**PYTHON ASSIGNMENT**

**Implement OOPS**

**Task 1: Classes and Their Attributes:** You are working as a software developer for TechShop, a company that sells electronic gadgets. Your task is to design and implement an application using Object-Oriented Programming (OOP) principles to manage customer information, product details, and orders. Below are the classes you need to create:

**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 calculate_total_orders(self):
        pass

    def get_customer_details(self):
        pass

    def update_customer_info(self, new_email, new_phone, new_address):
        self.email = new_email
        self.phone = new_phone
        self.address = new_address
```

**Products Class:**

**Attributes:**

• ProductID (int)

• ProductName (string)

• Description (string)

• Price (decimal)

**Methods:**

• GetProductDetails(): Retrieves and displays detailed information about the product.

• UpdateProductInfo(): Allows updates to product details (e.g., price, description).

• IsProductInStock(): Checks if the product is currently in stock.

```
class Product:
    def __init__(self, product_id, 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):
        pass

    def update_product_info(self, new_name, new_description, new_price):
        self.product_name = new_name
        self.description = new_description
        self.price = new_price
```

```
    def is_product_in_stock(self):
        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.

```
class Order:
    def __init__(self, order_id, customer, order_date):
        self.order_id = order_id
        self.customer = customer
        self.order_date = order_date
        self.total_amount = 0  # CalculateTotalAmount method should be called to calculate the total amount

    def calculate_total_amount(self):
        pass

    def get_order_details(self):
        pass

    def update_order_status(self, status):
         pass

    def cancel_order(self):
        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:
    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

    def calculate_subtotal(self):
        pass

    def get_order_detail_info(self):
        pass

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

    def add_discount(self, discount):
        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.

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

    def get_product(self):
        pass

    def get_quantity_in_stock(self):
        pass

    def add_to_inventory(self, quantity):
        pass

    def remove_from_inventory(self, quantity):
        pass

    def update_stock_quantity(self, new_quantity):
        pass
```

```
    def is_product_available(self, quantity_to_check):
        pass

    def get_inventory_value(self):
        pass

    def list_low_stock_products(self, threshold):
        pass

    def list_out_of_stock_products(self):
        pass

    def list_all_products(self):
        pass
```

**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).

```
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

    @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

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

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

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

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

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

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

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

    @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):
        self._price = value

class Order:
    def __init__(self, order_id, customer, order_date):
        self._order_id = order_id
        self._customer = customer
        self._order_date = order_date
        self._total_amount = 0

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

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

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

    @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):
        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

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

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

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

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

    @quantity.setter
    def quantity(self, value):
        self._quantity = value

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

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

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

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

    @quantity_in_stock.setter
    def quantity_in_stock(self, value):
        self._quantity_in_stock = value

    @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.

• **OrderDetails Class with Composition:** o Similarly, in the OrderDetails class, we want to establish composition relationships with both the Orders and Products classes to represent the details of each order, including the product being ordered.

o In the OrderDetails class, we've added two private attributes, order and product, of types Orders and Products, respectively, establishing composition relationships. The Order property provides access to the Orders object associated with the order detail, and the Product property provides access to the Products object representing the product in the order detail.

• **Customers and Products Classes**:

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

• **Inventory Class**:

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

```
class Order:
    def __init__(self, order_id, customer, order_date):
        self._order_id = order_id
        self._customer = customer
        self._order_date = order_date
        self._total_amount = 0

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

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

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

    @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):
        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

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

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

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

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

    @quantity.setter
    def quantity(self, value):
        self._quantity = value
```

**Task 5: Exceptions handling**

**• Data Validation:**

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

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

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

```
class InvalidDataException(Exception):
    pass


def register_user(email):
    if not is_valid_email(email):
        raise InvalidDataException("Invalid email address provided.")

def is_valid_email(email):
    pass
```

• **Inventory Management:**

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

```
class InsufficientStockException(Exception):
    pass


def process_order(order):
    for order_detail in order.order_details:
        if order_detail.quantity > order_detail.product.quantity_in_stock:
            raise InsufficientStockException("Insufficient stock for
product: {}".format(order_detail.product.product_name))
```

• **Order Processing:** o Challenge: Ensuring the order details are consistent and complete before processing.

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

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

```
class IncompleteOrderException(Exception):
    pass

def process_order(order):
    for order_detail in order.order_details:
        if not order_detail.product:
            raise IncompleteOrderException("Order detail is missing product
reference.")
```

• **Payment Processing:**

o Challenge: Handling payment failures or declined transactions.

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

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

```
class PaymentFailedException(Exception):
    pass

def process_payment(order):
    try:
        pass
    except PaymentFailedException as e:
        raise PaymentFailedException("Payment processing failed for order:
{}".format(order.order_id))
```

**• 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.

```
import logging

def log_error(message):
    try:
        logging.error(message)
    except IOError as e:
        raise IOError("Error occurred while logging: {}".format(str(e)))
```

**• Database Access:**

o Challenge: Managing database connections and queries.

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

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

```
class DatabaseOfflineException(Exception):
    pass

def execute_sql_query(query):
    try:
        pass
    except DatabaseOfflineException as e:
       raise DatabaseOfflineException("Database is offline. Unable to
execute query: {}".format(query))
```

• **Concurrency Control:**

o Challenge: Preventing data corruption in multi-user scenarios.

o Scenario: When two users simultaneously attempt to update the same order.

o Exception Handling: Implement optimistic concurrency control and handle ConcurrencyException by notifying users to retry.

```
class ConcurrencyException(Exception):
    pass

def update_order(order):
    try:
        pass
    except ConcurrencyException as e:
        raise ConcurrencyException("Concurrency issue encountered while
updating order: {}".format(order.order_id))
```

• **Security and Authentication:**

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

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

oException Handling:Implement custom Authentication Exception andAuthorization Exception to handle security-related issues.

```
class AuthenticationException(Exception):
    pass

class AuthorizationException(Exception):
    pass

def authenticate_user(username, password):
    if not is_valid_credentials(username, password):
        raise AuthenticationException("Invalid username or password.")

def authorize_access(user, resource):
    if not is_authorized(user, resource):
        raise AuthorizationException("User is not authorized to access this
resource.")
```

**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.

```
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 ID {} already
exists.".format(product.product_id))
        self.products.append(product)

    def update_product(self, product):
        existing_product = self.find_product_by_id(product.product_id)
        if not existing_product:
            raise ValueError("Product with ID {} does not
exist.".format(product.product_id))
        self.products.remove(existing_product)
        self.products.append(product)

    def remove_product(self, product_id):
        product = self.find_product_by_id(product_id)
        if not product:
            raise ValueError("Product with ID {} does not
exist.".format(product_id))
        self.products.remove(product)

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

• **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 ID {} does not
exist.".format(order_id))
        order.status = new_status
```

```
    def remove_order(self, order_id):
        order = self.find_order_by_id(order_id)
        if not order:
            raise ValueError("Order with ID {} does not
exist.".format(order_id))
        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
```

**• 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.

```
class OrderManager:

    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.

```
from sortedcontainers import SortedDict

class InventoryManager:
    def __init__(self):
        self.inventory = SortedDict()

    def add_to_inventory(self, product, quantity):
        if product.product_id in self.inventory:
            self.inventory[product.product_id] += quantity
        else:
            self.inventory[product.product_id] = quantity

    def remove_from_inventory(self, product, quantity):
        if product.product_id not in self.inventory:
            raise ValueError("Product with ID {} not found in
inventory.".format(product.product_id))
        if self.inventory[product.product_id] < quantity:
            raise ValueError("Insufficient stock for product with ID
```

```
{}.".format(product.product_id))
        self.inventory[product.product_id] -= quantity

    def get_inventory_quantity(self, product_id):
        return self.inventory.get(product_id, 0)
```

• **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.

```
class OrderManager:

    def process_order(self, order):
        try:
            self.validate_order_details(order)
            for order_detail in order.order_details:

inventory_manager.remove_from_inventory(order_detail.product,
order_detail.quantity)
            self.add_order(order)
        except Exception as e:
            raise e
```

• **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.

```
class ProductManager:

    def search_products(self, criteria):
        pass
```

• **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.

```
class ProductManager:

        def check_duplicate_product(self, product):
            if self.find_product_by_id(product.product_id):
                raise ValueError("Product with ID {} already
exists.".format(product.product_id))
            if self.find_product_by_name(product.product_name):
                raise ValueError("Product with name {} already
exists.".format(product.product_name))

        def find_product_by_name(self, product_name):
            for product in self.products:
                if product.product_name == product_name:
                    return product
            return None
```

• **Payment Records List:**

o Challenge: Managing a list of payment records for orders (List).

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

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

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

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

    def update_payment_status(self, order_id, new_status):
        pass

    def handle_payment_errors(self, order_id, error_message):
        pass
```

• **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.

```
class OrderManager:

    def validate_order_details(self, order):
        for order_detail in order.order_details:
            if not self.is_product_available(order_detail.product,
```

```
order_detail.quantity):
                raise ValueError("Product {} is not available in sufficient
quantity.".format(order_detail.product.product_name))

    def is_product_available(self, product, quantity):
        return
inventory_manager.get_inventory_quantity(product.product_id) >= 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.

**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.

**Code:**

```
import mysql.connector

class DatabaseConnector:
    def __init__(self, host, user, password, database):
        self.connection = mysql.connector.connect(
            host='localhost',
            user='root',
            password='Roshini2002*',
            database='3306'
        )
        self.cursor = self.connection.cursor()

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

class Customers:
    def __init__(self, db_connector):
        self.db_connector = db_connector

    def create_customer(self, first_name, last_name, email, phone,
address):
        query = "INSERT INTO Customers (FirstName, LastName, Email, Phone,
Address) VALUES (%s, %s, %s, %s, %s)"
        values = (first_name, last_name, email, phone, address)
        try:
            self.db_connector.cursor.execute(query, values)
            self.db_connector.connection.commit()
            print("Customer registered successfully!")
        except mysql.connector.Error as err:
```

```
            print("Error:", err.msg)

class Products:
    def __init__(self, db_connector):
        self.db_connector = db_connector

    def create_product(self, product_name, description, price):
        query = "INSERT INTO Products (ProductName, Description, Price)
VALUES (%s, %s, %s)"
        values = (product_name, description, price)
        try:
            self.db_connector.cursor.execute(query, values)
            self.db_connector.connection.commit()
            print("Product added successfully!")
        except mysql.connector.Error as err:
            print("Error:", err.msg)

class Orders:
    def __init__(self, db_connector):
        self.db_connector = db_connector

    def create_order(self, customer_id, order_date, total_amount):
        query = "INSERT INTO Orders (CustomerID, OrderDate, TotalAmount)
VALUES (%s, %s, %s)"
        values = (customer_id, order_date, total_amount)
        try:
            self.db_connector.cursor.execute(query, values)
            self.db_connector.connection.commit()
            print("Order created successfully!")
        except mysql.connector.Error as err:
            print("Error:", err.msg)
class OrderDetails:
    def __init__(self, db_connector):
        self.db_connector = db_connector

    def create_order_detail(self, order_id, product_id, quantity):
        query = "INSERT INTO OrderDetails (OrderID, ProductID, Quantity)
VALUES (%s, %s, %s)"
        values = (order_id, product_id, quantity)
        try:
            self.db_connector.cursor.execute(query, values)
            self.db_connector.connection.commit()
            print("Order detail added successfully!")
        except mysql.connector.Error as err:
            print("Error:", err.msg)

class Inventory:
    def __init__(self, db_connector):
        self.db_connector = db_connector

    def add_to_inventory(self, product_id, quantity_in_stock):
        query = "INSERT INTO Inventory (ProductID, QuantityInStock) VALUES
(%s, %s)"
        values = (product_id, quantity_in_stock)
        try:
            self.db_connector.cursor.execute(query, values)
            self.db_connector.connection.commit()
            print("Product added to inventory successfully!")
        except mysql.connector.Error as err:
            print("Error:", err.msg)
```

```
# Database connection parameters
HOST = 'localhost'
USER = 'root'
PASSWORD = 'Roshini2002*'
DATABASE = 'TechShop'

# Create a DatabaseConnector instance
db_connector = DatabaseConnector(HOST, USER, PASSWORD, DATABASE)

# Create a Customers instance
customers = Customers(db_connector)

# Register a new customer
customers.create_customer("John", "Doe", "john@example.com", "1234567890",
"123 Main St")

# Close the database connection when done
db_connector.close_connection()
```