**TechShop - An electronic gadgets shop**

**--------------------------------------------------------**

**Task 1: Database Design**

**1**. Create the database named "TechShop"

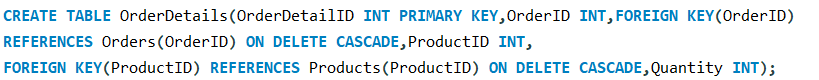


**2**. Define the schema for the Customers, Products, Orders, OrderDetails and Inventory tables based on the provided schema.



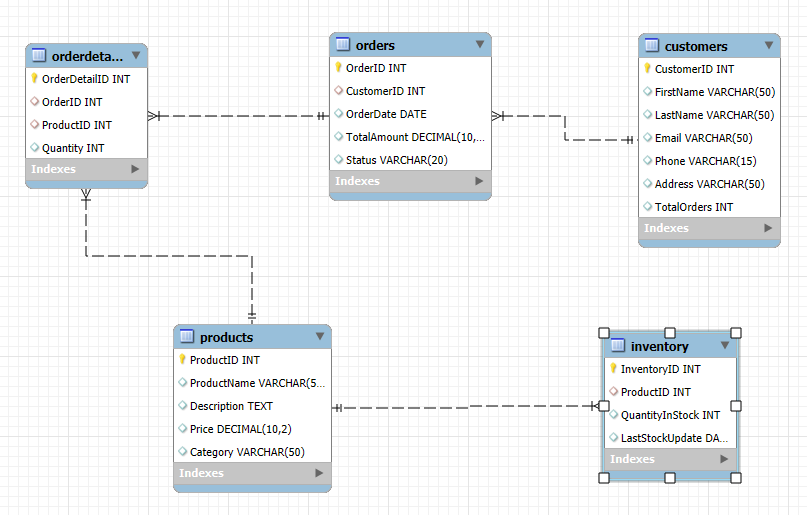








**3**. Create an ERD (Entity Relationship Diagram) for the database

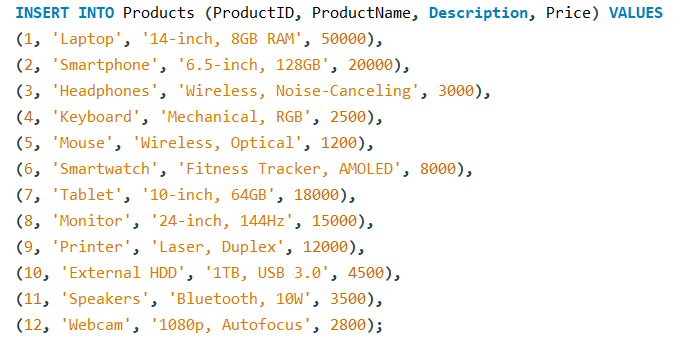


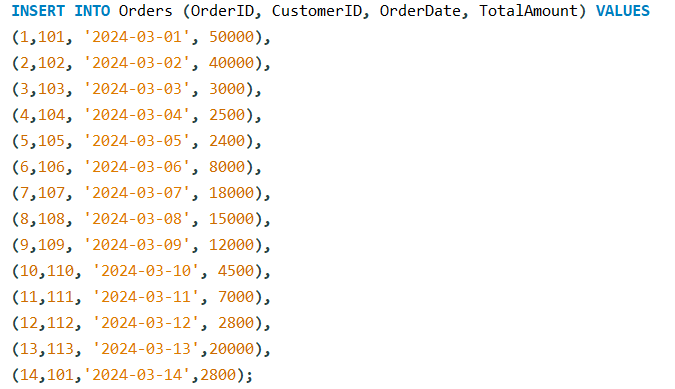
**4**. Create appropriate Primary Key and Foreign Key constraints for referential integrity.

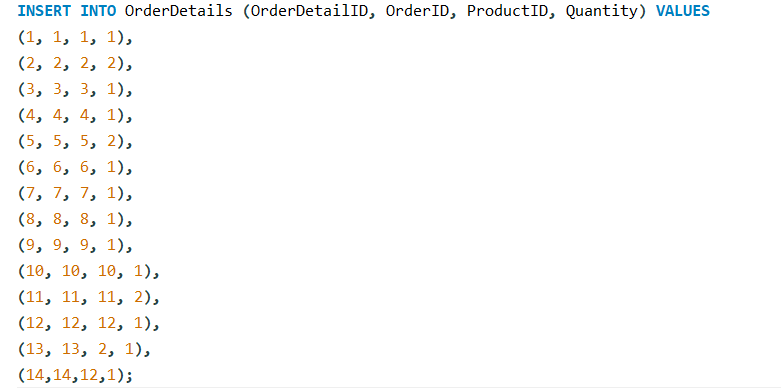
Successfully created the primary key and foreign key constraints.

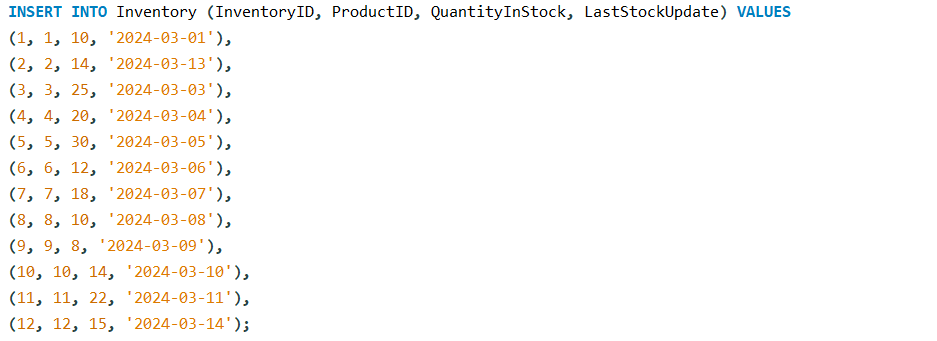
**5**. Insert at least 10 sample records into each of the following tables





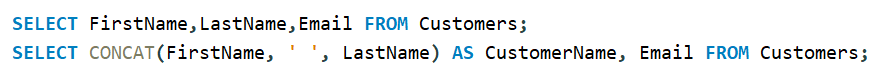




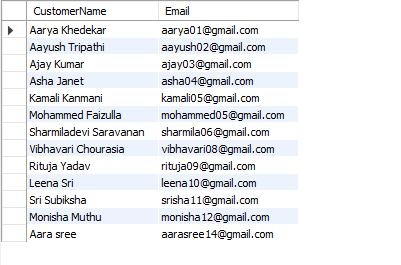


**Tasks 2: Select, Where, Between, AND, LIKE**

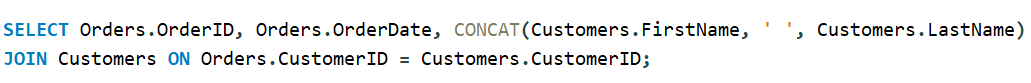
**1**. Write an SQL query to retrieve the names and emails of all customers

****

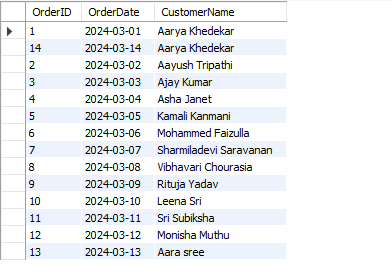
Output:



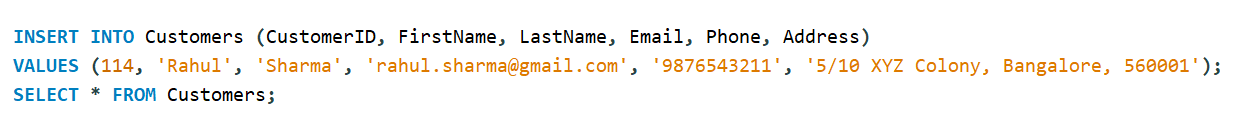
**2**. Write a SQL query to list all orders with their order dates and corresponding customer names.



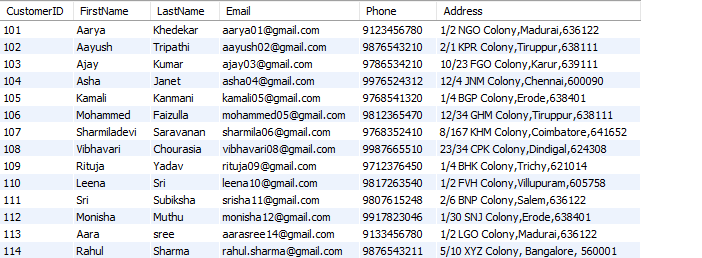
Output:



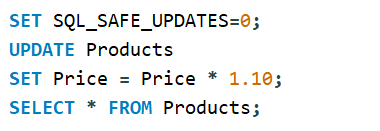
3. Write an SQL query to insert a new customer record into the "Customers" table. Include customer information such as name, email, and address.

****

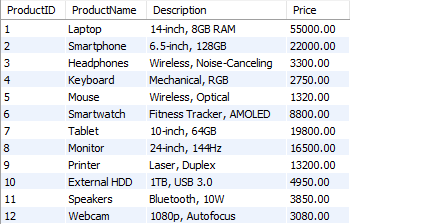
Output:



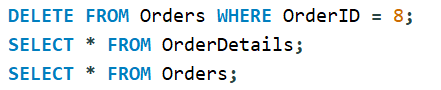
**4**.Write an SQL query to update the prices of all electronic gadgets in the "Products" table by increasing them by 10%



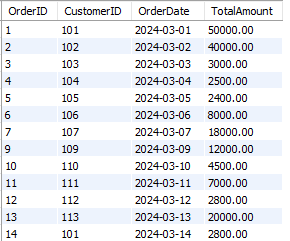
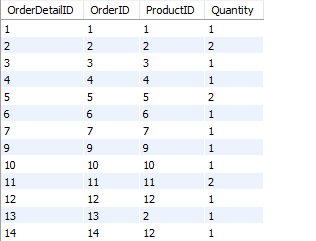
Output:



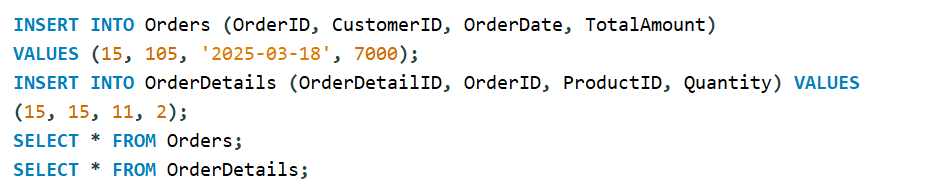
**5**. Write an SQL query to delete a specific order and its associated order details from the "Orders" and "OrderDetails" tables. Allow users to input the order ID as a parameter.



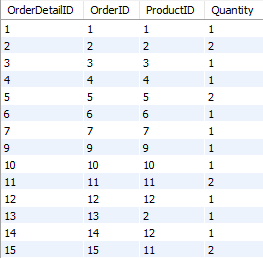
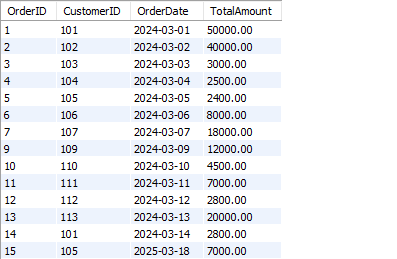
Output:



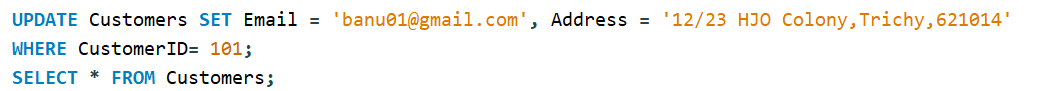
**6**. Write an SQL query to insert a new order into the "Orders" table. Include the customer ID, order date, and any other necessary information.



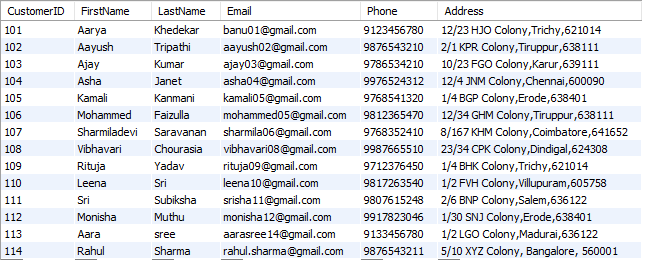
Output:



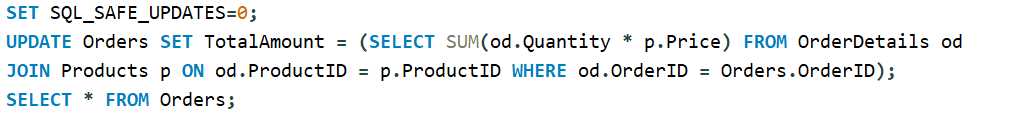
**7**.Write an SQL query to update the contact information (e.g., email and address) of a specific customer in the "Customers" table. Allow users to input the customer ID and new contact information



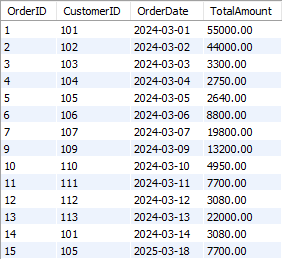
Output:



**8**. Write an SQL query to recalculate and update the total cost of each order in the "Orders" table based on the prices and quantities in the "OrderDetails" table.



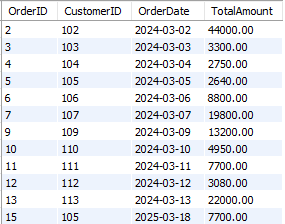
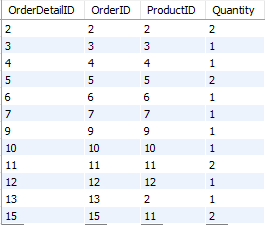
Output:



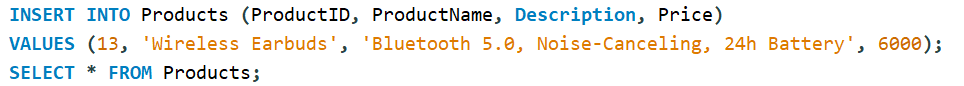
**9**. Write an SQL query to delete all orders and their associated order details for a specific customer from the "Orders" and "OrderDetails" tables. Allow users to input the customer ID as a parameter



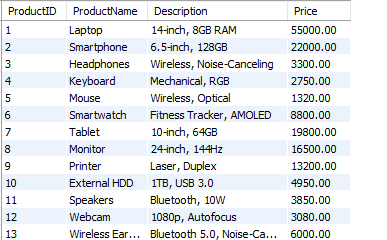
Output:

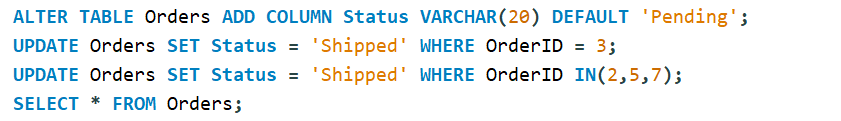
**10**. Write an SQL query to insert a new electronic gadget product into the "Products" table, including product name, category, price, and any other relevant details



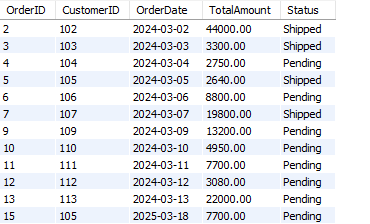
Output:



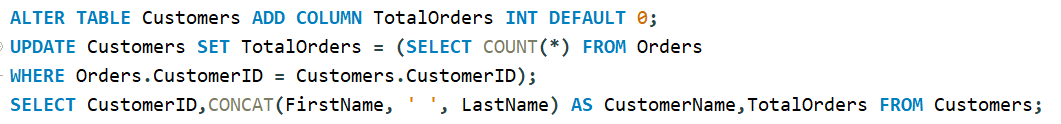
**11**. Write an SQL query to update the status of a specific order in the "Orders" table (e.g., from "Pending" to "Shipped"). Allow users to input the order ID and the new status.



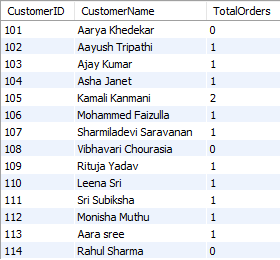
Output:



**12**. Write an SQL query to calculate and update the number of orders placed by each customer in the "Customers" table based on the data in the "Orders" table.

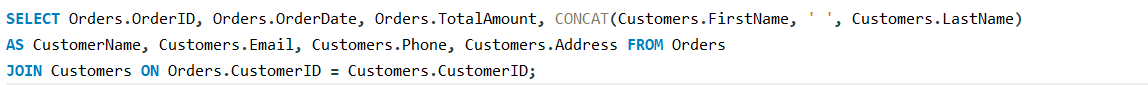


Output:

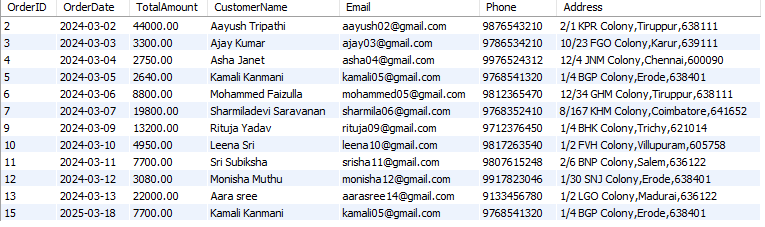


**Task 3. Aggregate functions, Having, Order By, Group By and Joins**

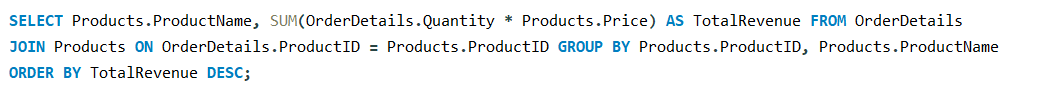
**1.** Write an SQL query to retrieve a list of all orders along with customer information (e.g., customer name) for each order.



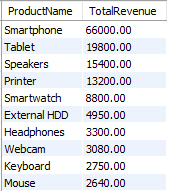
Output:



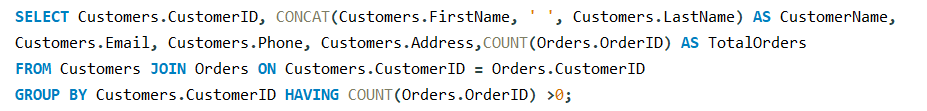
**2**. Write an SQL query to find the total revenue generated by each electronic gadget product. Include the product name and the total revenue.



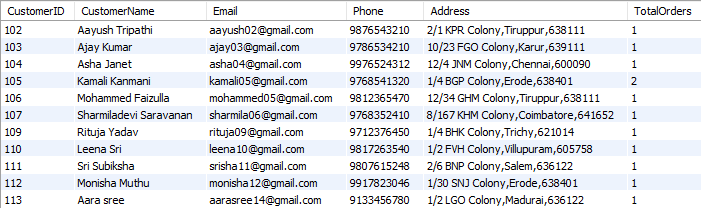
Output:



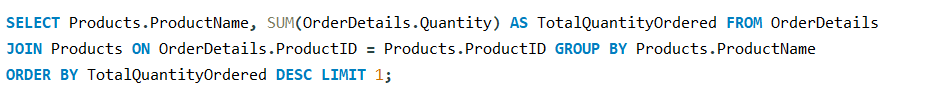
**3**. Write an SQL query to list all customers who have made at least one purchase. Include their names and contact information.



Output:



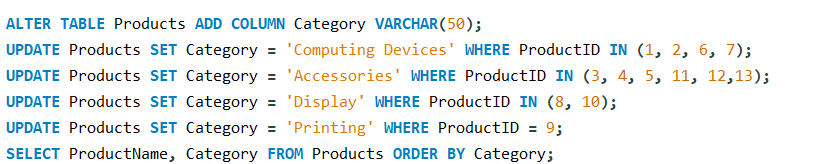
**4**.Write an SQL query to find the most popular electronic gadget, which is the one with the highest total quantity ordered. Include the product name and the total quantity ordered



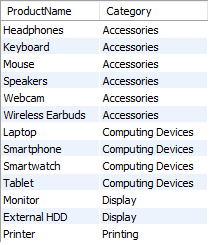
Output:



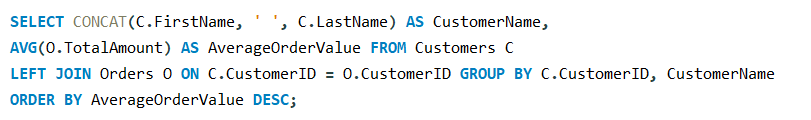
**5**. Write an SQL query to retrieve a list of electronic gadgets along with their corresponding categories.



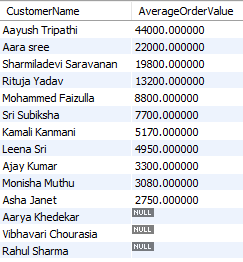
Output:

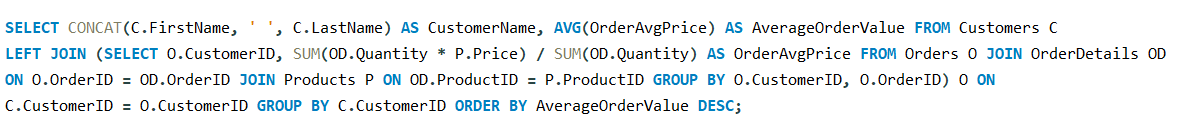


**6**. Write an SQL query to calculate the average order value for each customer. Include the customer's name and their average order value.

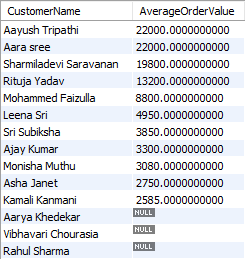


Output:

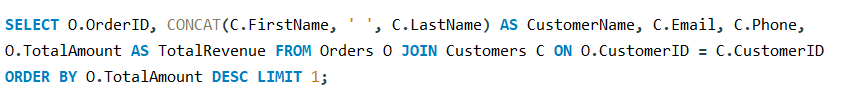




Output:



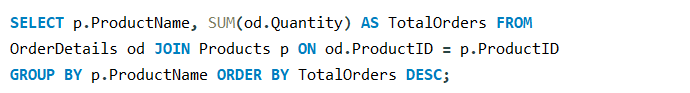
**7**. Write an SQL query to find the order with the highest total revenue. Include the order ID, customer information, and the total revenue.



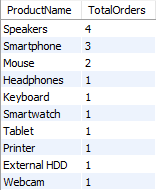
Output:



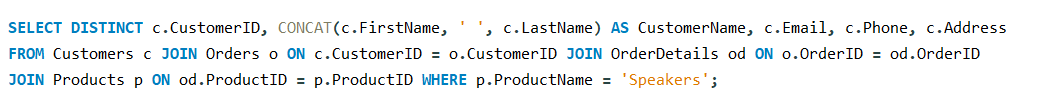
**8**. Write an SQL query to list electronic gadgets and the number of times each product has been ordered.



Output:



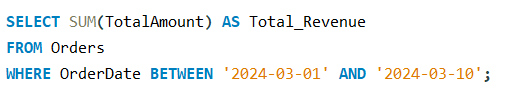
**9.** Write an SQL query to find customers who have purchased specific electronic gadget product. Allow users to input product name as parameter.



Output:



**10**. Write an SQL query to calculate the total revenue generated by all orders placed within a specific time period. Allow users to input the start and end dates as parameters.

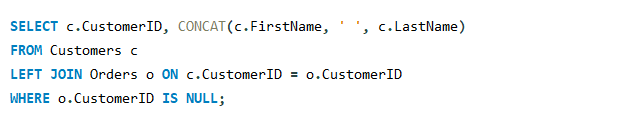


Output:

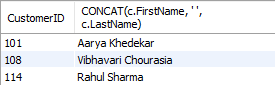


**Task 4: Subquery and its type**

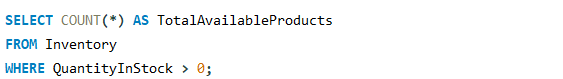
**1**. Write an SQL query to find out which customers have not placed any orders.



Output:



**2**. Write an SQL query to find the total number of products available for sale.

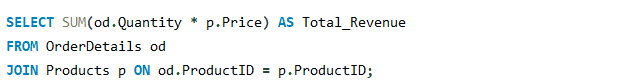


Output:



**3**. Write an SQL query to calculate the total revenue generated by TechShop.

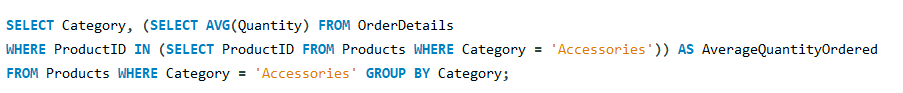




Output:



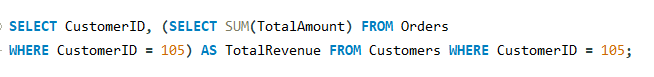
**4**. Write an SQL query to calculate the average quantity ordered for products in a specific category. Allow users to input the category name as a parameter.



Output:



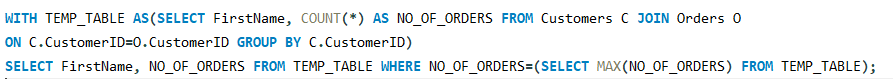
**5**. Write an SQL query to calculate the total revenue generated by a specific customer. Allow users to input the customer ID as a parameter.



Output:



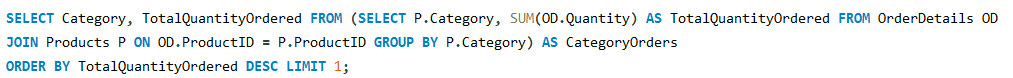
**6**. Write an SQL query to find the customers who have placed the most orders. List their names and the number of orders they've placed.



Output:



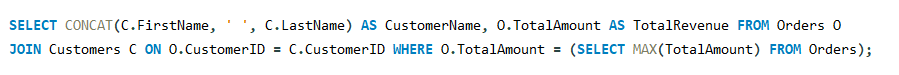
**7**. Write an SQL query to find the most popular product category, which is the one with the highest total quantity ordered across all orders.



Output:



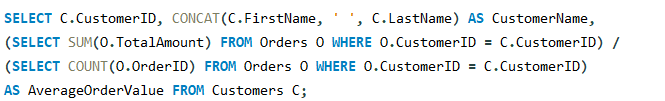
8. Write an SQL query to find the customer who has spent the most money (highest total revenue) on electronic gadgets. List their name and total spending.



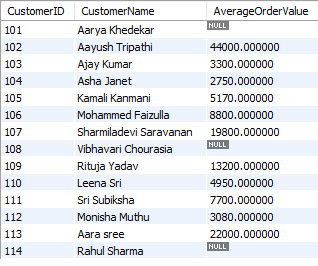
Output:



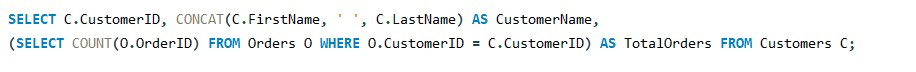
**9**. Write an SQL query to calculate the average order value (total revenue divided by the number of orders) for all customers.



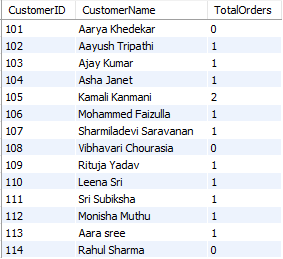
Output:



**10.** Write an SQL query to find the total number of orders placed by each customer and list their names along with the order count.



Output:



**OOPS IMPLEMENTATION**

**Package:entity**

Customers.java

|  |
| --- |
| package entity;  import exception.InvalidDataException;  public class Customers {  private int CustomerID;  private String FirstName;  private String LastName;  private String Email;  private String Phone;  private String Address;  public Customers(int customerID, String firstName, String lastName, String email, String phone, String address) {  setCustomerID(customerID);  setFirstName(firstName);  setLastName(lastName);  try {  setEmail(email);  setPhone(phone);  } catch (InvalidDataException e) {  System.***out***.println("Error: " + e.getMessage());  }  setAddress(address);  }  public Customers() {  // **TODO** Auto-generated constructor stub  }  public int getCustomerID() {  return CustomerID;  }  public void setCustomerID(int customerID) {  if (customerID > 0) this.CustomerID = customerID;  else throw new IllegalArgumentException("Customer ID must be positive.");  }  public String getFirstName() {  return FirstName;  }  public void setFirstName(String firstName) {  if (firstName != null && !firstName.trim().isEmpty()) this.FirstName = firstName;  else throw new IllegalArgumentException("First name cannot be null or empty.");  }  public String getLastName() {  return LastName;  }  public void setLastName(String lastName) {  if (lastName != null && !lastName.trim().isEmpty()) this.LastName = lastName;  else throw new IllegalArgumentException("Last name cannot be null or empty.");  }  public String getEmail() {  return Email;  }    public void setEmail(String email) throws InvalidDataException {  if (email != null && email.contains("@")) {  this.Email = email;  } else {  throw new InvalidDataException("Invalid email address. Please enter a valid email.");  }  }  public String getPhone() {  return Phone;  }  public void setPhone(String phone) throws InvalidDataException {  if (phone != null && phone.length() >= 10) {  this.Phone = phone;  } else {  throw new InvalidDataException("Phone number must be at least 10 digits.");  }  }  public String getAddress() {  return Address;  }  public void setAddress(String address) {  if (address != null && !address.trim().isEmpty()) this.Address = address;  else throw new IllegalArgumentException("Address cannot be null or empty.");  }  public void GetCustomerDetails() {  System.***out***.println("Customer ID : " + CustomerID);  System.***out***.println("Name : " + FirstName + " " + LastName);  System.***out***.println("Email : " + Email);  System.***out***.println("Phone : " + Phone);  System.***out***.println("Address : " + Address);  }  public void UpdateCustomerInfo(String email, String phone, String address) {  try {  setEmail(email);  setPhone(phone);  } catch (InvalidDataException e) {  System.***out***.println("Error: " + e.getMessage());  }  setAddress(address);  }  public int CalculateTotalOrders() {  return 0;  }      } |

Products.java

|  |
| --- |
| package entity;  public class Products {  private int ProductID;  private String ProductName;  private String Description;  private double Price;    public Products(int productID, String productName, String description, double price) {  setProductID(productID);  setProductName(productName);  setDescription(description);  setPrice(price);  }  public int getProductID() {  return ProductID;  }  public void setProductID(int productID) {  if (productID > 0) this.ProductID = productID;  else throw new IllegalArgumentException("Product ID must be positive.");  }  public String getProductName() {  return ProductName;  }  public void setProductName(String productName) {  if (productName != null && !productName.trim().isEmpty()) this.ProductName = productName;  else throw new IllegalArgumentException("Product name cannot be null or empty.");  }  public String getDescription() {  return Description;  }  public void setDescription(String description) {  if (description != null) this.Description = description;  else throw new IllegalArgumentException("Description cannot be null.");  }  public double getPrice() {  return Price;  }  public void setPrice(double price) {  if (price >= 0) this.Price = price;  else throw new IllegalArgumentException("Price cannot be negative.");  }  public void updateProductInfo(String description, double price) {  setDescription(description);  setPrice(price);  }  public void getProductDetails() {  System.***out***.println("Product ID : " + ProductID);  System.***out***.println("Name : " + ProductName);  System.***out***.println("Description : " + Description);  System.***out***.println("Price : $" + Price);  }  } |

Orders.java

|  |
| --- |
| package entity;  import java.time.LocalDate;  import java.time.LocalDateTime;  import java.util.List;  public class Orders {  private int orderID;  private Customers customer;  private LocalDateTime orderDate;  private double totalAmount;  private String status;  public Orders(int orderID, Customers customer, LocalDateTime orderDate, double totalAmount) {  setOrderID(orderID);  setCustomer(customer);  setOrderDate(orderDate);  setTotalAmount(totalAmount);  setStatus("Processing");  }  private int customerID;  private List<OrderDetails> orderDetails;  public Orders(int customerID, List<OrderDetails> orderDetails) {  this.setCustomerID(customerID);  this.orderDetails = orderDetails;  }  public void setOrderDetails(List<OrderDetails> orderDetails) {  this.orderDetails = orderDetails;  }  public Orders(int customerID2, Object customer2, LocalDateTime orderDate2, double totalAmount2) {  // TODO Auto-generated constructor stub  }  public Orders(Customers customer2, LocalDateTime now, double total, List<OrderDetails> orderDetailsList) {  // TODO Auto-generated constructor stub  }  public Orders(Customers customer2, LocalDate now, int count, List<OrderDetails> orderDetailsList) {  // TODO Auto-generated constructor stub  }  public Orders(Customers customer, LocalDateTime orderDate, double totalAmount, String status) {  this.customer = customer;  this.orderDate = orderDate;  this.totalAmount = totalAmount;  this.status = status;  }  public double calculateTotal() {  double total = 0.0;  if (orderDetails != null) {  for (OrderDetails od : orderDetails) {  Products product = od.getProduct();  if (product != null) {  total += product.getPrice() \* od.getQuantity();  }  }  }  return total;  }  public int getOrderID() {  return orderID;  }  public void setOrderID(int orderID) {  if (orderID > 0) this.orderID = orderID;  else throw new IllegalArgumentException("Order ID must be positive.");  }  public Customers getCustomer() {  return customer;  }  public void setCustomer(Customers customer) {  if (customer != null) this.customer = customer;  else throw new IllegalArgumentException("Customer cannot be null.");  }  public LocalDateTime getOrderDate() {  return orderDate;  }  public void setOrderDate(LocalDateTime orderDate2) {  if (orderDate2 != null) this.orderDate = orderDate2;  else throw new IllegalArgumentException("Order date cannot be null.");  }  public double getTotalAmount() {  return totalAmount;  }  public void setTotalAmount(double totalAmount) {  if (totalAmount >= 0) this.totalAmount = totalAmount;  else throw new IllegalArgumentException("Total amount cannot be negative.");  }  public String getStatus() {  return status;  }  public void setStatus(String status) {  if (status != null && !status.trim().isEmpty()) this.status = status;  else throw new IllegalArgumentException("Status cannot be null or empty.");  }  public List<OrderDetails> getOrderDetails() {  return orderDetails;  }  public int getCustomerID() {    return customerID;  }  public void setCustomerID(int customerID) {  this.customerID = customerID;  }  public Orders(int orderId, LocalDateTime orderDate, double totalAmount, String status) {  this.orderID = orderId;  this.orderDate = orderDate;  this.totalAmount = totalAmount;  this.status = status;  }  } |

OrderDetails.java

|  |
| --- |
| package entity;  import exception.\*;  public class OrderDetails {  private int OrderDetailID;  private Orders Order;  private Products Product;  private int Quantity;  private double Discount;    public OrderDetails(int orderDetailID, Orders order, Products product, int quantity) {  setOrderDetailID(orderDetailID);  setOrder(order);  setProduct(product);  setQuantity(quantity);  this.Discount = 0.0;  }  public OrderDetails(int orderDetailID, Products product, int quantity, double price) {  this.OrderDetailID = orderDetailID;  this.Product = product;  this.Quantity = quantity;  this.Discount = price;  }  public OrderDetails(Products product, int quantity) {  this.Product = product;  this.Quantity = quantity;  }  public OrderDetails(int orderDetailID2, int productID, int quantity2, int quantity3) {  // **TODO** Auto-generated constructor stub  }  public int getOrderDetailID() {  return OrderDetailID;  }  public void setOrderDetailID(int orderDetailID) {  if (orderDetailID > 0) this.OrderDetailID = orderDetailID;  else throw new IllegalArgumentException("Order Detail ID must be positive.");  }  public Orders getOrder() {  return Order;  }  public void setOrder(Orders order) {  if (order != null) this.Order = order;  else throw new IllegalArgumentException("Order cannot be null.");  }  public Products getProduct() {  return Product;  }  public void setProduct(Products product) {  if (product != null) {  this.Product = product;  } else {  throw new IncompleteOrderDetailException("Order detail must include a valid product.");  }  }  public int getQuantity() {  return Quantity;  }  public void setQuantity(int quantity) {  if (quantity > 0) this.Quantity = quantity;  else throw new IllegalArgumentException("Quantity must be positive.");  }  public double getDiscount() {  return Discount;  }  public void AddDiscount(double discount) {  if (discount >= 0 && discount <= 100) this.Discount = discount;  else throw new IllegalArgumentException("Discount must be between 0 and 100.");  }  public double CalculateSubtotal() {  double subtotal = Quantity \* Product.getPrice();  subtotal -= subtotal \* (Discount / 100.0);  return subtotal;  }  public void GetOrderDetailInfo() {  System.***out***.println("Order Detail ID : " + OrderDetailID);  System.***out***.println("Product : " + Product.getProductName());  System.***out***.println("Quantity : " + Quantity);  System.***out***.println("Price Each : $" + Product.getPrice());  System.***out***.println("Discount : " + Discount + "%");  System.***out***.println("Subtotal : $" + CalculateSubtotal());  }  *@Override*  public String toString() {  return "OrderDetailID: " + OrderDetailID + ", Product: " + Product.getProductName() + ", Quantity: " + Quantity;  }  public void UpdateQuantity(int quantity) {  setQuantity(quantity);  }  public int getProductId() {  return Product != null ? Product.getProductID() : -1;  }  } |

Inventory.java

|  |
| --- |
| package entity;  import java.time.LocalDateTime;  public class Inventory {  private int InventoryID;  private Products Product;  private int QuantityInStock;  private LocalDateTime LastStockUpdate;  public Inventory(int inventoryID, Products product, int quantityInStock) {  setInventoryID(inventoryID);  setProduct(product);  setQuantityInStock(quantityInStock);  this.LastStockUpdate = LocalDateTime.*now*();  }  public int getInventoryID() { return InventoryID; }  public void setInventoryID(int inventoryID) {  if (inventoryID > 0) this.InventoryID = inventoryID;  else throw new IllegalArgumentException("Inventory ID must be positive.");  }  public Products getProduct() { return Product; }  public void setProduct(Products product) {  if (product != null) this.Product = product;  else throw new IllegalArgumentException("Product cannot be null.");  }  public int getQuantityInStock() { return QuantityInStock; }  public void setQuantityInStock(int quantityInStock) {  if (quantityInStock >= 0) {  this.QuantityInStock = quantityInStock;  this.LastStockUpdate = LocalDateTime.*now*();  } else {  throw new IllegalArgumentException("Quantity must be non-negative.");  }  }  public LocalDateTime getLastStockUpdate() { return LastStockUpdate; }  public double getInventoryValue() { return Product.getPrice() \* QuantityInStock; }  public boolean isProductAvailable(int quantity) { return quantity <= QuantityInStock; }  } |

**Package:dao**

CustomerDAO.java

|  |
| --- |
| package dao;  import java.io.IOException;  import entity.Customers;  import exception.\*;  public interface CustomerDAO {    void registerCustomer(Customers customer) throws DuplicateEntryException, IOException;  boolean isEmailExists(String email);  Customers getCustomerById(int customerId) throws IOException;  void updateCustomerAccount(Customers customer) throws IOException;  } |

CustomerDAOImpl.java

|  |
| --- |
| package dao;  import entity.Customers;  import exception.DuplicateEntryException;  import util.DBConnUtil;  import java.io.IOException;  import java.sql.Connection;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  public class CustomerDAOImpl implements CustomerDAO {  @Override  public void registerCustomer(Customers customer) throws DuplicateEntryException, IOException {  if (isEmailExists(customer.getEmail())) {  throw new DuplicateEntryException("Email address already registered.");  }  Connection connection = null;  PreparedStatement stmt = null;  try {    connection = DBConnUtil.getDbConnection();  String insertQuery = "INSERT INTO Customers (CustomerID, FirstName, LastName, Email, Phone, Address) VALUES (?, ?, ?, ?, ?, ?)";  stmt = connection.prepareStatement(insertQuery);  stmt.setInt(1, customer.getCustomerID());  stmt.setString(2, customer.getFirstName());  stmt.setString(3, customer.getLastName());  stmt.setString(4, customer.getEmail());  stmt.setString(5, customer.getPhone());  stmt.setString(6, customer.getAddress());  int rowsAffected = stmt.executeUpdate();  if (rowsAffected > 0) {  System.out.println("Customer registered successfully.");  } else {  System.out.println("Customer registration failed.");  }  } catch (SQLException e) {  System.out.println("Error while registering customer: " + e.getMessage());  } finally {  try {  if (stmt != null) stmt.close();  if (connection != null) connection.close();  } catch (SQLException e) {  System.out.println("Error while closing resources: " + e.getMessage());  }  }  }    @Override  public boolean isEmailExists(String email) {    return false;  }  @Override  public Customers getCustomerById(int customerID) throws IOException {  Customers customer = null;  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  try {  conn = DBConnUtil.getDbConnection();  String query = "SELECT \* FROM Customers WHERE CustomerID = ?";  stmt = conn.prepareStatement(query);  stmt.setInt(1, customerID);  rs = stmt.executeQuery();  if (rs.next()) {  String firstName = rs.getString("FirstName");  String lastName = rs.getString("LastName");  String email = rs.getString("Email");  String phone = rs.getString("Phone");  String address=rs.getString("Address");  customer = new Customers(customerID, firstName, lastName, email, phone,address);  }  } catch (SQLException e) {  System.out.println("Error fetching customer: " + e.getMessage());  } finally {  try {  if (rs != null) rs.close();  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.out.println("Error closing resources: " + e.getMessage());  }  }  return customer;  }  @Override  public void updateCustomerAccount(Customers customer) throws IOException {  Connection conn = null;  PreparedStatement stmt = null;  try {  conn = DBConnUtil.getDbConnection();  String query = "UPDATE Customers SET Email = ?, Phone = ? WHERE CustomerID = ?";  stmt = conn.prepareStatement(query);  stmt.setString(1, customer.getEmail());  stmt.setString(2, customer.getPhone());  stmt.setInt(3, customer.getCustomerID());  int rowsUpdated = stmt.executeUpdate();  if (rowsUpdated > 0) {  System.out.println("Customer account updated successfully.");  } else {  System.out.println("No customer found with the given ID.");  }  } catch (SQLException e) {  System.out.println("Error updating customer: " + e.getMessage());  } finally {  try {  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.out.println("Error closing resources: " + e.getMessage());  }  }  }    } |

InventoryDAO.java

|  |
| --- |
| package dao;  import entity.Inventory;  import java.io.IOException;  import java.sql.SQLException;  import java.util.List;  public interface InventoryDAO {  void addInventory(Inventory inventory);  void removeFromInventory(int inventoryID, int quantity) throws IOException;  void addToInventory(int inventoryID, int quantity) throws IOException;  void updateStock(int inventoryID, int newQuantity) throws IOException;  void listLowStockProducts(int threshold);  void listOutOfStockProducts();  void listAllProducts();  List<Inventory> getInventoryList() throws SQLException, IOException;  static boolean isProductAvailable(int productID, int requiredQuantity) {  // TODO Auto-generated method stub  return false;  }  boolean addProductToInventory(int inventoryID,int productID, int quantity) throws IOException;  int getProductStock(int productID) throws SQLException, IOException;  boolean removeProductFromInventory(int inventoryID, int productID, int quantity) throws IOException;  Inventory findInventoryById(int inventoryID) throws IOException;    } |

InventoryDAOImpl.java

|  |
| --- |
| package dao;  import entity.\*;  import util.DBConnUtil;  import java.io.IOException;  import java.sql.Connection;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.util.\*;  public class InventoryDAOImpl implements InventoryDAO {  private static List<Inventory> *inventoryList* = new ArrayList<>();  public void addInventory(Inventory inventory) {  *inventoryList*.add(inventory);  }    *@Override*  public void removeFromInventory(int productId, int quantity) throws IOException {  String selectSql = "SELECT QuantityInStock FROM inventory WHERE ProductID = ?";  String updateSql = "UPDATE inventory SET QuantityInStock = ? WHERE ProductID = ?";  String deleteSql = "DELETE FROM inventory WHERE ProductID= ?";  try (Connection conn = DBConnUtil.*getDbConnection*();  PreparedStatement selectStmt = conn.prepareStatement(selectSql)) {  selectStmt.setInt(1, productId);  ResultSet rs = selectStmt.executeQuery();  if (rs.next()) {  int currentStock = rs.getInt("QuantityInStock");  System.***out***.println("🔍 Current stock: " + currentStock);  int newStock = currentStock - quantity;  if (newStock <= 0) {  try (PreparedStatement deleteStmt = conn.prepareStatement(deleteSql)) {  deleteStmt.setInt(1, productId);  int rows = deleteStmt.executeUpdate();  if (rows > 0) {  System.***out***.println("✅ Inventory deleted for Product ID: " + productId);  } else {  System.***out***.println("⚠️ Failed to delete inventory (Product ID: " + productId + ")");  }  }  } else {  try (PreparedStatement updateStmt = conn.prepareStatement(updateSql)) {  updateStmt.setInt(1, newStock);  updateStmt.setInt(2, productId);  int rows = updateStmt.executeUpdate();  if (rows > 0) {  System.***out***.println("✅ Inventory updated for Product ID: " + productId + ", new quantity: " + newStock);  } else {  System.***out***.println("⚠️ Failed to update inventory (Product ID: " + productId + ")");  }  }  }  } else {  System.***out***.println("⚠️ No inventory found for Product ID: " + productId);  }  } catch (SQLException e) {  System.***out***.println("❌ Error in removeFromInventory: " + e.getMessage());  }  }  public void addToInventory(int inventoryID, int quantity) throws IOException {  if (quantity <= 0) throw new IllegalArgumentException("Quantity must be positive.");  Inventory inv = findInventoryById(inventoryID);  inv.setQuantityInStock(inv.getQuantityInStock() + quantity);  }  public void updateStock(int inventoryID, int newQuantity) throws IOException {  Inventory inv = findInventoryById(inventoryID);  inv.setQuantityInStock(newQuantity);  }  public void listLowStockProducts(int threshold) {  System.***out***.println("Low Stock Products:");  for (Inventory inv : *inventoryList*) {  if (inv.getQuantityInStock() < threshold) {  System.***out***.println(inv.getProduct().getProductName() + " - Qty: " + inv.getQuantityInStock());  }  }  }  public void listOutOfStockProducts() {  System.***out***.println("Out of Stock Products:");  for (Inventory inv : *inventoryList*) {  if (inv.getQuantityInStock() == 0) {  System.***out***.println(inv.getProduct().getProductName());  }  }  }  public void listAllProducts() {  System.***out***.println("All Inventory Products:");  for (Inventory inv : *inventoryList*) {  System.***out***.println("Product: " + inv.getProduct().getProductName() +  " | Quantity: " + inv.getQuantityInStock());  }  }  *@Override*  public List<Inventory> getInventoryList() throws SQLException, IOException {  List<Inventory> inventoryList = new ArrayList<>();  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  try {  conn = DBConnUtil.*getDbConnection*();  String query = "SELECT \* FROM Inventory";  stmt = conn.prepareStatement(query);  rs = stmt.executeQuery();  while (rs.next()) {  int inventoryID = rs.getInt("InventoryID");  int productID = rs.getInt("ProductID");  int quantityInStock = rs.getInt("QuantityInStock");    System.***out***.println("Retrieved QuantityInStock for ProductID " + productID + ": " + quantityInStock);    if (quantityInStock < 0) {  System.***out***.println("Invalid QuantityInStock for ProductID " + productID + ". Setting it to 0.");  quantityInStock = 0;  }  ProductDAO productDAO = new ProductDAOImpl();  Products product = productDAO.getProductById(productID);  Inventory inventory = new Inventory(inventoryID, product, quantityInStock);  inventoryList.add(inventory);  }  } catch (SQLException e) {  System.***out***.println("Error retrieving inventory list: " + e.getMessage());  } finally {  try {  if (rs != null) rs.close();  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.***out***.println("Error closing resources: " + e.getMessage());  }  }  return inventoryList;  }    public static boolean isProductAvailable(int productID, int requiredQuantity) throws IOException {  Inventory inventory = null;  try {  InventoryDAO inventoryDAO = new InventoryDAOImpl();  inventory = inventoryDAO.findInventoryById(productID);  } catch (RuntimeException e) {  return false;  }  return inventory != null && inventory.getQuantityInStock() >= requiredQuantity;  }  public Inventory findInventoryById(int inventoryID)throws IOException {  for (Inventory inv : *inventoryList*) {  if (inv.getInventoryID() == inventoryID) return inv;  }  throw new RuntimeException("Inventory not found.");  }    *@Override*  public boolean addProductToInventory(int inventoryID, int productID, int quantity) throws IOException {  Connection conn = null;  PreparedStatement stmt = null;  try {  conn = DBConnUtil.*getDbConnection*();    String updateQuery = "UPDATE Inventory SET QuantityInStock = QuantityInStock + ? WHERE InventoryID = ?";  stmt = conn.prepareStatement(updateQuery);  stmt.setInt(1, quantity);  stmt.setInt(2, inventoryID);  int rowsAffected = stmt.executeUpdate();  if (rowsAffected == 0) {    stmt.close();  String insertQuery = "INSERT INTO Inventory (InventoryID, ProductID, QuantityInStock) VALUES (?, ?, ?)";  stmt = conn.prepareStatement(insertQuery);  stmt.setInt(1, inventoryID);  stmt.setInt(2, productID);  stmt.setInt(3, quantity);  int insertResult = stmt.executeUpdate();  if (insertResult > 0) {  System.***out***.println("✅ Inventory inserted successfully.");  return true;  } else {  System.***out***.println("❌ Failed to insert inventory.");  return false;  }  } else {  System.***out***.println("✅ Inventory updated successfully.");  return true;  }  } catch (SQLException e) {  System.***out***.println("❌ Error while updating/inserting inventory: " + e.getMessage());  return false;  } finally {  try {  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.***out***.println("⚠️ Error closing resources: " + e.getMessage());  }  }  }  public int getProductStock(int productID) throws SQLException, IOException {  int availableStock = 0;  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  try {  conn = DBConnUtil.*getDbConnection*();  String query = "SELECT QuantityInStock FROM Inventory WHERE ProductID = ?";  stmt = conn.prepareStatement(query);  stmt.setInt(1, productID);  rs = stmt.executeQuery();  if (rs.next()) {  availableStock = rs.getInt("QuantityInStock");  }  } catch (SQLException |IOException e) {  System.***out***.println("Error fetching product stock: " + e.getMessage());  throw e;  } finally {  try {  if (rs != null) rs.close();  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.***out***.println("Error closing resources: " + e.getMessage());  }  }  return availableStock;  }  *@Override*  public boolean removeProductFromInventory(int inventoryID, int productID, int quantity) throws IOException {  Connection conn = null;  PreparedStatement stmt = null;  try {  conn = DBConnUtil.*getDbConnection*();  String checkQuery = "SELECT QuantityInStock FROM Inventory WHERE InventoryID = ? AND ProductID = ?";  stmt = conn.prepareStatement(checkQuery);  stmt.setInt(1, inventoryID);  stmt.setInt(2, productID);  ResultSet rs = stmt.executeQuery();  if (rs.next()) {  int currentStock = rs.getInt("QuantityInStock");  if (currentStock >= quantity) {  stmt.close();  String updateQuery = "UPDATE Inventory SET QuantityInStock = QuantityInStock - ? WHERE InventoryID = ? AND ProductID = ?";  stmt = conn.prepareStatement(updateQuery);  stmt.setInt(1, quantity);  stmt.setInt(2, inventoryID);  stmt.setInt(3, productID);  int updated = stmt.executeUpdate();  if (updated > 0) {  System.***out***.println("✅ Inventory updated: Product removed.");  return true;  }  } else {  System.***out***.println("❌ Not enough stock to remove.");  }  } else {  System.***out***.println("❌ Inventory record not found.");  }  } catch (SQLException e) {  System.***out***.println("❌ Error during inventory removal: " + e.getMessage());  } finally {  try {  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.***out***.println("⚠️ Error closing resources: " + e.getMessage());  }  }  return false;  }  } |

OrderDAO.java

|  |
| --- |
| package dao;  import entity.OrderDetails;  import entity.Orders;  import exception.AuthenticationException;  import exception.AuthorizationException;  import java.io.IOException;  import java.sql.SQLException;  import java.time.LocalDateTime;  import java.util.List;  import java.util.SortedMap;  public interface OrderDAO {  void addOrder(Orders order);  void updateOrderStatus(int orderID, String newStatus);  void removeCanceledOrder(int orderID);  List<Orders> getOrdersList();  List<Orders> sortOrdersByDate(boolean ascending);  List<Orders> getOrdersByDateRange(LocalDateTime start, LocalDateTime end);  void processPayment(Orders order, double amount);  void cancelOrder(Orders order, String username, String password) throws AuthorizationException, AuthenticationException;  SortedMap<Integer, Integer> getFullInventory();  void addToInventory(int productID, int quantity);  Orders getOrderById(int orderID) throws IOException, SQLException;  void placeOrder(Orders order, List<OrderDetails> orderDetailsList) throws SQLException, IOException;  void updateOrderStatus(Orders order) throws SQLException, IOException;  } |

OrderDAOImpl.java

|  |
| --- |
| package dao;  import entity.\*;  import exception.\*;  import util.DBConnUtil;  import java.io.IOException;  import java.sql.Connection;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  import java.time.LocalDateTime;  import java.util.\*;  import java.sql.Timestamp;  public class OrderDAOImpl implements OrderDAO {  private static List<Orders> ordersList = new ArrayList<>();  private static SortedMap<Integer, Integer> inventory = new TreeMap<>();    @Override  public void addOrder(Orders order) {  ordersList.add(order);  }  @Override  public void updateOrderStatus(int orderID, String newStatus) {  for (Orders o : ordersList) {  if (o.getOrderID() == orderID) {  o.setStatus(newStatus);  return;  }  }  throw new RuntimeException("Order not found to update status.");  }  @Override  public void removeCanceledOrder(int orderID) {  Iterator<Orders> iterator = ordersList.iterator();  while (iterator.hasNext()) {  Orders o = iterator.next();  if ("Cancelled".equalsIgnoreCase(o.getStatus()) && o.getOrderID() == orderID) {  iterator.remove();  return;  }  }  throw new RuntimeException("Canceled order not found or already removed.");  }  @Override  public List<Orders> getOrdersList() {  return ordersList;  }  @Override  public List<Orders> sortOrdersByDate(boolean ascending) {  List<Orders> sorted = new ArrayList<>(ordersList);  sorted.sort((o1, o2) -> ascending ? o1.getOrderDate().compareTo(o2.getOrderDate())  : o2.getOrderDate().compareTo(o1.getOrderDate()));  return sorted;  }  @Override  public List<Orders> getOrdersByDateRange(LocalDateTime startDate, LocalDateTime endDate) {  List<Orders> filtered = new ArrayList<>();  for (Orders order : ordersList) {  if (!order.getOrderDate().isBefore(startDate) && !order.getOrderDate().isAfter(endDate)) {  filtered.add(order);  }  }  return filtered;  }  @Override  public void processPayment(Orders order, double amount) {  if (amount >= order.getTotalAmount()) {  System.out.println("Payment of $" + amount + " processed successfully.");  order.setStatus("Paid");  } else {  throw new PaymentFailedException("Insufficient payment. Please try again.");  }  }  @Override  public void cancelOrder(Orders order, String username, String password)  throws AuthorizationException, AuthenticationException {  if (authenticate(username, password)) {  if (authorize("ORDER\_CANCEL")) {  order.setStatus("Cancelled");  System.out.println("Order " + order.getOrderID() + " cancelled.");  } else {  throw new AuthorizationException("Access denied: Unauthorized to cancel orders.");  }  } else {  throw new AuthenticationException("Invalid credentials for cancelling order.");  }  }  private boolean authenticate(String username, String password) {  return "admin".equals(username) && "password123".equals(password);  }  private boolean authorize(String permission) {  return "ORDER\_CANCEL".equals(permission);  }  @Override  public SortedMap<Integer, Integer> getFullInventory() {  return inventory;  }  @Override  public void addToInventory(int productID, int quantity) {  inventory.put(productID, inventory.getOrDefault(productID, 0) + quantity);  }    public void placeOrder(Orders order, List<OrderDetails> orderDetailsList) throws SQLException, IOException {  Connection conn = null;  PreparedStatement orderStmt = null;  PreparedStatement detailStmt = null;  PreparedStatement inventoryStmt = null;  try {  conn = DBConnUtil.getDbConnection();  conn.setAutoCommit(false);    String insertOrderSQL = "INSERT INTO Orders (CustomerID, OrderDate, TotalAmount, Status) VALUES (?, ?, ?, ?)";  orderStmt = conn.prepareStatement(insertOrderSQL, Statement.RETURN\_GENERATED\_KEYS);  orderStmt.setInt(1, order.getCustomer().getCustomerID());  orderStmt.setTimestamp(2, Timestamp.valueOf(LocalDateTime.now()));  orderStmt.setDouble(3, 0);  orderStmt.setString(4, "Placed");  int orderRowsAffected = orderStmt.executeUpdate();  if (orderRowsAffected == 0) {  throw new RuntimeException("Failed to insert order. No rows affected.");  }  ResultSet rs = orderStmt.getGeneratedKeys();  if (!rs.next()) {  throw new RuntimeException("Order ID generation failed.");  }  int generatedOrderID = rs.getInt(1);  double total = 0;    String insertDetailsSQL = "INSERT INTO OrderDetails (OrderID, ProductID, Quantity, Subtotal) VALUES (?, ?, ?, ?)";  detailStmt = conn.prepareStatement(insertDetailsSQL);  String updateInventorySQL = "UPDATE Inventory SET QuantityInStock = QuantityInStock - ? WHERE ProductID = ?";  inventoryStmt = conn.prepareStatement(updateInventorySQL);  for (OrderDetails detail : orderDetailsList) {  double price = fetchProductPrice(detail.getProductId(), conn);  if (price < 0) {  throw new RuntimeException("Invalid price fetched for ProductID: " + detail.getProductId());  }  double subtotal = price \* detail.getQuantity();  total += subtotal;  detailStmt.setInt(1, generatedOrderID);  detailStmt.setInt(2, detail.getProductId());  detailStmt.setInt(3, detail.getQuantity());  detailStmt.setDouble(4, subtotal);  detailStmt.addBatch();  inventoryStmt.setInt(1, detail.getQuantity());  inventoryStmt.setInt(2, detail.getProductId());  inventoryStmt.addBatch();  }  detailStmt.executeBatch();  inventoryStmt.executeBatch();    String updateTotalSQL = "UPDATE Orders SET TotalAmount = ? WHERE OrderID = ?";  try (PreparedStatement updateTotalStmt = conn.prepareStatement(updateTotalSQL)) {  updateTotalStmt.setDouble(1, total);  updateTotalStmt.setInt(2, generatedOrderID);  updateTotalStmt.executeUpdate();  }    conn.commit();  System.out.println("✅ Order placed successfully. Order ID: " + generatedOrderID + " | Total: $" + total);  } catch (SQLException e) {  if (conn != null) {  try {  conn.rollback();  } catch (SQLException rollbackException) {  System.out.println("❌ Rollback failed: " + rollbackException.getMessage());  }  }  System.out.println("❌ Error placing order: " + e.getMessage());  throw new RuntimeException("Error placing order: " + e.getMessage(), e);  } finally {  try {  if (orderStmt != null) orderStmt.close();  if (detailStmt != null) detailStmt.close();  if (inventoryStmt != null) inventoryStmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.out.println("❌ Error closing resources: " + e.getMessage());  }  }  }  private double fetchProductPrice(int productID, Connection conn) throws SQLException {  String query = "SELECT Price FROM Products WHERE ProductID = ?";  try (PreparedStatement stmt = conn.prepareStatement(query)) {  stmt.setInt(1, productID);  ResultSet rs = stmt.executeQuery();  if (rs.next()) {  return rs.getDouble("Price");  } else {  throw new RuntimeException("Product with ID " + productID + " not found.");  }  }  }    @Override  public Orders getOrderById(int orderID) throws IOException {  Orders order = null;  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  try {  conn = DBConnUtil.getDbConnection();  String query = "SELECT \* FROM Orders WHERE OrderID = ?";  stmt = conn.prepareStatement(query);  stmt.setInt(1, orderID);  rs = stmt.executeQuery();  if (rs.next()) {  Timestamp orderTimestamp = rs.getTimestamp("OrderDate");  LocalDateTime orderDate = orderTimestamp.toLocalDateTime();  double totalAmount = rs.getDouble("TotalAmount");  String status = rs.getString("Status");  // Use a constructor that doesn't require Customer  order = new Orders(orderID, orderDate, totalAmount, status);  }  } catch (SQLException e) {  System.out.println("❌ Error retrieving order: " + e.getMessage());  } finally {  try {  if (rs != null) rs.close();  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  return order;  }    public void updateOrderStatus(Orders order) throws SQLException, IOException {  Connection conn = null;  PreparedStatement stmt = null;  try {  conn = DBConnUtil.getDbConnection();  String updateQuery = "UPDATE Orders SET Status = ? WHERE OrderID = ?";  stmt = conn.prepareStatement(updateQuery);  stmt.setString(1, order.getStatus());  stmt.setInt(2, order.getOrderID());  int rowsAffected = stmt.executeUpdate();  if (rowsAffected == 0) {  throw new RuntimeException("Order not found to update status.");  }  } catch (SQLException e) {  throw new SQLException("Error updating order status: " + e.getMessage());  } finally {  if (stmt != null) stmt.close();  if (conn != null) conn.close();  }  }  } |

OrderDetailsDAO.java

|  |
| --- |
| package dao;  import entity.OrderDetails;  import exception.InsufficientStockException;  import java.io.IOException;  import java.util.List;  public interface OrderDetailsDAO {  void addOrderDetail(OrderDetails orderDetail) throws InsufficientStockException, IOException;  void updateOrderDetailQuantity(int orderDetailID, int newQuantity) throws InsufficientStockException, IOException;  void removeOrderDetail(int orderDetailID);  List<OrderDetails> getAllOrderDetails();  OrderDetails getOrderDetailById(int orderDetailID); |

OrderDetailsDAOImpl.java

|  |
| --- |
| package dao;  import entity.\*;  import exception.InsufficientStockException;  import java.io.IOException;  import java.util.ArrayList;  import java.util.List;  public class OrderDetailsDAOImpl implements OrderDetailsDAO {  private List<OrderDetails> orderDetailsList = new ArrayList<>();  private InventoryDAOImpl inventoryDAO = new InventoryDAOImpl();  @Override  public void addOrderDetail(OrderDetails orderDetail) throws InsufficientStockException, IOException {  int productId = orderDetail.getProduct().getProductID();  int quantity = orderDetail.getQuantity();    if (!InventoryDAOImpl.isProductAvailable(productId, quantity)) {  throw new InsufficientStockException("Not enough stock for product ID: " + productId);  }  inventoryDAO.removeFromInventory(productId, quantity);  orderDetailsList.add(orderDetail);  }  @Override  public void updateOrderDetailQuantity(int orderDetailID, int newQuantity) throws InsufficientStockException, IOException {  for (OrderDetails od : orderDetailsList) {  if (od.getOrderDetailID() == orderDetailID) {  int productId = od.getProduct().getProductID();  int oldQuantity = od.getQuantity();  if (newQuantity > oldQuantity) {  int diff = newQuantity - oldQuantity;  if (!InventoryDAOImpl.isProductAvailable(productId, diff)) {  throw new InsufficientStockException("Not enough stock to update quantity.");  }  inventoryDAO.removeFromInventory(productId, diff);  } else if (newQuantity < oldQuantity) {  inventoryDAO.addToInventory(productId, oldQuantity - newQuantity);  }  od.setQuantity(newQuantity);  return;  }  }  }  @Override  public void removeOrderDetail(int orderDetailID) {  orderDetailsList.removeIf(od -> od.getOrderDetailID() == orderDetailID);  }  @Override  public List<OrderDetails> getAllOrderDetails() {  return orderDetailsList;  }  @Override  public OrderDetails getOrderDetailById(int orderDetailID) {  for (OrderDetails od : orderDetailsList) {  if (od.getOrderDetailID() == orderDetailID) {  return od;  }  }  return null;  }  } |

ProductDAO.java

|  |
| --- |
| package dao;  import java.io.IOException;  import java.util.List;  import entity.Products;  import exception.DuplicateEntryException;  public interface ProductDAO {  void addProduct(Products product) throws DuplicateEntryException, IOException;  void updateProduct(Products product) throws IOException;  void listAllProducts();  List<Products> getAllProducts();  List<Products> searchProductsByName(String name);  List<Products> searchProductsByPriceRange(double minPrice, double maxPrice);  Products getProductById(int productID) throws IOException;  void removeProduct(int productId, int quantity) throws IOException;  List<Products> searchProductsByKeyword(String keyword) throws IOException;    } |

ProductDAOImpl.java

|  |
| --- |
| package dao;  import entity.Products;  import exception.\*;  import util.DBConnUtil;  import java.io.IOException;  import java.sql.Connection;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.util.\*;  public class ProductDAOImpl implements ProductDAO {  private static List<Products> productList = new ArrayList<>();  private InventoryDAO inventoryDAO = new InventoryDAOImpl();  private Connection connection;    public List<Products> searchProductsByKeyword(String keyword) throws IOException {  List<Products> result = new ArrayList<>();  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  try {  conn = DBConnUtil.getDbConnection();  String sql = "SELECT \* FROM Products WHERE ProductName LIKE ?";  stmt = conn.prepareStatement(sql);  stmt.setString(1, "%" + keyword + "%");  rs = stmt.executeQuery();  while (rs.next()) {  Products product = new Products(  rs.getInt("ProductID"),  rs.getString("ProductName"),  rs.getString("Category"),  rs.getDouble("Price")    );  result.add(product);  }  } catch (SQLException e) {  System.out.println("Error searching products: " + e.getMessage());  } finally {  try {  if (rs != null) rs.close();  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  return result;  }  @Override  public void addProduct(Products product) throws DuplicateEntryException, IOException {  if (InventoryDAOImpl.isProductAvailable(product.getProductID(), 1)) {  throw new DuplicateEntryException("Product with the same ID already exists.");  }  PreparedStatement stmt = null;  try {  connection = DBConnUtil.getDbConnection();  String insertQuery = "INSERT INTO Products (ProductID, ProductName, Description, Price) VALUES (?, ?, ?, ?)";  stmt = connection.prepareStatement(insertQuery);  stmt.setInt(1, product.getProductID());  stmt.setString(2, product.getProductName());  stmt.setString(3, product.getDescription());  stmt.setDouble(4, product.getPrice());  int rowsAffected = stmt.executeUpdate();  if (rowsAffected > 0) {  System.out.println("Product added successfully.");  productList.add(product); // Maintain in-memory list  } else {  System.out.println("Failed to add product.");  }  } catch (SQLException e) {  System.out.println("Error while adding product: " + e.getMessage());  } finally {  closeResources(stmt, connection);  }  }  @Override  public void updateProduct(Products product) throws IOException {  PreparedStatement stmt = null;  try {  connection = DBConnUtil.getDbConnection();  String updateQuery = "UPDATE Products SET ProductName = ?, Description = ?, Price = ? WHERE ProductID = ?";  stmt = connection.prepareStatement(updateQuery);  stmt.setString(1, product.getProductName());  stmt.setString(2, product.getDescription());  stmt.setDouble(3, product.getPrice());  stmt.setInt(4, product.getProductID());  int rowsAffected = stmt.executeUpdate();  if (rowsAffected > 0) {  System.out.println("Product updated successfully.");  } else {  System.out.println("No product found with the given ProductID.");  }  } catch (SQLException e) {  System.out.println("Error while updating product: " + e.getMessage());  } finally {  closeResources(stmt, connection);  }  }  @Override  public void removeProduct(int productId, int quantity) throws IOException {  InventoryDAO inventoryDAO = new InventoryDAOImpl();  inventoryDAO.removeFromInventory(productId, quantity);  String checkInventory = "SELECT \* FROM inventory WHERE ProductID = ?";  String deleteProduct = "DELETE FROM products WHERE ProductID = ?";  try (Connection conn = DBConnUtil.getDbConnection();  PreparedStatement checkStmt = conn.prepareStatement(checkInventory)) {  checkStmt.setInt(1, productId);  ResultSet rs = checkStmt.executeQuery();  if (!rs.next()) {  try (PreparedStatement deleteStmt = conn.prepareStatement(deleteProduct)) {  deleteStmt.setInt(1, productId);  int rows = deleteStmt.executeUpdate();  if (rows > 0) {  System.out.println("✅ Product deleted from database (Product ID: " + productId + ")");  } else {  System.out.println("⚠️ Product not deleted (Product ID: " + productId + ")");  }  }  } else {  System.out.println("ℹ️ Inventory still exists for Product ID: " + productId);  }  } catch (SQLException e) {  System.out.println("❌ Error in removeProduct: " + e.getMessage());  }  }  @Override  public void listAllProducts() {  for (Products p : productList) {  p.getProductDetails();  System.out.println("-------------------------");  }  }  public List<Products> searchProductsByName(String name) {  if (name == null || name.trim().isEmpty()) {  throw new IllegalArgumentException("Search name cannot be null or empty.");  }  return productList.stream()  .filter(p -> p.getProductName().toLowerCase().contains(name.toLowerCase()))  .toList();  }  public List<Products> searchProductsByPriceRange(double minPrice, double maxPrice) {  if (minPrice > maxPrice) {  throw new IllegalArgumentException("Minimum price cannot be greater than maximum price.");  }  return productList.stream()  .filter(p -> p.getPrice() >= minPrice && p.getPrice() <= maxPrice)  .toList();  }  public List<Products> getAllProducts() {  return productList;  }  public void updateInventoryOnOrder(int productID, int quantityOrdered) throws IOException {  try {  inventoryDAO.removeFromInventory(productID, quantityOrdered);  } catch (InsufficientStockException e) {  System.out.println("[ERROR] Order failed due to insufficient stock: " + e.getMessage());  }  }  private void closeResources(PreparedStatement stmt, Connection conn) {  try {  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.out.println("Error closing resources: " + e.getMessage());  }  }  public Products getProductById(int productID) throws IOException {  Products product = null;  Connection conn = null;  PreparedStatement stmt = null;  try {  conn = DBConnUtil.getDbConnection();  String query = "SELECT \* FROM Products WHERE ProductID = ?";  stmt = conn.prepareStatement(query);  stmt.setInt(1, productID);  ResultSet rs = stmt.executeQuery();  if (rs.next()) {  product = new Products(  rs.getInt("ProductID"),  rs.getString("ProductName"),  rs.getString("Description"),  rs.getDouble("Price")  );  }  } catch (SQLException e) {  System.out.println("Error fetching product by ID: " + e.getMessage());  } finally {  try {  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.out.println("Error closing DB resources: " + e.getMessage());  }  }  return product;  }  } |

**Package:exception**

AuthenticationException.java

|  |
| --- |
| package exception;  public class AuthenticationException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public AuthenticationException(String message) {  super(message);  }  } |

AuthorizationException.java

|  |
| --- |
| package exception;  public class AuthorizationException extends Exception {  private static final long ***serialVersionUID*** = 1L;  public AuthorizationException(String message) {  super(message);  }  } |

ConcurrencyException.java

|  |
| --- |
| package exception;  public class ConcurrencyException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public ConcurrencyException(String message) {  super(message);  }  } |

DuplicateEntryException.java

|  |
| --- |
| package exception;  public class DuplicateEntryException extends RuntimeException {  private static final long ***serialVersionUID*** = 1L;  public DuplicateEntryException(String message) {  super(message);  }  } |

DuplicateProductException.java

|  |
| --- |
| package exception;  public class DuplicateProductException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public DuplicateProductException(String message) {  super(message);  }  } |

IncompleteOrderDetailException.java

|  |
| --- |
| package exception;  public class IncompleteOrderDetailException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public IncompleteOrderDetailException(String message) {  super(message);  }  } |

InsufficientStockException.java

|  |
| --- |
| package exception;  public class InsufficientStockException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public InsufficientStockException(String message) {  super(message);  }  } |

InvalidDataException.java

|  |
| --- |
| package exception;  public class InvalidDataException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public InvalidDataException(String message) {  super(message);  }  } |

PaymentFailedException.java

|  |
| --- |
| package exception;  public class PaymentFailedException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public PaymentFailedException(String message) {  super(message);  }  } |

ProductInUseException.java

|  |
| --- |
| package exception;  public class ProductInUseException extends RuntimeException {  private static final long ***serialVersionUID*** = 1L;  public ProductInUseException(String message) {  super(message);  }  } |

ProductNotFoundException.java

|  |
| --- |
| package exception;  public class ProductNotFoundException extends RuntimeException {    private static final long ***serialVersionUID*** = 1L;  public ProductNotFoundException(String message) {  super(message);  }  } |

**Package:main**

MainModule.java

|  |
| --- |
| package main;  import entity.\*;  import java.time.format.DateTimeFormatter;  import util.\*;  import java.util.\*;  import java.io.IOException;  import java.sql.\*;  import java.time.LocalDateTime;  import exception.\*;  import dao.\*;  public class MainModule {  public static void main(String[] args) throws SQLException, IOException {  Scanner scanner = new Scanner(System.***in***);  while (true) {  System.***out***.println("\n--- TechShop System ---");  System.***out***.println("1. Customer Registration");  System.***out***.println("2. Product Catalog Management");  System.***out***.println("3. Place Order");  System.***out***.println("4. Track Order Status");  System.***out***.println("5. Inventory Management");  System.***out***.println("6. Sales Reporting");  System.***out***.println("7. Update Customer Account");  System.***out***.println("8. Payment Processing");  System.***out***.println("9. Product Search and Recommendations");  System.***out***.println("0. Exit");  System.***out***.print("Choose an option: ");  int choice = scanner.nextInt();  scanner.nextLine();  switch (choice) {  case 1:    *handleCustomerRegistration*(scanner);  break;  case 2:    *handleProductCatalogManagement*(scanner);  break;  case 3:    *handlePlaceOrder*(scanner);  break;  case 4:  *handleTrackOrderStatus*(scanner);  break;  case 5:    *handleInventoryManagement*(scanner);  break;  case 6:    *handleSalesReporting*(scanner);  break;  case 7:    try {  *handleUpdateCustomerAccount*(scanner);  } catch (Exception e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  break;  case 8:    *handlePaymentProcessing*(scanner);  break;  case 9:    *handleProductSearchAndRecommendations*(scanner);  break;  case 0:  System.***out***.println("Exiting TechShop System.");  return;  default:  System.***out***.println("Invalid choice. Please try again.");  }  }  }  private static void handleCustomerRegistration(Scanner scanner) {  System.***out***.println("\n--- Customer Registration ---");  System.***out***.println("Enter CustomerId: ");  int CustomerID = scanner.nextInt();  scanner.nextLine();  System.***out***.println("Enter FirstName: ");  String FirstName = scanner.nextLine();  System.***out***.println("Enter LastName: ");  String LastName = scanner.nextLine();  System.***out***.println("Enter Email: ");  String Email= scanner.nextLine();    System.***out***.println("Enter Phone: ");  String Phone = scanner.nextLine();  System.***out***.println("Enter Address: ");  String Address= scanner.nextLine();  Customers newCustomer = new Customers(CustomerID, FirstName,LastName, Email, Phone, Address);  CustomerDAO customerDAO = new CustomerDAOImpl();  try {  customerDAO.registerCustomer(newCustomer);  } catch (DuplicateEntryException | IOException e) {  System.***out***.println("Error: " + e.getMessage());  }  }    private static void handleProductCatalogManagement(Scanner scanner) throws IOException {  System.***out***.println("\n--- Product Catalog Management ---");  System.***out***.println("1. Add Product");  System.***out***.println("2. Update Product");  System.***out***.println("3. Delete Product");  System.***out***.print("Choose an option: ");  int choice = scanner.nextInt();  scanner.nextLine();  switch (choice) {  case 1:  System.***out***.println("Enter Product ID: ");  int productID = scanner.nextInt();  scanner.nextLine();  System.***out***.println("Enter Product Name: ");  String productName = scanner.nextLine();  System.***out***.println("Enter Description: ");  String description = scanner.nextLine();  System.***out***.println("Enter Price: ");  double price = scanner.nextDouble();  scanner.nextLine();  Products newProduct = new Products(productID, productName, description, price);  ProductDAO productDAO = new ProductDAOImpl();  try {  productDAO.addProduct(newProduct);  } catch (DuplicateEntryException | IOException e) {  System.***out***.println("Error: " + e.getMessage());  }  break;  case 2:  System.***out***.println("Enter Product ID to Update: ");  int productID1 = scanner.nextInt();  scanner.nextLine();  System.***out***.println("Enter Updated Product Name: ");  String updatedName = scanner.nextLine();  System.***out***.println("Enter Updated Price: ");  double updatedPrice = scanner.nextDouble();  scanner.nextLine();  System.***out***.println("Enter Updated Description: ");  String updatedDescription = scanner.nextLine();    Products updatedProduct = new Products(productID1, updatedName, updatedDescription, updatedPrice);  ProductDAO productDAO1 = new ProductDAOImpl();  try {  productDAO1.updateProduct(updatedProduct);  } catch (DuplicateEntryException e) {  System.***out***.println("Error: " + e.getMessage());  }  break;  case 3:  System.***out***.println("Enter Product ID to Delete: ");  int deleteProductID = scanner.nextInt();  scanner.nextLine();  System.***out***.println("Enter quantity to Delete: ");  int quantity = scanner.nextInt();  scanner.nextLine();  ProductDAO productDAO2 = new ProductDAOImpl();  try {  productDAO2.removeProduct(deleteProductID,quantity);  } catch (ProductNotFoundException | ProductInUseException e) {  System.***out***.println("Error: " + e.getMessage());  }  break;  default:  System.***out***.println("Invalid choice. Please try again.");  }  }  private static void handlePlaceOrder(Scanner scanner) throws SQLException, IOException {  System.***out***.println("--- Place Order ---");  System.***out***.print("Enter Customer ID: ");  int customerID = scanner.nextInt();  CustomerDAO customerDAO = new CustomerDAOImpl();  Customers customer = customerDAO.getCustomerById(customerID);  if (customer == null) {  System.***out***.println("❌ Customer not found. Please check the Customer ID.");  return;  }  System.***out***.print("Enter number of products in the order: ");  int count = scanner.nextInt();  List<OrderDetails> orderDetailsList = new ArrayList<>();  ProductDAO productDAO = new ProductDAOImpl();  InventoryDAO inventoryDAO = new InventoryDAOImpl();  double totalAmount = 0.0;  for (int i = 0; i < count; i++) {  System.***out***.print("Enter Product ID: ");  int productID = scanner.nextInt();  Products product = productDAO.getProductById(productID);  if (product != null) {  System.***out***.print("Enter Quantity: ");  int quantity = scanner.nextInt();  int availableStock = inventoryDAO.getProductStock(productID);  if (quantity > availableStock) {  System.***out***.println("Only " + availableStock + " units available in stock.");  System.***out***.print("Do you want to place order for available quantity? (yes/no): ");  String choice = scanner.next();  if (choice.equalsIgnoreCase("yes")) {  quantity = availableStock;  } else {  System.***out***.println("Skipping product: " + product.getProductName());  continue;  }  }  double price = product.getPrice() \* quantity;  totalAmount += price;  orderDetailsList.add(new OrderDetails(0, product, quantity, price));  } else {  System.***out***.println("Product not found.");  }  }  if (orderDetailsList.isEmpty()) {  System.***out***.println("❌ No products selected. Order was not placed.");  return;  }  Orders order = new Orders(customer, LocalDateTime.*now*(), totalAmount, "Processing");  order.setOrderDetails(orderDetailsList);  OrderDAO dao = new OrderDAOImpl();  dao.placeOrder(order, orderDetailsList);  }  private static void handleTrackOrderStatus(Scanner scanner) throws IOException, SQLException {  System.***out***.println("\n--- Track Order Status ---");  System.***out***.print("Enter Order ID: ");  int orderID = scanner.nextInt();  scanner.nextLine();  OrderDAO orderDAO = new OrderDAOImpl();  Orders order = orderDAO.getOrderById(orderID);  if (order != null) {  DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*("yyyy-MM-dd");  String formattedDate = order.getOrderDate().format(formatter);  System.***out***.println("\n📦 Order Details:");  System.***out***.println("Order ID : " + order.getOrderID());  System.***out***.println("Order Date : " + formattedDate);  System.***out***.println("Total Amount : $" + order.getTotalAmount());  System.***out***.println("Status : " + order.getStatus());  } else {  System.***out***.println("❌ Order not found for ID: " + orderID);  }  }  private static void handleInventoryManagement(Scanner scanner) throws IOException, SQLException {  InventoryDAO inventoryDAO = new InventoryDAOImpl();  while (true) {  System.***out***.println("\n--- 🗂 Inventory Management ---");  System.***out***.println("1. Add Product to Inventory");  System.***out***.println("2. Remove Product from Inventory");  System.***out***.println("3. View All Products in Inventory");  System.***out***.println("4. Go Back to Main Menu");  System.***out***.print("Enter your choice: ");  int choice = scanner.nextInt();  switch (choice) {  case 1:  System.***out***.print("Enter Inventory ID: ");  int addInventoryID = scanner.nextInt();  System.***out***.print("Enter Product ID: ");  int addProductID = scanner.nextInt();  System.***out***.print("Enter Quantity to Add: ");  int addQuantity = scanner.nextInt();  if (addQuantity <= 0) {  System.***out***.println("❌ Quantity must be greater than 0.");  break;  }  if (inventoryDAO.addProductToInventory(addInventoryID, addProductID, addQuantity)) {  System.***out***.println("✅ Product added to inventory.");  } else {  System.***out***.println("❌ Failed to add product. Check IDs.");  }  break;  case 2:  System.***out***.print("Enter Inventory ID: ");  int removeInventoryID = scanner.nextInt();  System.***out***.print("Enter Product ID: ");  int removeProductID = scanner.nextInt();  System.***out***.print("Enter Quantity to Remove: ");  int removeQuantity = scanner.nextInt();  if (removeQuantity <= 0) {  System.***out***.println("❌ Quantity must be greater than 0.");  break;  }  if (inventoryDAO.removeProductFromInventory(removeInventoryID, removeProductID, removeQuantity)) {  System.***out***.println("✅ Product removed from inventory.");  } else {  System.***out***.println("❌ Failed to remove product. Not enough quantity or invalid ID.");  }  break;  case 3:  System.***out***.println("\n📦 All Products in Inventory:");  List<Inventory> allInventory = inventoryDAO.getInventoryList();  for (Inventory inv : allInventory) {    Products prod = inv.getProduct();  System.***out***.println("Product ID: " + prod.getProductID() +  " | Name: " + prod.getProductName() +  " | Quantity: " + inv.getQuantityInStock());  }  break;  case 4:  return;  default:  System.***out***.println("❌ Invalid choice. Please select a valid option.");  }  }  }  private static void handleSalesReporting(Scanner scanner) throws IOException {  System.***out***.println("\n--- Sales Reporting ---");  Connection conn = null;  PreparedStatement stmt = null;  ResultSet rs = null;  try {  conn = DBConnUtil.*getDbConnection*();  String sql = "SELECT COUNT(\*) AS totalOrders, SUM(TotalAmount) AS totalSales FROM Orders";  stmt = conn.prepareStatement(sql);  rs = stmt.executeQuery();  if (rs.next()) {  int totalOrders = rs.getInt("totalOrders");  double totalSales = rs.getDouble("totalSales");  System.***out***.println("Total Orders : " + totalOrders);  System.***out***.println("Total Sales : $" + totalSales);  } else {  System.***out***.println("No orders found.");  }  } catch (SQLException e) {  System.***out***.println("Error generating sales report: " + e.getMessage());  } finally {  try {  if (rs != null) rs.close();  if (stmt != null) stmt.close();  if (conn != null) conn.close();  } catch (SQLException e) {  System.***out***.println("Error closing resources: " + e.getMessage());  }  }  }  private static void handleUpdateCustomerAccount(Scanner scanner) throws Exception {  System.***out***.println("\n--- Update Customer Account ---");  System.***out***.print("Enter Customer ID: ");  int customerId = scanner.nextInt();  scanner.nextLine();  CustomerDAO customerDAO = new CustomerDAOImpl();  Customers existingCustomer = customerDAO.getCustomerById(customerId);  if (existingCustomer == null) {  System.***out***.println("Customer not found.");  return;  }  System.***out***.print("Enter New Email: ");  String newEmail = scanner.nextLine();  System.***out***.print("Enter New Phone: ");  String newPhone = scanner.nextLine();    existingCustomer.setEmail(newEmail);  existingCustomer.setPhone(newPhone);  customerDAO.updateCustomerAccount(existingCustomer);    }  private static void handlePaymentProcessing(Scanner scanner) throws IOException, SQLException {  System.***out***.println("\n--- Payment Processing ---");  System.***out***.print("Enter Order ID: ");  int orderID = scanner.nextInt();  scanner.nextLine();    OrderDAO orderDAO = new OrderDAOImpl();  Orders order = orderDAO.getOrderById(orderID);    if (order == null) {  System.***out***.println("❌ Order not found with ID: " + orderID);  return;  }  System.***out***.println("Total amount to be paid: $" + order.getTotalAmount());  System.***out***.print("Enter payment amount: ");  double paymentAmount = scanner.nextDouble();    if (paymentAmount < order.getTotalAmount()) {  System.***out***.println("❌ Insufficient payment. Please pay the full amount.");  return;  }    double change = paymentAmount - order.getTotalAmount();  System.***out***.println("Payment successful. Change: $" + change);    try {  order.setStatus("Paid");  orderDAO.updateOrderStatus(order);  System.***out***.println("✅ Order status updated to 'Paid'.");  } catch (SQLException e) {  System.***out***.println("Error updating order status: " + e.getMessage());  }  }  private static void handleProductSearchAndRecommendations(Scanner scanner) throws IOException {  System.***out***.println("\n--- Product Search and Recommendations ---");  System.***out***.print("Enter search keyword: ");  String keyword = scanner.nextLine();  ProductDAO productDAO = new ProductDAOImpl();  List<Products> matchingProducts = productDAO.searchProductsByKeyword(keyword);  if (matchingProducts.isEmpty()) {  System.***out***.println("No products found with keyword: " + keyword);  } else {  System.***out***.println("\n🔍 Search Results:");  for (Products product : matchingProducts) {  System.***out***.println("Product ID: " + product.getProductID() +  " | Name: " + product.getProductName() +  " | Price: $" + product.getPrice());  }  }  }  } |

**Package:util**

DBConnUtil.java

|  |
| --- |
| package util;  import java.io.IOException;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  public class DBConnUtil {    private static final String ***fileName***="C:\\Users\\vtcve\\eclipse-workspace\\TECHSHOP\\Techshop\_oops\\src\\db.properties";  public static Connection getDbConnection() throws IOException {  Connection con=null;  String connString=null;  connString=DBPropertyUtil.*getConnectionString*(***fileName***);  if(connString!=null)  {  try  {  con=DriverManager.*getConnection*(connString);    }  catch (SQLException e)  {  System.***out***.println("Error While Establishing DBConnection........");  e.printStackTrace();  }  }  return con;}  } |

DBPropertyUtil.java

|  |
| --- |
| package util;  import java.io.FileInputStream;  import java.io.IOException;  import java.util.Properties;  public class DBPropertyUtil {      public static String getConnectionString(String fileName)throws IOException {    String connStr=null;  Properties props=new Properties();  FileInputStream fis=new FileInputStream(fileName);    props.load(fis);  String user=props.getProperty("user");  String password=props.getProperty("password");  String protocol=props.getProperty("protocol");  String system=props.getProperty("system");  String database=props.getProperty("database");  String port=props.getProperty("port");  connStr=protocol+"//"+system+":"+port+"/"+database+"?user="+user+"&password="+password;  return connStr;  }  } |

Db.properties

|  |
| --- |
| user=root  password=@Tamil9988  protocol=jdbc:mysql:  system=localhost  database=TechShop  port=3306 |

IMPLEMENTATION

1. Customer Registration

A screenshot of a computer

AI-generated content may be incorrect.

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

2. Product Catalog Management

1)Add Product:

**A computer screen with white text

AI-generated content may be incorrect.**

**OUTPUT:**

**A screenshot of a computer

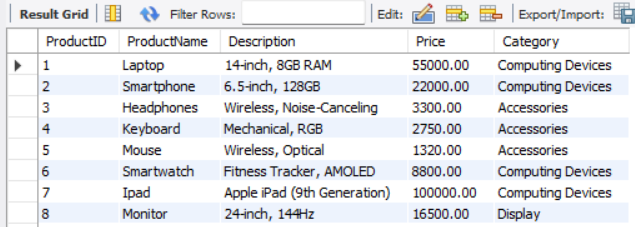
AI-generated content may be incorrect.**

2)Update Product:

**A screenshot of a computer

AI-generated content may be incorrect.**

**OUTPUT:**

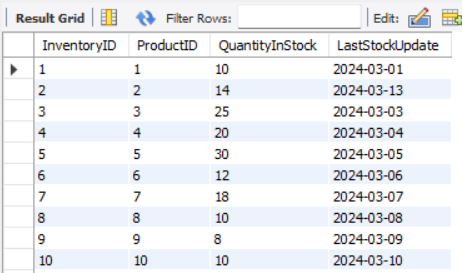
****

3)Delete Product:

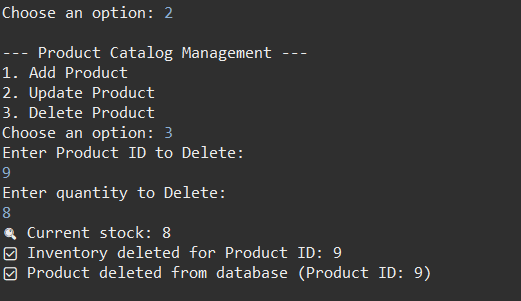
A screenshot of a computer

AI-generated content may be incorrect.

**OUTPUT:**

****

3)Delete Product:

****

**OUTPUT:**

**A screenshot of a computer

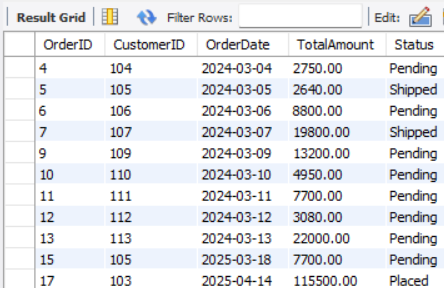
AI-generated content may be incorrect.**

3.Place Order

**A screenshot of a computer

AI-generated content may be incorrect.**

**OUTPUT:Orders Table:**

****

**OrderDetails Table:**

**A screenshot of a table

AI-generated content may be incorrect.**

3.Place Order

**A screen shot of a computer

AI-generated content may be incorrect.**

**OUTPUT:**

**Orders Table:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**OrderDetails Table:**

****

4.Track Order Status

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**OUTPUT matches Orders table:**

****

5. Inventory Management

1) Add Product to Inventory:

**A screenshot of a computer

AI-generated content may be incorrect.**

**OUTPUT:**

**A screenshot of a spreadsheet

AI-generated content may be incorrect.**

2) Remove Product from Inventory:

**A screenshot of a computer

AI-generated content may be incorrect.**

**OUTPUT:**

**A screenshot of a spreadsheet

AI-generated content may be incorrect.**

3)View All Products in Inventory:

**A screenshot of a computer

AI-generated content may be incorrect.**

6. Sales Reporting

**A screen shot of a computer

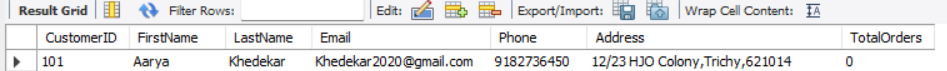
AI-generated content may be incorrect.**

7. Update Customer Account

**A screenshot of a computer

AI-generated content may be incorrect.**

**OUTPUT:**

****

**8. Payment Processing**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**8. Payment Processing**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**9. Product Search and Recommendations**

**A screenshot of a computer

AI-generated content may be incorrect.**