HEXAWARE ASSIGNMENT - 1

NAME: AISHWARYA B
SUPERSET ID: 5006869
ASSIGNMENT NAME: 1- ELECTRONIC GADGET - OOPS IMPLEMENTATION

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:

Customer 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 customer information.
• UpdateCustomerInfo(): Allows the customer to update their information (e.g., email, phone, or address).

```
class Customer:
    def __init__(self, customer_id: int, first_name: str, last_name: str,
                 email: str, phone: str, address: str, order_count: int = 0):
        self.__customer_id = customer_id
        self.__first_name = first_name
        self.__last_name = last_name
        self.__email = email
        self.__phone = phone
        self.__address = address
```

```
        self.__order_count = order_count

    # ---------- Getters & Setters ----------
    @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):
        if value.strip():
            self.__first_name = value
        else:
            raise ValueError("First name cannot be empty")

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

    @last_name.setter
    def last_name(self, value):
        if value.strip():
            self.__last_name = value
        else:
            raise ValueError("Last name cannot be empty")

    @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("Invalid phone number")

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

    @address.setter
    def address(self, value):
        if value.strip():
            self.__address = value
        else:
            raise ValueError("Address cannot be empty")

    @property
    def order_count(self):
        return self.__order_count

    # ---------- Methods ----------
    def calculate_total_orders(self, order_list: list):
                    return sum(1 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}\n"
            f"Total Orders: {self.__order_count}"
        )

            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 product information.
• UpdateProductInfo(): Allows updates to product details (e.g., price, description).
• IsProductInStock(): Checks if the product is currently in stock.

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

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

#getters and setters

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

    @product_name.setter
    def product_name(self, value):
        if value.strip():
```

```
            self.__product_name = value
        else:
            raise ValueError("Product name cannot be empty.")

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

    @description.setter
    def description(self, value):
        if value.strip():
            self.__description = value
        else:
            raise ValueError("Description cannot be empty.")

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

    @price.setter
    def price(self, value):
        if value >= 0:
            self.__price = value
        else:
            raise ValueError("Price must be non-negative.")

    @property
    def category(self):
        return self.__category

    @category.setter
    def category(self, value):
        if value.strip():
            self.__category = value
        else:
            raise ValueError("Category cannot be empty.")

    # ---------- Methods ----------
    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:.2f}\n"
            f"Category: {self.__category}"
        )

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

    def is_product_in_stock(self, inventory_list: list) -> bool:
        for item in inventory_list:
            if item.product.product_id == self.__product_id:
                return item.quantity_in_stock > 0
        return False
__all__ = ['Product']
```

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
from entity.customer import Customer
from entity.order_detail import OrderDetail

class Order:
    def __init__(self, order_id: int, customer: Customer, order_date: datetime, status: str = "Packing"):
        self.__order_id = order_id
        self.__customer = customer
        self.__order_date = order_date
        self.__status = status
        self.__order_details = []  # List of OrderDetail
        self.__total_amount = 0.0

    # ---------- Properties ----------
    @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 status(self):
        return self.__status

    @status.setter
    def status(self, value):
        allowed_status = ["Packing", "Shipped", "Out for Delivery", "Delivered", "Cancelled"]
        if value in allowed_status:
            self.__status = value
        else:
            raise ValueError("Invalid order status.")

    @property
    def total_amount(self):
```

```
        return self.__total_amount

    @property
    def order_details(self):
        return self.__order_details

    # ---------- Methods ----------
    def add_order_detail(self, order_detail: OrderDetail):
        self.__order_details.append(order_detail)
        self.calculate_total_amount()

    def set_order_details(self, details: list):
        self.__order_details = details
        self.calculate_total_amount()

    def calculate_total_amount(self):
            self.__total_amount = sum(detail.calculate_subtotal() for
detail in self.__order_details)
          return self.__total_amount   # ✅ make sure you return the
value!


    def get_order_details(self):
        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.strftime('%Y-%m-%d')}\n"
            f"Status: {self.__status}\n"
        )
        if not self.__order_details:
            details += "\n(No order details found.)"
        else:
            for detail in self.__order_details:
                details += f"\n{detail.get_order_detail_info()}"
        details += f"\nTotal Amount: ₹{self.__total_amount:.2f}"
        return details
```

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.

```
from typing import TYPE_CHECKING
if TYPE_CHECKING:
    from entity.order import Order

from entity.product import Product

class OrderDetail:
     def __init__(self, order_detail_id: int, order: "Order", product:
Product, quantity: int):
        self.__order_detail_id = order_detail_id
        self.__order = order
        self.__product = product
        self.__quantity = quantity
        self.__discount = 0.0  # Optional: percentage discount

    # ---------- Properties ----------
    @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
```

```python
    @property
    def quantity(self):
        return self.__quantity

    @quantity.setter
    def quantity(self, value):
        if value > 0:
            self.__quantity = value
        else:
            raise ValueError("Quantity must be greater than zero.")

    @property
    def discount(self):
        return self.__discount

    # ---------- Business Logic ----------
    def calculate_subtotal(self):
        subtotal = self.__product.price * self.__quantity
        if self.__discount > 0:
            subtotal -= (subtotal * self.__discount / 100)
        return subtotal

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

    def add_discount(self, discount_percentage: float):
        if 0 <= discount_percentage <= 100:
            self.__discount = discount_percentage
        else:
            raise ValueError("Discount must be between 0 and 100.")

    def get_order_detail_info(self):
        return (
            f"OrderDetail ID: {self.__order_detail_id}, "
            f"Product: {self.__product.product_name}, "
            f"Price: ₹{self.__product.price:.2f}, "
            f"Quantity: {self.__quantity}, "
            f"Discount: {self.__discount:.1f}%, "
            f"Subtotal: ₹{self.calculate_subtotal():.2f}"
        )
```

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.

```
from datetime import datetime

class Inventory:
        def  __init__(self,  inventory_id,  product,  quantity_in_stock,
last_stock_update=None):
        self.__inventory_id = inventory_id
          self.__product = product  # Composition: product is an object
of Product class
        self.__quantity_in_stock = quantity_in_stock
        self.__last_stock_update = last_stock_update or datetime.now()

    # ---------- Getters ----------
    @property
    def inventory_id(self):
        return self.__inventory_id
```

```python
    @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

    def get_product(self):
        """Returns the associated product object."""
        return self.__product

    def get_quantity_in_stock(self):
        """Returns the quantity in stock."""
        return self.__quantity_in_stock

    # ---------- Inventory Operations ----------
    def add_to_inventory(self, quantity: int):
        """Adds specified quantity to inventory."""
        if quantity <= 0:
            raise ValueError("Quantity must be positive.")
        self.__quantity_in_stock += quantity
        self.__last_stock_update = datetime.now()

    def remove_from_inventory(self, quantity: int):
        """Removes specified quantity from inventory."""
        if quantity <= 0:
            raise ValueError("Quantity must be positive.")
        if quantity > self.__quantity_in_stock:
            raise ValueError("Not enough stock to remove.")
        self.__quantity_in_stock -= quantity
        self.__last_stock_update = datetime.now()

    def update_stock_quantity(self, new_quantity: int):
        """Updates inventory stock to a new quantity."""
        if new_quantity < 0:
            raise ValueError("Quantity cannot be negative.")
        self.__quantity_in_stock = new_quantity
```

```
        self.__last_stock_update = datetime.now()

    def is_product_available(self, quantity_to_check: int):
        """Checks if requested quantity is available."""
        return self.__quantity_in_stock >= quantity_to_check

    def get_inventory_value(self):
        """Calculates total stock value (price * quantity)."""
        return self.__product.price * self.__quantity_in_stock

    def get_inventory_details(self):
        """Returns full inventory details as string."""
        return (
            f"Inventory ID: {self.__inventory_id}, "
            f"Product: {self.__product.product_name}, "
            f"Stock: {self.__quantity_in_stock}, "
                                                f"Last    Updated:
{self.__last_stock_update.strftime('%Y-%m-%d %H:%M:%S')}"
        )

    def __str__(self):
        return self.get_inventory_details()

    # ---------- Static Utility Methods ----------
    @staticmethod
    def list_low_stock_products(inventory_list: list, threshold: int):
        """Lists products below threshold stock."""
                    return   [inv   for   inv   in   inventory_list   if
inv.get_quantity_in_stock() < threshold]

    @staticmethod
    def list_out_of_stock_products(inventory_list: list):
        """Lists all out-of-stock products."""
                    return   [inv   for   inv   in   inventory_list   if
inv.get_quantity_in_stock() == 0]

    @staticmethod
    def list_all_products(inventory_list: list):
        """Lists all inventory items."""
        return [str(inv) for inv in inventory_list]
```

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.

All the required classes, such as Customer, Product, Order, OrderDetail, and Inventory, were already created and implemented in Task 1 itself using proper object-oriented design.

Each class has:

- Clearly defined attributes with the right data types
- A constructor to initialize all the fields
- And meaningful methods to handle things like updating customer info, calculating subtotals, checking stock, and more

Since the core structure and logic of these classes are already handled earlier, Task 2 is considered complete and doesn't need any extra implementation here.

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).

All the entity classes in the application such as Customer, Product, Order, OrderDetail, and Inventory follows the OOP principle of Encapsulation.

- Each class has its attributes declared as private using double underscores (e.g., self.__price, self.__quantity_in_stock).
- Public getter and setter methods (via @property) are provided to control access to those private attributes.
- The setters include data validation logic, such as:
    - Price must be non-negative
    - Quantity must be a positive integer
    - Email or phone cannot be empty (in some cases)

This ensures data integrity and enforces controlled access to object state, which is the main goal of encapsulation.

Task 4:

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

1. Order 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 Customer, establishing a composition relationship. The Customer property provides access to the Customer object associated with the order.

The Order and OrderDetail classes use composition to include other objects as part of their structure.

- An order holds a Customer object, showing that an order *belongs to* a customer.
- OrderDetail holds a Product object, showing that an order detail *refers to* a specific product.

2. 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.

The OrderDetail class uses composition to include both an Order and a Product object.

- It has a private order attribute (type Order) and a product attribute (type Product).
- This means each order detail *belongs to* a specific order and *refers to* a specific product.

In the OrderDetail class of the TechShop application, I have implemented composition by including two objects as part of its attributes:

A. Order – represents the parent order that this detail belongs to

B. Product – represents the specific product that is being ordered in that line item

These objects are included using the following private attributes:

- __order (of type Order)
- __product (of type Product)

By doing this, each OrderDetail object tightly connects to a specific Order and a specific Product, which means:

- An order detail "has-a" Order
- An order detail "has-a" Product

This design follows the OOP principle of composition, where the OrderDetail class is composed of other fully-functional objects (Order and Product). This structure ensures that:

- We can trace every product to the exact order it belongs to
- We can retrieve product details (like price, name) directly from the product object
- The code is modular, easier to maintain, and logically organized.

Therefore, this approach makes the application highly scalable and realistic. It mimics how real-world billing systems work ,every item on a bill (order detail) refers to - *what* product was bought and *which* order it was bought

It also improves data integrity, as all order-product relationships are strictly maintained within the structure of the class using composition.

• 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.

3. 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.

In this TechShop project, I didn’t establish composition *inside* the Customer or Product classes themselves. These classes act more like independent building blocks or core entities.

Instead, they’re used as components in other classes like Order and OrderDetail. Here's what I mean:

- A Customer doesn’t need to contain orders — but an Order definitely needs a Customer.
- A Product doesn’t contain order history — but an OrderDetail absolutely needs a Product.

So, I’ve designed Customer and Product as self-contained classes. Then, I used composition in Order and OrderDetail by linking them with Customer and Product respectively.

This way, the relationships stay clean and logical:

- Order "has a" Customer
- OrderDetail "has a" Product

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

class InvalidDataException(Exception):

    def __init__(self, message="Invalid data provided."):

        super().__init__(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.

```
# exception/insufficient_stock_exception.py

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

• 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

class IncompleteOrderException(Exception):
    def __init__(self, message="Order is incomplete or missing data."):
        super().__init__(message)
```

• 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.

```
# exception/payment_failed_exception.py

class PaymentFailedException(Exception):
    def __init__(self, message="Payment could not be processed."):
        super().__init__(message)
```

• 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.

```
# exception/file_io_exception.py

class FileIOException(Exception):
    def __init__(self, message="File input/output error occurred."):
        super().__init__(message)
```

• 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

class DBConnectionException(Exception):
    def __init__(self, message="Failed to connect to the database."):
        super().__init__(message)
```

• 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

class ConcurrencyException(Exception):
        def __init__(self, message="Data conflict occurred due to
concurrent access."):
        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

class AuthenticationException(Exception):
      def __init__(self, message="Authentication failed. Please login
again."):
        super().__init__(message)
```

```python
# exception/authorization_exception.py

class AuthorizationException(Exception):
    def __init__(self, message="You are not authorized to perform this action."):
        super().__init__(message)
```

SUMMARY FOR THE EXCEPTIONS AND PURPOSES IN THE APPLICATION:

Exception File	Purpose
authentication_exception.py	Invalid login credentials
authorization_exception.py	Accessing unauthorised areas
concurrency_exception.py	Concurrent updates causing data conflicts
db_connection_exception.py	Failure to connect with MySQL
file_io_exception.py	File read/write issues
incomplete_order_exception.py	Missing info during order placement
insufficient_stock_exception.py	Not enough stock in inventory
invalid_data_exception.py	Validation errors, duplicates, and empty fields
payment_failed_exception.py	Payment gateway declined or invalid payment

Task 6:

Collections

• **Managing Product 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
from entity.product import Product
from util.db_conn_util import DBConnection
from exception.invalid_data_exception import InvalidDataException


class ProductServiceImpl:
    def add_product(self, product: Product):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            # Check for duplicate ID or name
             cursor.execute("SELECT * FROM products WHERE productid = %s
OR LOWER(productname) = %s",
                                                    (product.product_id,
product.product_name.lower()))
            if cursor.fetchone():
                    raise InvalidDataException(f"Product with ID or name
'{product.product_name}' already exists.")

            cursor.execute("""
                        INSERT INTO products (productid, productname,
description, price, category)
                VALUES (%s, %s, %s, %s, %s)
            """, (
                product.product_id,
                product.product_name,
                product.description,
                product.price,
                product.category
            ))
            conn.commit()
            print("✅ Product added successfully.")
        except Exception as e:
            if conn:
                conn.rollback()
                raise InvalidDataException(f"❌ Failed to add product:
{e}")
        finally:
            DBConnection.close_connection(conn)
```

```
        def update_product(self, product_id: int, new_price=None,
new_description=None):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            updates = []
            values = []

            if new_price is not None:
                updates.append("price = %s")
                values.append(new_price)

            if new_description is not None:
                updates.append("description = %s")
                values.append(new_description)

            if not updates:
                        raise InvalidDataException("No update values
provided.")

            values.append(product_id)
               query = f"UPDATE products SET {', '.join(updates)} WHERE
productid = %s"
            cursor.execute(query, tuple(values))
            conn.commit()

            if cursor.rowcount == 0:
                raise InvalidDataException("Product not found.")
            print("✅ Product updated successfully.")
        except Exception as e:
            if conn:
                conn.rollback()
              raise InvalidDataException(f"❌ Failed to update product:
{e}")
        finally:
            DBConnection.close_connection(conn)

                    def     remove_product(self,     product_id:     int,
existing_order_product_ids=None):
        conn = None
        try:
```

```
                    if existing_order_product_ids and product_id in
existing_order_product_ids:
                    raise InvalidDataException("Cannot remove product: It
is associated with existing orders.")

            conn = DBConnection.get_connection()
            cursor = conn.cursor()
             cursor.execute("DELETE FROM products WHERE productid = %s",
(product_id,))
            conn.commit()

            if cursor.rowcount == 0:
                raise InvalidDataException("Product not found.")
            print("✅ Product removed successfully.")
        except Exception as e:
            if conn:
                conn.rollback()
              raise InvalidDataException(f"❌ Failed to remove product:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def find_product_by_id(self, product_id: int):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
               cursor.execute("SELECT * FROM products WHERE productid =
%s", (product_id,))
            row = cursor.fetchone()
            if row:
                return Product(*row)
            return None
        except Exception as e:
               raise InvalidDataException(f"❌ Failed to fetch product:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def search_products_by_name(self, keyword: str):
        if not keyword.strip():
                    raise InvalidDataException("Search keyword cannot be
empty.")
```

```
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
                        cursor.execute("SELECT * FROM products WHERE
LOWER(productname) LIKE %s", (f"%{keyword.lower()}%",))
            rows = cursor.fetchall()

            if not rows:
                raise InvalidDataException(f"No products found matching
'{keyword}'.")

            result = []
            for row in rows:
                            product = Product(row[0], row[1], row[2],
float(row[3]), row[4])
                result.append(product)
            return result
        except Exception as e:
              raise InvalidDataException(f"❌ Error searching products:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def list_all_products(self):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM products")
            rows = cursor.fetchall()
                return [Product(*row).get_product_details() for row in
rows]
        except Exception as e:
                raise InvalidDataException(f"❌ Error listing products:
{e}")
        finally:
            DBConnection.close_connection(conn)
```

EXPLANATION:

In the ProductServiceImpl class, I manage the product catalog using database-backed collection logic. Operations such as:

- Adding new products (add_product())
- Updating product information (update_product())
- Removing products (remove_product())

are implemented in this class. These methods internally work like managing a list of product entries.

I also added exception handling:

- Prevent adding duplicate products
- Raise an error when trying to update or delete a non-existent product

• **Managing Orders List:**
o Challenge: Maintaining a list of customer orders (List).
o Scenario: Adding new orders, updating order statuses, and removing cancelled orders.
o Solution: Implement methods to add new orders, update order statuses, and remove cancelled orders. Ensure that updates are synchronised with inventory and payment records.

```
from entity.order import Order
from entity.order_detail import OrderDetail
from entity.product import Product
from entity.customer import Customer
from util.db_conn_util import DBConnection
from exception.invalid_data_exception import InvalidDataException
from          dao.implementation.inventory_service_impl          import
InventoryServiceImpl
from datetime import datetime


class OrderServiceImpl:
    def __init__(self):
        self.inventory_service = InventoryServiceImpl()

    def add_order(self, order: Order):
        conn = None
        try:
```

```
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            # Check if order ID already exists
              cursor.execute("SELECT * FROM orders WHERE orderid = %s",
(order.order_id,))
            if cursor.fetchone():
                                raise InvalidDataException(f"❌ Order ID
{order.order_id} already exists.")

            # Inventory validation
            for detail in order.order_details:
                productid = detail.product.product_id
                quantity = detail.quantity

                                                        inventory    =
self.inventory_service.get_inventory(productid)
                if not inventory:
                                raise InvalidDataException(f"❌ Product ID
{productid} not found in inventory.")
                if not inventory.is_product_available(quantity):
                    raise InvalidDataException(
                                    f"❌ Not enough stock for product ID
{productid}. "
                                      f"Requested: {quantity}, Available:
{inventory.get_quantity_in_stock()}"
                    )

            # Calculate total before inserting
            total_amount = order.calculate_total_amount()
            print(f"🧾 Total Order Amount: ₹{total_amount:.2f}")

            # Insert into orders table
            cursor.execute(
                """
                    INSERT INTO orders (orderid, customerid, orderdate,
totalamount, status)
                VALUES (%s, %s, %s, %s, %s)
                """,
                            (order.order_id, order.customer.customer_id,
order.order_date, total_amount, order.status)
            )
```

```
            # Insert order details and update inventory
            for detail in order.order_details:
                cursor.execute(
                    """
                       INSERT INTO orderdetails (orderdetailid, orderid,
productid, quantity)
                    VALUES (%s, %s, %s, %s)
                    """,
                                (detail.order_detail_id,  order.order_id,
detail.product.product_id, detail.quantity)
                )


self.inventory_service.process_order_detail(detail.product.product_id,
detail.quantity)

            conn.commit()
            print("✅ Order placed and inventory updated.")

        except Exception as e:
            if conn:
                conn.rollback()
            # RAISE to let main module decide what to do
            raise InvalidDataException(f"❌ Order failed: {e}")
        finally:
            DBConnection.close_connection(conn)

    def update_order_status(self, order_id: int, new_status: str):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
             cursor.execute("UPDATE orders SET status = %s WHERE orderid
= %s", (new_status, order_id))
            if cursor.rowcount == 0:
                raise InvalidDataException("Order not found.")
            conn.commit()
        except Exception as e:
            if conn:
                conn.rollback()
               raise InvalidDataException(f"❌ Failed to update order:
{e}")
        finally:
```

```
            DBConnection.close_connection(conn)

    def remove_cancelled_orders(self):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
                    cursor.execute("DELETE FROM orders WHERE status =
'cancelled'")
            conn.commit()
        except Exception as e:
            if conn:
                conn.rollback()
              raise InvalidDataException(f"❌ Failed to remove cancelled
orders: {e}")
        finally:
            DBConnection.close_connection(conn)

    def list_all_orders(self):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
                cursor.execute("SELECT orderid, customerid, orderdate,
totalamount, status FROM orders")
            rows = cursor.fetchall()
            result = []

               from dao.implementation.order_detail_service_impl import
OrderDetailServiceImpl
            order_detail_service = OrderDetailServiceImpl()

            for row in rows:
                dummy_customer = Customer(row[1], "", "", "", "", "")
                order = Order(row[0], dummy_customer, row[2], row[4])

                # ✅ Fetch and add order details to correctly calculate
total
                                                        details    =
order_detail_service.get_order_details_by_order_id(order.order_id)
                for detail in details:
                    order.add_order_detail(detail)
```

```
                result.append(order.get_order_details())
            return result
        except Exception as e:
            print("❌", e)
            return []
        finally:
            DBConnection.close_connection(conn)


    def get_orders_by_customer(self, customer_id: int):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
               cursor.execute("SELECT * FROM orders WHERE customerid =
%s", (customer_id,))
            rows = cursor.fetchall()
            result = []
               from dao.implementation.order_detail_service_impl import
OrderDetailServiceImpl
            order_detail_service = OrderDetailServiceImpl()

            for row in rows:
                dummy_customer = Customer(row[1], "", "", "", "", "")
                order = Order(row[0], dummy_customer, row[2], row[4])

                # ✅ Fetch order details and add to order
                                                        details    =
order_detail_service.get_order_details_by_order_id(order.order_id)
                for detail in details:
                    order.add_order_detail(detail)

                result.append(order)
            return result
        except Exception as e:
            print("❌", e)
            return []
        finally:
            DBConnection.close_connection(conn)


    def get_orders_sorted_by_date(self, ascending=True):
        conn = None
```

```
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            order_clause = "ASC" if ascending else "DESC"
               cursor.execute(f"SELECT * FROM orders ORDER BY orderdate
{order_clause}")
            return cursor.fetchall()
        except Exception as e:
                raise InvalidDataException(f"❌ Failed to sort orders:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def fetch_existing_order_details(self):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT orderdetailid FROM orderdetails")
            ids = [row[0] for row in cursor.fetchall()]
            return ids
        except Exception as e:
                raise InvalidDataException(f"❌ Failed to fetch order
detail IDs: {e}")
        finally:
            DBConnection.close_connection(conn)
```

In the OrderServiceImpl class, I manage all customer orders like a collection (similar to a List<Order>). I've implemented the core operations such as:

- Placing new orders using add_order()
- Updating the order status through update_order_status()
- Removing cancelled orders with remove_cancelled_orders()

These methods keep the order list updated, and also ensure synchronization with:

- Inventory (stock is updated using process_order_detail() when an order is placed) and Payments (each order is linked to a corresponding payment through PaymentServiceImpl)

I've also handled edge cases like:

- Preventing duplicate order IDs
- Checking stock before confirming an order
- Auto-generating the total order amount based on line items.

• 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.

*I've implemented sorting logic **inside my OrderServiceImpl** class using the method get_orders_sorted_by_date().*
This lets me retrieve orders in either ascending or descending order based on the date they were placed.
Internally, it runs a SQL query using ORDER BY orderdate ASC/DESC, making the sorting efficient directly from the database instead of doing it in Python.
This helps display recent orders first or track order history based on time.

• 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 synchronised with product additions and removals.

```
from util.db_conn_util import DBConnection
from entity.inventory import Inventory
from entity.product import Product
from exception.invalid_data_exception import InvalidDataException
from exception.insufficient_stock_exception import
InsufficientStockException
from datetime import date


class InventoryServiceImpl:
    def add_inventory(self, inventory: Inventory):
        conn = None
```

```
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            # Check if inventory already exists
              cursor.execute("SELECT * FROM inventory WHERE productid =
%s", (inventory.product.product_id,))
            if cursor.fetchone():
                  raise InvalidDataException(f"Inventory already exists
for Product ID {inventory.product.product_id}.")

            cursor.execute(
                """
                        INSERT INTO inventory (inventoryid, productid,
quantityinstock, laststockupdate)
                VALUES (%s, %s, %s, %s)
                """,
                (
                    inventory.inventory_id,
                    inventory.product.product_id,
                    inventory.quantity_in_stock,
                    inventory.last_stock_update
                )
            )
            conn.commit()
            print("✅ Inventory added.")
        except Exception as e:
            if conn:
                conn.rollback()
            print("❌", e)
        finally:
            DBConnection.close_connection(conn)

    def update_inventory_quantity(self, product_id: int, new_quantity:
int):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            cursor.execute(
                        "UPDATE inventory SET quantityinstock = %s,
laststockupdate = %s WHERE productid = %s",
                (new_quantity, date.today(), product_id)
```

```
            )
            if cursor.rowcount == 0:
                    raise InvalidDataException(f"No inventory found for
Product ID {product_id}.")
            conn.commit()
        except Exception as e:
            if conn:
                conn.rollback()
            print("❌", e)
        finally:
            DBConnection.close_connection(conn)

    def remove_inventory(self, product_id: int):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
               cursor.execute("DELETE FROM inventory WHERE productid =
%s", (product_id,))
            if cursor.rowcount == 0:
                    raise InvalidDataException(f"No inventory found for
Product ID {product_id}.")
            conn.commit()
        except Exception as e:
            if conn:
                conn.rollback()
            print("❌", e)
        finally:
            DBConnection.close_connection(conn)

    def get_inventory(self, product_id: int) -> Inventory:
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
                        cursor.execute("SELECT  inventoryid,  productid,
quantityinstock, laststockupdate FROM inventory WHERE productid = %s",
(product_id,))
            row = cursor.fetchone()
            if row:
                # ✅ Load product properly
                    from dao.implementation.product_service_impl import
ProductServiceImpl
```

```
                                                    product    =
ProductServiceImpl().find_product_by_id(row[1])
                if product:
                        return Inventory(row[0], product, row[2], row[3])
# ✅ last_stock_update included
            return None
        except Exception as e:
            print("❌", e)
            return None
        finally:
            DBConnection.close_connection(conn)

    def list_inventory_sorted_by_product_id(self):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
                cursor.execute("SELECT productid, quantityinstock FROM
inventory ORDER BY productid")
            rows = cursor.fetchall()
             return [f"Product ID: {row[0]} | Qty: {row[1]}" for row in
rows]
        except Exception as e:
            print("❌", e)
            return []
        finally:
            DBConnection.close_connection(conn)

     def process_order_detail(self, product_id: int, quantity_ordered:
int):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

             cursor.execute("SELECT quantityinstock FROM inventory WHERE
productid = %s", (product_id,))
            result = cursor.fetchone()
            if not result:
                    raise InvalidDataException("Product not found in
inventory.")

            available = result[0]
```

```
            if quantity_ordered > available:
                raise InsufficientStockException(
                                    f"Only {available} available, but
{quantity_ordered} requested."
                )

            new_quantity = available - quantity_ordered
            cursor.execute(
                        "UPDATE inventory SET quantityinstock = %s,
laststockupdate = %s WHERE productid = %s",
                (new_quantity, date.today(), product_id)
            )
            conn.commit()
        except Exception as e:
            if conn:
                conn.rollback()
            print("❌", e)
        finally:
            DBConnection.close_connection(conn)
```

I've implemented sorted inventory management inside the InventoryServiceImpl class using the list_inventory_sorted_by_product_id() method.
This mimics a SortedList<int, Inventory> by querying the inventory table and sorting it based on productid via SQL's ORDER BY clause.
This helps me quickly track stock levels per product, especially useful when displaying inventory or checking availability during order placement.
It keeps the inventory updated and in sync whenever new products are added, stock is updated, or products are removed.

• **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 process_order_detail(self, product_id: int, quantity_ordered: int):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
```

```
            cursor.execute("SELECT quantityinstock FROM inventory WHERE
productid = %s", (product_id,))
            result = cursor.fetchone()
            if not result:
                    raise InvalidDataException("Product not found in
inventory.")

            available = result[0]
            if quantity_ordered > available:
                raise InsufficientStockException(
                                    f"Only {available} available, but
{quantity_ordered} requested."
                )

            new_quantity = available - quantity_ordered
            cursor.execute(
                        "UPDATE inventory SET quantityinstock = %s,
laststockupdate = %s WHERE productid = %s",
                (new_quantity, date.today(), product_id)
            )
            conn.commit()
        except Exception as e:
            if conn:
                conn.rollback()
            print("❌", e)
        finally:
            DBConnection.close_connection(conn)
```

While placing an order, I made sure the system reduces the product stock accurately using the process_order_detail() method inside InventoryServiceImpl.
It checks the available quantity and updates it by subtracting the ordered units. If the stock isn't enough, it raises a custom InsufficientStockException.
This ensures the inventory stays consistent and prevents users from ordering items that are out of 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.

```
from entity.product import Product
from util.db_conn_util import DBConnection
from exception.invalid_data_exception import InvalidDataException


class ProductServiceImpl:
    def add_product(self, product: Product):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            # Check for duplicate ID or name
             cursor.execute("SELECT * FROM products WHERE productid = %s
OR LOWER(productname) = %s",
                                                    (product.product_id,
product.product_name.lower()))
            if cursor.fetchone():
                    raise InvalidDataException(f"Product with ID or name
'{product.product_name}' already exists.")

            cursor.execute("""
                        INSERT INTO products (productid, productname,
description, price, category)
                VALUES (%s, %s, %s, %s, %s)
            """, (
                product.product_id,
                product.product_name,
                product.description,
                product.price,
                product.category
            ))
            conn.commit()
            print("✅ Product added successfully.")
        except Exception as e:
            if conn:
                conn.rollback()
                raise InvalidDataException(f"❌ Failed to add product:
{e}")
        finally:
            DBConnection.close_connection(conn)
```

```
        def update_product(self, product_id: int, new_price=None,
new_description=None):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            updates = []
            values = []

            if new_price is not None:
                updates.append("price = %s")
                values.append(new_price)

            if new_description is not None:
                updates.append("description = %s")
                values.append(new_description)

            if not updates:
                        raise InvalidDataException("No update values
provided.")

            values.append(product_id)
               query = f"UPDATE products SET {', '.join(updates)} WHERE
productid = %s"
            cursor.execute(query, tuple(values))
            conn.commit()

            if cursor.rowcount == 0:
                raise InvalidDataException("Product not found.")
            print("✅ Product updated successfully.")
        except Exception as e:
            if conn:
                conn.rollback()
              raise InvalidDataException(f"❌ Failed to update product:
{e}")
        finally:
            DBConnection.close_connection(conn)

                    def     remove_product(self,     product_id:     int,
existing_order_product_ids=None):
        conn = None
        try:
```

```
                    if existing_order_product_ids and product_id in
existing_order_product_ids:
                    raise InvalidDataException("Cannot remove product: It
is associated with existing orders.")

            conn = DBConnection.get_connection()
            cursor = conn.cursor()
             cursor.execute("DELETE FROM products WHERE productid = %s",
(product_id,))
            conn.commit()

            if cursor.rowcount == 0:
                raise InvalidDataException("Product not found.")
            print("✅ Product removed successfully.")
        except Exception as e:
            if conn:
                conn.rollback()
              raise InvalidDataException(f"❌ Failed to remove product:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def find_product_by_id(self, product_id: int):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
               cursor.execute("SELECT * FROM products WHERE productid =
%s", (product_id,))
            row = cursor.fetchone()
            if row:
                return Product(*row)
            return None
        except Exception as e:
               raise InvalidDataException(f"❌ Failed to fetch product:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def search_products_by_name(self, keyword: str):
        if not keyword.strip():
                    raise InvalidDataException("Search keyword cannot be
empty.")
```

```
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
                        cursor.execute("SELECT * FROM products WHERE
LOWER(productname) LIKE %s", (f"%{keyword.lower()}%",))
            rows = cursor.fetchall()

            if not rows:
                raise InvalidDataException(f"No products found matching
'{keyword}'.")

            result = []
            for row in rows:
                            product = Product(row[0], row[1], row[2],
float(row[3]), row[4])
                result.append(product)
            return result
        except Exception as e:
              raise InvalidDataException(f"❌ Error searching products:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def list_all_products(self):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM products")
            rows = cursor.fetchall()
                return [Product(*row).get_product_details() for row in
rows]
        except Exception as e:
                raise InvalidDataException(f"❌ Error listing products:
{e}")
        finally:
            DBConnection.close_connection(conn)
```

I implemented a product search feature in the ProductServiceImpl class so customers can easily find products by typing keywords from the product name.
The search is flexible, case-insensitive, and returns all matching results.
I've also handled cases where the search term is empty or invalid by raising an exception or returning an empty list.
This gives the customer a smoother and smarter shopping experience, just like modern e-commerce platforms.

• **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: Product):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

            # Check for duplicate ID or name
             cursor.execute("SELECT * FROM products WHERE productid = %s
OR LOWER(productname) = %s",
                                                    (product.product_id,
product.product_name.lower()))
            if cursor.fetchone():
                    raise InvalidDataException(f"Product with ID or name
'{product.product_name}' already exists.")

            cursor.execute("""
                        INSERT INTO products (productid, productname,
description, price, category)
                VALUES (%s, %s, %s, %s, %s)
            """, (
                product.product_id,
                product.product_name,
                product.description,
                product.price,
                product.category
            ))
            conn.commit()
            print("✅ Product added successfully.")
```

```
        except Exception as e:
            if conn:
                conn.rollback()
                raise InvalidDataException(f"❌ Failed to add product:
{e}")
        finally:
            DBConnection.close_connection(conn)
```

In my ProductServiceImpl, I added a validation step before inserting any new product into the database. This checks whether a product with the same name already exists. If it does, the system raises an InvalidDataException to prevent duplicates. This avoids confusion for customers, keeps the catalog clean, and ensures reliable product tracking.

• **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.

```
from entity.payment import Payment
from util.db_conn_util import DBConnection
from exception.invalid_data_exception import InvalidDataException
from exception.payment_failed_exception import PaymentFailedException

class PaymentServiceImpl:
    def record_payment(self, payment: Payment):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()

                cursor.execute("SELECT * FROM payment WHERE paymentid =
%s", (payment.payment_id,))
            if cursor.fetchone():
                            raise InvalidDataException(f"Payment with ID
{payment.payment_id} already exists.")

            cursor.execute("""
                        INSERT INTO payment (paymentid, orderid, amount,
paymentdate, paymentmethod, status)
                VALUES (%s, %s, %s, %s, %s, %s)
```

```
            """, (
                payment.payment_id,
                payment.order.order_id,
                payment.amount,
                payment.payment_date.strftime('%Y-%m-%d'),
                payment.payment_method,
                payment.status
            ))

            conn.commit()
            print("✅ Payment recorded successfully.")
        except Exception as e:
            if conn:
                conn.rollback()
              raise InvalidDataException(f"❌ Failed to record payment:
{e}")
        finally:
            DBConnection.close_connection(conn)

    def update_payment_status(self, payment_id: int, new_status: str):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
                 cursor.execute("UPDATE payment SET status = %s WHERE
paymentid = %s", (new_status, payment_id))

            if cursor.rowcount == 0:
                raise InvalidDataException("Payment not found.")

            if new_status.lower() == "failed":
                    raise PaymentFailedException("Payment failed. Please
try again.")

            conn.commit()
            print("✅ Payment status updated.")
        except Exception as e:
            if conn:
                conn.rollback()
               raise InvalidDataException(f"❌ Failed to update payment
status: {e}")
        finally:
            DBConnection.close_connection(conn)
```

```
    def find_payment_by_order_id(self, order_id: int):
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
             cursor.execute("SELECT * FROM payment WHERE orderid = %s",
(order_id,))
            row = cursor.fetchone()
            if row:
                from entity.order import Order
                dummy_order = Order(row[1], None, row[3])
                    return Payment(row[0], dummy_order, row[2], row[3],
row[4], row[5])
            return None
        except Exception as e:
             raise InvalidDataException(f"❌ Error fetching payment for
order {order_id}: {e}")
        finally:
            DBConnection.close_connection(conn)

    def list_all_payments(self):
        conn = None
        try:
            conn = DBConnection.get_connection()
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM payment")
            rows = cursor.fetchall()
            result = []
            for row in rows:
                from entity.order import Order
                dummy_order = Order(row[1], None, row[3])
                 payment = Payment(row[0], dummy_order, row[2], row[3],
row[4], row[5])
                result.append(payment.get_payment_info())
            return result
        except Exception as e:
                raise InvalidDataException(f"❌ Error listing payments:
{e}")
        finally:
            DBConnection.close_connection(conn)
```

In my TechShop application, I manage payments using a collection-like approach via the PaymentServiceImpl class.

Each payment is tied to an order, and I ensure that the system records payments, tracks their statuses (e.g., Paid, Pending, Failed), and retrieves details efficiently. The code also validates inputs to avoid duplicates and ensure data consistency.

Key methods include:

- record_payment(): Saves payment details for an order.
- update_payment_status(): Updates the status (e.g., after COD is delivered).
- find_payment_by_order_id(): Fetches payment for a specific order.
- list_all_payments(): Returns all recorded payments.

This ensures payment and order records are always in sync, with proper exception handling.

• **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.

```
for detail in order.order_details:
    productid = detail.product.product_id
    quantity = detail.quantity

    inventory = self.inventory_service.get_inventory(productid)
    if not inventory:
            raise InvalidDataException(f"❌ Product ID {productid} not
found in inventory.")

    if not inventory.is_product_available(quantity):
        raise InvalidDataException(
            f"❌ Not enough stock for product ID {productid}. "
                                f"Requested: {quantity}, Available:
{inventory.get_quantity_in_stock()}"
        )
```

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.

```
import mysql.connector
from util.db_property_util import load_db_properties
from exception.db_connection_exception import DBConnectionException

class DBConnection:
    @staticmethod
    def get_connection():
        """Return a fresh MySQL connection."""
        try:
            props = load_db_properties()
            return mysql.connector.connect(
                host=props['host'],
                port=int(props['port']),
                user=props['user'],
                password=props['password'],
                database=props['database']
            )
        except Exception as e:
                raise DBConnectionException(f"❌ Database connection
failed: {e}")

    @staticmethod
    def close_connection(conn):
        """Close the given connection."""
        try:
            if conn and conn.is_connected():
                conn.close()
        except Exception as e:
            print("❌ Error closing DB connection:", e)
```

I created a DBConnection utility class in db_conn_util.py that handles connecting to and disconnecting from the MySQL database (techshop).

Each entity class (like Customer, Product, Order, etc.) is implemented with separate DAO classes that perform full CRUD operations using SQL queries. These DAO classes interact with the database through the DBConnection utility.

USE CASES:-

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.

```
PS C:\Users\Aishwarya\OneDrive\Desktop\hexaware\ASSIGNMENT 1\TechShop> python main/main_module.py
>>

========= TECHSHOP LOGIN =========
1. Customer
2. Admin
0. Exit
==================================
Enter your choice: 1

===== CUSTOMER MENU =====
1. Register
2. Login
0. Back
Enter your choice: 1
Customer ID: 12
First Name: Laura
Last Name: Jacob
Email: laurajacob@gmail.com
Phone: 9456712873
Address:  11/33, Nynan Street, Mandaveli, Chennai
✅ Customer inserted into MySQL database.
✅ Registered successfully. Please login.
```

BEFORE REGISTRATION:

```
mysql> select * from customer;
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------------+------------+
| customerid | firstname | lastname | email                 | phone      | address                                             | ordercount |
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------------+------------+
|          1 | Neha      | Ashok    | nehaashok@gmail.com   | 9856423715 | No.21, Richard Street, Chennai                      | 1          |
|          2 | Arjun     | Nagesh   | arjunnagesh@gmail.com | 8435127648 | Plot 12/K, Anugraha Apartments, Anna Nagar, Chennai | 1          |
|          3 | Sophie    | Victor   | sophievic@gmail.com   | 6473519421 | No.31/A, Ganga Roads,Parrys, Chennai                | 1          |
|          4 | Syed      | Ahmed    | syedahmed@gmail.com   | 8429516443 | FF1,Block2, Lake ViewApartments,Adayar, Chennai     | 0          |
|          5 | Kiran     | Ghosh    | kiranghosh@gmail.com  | 9600044476 | C-34,Block22, Shanti Park,Egmore, Chennai           | 1          |
|          6 | Johnson   | Jacob    | johnsonjacob@gmail.com | 7331448349 | 11/33, Nynan Street, Mandaveli, Chennai             | 1          |
|          7 | Sriram    | Sai      | sriramsai@gmail.com   | 9487916253 | Flat 3B,Silver Residency, Moggapair, Chennai        | 1          |
|          8 | Nainika   | Menon    | nainikamenon@gmail.com | 8465137642 | 7A, Brigade Residency, Perungudi, Chennai           | 1          |
|          9 | Tanya     | Reddy    | tanyareddy@gmail.com  | 6793451287 | Plot 16,Janani Enclave, Padur, Chennai              | 1          |
|         10 | Swarna    | Shree    | swarnashree@gmail.com | 9786945312 | 65/78, Nungabamkkam, Chennai                        | 1          |
|         11 | Lakshmi   | Naren    | lakshminaren@gmail.com | 9427518643 | Flat 13D, Violet Meadows, Velachery, Chennai        | 0          |
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------------+------------+
11 rows in set (0.18 sec)
```

AFTER REGISTRATION:

```
mysql> select * from customer;
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------------+------------+
| customerid | firstname | lastname | email                 | phone      | address                                             | ordercount |
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------------+------------+
|          1 | Neha      | Ashok    | nehaashok@gmail.com   | 9856423715 | No.21, Richard Street, Chennai                      | 1          |
|          2 | Arjun     | Nagesh   | arjunnagesh@gmail.com | 8435127648 | Plot 12/K, Anugraha Apartments, Anna Nagar, Chennai | 1          |
|          3 | Sophie    | Victor   | sophievic@gmail.com   | 6473519421 | No.31/A, Ganga Roads,Parrys, Chennai                | 1          |
|          4 | Syed      | Ahmed    | syedahmed@gmail.com   | 8429516443 | FF1,Block2, Lake ViewApartments,Adayar, Chennai     | 0          |
|          5 | Kiran     | Ghosh    | kiranghosh@gmail.com  | 9600044476 | C-34,Block22, Shanti Park,Egmore, Chennai           | 1          |
|          6 | Johnson   | Jacob    | johnsonjacob@gmail.com | 7331448349 | 11/33, Nynan Street, Mandaveli, Chennai             | 1          |
|          7 | Sriram    | Sai      | sriramsai@gmail.com   | 9487916253 | Flat 3B,Silver Residency, Moggapair, Chennai        | 1          |
|          8 | Nainika   | Menon    | nainikamenon@gmail.com | 8465137642 | 7A, Brigade Residency, Perungudi, Chennai           | 1          |
|          9 | Tanya     | Reddy    | tanyareddy@gmail.com  | 6793451287 | Plot 16,Janani Enclave, Padur, Chennai              | 1          |
|         10 | Swarna    | Shree    | swarnashree@gmail.com | 9786945312 | 65/78, Nungabamkkam, Chennai                        | 1          |
|         11 | Lakshmi   | Naren    | lakshminaren@gmail.com | 9427518643 | Flat 13D, Violet Meadows, Velachery, Chennai        | 0          |
|         12 | Laura     | Jacob    | laurajacob@gmail.com  | 9456712873 |  11/33, Nynan Street, Mandaveli, Chennai            | 0          |
+------------+-----------+----------+-----------------------+------------+-----------------------------------------------------+------------+
12 rows in set (0.01 sec)
```

- Customerid = 12, Name = Laura, has been added to the database after registration.

2: Product Catalogue Management

Description: TechShop regularly updates its product catalogue 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 catalogue. Implement database connectivity to update product information. Handle changes in product details and ensure data consistency.

```
========== ADMIN MENU ==========
1. View All Orders
2. Update Order Status
3. Add New Product
4. Remove Discontinued Product
5. Update Product Info (Price/Description)
6. Update Inventory Stock
7. Generate Sales Report
0. Logout
Enter your choice: 5
Enter Product ID to update: 1006
Enter new price (leave blank to skip): 10110.00
Enter new description (leave blank to skip):
✅ Product updated successfully.
```

BEFORE UPDATE:

```
mysql> select * from products;
+-----------+---------------------------+-----------------------------------------------------------------------+-----------+-------------+
| productid | productname               | description                                                           | price     | category    |
+-----------+---------------------------+-----------------------------------------------------------------------+-----------+-------------+
|      1001 | redmi note 13             | 6.6-inch AMOLED display, 128GB storage, 5G support                    |  17598.90 | Smartphone  |
|      1002 | boat airdopes 161         | bluetooth 5.2, up to 40 hrs playback, fast charging                   |   1428.90 | Earbuds     |
|      1003 | dell inspiron 15          | 15.6-inch FHD, i5 12th Gen, 512GB SSD, 8GB RAM                        |  57198.90 | Laptop      |
|      1004 | logitech m331 silent mouse | 2.4GHz wireless, ergonomic, 18-month battery                          |    988.90 | Mouse       |
|      1005 | lg ultragear 27-inch      | QHD gaming monitor, 144Hz, IPS panel                                  |  26398.90 | Monitor     |
|      1006 | jbl flip 6                | portable bluetooth speaker, IP67 waterproof, 12 hrs battery           |   9898.90 | Speaker     |
```

AFTER UPDATE:

```
mysql> select * from products;
+-----------+---------------------------+-----------------------------------------------------------------------+-----------+-------------+
| productid | productname               | description                                                           | price     | category    |
+-----------+---------------------------+-----------------------------------------------------------------------+-----------+-------------+
|      1001 | redmi note 13             | 6.6-inch AMOLED display, 128GB storage, 5G support                    |  17598.90 | Smartphone  |
|      1002 | boat airdopes 161         | bluetooth 5.2, up to 40 hrs playback, fast charging                   |   1428.90 | Earbuds     |
|      1003 | dell inspiron 15          | 15.6-inch FHD, i5 12th Gen, 512GB SSD, 8GB RAM                        |  57198.90 | Laptop      |
|      1004 | logitech m331 silent mouse | 2.4GHz wireless, ergonomic, 18-month battery                          |    988.90 | Mouse       |
|      1005 | lg ultragear 27-inch      | QHD gaming monitor, 144Hz, IPS panel                                  |  26398.90 | Monitor     |
|      1006 | jbl flip 6                | portable bluetooth speaker, IP67 waterproof, 12 hrs battery           |  10110.00 | Speaker     |
```

- *The price of the productid = 1006 has changed from rs. 9898/- to rs. 10110/-*

3: Placing Customer Orders

Description: Customers browse the product catalogue 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.

```
========= CUSTOMER MENU =========
1. View Product Catalog
2. Search Products
3. Place Order
4. Track My Orders
5. Update My Account
6. Logout
================================
Enter your choice: 3
Product ID: 1002
Quantity: 2
Add more products? (y/n): n
🧾 Total Order Amount: ₹2857.80
✅ Order placed and inventory updated.
Enter payment method (COD/UPI/NetBanking): upi
✅ Payment recorded successfully.
✅ Order placed and payment recorded.
```

```
mysql> select * from orders;
+---------+------------+------------+-------------+------------------+
| orderid | customerid | orderdate  | totalamount | status           |
+---------+------------+------------+-------------+------------------+
|    2001 |          1 | 2025-06-01 |    19027.80 | Packing          |
|    2002 |          3 | 2025-06-02 |      988.90 | Packing          |
|    2004 |          2 | 2025-06-04 |    79196.70 | Packing          |
|    2006 |          6 | 2025-06-05 |     1978.90 | Packing          |
|    2007 |          7 | 2025-06-05 |    17598.90 | Packing          |
|    2008 |          8 | 2025-06-06 |    35197.80 | Packing          |
|    2009 |          9 | 2025-06-07 |     1299.00 | Packing          |
|    2010 |         10 | 2025-06-08 |    22999.00 | Packing          |
|    2011 |          5 | 2025-06-10 |    57198.90 | Out for Delivery |
|    2012 |          4 | 2025-06-26 |   129999.00 | Packing          |
|    2014 |          6 | 2025-06-26 |   129999.00 | Packing          |
|    2015 |         11 | 2025-06-26 |    78999.00 | Packing          |
|    2016 |         10 | 2025-06-26 |     4999.00 | Packing          |
|    2017 |          7 | 2025-06-26 |   139999.00 | Delivered        |
|    2018 |         12 | 2025-06-26 |     2857.80 | Packing          |
+---------+------------+------------+-------------+------------------+
15 rows in set (0.05 sec)
```

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.

```
========= CUSTOMER MENU =========
1. View Product Catalog
2. Search Products
3. Place Order
4. Track My Orders
5. Update My Account
6. Logout
================================
Enter your choice: 4

=======================================
Order ID: 2018
Customer: Laura Jacob
Order Date: 2025-06-26
Status: Packing
OrderDetail ID: 3018, Product: boat airdopes 161, Price: ₹1428.90, Quantity: 2, Discount: 0.0%, Subtotal: ₹2857.80
Total Amount: ₹2857.80
💳 Payment Method: UPI
📅 Payment Date: 2025-06-26
✅ Payment Status: Paid
💰 Amount Paid: ₹2857.80
---------------------------------------
```

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.

1. **Adding new product**

```
========= ADMIN MENU =========
1. View All Orders
2. Update Order Status
3. Add New Product
4. Remove Discontinued Product
5. Update Product Info (Price/Description)
6. Update Inventory Stock
7. Generate Sales Report
0. Logout
Enter your choice: 3
Product ID: 1023
Product Name: Apple iPad Air 11" with M3 chip
Description: Liquid Retina Display, 256GB, 12MP Front/Back Camera, Wi-Fi 6E, Touch ID
Price: 65999.00
Category: Tablet
✅ Product added successfully.
```

In MySql Database:

```
|      1021 | HP Envy x360                      | 14-inch 2-in-1 touch, Ryzen 5, 16GB RAM, 512GB SSD                                  |  78999.00 | Laptop     |
|      1022 | ASUS Vivobook 16X 13th Gen        | Intel Core i5-13420H Processor 2.1 GHz,16.0-inch, FHD+                              |  68990.00 | Laptop     |
|      1023 | Apple iPad Air 11" with M3 chip   | Liquid Retina Display, 256GB, 12MP Front/Back Camera, Wi-Fi 6E, Touch ID             |  65999.00 | Tablet     |
+-----------+-----------------------------------+-------------------------------------------------------------------------------------+-----------+------------+
23 rows in set (0.00 sec)
```

2. **Updating stock levels:**

```
========= ADMIN MENU =========
1. View All Orders
2. Update Order Status
3. Add New Product
4. Remove Discontinued Product
5. Update Product Info (Price/Description)
6. Update Inventory Stock
7. Generate Sales Report
0. Logout
Enter your choice: 6
Enter Product ID to update stock: 1016
Enter new stock quantity: 40
✅ Stock updated.
```

In MySql Database:

BEFORE UPDATION:

```
|        4010 |      1010 |                 7 | 2025-06-10       |
|        4011 |      1015 |                 5 | 2025-06-25       |
|        4012 |      1016 |                 4 | 2025-06-26       |
```

AFTER UPDATION:

```
|        4011 |      1015 |                 5 | 2025-06-25       |
|        4012 |      1016 |                40 | 2025-06-26       |
|        4013 |      1017 |                10 | 2025-06-25       |
```

- *Changed the productid = 1016 stock to 40 from 4*

3. Removing discontinued items:

```
========== ADMIN MENU ==========
1. Add New Product
2. Remove Discontinued Product
3. Update Inventory Stock
4. Generate Sales Report
0. Logout
Enter your choice: 2
Enter Product ID to remove: 1011
✅ Product removed successfully.
✅ Product removed.
```

In MySql Database:

```
mysql> select * from inventory;
+-------------+-----------+-----------------+-----------------+
| inventoryid | productid | quantityinstock | laststockupdate |
+-------------+-----------+-----------------+-----------------+
|        4001 |      1001 |              20 | 2025-06-10      |
|        4002 |      1002 |              48 | 2025-06-26      |
|        4003 |      1003 |              10 | 2025-06-10      |
|        4004 |      1004 |              30 | 2025-06-10      |
|        4005 |      1005 |              25 | 2025-06-10      |
|        4006 |      1006 |              15 | 2025-06-10      |
|        4007 |      1007 |              18 | 2025-06-10      |
|        4008 |      1008 |              12 | 2025-06-10      |
|        4009 |      1009 |              30 | 2025-06-10      |
|        4010 |      1010 |               7 | 2025-06-10      |
|        4011 |      1015 |               5 | 2025-06-25      |
|        4012 |      1016 |               4 | 2025-06-26      |
|        4013 |      1017 |              10 | 2025-06-25      |
|        4014 |      1018 |               6 | 2025-06-25      |
|        4015 |      1019 |              19 | 2025-06-26      |
|        4016 |      1020 |              14 | 2025-06-26      |
|        4017 |      1021 |              21 | 2025-06-26      |
|        5022 |      1012 |              54 | 2025-06-26      |
+-------------+-----------+-----------------+-----------------+
18 rows in set (0.00 sec)

mysql> select * from products;
+-----------+---------------------------+-----------------------------------------------------------------+-----------+------------+
| productid | productname               | description                                                     | price     | category   |
+-----------+---------------------------+-----------------------------------------------------------------+-----------+------------+
|      1001 | redmi note 13             | 6.6-inch AMOLED display, 128GB storage, 5G support              |  17598.90 | Smartphone |
|      1002 | boat airdopes 161         | bluetooth 5.2, up to 40 hrs playback, fast charging             |   1428.90 | Earbuds    |
|      1003 | dell inspiron 15          | 15.6-inch FHD, i5 12th Gen, 512GB SSD, 8GB RAM                  |  57198.90 | Laptop     |
|      1004 | logitech m331 silent mouse | 2.4GHz wireless, ergonomic, 18-month battery                    |    988.90 | Mouse      |
|      1005 | lg ultragear 27-inch      | QHD gaming monitor, 144Hz, IPS panel                            |  26398.90 | Monitor    |
|      1006 | jbl flip 6                | portable bluetooth speaker, IP67 waterproof, 12 hrs battery     |   9898.90 | Speaker    |
|      1007 | fire-boltt ninja call pro | smartwatch with bluetooth calling, health tracking              |   1978.90 | Smartwatch |
|      1008 | samsung t7 1tb ssd        | portable external SSD, USB 3.2, 1050MB/s                        |  10448.90 | Storage    |
|      1009 | mi wireless charger 20w   | fast wireless charging pad, Qi certified                        |   1428.90 | Charger    |
|      1010 | sony wh-1000xm4           | over-ear noise cancelling headphones, 30h battery               |  25298.90 | Headphones |
|      1012 | iPhone 15 Pro Max         | 6.7-inch Super Retina XDR, A17 Pro chip, 256GB                  | 139999.00 | Smartphone |
|      1013 | iPhone 14                 | 6.1-inch OLED, A15 chip, 128GB storage, 5G enabled              |  74999.00 | Smartphone |
|      1014 | MacBook Air M2            | 13.6-inch Liquid Retina, 8GB RAM, 256GB SSD                     | 114900.00 | Laptop     |
|      1015 | MacBook Pro M3            | 14-inch Liquid Retina XDR, 16GB RAM, 512GB SSD                  | 189900.00 | Laptop     |
|      1016 | Samsung Galaxy S24 Ultra  | 6.8-inch QHD+ AMOLED, 12GB RAM, 256GB                           | 129999.00 | Smartphone |
|      1017 | Samsung Galaxy A54        | 6.4-inch Super AMOLED, Exynos 1380, 8GB RAM                     |  38999.00 | Smartphone |
|      1018 | Lenovo Legion 5 Pro       | 16-inch QHD, Ryzen 7, RTX 3060, 1TB SSD                         | 129999.00 | Laptop     |
|      1019 | Realme Buds Air 5 Pro     | ANC earbuds with 50db noise cancellation, LDAC codec            |   4999.00 | Earbuds    |
|      1020 | OnePlus Nord CE4          | Snapdragon 7 Gen 3, AMOLED, 8GB RAM, 128GB                      |  24999.00 | Smartphone |
|      1021 | HP Envy x360              | 14-inch 2-in-1 touch, Ryzen 5, 16GB RAM, 512GB SSD              |  78999.00 | Laptop     |
|      1022 | ASUS Vivobook 16X 13th Gen | Intel Core i5-13420H Processor 2.1 GHz,16.0-inch, FHD+          |  68990.00 | Laptop     |
+-----------+---------------------------+-----------------------------------------------------------------+-----------+------------+
21 rows in set (0.00 sec)
```

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.

```
========= ADMIN MENU =========
1. View All Orders
2. Update Order Status
3. Add New Product
4. Remove Discontinued Product
5. Update Product Info (Price/Description)
6. Update Inventory Stock
7. Generate Sales Report
0. Logout
Enter your choice: 7

==========  SALES REPORT ==========
 Order ID: 2015 |  Date: 2025-06-26 |  Total: ₹78999.00 |  Payment: Paid | Paid: ₹78999.00
 Order ID: 2016 |  Date: 2025-06-26 |  Total: ₹4999.00 |  Payment: Paid | Paid: ₹4999.00
 Order ID: 2018 |  Date: 2025-06-26 |  Total: ₹2857.80 |  Payment: Paid | Paid: ₹2857.80
 Order ID: 2017 |  Date: 2025-06-26 |  Total: ₹139999.00 |  Payment: Paid | Paid: ₹139999.00
 Order ID: 2014 |  Date: 2025-06-26 |  Total: ₹129999.00 |  Payment: Pending | Paid: ₹129999.00
 Order ID: 2011 |  Date: 2025-06-10 |  Total: ₹57198.90 |  Payment: Paid | Paid: ₹57198.90
 Order ID: 2009 |  Date: 2025-06-07 |  Total: ₹1299.00 |  Payment: Paid | Paid: ₹1299.00
 Order ID: 2008 |  Date: 2025-06-06 |  Total: ₹35197.80 |  Payment: Paid | Paid: ₹35197.80
 Order ID: 2006 |  Date: 2025-06-05 |  Total: ₹1978.90 |  Payment: Paid | Paid: ₹1978.90
 Order ID: 2007 |  Date: 2025-06-05 |  Total: ₹17598.90 |  Payment: Paid | Paid: ₹17598.90
 Order ID: 2004 |  Date: 2025-06-04 |  Total: ₹79196.70 |  Payment: Paid | Paid: ₹79196.70
 Order ID: 2002 |  Date: 2025-06-02 |  Total: ₹988.90 |  Payment: Paid | Paid: ₹988.90
 Order ID: 2001 |  Date: 2025-06-01 |  Total: ₹19027.80 |  Payment: Paid | Paid: ₹19027.80
====================================
```

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.

```
========== CUSTOMER MENU ==========
1. View Product Catalog
2. Search Products
3. Place Order
4. Track My Orders
5. Update My Account
6. Logout
==================================
Enter your choice: 5
Leave field blank to skip.
New Email: laurajacob22404@gmail.com
New Phone: 9456712873
New Address: 11/33, Nynan Street, Mandaveli, Chennai
✅ Customer info updated in DB.
✅ Customer info updated.
```

```
|        12 | Laura     | Jacob    | laurajacob22404@gmail.com | 9456712873 | 11/33, Nynan Street, Mandaveli, Chennai
12 rows in set (0.02 sec)
```

Note: Email updated!

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.

Product catalogue:

```
========= CUSTOMER MENU =========
1. View Product Catalog
2. Search Products
3. Place Order
4. Track My Orders
5. Update My Account
6. Logout
================================
Enter your choice: 1
Product ID: 1001
Name: redmi note 13
Description: 6.6-inch AMOLED display, 128GB storage, 5G support
Price: ₹17598.90
Category: Smartphone
Product ID: 1002
Name: boat airdopes 161
Description: bluetooth 5.2, up to 40 hrs playback, fast charging
Price: ₹1428.90
Category: Earbuds
Product ID: 1003
Name: dell inspiron 15
Description: 15.6-inch FHD, i5 12th Gen, 512GB SSD, 8GB RAM
Price: ₹57198.90
Category: Laptop
Product ID: 1004
Name: logitech m331 silent mouse
Description: 2.4GHz wireless, ergonomic, 18-month battery
Price: ₹988.90
Category: Mouse
```

Search recommendations:

Search keyword: iphone
Displayed 3 results that has keyword iphone

```
========= CUSTOMER MENU =========
1. View Product Catalog
2. Search Products
3. Place Order
4. Track My Orders
5. Update My Account
6. Logout
================================
Enter your choice: 2
Enter product name to search: iphone
Product ID: 1011
Name: Iphone 14 plus
Description: 6.7-inch Super Retina XDR display, A15 Bionic chip, 128GB storage, dual camera
Price: ₹72999.00
Category: Smartphone
Product ID: 1012
Name: iPhone 15 Pro Max
Description: 6.7-inch Super Retina XDR, A17 Pro chip, 256GB
Price: ₹139999.00
Category: Smartphone
Product ID: 1013
Name: iPhone 14
Description: 6.1-inch OLED, A15 chip, 128GB storage, 5G enabled
Price: ₹74999.00
Category: Smartphone
```