order_detail_id(self):
        return self._order_detail_id

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

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

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

    @quantity.setter
    def quantity(self, value):
        if not isinstance(value, int) or value <= 0:
            raise ValueError("Quantity must be a positive integer.")
        self._quantity = value

    @property
    def discount(self):
        return self._discount
```

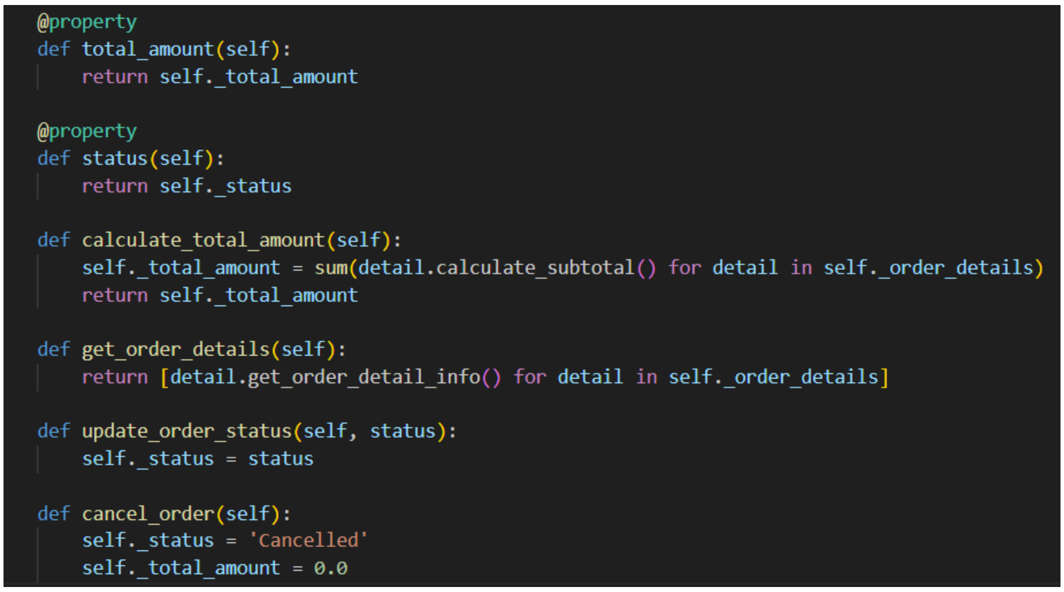


```python
    @discount.setter
    def discount(self, value):
        if not (0 <= value <= 1):
            raise ValueError("Discount must be between 0 and 1.")
        self._discount = value

    def calculate_subtotal(self):
        return (self.product.price * self.quantity) * (1 - self.discount)

    def get_order_detail_info(self):
        return vars(self)

    def update_quantity(self, quantity):
        self.quantity = quantity

    def add_discount(self, discount):
        self.discount = discount
```

**ORDER:**

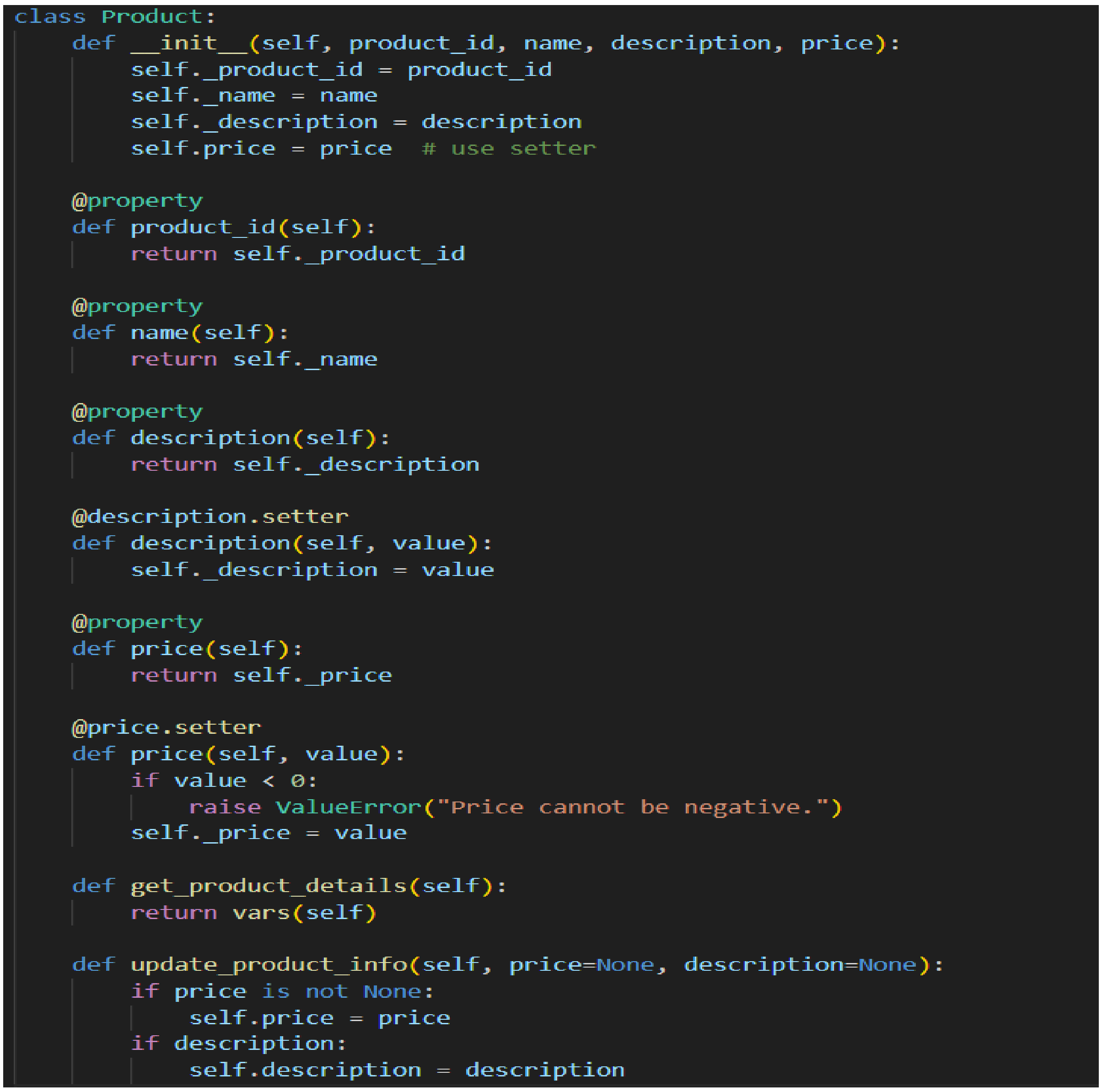


```python
from datetime import datetime

class Order:
    def __init__(self, order_id, customer, order_date=None):
        self._order_id = order_id
        self._customer = customer
        self._order_date = order_date if order_date else datetime.now()
        self._order_details = []
        self._total_amount = 0.0
        self._status = 'Processing'
        customer.orders.append(self)

    @property
    def order_id(self):
        return self._order_id

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

    @property
    def order_date(self):
        return self._order_date

    @property
    def order_details(self):
        return self._order_details
```



```python
    @property
    def total_amount(self):
        return self._total_amount

    @property
    def status(self):
        return self._status

    def calculate_total_amount(self):
        self._total_amount = sum(detail.calculate_subtotal() for detail in self._order_details)
        return self._total_amount

    def get_order_details(self):
        return [detail.get_order_detail_info() for detail in self._order_details]

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

    def cancel_order(self):
        self._status = 'Cancelled'
        self._total_amount = 0.0
```

**PRODUCT:**



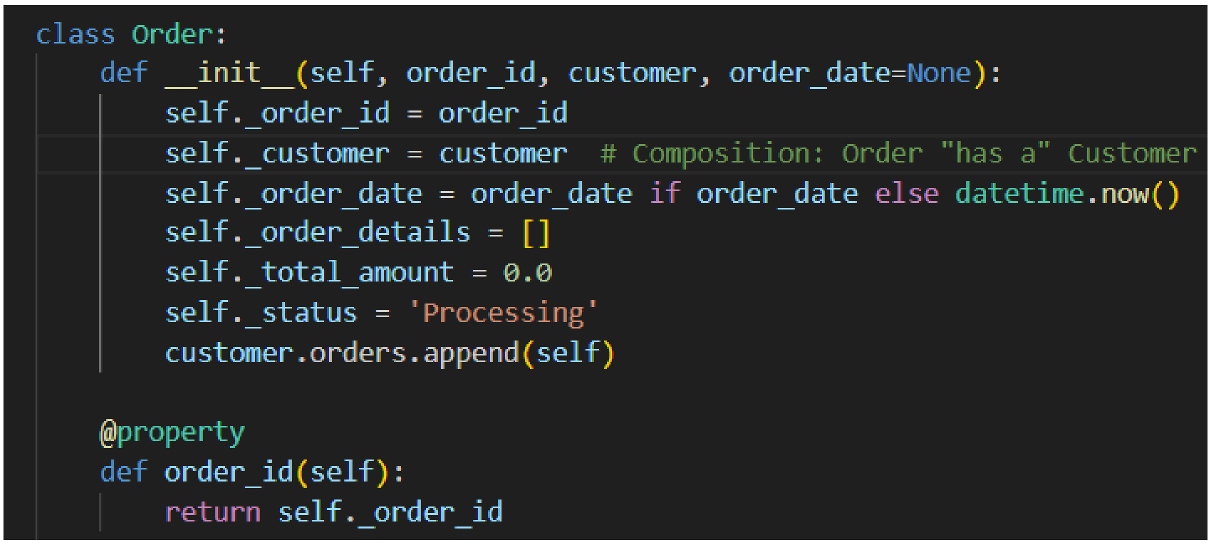
```python
class Product:
    def __init__(self, product_id, name, description, price):
        self._product_id = product_id
        self._name = name
        self._description = description
        self.price = price  # use setter

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

    @property
    def name(self):
        return self._name

    @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:
            raise ValueError("Price cannot be negative.")
        self._price = value

    def get_product_details(self):
        return vars(self)

    def update_product_info(self, price=None, description=None):
        if price is not None:
            self.price = price
        if description:
            self.description = description
```

## Task 4: Composition:

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

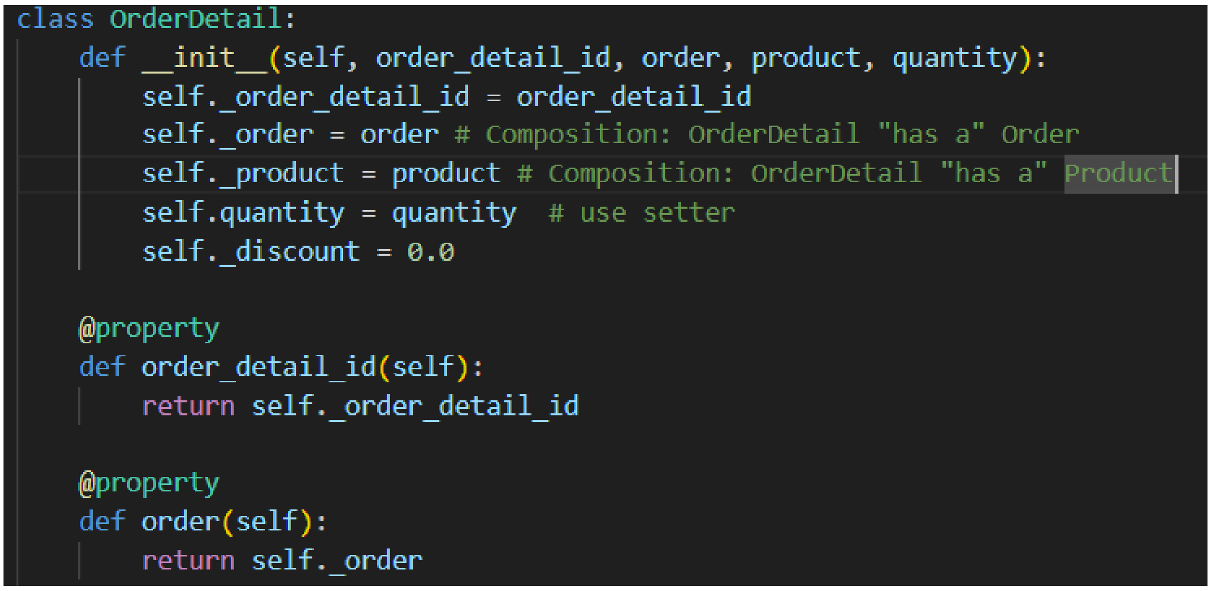
- Orders Class with Composition:
  - o In the Orders class, we want to establish a composition relationship with the Customers class, indicating that each order is associated with a specific customer.
  - o In the Orders class, we've added a private attribute customer of type Customers, establishing a composition relationship. The Customer property provides access to the Customers object associated with the order.
- OrderDetails Class with Composition:
  - o 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.
  - o 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.
- Customers and Products Classes:
  - o 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.
- Inventory Class:
  - o 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.

**Order Class – Composition with Customer:**



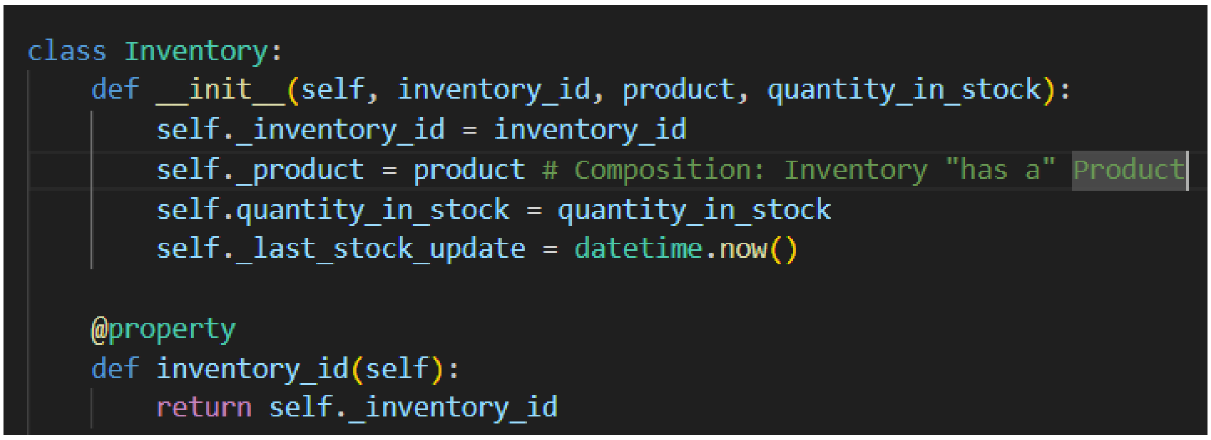
```python
class Order:
    def __init__(self, order_id, customer, order_date=None):
        self._order_id = order_id
        self._customer = customer  # Composition: Order "has a" Customer
        self._order_date = order_date if order_date else datetime.now()
        self._order_details = []
        self._total_amount = 0.0
        self._status = 'Processing'
        customer.orders.append(self)

    @property
    def order_id(self):
        return self._order_id
```

**OrderDetail Class – Composition with Order and Product:**



```python
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity):
        self._order_detail_id = order_detail_id
        self._order = order # Composition: OrderDetail "has a" Order
        self._product = product # Composition: OrderDetail "has a" Product
        self.quantity = quantity  # use setter
        self._discount = 0.0

    @property
    def order_detail_id(self):
        return self._order_detail_id

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

**Inventory Class – Composition with Product:**



```python
class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock):
        self._inventory_id = inventory_id
        self._product = product # Composition: Inventory "has a" Product
        self.quantity_in_stock = quantity_in_stock
        self._last_stock_update = datetime.now()

    @property
    def inventory_id(self):
        return self._inventory_id
```

## 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.

• **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.

• **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.

• **Payment Processing:**

o Challenge: Handling payment failures or declined transactions.

o Scenario: When processing a payment for an order and the payment is declined.

oException Handling: Handle payment-specific exceptions (e.g., PaymentFailedException) and initiate retry or cancellation processes.

• **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.

• **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.

• **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.

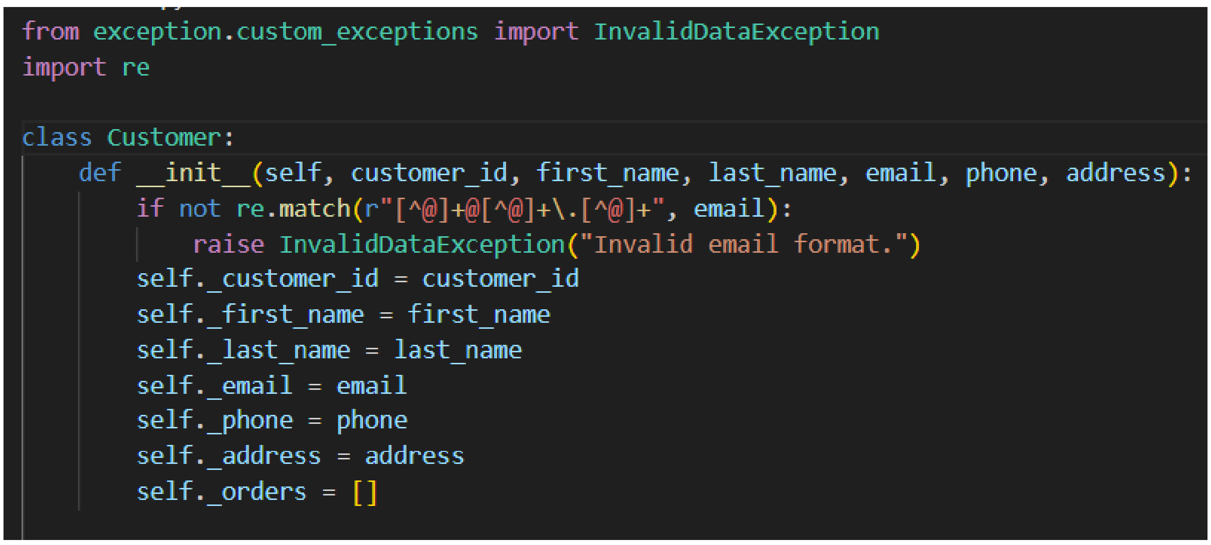
• **Security and Authentication:**

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

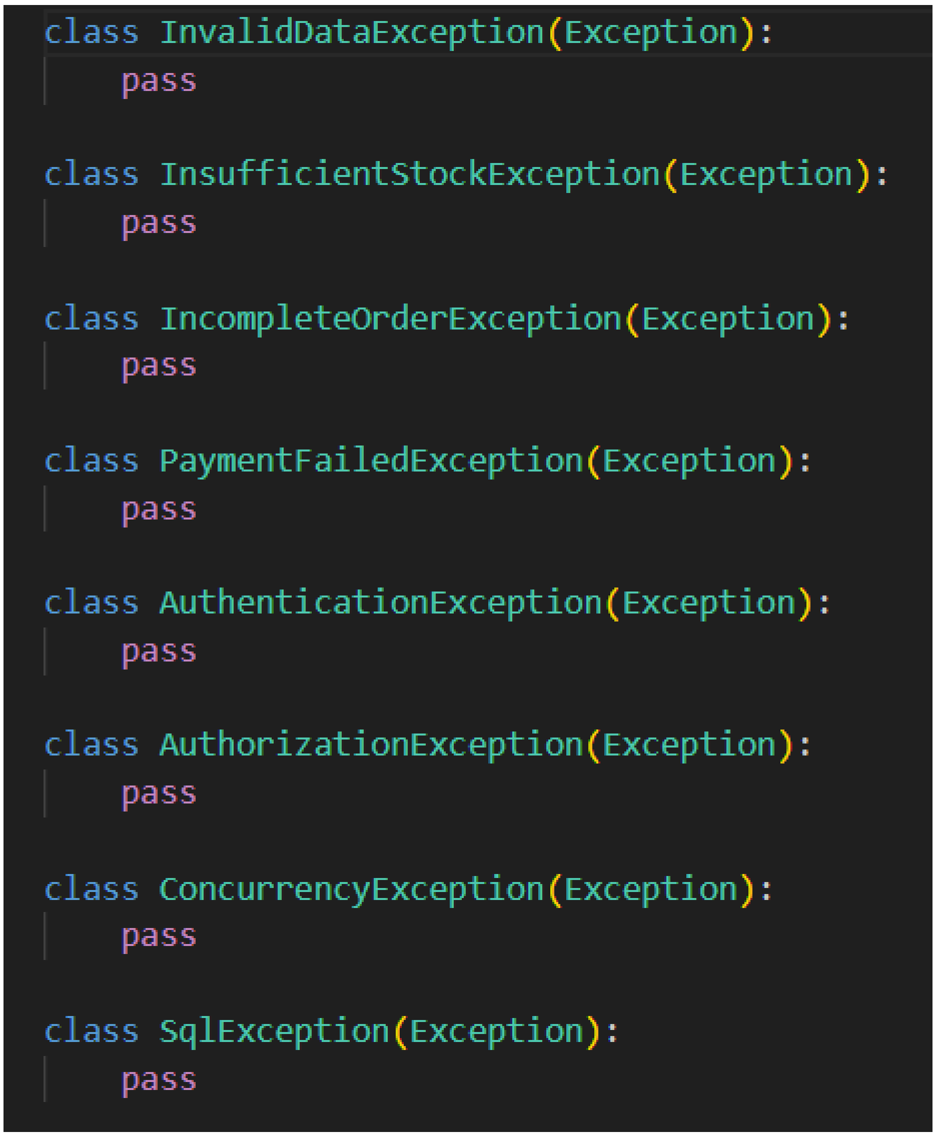
o Scenario: When a user tries to access sensitive information without proper authentication.

oException Handling: Implement custom AuthenticationException and AuthorizationException to handle security-related issues.

DATA VALIDATION:

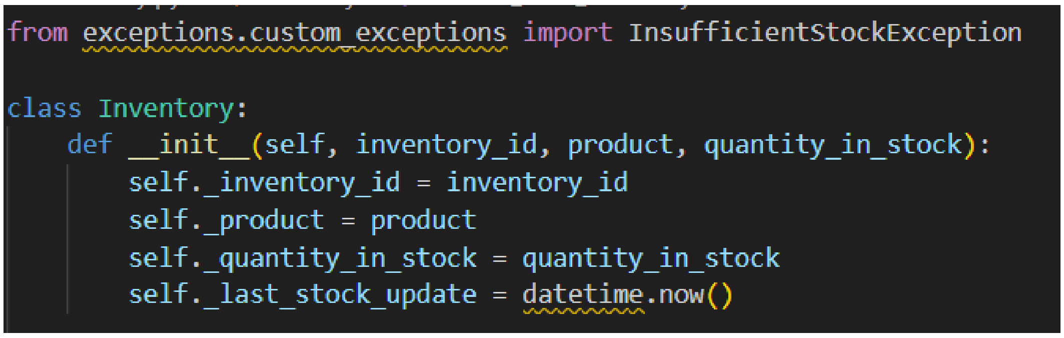


```python
from exception.custom_exceptions import InvalidDataException
import re

class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        if not re.match(r"[^@]+@[^@]+\.[^@]+", email):
            raise InvalidDataException("Invalid email format.")
        self._customer_id = customer_id
        self._first_name = first_name
        self._last_name = last_name
        self._email = email
        self._phone = phone
        self._address = address
        self._orders = []
```

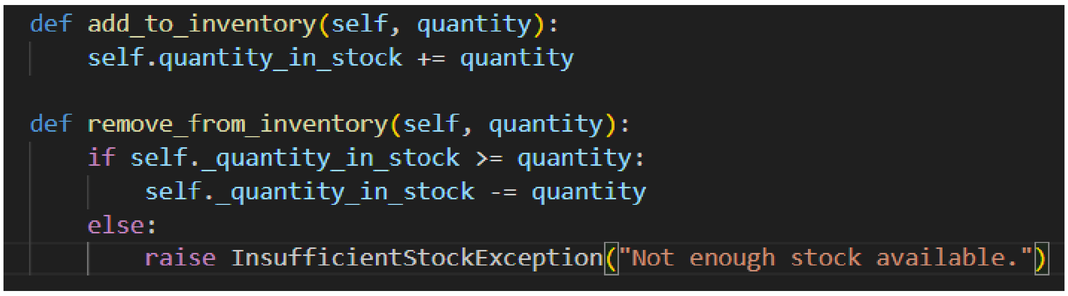


```python
class InvalidDataException(Exception):
    pass

class InsufficientStockException(Exception):
    pass

class IncompleteOrderException(Exception):
    pass

class PaymentFailedException(Exception):
    pass

class AuthenticationException(Exception):
    pass

class AuthorizationException(Exception):
    pass

class ConcurrencyException(Exception):
    pass

class SqlException(Exception):
    pass
```

INVENTORY MANAGEMENT:

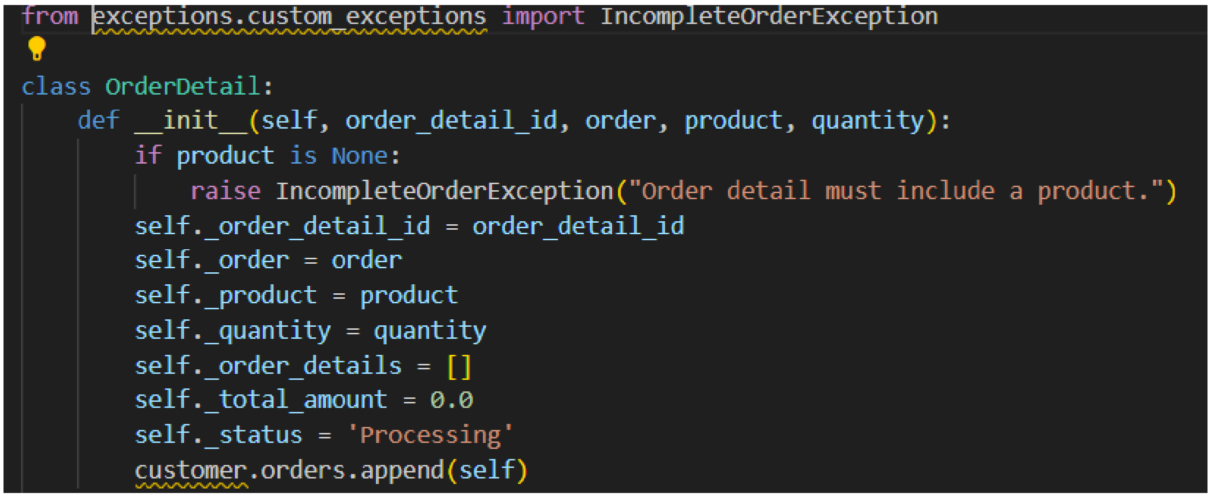


```python
from exceptions.custom_exceptions import InsufficientStockException

class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock):
        self._inventory_id = inventory_id
        self._product = product
        self._quantity_in_stock = quantity_in_stock
        self._last_stock_update = datetime.now()
```



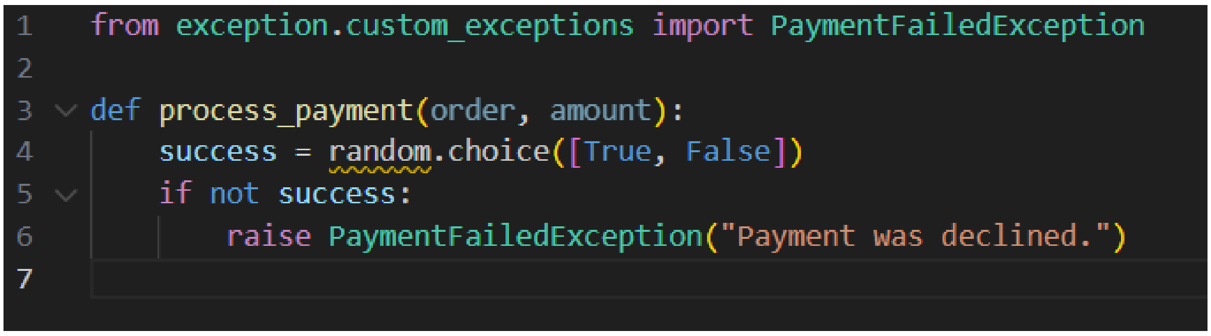
```python
def add_to_inventory(self, quantity):
    self.quantity_in_stock += quantity

def remove_from_inventory(self, quantity):
    if self._quantity_in_stock >= quantity:
        self._quantity_in_stock -= quantity
    else:
        raise InsufficientStockException("Not enough stock available.")
```

ORDER PROCESSING:



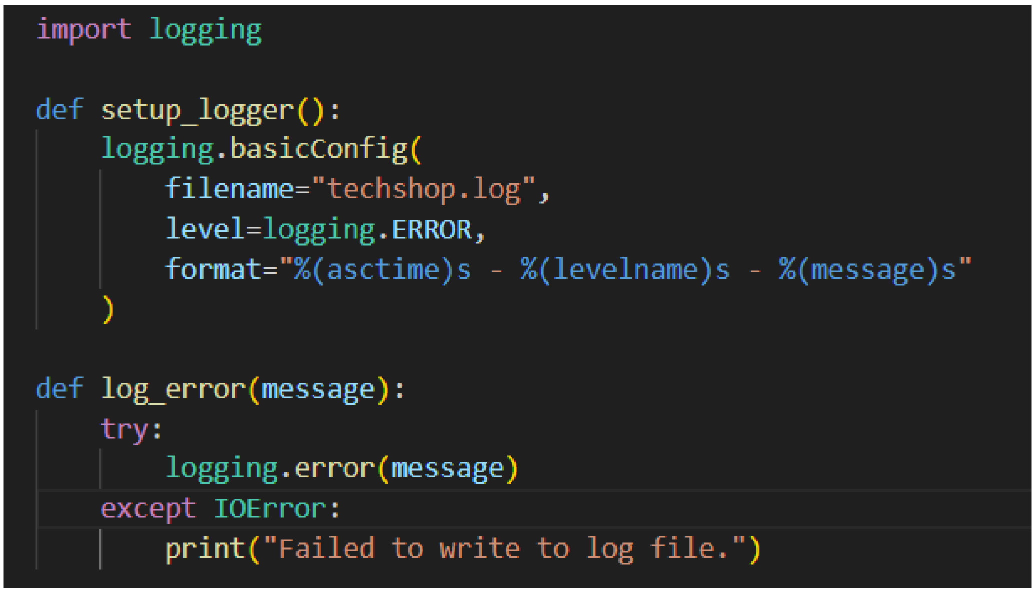
```python
from exceptions.custom_exceptions import IncompleteOrderException

class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity):
        if product is None:
            raise IncompleteOrderException("Order detail must include a product.")
        self._order_detail_id = order_detail_id
        self._order = order
        self._product = product
        self._quantity = quantity
        self._order_details = []
        self._total_amount = 0.0
        self._status = 'Processing'
        customer.orders.append(self)
```

PAYMENT PROCESSING:



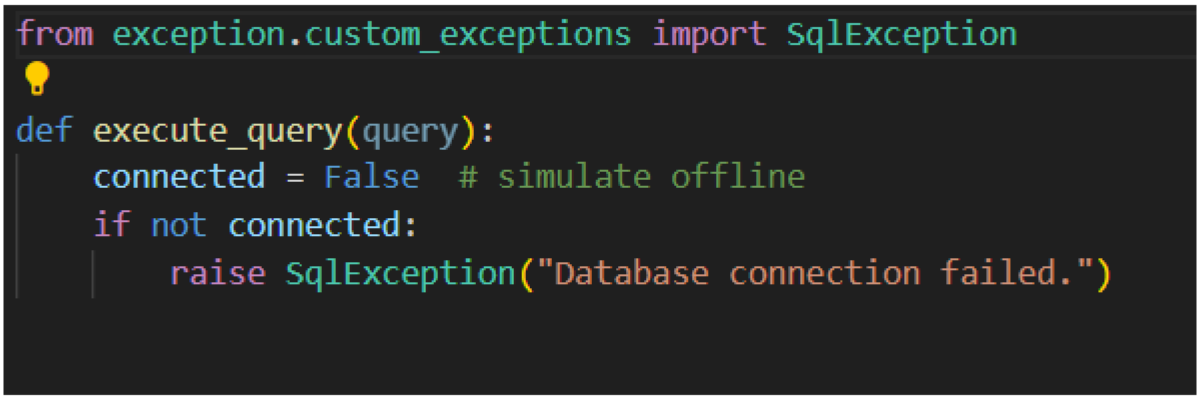
```python
from exception.custom_exceptions import PaymentFailedException

def process_payment(order, amount):
    success = random.choice([True, False])
    if not success:
        raise PaymentFailedException("Payment was declined.")

```

FILE I/O (E.g., LOGGING):



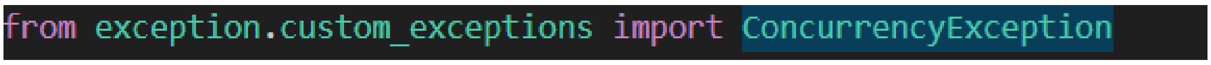
```python
import logging

def setup_logger():
    logging.basicConfig(
        filename="techshop.log",
        level=logging.ERROR,
        format="%(asctime)s - %(levelname)s - %(message)s"
    )

def log_error(message):
    try:
        logging.error(message)
    except IOError:
        print("Failed to write to log file.")
```

DATABASE ACCESS:

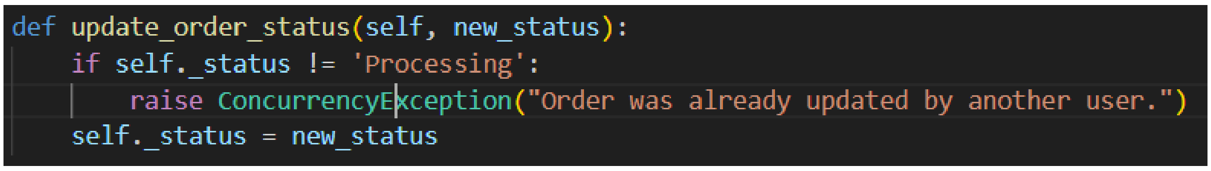


```python
from exception.custom_exceptions import SqlException

def execute_query(query):
    connected = False  # simulate offline
    if not connected:
        raise SqlException("Database connection failed.")
```

CONCURRENCY CONTROL:

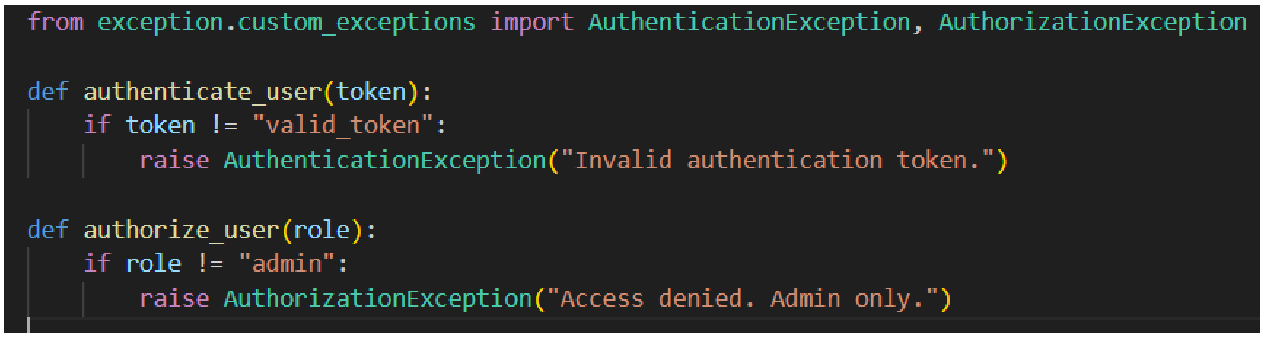


```python
from exception.custom_exceptions import ConcurrencyException
```



```python
def update_order_status(self, new_status):
    if self._status != 'Processing':
        raise ConcurrencyException("Order was already updated by another user.")
    self._status = new_status
```

SECURITY AND AUTHENTICATION:



```python
from exception.custom_exceptions import AuthenticationException, AuthorizationException

def authenticate_user(token):
    if token != "valid_token":
        raise AuthenticationException("Invalid authentication token.")

def authorize_user(role):
    if role != "admin":
        raise AuthorizationException("Access denied. Admin only.")
```

## Task 6: Collections

• **Managing Products List:**

o 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:**

o Challenge: Maintaining a list of customer orders (List).

o Scenario: Adding new orders, updating order statuses, and removing canceled orders.

oSolution: 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:**

o 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:**

o Challenge: Managing product inventory with a SortedList based on product IDs.

o Scenario: Tracking the quantity in stock for each product and quickly retrieving inventory information.

o Solution: Implement a SortedList where keys are product IDs. Ensure that inventory updates are synchronized with product additions and removals.

• **Handling Inventory Updates:**

o Challenge: Ensuring that inventory is updated correctly when processing orders.

o Scenario: Decrementing product quantities in stock when orders are placed.

o Solution: Implement a method to update inventory quantities when orders are processed. Handle exceptions for insufficient stock.

• **Product Search and Retrieval:**

o Challenge: Implementing a search functionality to find products based on various criteria (e.g., name, category).

o 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:**

o Challenge: Preventing duplicate products from being added to the list.

o Scenario: When a product with the same name or SKU is added.

o 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).

o Scenario: Recording and updating payment information for each order.

o Solution: Implement methods to record payments, update payment statuses, and handle payment errors. Ensure that payment records are consistent with order records.

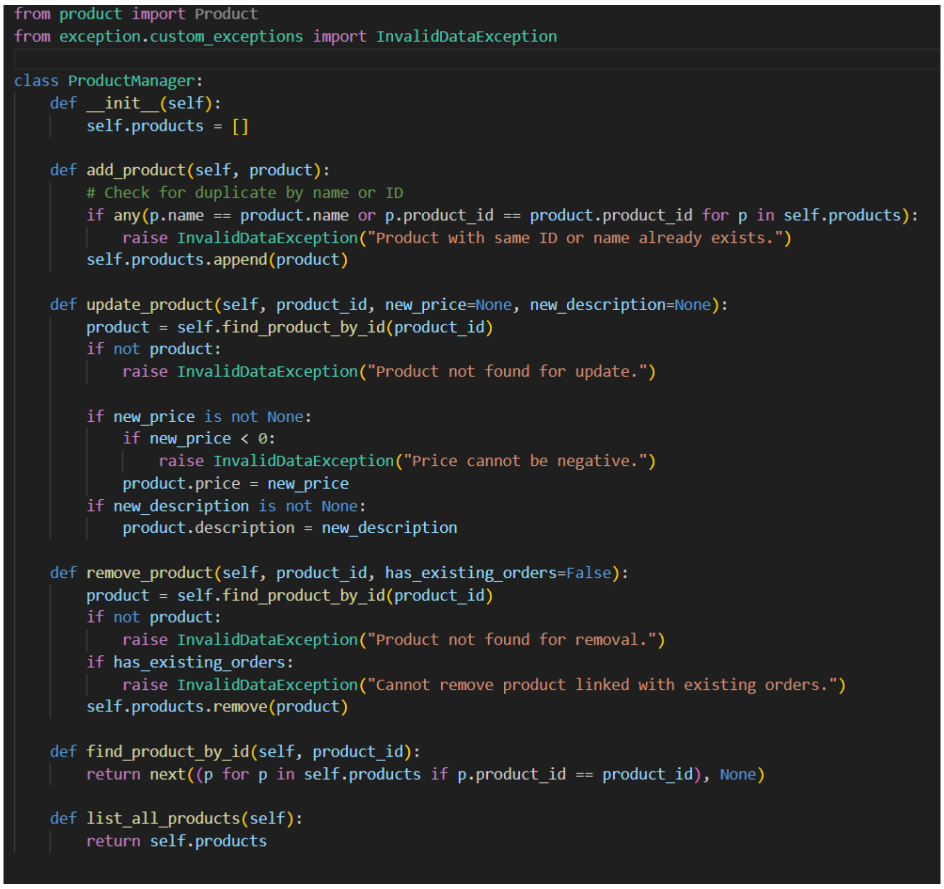
• **OrderDetails and Products Relationship:**

o Challenge: Managing the relationship between OrderDetails and Products.

o Scenario: Ensuring that order details accurately reflect the products available in the inventory.

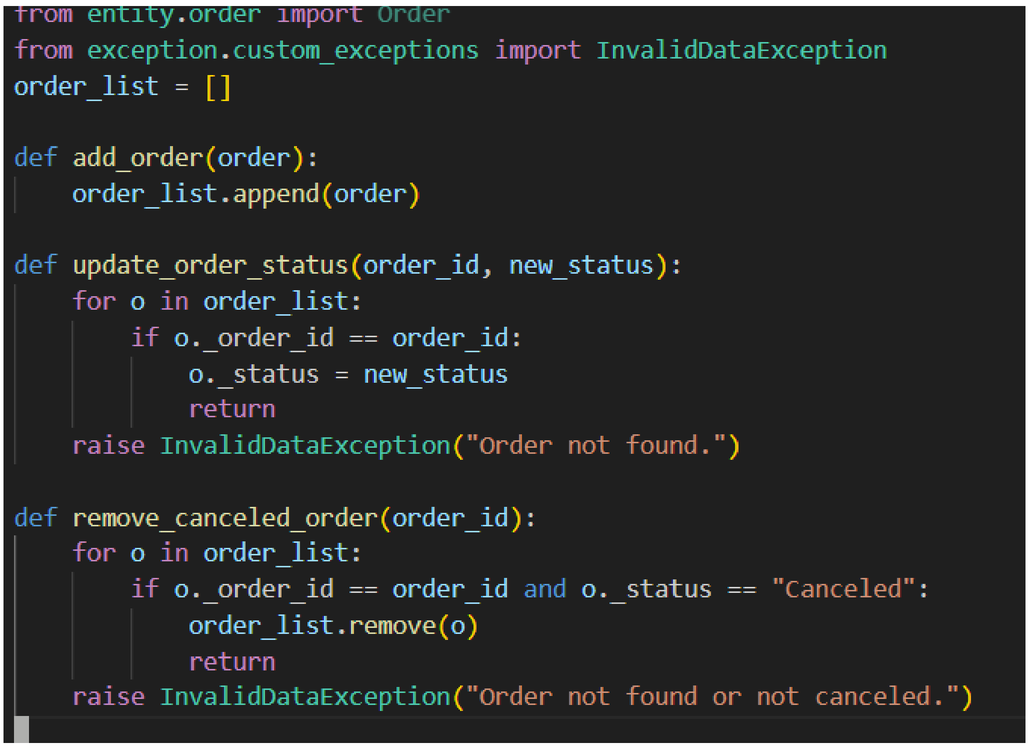
o Solution: Implement methods to validate product availability in the inventory before adding order details. Handle exceptions for unavailable products.

## Managing Products List:



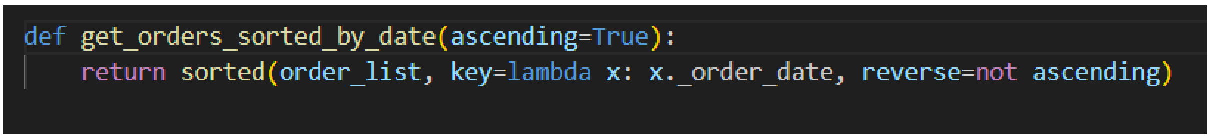
```python
from product import Product
from exception.custom_exceptions import InvalidDataException

class ProductManager:
    def __init__(self):
        self.products = []

    def add_product(self, product):
        # Check for duplicate by name or ID
        if any(p.name == product.name or p.product_id == product.product_id for p in self.products):
            raise InvalidDataException("Product with same ID or name already exists.")
        self.products.append(product)

    def update_product(self, product_id, new_price=None, new_description=None):
        product = self.find_product_by_id(product_id)
        if not product:
            raise InvalidDataException("Product not found for update.")

        if new_price is not None:
            if new_price < 0:
                raise InvalidDataException("Price cannot be negative.")
            product.price = new_price
        if new_description is not None:
            product.description = new_description

    def remove_product(self, product_id, has_existing_orders=False):
        product = self.find_product_by_id(product_id)
        if not product:
            raise InvalidDataException("Product not found for removal.")
        if has_existing_orders:
            raise InvalidDataException("Cannot remove product linked with existing orders.")
        self.products.remove(product)

    def find_product_by_id(self, product_id):
        return next((p for p in self.products if p.product_id == product_id), None)

    def list_all_products(self):
        return self.products
```

## Managing Orders List:



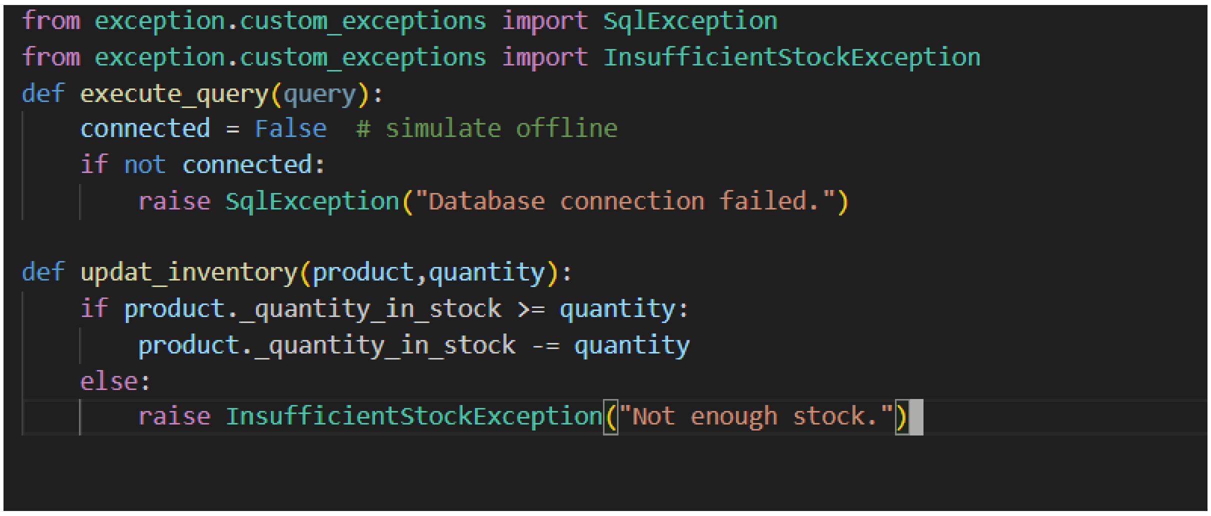
```python
from entity.order import Order
from exception.custom_exceptions import InvalidDataException
order_list = []

def add_order(order):
    order_list.append(order)

def update_order_status(order_id, new_status):
    for o in order_list:
        if o._order_id == order_id:
            o._status = new_status
            return
    raise InvalidDataException("Order not found.")

def remove_canceled_order(order_id):
    for o in order_list:
        if o._order_id == order_id and o._status == "Canceled":
            order_list.remove(o)
            return
    raise InvalidDataException("Order not found or not canceled.")
```

**Sorting Orders by Date:**



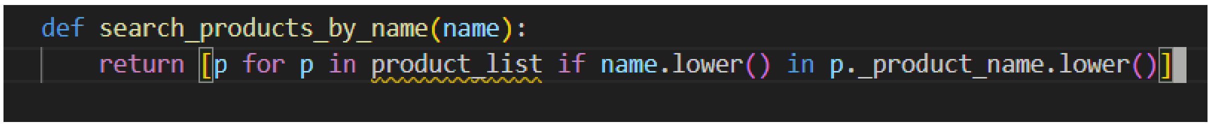
```python
def get_orders_sorted_by_date(ascending=True):
    return sorted(order_list, key=lambda x: x._order_date, reverse=not ascending)
```

**Handling Inventory Updates:**



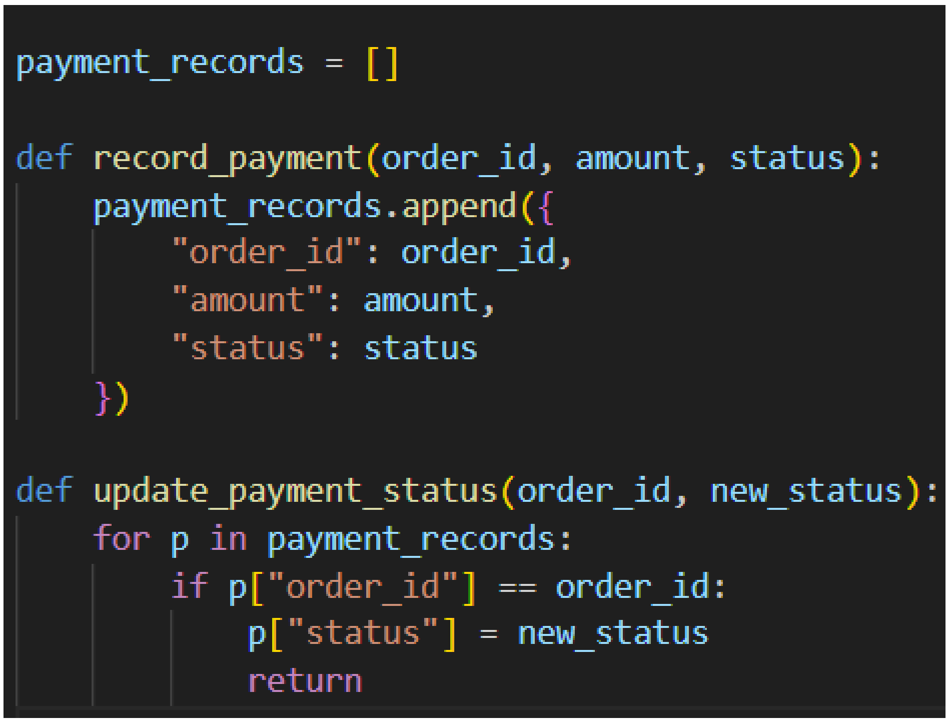
```python
from exception.custom_exceptions import SqlException
from exception.custom_exceptions import InsufficientStockException
def execute_query(query):
    connected = False  # simulate offline
    if not connected:
        raise SqlException("Database connection failed.")

def updat_inventory(product,quantity):
    if product._quantity_in_stock >= quantity:
        product._quantity_in_stock -= quantity
    else:
        raise InsufficientStockException("Not enough stock.")
```

**Product Search and Retrieval:**



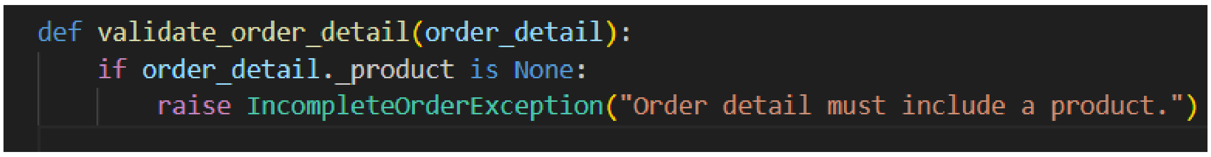
```python
def search_products_by_name(name):
    return [p for p in product_list if name.lower() in p._product_name.lower()]
```

**Payment Records List:**



```python
payment_records = []

def record_payment(order_id, amount, status):
    payment_records.append({
        "order_id": order_id,
        "amount": amount,
        "status": status
    })

def update_payment_status(order_id, new_status):
    for p in payment_records:
        if p["order_id"] == order_id:
            p["status"] = new_status
            return
```

**OrderDetails and Products Relationship:**



```python
def validate_order_detail(order_detail):
    if order_detail._product is None:
        raise IncompleteOrderException("Order detail must include a product.")
```

**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:**

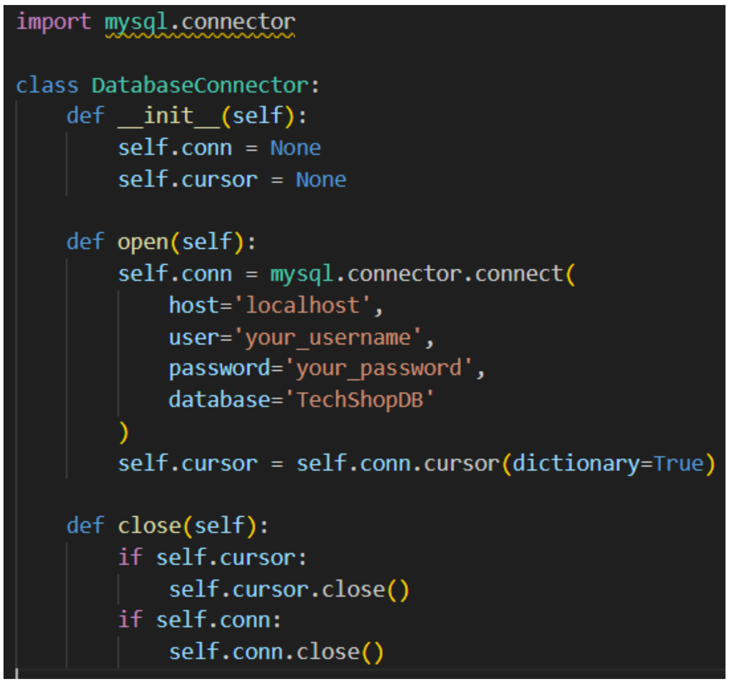
When customers make payments for their orders, the payment details (e.g., payment method, amount) must be recorded in the database.

Task: Develop a payment processing system that interacts with the database to record payment transactions, validate payment information, and handle errors.

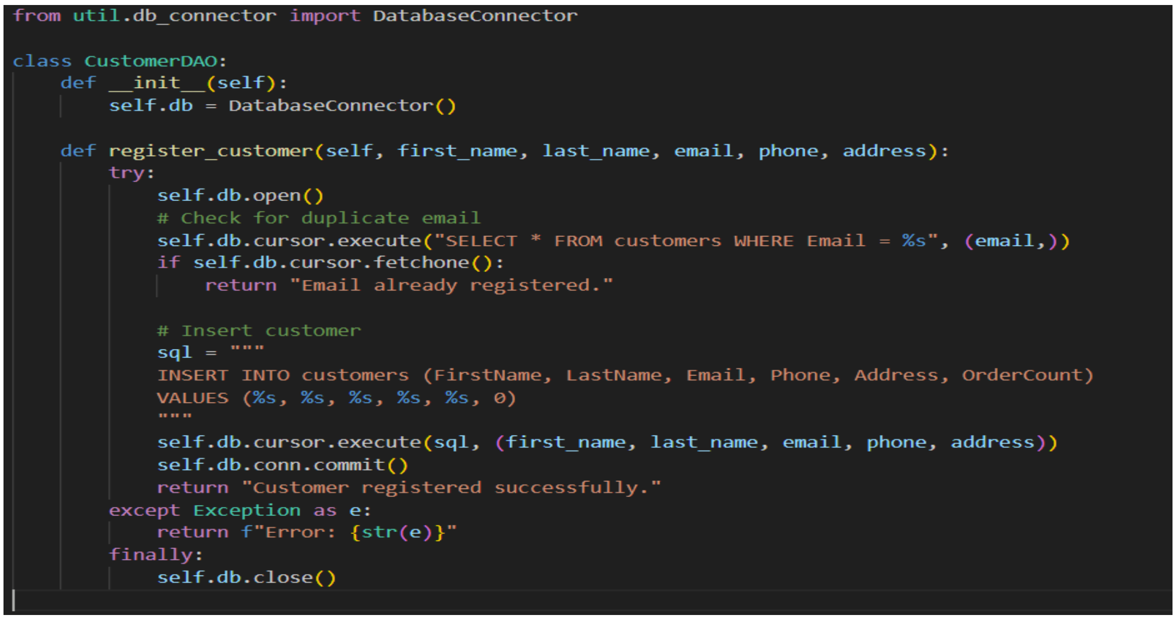
**9: Product Search and Recommendations Description:**

Customers should be able to search for products based on various criteria (e.g., name, category) and receive product recommendations.

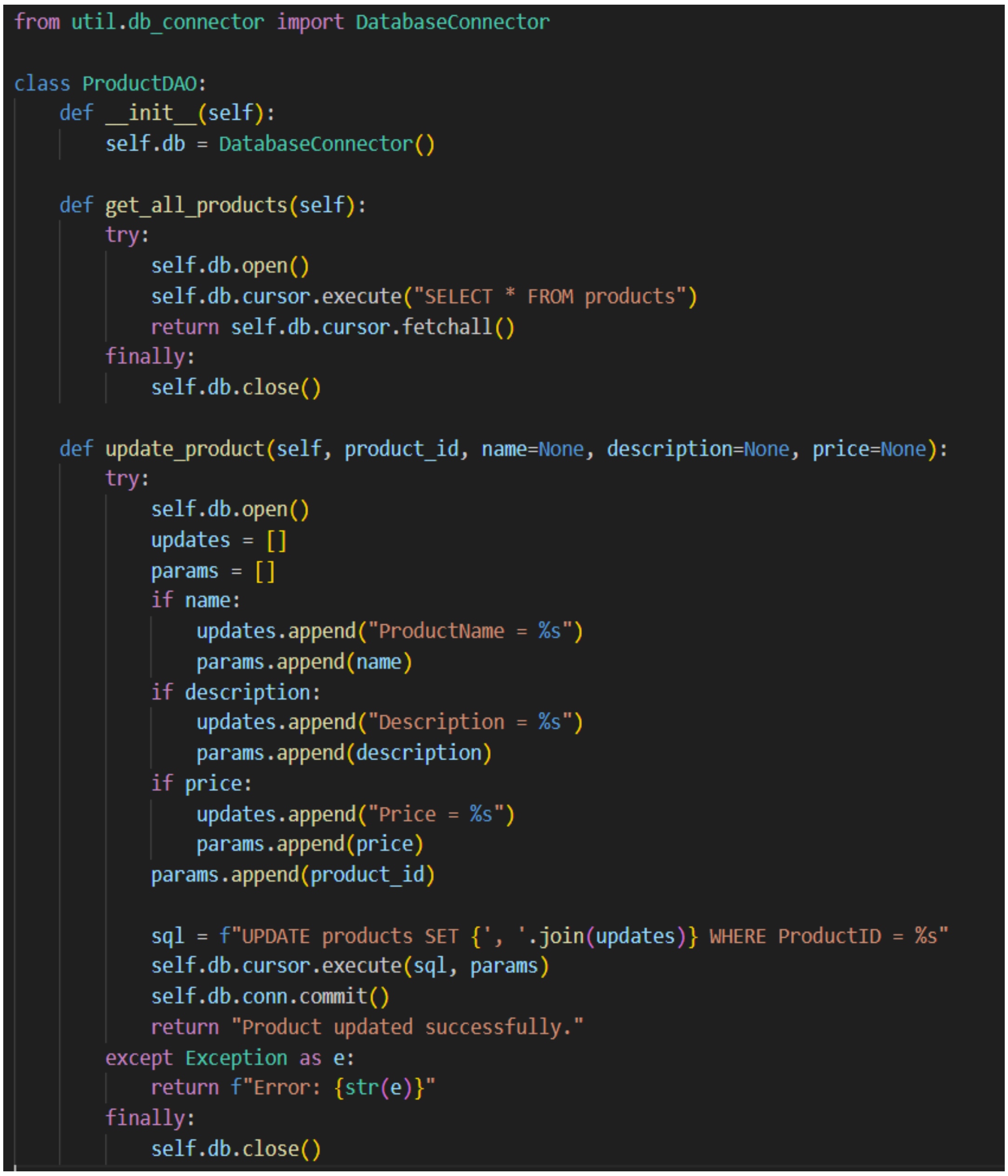
Task: Implement a product search and recommendation engine that uses database connectivity to retrieve relevant product information.



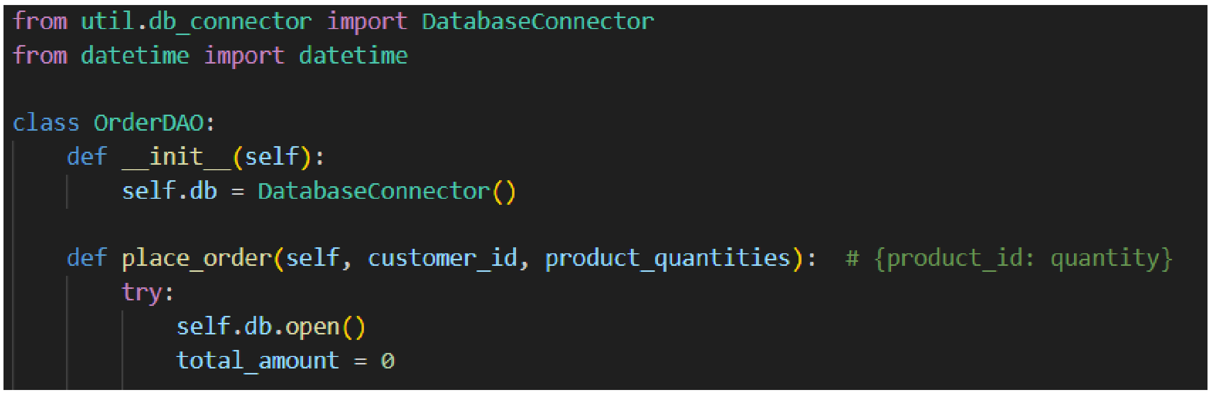
```python
import mysql.connector

class DatabaseConnector:
    def __init__(self):
        self.conn = None
        self.cursor = None

    def open(self):
        self.conn = mysql.connector.connect(
            host='localhost',
            user='your_username',
            password='your_password',
            database='TechShopDB'
        )
        self.cursor = self.conn.cursor(dictionary=True)

    def close(self):
        if self.cursor:
            self.cursor.close()
        if self.conn:
            self.conn.close()
```



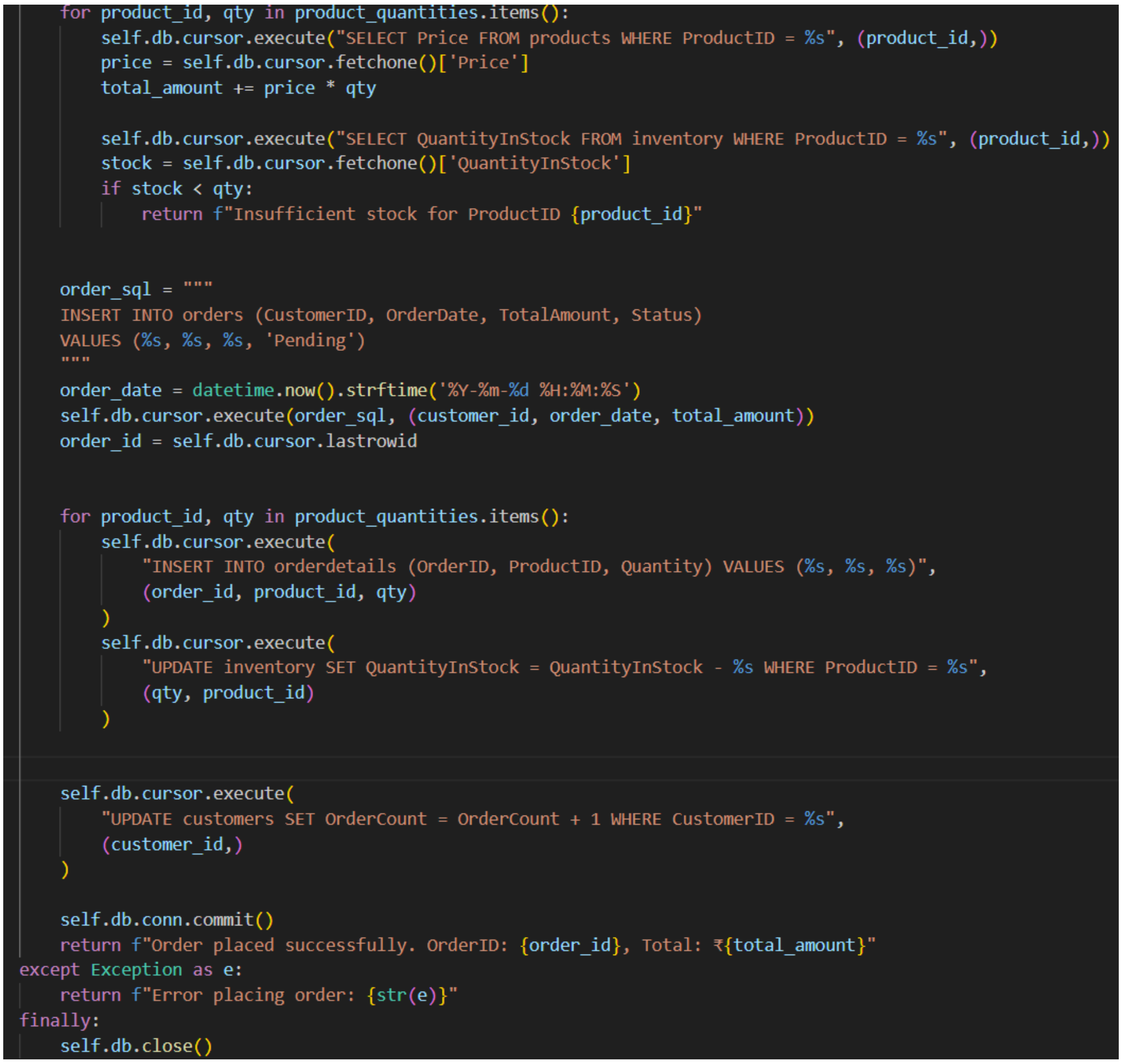
```python
from util.db_connector import DatabaseConnector

class CustomerDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def register_customer(self, first_name, last_name, email, phone, address):
        try:
            self.db.open()
            # Check for duplicate email
            self.db.cursor.execute("SELECT * FROM customers WHERE Email = %s", (email,))
            if self.db.cursor.fetchone():
                return "Email already registered."

            # Insert customer
            sql = """
            INSERT INTO customers (FirstName, LastName, Email, Phone, Address, OrderCount)
            VALUES (%s, %s, %s, %s, %s, 0)
            """
            self.db.cursor.execute(sql, (first_name, last_name, email, phone, address))
            self.db.conn.commit()
            return "Customer registered successfully."
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()
```



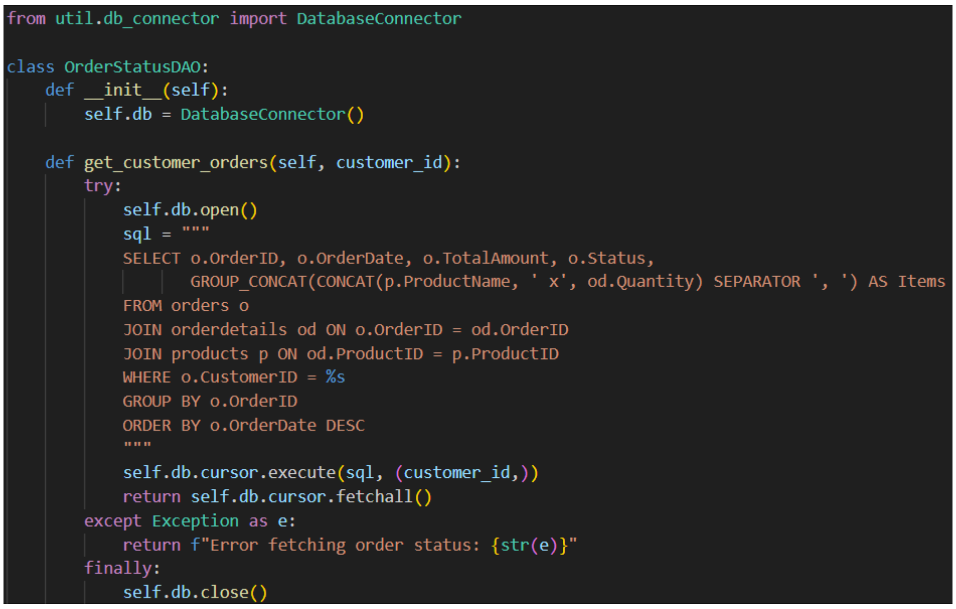
```python
from util.db_connector import DatabaseConnector

class ProductDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def get_all_products(self):
        try:
            self.db.open()
            self.db.cursor.execute("SELECT * FROM products")
            return self.db.cursor.fetchall()
        finally:
            self.db.close()

    def update_product(self, product_id, name=None, description=None, price=None):
        try:
            self.db.open()
            updates = []
            params = []
            if name:
                updates.append("ProductName = %s")
                params.append(name)
            if description:
                updates.append("Description = %s")
                params.append(description)
            if price:
                updates.append("Price = %s")
                params.append(price)
            params.append(product_id)

            sql = f"UPDATE products SET {', '.join(updates)} WHERE ProductID = %s"
            self.db.cursor.execute(sql, params)
            self.db.conn.commit()
            return "Product updated successfully."
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()
```



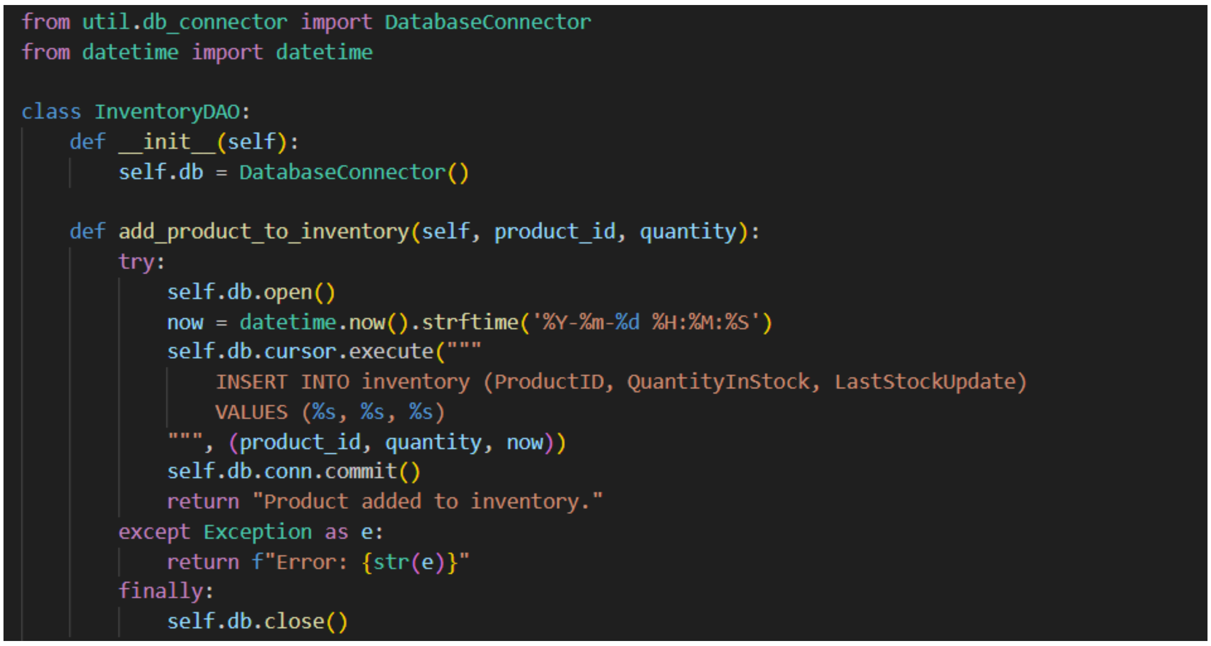
```python
from util.db_connector import DatabaseConnector
from datetime import datetime

class OrderDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def place_order(self, customer_id, product_quantities):  # {product_id: quantity}
        try:
            self.db.open()
            total_amount = 0
```



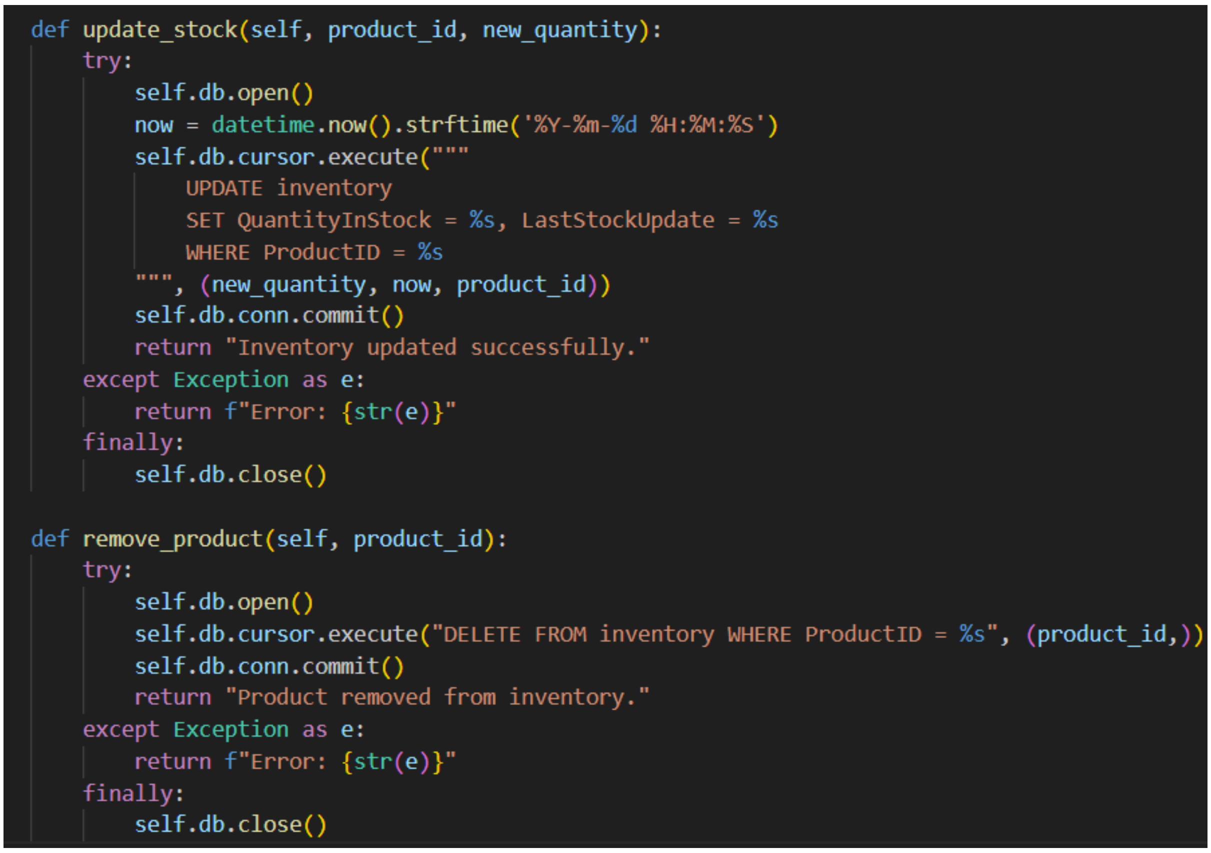
```python
    for product_id, qty in product_quantities.items():
        self.db.cursor.execute("SELECT Price FROM products WHERE ProductID = %s", (product_id,))
        price = self.db.cursor.fetchone()['Price']
        total_amount += price * qty

        self.db.cursor.execute("SELECT QuantityInStock FROM inventory WHERE ProductID = %s", (product_id,))
        stock = self.db.cursor.fetchone()['QuantityInStock']
        if stock < qty:
            return f"Insufficient stock for ProductID {product_id}"


    order_sql = """
    INSERT INTO orders (CustomerID, OrderDate, TotalAmount, Status)
    VALUES (%s, %s, %s, 'Pending')
    """
    order_date = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    self.db.cursor.execute(order_sql, (customer_id, order_date, total_amount))
    order_id = self.db.cursor.lastrowid


    for product_id, qty in product_quantities.items():
        self.db.cursor.execute(
            "INSERT INTO orderdetails (OrderID, ProductID, Quantity) VALUES (%s, %s, %s)",
            (order_id, product_id, qty)
        )
        self.db.cursor.execute(
            "UPDATE inventory SET QuantityInStock = QuantityInStock - %s WHERE ProductID = %s",
            (qty, product_id)
        )


    self.db.cursor.execute(
        "UPDATE customers SET OrderCount = OrderCount + 1 WHERE CustomerID = %s",
        (customer_id,)
    )

    self.db.conn.commit()
    return f"Order placed successfully. OrderID: {order_id}, Total: ₹{total_amount}"
except Exception as e:
    return f"Error placing order: {str(e)}"
finally:
    self.db.close()
```



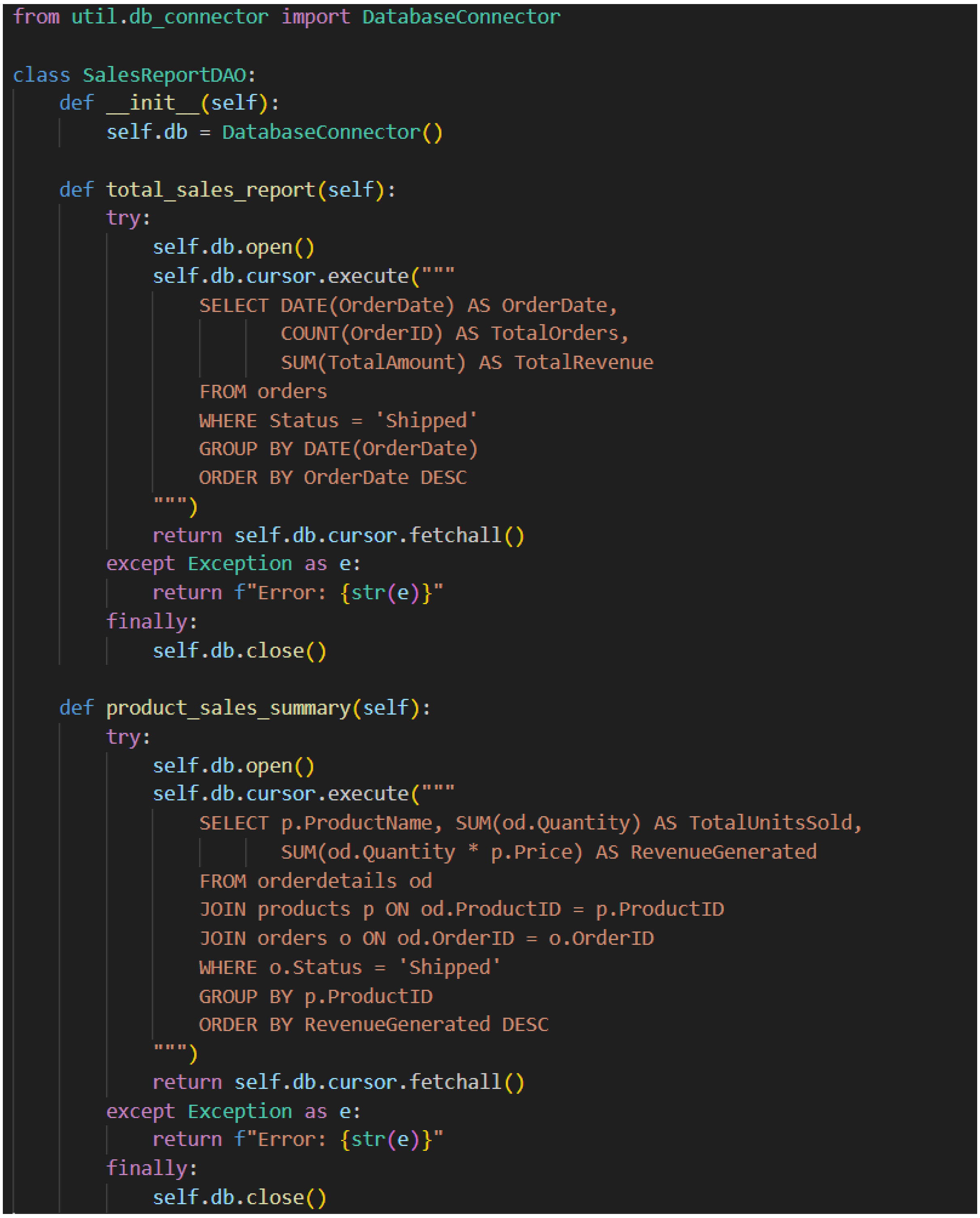
```python
from util.db_connector import DatabaseConnector

class OrderStatusDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def get_customer_orders(self, customer_id):
        try:
            self.db.open()
            sql = """
            SELECT o.OrderID, o.OrderDate, o.TotalAmount, o.Status,
                   GROUP_CONCAT(CONCAT(p.ProductName, ' x', od.Quantity) SEPARATOR ', ') AS Items
            FROM orders o
            JOIN orderdetails od ON o.OrderID = od.OrderID
            JOIN products p ON od.ProductID = p.ProductID
            WHERE o.CustomerID = %s
            GROUP BY o.OrderID
            ORDER BY o.OrderDate DESC
            """
            self.db.cursor.execute(sql, (customer_id,))
            return self.db.cursor.fetchall()
        except Exception as e:
            return f"Error fetching order status: {str(e)}"
        finally:
            self.db.close()
```



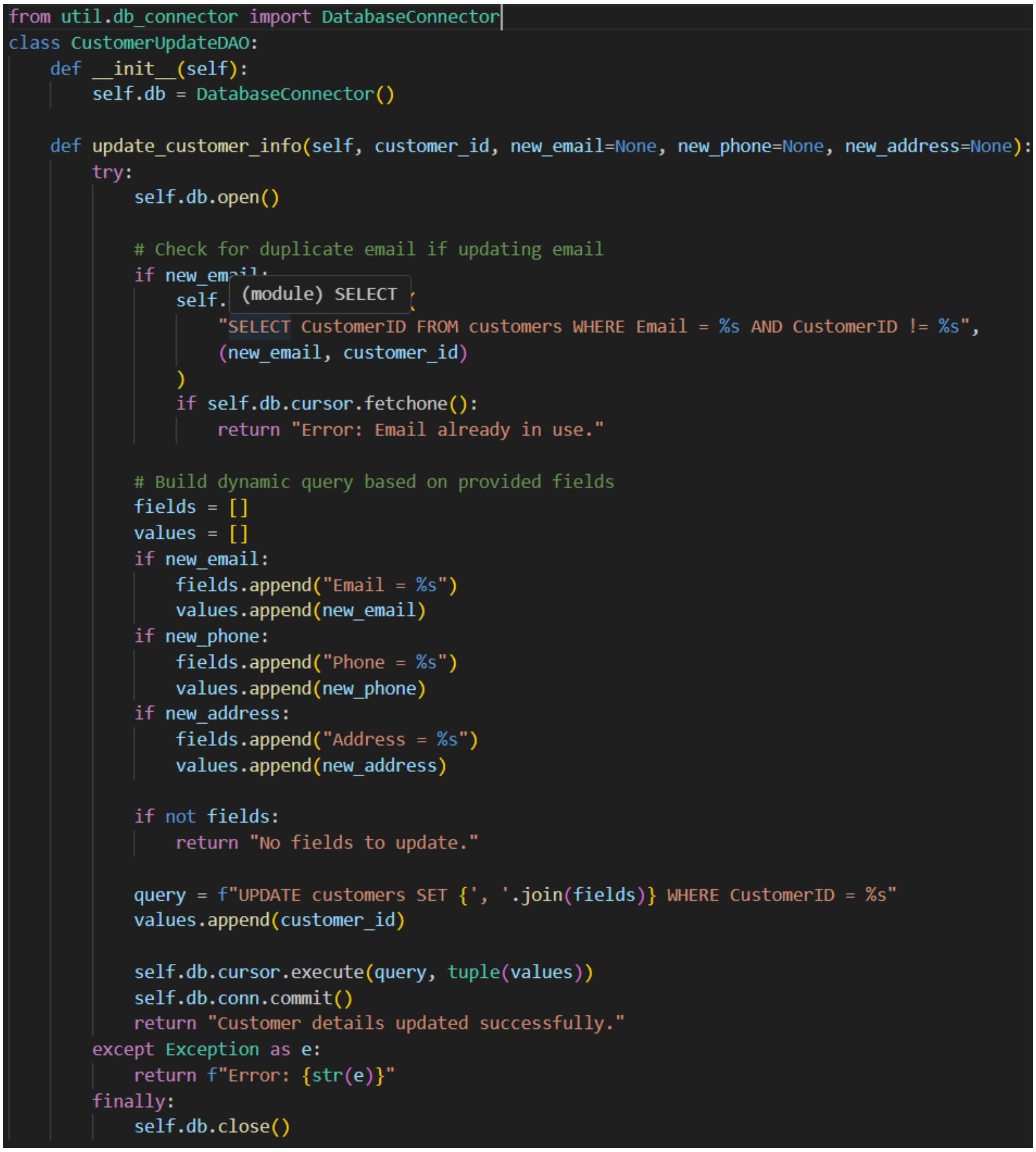
```python
from util.db_connector import DatabaseConnector
from datetime import datetime

class InventoryDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def add_product_to_inventory(self, product_id, quantity):
        try:
            self.db.open()
            now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            self.db.cursor.execute("""
                INSERT INTO inventory (ProductID, QuantityInStock, LastStockUpdate)
                VALUES (%s, %s, %s)
            """, (product_id, quantity, now))
            self.db.conn.commit()
            return "Product added to inventory."
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()
```



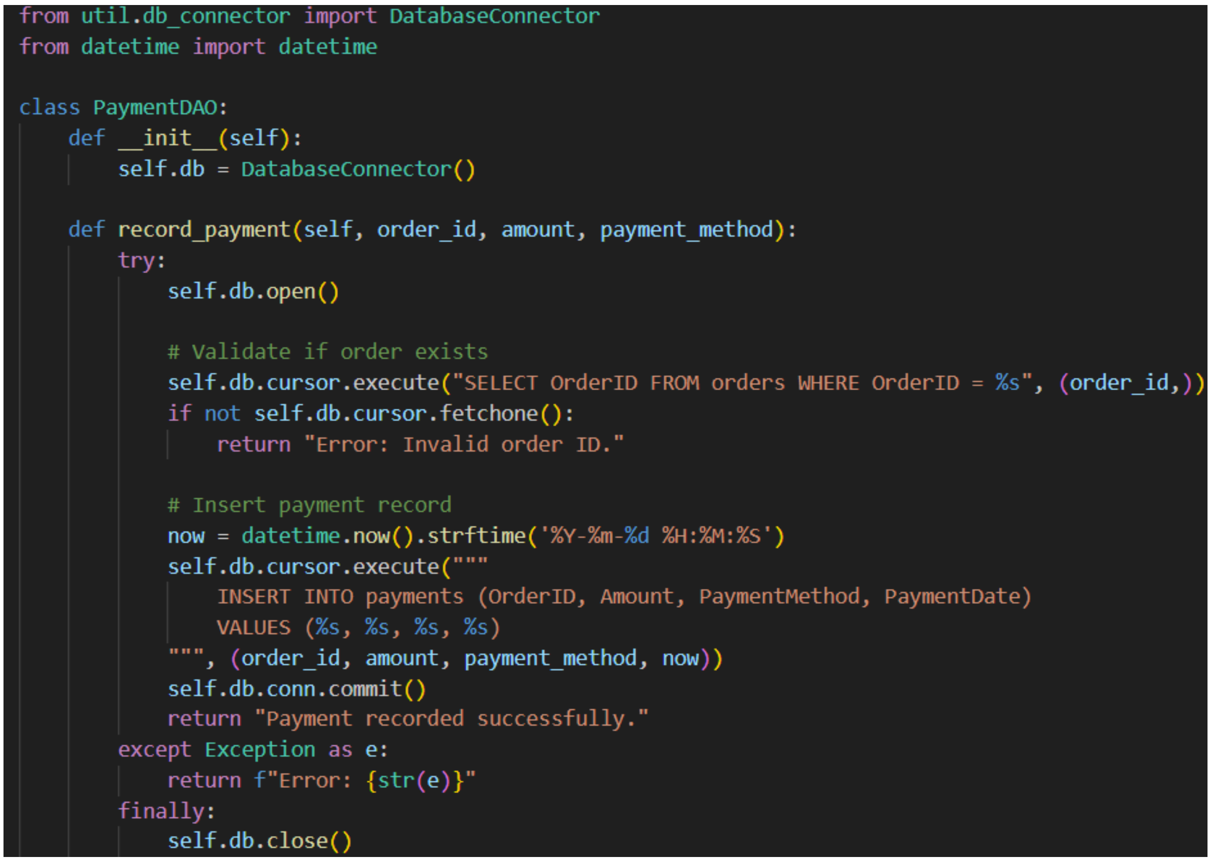
```python
def update_stock(self, product_id, new_quantity):
    try:
        self.db.open()
        now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        self.db.cursor.execute("""
            UPDATE inventory
            SET QuantityInStock = %s, LastStockUpdate = %s
            WHERE ProductID = %s
        """, (new_quantity, now, product_id))
        self.db.conn.commit()
        return "Inventory updated successfully."
    except Exception as e:
        return f"Error: {str(e)}"
    finally:
        self.db.close()

def remove_product(self, product_id):
    try:
        self.db.open()
        self.db.cursor.execute("DELETE FROM inventory WHERE ProductID = %s", (product_id,))
        self.db.conn.commit()
        return "Product removed from inventory."
    except Exception as e:
        return f"Error: {str(e)}"
    finally:
        self.db.close()
```

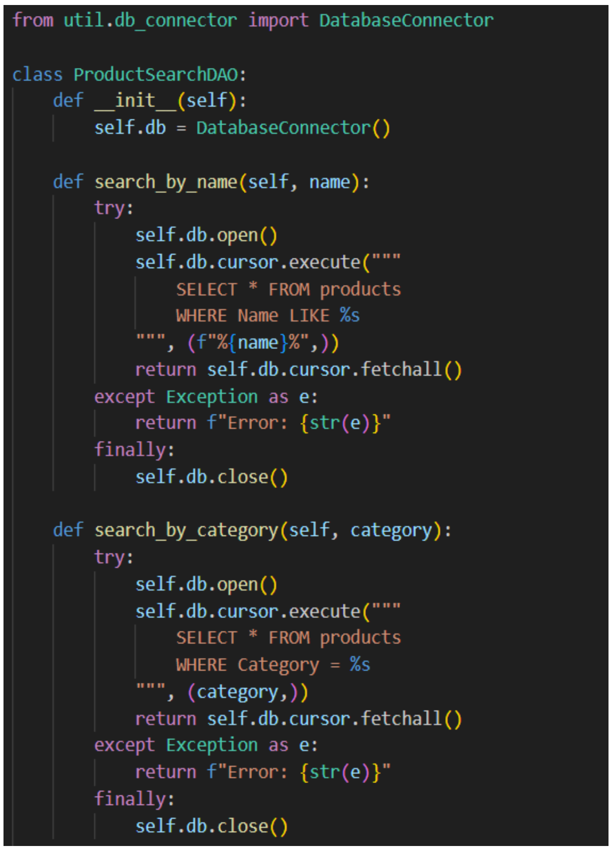


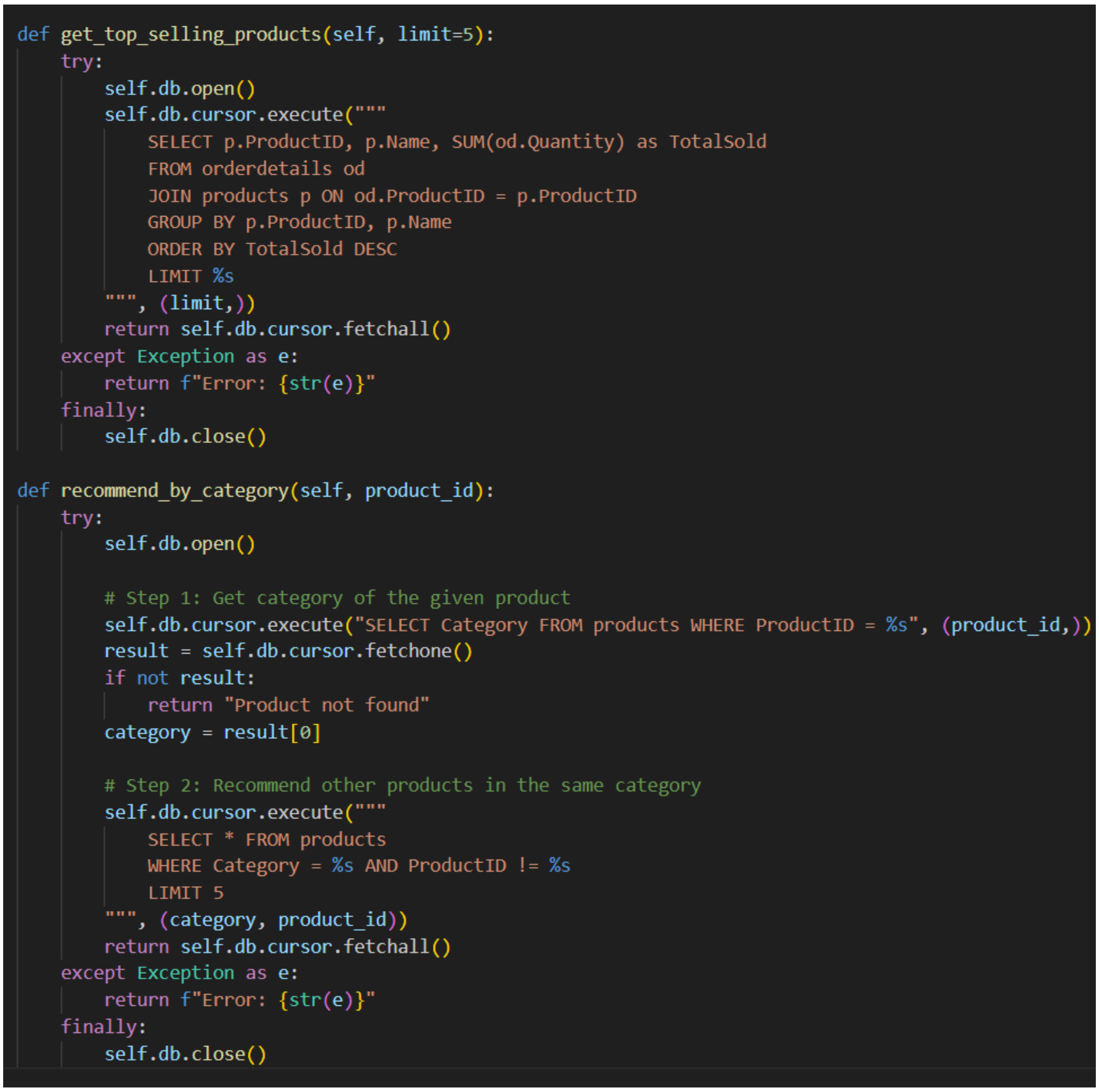
```python
from util.db_connector import DatabaseConnector

class SalesReportDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def total_sales_report(self):
        try:
            self.db.open()
            self.db.cursor.execute("""
                SELECT DATE(OrderDate) AS OrderDate,
                       COUNT(OrderID) AS TotalOrders,
                       SUM(TotalAmount) AS TotalRevenue
                FROM orders
                WHERE Status = 'Shipped'
                GROUP BY DATE(OrderDate)
                ORDER BY OrderDate DESC
            """)
            return self.db.cursor.fetchall()
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()

    def product_sales_summary(self):
        try:
            self.db.open()
            self.db.cursor.execute("""
                SELECT p.ProductName, SUM(od.Quantity) AS TotalUnitsSold,
                       SUM(od.Quantity * p.Price) AS RevenueGenerated
                FROM orderdetails od
                JOIN products p ON od.ProductID = p.ProductID
                JOIN orders o ON od.OrderID = o.OrderID
                WHERE o.Status = 'Shipped'
                GROUP BY p.ProductID
                ORDER BY RevenueGenerated DESC
            """)
            return self.db.cursor.fetchall()
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()
```



```python
from util.db_connector import DatabaseConnector
class CustomerUpdateDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def update_customer_info(self, customer_id, new_email=None, new_phone=None, new_address=None):
        try:
            self.db.open()

            # Check for duplicate email if updating email
            if new_em
                self. (module) SELECT (
                    "SELECT CustomerID FROM customers WHERE Email = %s AND CustomerID != %s",
                    (new_email, customer_id)
                )
                if self.db.cursor.fetchone():
                    return "Error: Email already in use."

            # Build dynamic query based on provided fields
            fields = []
            values = []
            if new_email:
                fields.append("Email = %s")
                values.append(new_email)
            if new_phone:
                fields.append("Phone = %s")
                values.append(new_phone)
            if new_address:
                fields.append("Address = %s")
                values.append(new_address)

            if not fields:
                return "No fields to update."

            query = f"UPDATE customers SET {', '.join(fields)} WHERE CustomerID = %s"
            values.append(customer_id)

            self.db.cursor.execute(query, tuple(values))
            self.db.conn.commit()
            return "Customer details updated successfully."
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()
```


```python
from util.db_connector import DatabaseConnector
from datetime import datetime

class PaymentDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def record_payment(self, order_id, amount, payment_method):
        try:
            self.db.open()

            # Validate if order exists
            self.db.cursor.execute("SELECT OrderID FROM orders WHERE OrderID = %s", (order_id,))
            if not self.db.cursor.fetchone():
                return "Error: Invalid order ID."

            # Insert payment record
            now = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            self.db.cursor.execute("""
                INSERT INTO payments (OrderID, Amount, PaymentMethod, PaymentDate)
                VALUES (%s, %s, %s, %s)
            """, (order_id, amount, payment_method, now))
            self.db.conn.commit()
            return "Payment recorded successfully."
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()
```

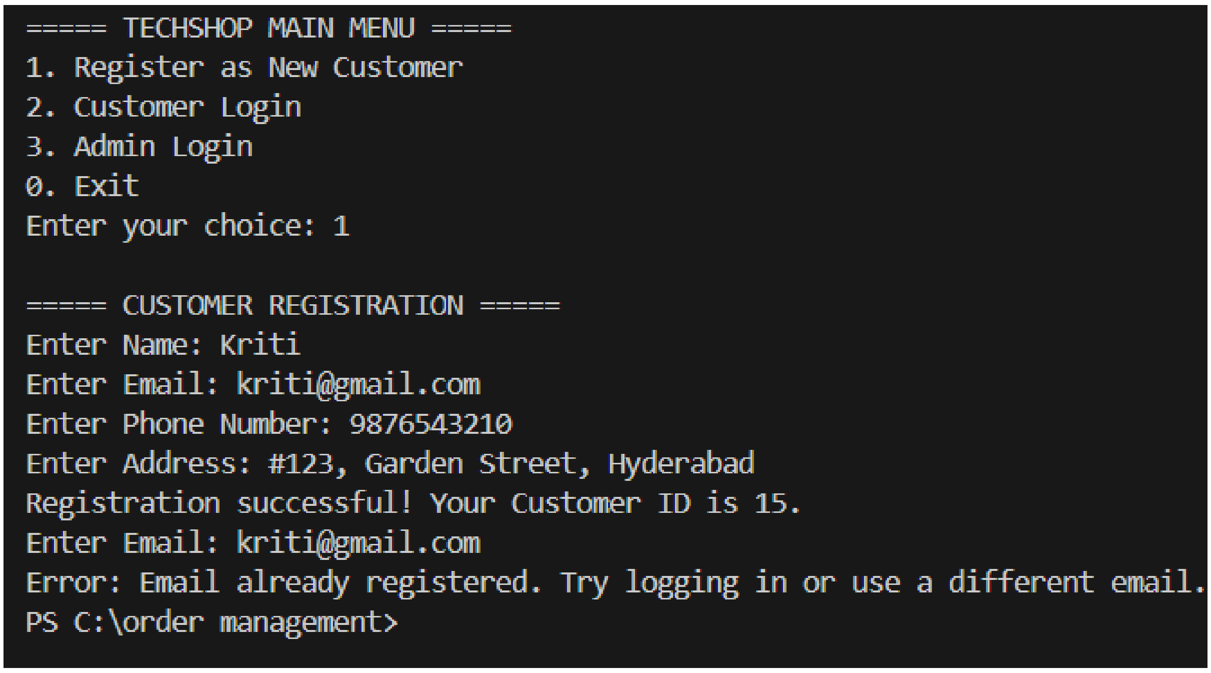

```python
from util.db_connector import DatabaseConnector

class ProductSearchDAO:
    def __init__(self):
        self.db = DatabaseConnector()

    def search_by_name(self, name):
        try:
            self.db.open()
            self.db.cursor.execute("""
                SELECT * FROM products
                WHERE Name LIKE %s
            """, (f"%{name}%",))
            return self.db.cursor.fetchall()
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()

    def search_by_category(self, category):
        try:
            self.db.open()
            self.db.cursor.execute("""
                SELECT * FROM products
                WHERE Category = %s
            """, (category,))
            return self.db.cursor.fetchall()
        except Exception as e:
            return f"Error: {str(e)}"
        finally:
            self.db.close()
```



```python
def get_top_selling_products(self, limit=5):
    try:
        self.db.open()
        self.db.cursor.execute("""
            SELECT p.ProductID, p.Name, SUM(od.Quantity) as TotalSold
            FROM orderdetails od
            JOIN products p ON od.ProductID = p.ProductID
            GROUP BY p.ProductID, p.Name
            ORDER BY TotalSold DESC
            LIMIT %s
        """, (limit,))
        return self.db.cursor.fetchall()
    except Exception as e:
        return f"Error: {str(e)}"
    finally:
        self.db.close()

def recommend_by_category(self, product_id):
    try:
        self.db.open()

        # Step 1: Get category of the given product
        self.db.cursor.execute("SELECT Category FROM products WHERE ProductID = %s", (product_id,))
        result = self.db.cursor.fetchone()
        if not result:
            return "Product not found"
        category = result[0]

        # Step 2: Recommend other products in the same category
        self.db.cursor.execute("""
            SELECT * FROM products
            WHERE Category = %s AND ProductID != %s
            LIMIT 5
        """, (category, product_id))
        return self.db.cursor.fetchall()
    except Exception as e:
        return f"Error: {str(e)}"
    finally:
        self.db.close()
```

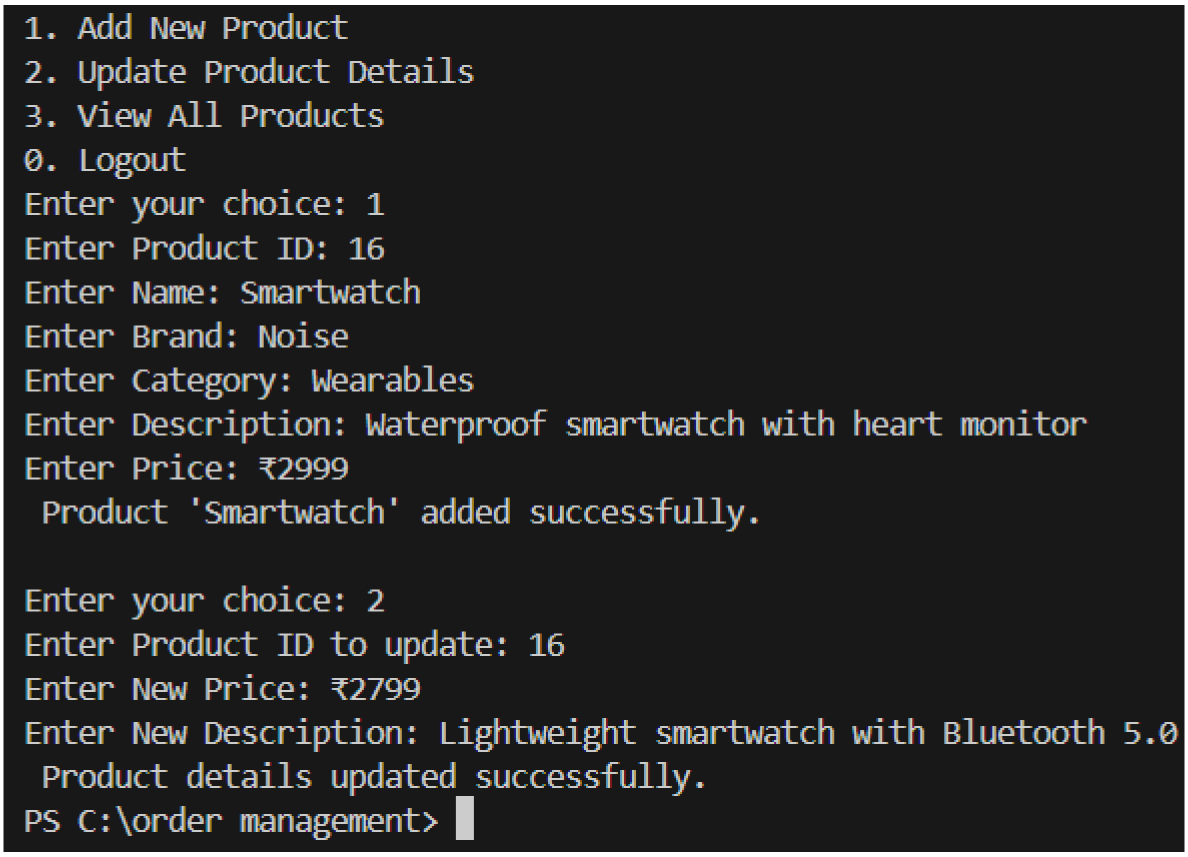
**OUTPUT SCREEN SHOTS**

1: Customer Registration:



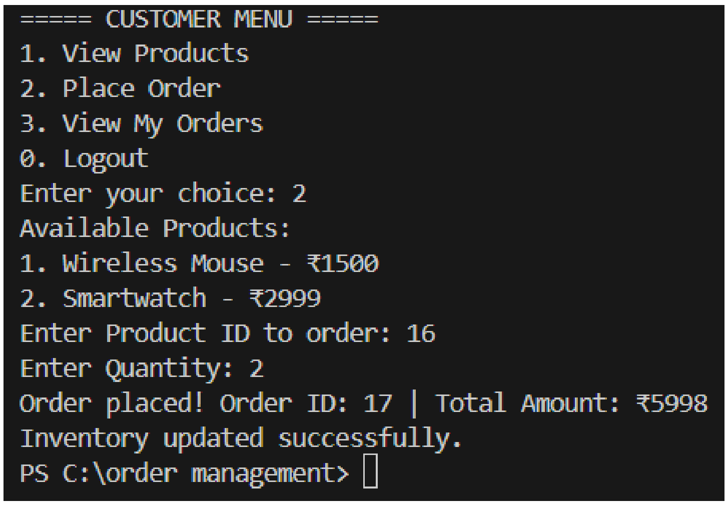
```
===== TECHSHOP MAIN MENU =====
1. Register as New Customer
2. Customer Login
3. Admin Login
0. Exit
Enter your choice: 1

===== CUSTOMER REGISTRATION =====
Enter Name: Kriti
Enter Email: kriti@gmail.com
Enter Phone Number: 9876543210
Enter Address: #123, Garden Street, Hyderabad
Registration successful! Your Customer ID is 15.
Enter Email: kriti@gmail.com
Error: Email already registered. Try logging in or use a different email.
PS C:\order management>
```

2: Product Catalog Management:



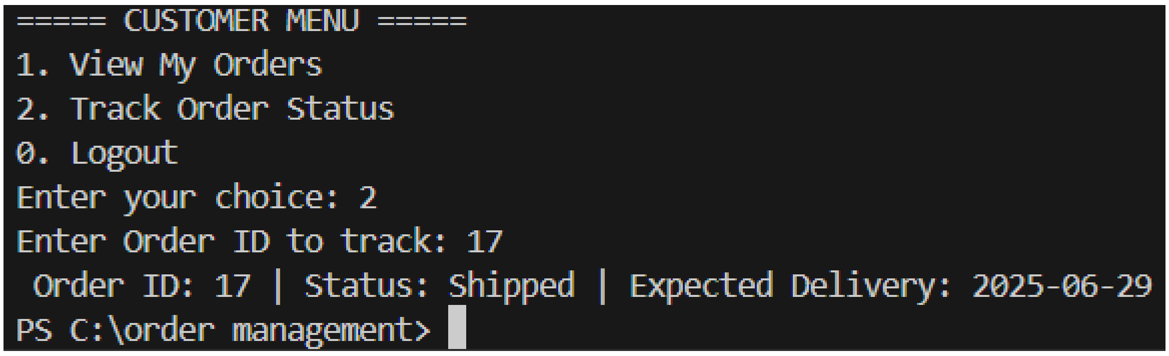
```
1. Add New Product
2. Update Product Details
3. View All Products
0. Logout
Enter your choice: 1
Enter Product ID: 16
Enter Name: Smartwatch
Enter Brand: Noise
Enter Category: Wearables
Enter Description: Waterproof smartwatch with heart monitor
Enter Price: ₹2999
 Product 'Smartwatch' added successfully.

Enter your choice: 2
Enter Product ID to update: 16
Enter New Price: ₹2799
Enter New Description: Lightweight smartwatch with Bluetooth 5.0
 Product details updated successfully.
PS C:\order management>
```

3: Placing Customer Orders:



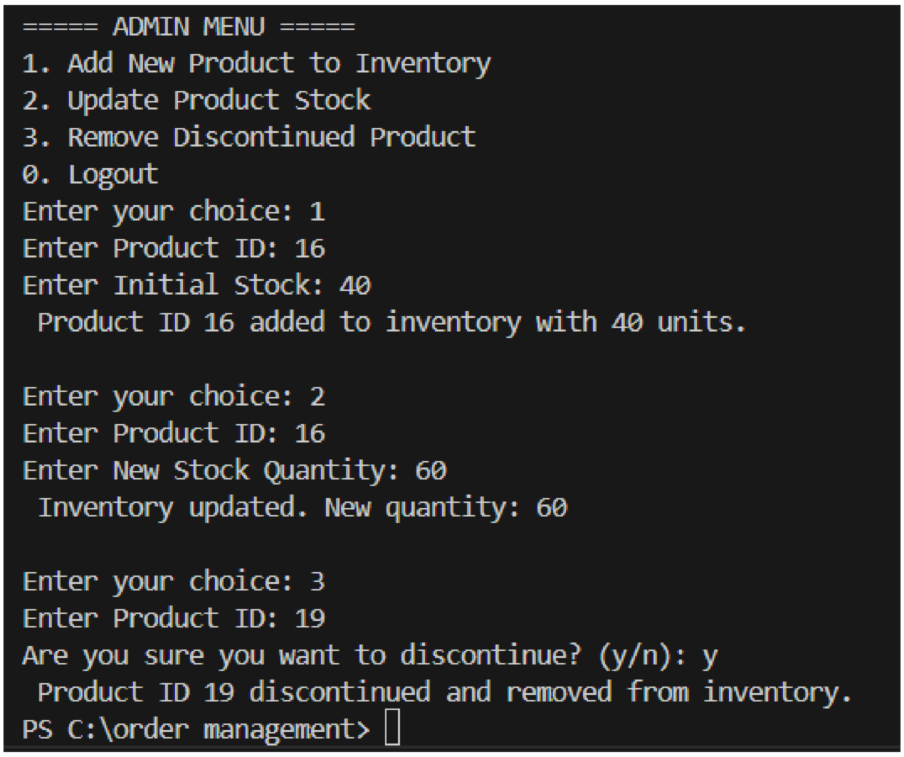
```
===== CUSTOMER MENU =====
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 2
Available Products:
1. Wireless Mouse - ₹1500
2. Smartwatch - ₹2999
Enter Product ID to order: 16
Enter Quantity: 2
Order placed! Order ID: 17 | Total Amount: ₹5998
Inventory updated successfully.
PS C:\order management> 
```

4: Tracking Order Status:



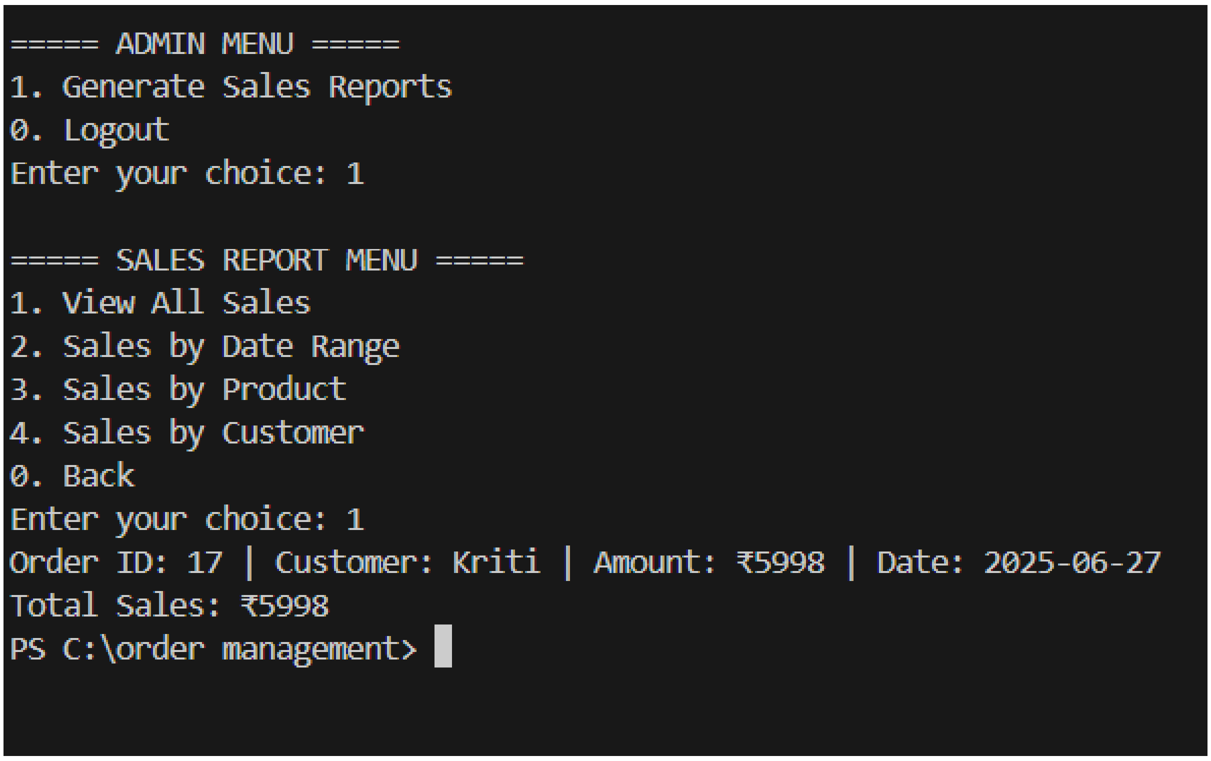
```
===== CUSTOMER MENU =====
1. View My Orders
2. Track Order Status
0. Logout
Enter your choice: 2
Enter Order ID to track: 17
 Order ID: 17 | Status: Shipped | Expected Delivery: 2025-06-29
PS C:\order management> 
```

5: Inventory Management:



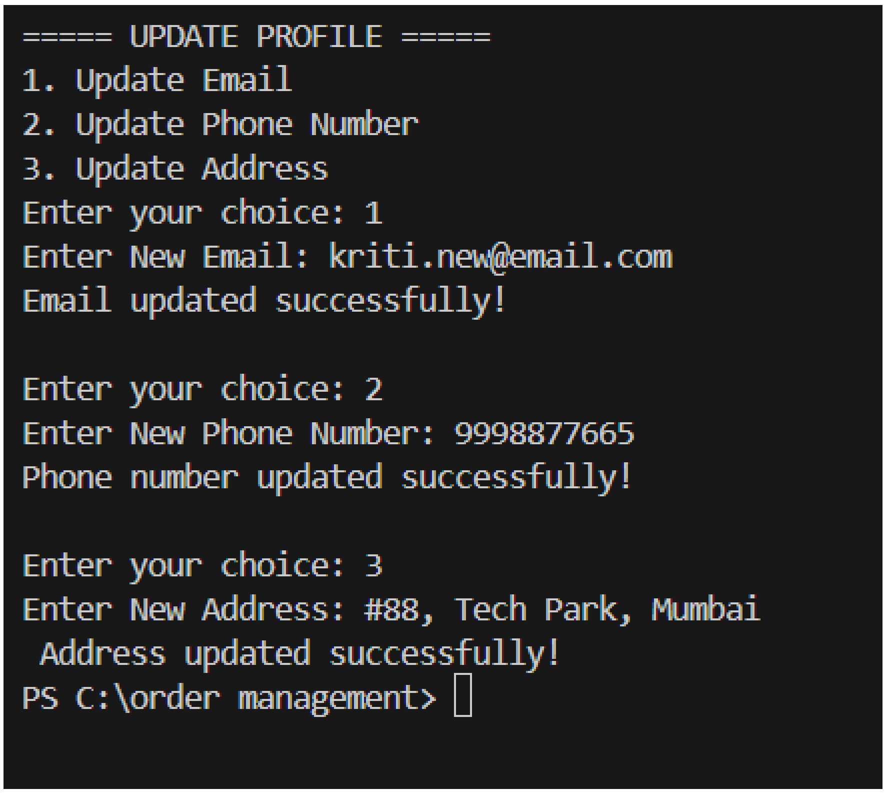
```
===== ADMIN MENU =====
1. Add New Product to Inventory
2. Update Product Stock
3. Remove Discontinued Product
0. Logout
Enter your choice: 1
Enter Product ID: 16
Enter Initial Stock: 40
 Product ID 16 added to inventory with 40 units.

Enter your choice: 2
Enter Product ID: 16
Enter New Stock Quantity: 60
 Inventory updated. New quantity: 60

Enter your choice: 3
Enter Product ID: 19
Are you sure you want to discontinue? (y/n): y
 Product ID 19 discontinued and removed from inventory.
PS C:\order management> 
```

6: Sales Reporting:



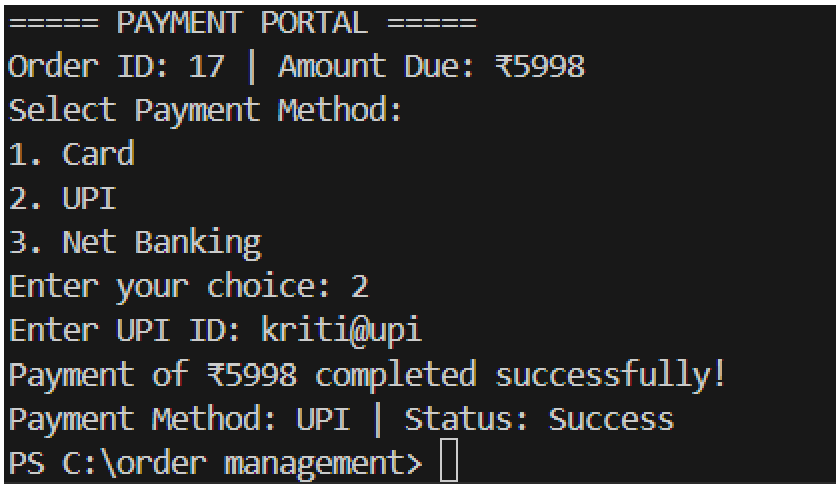
```
===== ADMIN MENU =====
1. Generate Sales Reports
0. Logout
Enter your choice: 1

===== SALES REPORT MENU =====
1. View All Sales
2. Sales by Date Range
3. Sales by Product
4. Sales by Customer
0. Back
Enter your choice: 1
Order ID: 17 | Customer: Kriti | Amount: ₹5998 | Date: 2025-06-27
Total Sales: ₹5998
PS C:\order management>
```

7: Customer Account Updates:



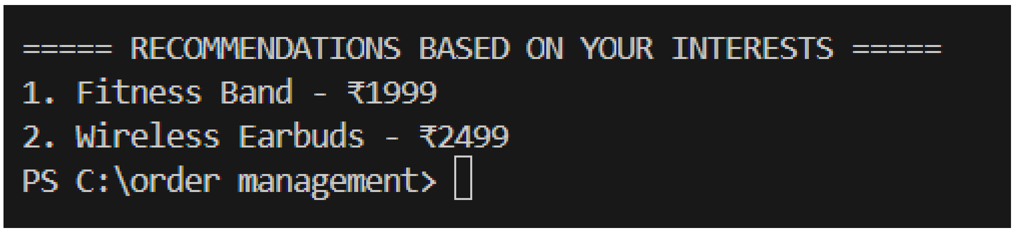
```
===== UPDATE PROFILE =====
1. Update Email
2. Update Phone Number
3. Update Address
Enter your choice: 1
Enter New Email: kriti.new@email.com
Email updated successfully!

Enter your choice: 2
Enter New Phone Number: 9998877665
Phone number updated successfully!

Enter your choice: 3
Enter New Address: #88, Tech Park, Mumbai
 Address updated successfully!
PS C:\order management>
```

8: Payment Processing:



```
===== PAYMENT PORTAL =====
Order ID: 17 | Amount Due: ₹5998
Select Payment Method:
1. Card
2. UPI
3. Net Banking
Enter your choice: 2
Enter UPI ID: kriti@upi
Payment of ₹5998 completed successfully!
Payment Method: UPI | Status: Success
PS C:\order management>
```

9: Product Search and Recommendation:



```
===== RECOMMENDATIONS BASED ON YOUR INTERESTS =====
1. Fitness Band - ₹1999
2. Wireless Earbuds - ₹2499
PS C:\order management>
```