IMPLEMENT OOPs

NAME: HARSHA K

Task 1: Classes and Their Attributes:

Customers Class:

Attributes:

• CustomerID (int)

• FirstName (string)

• LastName (string)

• Email (string)

• Phone (string)

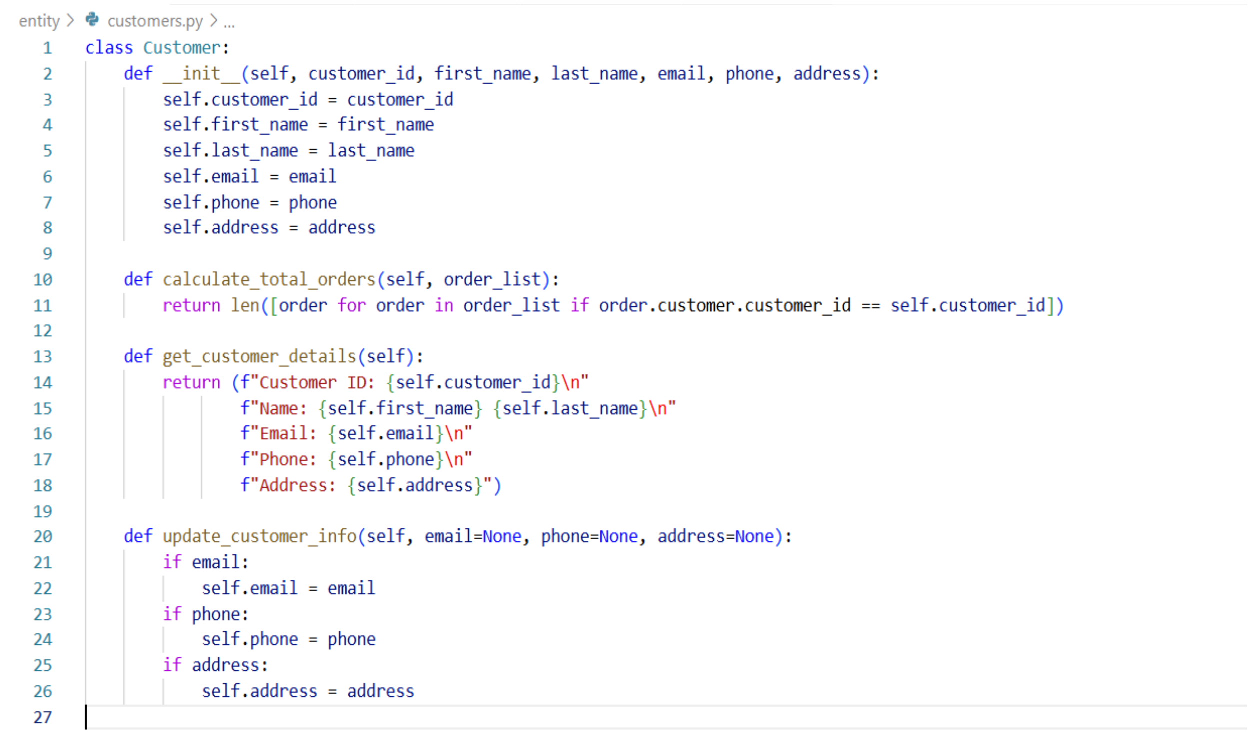
• Address (string)

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).



entity > customers.py > ...

```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

    def calculate_total_orders(self, order_list):
        return len([order for order in order_list if order.customer.customer_id == self.customer_id])

    def get_customer_details(self):
        return (f"Customer ID: {self.customer_id}\n"
                f"Name: {self.first_name} {self.last_name}\n"
                f"Email: {self.email}\n"
                f"Phone: {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)

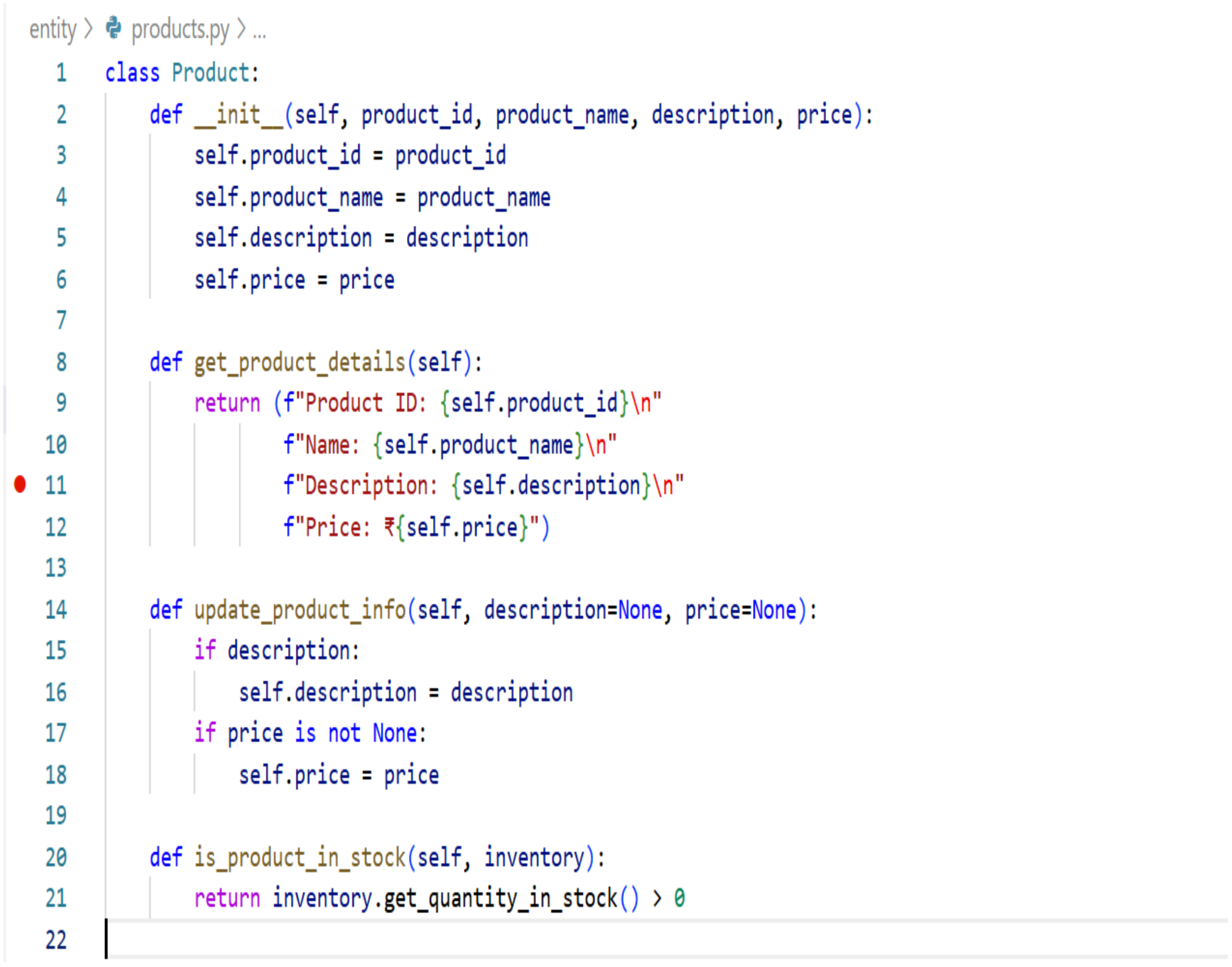
• Price (decimal)

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.



entity > products.py > ...

```python
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

    def get_product_details(self):
        return (f"Product ID: {self.product_id}\n"
                f"Name: {self.product_name}\n"
                f"Description: {self.description}\n"
                f"Price: ₹{self.price}")

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

    def is_product_in_stock(self, inventory):
        return inventory.get_quantity_in_stock() > 0

```

Orders Class:

Attributes:

• OrderID (int)

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

• TotalAmount (decimal)

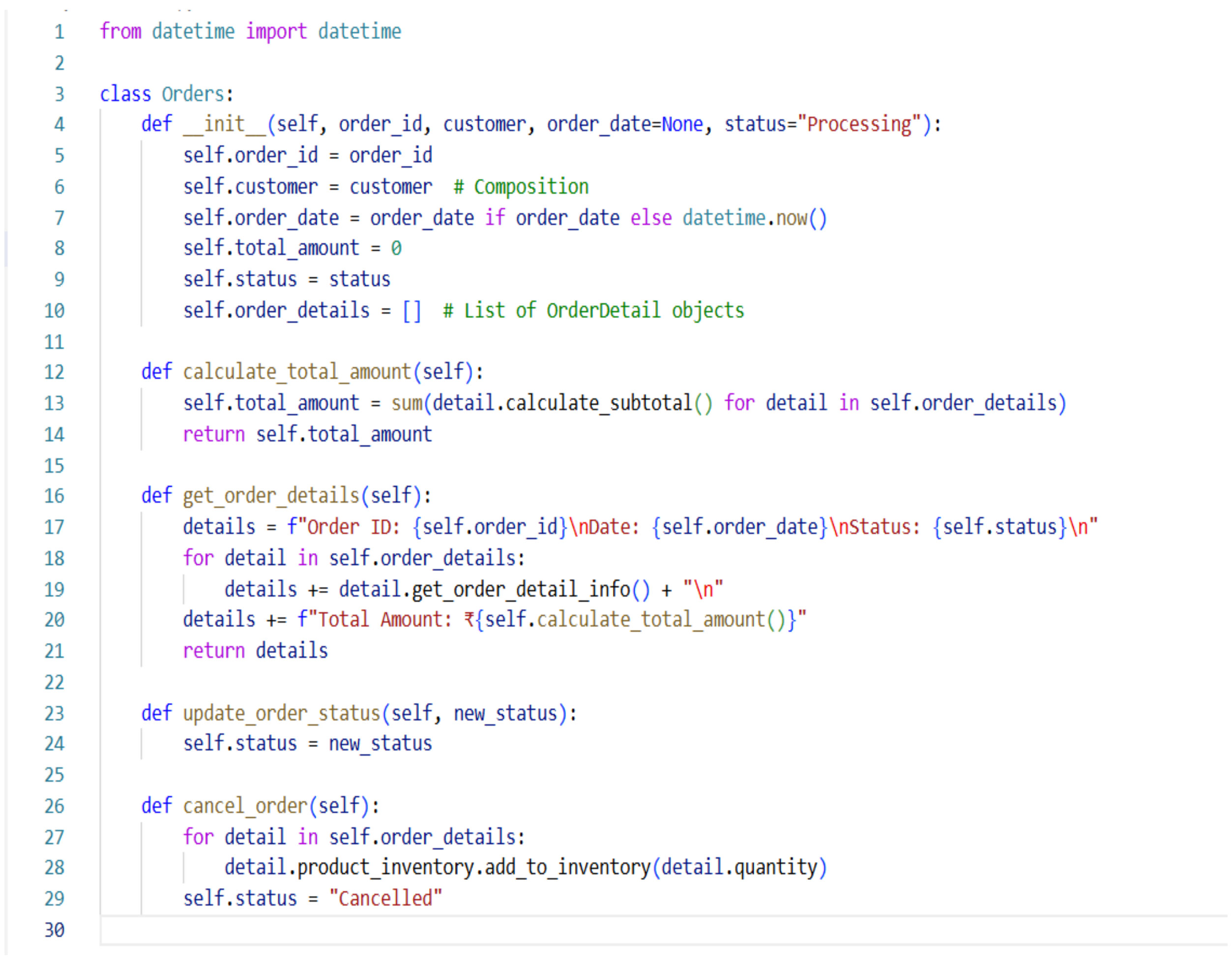
Methods:

• 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 Orders:
    def __init__(self, order_id, customer, order_date=None, status="Processing"):
        self.order_id = order_id
        self.customer = customer  # Composition
        self.order_date = order_date if order_date else datetime.now()
        self.total_amount = 0
        self.status = status
        self.order_details = []  # List of OrderDetail objects

    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):
        details = f"Order ID: {self.order_id}\nDate: {self.order_date}\nStatus: {self.status}\n"
        for detail in self.order_details:
            details += detail.get_order_detail_info() + "\n"
        details += f"Total Amount: ₹{self.calculate_total_amount()}"
        return details

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

    def cancel_order(self):
        for detail in self.order_details:
            detail.product_inventory.add_to_inventory(detail.quantity)
        self.status = "Cancelled"

```

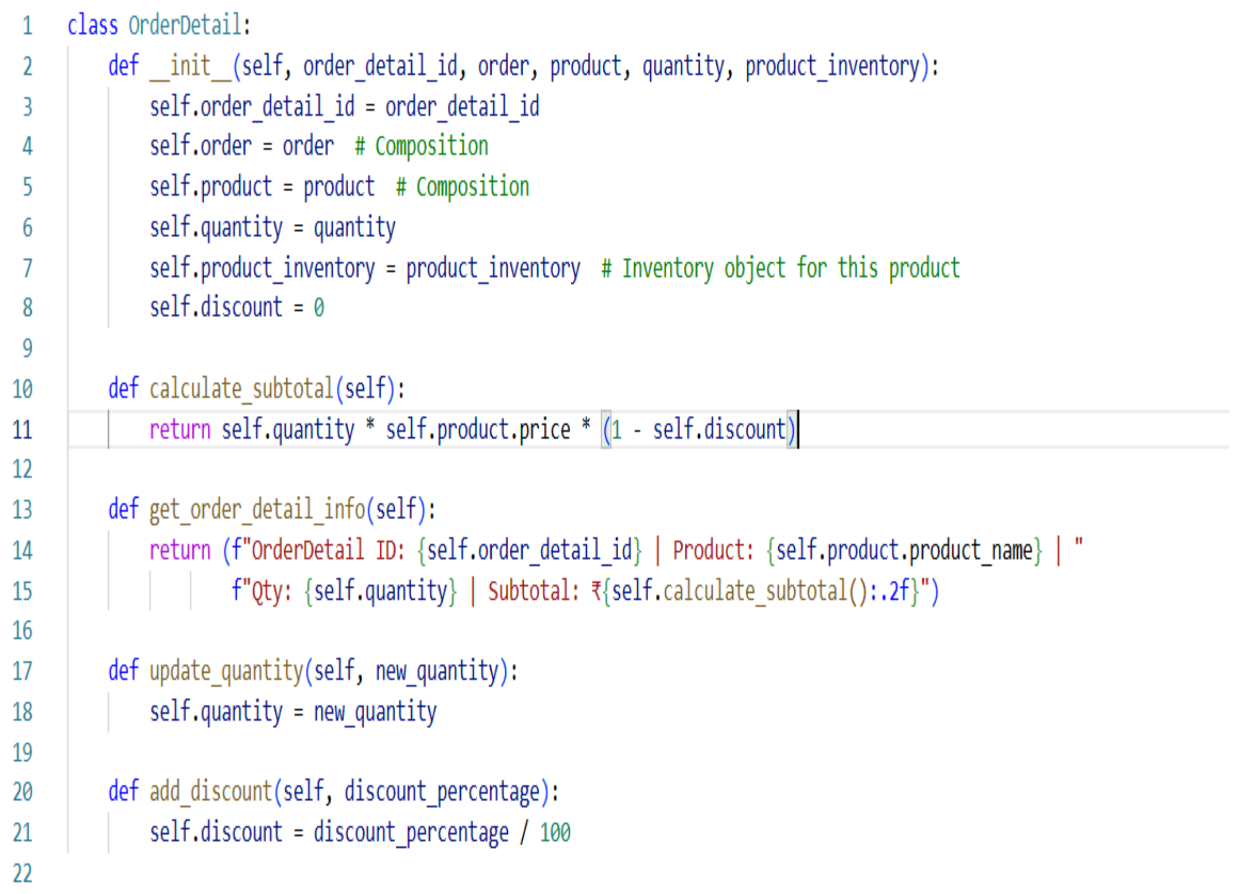
OrderDetails Class:

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, product_inventory):
        self.order_detail_id = order_detail_id
        self.order = order  # Composition
        self.product = product  # Composition
        self.quantity = quantity
        self.product_inventory = product_inventory  # Inventory object for this product
        self.discount = 0

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

    def get_order_detail_info(self):
        return (f"OrderDetail ID: {self.order_detail_id} | Product: {self.product.product_name} | "
                f"Qty: {self.quantity} | Subtotal: ₹{self.calculate_subtotal():.2f}")

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

    def add_discount(self, discount_percentage):
        self.discount = discount_percentage / 100
```

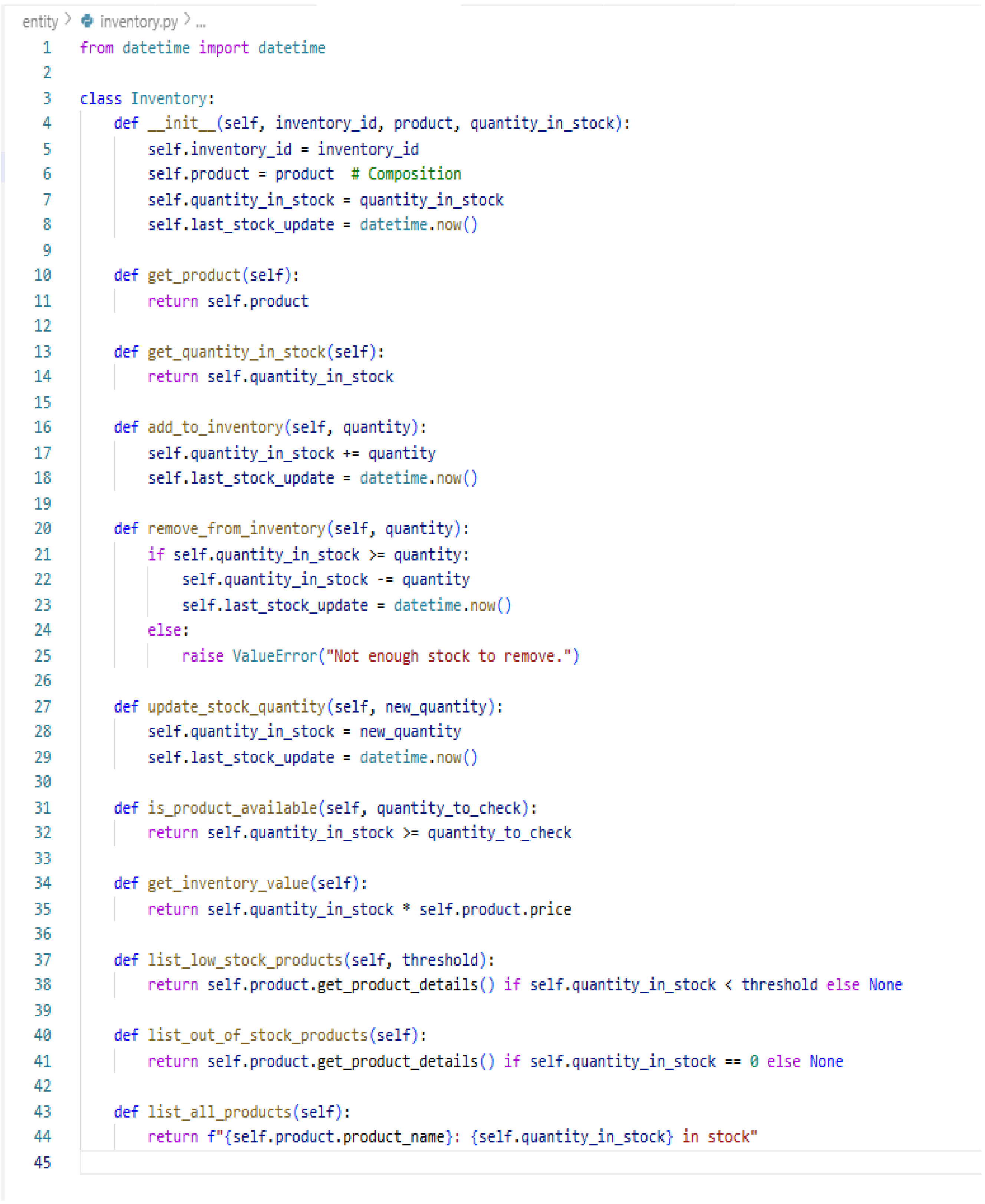
Inventory class:

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.


entity > inventory.py > ...

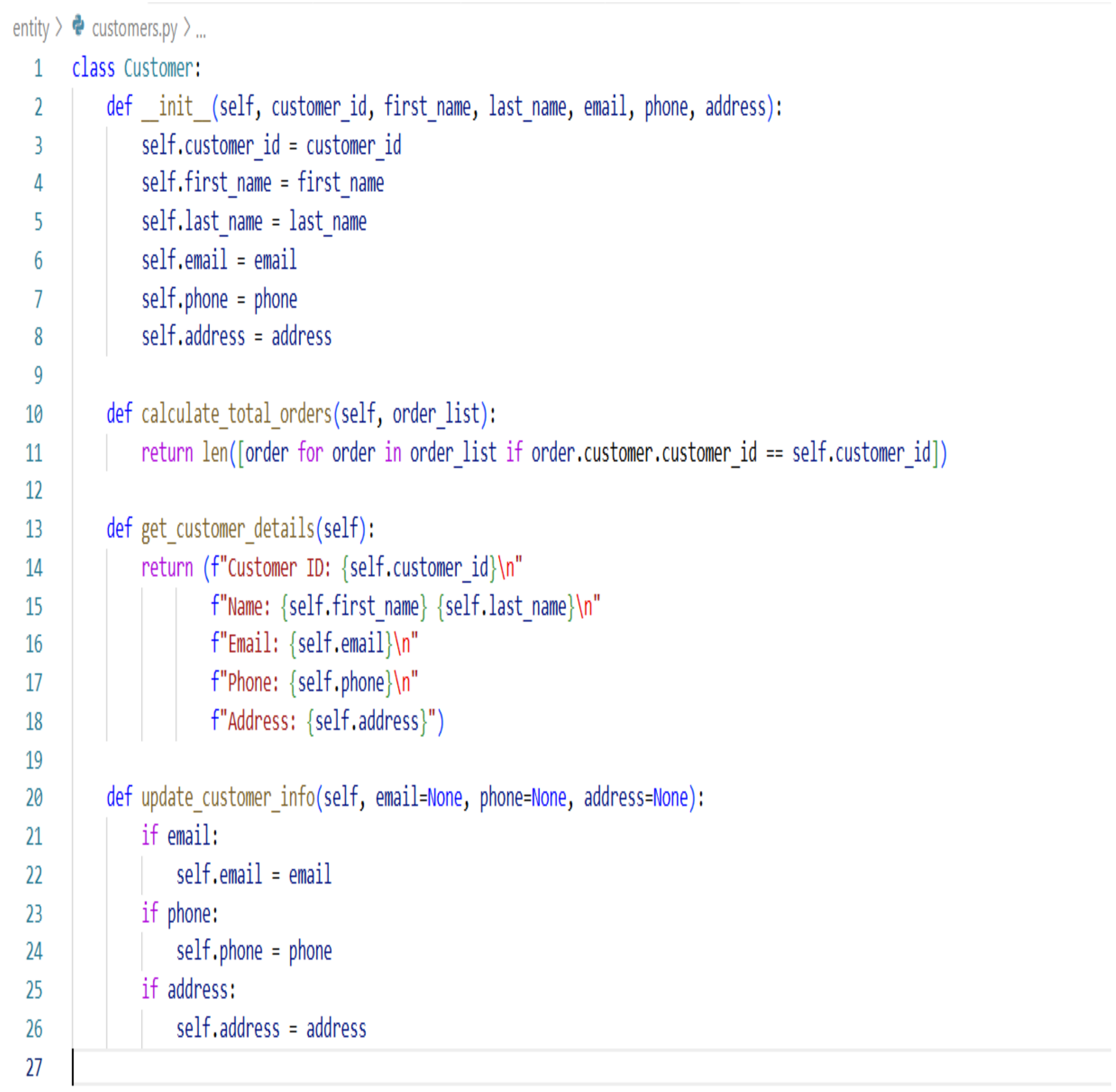
```python
from datetime import datetime

class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock):
        self.inventory_id = inventory_id
        self.product = product  # Composition
        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()
        else:
            raise ValueError("Not enough stock to remove.")

    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.quantity_in_stock * self.product.price

    def list_low_stock_products(self, threshold):
        return self.product.get_product_details() if self.quantity_in_stock < threshold else None

    def list_out_of_stock_products(self):
        return self.product.get_product_details() if self.quantity_in_stock == 0 else None

    def list_all_products(self):
        return f"{self.product.product_name}: {self.quantity_in_stock} in stock"

```

Task 2: Class Creation:

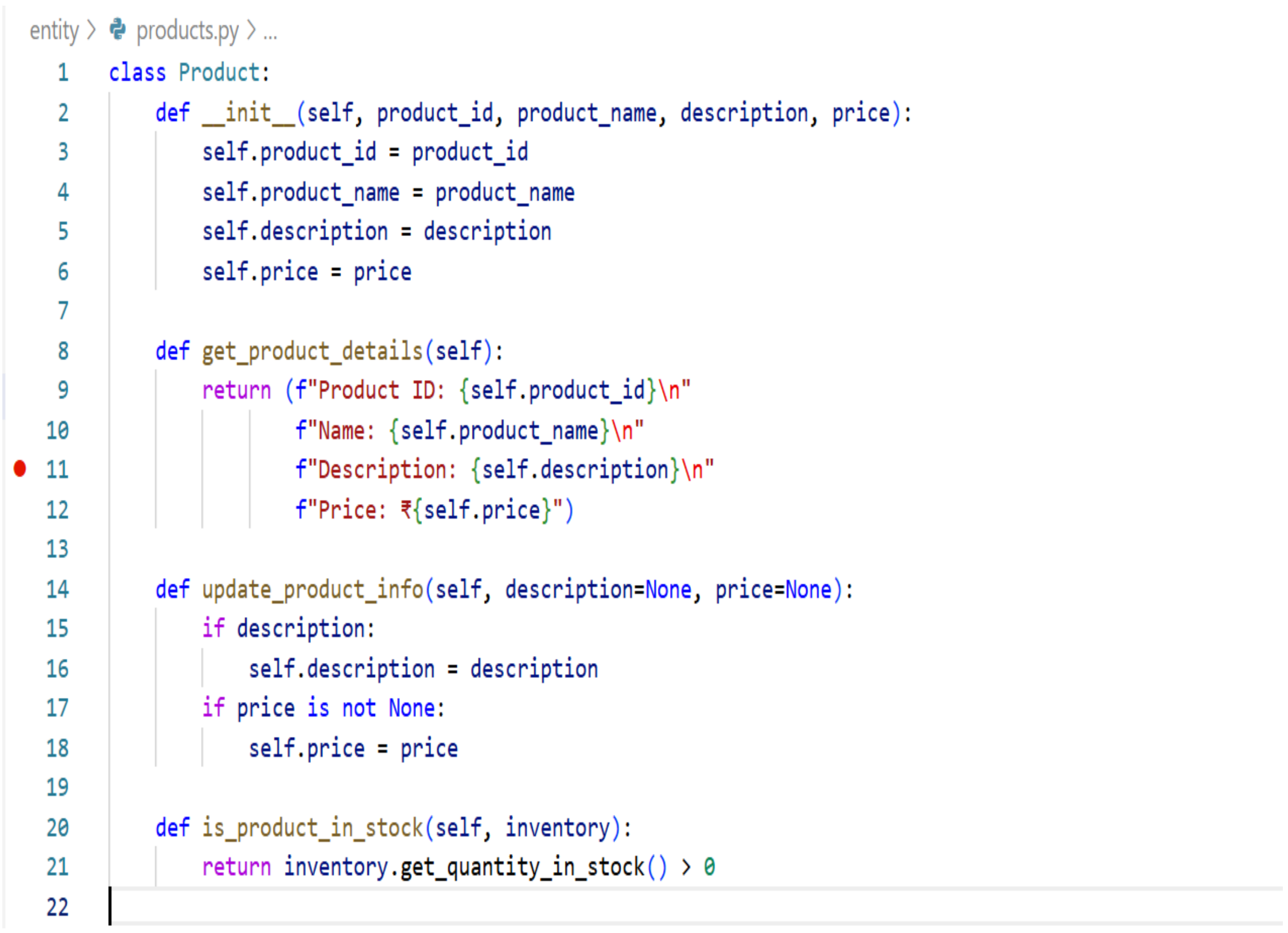
• Create the classes (Customers, Products, Orders, OrderDetails and Inventory) with the specified attributes.

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

• Implement methods as specified.

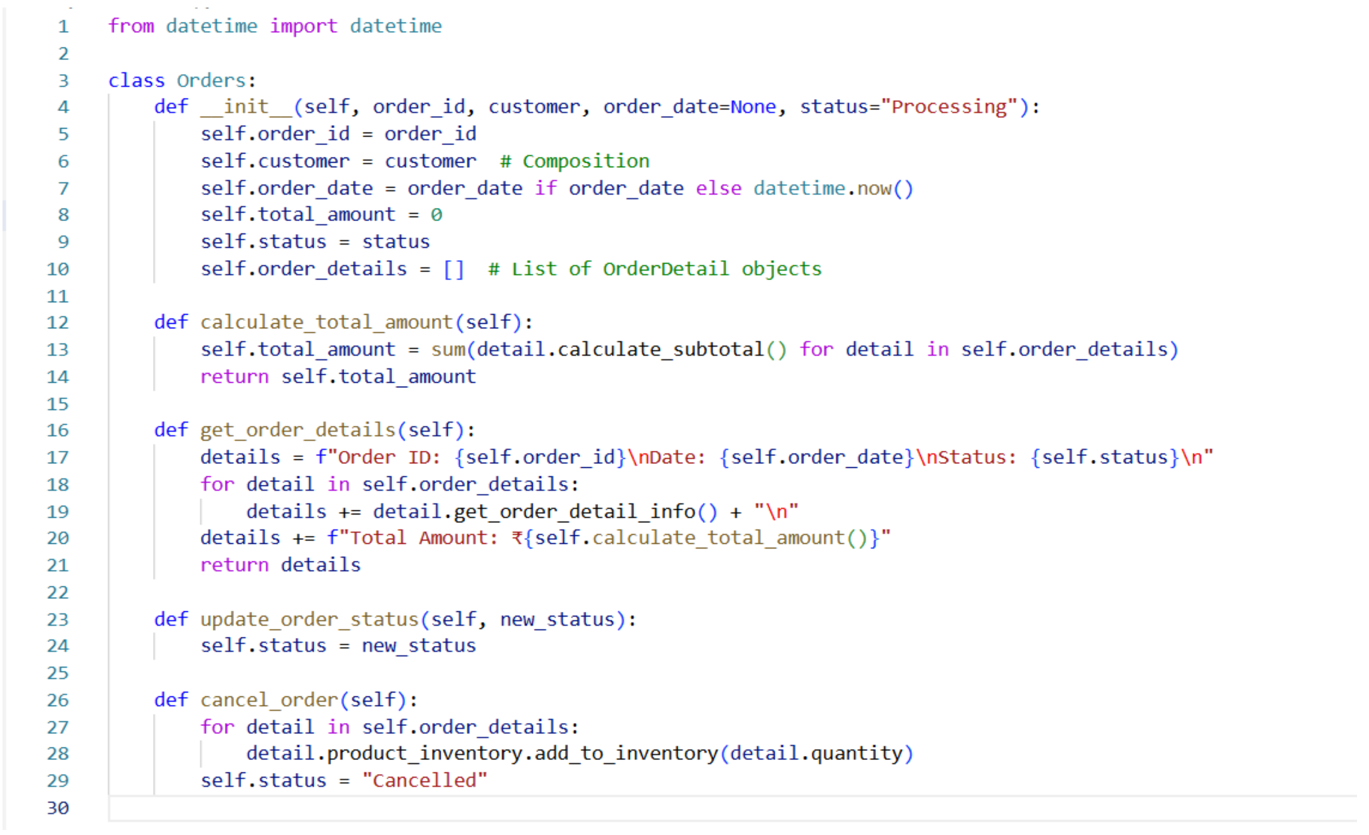


```python
class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address

    def calculate_total_orders(self, order_list):
        return len([order for order in order_list if order.customer.customer_id == self.customer_id])

    def get_customer_details(self):
        return (f"Customer ID: {self.customer_id}\n"
                f"Name: {self.first_name} {self.last_name}\n"
                f"Email: {self.email}\n"
                f"Phone: {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

```



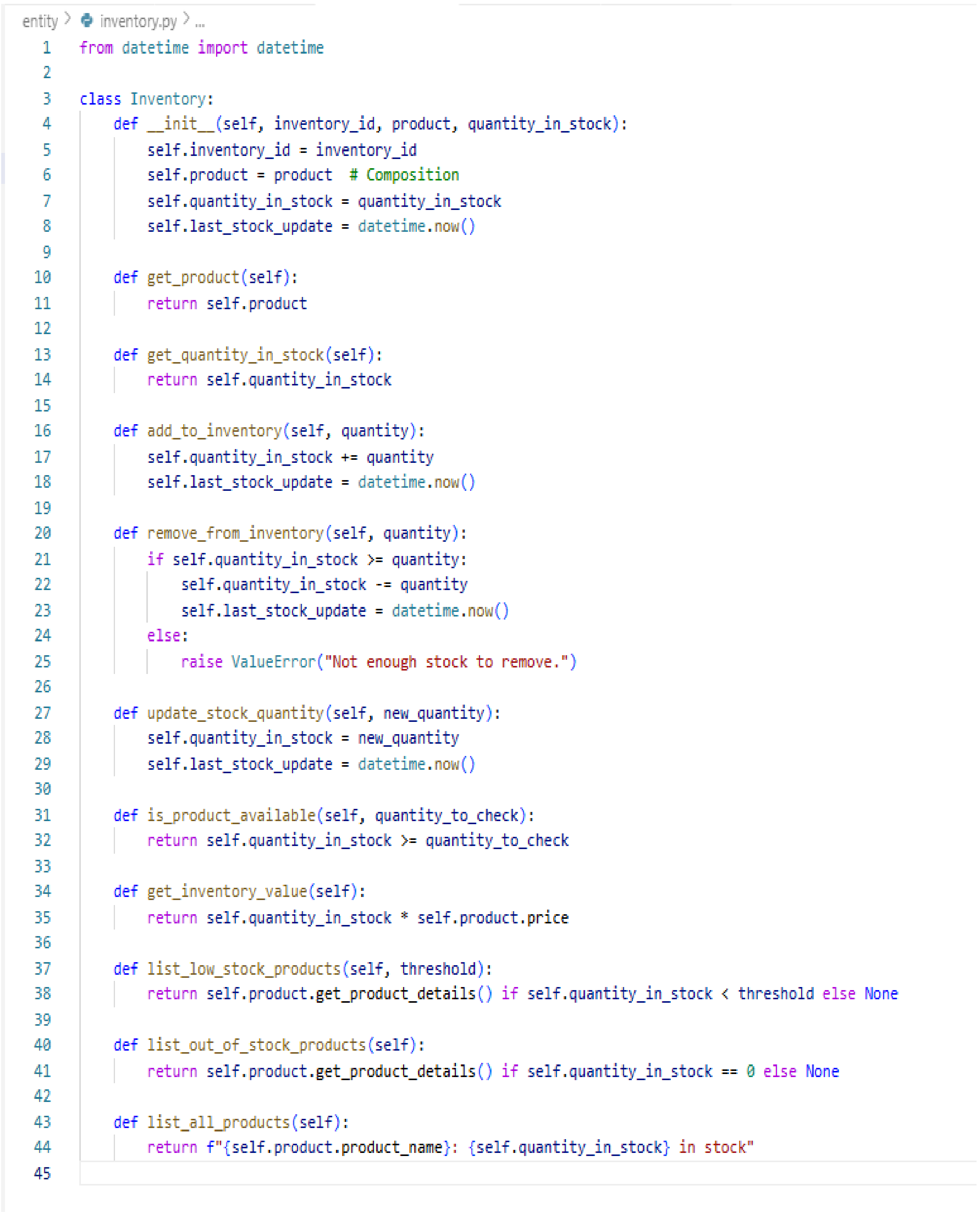
entity > products.py > ...

```python
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

    def get_product_details(self):
        return (f"Product ID: {self.product_id}\n"
                f"Name: {self.product_name}\n"
                f"Description: {self.description}\n"
                f"Price: ₹{self.price}")

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

    def is_product_in_stock(self, inventory):
        return inventory.get_quantity_in_stock() > 0

```



```python
from datetime import datetime

class Orders:
    def __init__(self, order_id, customer, order_date=None, status="Processing"):
        self.order_id = order_id
        self.customer = customer  # Composition
        self.order_date = order_date if order_date else datetime.now()
        self.total_amount = 0
        self.status = status
        self.order_details = []  # List of OrderDetail objects

    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):
        details = f"Order ID: {self.order_id}\nDate: {self.order_date}\nStatus: {self.status}\n"
        for detail in self.order_details:
            details += detail.get_order_detail_info() + "\n"
        details += f"Total Amount: ₹{self.calculate_total_amount()}"
        return details

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

    def cancel_order(self):
        for detail in self.order_details:
            detail.product_inventory.add_to_inventory(detail.quantity)
        self.status = "Cancelled"

```

```python
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity, product_inventory):
        self.order_detail_id = order_detail_id
        self.order = order  # Composition
        self.product = product  # Composition
        self.quantity = quantity
        self.product_inventory = product_inventory  # Inventory object for this product
        self.discount = 0

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

    def get_order_detail_info(self):
        return (f"OrderDetail ID: {self.order_detail_id} | Product: {self.product.product_name} | "
                f"Qty: {self.quantity} | Subtotal: ₹{self.calculate_subtotal():.2f}")

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

    def add_discount(self, discount_percentage):
        self.discount = discount_percentage / 100
```

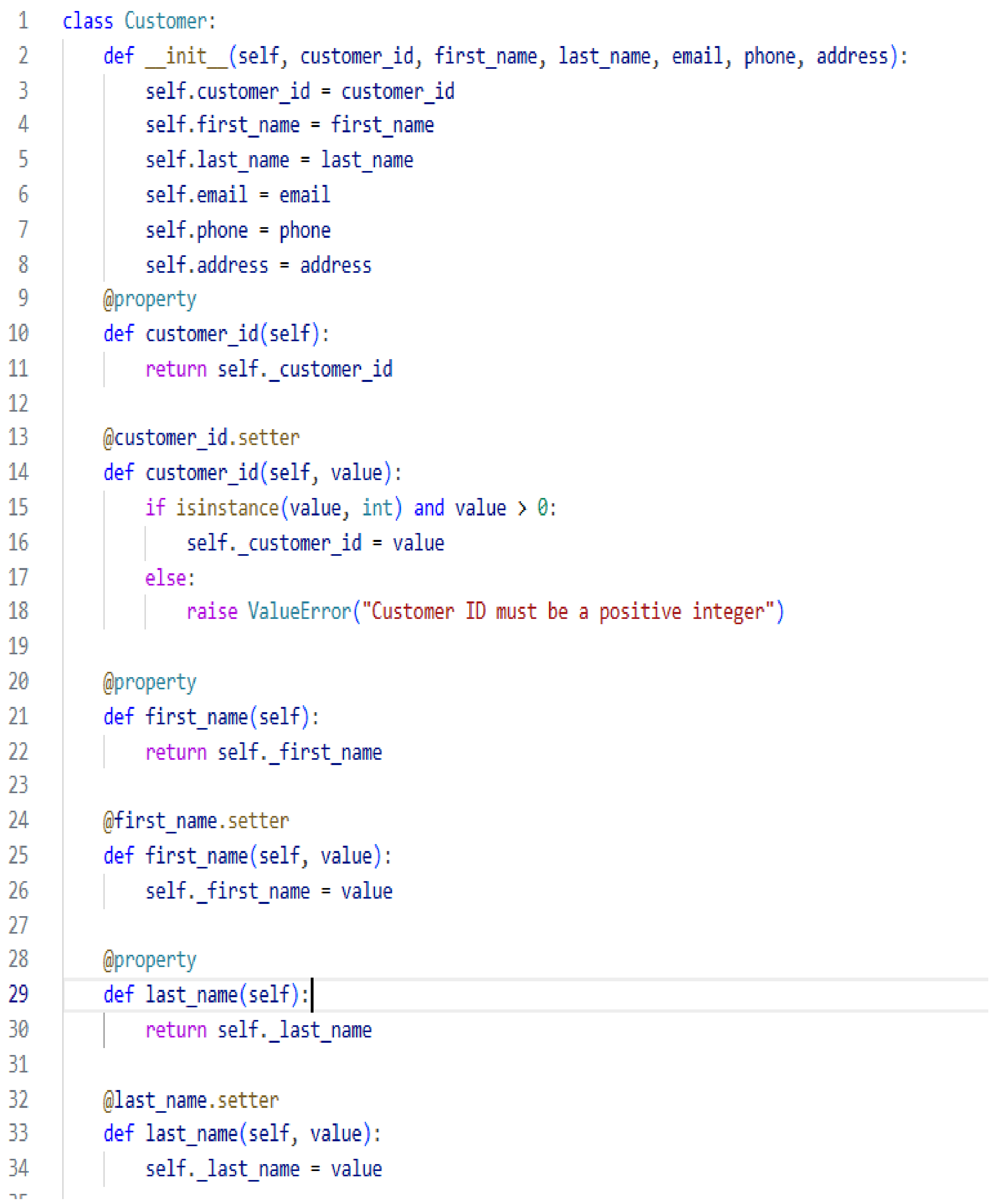


```python
from datetime import datetime

class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock):
        self.inventory_id = inventory_id
        self.product = product  # Composition
        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()
        else:
            raise ValueError("Not enough stock to remove.")

    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.quantity_in_stock * self.product.price

    def list_low_stock_products(self, threshold):
        return self.product.get_product_details() if self.quantity_in_stock < threshold else None

    def list_out_of_stock_products(self):
        return self.product.get_product_details() if self.quantity_in_stock == 0 else None

    def list_all_products(self):
        return f"{self.product.product_name}: {self.quantity_in_stock} in stock"

```

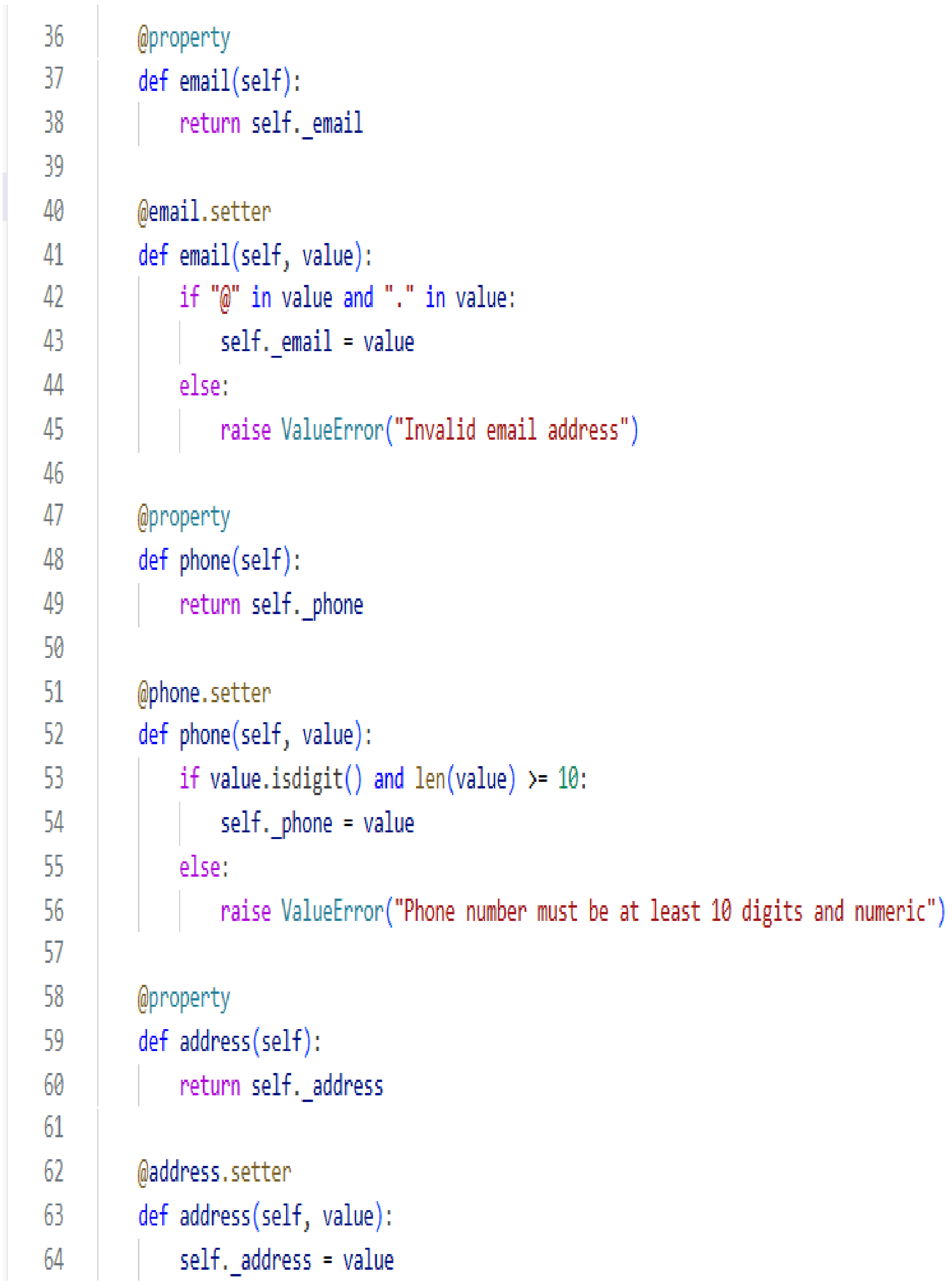
Task 3: Encapsulation:

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

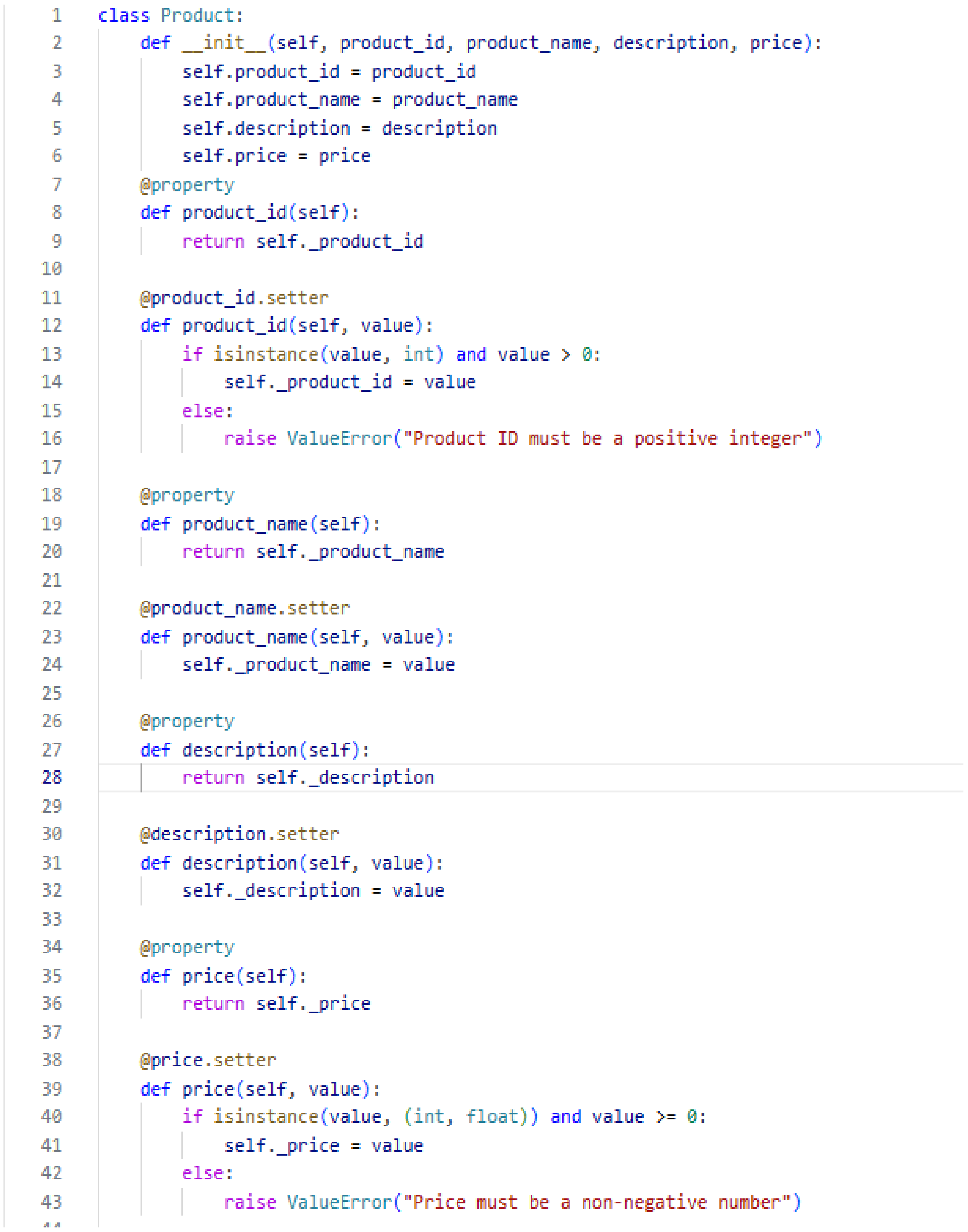
• Add data validation logic to setter methods (e.g., ensure that prices are non-negative, quantities are positive integers).



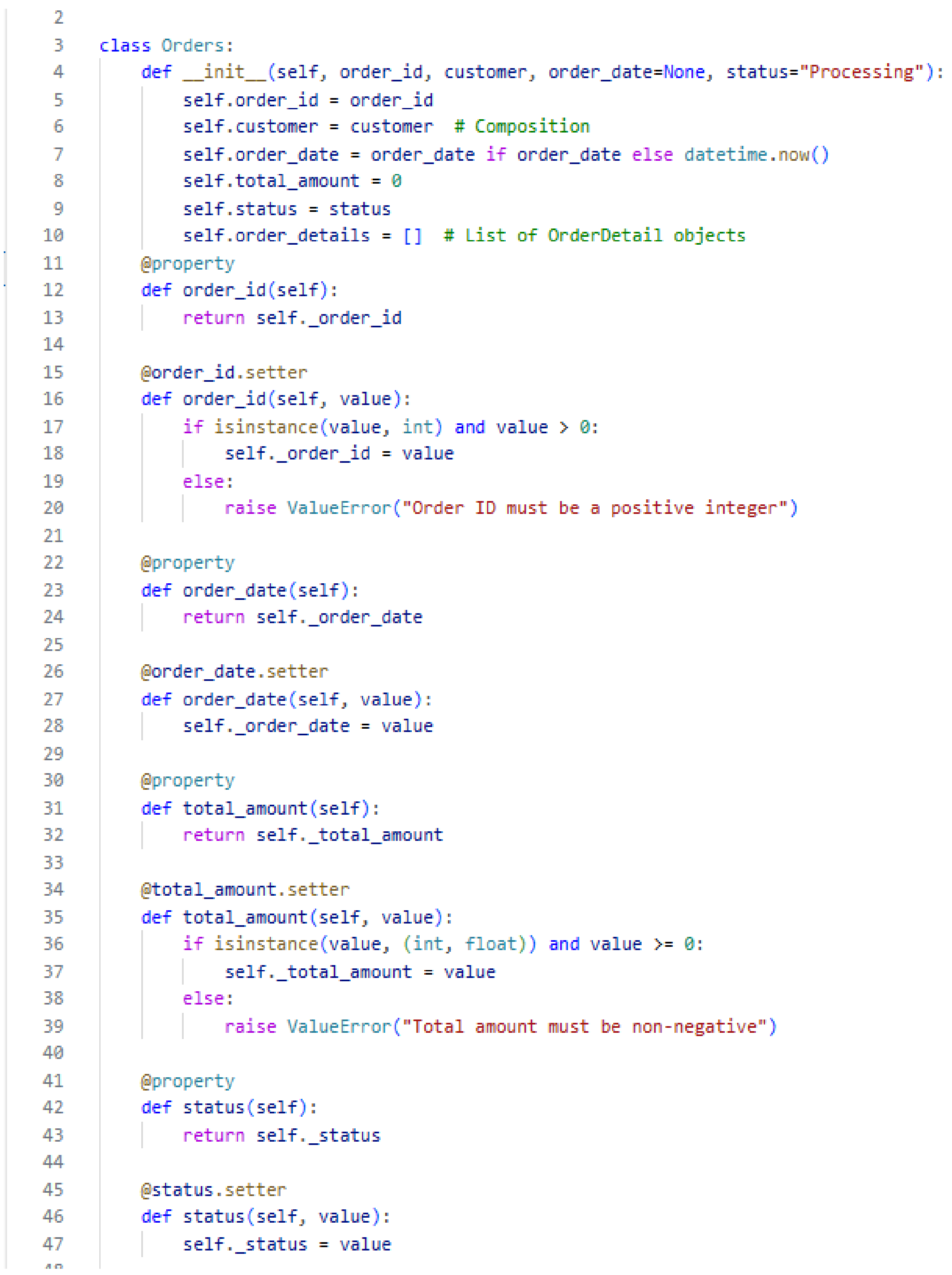
```python
class Customer:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address
    @property
    def customer_id(self):
        return self._customer_id

    @customer_id.setter
    def customer_id(self, value):
        if isinstance(value, int) and value > 0:
            self._customer_id = value
        else:
            raise ValueError("Customer ID must be a positive integer")

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

    @first_name.setter
    def first_name(self, value):
        self._first_name = value

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

    @last_name.setter
    def last_name(self, value):
        self._last_name = value
```



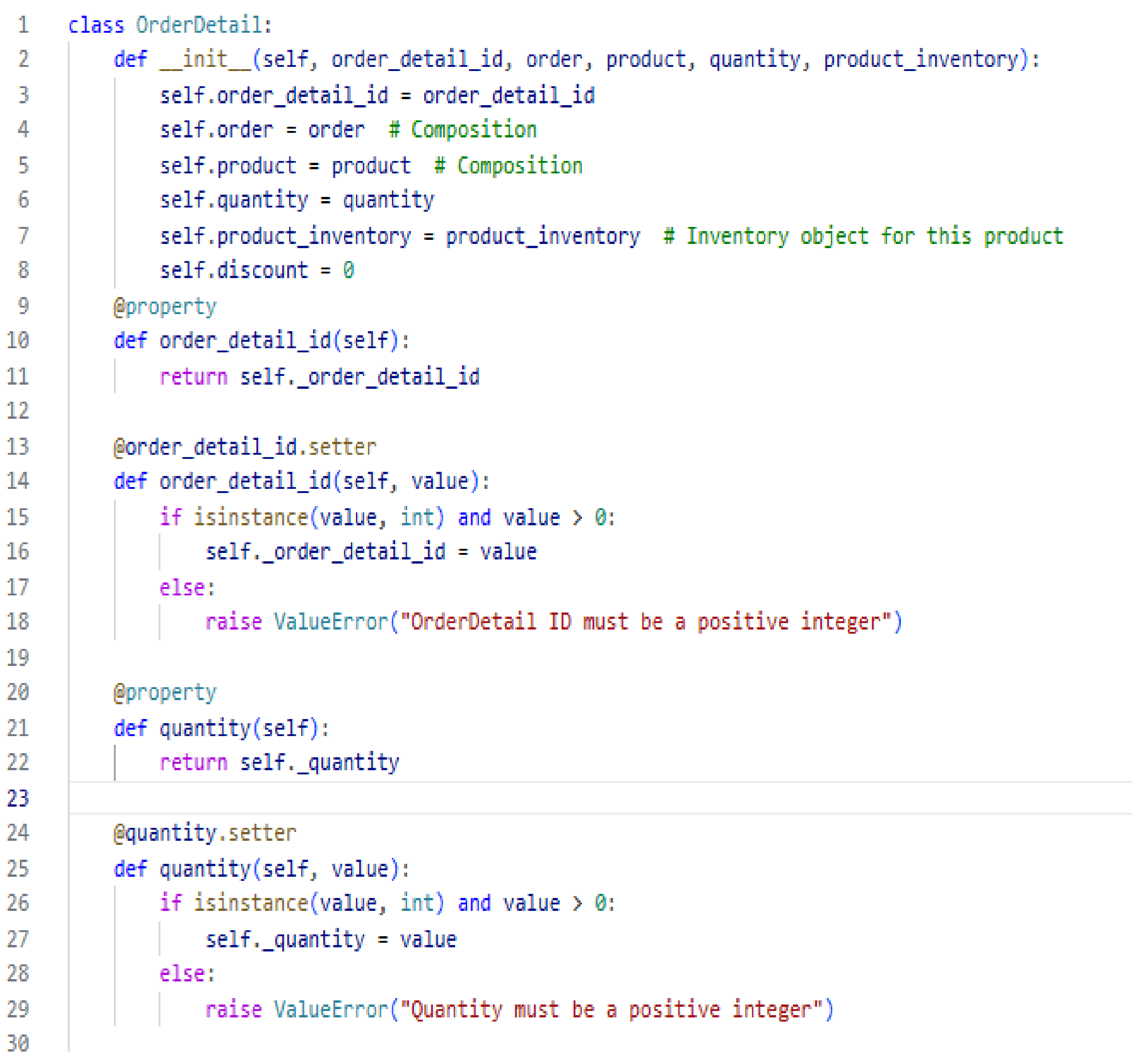
```python
    @property
    def email(self):
        return self._email

    @email.setter
    def email(self, value):
        if "@" in value and "." in value:
            self._email = value
        else:
            raise ValueError("Invalid email address")

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

    @phone.setter
    def phone(self, value):
        if value.isdigit() and len(value) >= 10:
            self._phone = value
        else:
            raise ValueError("Phone number must be at least 10 digits and numeric")

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

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



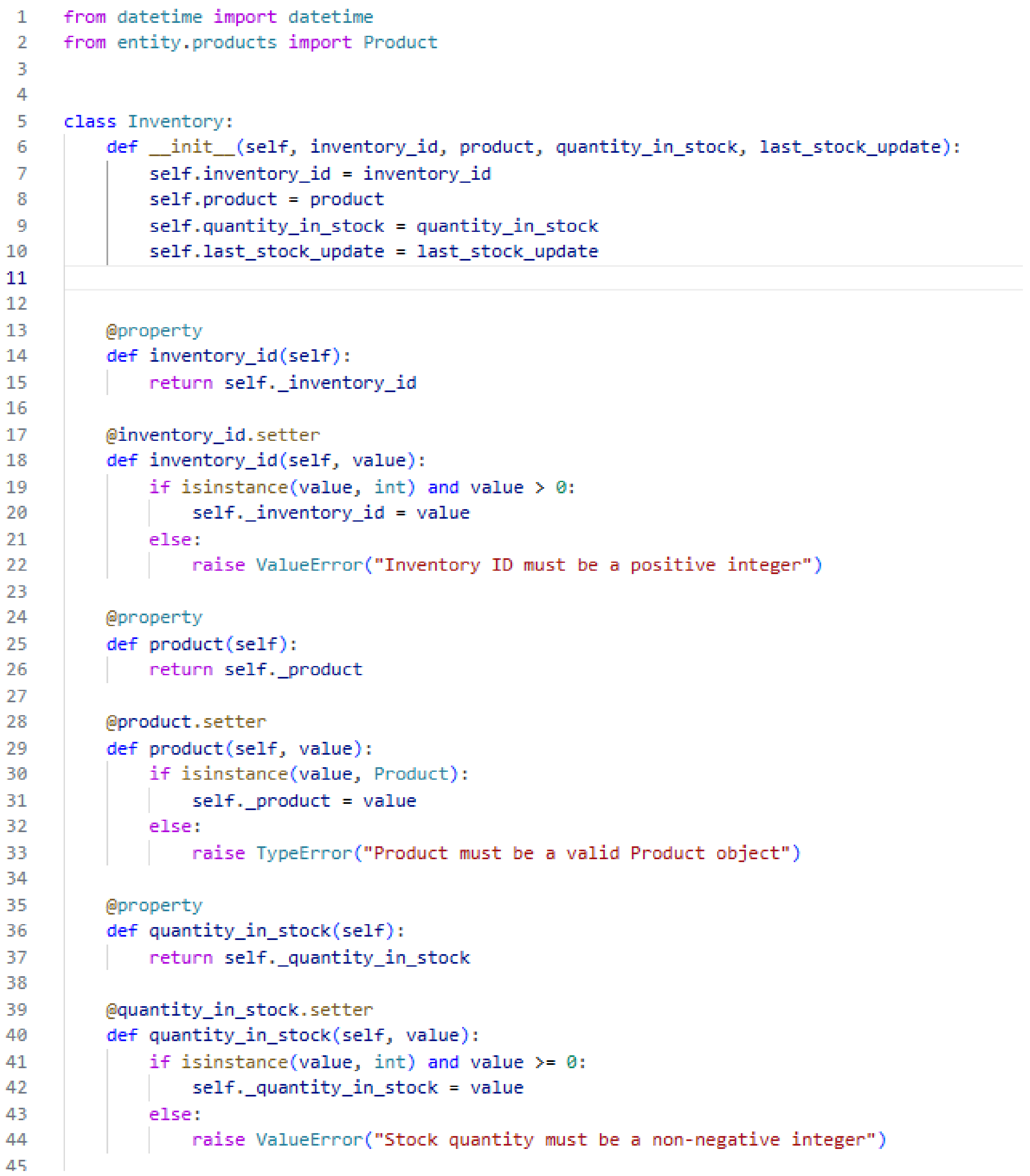
```python
class Product:
    def __init__(self, product_id, product_name, description, price):
        self.product_id = product_id
        self.product_name = product_name
        self.description = description
        self.price = price
    @property
    def product_id(self):
        return self._product_id

    @product_id.setter
    def product_id(self, value):
        if isinstance(value, int) and value > 0:
            self._product_id = value
        else:
            raise ValueError("Product ID must be a positive integer")

    @property
    def product_name(self):
        return self._product_name

    @product_name.setter
    def product_name(self, value):
        self._product_name = 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 isinstance(value, (int, float)) and value >= 0:
            self._price = value
        else:
            raise ValueError("Price must be a non-negative number")
```



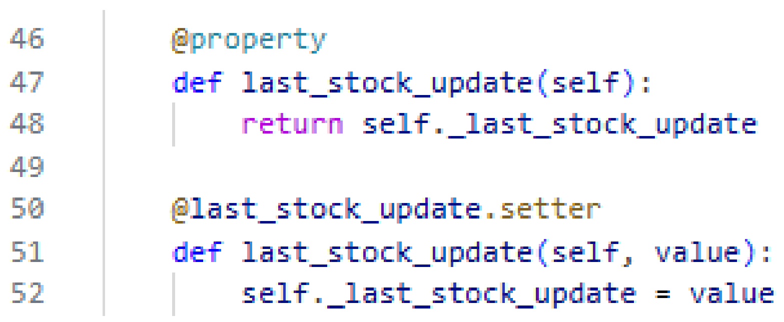
```python

class Orders:
    def __init__(self, order_id, customer, order_date=None, status="Processing"):
        self.order_id = order_id
        self.customer = customer  # Composition
        self.order_date = order_date if order_date else datetime.now()
        self.total_amount = 0
        self.status = status
        self.order_details = []  # List of OrderDetail objects
    @property
    def order_id(self):
        return self._order_id

    @order_id.setter
    def order_id(self, value):
        if isinstance(value, int) and value > 0:
            self._order_id = value
        else:
            raise ValueError("Order ID must be a positive integer")

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

    @order_date.setter
    def order_date(self, value):
        self._order_date = value

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

    @total_amount.setter
    def total_amount(self, value):
        if isinstance(value, (int, float)) and value >= 0:
            self._total_amount = value
        else:
            raise ValueError("Total amount must be non-negative")

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

    @status.setter
    def status(self, value):
        self._status = value
```



```python
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity, product_inventory):
        self.order_detail_id = order_detail_id
        self.order = order  # Composition
        self.product = product  # Composition
        self.quantity = quantity
        self.product_inventory = product_inventory  # Inventory object for this product
        self.discount = 0
    @property
    def order_detail_id(self):
        return self._order_detail_id

    @order_detail_id.setter
    def order_detail_id(self, value):
        if isinstance(value, int) and value > 0:
            self._order_detail_id = value
        else:
            raise ValueError("OrderDetail ID must be a positive integer")

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

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

```



```python
from datetime import datetime
from entity.products import Product


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


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

    @inventory_id.setter
    def inventory_id(self, value):
        if isinstance(value, int) and value > 0:
            self._inventory_id = value
        else:
            raise ValueError("Inventory ID must be a positive integer")

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

    @product.setter
    def product(self, value):
        if isinstance(value, Product):
            self._product = value
        else:
            raise TypeError("Product must be a valid Product object")

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

    @quantity_in_stock.setter
    def quantity_in_stock(self, value):
        if isinstance(value, int) and value >= 0:
            self._quantity_in_stock = value
        else:
            raise ValueError("Stock quantity must be a non-negative integer")

```



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

    @last_stock_update.setter
    def last_stock_update(self, value):
        self._last_stock_update = value
```

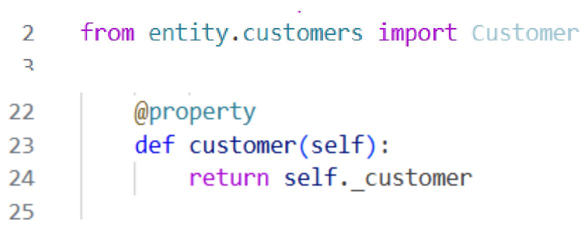
Task 4: Composition:

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.

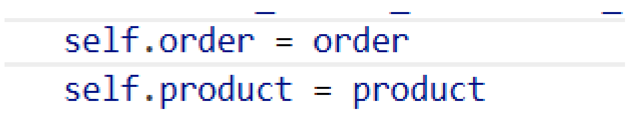


```python
from entity.customers import Customer

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

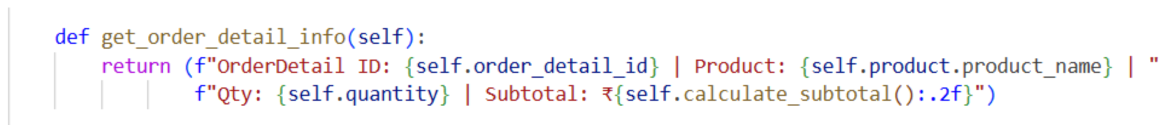
• OrderDetails Class with Composition:

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.



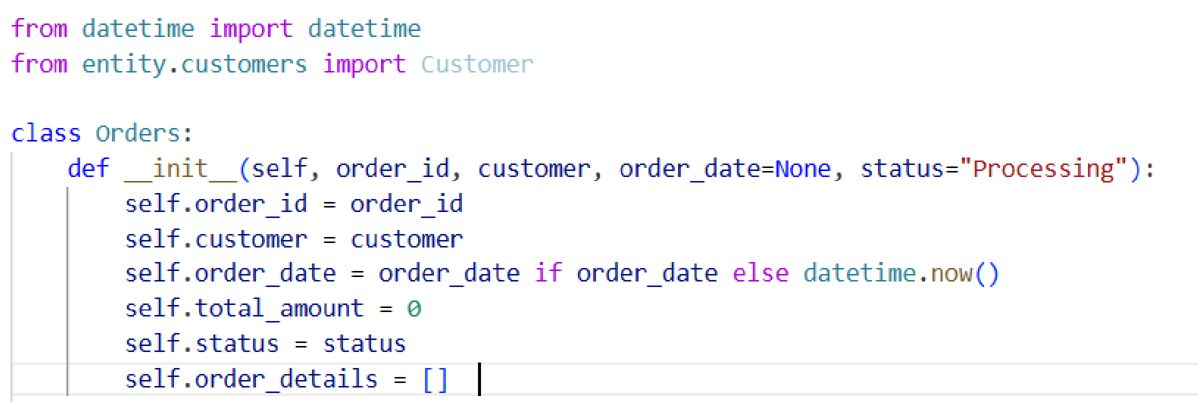
```python
self.order = order
self.product = product
```



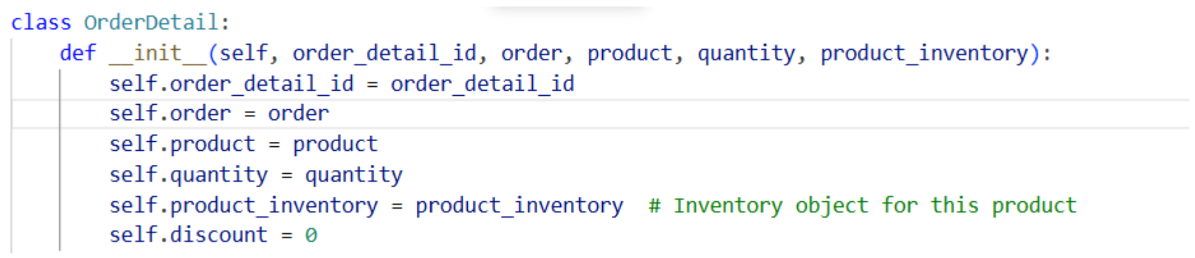
```python
def get_order_detail_info(self):
    return (f"OrderDetail ID: {self.order_detail_id} | Product: {self.product.product_name} | "
            f"Qty: {self.quantity} | Subtotal: ₹{self.calculate_subtotal():.2f}")
```

• **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.



```python
from datetime import datetime
from entity.customers import Customer

class Orders:
    def __init__(self, order_id, customer, order_date=None, status="Processing"):
        self.order_id = order_id
        self.customer = customer
        self.order_date = order_date if order_date else datetime.now()
        self.total_amount = 0
        self.status = status
        self.order_details = []
```

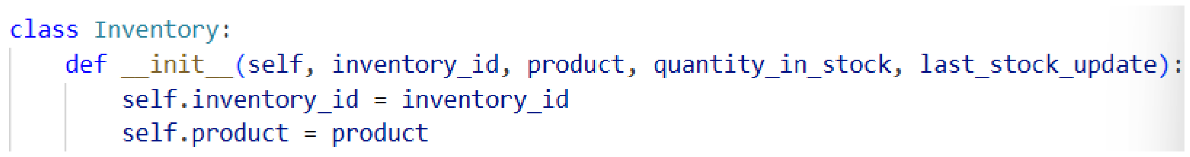


```python
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity, product_inventory):
        self.order_detail_id = order_detail_id
        self.order = order
        self.product = product
        self.quantity = quantity
        self.product_inventory = product_inventory  # Inventory object for this product
        self.discount = 0
```

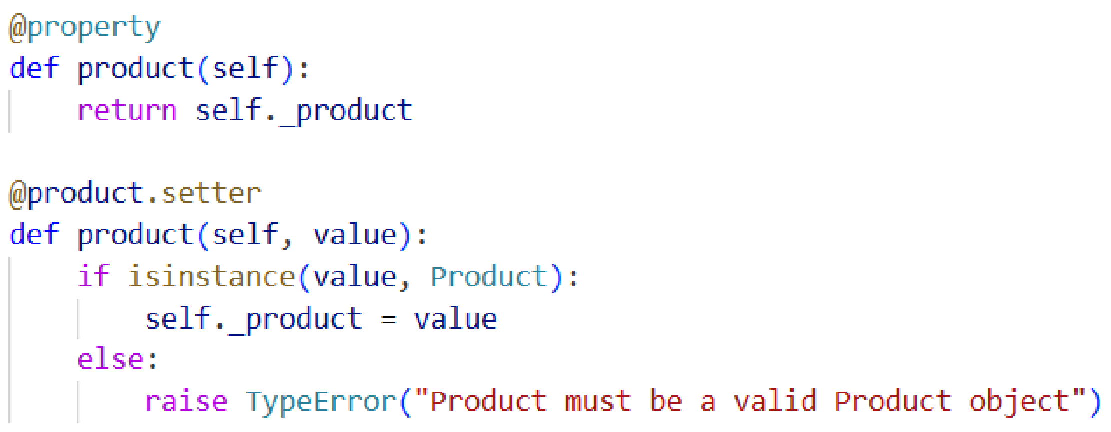
the Product class is being *used* (composed) inside the OrderDetail class

• **Inventory Class:**

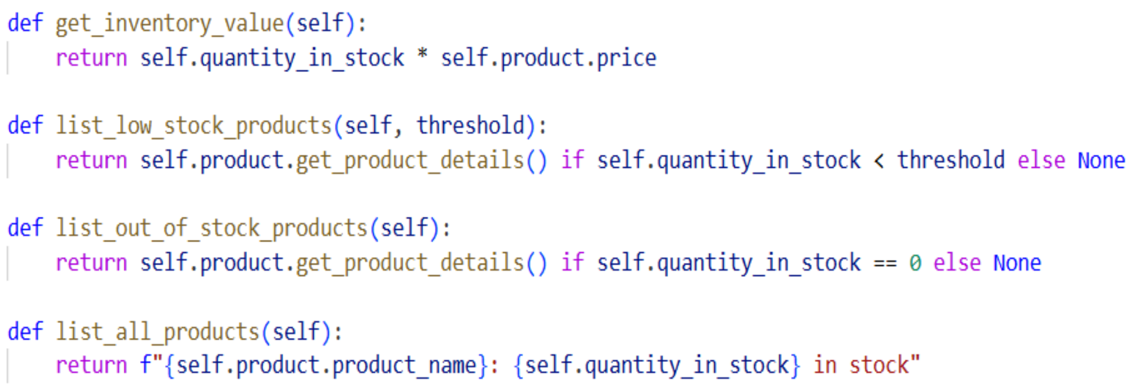
o The Inventory class represents the inventory of products available for sale. It can have composition relationships with the Products class to indicate which products are in the inventory.



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



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

@product.setter
def product(self, value):
    if isinstance(value, Product):
        self._product = value
    else:
        raise TypeError("Product must be a valid Product object")
```



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

def list_low_stock_products(self, threshold):
    return self.product.get_product_details() if self.quantity_in_stock < threshold else None

def list_out_of_stock_products(self):
    return self.product.get_product_details() if self.quantity_in_stock == 0 else None

def list_all_products(self):
    return f"{self.product.product_name}: {self.quantity_in_stock} in stock"
```

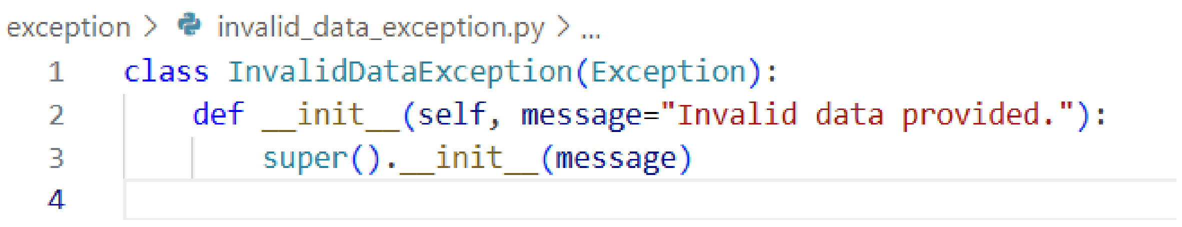
Task 5: Exceptions handling

- **Data Validation:**

o Challenge: Validate user inputs and data from external sources (e.g., user registration, order placement).

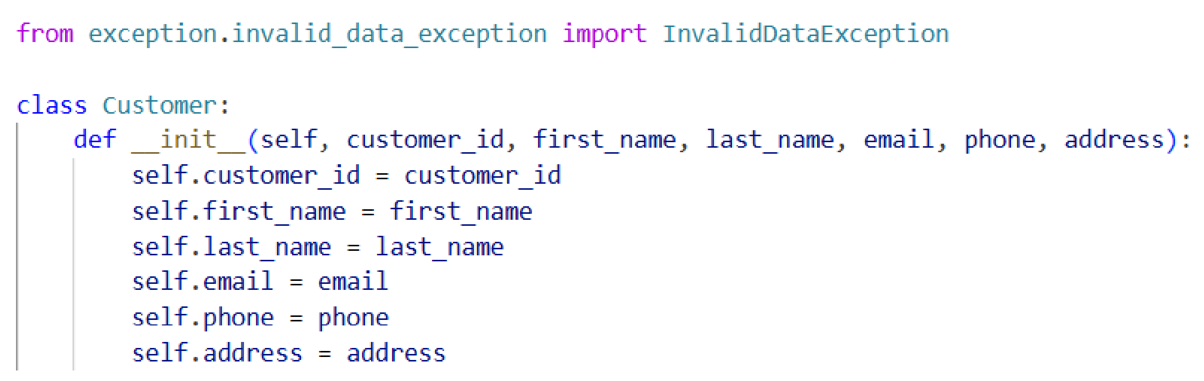
o Scenario: When a user enters an invalid email address during registration.

o Exception Handling: Throw a custom InvalidDataException with a clear error message.

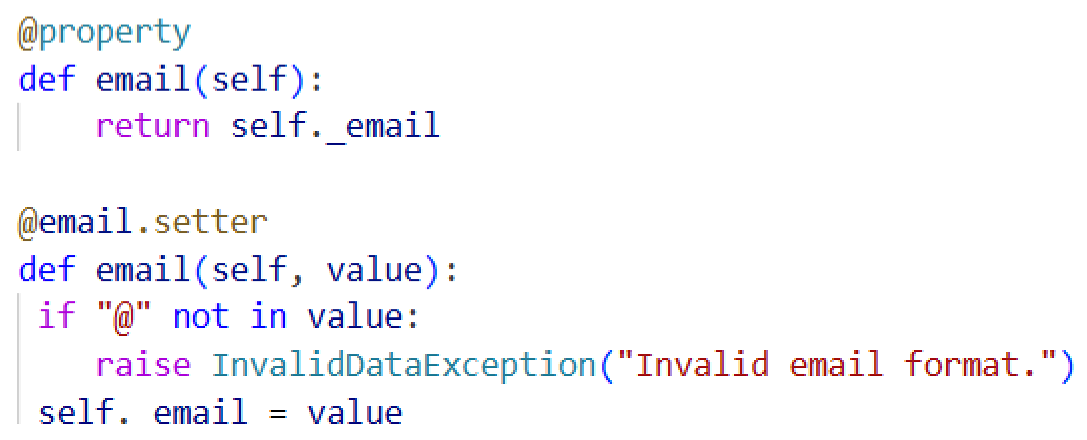


exception > invalid_data_exception.py > ...

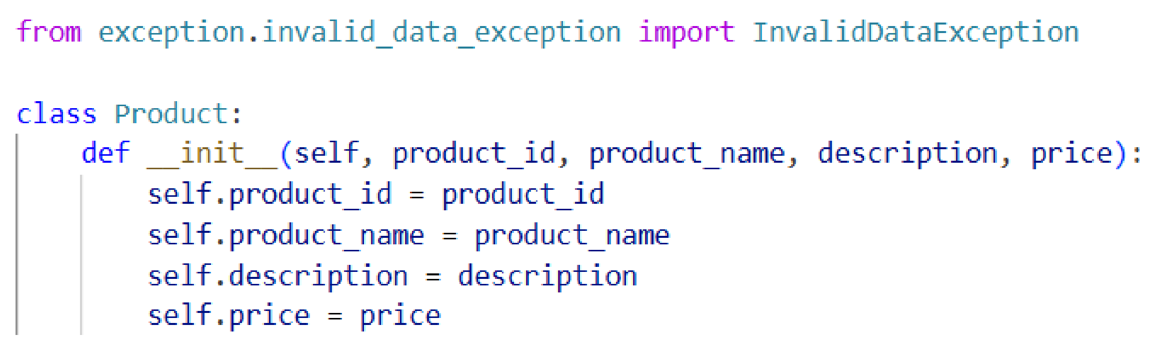
```python
class InvalidDataException(Exception):
    def __init__(self, message="Invalid data provided."):
        super().__init__(message)

```



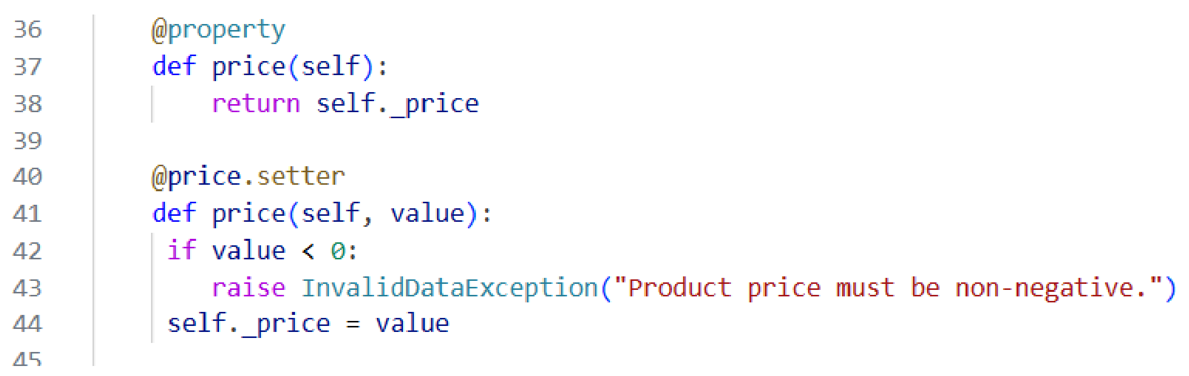
```
from exception.invalid_data_exception import InvalidDataException

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
```



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

    @email.setter
    def email(self, value):
     if "@" not in value:
        raise InvalidDataException("Invalid email format.")
     self. email = value
```



```
from exception.invalid_data_exception import InvalidDataException

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
```



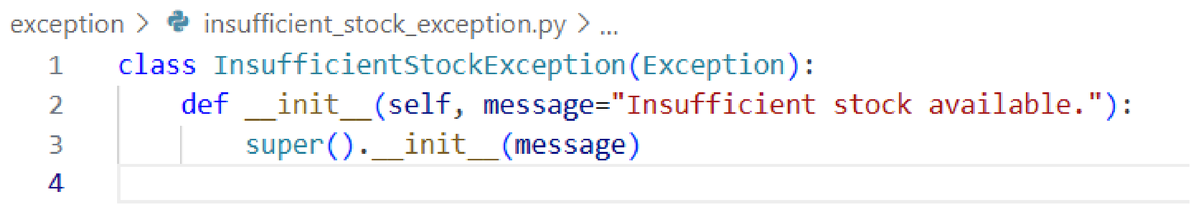
```
    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
     if value < 0:
        raise InvalidDataException("Product price must be non-negative.")
     self._price = value
```

• **Inventory Management:**

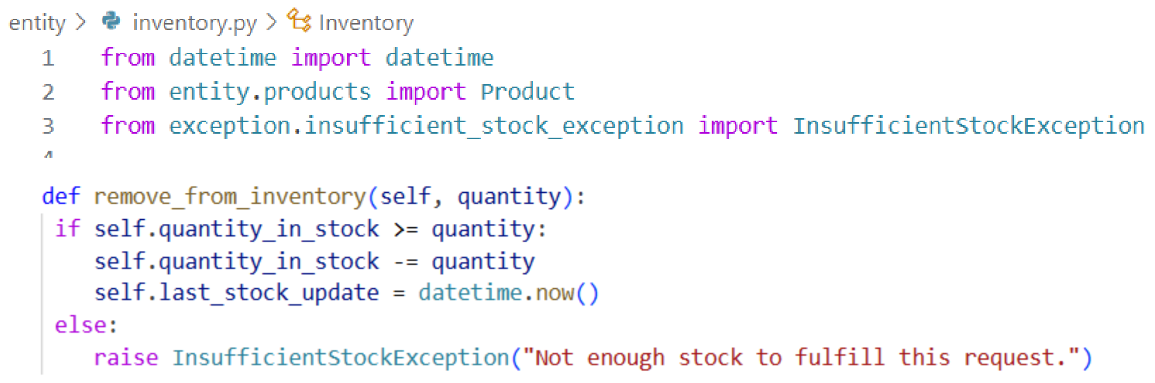
o Challenge: Handling inventory-related issues, such as selling more products than are in stock.

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.



```python
class InsufficientStockException(Exception):
    def __init__(self, message="Insufficient stock available."):
        super().__init__(message)

```



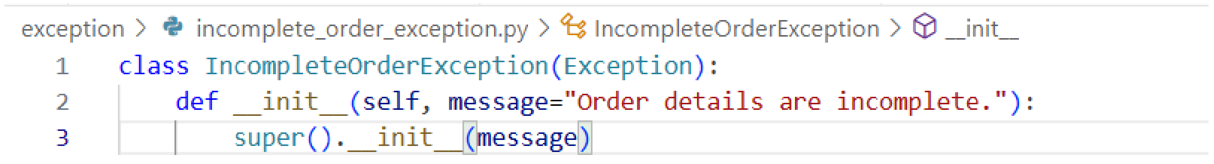
```python
from datetime import datetime
from entity.products import Product
from exception.insufficient_stock_exception import InsufficientStockException

def remove_from_inventory(self, quantity):
    if self.quantity_in_stock >= quantity:
        self.quantity_in_stock -= quantity
        self.last_stock_update = datetime.now()
    else:
        raise InsufficientStockException("Not enough stock to fulfill this request.")
```

• **Order Processing:**

o Challenge: Ensuring the order details are consistent and complete before processing.

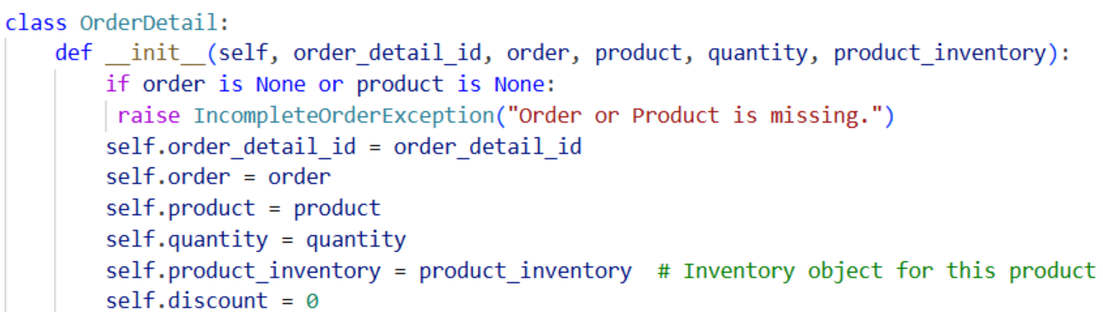
o Scenario: When an order detail lacks a product reference.

o Exception Handling: Throw an IncompleteOrderException with a message explaining the issue.


exception > incomplete_order_exception.py > IncompleteOrderException > __init__

```python
class IncompleteOrderException(Exception):
    def __init__(self, message="Order details are incomplete."):
        super().__init__(message)
```

```python
from exception.incomplete_order_exception import IncompleteOrderException
```



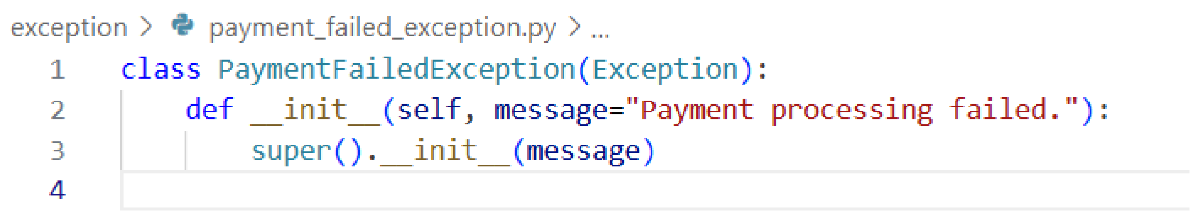
```python
class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity, product_inventory):
        if order is None or product is None:
            raise IncompleteOrderException("Order or Product is missing.")
        self.order_detail_id = order_detail_id
        self.order = order
        self.product = product
        self.quantity = quantity
        self.product_inventory = product_inventory  # Inventory object for this product
        self.discount = 0
```

• **Payment Processing**:

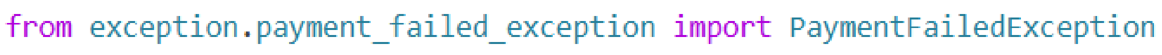
o Challenge: Handling payment failures or declined transactions.

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

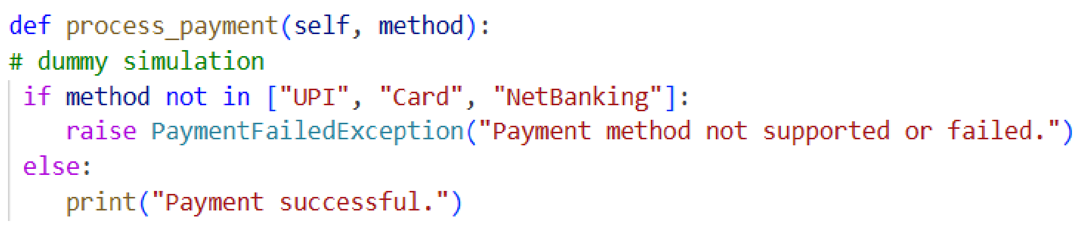
o Exception Handling: Handle payment-specific exceptions (e.g., PaymentFailedException) and initiate retry or cancellation processes.



```python
class PaymentFailedException(Exception):
    def __init__(self, message="Payment processing failed."):
        super().__init__(message)
```



```python
from exception.payment_failed_exception import PaymentFailedException
```



```python
def process_payment(self, method):
# dummy simulation
 if method not in ["UPI", "Card", "NetBanking"]:
    raise PaymentFailedException("Payment method not supported or failed.")
 else:
    print("Payment successful.")
```

• **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).

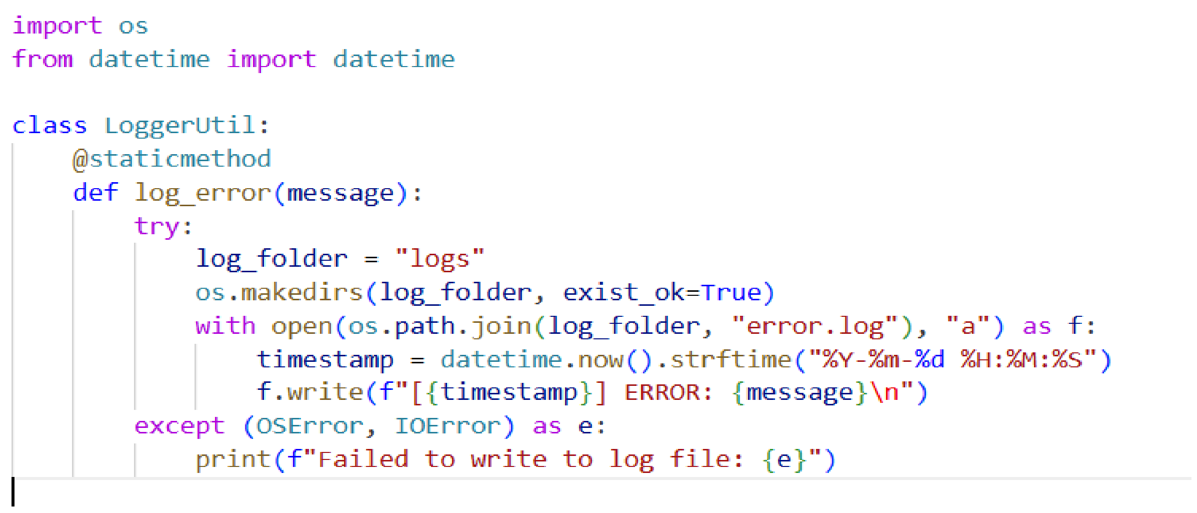
o Exception Handling: Handle file I/O exceptions (e.g., IOException) and log them appropriately.

File I/O Exception Handling (Logging):

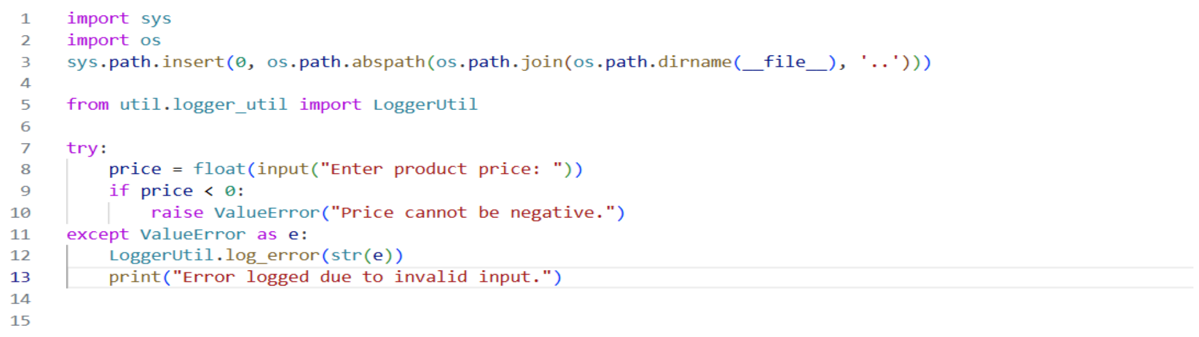
- We created a utility class `LoggerUtil` under the `util/` package to handle error logging into a log file.

- The application uses Python's built-in exceptions like `FileNotFoundError` and `IOError` to catch file-related issues.

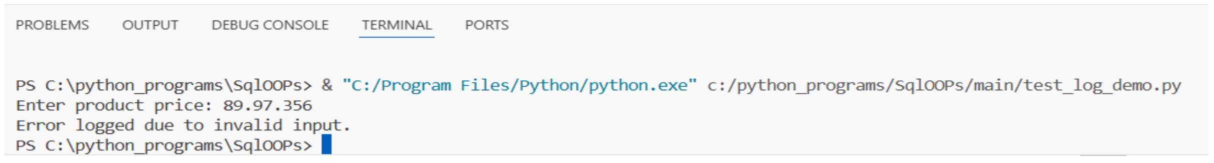
- In case of file write failure (e.g., missing folder or permission issue), the system catches the error and prints a fallback message.



```python
import os
from datetime import datetime

class LoggerUtil:
    @staticmethod
    def log_error(message):
        try:
            log_folder = "logs"
            os.makedirs(log_folder, exist_ok=True)
            with open(os.path.join(log_folder, "error.log"), "a") as f:
                timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                f.write(f"[{timestamp}] ERROR: {message}\n")
        except (OSError, IOError) as e:
            print(f"Failed to write to log file: {e}")
```



```python
import sys
import os
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from util.logger_util import LoggerUtil

try:
    price = float(input("Enter product price: "))
    if price < 0:
        raise ValueError("Price cannot be negative.")
except ValueError as e:
    LoggerUtil.log_error(str(e))
    print("Error logged due to invalid input.")
```



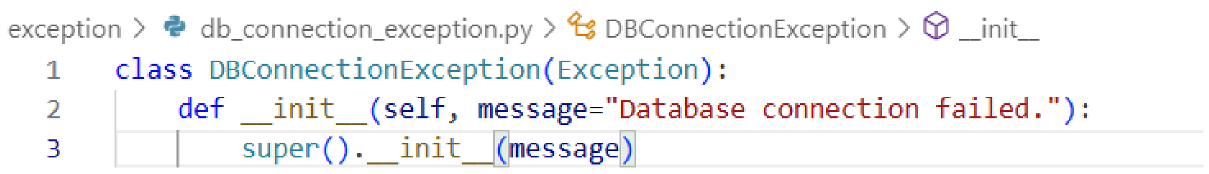
```
PS C:\python_programs\SqlOOPs> & "C:/Program Files/Python/python.exe" c:/python_programs/SqlOOPs/main/test_log_demo.py
Enter product price: 89.97.356
Error logged due to invalid input.
PS C:\python_programs\SqlOOPs> 
```

• **Database Access:**

o Challenge: Managing database connections and queries.

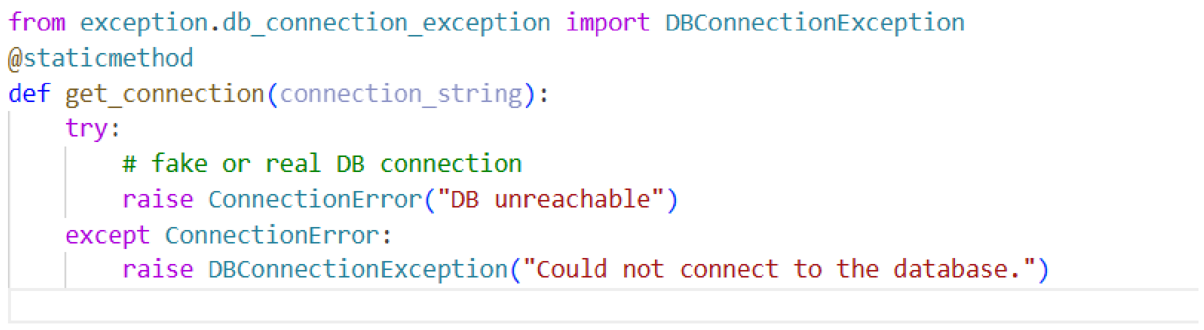
o Scenario: When executing a SQL query and the database is offline.

o Exception Handling: Handle database-specific exceptions (e.g., SqlException) and implement connection retries or failover mechanisms.



exception > db_connection_exception.py > DBConnectionException > __init__

```python
class DBConnectionException(Exception):
    def __init__(self, message="Database connection failed."):
        super().__init__(message)
```



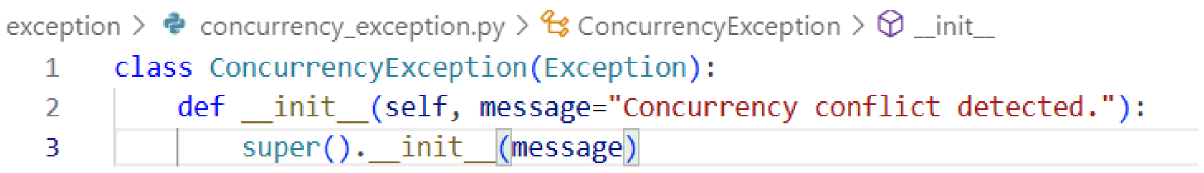
```python
from exception.db_connection_exception import DBConnectionException
@staticmethod
def get_connection(connection_string):
    try:
        # fake or real DB connection
        raise ConnectionError("DB unreachable")
    except ConnectionError:
        raise DBConnectionException("Could not connect to the database.")
```

• **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.



exception > concurrency_exception.py > ConcurrencyException > __init__

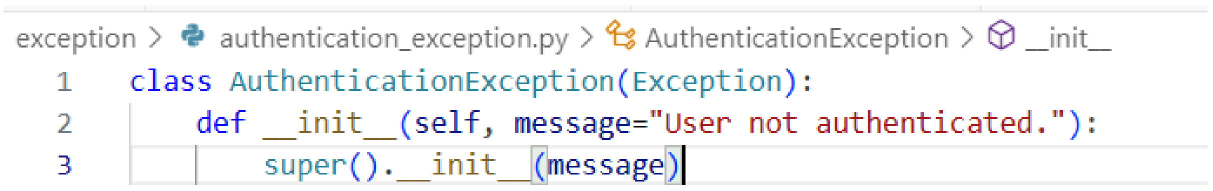
```python
class ConcurrencyException(Exception):
    def __init__(self, message="Concurrency conflict detected."):
        super().__init__(message)
```

• **Security and Authentication:**

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

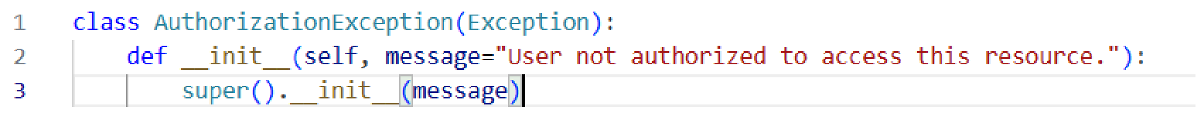
o Scenario: When a user tries to access sensitive information without proper authentication.

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



exception > authentication_exception.py > AuthenticationException > __init__

```python
class AuthenticationException(Exception):
    def __init__(self, message="User not authenticated."):
        super().__init__(message)
```



```python
class AuthorizationException(Exception):
    def __init__(self, message="User not authorized to access this resource."):
        super().__init__(message)
```

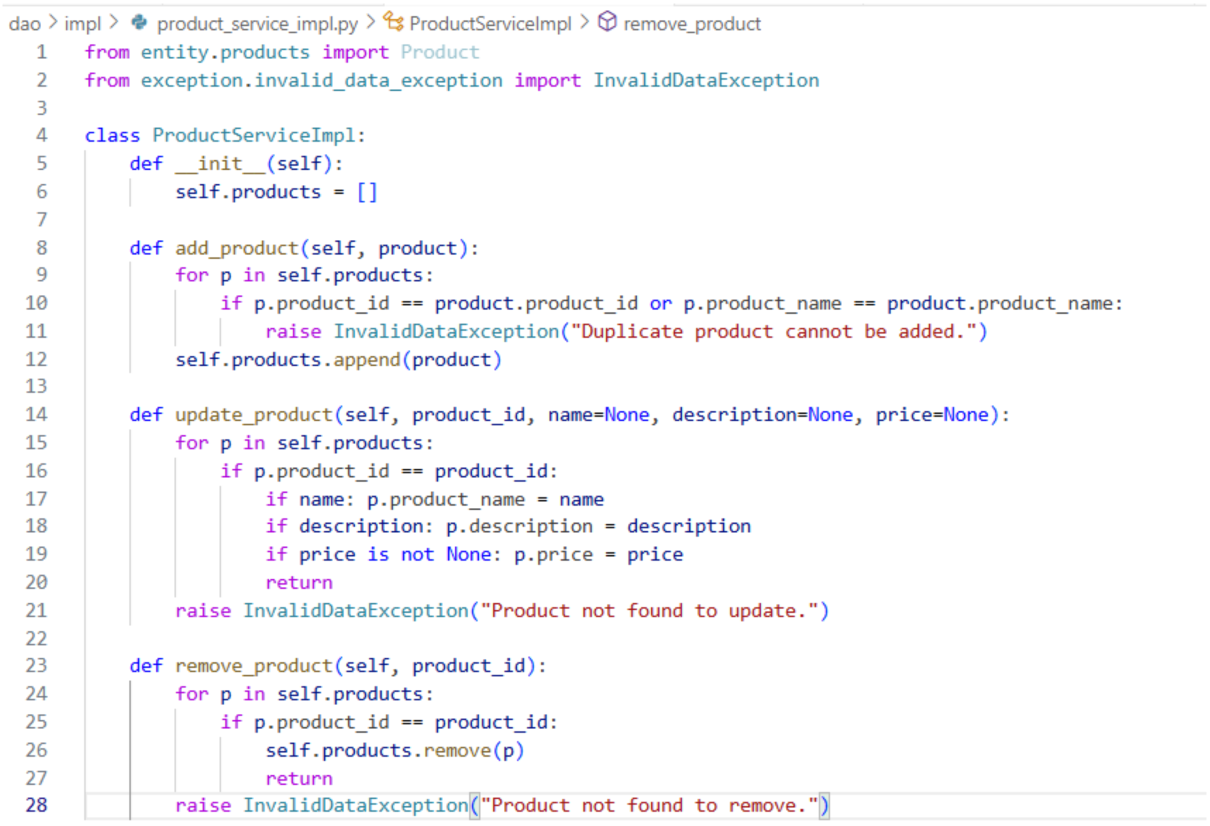
Task 6: Collections

• 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.



dao > impl > product_service_impl.py > ProductServiceImpl > remove_product

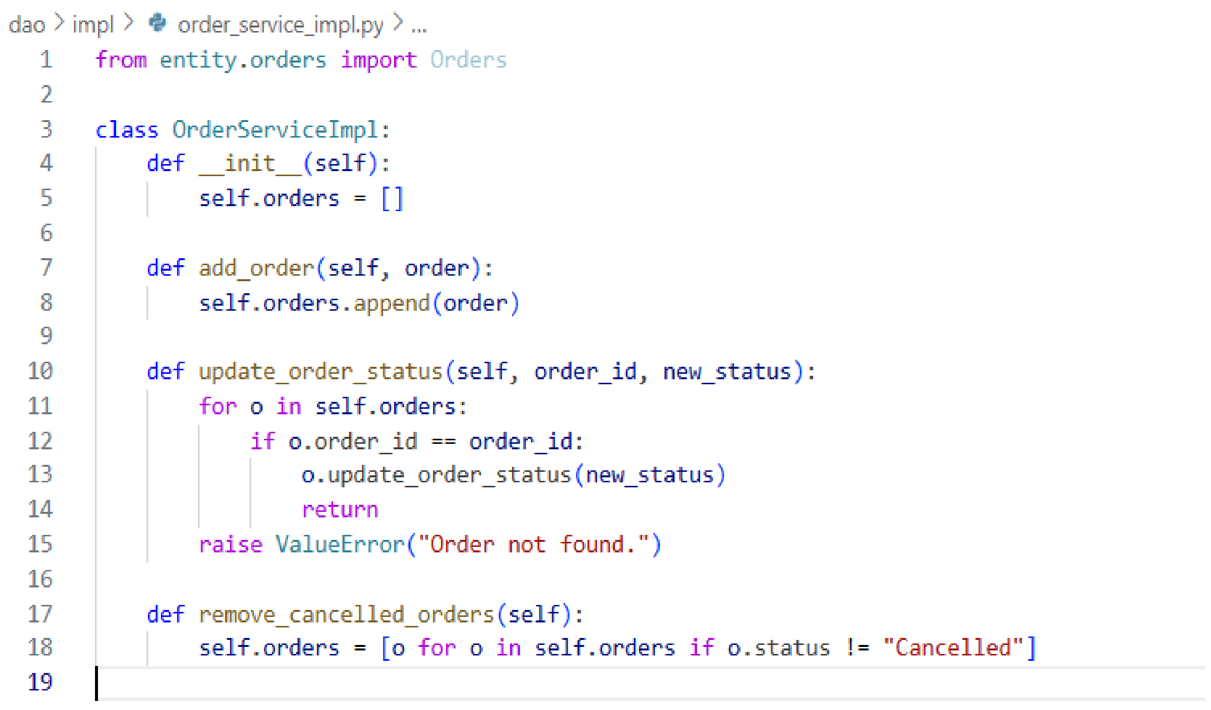
```python
from entity.products import Product
from exception.invalid_data_exception import InvalidDataException

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

    def add_product(self, product):
        for p in self.products:
            if p.product_id == product.product_id or p.product_name == product.product_name:
                raise InvalidDataException("Duplicate product cannot be added.")
        self.products.append(product)

    def update_product(self, product_id, name=None, description=None, price=None):
        for p in self.products:
            if p.product_id == product_id:
                if name: p.product_name = name
                if description: p.description = description
                if price is not None: p.price = price
                return
        raise InvalidDataException("Product not found to update.")

    def remove_product(self, product_id):
        for p in self.products:
            if p.product_id == product_id:
                self.products.remove(p)
                return
        raise InvalidDataException("Product not found to remove.")
```

• **Managing Orders List:**

o 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.



dao > impl > order_service_impl.py > ...

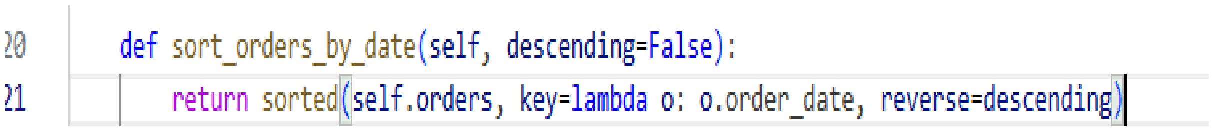
```python
from entity.orders import Orders

class OrderServiceImpl:
    def __init__(self):
        self.orders = []

    def add_order(self, order):
        self.orders.append(order)

    def update_order_status(self, order_id, new_status):
        for o in self.orders:
            if o.order_id == order_id:
                o.update_order_status(new_status)
                return
        raise ValueError("Order not found.")

    def remove_cancelled_orders(self):
        self.orders = [o for o in self.orders if o.status != "Cancelled"]

```

• **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.



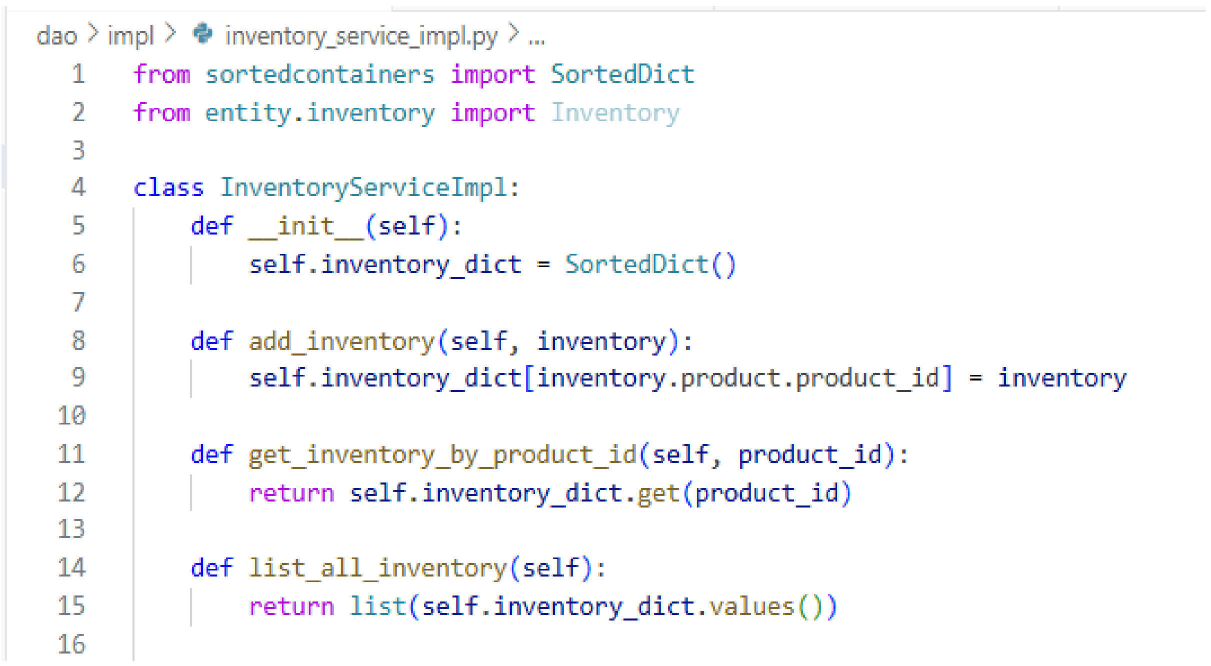
```python
    def sort_orders_by_date(self, descending=False):
        return sorted(self.orders, key=lambda o: o.order_date, reverse=descending)
```

• **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.



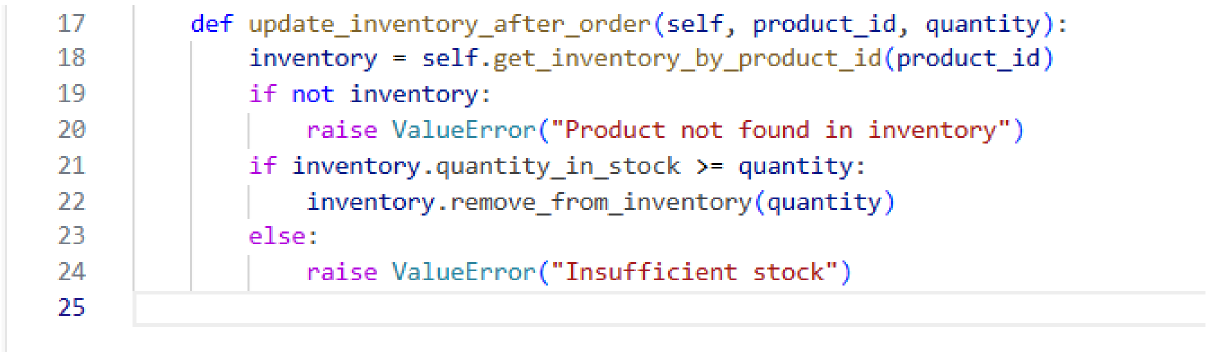
```python
from sortedcontainers import SortedDict
from entity.inventory import Inventory

class InventoryServiceImpl:
    def __init__(self):
        self.inventory_dict = SortedDict()

    def add_inventory(self, inventory):
        self.inventory_dict[inventory.product.product_id] = inventory

    def get_inventory_by_product_id(self, product_id):
        return self.inventory_dict.get(product_id)

    def list_all_inventory(self):
        return list(self.inventory_dict.values())

```

• **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.

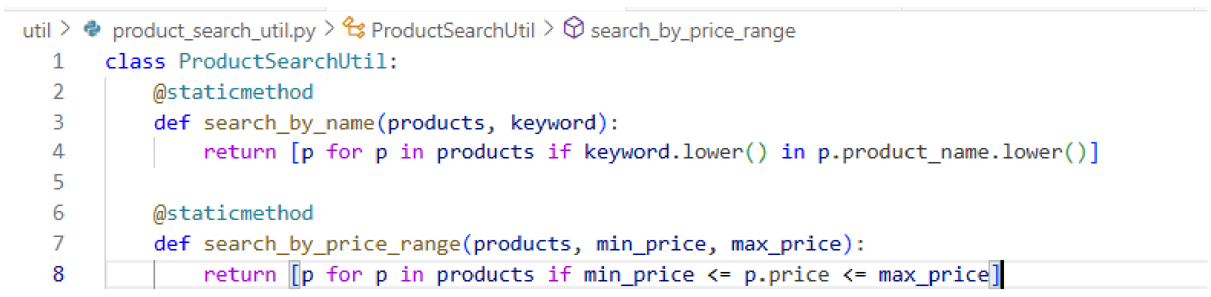


```python
    def update_inventory_after_order(self, product_id, quantity):
        inventory = self.get_inventory_by_product_id(product_id)
        if not inventory:
            raise ValueError("Product not found in inventory")
        if inventory.quantity_in_stock >= quantity:
            inventory.remove_from_inventory(quantity)
        else:
            raise ValueError("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.



util > product_search_util.py > ProductSearchUtil > search_by_price_range

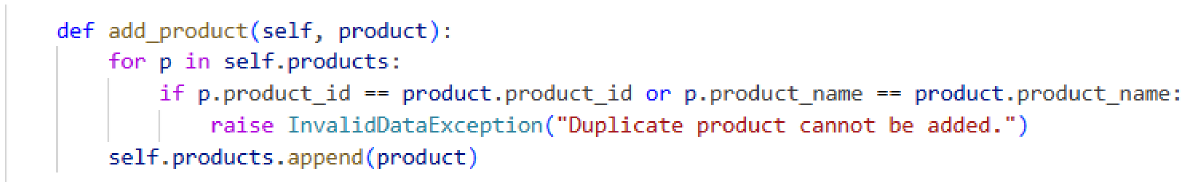
```
class ProductSearchUtil:
    @staticmethod
    def search_by_name(products, keyword):
        return [p for p in products if keyword.lower() in p.product_name.lower()]

    @staticmethod
    def search_by_price_range(products, min_price, max_price):
        return [p for p in products if min_price <= p.price <= max_price]
```

• **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.



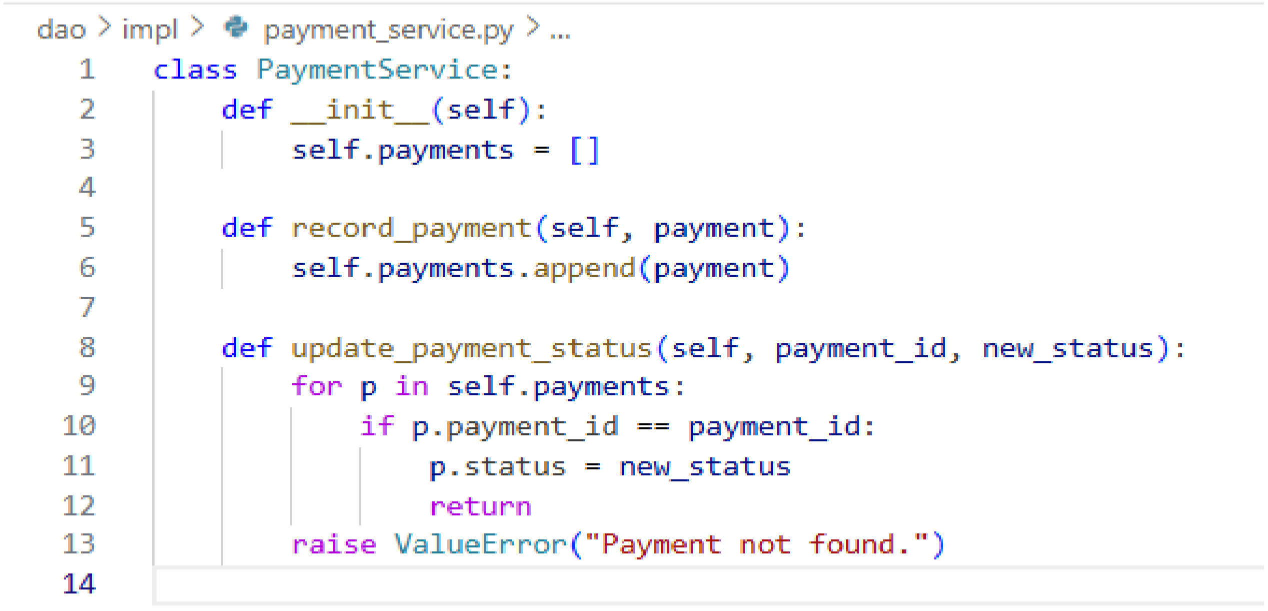
```
def add_product(self, product):
    for p in self.products:
        if p.product_id == product.product_id or p.product_name == product.product_name:
            raise InvalidDataException("Duplicate product cannot be added.")
    self.products.append(product)
```

• 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



dao > impl > payment_service.py > ...

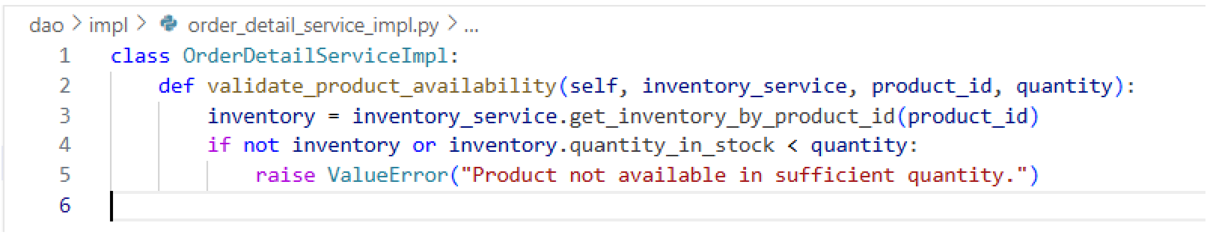
```python
class PaymentService:
    def __init__(self):
        self.payments = []

    def record_payment(self, payment):
        self.payments.append(payment)

    def update_payment_status(self, payment_id, new_status):
        for p in self.payments:
            if p.payment_id == payment_id:
                p.status = new_status
                return
        raise ValueError("Payment not found.")

```

• OrderDetails and Products :

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.



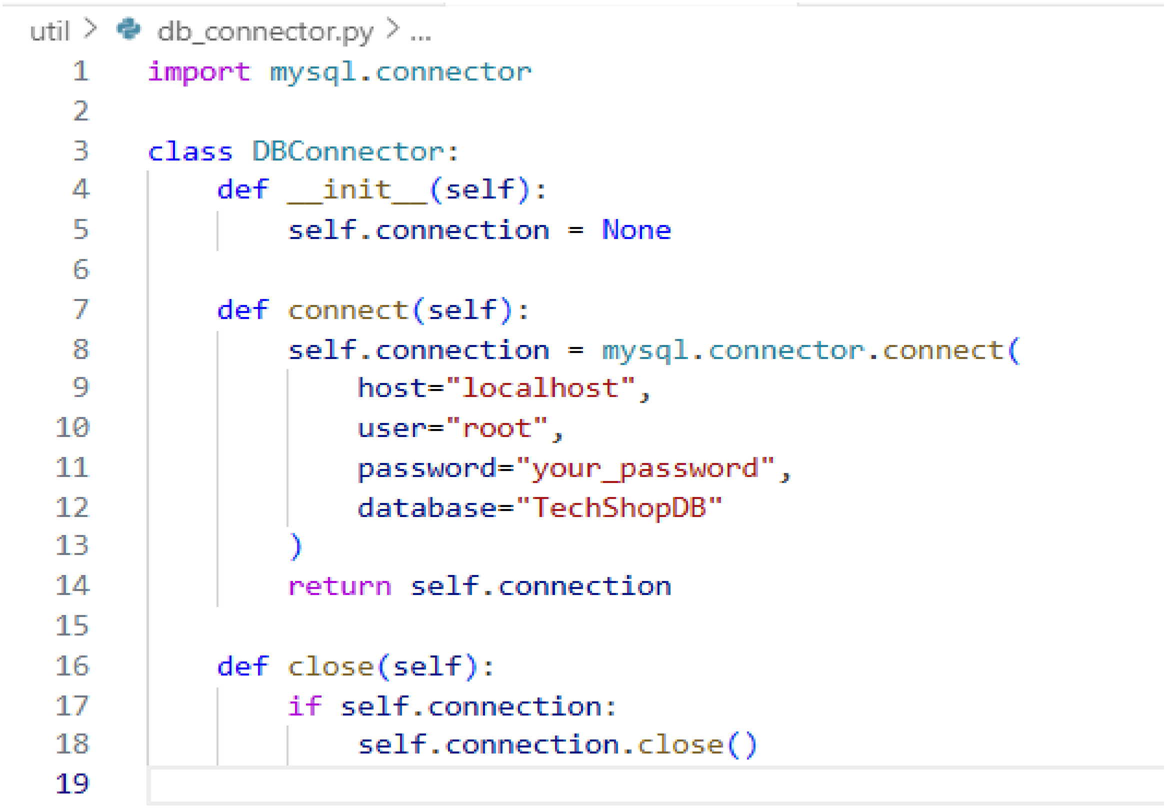
dao > impl > order_detail_service_impl.py > ...

```python
class OrderDetailServiceImpl:
    def validate_product_availability(self, inventory_service, product_id, quantity):
        inventory = inventory_service.get_inventory_by_product_id(product_id)
        if not inventory or inventory.quantity_in_stock < quantity:
            raise ValueError("Product not available in sufficient quantity.")

```

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.



util > db_connector.py > ...

```python
import mysql.connector

class DBConnector:
    def __init__(self):
        self.connection = None

    def connect(self):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="your_password",
            database="TechShopDB"
        )
        return self.connection

    def close(self):
        if self.connection:
            self.connection.close()

```

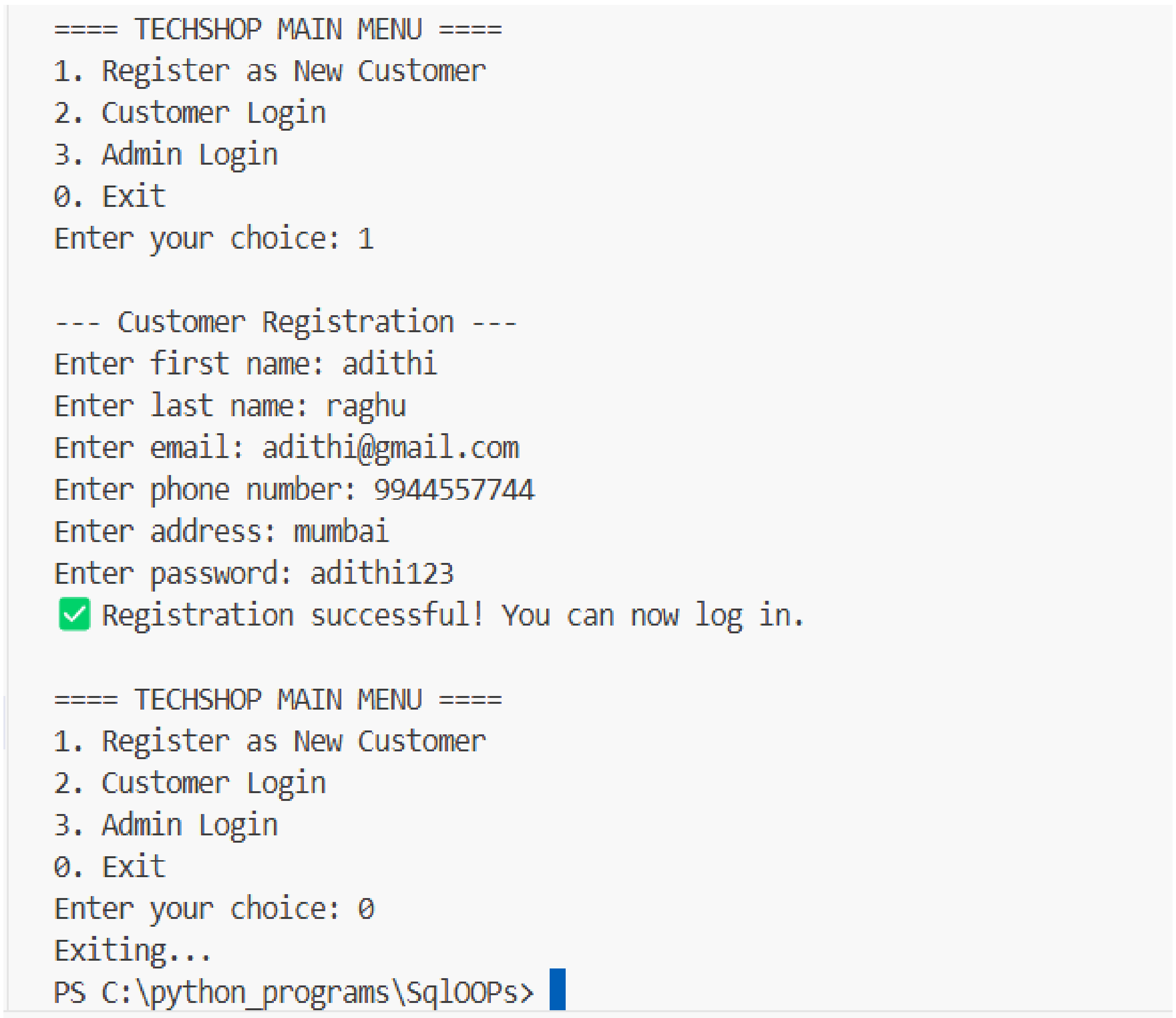
1: Customer Registration

Description: When a new customer registers on the TechShop website, their information (e.g., name, email, phone) needs to be stored in the database.

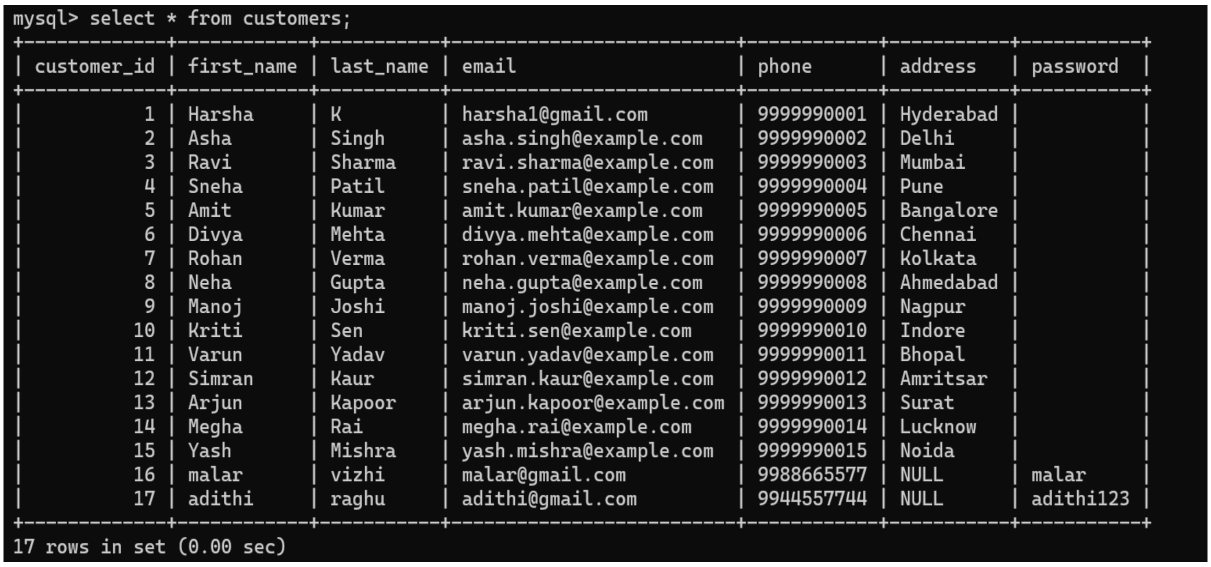
Task: Implement a registration form and database connectivity to insert new customer records. Ensure proper data validation and error handling for duplicate email addresses.

Code:

```
def register_customer():
    customer_dao = CustomerDAOImpl()


    print("\n--- Customer Registration ---")
    first_name = input("Enter first name: ")
    last_name = input("Enter last name: ")
    email = input("Enter email: ")
    phone = input("Enter phone number: ")
    address = input("Enter address: ")
    password = input("Enter password: ")
    # Check for duplicate email
    existing_customer = customer_dao.get_customer_by_email(email)
    if existing_customer:
        print(" Email already registered. Try logging in.")
        return
    customer = Customer(0, first_name, last_name, email, phone, address, password)


    customer_dao.register_customer(customer)
    print(" Registration successful! You can now log in.")
```



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

--- Customer Registration ---
Enter first name: adithi
Enter last name: raghu
Enter email: adithi@gmail.com
Enter phone number: 9944557744
Enter address: mumbai
Enter password: adithi123
✅ Registration successful! You can now log in.

==== TECHSHOP MAIN MENU ====
1. Register as New Customer
2. Customer Login
3. Admin Login
0. Exit
Enter your choice: 0
Exiting...
PS C:\python_programs\SqlOOPs>
```



```
mysql> select * from customers;
+-------------+------------+-----------+--------------------------+------------+-----------+-----------+
| customer_id | first_name | last_name | email                    | phone      | address   | password  |
+-------------+------------+-----------+--------------------------+------------+-----------+-----------+
|           1 | Harsha     | K         | harsha1@gmail.com        | 9999990001 | Hyderabad |           |
|           2 | Asha       | Singh     | asha.singh@example.com   | 9999990002 | Delhi     |           |
|           3 | Ravi       | Sharma    | ravi.sharma@example.com  | 9999990003 | Mumbai    |           |
|           4 | Sneha      | Patil     | sneha.patil@example.com  | 9999990004 | Pune      |           |
|           5 | Amit       | Kumar     | amit.kumar@example.com   | 9999990005 | Bangalore |           |
|           6 | Divya      | Mehta     | divya.mehta@example.com  | 9999990006 | Chennai   |           |
|           7 | Rohan      | Verma     | rohan.verma@example.com  | 9999990007 | Kolkata   |           |
|           8 | Neha       | Gupta     | neha.gupta@example.com   | 9999990008 | Ahmedabad |           |
|           9 | Manoj      | Joshi     | manoj.joshi@example.com  | 9999990009 | Nagpur    |           |
|          10 | Kriti      | Sen       | kriti.sen@example.com    | 9999990010 | Indore    |           |
|          11 | Varun      | Yadav     | varun.yadav@example.com  | 9999990011 | Bhopal    |           |
|          12 | Simran     | Kaur      | simran.kaur@example.com  | 9999990012 | Amritsar  |           |
|          13 | Arjun      | Kapoor    | arjun.kapoor@example.com | 9999990013 | Surat     |           |
|          14 | Megha      | Rai       | megha.rai@example.com    | 9999990014 | Lucknow   |           |
|          15 | Yash       | Mishra    | yash.mishra@example.com  | 9999990015 | Noida     |           |
|          16 | malar      | vizhi     | malar@gmail.com          | 9988665577 | NULL      | malar     |
|          17 | adithi     | raghu     | adithi@gmail.com         | 9944557744 | NULL      | adithi123 |
+-------------+------------+-----------+--------------------------+------------+-----------+-----------+
17 rows in set (0.00 sec)
```

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.

Code:

```
print("\n====== Customer Menu ======")
    print("1. View Products")
    print("2. Place Order")
    print("3. View My Orders")
    print("0. Logout")

    choice = input("Enter your choice: ")

    if choice == '1':
      products = product_dao.get_all_products()
      for p in products:
        inventory = inventory_dao.find_by_product_id(p.product_id)
        stock = inventory.quantity_in_stock if inventory else "N/A"
        print(f"{p.get_product_details()} | Available: {stock}")
```

```
==== TECHSHOP MAIN MENU ====
1. Register as New Customer
2. Customer Login
3. Admin Login
0. Exit
Enter your choice: 2
Enter Customer ID: 17
Welcome adithi!

====== Customer Menu ======
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 1
Product ID: 101
Name: Laptop
Description: Gaming Laptop
Price: ₹75000.00 | Available: 10
Product ID: 102
Name: Smartphone
Description: Latest Android Phone
Price: ₹25000.00 | Available: 20
Product ID: 103
Name: Headphones
Description: Wireless Bluetooth Headphones
Price: ₹3000.00 | Available: 15
Product ID: 104
Name: Monitor
Description: 24-inch Full HD Monitor
Price: ₹10000.00 | Available: 12
Product ID: 105
```

```
Name: Smartwatch
Description: Fitness Smartwatch
Price: ₹5500.00 | Available: 14
Product ID: 112
Name: Speaker
Description: Bluetooth Portable Speaker
Price: ₹3500.00 | Available: 10
Product ID: 113
Name: Camera
Description: Digital Camera 24MP
Price: ₹22000.00 | Available: 5
Product ID: 114
Name: TV
Description: 43-inch Smart LED TV
Price: ₹32000.00 | Available: 7
Product ID: 115
Name: Charger
Description: Fast Charging Adapter
Price: ₹1000.00 | Available: 20

====== Customer Menu ======
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 0

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

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.

Code:

```
elif choice == '2':
    order_id = randint(2000, 9999)
    order = Orders(order_id, customer)
    order_details = []

    while True:
        pid = int(input("Enter Product ID: "))
        qty = int(input("Enter Quantity: "))

        product = product_dao.find_product_by_id(pid)
        if product:
            try:
                product_inventory = inventory_dao.find_by_product_id(product.product_id)
                if product_inventory and qty <= product_inventory.quantity_in_stock:
                    detail_id = randint(3000, 9999)
                    order_detail = OrderDetail(detail_id, order, product, qty, product_inventory)
                    order.add_order_detail(order_detail)
                    order_details.append(order_detail)
                else:
                    print("Not enough stock available or inventory not found.")
            except Exception as e:
                print("Error fetching inventory:", e)
        else:
            print("Product not found.")

        cont = input("Add more? (y/n): ")
        if cont.lower() != 'y':
            break

    # Insert order and details only if we have valid entries
    if order_details:
        order.total_amount = order.calculate_total_amount()
        order_dao.insert_order(order)

        for od in order_details:
            try:
                order_detail_dao.insert_order_detail(od)
            except Exception as e:
                print("Error inserting order detail:", e)

        print("Order placed. Total amount:", order.total_amount)
```

```
==== TECHSHOP MAIN MENU ====
1. Register as New Customer
2. Customer Login
3. Admin Login
0. Exit
Enter your choice: 2
Enter Customer ID: 16
Welcome malar!

====== Customer Menu ======
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 2
Enter Product ID: 115
Enter Quantity: 1
Add more? (y/n): y
Enter Product ID: 112
Enter Quantity: 2
Add more? (y/n): n
Order 2214 placed for Customer: malar
Order detail added: Product - Charger, Quantity - 1
Order detail added: Product - Speaker, Quantity - 2
Order placed. Total amount: 8000.00
```

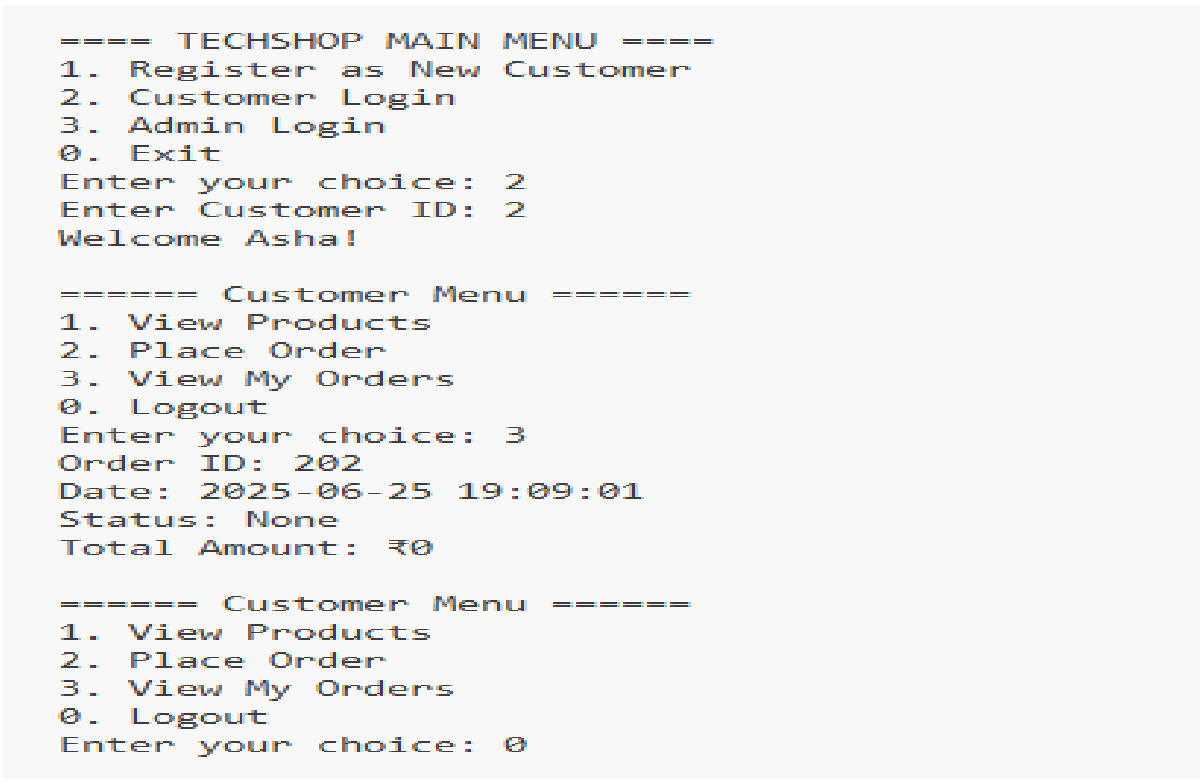
4: Tracking Order Status

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.

Code:

```
 elif choice == '3':
            orders = order_dao.get_orders_by_customer_id(customer.customer_id)
            for o in orders:
                print(o.get_order_details())


        elif choice == '0':
            break
        else:
            print("Invalid choice")
```



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

====== Customer Menu ======
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 3
Order ID: 202
Date: 2025-06-25 19:09:01
Status: None
Total Amount: ₹0

====== Customer Menu ======
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 0
```

5: Inventory Management

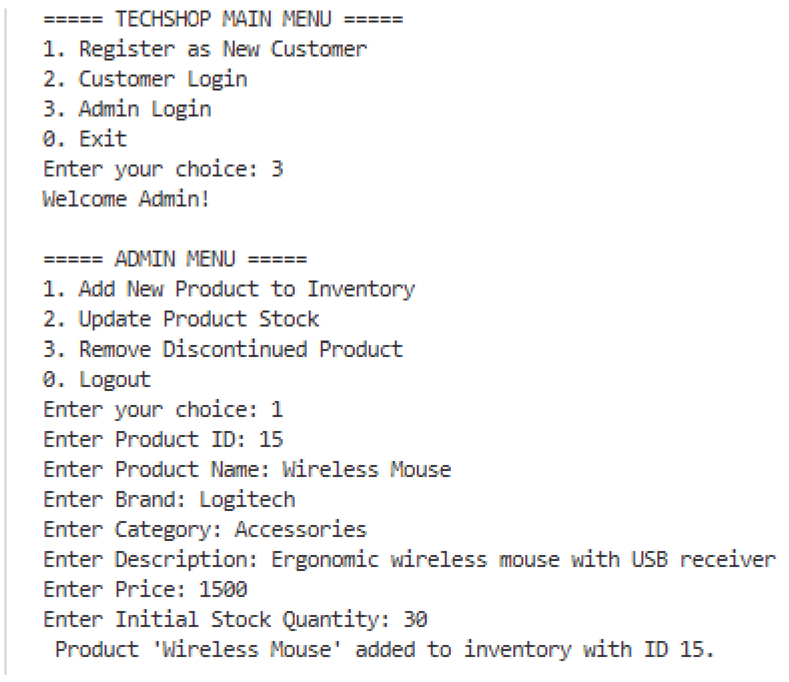
Description: TechShop needs to manage product inventory, including adding new products, updating stock levels, and removing discontinued items.

Task: Create an inventory management system with database connectivity. Implement features for adding new products, updating quantities, and handling discontinued products.

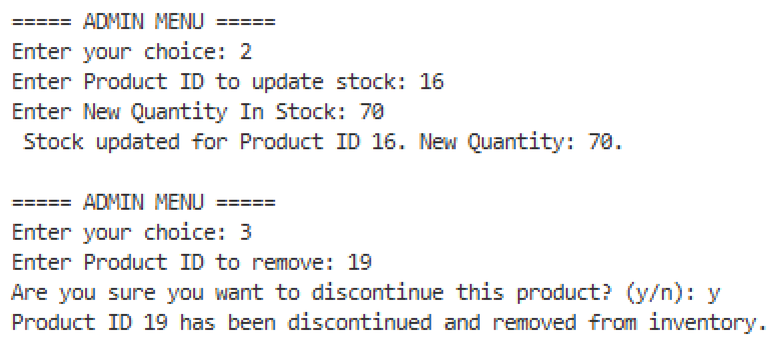
Code:

```
from util.db_connector import DBConnector

class InventoryDBService:

    def add_or_update_inventory(self, inventory):

        try:

            conn = DBConnector().connect()

            cursor = conn.cursor()


            cursor.execute("SELECT * FROM Inventory WHERE ProductID = %s", 
(inventory.product.product_id,))

            if cursor.fetchone():

                # Update existing inventory

                query = "UPDATE Inventory SET QuantityInStock=%s, LastStockUpdate=%s WHERE 
ProductID=%s"

                cursor.execute(query, (inventory.quantity_in_stock, inventory.last_stock_update, 
inventory.product.product_id))

            else:

                # Insert new inventory

                query = "INSERT INTO Inventory (InventoryID, ProductID, QuantityInStock, 
LastStockUpdate) VALUES (%s, %s, %s, %s)"

                cursor.execute(query, (inventory.inventory_id, inventory.product.product_id, 
inventory.quantity_in_stock, inventory.last_stock_update))


            conn.commit()


        except Exception as e:
```

```
        print("Error managing inventory:", e)
    finally:
        cursor.close()
        conn.close()
```



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

===== 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: 15
Enter Product Name: Wireless Mouse
Enter Brand: Logitech
Enter Category: Accessories
Enter Description: Ergonomic wireless mouse with USB receiver
Enter Price: 1500
Enter Initial Stock Quantity: 30
 Product 'Wireless Mouse' added to inventory with ID 15.
```



```
===== ADMIN MENU =====
Enter your choice: 2
Enter Product ID to update stock: 16
Enter New Quantity In Stock: 70
 Stock updated for Product ID 16. New Quantity: 70.

===== ADMIN MENU =====
Enter your choice: 3
Enter Product ID to remove: 19
Are you sure you want to discontinue this product? (y/n): y
Product ID 19 has been discontinued and removed from inventory.
```

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.

Code:

```
from util.db_connector import DBConnector
```

```
class SalesReportService:
    def get_sales_report(self):
        try:
            conn = DBConnector().connect()
            cursor = conn.cursor()

            query = """
                SELECT o.OrderID, c.FirstName, c.LastName, o.TotalAmount, o.OrderDate
                FROM Orders o
                JOIN Customers c ON o.CustomerID = c.CustomerID
            """
            cursor.execute(query)
            return cursor.fetchall()

        except Exception as e:
            print("Error generating sales report:", e)
        finally:
            cursor.close()
            conn.close()
```

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

===== ADMIN MENU =====
1. Inventory Management
2. Generate Sales Reports
0. Logout
Enter your choice: 2

===== SALES REPORT MENU =====
1. View All Sales
2. View Sales by Date Range
3. View Sales by Product
4. View Sales by Customer
0. Back
Enter your choice: 1

====== All Sales Report ======
Order ID: 17 | Customer: malar (Customer ID: 16) | Total: ₹8000.00 | Date: 2025-06-27
Order ID: 18 | Customer: ramesh (Customer ID: 19) | Total: ₹5200.00 | Date: 2025-06-26
Order ID: 19 | Customer: kriti  (Customer ID: 20) | Total: ₹1500.00 | Date: 2025-06-25
Total Sales: ₹14,700.00

===== SALES REPORT MENU =====
Enter your choice: 2
Enter Start Date (YYYY-MM-DD): 2025-06-25
Enter End Date (YYYY-MM-DD): 2025-06-26

====== Sales from 2025-06-25 to 2025-06-26 ======
Order ID: 18 | Customer: ramesh | Total: ₹5200.00
Order ID: 19 | Customer: kriti  | Total: ₹1500.00
Total Sales in Range: ₹6700.00

===== SALES REPORT MENU =====
Enter your choice: 3
Enter Product Name: Speaker

====== Sales Report for Product: Speaker ======
Order ID: 17 | Customer: malar | Quantity: 2 | Amount: ₹6000.00
Total Units Sold: 2 | Total Revenue: ₹6000.00

===== SALES REPORT MENU =====
Enter your choice: 4
Enter Customer Name: malar

====== Sales Report for Customer: malar ======
Order ID: 17 | Total: ₹8000.00 | Date: 2025-06-27
Total Amount Spent: ₹8000.00
```

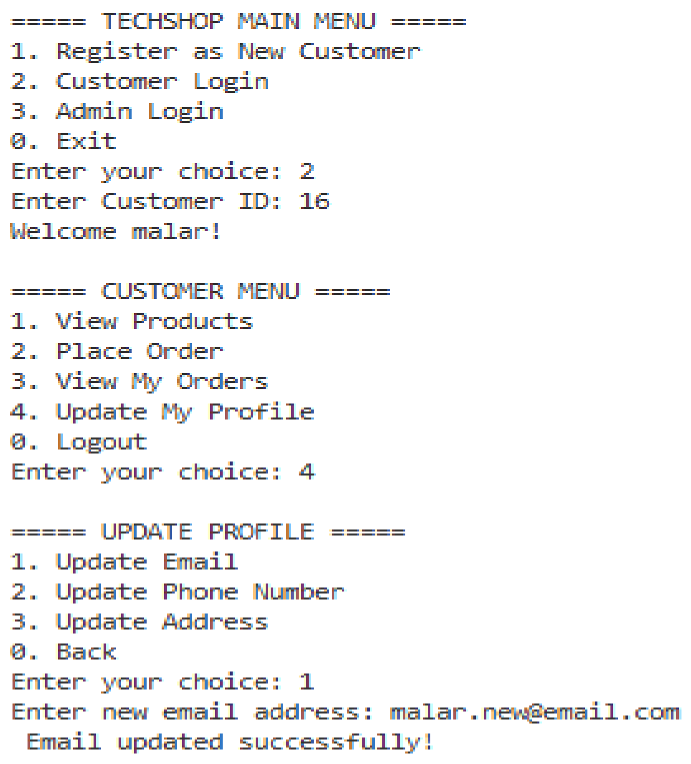
7: Customer Account Updates

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.

Code:

```
from util.db_connector import DBConnector

class CustomerUpdateService:

    def update_customer_info(self, customer_id, email=None, phone=None):

        try:

            conn = DBConnector().connect()

            cursor = conn.cursor()


            if email:

                cursor.execute("UPDATE Customers SET Email = %s WHERE CustomerID = %s", (email, customer_id))

            if phone:

                cursor.execute("UPDATE Customers SET Phone = %s WHERE CustomerID = %s", (phone, customer_id))


            conn.commit()

        except Exception as e:

            print("Error updating customer info:", e)

        finally:

            cursor.close()

            conn.close()
```



```
===== TECHSHOP MAIN MENU =====
1. Register as New Customer
2. Customer Login
3. Admin Login
0. Exit
Enter your choice: 2
Enter Customer ID: 16
Welcome malar!

===== CUSTOMER MENU =====
1. View Products
2. Place Order
3. View My Orders
4. Update My Profile
0. Logout
Enter your choice: 4

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

8: Payment Processing

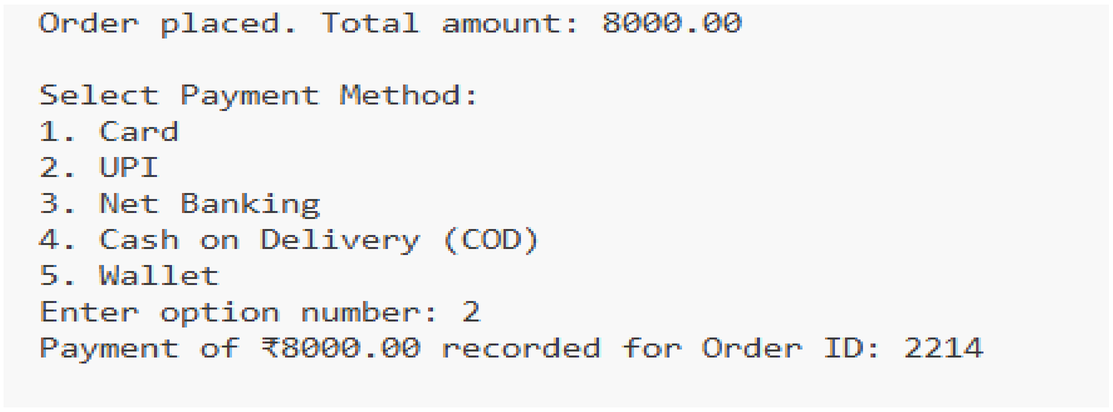
Description: When customers make payments for their orders, the payment details (e.g., payment method, amount) must be recorded in the database.

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

Code:

```
print("\nSelect Payment Method:")

print("1. Card")

print("2. UPI")

print("3. Net Banking")

print("4. Cash on Delivery (COD)")

print("5. Wallet")


method_input = input("Enter option number: ")
```

```
payment_methods = {
    "1": "card",
    "2": "upi",
    "3": "netbanking",
    "4": "cod",
    "5": "wallet"
}
payment_method = payment_methods.get(method_input)

if not payment_method:
    print("Invalid choice. Defaulting to 'cod'")
    payment_method = "cod"

payment = Payment(
    payment_id=randint(4000, 9999),
    order=order,
    amount=order.total_amount,
    payment_date=datetime.now(),
    payment_method=payment_method,
    status="Completed )
payment_dao.insert_payment(payment)
```



```
Order placed. Total amount: 8000.00

Select Payment Method:
1. Card
2. UPI
3. Net Banking
4. Cash on Delivery (COD)
5. Wallet
Enter option number: 2
Payment of ₹8000.00 recorded for Order ID: 2214
```

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.

Code:

```
from util.db_connector import DBConnector

class ProductSearchService:

    def search_products_by_name(self, keyword):

        try:

            conn = DBConnector().connect()

            cursor = conn.cursor()


            query = "SELECT * FROM Products WHERE ProductName LIKE %s"

            cursor.execute(query, (f"%{keyword}%",))

            return cursor.fetchall()


        except Exception as e:

            print("Error searching products:", e)

        finally:

            cursor.close()

            conn.close()


if choice == '1':

            products = product_dao.get_all_products()

            for p in products:

                inventory = inventory_dao.find_by_product_id(p.product_id)

                stock = inventory.quantity_in_stock if inventory else "N/A"

                print(f"{p.get_product_details()} | Available: {stock}")
```

```
==== TECHSHOP MAIN MENU ====
1. Register as New Customer
2. Customer Login
3. Admin Login
0. Exit
Enter your choice: 2
Enter Customer ID: 17
Welcome adithi!

====== Customer Menu ======
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 1
Product ID: 101
Name: Laptop
Description: Gaming Laptop
Price: ₹75000.00 | Available: 10
Product ID: 102
Name: Smartphone
Description: Latest Android Phone
Price: ₹25000.00 | Available: 20
Product ID: 103
Name: Headphones
Description: Wireless Bluetooth Headphones
Price: ₹3000.00 | Available: 15
Product ID: 104
Name: Monitor
Description: 24-inch Full HD Monitor
Price: ₹10000.00 | Available: 12
Product ID: 105
```

```
Name: Smartwatch
Description: Fitness Smartwatch
Price: ₹5500.00 | Available: 14
Product ID: 112
Name: Speaker
Description: Bluetooth Portable Speaker
Price: ₹3500.00 | Available: 10
Product ID: 113
Name: Camera
Description: Digital Camera 24MP
Price: ₹22000.00 | Available: 5
Product ID: 114
Name: TV
Description: 43-inch Smart LED TV
Price: ₹32000.00 | Available: 7
Product ID: 115
Name: Charger
Description: Fast Charging Adapter
Price: ₹1000.00 | Available: 20

====== Customer Menu ======
1. View Products
2. Place Order
3. View My Orders
0. Logout
Enter your choice: 0

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