PYTHON ASSIGNMENT

Implement OOPS

Sivaprakas B M

Task 1: Classes and Their Attributes:

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

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).

CODE :

```
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 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
        update_customer_info_in_database(self.customer_id, email, phone,
address)


def fetch_orders_for_customer(customer_id):
    pass


def update_customer_info_in_database(customer_id, email, phone, address):
    pass
```

PRODUT CLASS :

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.

CODE :

```python
class Product:
    def __init__(self, product_id, product_name, description, price,
stock_quantity):
        self.product_id = product_id
        self.product_name = product_name
        self.description = description
        self.price = price
        self.stock_quantity = stock_quantity

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

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

        update_product_info_in_database(self.product_id, price,
description)

    def is_product_in_stock(self):
        return self.stock_quantity > 0


def update_product_info_in_database(product_id, price, description):
    pass
```

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.

CODE :

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

    def calculate_total_amount(self):
        # Assume orders are fetched from a database or another source
        order_items = fetch_order_items_for_order(self.order_id)
        total_amount = sum(item.amount for item in order_items)
        return total_amount

    def get_order_details(self):
        # Assume orders are fetched from a database or another source
        order_items = fetch_order_items_for_order(self.order_id)
        order_details = f"Order ID: {self.order_id}\n" \
                        f"Customer: {self.customer.first_name}
{self.customer.last_name}\n" \
                        f"Order Date: {self.order_date}\n" \
                        f"Total Amount: ${self.total_amount}\n" \
                        "Order Items:\n"
        for item in order_items:
            order_details += f"- Product: {item.product.product_name},
Quantity: {item.quantity}, " \
                             f"Price: ${item.product.price}, Total:
${item.amount}\n"
        return order_details

    def update_order_status(self, status):

        update_order_status_in_database(self.order_id, status)

    def cancel_order(self):
e
        cancel_order_in_database(self.order_id)
```

```
def fetch_order_items_for_order(order_id):

    pass


def update_order_status_in_database(order_id, status):

    pass


def cancel_order_in_database(order_id):

    pass
```

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. class OrderDetail:

CODE :

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

    def calculate_total_amount(self):
        # Assume orders are fetched from a database or another source
        order_items = fetch_order_items_for_order(self.order_id)
        total_amount = sum(item.amount for item in order_items)
```

```python
        return total_amount

    def get_order_details(self):
        # Assume orders are fetched from a database or another source
        order_items = fetch_order_items_for_order(self.order_id)
        order_details = f"Order ID: {self.order_id}\n" \
                        f"Customer: {self.customer.first_name}
{self.customer.last_name}\n" \
                        f"Order Date: {self.order_date}\n" \
                        f"Total Amount: ${self.total_amount}\n" \
                        "Order Items:\n"
        for item in order_items:
            order_details += f"- Product: {item.product.product_name},
Quantity: {item.quantity}, " \
                             f"Price: ${item.product.price}, Total:
${item.amount}\n"
        return order_details

    def update_order_status(self, status):

        update_order_status_in_database(self.order_id, status)

    def cancel_order(self):

        cancel_order_in_database(self.order_id)


def fetch_order_items_for_order(order_id):

    pass


def update_order_status_in_database(order_id, status):

    pass


def cancel_order_in_database(order_id):

    pass
```

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.

CODE :

```
class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock,
last_stock_update=None):
        self.inventory_id = inventory_id
        self.product = product
        self.quantity_in_stock = quantity_in_stock
        self.last_stock_update = last_stock_update

    def get_product(self):
        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 quantity > self.quantity_in_stock:
            raise ValueError("Quantity to remove exceeds available stock")
        self.quantity_in_stock -= quantity
        self.last_stock_update = datetime.now()
```

```python
    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(self, threshold):
        if self.quantity_in_stock < threshold:
            return f"Product: {self.product.product_name}, Quantity:
{self.quantity_in_stock}"

    def list_out_of_stock_products(self):
        if self.quantity_in_stock == 0:
            return f"Product: {self.product.product_name} is out of 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).

CODE :

```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

    @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

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

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

    @email.setter
    def email(self, value):
        # Add email validation logic here
        self.__email = value

    @phone.setter
    def phone(self, value):
        # Add phone validation logic here
        self.__phone = value

    @address.setter
    def address(self, value):
        # Add address validation logic here
        self.__address = value


class Product:
    def __init__(self, product_id, product_name, description, price,
stock_quantity):
        self.__product_id = product_id
        self.__product_name = product_name
        self.__description = description
        self.__price = price
        self.__stock_quantity = stock_quantity

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

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

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

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

    @property
    def stock_quantity(self):
        return self.__stock_quantity

    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError("Price must be non-negative")
        self.__price = value
    @stock_quantity.setter
    def stock_quantity(self, value):
        if value < 0:
            raise ValueError("Stock quantity must be non-negative")
        self.__stock_quantity = value


class Order:
    def __init__(self, order_id, customer, order_date, total_amount):
        self.__order_id = order_id
        self.__customer = customer
        self.__order_date = order_date
        self.__total_amount = total_amount

    @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 total_amount(self):
        return self.__total_amount

    @total_amount.setter
    def total_amount(self, value):
        if value < 0:
            raise ValueError("Total amount must be non-negative")
        self.__total_amount = value


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

    @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 value < 0:
            raise ValueError("Quantity must be non-negative")
        self.__quantity = value


class Inventory:
    def __init__(self, inventory_id, product, quantity_in_stock,
last_stock_update=None):
        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

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

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

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

    @quantity_in_stock.setter
    def quantity_in_stock(self, value):
        if value < 0:
            raise ValueError("Quantity in stock must be non-negative")
        self.__quantity_in_stock = value

    @last_stock_update.setter
    def last_stock_update(self, value):
        # Add validation logic for last stock update
        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:

- 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.
- 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:**

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

• **Customers and Products Classes:**

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

CODE :

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


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

```
class Order:
    def __init__(self, order_id, customer, order_date, total_amount):
        self.order_id = order_id
        self.customer = customer  # Composition relationship with Customer
class
        self.order_date = order_date
        self.total_amount = total_amount


class OrderDetail:
    def __init__(self, order_detail_id, order, product, quantity):
        self.order_detail_id = order_detail_id
        self.order = order  # Composition relationship with Order class
        self.product = product  # Composition relationship with Product
class
        self.quantity = quantity


class Inventory:
    def __init__(self):
        self.inventory_items = []

    def add_to_inventory(self, product, quantity):
        # Add product to inventory with specified quantity
        self.inventory_items.append({"product": product, "quantity":
quantity})

    def remove_from_inventory(self, product, quantity):
        # Remove specified quantity of product from inventory
        for item in self.inventory_items:
            if item["product"] == product:
                item["quantity"] -= quantity
                break

    def list_low_stock_products(self, threshold):
        low_stock_products = []
        for item in self.inventory_items:
            if item["quantity"] < threshold:
                low_stock_products.append(item["product"])
        return low_stock_products

    def list_out_of_stock_products(self):
        out_of_stock_products = []
        for item in self.inventory_items:
            if item["quantity"] == 0:
                out_of_stock_products.append(item["product"])
        return out_of_stock_products
```

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.

o Exception 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.

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

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


class InsufficientStockException(Exception):
    pass


class IncompleteOrderException(Exception):
    pass


class PaymentFailedException(Exception):
    pass


class FileIOException(Exception):
    pass


class DatabaseConnectionException(Exception):
    pass


class ConcurrencyException(Exception):
    pass


class AuthenticationException(Exception):
    pass
```

```python
class AuthorizationException(Exception):
    pass


class User:
    def __init__(self, email):
        if "@" not in email:
            raise InvalidDataException("Invalid email address")
        self.email = email


class Order:
    def __init__(self, quantity, stock_quantity):
        if quantity > stock_quantity:
            raise InsufficientStockException("Insufficient stock to fulfill
order")
        self.quantity = quantity


class OrderDetail:
    def __init__(self, product):
        if not product:
            raise IncompleteOrderException("Order detail must include a
product reference")
        self.product = product


class PaymentProcessor:
    def process_payment(self):
        # Simulate payment processing
        if not payment_successful:
            raise PaymentFailedException("Payment failed")


class FileManager:
    def write_to_file(self):
        try:
            # Code to write data to file
            pass
        except IOError:
            raise FileIOException("Error writing data to file")


class DatabaseManager:
    def execute_query(self, query):
        try:
            # Code to execute SQL query
            pass
        except SqlException:
            raise DatabaseConnectionException("Database connection error")


class OrderProcessor:
    def process_order(self, order):
        try:
            # Code to process order
            pass
        except ConcurrencyException:
            raise ConcurrencyException("Order processing failed due to
concurrent access")
        except Exception as e:
            raise e  # Re-raise any other exceptions


class AuthenticationService:
    def authenticate_user(self, user):
        if not authenticated:
            raise AuthenticationException("Authentication failed")

    def authorize_access(self, user):
        if not authorized:
            raise AuthorizationException("Unauthorized access")


# Example usage:
try:
    # Attempt user registration with invalid email
    user = User("invalid_email")
except InvalidDataException as e:
    print(e)

try:
    # Attempt to place an order with insufficient stock
    order = Order(20, 15)
except InsufficientStockException as e:
    print(e)

try:
    # Attempt to create an incomplete order detail
    order_detail = OrderDetail(None)
except IncompleteOrderException as e:
    print(e)



try:
    # Attempt file I/O operation with IO error
    file_manager = FileManager()
    file_manager.write_to_file()
except FileIOException as e:
    print(e)

try:
    # Attempt database query with connection error
    database_manager = DatabaseManager()
    database_manager.execute_query("SELECT * FROM table")
except DatabaseConnectionException as e:
    print(e)

try:
    # Attempt order processing with concurrency issue
    order_processor = OrderProcessor()
    order_processor.process_order(Order)
except ConcurrencyException as e:
    print(e)

try:
    # Attempt user authentication with failed authentication
    authentication_service = AuthenticationService()
    authentication_service.authenticate_user(user)
except AuthenticationException as e:
    print(e)
```

```python
try:
    # Attempt user access authorization with unauthorized access
    authentication_service.authorize_access(user)
except AuthorizationException as e:
    print(e)
```

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.

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

    def add_product(self, product):
        if self.find_product_by_id(product.product_id):
            raise ValueError("Product with the same ID already exists")
        self.products.append(product)

    def update_product(self, product_id, new_product):
        existing_product = self.find_product_by_id(product_id)
        if not existing_product:
            raise ValueError("Product with the given ID does not exist")
        existing_product.product_name = new_product.product_name
        existing_product.description = new_product.description
        existing_product.price = new_product.price
        existing_product.stock_quantity = new_product.stock_quantity

    def remove_product(self, product_id):
        existing_product = self.find_product_by_id(product_id)
        if not existing_product:
            raise ValueError("Product with the given ID does not exist")
        # Check if the product has existing orders before removing it
        if existing_product.stock_quantity > 0:
            raise ValueError("Cannot remove product with existing orders")
        self.products.remove(existing_product)

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

    def display_products(self):
        for product in self.products:
            print(product)
```

• **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.

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

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

    def update_order_status(self, order_id, new_status):
        order = self.find_order_by_id(order_id)
        if not order:
            raise ValueError("Order with the given ID does not exist")
        if new_status not in ["Pending", "Processing", "Shipped",
"Cancelled"]:
            raise ValueError("Invalid order status")
        order.status = new_status

    def remove_cancelled_orders(self):
        cancelled_orders = [order for order in self.orders if order.status
== "Cancelled"]
        for order in cancelled_orders:
            self.orders.remove(order)

    def find_order_by_id(self, order_id):
        for order in self.orders:
            if order.order_id == order_id:
                return order
        return None

    def display_orders(self):
        for order in self.orders:
            print(order)
```

• **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.

```
def sort_orders_by_date(self, ascending=True):
    self.orders.sort(key=lambda x: x.order_date, reverse=not ascending)
```

• **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.

```
class InventoryManager:
    def __init__(self):
        self.inventory = SortedList(key=lambda x: x.product_id)

    def add_product_to_inventory(self, product):
        index = self._binary_search(product.product_id)
        if index >= 0:
            raise ValueError("Product with the same ID already exists")
        self.inventory.add(product)

    def remove_product_from_inventory(self, product_id):
        index = self._binary_search(product_id)
        if index < 0:
            raise ValueError("Product with the given ID does not exist")
        del self.inventory[index]

    def update_stock_quantity(self, product_id, new_quantity):
        index = self._binary_search(product_id)
        if index < 0:
            raise ValueError("Product with the given ID does not exist")
        self.inventory[index].stock_quantity = new_quantity

    def get_product_by_id(self, product_id):
        index = self._binary_search(product_id)
        if index < 0:
            return None
        return self.inventory[index]

    def _binary_search(self, product_id):
        left, right = 0, len(self.inventory) - 1
        while left <= right:
            mid = (left + right) // 2
            if self.inventory[mid].product_id == product_id:
                return mid
            elif self.inventory[mid].product_id < product_id:
                left = mid + 1
            else:
                right = mid - 1
        return -1

    def display_inventory(self):
        for product in self.inventory:
            print(product)
```

• 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.

```
def display_inventory(self):
    for product_id, item in self.inventory.items():
        print(f"Product ID: {product_id}, Name:
{item['product'].product_name}, "
              f"Quantity in Stock: {item['quantity']}")
```

• 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 _is_duplicate_product(self, new_product):
    for product in self.products:
        if product.product_name.lower() == new_product.product_name.lower()
\
                or product.sku.lower() == new_product.sku.lower():
            return True
    return False
```

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.

```
def add_order_detail(self, order, order_detail):
    if not
self.inventory_manager.is_product_available(order_detail.product,
order_detail.quantity):
        raise ValueError("Product is not available in sufficient quantity")
    order.add_order_detail(order_detail)
```

Task 7: Database Connectivity

• Implement a DatabaseConnector class responsible for establishing a connection to the "TechShop" database. This class should include methods for opening, closing, and managing database connections.

• Implement classes for Customers, Products, Orders, OrderDetails, Inventory with properties, constructors, and methods for CRUD (Create, Read, Update, Delete) operations.

1: Customer Registration

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

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

2: Product Catalog Management

Description: TechShop regularly updates its product catalog with new items and changes in product details (e.g., price, description). These changes need to be reflected in the database.

Task: Create an interface to manage the product catalog. Implement database connectivity to update product information. Handle changes in product details and ensure data consistency.

3: Placing Customer Orders

Description: Customers browse the product catalog and place orders for products they want to purchase. The orders need to be stored in the database.

Task: Implement an order processing system. Use database connectivity to record customer orders, update product quantities in inventory, and calculate order totals.

4: Tracking Order Status

Description: Customers and employees need to track the status of their orders. The order status information is stored in the database.

Task: Develop a feature that allows users to view the status of their orders. Implement database connectivity to retrieve and display order status information.

5: Inventory Management

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

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

6: Sales Reporting

Description: TechShop management requires sales reports for business analysis. The sales data is stored in the database.

Task: Design and implement a reporting system that retrieves sales data from the database and generates reports based on specified criteria.

7: Customer Account Updates

Description: Customers may need to update their account information, such as changing their email address or phone number.

Task: Implement a user profile management feature with database connectivity to allow customers to update their account details. Ensure data validation and integrity.

8: Payment Processing

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

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

9: Product Search and Recommendations

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

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

```python
import mysql.connector


class DatabaseConnector:
    def __init__(self, host, user, password, database):
        self.host = host
        self.user = user
        self.password = password
        self.database = database
        self.conn = None

    def open_connection(self):
        self.conn = mysql.connector.connect(
            host=self.host,
            user=self.user,
            password=self.password,
            database=self.database
        )
        print("Connected to database successfully.")

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

    def execute_query(self, query, parameters=None):
        if not self.conn:
            raise ValueError("Connection is not open.")
        cursor = self.conn.cursor()
        if parameters:
            cursor.execute(query, parameters)
        else:
            cursor.execute(query)
        self.conn.commit()
        return cursor
```

```python
# Example usage:
db_connector = DatabaseConnector(host="localhost", user="root",
password="Siva@2003", database="TechShop")
db_connector.open_connection()

# Create instances of classes and save to database
```

```
customer1 = Customer(1, "Siva", "Prakas", "sivaprakas@gmail.com",
"1234567890")
customer1.save_to_database(db_connector)

product1 = Product(1, "Laptop", "Normal laptop", 80000.00, 10)
product1.save_to_database(db_connector)

order1 = Order(1, 1, "2024-01-01", 800.00)
order1.save_to_database(db_connector)

order_detail1 = OrderDetail(1, 1, 1, 1, 80.00)
order_detail1.save_to_database(db_connector)

inventory1 = Inventory(1, 1, 10, "2024-01-01")
inventory1.save_to_database(db_connector)

db_connector.close_connection()
```