TECHSHOP – ELECTRONIC GADGET SHOP

Task 1: Classes and Their Attributes

Customers Class:

```python
class Customers:
    def __init__(self, customer_id, first_name, last_name, email, phone, address):
        self.CustomerID = customer_id
        self.FirstName = first_name
        self.LastName = last_name
        self.Email = email
        self.Phone = phone
        self.Address = address
        self.orders = []  # Assuming orders are stored in a list

    def CalculateTotalOrders(self):
        return len(self.orders)

    def GetCustomerDetails(self):
        return f"Customer ID: {self.CustomerID}\nName: {self.FirstName} {self.LastName}\nEmail: {self.Email}\nPhone: {self.Phone}\nAddress: {self.Address}"

    def UpdateCustomerInfo(self, email=None, phone=None, address=None):
        if email:
            self.Email = email
        if phone:
            self.Phone = phone
        if address:
            self.Address = address
        # Getter and setter methods for the private attributes

    @property
    def customer_id(self):
        return self.__customer_id
```

```python
@property
def customer_id(self):
    return self.__customer_id

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

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

1 usage
@property
def email(self):
    return self.__email

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

1 usage
@property
def address(self):
    return self.__address
```

```
1 usage
@email.setter
def email(self, value):
    # Implementation logic to validate and set the email attribute
    pass

@email.setter
def email(self, new_email):
    if "@" in new_email:
        self.__email = new_email
    else:
        raise InvalidDataException("Invalid email format. Please provide a valid email address.")
@phone.setter
def phone(self, value):
    # Implementation logic to validate and set the phone attribute
    pass

@address.setter
def address(self, value):
    # Implementation logic to validate and set the address attribute
    pass
```

2.Products class

```
class Products:
    def __init__(self, product_id, product_name, description, price):
        self.ProductID = product_id
        self.ProductName = product_name
        self.Description = description
        self.Price = price

    5 usages (5 dynamic)
    @property
    def product_id(self):
        return self.__product_id

    2 usages (2 dynamic)
    @property
    def product_name(self):
        return self.__product_name

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

    3 usages (2 dynamic)
    @property
    def price(self):
        return self.__price
```

```python
    @price.setter
    def price(self, new_price):
        if new_price >= 0:
            self.__price = new_price
        else:
            raise ValueError("Price cannot be negative.")
    def GetProductDetails(self):
        # Implement code to retrieve and display product details
        pass

    def UpdateProductInfo(self, new_price, new_description):
        self.Price = new_price
        self.Description = new_description

    def IsProductInStock(self):
        # Implement code to check product availability
        pass
```

3.Orders Class:

```python
class Orders:
    def __init__(self, order_id, customer, order_date):
        self.__order_date = order_date
        self.OrderID = order_id
        self.Customer = customer# Composition relationship with Customers class
        self.OrderDate = order_date
        self.TotalAmount = 0  # Initialize total amount to zero
        self.order_status = "Pending"


    7 usages (7 dynamic)
    @property
    def order_id(self):
        return self.__order_id

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

    7 usages (6 dynamic)
    @property
    def order_date(self):
        return self.__order_date
```

```python
@property
def total_amount(self):
    return self.__total_amount

2 usages (2 dynamic)
@total_amount.setter
def total_amount(self, amount):
    if amount >= 0:
        self.__total_amount = amount
    else:
        raise ValueError("Total amount cannot be negative.")

def CalculateTotalAmount(self):
    # Logic to calculate the total amount based on products in the order
    pass

def GetOrderDetails(self):
    # Logic to retrieve and display order details
    pass

def UpdateOrderStatus(self, new_status):
    # Simulating concurrent update scenario
    current_status = self.order_status
    if current_status == "Pending":
        self.order_status = new_status
    else:
        raise ConcurrencyException("Order status has been updated by another user. Please retry.")
```

```python
def CancelOrder(self):
    # Logic to cancel the order and adjust stock levels for products
    pass
def add_order_detail(self, order_detail):
    self.order_details.append(order_detail)

def calculate_total_amount(self):
    total = sum(order_detail.calculate_subtotal() for order_detail in self.order_details)
    return total

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

4.Orderdetails class:

```python
class OrderDetails:
    def __init__(self, order_detail_id, order, product, quantity):
        if not product:
            raise IncompleteOrderException("Product reference is missing in order details")
        self.__order = order
        self.__product = product
        self.__quantity = quantity

    @property
    def order_detail_id(self):
        return self.__order_detail_id

    @property
    def order(self):
        return self.__order

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

    5 usages (3 dynamic)
    @property
    def quantity(self):
        return self.__quantity
```

```python
    @quantity.setter
    def quantity(self, value):
        if value > 0:
            self.__quantity = value
        else:
            raise ValueError("Quantity must be a positive integer.")

    3 usages (2 dynamic)
    def calculate_subtotal(self):
        return self.__product.price * self.__quantity

    def get_order_detail_info(self):
        return f"Order Detail ID: {self.__order_detail_id}\n" \
               f"Product: {self.__product.product_name}\n" \
               f"Quantity: {self.__quantity}\n" \
               f"Subtotal: {self.calculate_subtotal()}"

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

    def add_discount(self):
        # Apply discount logic here
        pass

    def calculate_total_amount(self):
        total = sum(order_detail.calculate_subtotal() for order_detail in self.order_details)
        return total
```

5.Inventory class:

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


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

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

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

    @quantity_in_stock.setter
    def quantity_in_stock(self, new_quantity):
        if new_quantity >= 0:
            self.__quantity_in_stock = new_quantity
        else:
            raise ValueError("Quantity must be a non-negative integer.")
```

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

@last_stock_update.setter
def last_stock_update(self, new_update):
    # Add validation logic for the last stock update if needed
    self.__last_stock_update = new_update

def GetProduct(self):
    return self.Product

def GetQuantityInStock(self):
    return self.QuantityInStock

def AddToInventory(self, quantity):
    self.QuantityInStock += quantity

def RemoveFromInventory(self, quantity):
    if self.QuantityInStock >= quantity:
        self.QuantityInStock -= quantity
    else:
        print("Insufficient quantity in stock.")

def UpdateStockQuantity(self, new_quantity):
    self.QuantityInStock = new_quantity

def IsProductAvailable(self, quantity_to_check):
    return self.QuantityInStock >= quantity_to_check
```

```
    def IsProductAvailable(self, quantity_to_check):
        return self.QuantityInStock >= quantity_to_check

    def GetInventoryValue(self):
        return self.Product.Price * self.QuantityInStock

    def ListLowStockProducts(self, threshold):
        if self.QuantityInStock < threshold:
            return self.Product

    def ListOutOfStockProducts(self):
        if self.QuantityInStock == 0:
            return self.Product

    def ListAllProducts(self):
        return self.Product, self.QuantityInStock

##Data validation
```

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.

```
3 usages
class InvalidDataException(Exception):
    """Exception raised for invalid data."""

    def __init__(self, message):
        self.message = message
        super().__init__(self.message)


1 usage
def validate_email(email):
    """Validate email address format."""
    # Basic email format validation
    if "@" not in email or "." not in email:
        raise InvalidDataException("Invalid email address format")
```

```python
1 usage
def register_user(email):
    try:
        validate_email(email)
        # Proceed with user registration if email is valid
        print("User registered successfully")
    except InvalidDataException as e:
        print(f"Error: {e}")

    # Test with invalid email


invalid_email = "invalid_email"
register_user(invalid_email)
```

Output:

```
Error: Invalid email address format
```

• **Inventory Management:**

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

o Scenario: When processing an order with a quantity that exceeds the available stock.

o Exception Handling: Throw an InsufficientStockException and update the order status accordingly.

```python
#inventory management
2 usages
class InsufficientStockException(Exception):
    """Exception raised for insufficient stock."""

    def __init__(self, product_name, requested_quantity, available_quantity):
        self.product_name = product_name
        self.requested_quantity = requested_quantity
        self.available_quantity = available_quantity
        message = f"Insufficient stock for product '{product_name}'. Requested quantity: {requested_quantity}, Available quantity: {available_quantity}"
        super().__init__(message)


2 usages
class OrderProcessor:
    2 usages
    def process_order(self, product_name, requested_quantity):
        # Check available stock for the product (Assuming inventory management logic here)
        available_quantity = self.check_available_stock(product_name)

        if requested_quantity > available_quantity:
            raise InsufficientStockException(product_name, requested_quantity, available_quantity)

        # Proceed with order processing if stock is sufficient
        print("Order processed successfully")
```

```
    1 usage
    def check_available_stock(self, product_name):
        # Placeholder method to check available stock (replace with actual logic)
        # This could involve querying the database or accessing an inventory management system
        # For simplicity, we'll return a hardcoded value
        if product_name == "Smartphone":
            return 10  # Assuming 10 smartphones are available in stock
        elif product_name == "Laptop":
            return 5  # Assuming 5 laptops are available in stock
        else:
            return 0  # Default to 0 if product is not found in inventory


# Example usage:
usage
def main():
    processor = OrderProcessor()

    try:
        # Attempt to process an order with a quantity that exceeds available stock
        processor.process_order("Smartphone", 15)
    except InsufficientStockException as e:
        # Handle InsufficientStockException by updating order status or informing the user
        print(f"Error: {e}")
```

```
if __name__ == "__main__":
    main()
```

Output :

```
Error: Insufficient stock for product 'Smartphone'. Requested quantity: 15, Available quantity: 10
```

• **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.

```python
## order processing

3 usages
class IncompleteOrderException(Exception):
    """Exception raised for incomplete order details."""

    def __init__(self, message="Incomplete order details: Product reference is missing"):
        self.message = message
        super().__init__(self.message)

2 usages
class OrderProcessor:
    2 usages
    def process_order(self, order_details):
        # Check if any order detail lacks a product reference
        if any(detail.get("product_id") is None for detail in order_details):
            raise IncompleteOrderException()

        # Proceed with order processing if all order details are complete
        print("Order processed successfully")
```

```python
# Example usage:
1 usage
def main():
    processor = OrderProcessor()

    try:
        # Attempt to process an order with incomplete order details
        incomplete_order = [
            {"product_id": 1, "quantity": 2},
            {"quantity": 3}  # Missing product reference
        ]
        processor.process_order(incomplete_order)
    except IncompleteOrderException as e:
        # Handle IncompleteOrderException by informing the user or logging the error
        print(f"Error: {e}")

if __name__ == "__main__":
    main()
```

Output:

```
Error: Incomplete order details: Product reference is missing
```

- **Payment Processing:**

o Challenge: Handling payment failures or declined transactions.

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

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

```python
class PaymentFailedException(Exception):
    """Exception raised for failed payment transactions."""

    def __init__(self, message="Payment failed"):
        self.message = message
        super().__init__(self.message)

class PaymentProcessor:
    def process_payment(self, order_id, amount):
        # Simulate payment processing (replace with actual payment gateway integration)
        # For demonstration purposes, we'll simulate a payment failure scenario
        payment_success = False  # Simulating payment failure

        if not payment_success:
            raise PaymentFailedException()

        # Proceed with order completion if payment is successful
        print("Payment processed successfully")
```

```python
# Example usage:
def main():
    processor = PaymentProcessor()

    try:
        # Attempt to process a payment
        processor.process_payment(order_id=1234, amount=100.00)
    except PaymentFailedException as e:
        # Handle PaymentFailedException by initiating retry or cancellation processes
        print(f"Error: {e}")
        # Retry payment or initiate cancellation process here

if __name__ == "__main__":
    main()
```

Output:

```
Error: Payment failed
```

• File I/O (e.g., Logging):

o Challenge: Logging errors and events to files or databases.

o Scenario: When an error occurs during data persistence (e.g., writing a log entry).

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

```
import logging
2 usages
class FileIOException(Exception):
    """Exception raised for file I/O errors."""

    def __init__(self, message="File I/O error occurred"):
        self.message = message
        super().__init__(self.message)

1 usage
class FileIOHandler:
    def __init__(self, log_file):
        self.log_file = log_file

    1 usage
    def write_to_file(self, data):
        try:
            # Perform file I/O operation (e.g., writing to a log file)
            with open(self.log_file, 'a') as f:
                f.write(data + '\n')
        except IOError as e:
            # Print the file I/O error message
            print("Error: File I/O error occurred -", str(e))
```

```
            print("Error: File I/O error occurred -", str(e))
            raise FileIOException(str(e))

# Example usage:
1 usage
def main():
    log_file = "error_log.txt"
    handler = FileIOHandler(log_file)

    try:
        # Attempt file I/O operation
        handler.write_to_file("Sample log entry")
    except FileIOException as e:
        # Handle file I/O exception
        print(f"Error occurred: {e}")
    finally:
        print("Exception handled successfully.")

if __name__ == "__main__":
    main()
```

Output:

```
Exception handled successfully.
```

- **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.

```python
## database access

import mysql.connector
from mysql.connector import Error
import time

3 usages
class DatabaseAccessException(Exception):
    """Exception raised for database access errors."""

    def __init__(self, message="Database access error occurred"):
        self.message = message
        super().__init__(self.message)

1 usage
class DatabaseManager:
    def __init__(self, db_config):
        self.db_config = db_config
        self.connection = None

    1 usage
    def connect(self):
        try:
            self.connection = mysql.connector.connect(**self.db_config)
            print("Connected to database successfully")
        except Error as e:
            print("Error: Failed to connect to database -", e)
            raise DatabaseAccessException(str(e))

    28 usages (27 dynamic)
    def execute_query(self, query):
        try:
            cursor = self.connection.cursor()
            cursor.execute(query)
            result = cursor.fetchall()
            cursor.close()
            return result
        except Error as e:
            print("Error: Failed to execute query -", e)
            raise DatabaseAccessException(str(e))

# Example usage:
```

```python
# Example usage:
1 usage
def main():
    db_config = {
        'host': 'localhost',
        'database': 'Techshop',
        'user': 'root',
        'password': 'root',
        'port': '3306'
    }

    db_manager = DatabaseManager(db_config)

    # Retry logic with a maximum of 3 attempts
    max_attempts = 3
    attempt = 1
    while attempt <= max_attempts:
        try:
            # Attempt to connect to the database
            db_manager.connect()

            # Example SQL query
            query = "SELECT * FROM customers"
            result = db_manager.execute_query(query)
            print("Query executed successfully")
            print(result)
```

```python
            # Example SQL query
            query = "SELECT * FROM customers"
            result = db_manager.execute_query(query)
            print("Query executed successfully")
            print(result)
            break  # Exit loop if successful
        except DatabaseAccessException as e:
            print(f"Error occurred: {e}")
            if attempt < max_attempts:
                print("Retrying...")
                attempt += 1
                time.sleep(5)  # Wait before retrying
            else:
                print("Maximum retry attempts reached. Exiting.")
                break

if __name__ == "__main__":
    main()
```

Output:

```
Connected to database successfully
```

• **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.

```python
## concurrency control

3 usages
class ConcurrencyException(Exception):
    """Exception raised for concurrency control conflicts."""

    def __init__(self, message="Concurrency control conflict occurred"):
        self.message = message
        super().__init__(self.message)

3 usages
class OrderManager:
    def __init__(self, order_id, current_version):
        self.order_id = order_id
        self.current_version = current_version

    1 usage
    def update_order(self, new_version):
        if new_version != self.current_version:
            raise ConcurrencyException("Order has been modified by another user. Please retry.")
```

```python
# Example usage:
1 usage
def main():
    order_id = 12
    current_version = 5
    new_version = 6

    order_manager = OrderManager(order_id, current_version)

    try:
        order_manager.update_order(new_version)
        print("Order updated successfully")
    except ConcurrencyException as e:
        print(f"Concurrency conflict: {e}")
        print("Please retry the operation.")

if __name__ == "__main__":
    main()
```

Output:

```
Concurrency conflict: Order has been modified by another user. Please retry.
```

• **Security and Authentication:**

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

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

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

```python
## security and authentication
2 usages
class AuthenticationException(Exception):
    """Exception raised for authentication errors."""

    def __init__(self, message="Authentication failed"):
        self.message = message
        super().__init__(self.message)

2 usages
class AuthorizationException(Exception):
    """Exception raised for authorization errors."""

    def __init__(self, message="Unauthorized access"):
        self.message = message
        super().__init__(self.message)

1 usage
class UserManager:
    def __init__(self, username, password):
        self.username = username
        self.password = password
```

```python
    def authenticate(self, entered_password):
        if entered_password != self.password:
            raise AuthenticationException("Invalid username or password")

    1 usage
    def authorize(self, role):
        if role != 'admin':
            raise AuthorizationException("Access denied. You are not authorized to perform this action.")

# Example usage:
1 usage
def main():
    username = 'user123'
    password = 'password123'
    entered_password = 'password123'
    role = 'user'

    user_manager = UserManager(username, password)

    try:
        user_manager.authenticate(entered_password)
        user_manager.authorize(role)
        print("Authentication and authorization successful. Access granted.")
    except AuthenticationException as e:
        print(f"Authentication error: {e}")
    except AuthorizationException as e:
```

```
    try:
        user_manager.authenticate(entered_password)
        user_manager.authorize(role)
        print("Authentication and authorization successful. Access granted.")
    except AuthenticationException as e:
        print(f"Authentication error: {e}")
    except AuthorizationException as e:
        print(f"Authorization error: {e}")

if __name__ == "__main__":
    main()
```

Output:

```
Authorization error: Access denied. You are not authorized to perform this action.
```

Task 6: Collections

• **Managing Products List:**

o Challenge: Maintaining a list of products available for sale (List<Products>).

o Scenario: Adding, updating, and removing products from the list.

o Solution: Implement methods to add, update, and remove products. Handle exceptions for duplicate products, invalid updates, or removal of products with existing orders.

```
class Collections:
    def __init__(self):
        self.products_list = []

    2 usages
    def add_product(self, product):
        if product not in self.products_list:
            self.products_list.append(product)
            print(f"Product {product} added successfully.")
        else:
            raise Exception("Product already exists in the list.")

    1 usage
    def update_product(self, old_product, new_product):
        if old_product in self.products_list:
            index = self.products_list.index(old_product)
            self.products_list[index] = new_product
            print(f"Product {old_product} updated to {new_product}.")
        else:
            raise Exception("Product does not exist in the list for updating.")
```

```
    1 usage
    def update_product(self, old_product, new_product):
        if old_product in self.products_list:
            index = self.products_list.index(old_product)
            self.products_list[index] = new_product
            print(f"Product {old_product} updated to {new_product}.")
        else:
            raise Exception("Product does not exist in the list for updating.")

    1 usage
    def remove_product(self, product):
        if product in self.products_list:
            self.products_list.remove(product)
            print(f"Product {product} removed successfully.")
        else:
            raise Exception("Product not found for removal.")

# Example Usage
techshop_collections = Collections()
techshop_collections.add_product("Laptop")
techshop_collections.add_product("Smartphone")
techshop_collections.update_product( old_product: "Laptop",  new_product: "Gaming Laptop")
techshop_collections.remove_product("Smartphone")
```

Output:

```
Product Laptop added successfully.
Product Smartphone added successfully.
Product Laptop updated to Gaming Laptop.
Product Smartphone removed successfully.
```

• **Managing Orders List:**

o Challenge: Maintaining a list of customer orders (List<Orders>).

o Scenario: Adding new orders, updating order statuses, and removing canceled orders.

o Solution: Implement methods to add new orders, update order statuses, and remove canceled orders. Ensure that updates are synchronized with inventory and payment records.

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

    6 usages
    def add_order(self, order):
        # Implement synchronization with inventory and payment records here
        self.orders.append(order)
        print(f"Order {order.order_id} added successfully.")

    1 usage
    def update_order_status(self, order_id, new_status):
        for order in self.orders:
            if order.order_id == order_id:
                # Implement synchronization with inventory and payment records here
                order.status = new_status
                print(f"Order {order_id} status updated to {new_status}.")
                return
        print(f"Order {order_id} not found.")

    1 usage
    def remove_canceled_orders(self):
        canceled_orders = [order for order in self.orders if order.status == "CANCELED"]
        for order in canceled_orders:
```

```
            self.orders.remove(order)
            print(f"Canceled order {order.order_id} removed successfully.")
# Example usage:
1 usage
def main():
    order_manager = OrderManager()

    # Add new orders
    order1 = Order(1, 101, "2022-01-15", 1000.00, "PENDING")
    order2 = Order(2, 102, "2022-01-20", 1500.00, "PENDING")
    order3 = Order(3, 103, "2022-01-25", 2000.00, "CANCELED")

    order_manager.add_order(order1)
    order_manager.add_order(order2)
    order_manager.add_order(order3)

    # Update order status
    order_manager.update_order_status( order_id: 1,  new_status: "SHIPPED")

    # Remove canceled orders
    order_manager.remove_canceled_orders()

if __name__ == "__main__":
    main()
```

Output:

```
Order 1 added successfully.
Order 2 added successfully.
Order 3 added successfully.
Order 1 status updated to SHIPPED.
Canceled order 3 removed successfully.
```

• **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<Orders> collection and provide custom sorting methods for order date. Consider implementing SortedList if you need frequent sorting operations.

```python
from datetime import datetime

6 usages
class Order:
    def __init__(self, order_id, order_date, total_amount, status):
        self.order_id = order_id
        self.order_date = order_date
        self.total_amount = total_amount
        self.status = status

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

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

    1 usage
    def sort_orders_by_date_asc(self):
        self.orders.sort(key=lambda x: x.order_date)
```

```python
    def sort_orders_by_date_asc(self):
        self.orders.sort(key=lambda x: x.order_date)

    1 usage
    def sort_orders_by_date_desc(self):
        self.orders.sort(key=lambda x: x.order_date, reverse=True)

# Example usage:
1 usage
def main():
    order_manager = OrderManager()

    # Add orders
    order1 = Order(1, datetime( year: 2022,  month: 1,  day: 15), 1000.00, "SHIPPED")
    order2 = Order(2, datetime( year: 2022,  month: 1,  day: 10), 2000.00, "PENDING")
    order3 = Order(3, datetime( year: 2022,  month: 1,  day: 20), 1500.00, "SHIPPED")

    order_manager.add_order(order1)
    order_manager.add_order(order2)
    order_manager.add_order(order3)

    # Sort orders by date in ascending order
    order_manager.sort_orders_by_date_asc()
    print("Orders sorted by date in ascending order:")
    for order in order_manager.orders:
        print(f"Order ID: {order.order_id}, Date: {order.order_date}, Total Amount: {order.total_amount}, Status: {order.status}")
```

```
    # Sort orders by date in ascending order
    order_manager.sort_orders_by_date_asc()
    print("Orders sorted by date in ascending order:")
    for order in order_manager.orders:
        print(f"Order ID: {order.order_id}, Date: {order.order_date}, Total Amount: {order.total_amount}, Status: {order.status}")

    # Sort orders by date in descending order
    order_manager.sort_orders_by_date_desc()
    print("\nOrders sorted by date in descending order:")
    for order in order_manager.orders:
        print(f"Order ID: {order.order_id}, Date: {order.order_date}, Total Amount: {order.total_amount}, Status: {order.status}")

if __name__ == "__main__":
    main()
```

Output:

```
Orders sorted by date in ascending order:
Order ID: 2, Date: 2022-01-10 00:00:00, Total Amount: 2000.0, Status: PENDING
Order ID: 1, Date: 2022-01-15 00:00:00, Total Amount: 1000.0, Status: SHIPPED
Order ID: 3, Date: 2022-01-20 00:00:00, Total Amount: 1500.0, Status: SHIPPED

Orders sorted by date in descending order:
Order ID: 3, Date: 2022-01-20 00:00:00, Total Amount: 1500.0, Status: SHIPPED
Order ID: 1, Date: 2022-01-15 00:00:00, Total Amount: 1000.0, Status: SHIPPED
Order ID: 2, Date: 2022-01-10 00:00:00, Total Amount: 2000.0, Status: PENDING
```

• **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<int, Inventory> where keys are product IDs. Ensure that inventory updates are synchronized with product additions and removals.

```python
class InventoryManagement:
    def __init__(self):
        self.inventory_list = SortedList()

    3 usages
    def add_product_to_inventory(self, product_id, product_name, quantity, price):
        new_product = Inventory(product_id, product_name, quantity, price)
        self.inventory_list.add((product_id, new_product))

    def remove_product_from_inventory(self, product_id):
        for item in self.inventory_list:
            if item[1].product_id == product_id:
                self.inventory_list.remove(item)
                break

    def update_product_quantity(self, product_id, new_quantity):
        for item in self.inventory_list:
            if item[1].product_id == product_id:
                item[1].quantity = new_quantity
                break

    def get_product_details(self, product_id):
        for item in self.inventory_list:
            if item[1].product_id == product_id:
                return item[1]
        return None
```

```python
    2 usages
    def list_all_products(self):
        for item in self.inventory_list:
            print(f"Product ID: {item[1].product_id}, Name: {item[1].product_name}, Quantity: {item[1].quantity}, Price: {item[1].price}")

# Usage
inventory_system = InventoryManagement()
inventory_system.add_product_to_inventory( product_id: 1,  product_name: "Product A",  quantity: 50,  price: 10.99)
inventory_system.add_product_to_inventory( product_id: 2,  product_name: "Product B",  quantity: 30,  price: 20.49)
inventory_system.add_product_to_inventory( product_id: 3,  product_name: "Product C",  quantity: 20,  price: 15.75)

inventory_system.list_all_products()

#decrementing from inventory
3 usages
class Inventory:
    def __init__(self, product, quantity):
        self.product = product
        self.quantity = quantity

    2 usages
    def RemoveFromInventory(self, quantity):
        if self.quantity >= quantity:
            self.quantity -= quantity
            print(f"{quantity} units of {self.product} removed from inventory.")
        else:
            raise ValueError("Insufficient stock to fulfill the order.")
```

```python
# Example of how to use the Inventory class and update stock quantities
inventory_item = Inventory( product: "Product A",  quantity: 50)
print("Initial Stock Quantity:", inventory_item.quantity)

try:
    inventory_item.RemoveFromInventory(20)  # Try to remove 20 units
    print("Updated Stock Quantity:", inventory_item.quantity)
    inventory_item.RemoveFromInventory(10)  # Try to remove 40 units (will raise an exception)
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
Initial Stock Quantity: 50
20 units of Product A removed from inventory.
Updated Stock Quantity: 30
10 units of Product A removed from inventory.
```

• **Handling Inventory Updates:**

o Challenge: Ensuring that inventory is updated correctly when processing orders.

o Scenario: Decrementing product quantities in stock when orders are placed.

o Solution: Implement a method to update inventory quantities when orders are processed. Handle exceptions for insufficient stock.

```python
## handling inventory updates
3 usages
class InventoryManager:
    def __init__(self):
        # Initialize inventory as a dictionary with product IDs as keys and quantities as values
        self.inventory = {
            57: 20,  # Example: Product ID 57 with initial quantity of 20
            # Add more products as needed
        }

    2 usages
    def update_inventory(self, product_id, quantity):
        # Check if the product exists in the inventory
        if product_id in self.inventory:
            # Check if there is sufficient stock
            if self.inventory[product_id] >= quantity:
                # Decrement the product quantity
                self.inventory[product_id] -= quantity
                print(f"Inventory updated: Product ID {product_id}, New Quantity: {self.inventory[product_id]}")
            else:
                # Raise exception for insufficient stock
                raise ValueError(f"Insufficient stock for Product ID {product_id}")
        else:
            # Product does not exist in inventory
            raise ValueError(f"Product ID {product_id} not found in inventory")
```

```
            raise ValueError(f"Product ID {product_id} not found in inventory")

# Create an instance of InventoryManager
inventory_manager = InventoryManager()

# Attempt to update inventory with product ID 57 and quantity 10
try:
    inventory_manager.update_inventory( product_id: 57,  quantity: 10)
except ValueError as e:
    print(e)  # Print error message for insufficient stock

# Attempt to update inventory with product ID 100 (non-existing)
try:
    inventory_manager.update_inventory( product_id: 60,  quantity: 5)
except ValueError as e:
    print(e)  # Print error message for non-existing product
```

Output:

```
Inventory updated: Product ID 57, New Quantity: 10
Product ID 60 not found in inventory
```

• **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<Products> collection. Handle exceptions for invalid search criteria.

```python
# product search and retrieval
4 usages
class Product:
    def __init__(self, name, category, price):
        self.name = name
        self.category = category
        self.price = price


# Sample list of products
products = [
    Product( name: "Laptop",  category: "Electronics",  price: 1200),
    Product( name: "Headphones",  category: "Electronics",  price: 100),
    Product( name: "T-shirt",  category: "Apparel",  price: 20),
    Product( name: "Sneakers",  category: "Apparel",  price: 50)
]


3 usages
def search_products(criteria, value):
    try:
        if criteria == "name":
            result = [product for product in products if product.name.lower() == value.lower()]
        elif criteria == "category":
            result = [product for product in products if product.category.lower() == value.lower()]
        else:
            raise ValueError("Invalid search criteria. Please search by 'name' or 'category'.")
```

```python
        if result:
            for product in result:
                print(f"Product Name: {product.name}, Category: {product.category}, Price: ${product.price}")
        else:
            print("No products found matching the search criteria.")

    except ValueError as e:
        print(f"Error: {e}")


# Search for products by name
search_products( criteria: "name",  value: "Laptop")

# Search for products by category
search_products( criteria: "category",  value: "Apparel")

# Search for products with an invalid criteria
search_products( criteria: "invalid",  value: "test")
```

Output:

```
Product Name: Laptop, Category: Electronics, Price: $1200
Product Name: T-shirt, Category: Apparel, Price: $20
Product Name: Sneakers, Category: Apparel, Price: $50
Error: Invalid search criteria. Please search by 'name' or 'category'.
```

• **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.

```python
# duplicate product handling

from sortedcontainers import SortedList


2 usages
class Inventory:
    def __init__(self, product_id, quantity):
        self.product_id = product_id
        self.quantity = quantity


2 usages
class InventoryManager:
    def __init__(self):
        self.inventory = SortedList()

    3 usages
    def add_product_to_inventory(self, product_id, quantity):
        # Check for duplicate product IDs
        for key, inventory in self.inventory:
            if key == product_id:
                raise ValueError(f"Product with ID {product_id} already exists in inventory.")

        new_inventory = Inventory(product_id, quantity)
        self.inventory.add((product_id, new_inventory))
        print(f"Product {product_id} added to inventory with quantity {quantity}.")
```

```python
    def remove_product_from_inventory(self, product_id):
        for idx, (key, inventory) in enumerate(self.inventory):
            if key == product_id:
                del self.inventory[idx]
                print(f"Product {product_id} removed from inventory.")
                return
        print(f"Product {product_id} not found in inventory.")

    def update_inventory_quantity(self, product_id, new_quantity):
        for idx, (key, inventory) in enumerate(self.inventory):
            if key == product_id:
                inventory.quantity = new_quantity
                self.inventory[idx] = (product_id, inventory)
                print(f"Inventory quantity for product {product_id} updated to {new_quantity}.")
                return
        print(f"Product {product_id} not found in inventory.")

    1 usage
    def get_inventory_info(self):
        print("Inventory Information:")
        for key, inventory in self.inventory:
            print(f"Product ID: {inventory.product_id}, Quantity: {inventory.quantity}")
```

```
        print("Inventory Information:")
        for key, inventory in self.inventory:
            print(f"Product ID: {inventory.product_id}, Quantity: {inventory.quantity}")


# Example usage:
1 usage
def main():
    inventory_manager = InventoryManager()

    # Add products to inventory
    inventory_manager.add_product_to_inventory( product_id: 1,  quantity: 10)
    inventory_manager.add_product_to_inventory( product_id: 2,  quantity: 20)

    # Try to add a duplicate product
    try:
        inventory_manager.add_product_to_inventory( product_id: 1,  quantity: 15)
    except ValueError as e:
        print(e)  # Print error message

    # Display inventory information
    inventory_manager.get_inventory_info()


if __name__ == "__main__":
    main()
```

Output:

```
Product with ID 1 already exists in inventory.
```

- **Payment Records List:**

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

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.

```python
# payment record list

class PaymentRecord:
    def __init__(self, order_id, amount, payment_method, status):
        self.order_id = order_id
        self.amount = amount
        self.payment_method = payment_method
        self.status = status

class PaymentRecordManager:
    def __init__(self):
        self.payment_records = []

    def record_payment(self, payment_record):
        self.payment_records.append(payment_record)
        print(f"Payment recorded for order {payment_record.order_id}.")

    def update_payment_status(self, order_id, new_status):
        for record in self.payment_records:
            if record.order_id == order_id:
                record.status = new_status
                print(f"Payment status updated for order {order_id} to {new_status}.")
                return
        print(f"No payment record found for order {order_id}.")

# Example usage:
def main():
    payment_manager = PaymentRecordManager()

    # Record payments
    payment1 = PaymentRecord(order_id: 1, amount: 1000.00, payment_method: "Credit Card", status: "PAID")
    payment2 = PaymentRecord(order_id: 2, amount: 1500.00, payment_method: "PayPal", status: "PENDING")

    payment_manager.record_payment(payment1)
    payment_manager.record_payment(payment2)

    # Update payment status
    payment_manager.update_payment_status(order_id: 1, new_status: "COMPLETE")

if __name__ == "__main__":
    main()
```

Output:

```
Inventory Information:
Product ID: 1, Quantity: 10
Product ID: 2, Quantity: 20
Payment recorded for order 1.
Payment recorded for order 2.
Payment status updated for order 1 to COMPLETE.
```

• **OrderDetails and Products Relationship:**

o Challenge: Managing the relationship between OrderDetails and Products.

o Scenario: Ensuring that order details accurately reflect the products available in the inventory.

o Solution: Implement methods to validate product availability in the inventory before adding order details. Handle exceptions for unavailable products.

```
class InventoryManager:
    def __init__(self):
        self.inventory = {}

    1 usage (1 dynamic)
    def check_product_availability(self, product_id, quantity):
        if product_id in self.inventory:
            available_quantity = self.inventory[product_id]
            return available_quantity >= quantity
        return False

1 usage
class OrderDetailsManager:
    def __init__(self, inventory_manager):
        self.order_details = []
        self.inventory_manager = inventory_manager

    3 usages
    def add_order_detail(self, order_id, product_id, quantity):
        if self.inventory_manager.check_product_availability(product_id, quantity):
            order_detail = OrderDetail(order_id, product_id, quantity)
            self.order_details.append(order_detail)
            print(f"Order detail added for product {product_id} with quantity {quantity}.")
        else:
            print(f"Product {product_id} is not available in sufficient quantity.")
```

```
        else:
            print(f"Product {product_id} is not available in sufficient quantity.")

# Example usage:
1 usage
def main():
    inventory_manager = InventoryManager()
    inventory_manager.inventory = {1: 10, 2: 5, 3: 20}  # Example inventory data

    order_details_manager = OrderDetailsManager(inventory_manager)

    # Add order details
    order_details_manager.add_order_detail( order_id: 1,  product_id: 1,  quantity: 5)  # Available
    order_details_manager.add_order_detail( order_id: 2,  product_id: 2,  quantity: 10)  # Available
    order_details_manager.add_order_detail( order_id: 3,  product_id: 3,  quantity: 25)  # Not available

if __name__ == "__main__":
    main()
```

Output:

```
Order detail added for product 1 with quantity 5.
Product 2 is not available in sufficient quantity.
Product 3 is not available in sufficient quantity.
```

Task 7: Database Connectivity

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

```python
Database connectivity
mport mysql.connector

usages
lass DatabaseConnector:
    def __init__(self, host, username, password, database):
        self.host = host
        self.username = username
        self.password = password
        self.database = database
        self.connection = None

    3 usages (1 dynamic)
    def open_connection(self):
        self.connection = mysql.connector.connect(
            host=self.host,
            user=self.username,
            password=self.password,
            database=self.database
        )

    3 usages (1 dynamic)
    def close_connection(self):
        if self.connection:
            self.connection.close()
```

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

Customers class:

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

    def create_customer(self, name, email, phone, address):
        query = f"INSERT INTO customers (name, email, phone, address) VALUES ('{name}', '{email}', '{phone}', '{address}')"
        self.db_connector.execute_query(query)

    def read_customer(self, customer_id):
        query = f"SELECT * FROM customers WHERE id = {customer_id}"
        result = self.db_connector.execute_query(query)
        return result

    def update_customer(self, customer_id, email=None, phone=None, address=None):
        query = f"UPDATE customers SET"
        if email:
            query += f" email = '{email}',"
        if phone:
            query += f" phone = '{phone}',"
        if address:
            query += f" address = '{address}',"
        query = query.rstrip(',') + f" WHERE id = {customer_id}"
        self.db_connector.execute_query(query)

    def delete_customer(self, customer_id):
        query = f"DELETE FROM customers WHERE id = {customer_id}"
        self.db_connector.execute_query(query)
```

Products class:

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

    def create_product(self, name, price, description):
        query = f"INSERT INTO products (name, price, description) VALUES ('{name}', {price}, '{description}')"
        self.db_connector.execute_query(query)

    def read_product(self, product_id):
        query = f"SELECT * FROM products WHERE id = {product_id}"
        result = self.db_connector.execute_query(query)
        return result

    def update_product(self, product_id, price=None, description=None):
        query = f"UPDATE products SET"
        if price:
            query += f" price = {price},"
        if description:
            query += f" description = '{description}',"
        query = query.rstrip(',') + f" WHERE id = {product_id}"
        self.db_connector.execute_query(query)

    def delete_product(self, product_id):
        query = f"DELETE FROM products WHERE id = {product_id}"
        self.db_connector.execute_query(query)
```

Orders class:

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

    def create_order(self, customer_id, product_id, quantity):
        query = f"INSERT INTO orders (customer_id, product_id, quantity) VALUES ({customer_id}, {product_id}, {quantity})"
        self.db_connector.execute_query(query)

    def read_order(self, order_id):
        query = f"SELECT * FROM orders WHERE id = {order_id}"
        result = self.db_connector.execute_query(query)
        return result

    def update_order(self, order_id, status):
        query = f"UPDATE orders SET status = '{status}' WHERE id = {order_id}"
        self.db_connector.execute_query(query)

    def delete_order(self, order_id):
        query = f"DELETE FROM orders WHERE id = {order_id}"
        self.db_connector.execute_query(query)
```

Orderdetails class:

```
class OrderDetails:
    def __init__(self, db_connector):
        self.db_connector = db_connector

    def create_order_detail(self, order_id, product_id, quantity):
        query = f"INSERT INTO order_details (order_id, product_id, quantity) VALUES ({order_id}, {product_id}, {quantity})"
        self.db_connector.execute_query(query)

    def read_order_detail(self, order_detail_id):
        query = f"SELECT * FROM order_details WHERE id = {order_detail_id}"
        result = self.db_connector.execute_query(query)
        return result

    def update_order_detail(self, order_detail_id, quantity):
        query = f"UPDATE order_details SET quantity = {quantity} WHERE id = {order_detail_id}"
        self.db_connector.execute_query(query)

    def delete_order_detail(self, order_detail_id):
        query = f"DELETE FROM order_details WHERE id = {order_detail_id}"
        self.db_connector.execute_query(query)
2 usages
```

Inventory class:

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

    def add_to_inventory(self, product_id, quantity):
        query = f"UPDATE inventory SET quantity = quantity + {quantity} WHERE product_id = {product_id}"
        self.db_connector.execute_query(query)

    def remove_from_inventory(self, product_id, quantity):
        query = f"UPDATE inventory SET quantity = quantity - {quantity} WHERE product_id = {product_id}"
        self.db_connector.execute_query(query)

    def update_stock_quantity(self, product_id, new_quantity):
        query = f"UPDATE inventory SET quantity = {new_quantity} WHERE product_id = {product_id}"
        self.db_connector.execute_query(query)

    def is_product_available(self, product_id, quantity_to_check):
        query = f"SELECT quantity FROM inventory WHERE product_id = {product_id}"
        result = self.db_connector.execute_query(query)
        if result and result[0][0] >= quantity_to_check:
            return True
        return False
```

```python
    def get_inventory_value(self):
        query = "SELECT SUM(products.price * inventory.quantity) FROM products INNER JOIN inventory ON products.id = inventory.product_id"
        result = self.db_connector.execute_query(query)
        return result[0][0]

    def list_low_stock_products(self, threshold):
        query = f"SELECT products.name, inventory.quantity FROM products INNER JOIN inventory ON products.id = inventory.product_id WHERE inventory.quantity < {thres
        result = self.db_connector.execute_query(query)
        return result

    def list_out_of_stock_products(self):
        query = "SELECT products.name FROM products LEFT JOIN inventory ON products.id = inventory.product_id WHERE inventory.quantity IS NULL"
        result = self.db_connector.execute_query(query)
        return result

    def list_all_products(self):
        query = "SELECT products.name, inventory.quantity FROM products INNER JOIN inventory ON products.id = inventory.product_id"
        result = self.db_connector.execute_query(query)
        return result
```

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.

```python
import mysql.connector

# Establish connection to the database
mydb = mysql.connector.connect(
  host="localhost",
  user="root",
  password="root",
  database="techshop"
)

# Create a cursor object
mycursor = mydb.cursor()
1 usage
def register_customer(customerid,firstname, lastname,email, phone,Address):
    sql = "INSERT INTO customers (customerid,firstname, lastname,email, phone,Address) VALUES (%s, %s, %s,%s,%s,%s)"
    val = (customerid,firstname, lastname,email, phone,Address)
    mycursor.execute(sql, val)
    mydb.commit()
    print("Customer registered successfully!")
try:
    register_customer( customerid: 1, firstname: "John", lastname: " Doe",  email: "johndoe@example.com",  phone: "1234567890", Address: "123 main st")
except mysql.connector.IntegrityError as e:
    print("Error: Duplicate email address. Please use a different email.")
except Exception as e:
    print("An error occurred:", e)
finally:
    print("Customer registered successfully!")
mycursor.close()
mydb.close()
```

Output:

```
Customer registered successfully!
```

2: Product Catalog Management

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

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

```python
# Create a cursor object to interact with the database
cursor = db_connection.cursor()

# Function to update product information in the database
def update_product_info(product_id, new_price, new_description):
    update_query = "UPDATE products SET price = %s, description = %s WHERE id = %s"
    cursor.execute(update_query, params: (new_price, new_description, product_id))
    db_connection.commit()
    print("Product information updated successfully.")

# Function to add a new product to the catalog
def add_new_product(product_name, price, description):
    insert_query = "INSERT INTO products (name, price, description) VALUES (%s, %s, %s)"
    cursor.execute(insert_query, params: (product_name, price, description))
    db_connection.commit()
    print("New product added to the catalog.")

# Function to delete a product from the catalog
def delete_product(product_id):
    delete_query = "DELETE FROM products WHERE id = %s"
    cursor.execute(delete_query, params: (product_id,))
    db_connection.commit()
    print("Product deleted from the catalog.")

# Close the cursor and database connection
cursor.close()
db_connection.close()
```

Output:

```
Order ID 12 is currently 'SHIPPED'.
```

3: Placing Customer Orders

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

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

```python
# Creating a cursor to execute SQL queries
mycursor = mydb.cursor()

1 usage
def place_order(orderid,customerid,orderdate, totalamount, status):
    sql = "INSERT INTO orders (orderid,customerid,orderdate, totalamount, status) VALUES (%s, %s, %s,%s,%s)"
    val = (orderid,customerid,orderdate, totalamount, status)
    mycursor.execute(sql, val)
    mydb.commit()
    print("Order placed sucessfully!")
1 usage
def calculate_order_total(orderdetailid):
    sql = "SELECT SUM(products.price * orderdetails.quantity) FROM orderdetails JOIN products ON orderdetails.productid = products.productid WHERE orderdetailid = %s
    val = (orderdetailid,)
    mycursor.execute(sql, val)
    total_amount = mycursor.fetchone()[0]
    return total_amount
place_order( orderid: 889, customerid: 1, orderdate: "2003-1-1", totalamount: 1000, status: "pending")
calculate_order_total(132)
```

Output:

```
Order placed sucessfully!
```

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.

```python
tracking order status

Establishing a connection to the database
import mysql.connector
from mysql.connector import Error

usages
class DatabaseConnector:
    def __init__(self, host="localhost", database="TechShop", user="root", password="root"):
        self.host = host
        self.database = database
        self.user = user
        self.password = password
        self.conn = None

    3 usages (1 dynamic)
    def open_connection(self):
        try:
            self.conn = mysql.connector.connect(
                host=self.host,
                database=self.database,
                user=self.user,
                password=self.password
            )
            if self.conn.is_connected():
                return self.conn
        except Error as e:
            print(f"Error connecting to MySQL database: {e}")
```

```
    3 usages (1 dynamic)
    def close_connection(self):
        if self.conn is not None and self.conn.is_connected():
            self.conn.close()
            print("MySQL connection is closed")

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


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

    1 usage
    def get_order_status(self, orderId):
        conn = self.db_connector.open_connection()
        if conn is not None:
            try:
                cursor = conn.cursor()
                query = "SELECT status FROM orders WHERE orderId = %s"  # Adjusted the column name here
                cursor.execute(query, (orderId,))
                result = cursor.fetchone()
                if result:
                    return f"Order ID {orderId} is currently '{result[0]}'."
                else:
```

```
        if conn is not None:
            try:
                cursor = conn.cursor()
                query = "SELECT status FROM orders WHERE orderId = %s"  # Adjusted the column name here
                cursor.execute(query, (orderId,))
                result = cursor.fetchone()
                if result:
                    return f"Order ID {orderId} is currently '{result[0]}'."
                else:
                    return "Order not found."
            except Error as e:
                print(f"Error fetching order status: {e}")
            finally:
                cursor.close()
                self.db_connector.close_connection()
        else:
            return "Failed to connect to the database."


if __name__ == "__main__":
    db_connector = DatabaseConnector(user='root', password='root')  # Add correct credentials
    orders = Orders(db_connector)
    print(orders.get_order_status(12))  # Assuming 12 is a valid order ID
```

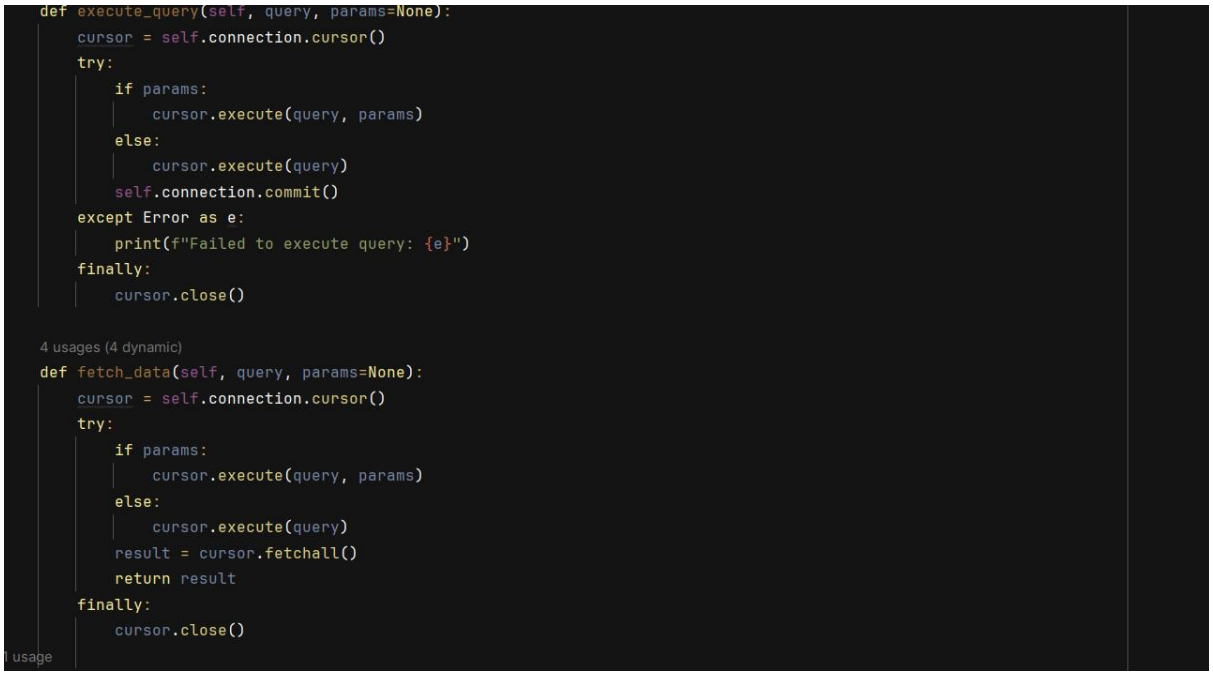
Output:

```
Order ID 12 is currently 'SHIPPED'.
```

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.



```
def execute_query(self, query, params=None):
    cursor = self.connection.cursor()
    try:
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        self.connection.commit()
    except Error as e:
        print(f"Failed to execute query: {e}")
    finally:
        cursor.close()

4 usages (4 dynamic)
def fetch_data(self, query, params=None):
    cursor = self.connection.cursor()
    try:
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        result = cursor.fetchall()
        return result
    finally:
        cursor.close()
1 usage
```

```python
class InventoryManagement:
    def __init__(self, db_connector):
        self.db = db_connector

    1 usage
    def add_product(self, productid,productname, description, price):
        query = "INSERT INTO products (productid,productname, description, price) VALUES (%s, %s, %s, %s)"
        params = (productid,productname, description, price)
        self.db.execute_query(query, params)

    1 usage
    def update_product_price(self, productid, new_price):
        query = "UPDATE products SET price = %s WHERE productid = %s"
        params = (new_price, productid)
        self.db.execute_query(query, params)

    1 usage
    def remove_product(self, productid):
        query = "DELETE FROM products WHERE productid = %s"
        params = (productid,)
        self.db.execute_query(query, params)

    1 usage
    def list_all_products(self):
        query = "SELECT * FROM products"
        return self.db.fetch_data(query)
```

```python
1 usage
def main():
    host = 'localhost'
    database = 'TechShop'
    user = 'root'
    password = 'root'

    db_connector = DatabaseConnector(host, database, user, password)
    db_connector.open_connection()

    inventory_manager = InventoryManagement(db_connector)

    # Adding a product
    inventory_manager.add_product( productid: 80, productname: "Dell Laptop",  description: "Latest laptop",  price: 150000)

    # Updating product quantity
    inventory_manager.update_product_price( productid: 1,  new_price: 2500)  # Assuming '1' is the product ID

    # Removing a product
    inventory_manager.remove_product(2)  # Assuming '2' is the product ID to be deleted

    # Listing all products
    products = inventory_manager.list_all_products()
    for product in products:
        print(product)

    db_connector.close_connection()
```

```python
    db_connector.close_connection()

if __name__ == "__main__":
    main()
```

Output:

```
Inventory Information:
Product ID: 1, Quantity: 10
Product ID: 2, Quantity: 20
```

```
Product 1 added to inventory with quantity 10.
Product 2 added to inventory with quantity 20.
```

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.

```
1 usage
class SalesReporting:
    def __init__(self, db_connector):
        self.db = db_connector

    1 usage
    def total_sales(self):
        query = """
        SELECT SUM(od.Quantity * p.Price) AS TotalSales
        FROM orderdetails od
        JOIN products p ON od.ProductID = p.ProductID;
        """
        results = self.db.fetch_data(query)
        return results[0][0] if results else 0

    1 usage
    def sales_by_category(self):
        query = """
        SELECT p.Categories, SUM(od.Quantity * p.Price) AS Sales
        FROM orderdetails od
        JOIN products p ON od.ProductID = p.ProductID
        GROUP BY p.Categories;
        """
        return self.db.fetch_data(query)
```

```
1 usage
def daily_sales_report(self, date):
    query = """
    SELECT o.OrderDate, SUM(od.Quantity * p.Price) AS DailySales
    FROM orders o
    JOIN orderdetails od ON o.OrderID = od.OrderID
    JOIN products p ON od.ProductID = p.ProductID
    WHERE o.OrderDate = %s
    GROUP BY o.OrderDate;
    """
    return self.db.fetch_data(query, (date,))
```

```
database = 'TechShop'
user = 'root'
password = 'root'

db_connector = DatabaseConnector(host, database, user, password)
db_connector.open_connection()

reporting = SalesReporting(db_connector)

# Total sales
total_sales = reporting.total_sales()
print(f"Total Sales: ${total_sales}")

# Sales by category
category_sales = reporting.sales_by_category()
for category, sales in category_sales:
    print(f"{category}: ${sales}")

# Daily sales report
daily_sales = reporting.daily_sales_report('2023-05-04')  # example date
for report in daily_sales:
    print(f"Sales on {report[0]}: ${report[1]}")

db_connector.close_connection()

__name__ == "__main__":
main()
```

Output:

```
Total Sales: $84150.00
Mobile phone: $2200.00
Drone: $15400.00
Headphones: $13750.00
Camera: $6600.00
Home appliances: $22000.00
laptop: $6600.00
tablet: $17600.00
MySQL connection is closed
```

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.

```python
import mysql.connector
from mysql.connector import Error

1 usage
class UserProfileManager:
    def __init__(self, host, database, user, password):
        self.host = host
        self.database = database
        self.user = user
        self.password = password
        self.connection = None

    2 usages (1 dynamic)
    def open_connection(self):
        try:
            self.connection = mysql.connector.connect(
                host=self.host,
                database=self.database,
                user=self.user,
                password=self.password
            )
            if self.connection.is_connected():
                print("Connected to MySQL database")
        except Error as e:
            print(f"Error while connecting to MySQL: {e}")
```

```
    2 usages (1 dynamic)
    def close_connection(self):
        if self.connection and self.connection.is_connected():
            self.connection.close()
            print("MySQL connection is closed")

    1 usage
    def update_customer_info(self, customer_id, new_email, new_phone):
        try:
            cursor = self.connection.cursor()
            update_query = """
            UPDATE customers
            SET Email = %s, Phone = %s
            WHERE CustomerID = %s
            """
            cursor.execute(update_query, (new_email, new_phone, customer_id))
            self.connection.commit()
            print("Customer information updated successfully")
        except Error as e:
            print(f"Error while updating customer information: {e}")
        finally:
            cursor.close()
```

```
# Example usage:
1 usage
def main():
    host = 'localhost'
    database = 'Techshop'
    user = 'root'
    password = 'root'

    manager = UserProfileManager(host, database, user, password)
    manager.open_connection()

    # Example: Update customer information
    customer_id = 1
    new_email = 'new_email@example.com'
    new_phone = '123-456-7890'
    manager.update_customer_info(customer_id, new_email, new_phone)

    manager.close_connection()

if __name__ == "__main__":
    main()
```

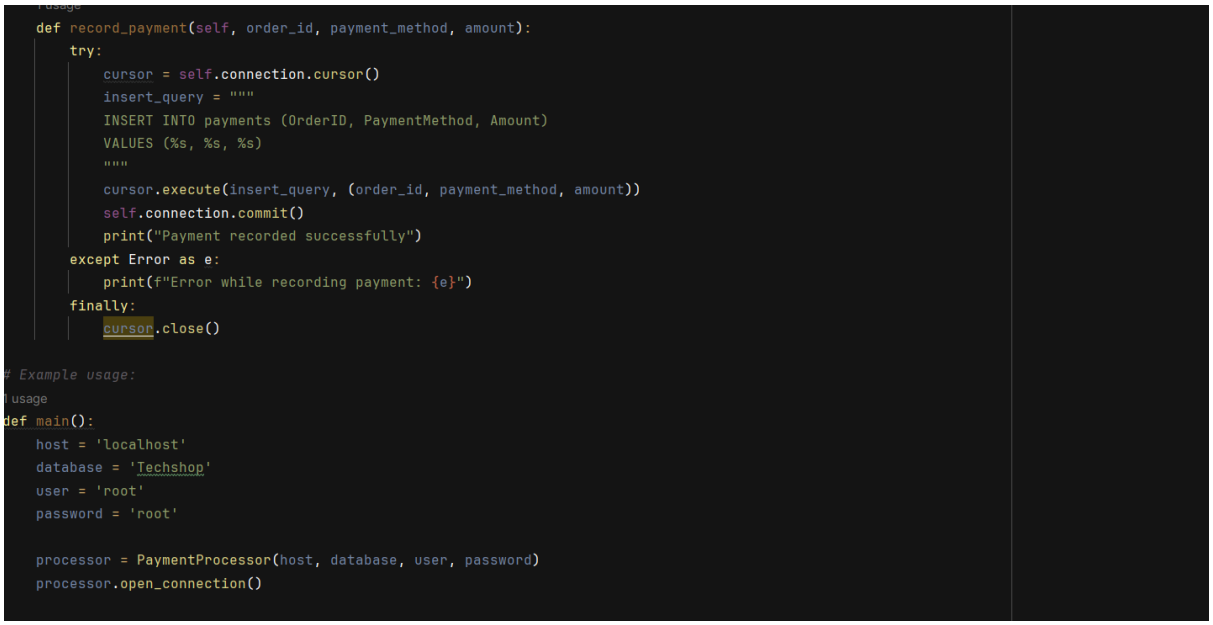
Output:

```
Connected to MySQL database
Customer information updated successfully
MySQL connection is closed
```

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.

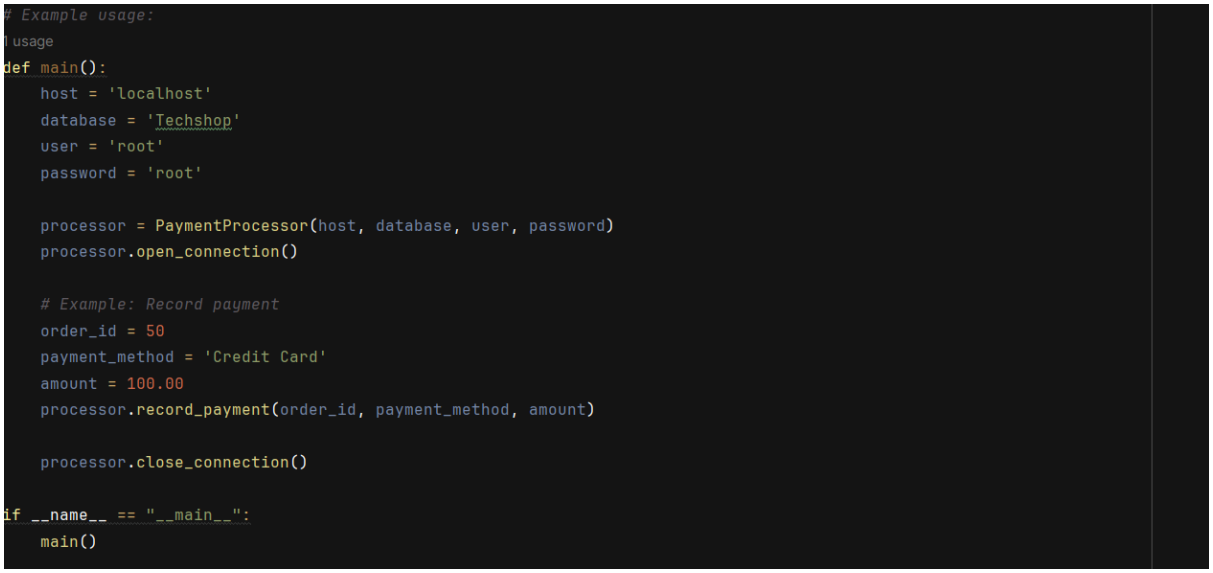
Task: Develop a payment processing system that interacts with the database to record payment transactions, validate payment information, and handle errors.



```
    def record_payment(self, order_id, payment_method, amount):
        try:
            cursor = self.connection.cursor()
            insert_query = """
            INSERT INTO payments (OrderID, PaymentMethod, Amount)
            VALUES (%s, %s, %s)
            """
            cursor.execute(insert_query, (order_id, payment_method, amount))
            self.connection.commit()
            print("Payment recorded successfully")
        except Error as e:
            print(f"Error while recording payment: {e}")
        finally:
            cursor.close()

# Example usage:
usage
def main():
    host = 'localhost'
    database = 'Techshop'
    user = 'root'
    password = 'root'

    processor = PaymentProcessor(host, database, user, password)
    processor.open_connection()
```



```
# Example usage:
usage
def main():
    host = 'localhost'
    database = 'Techshop'
    user = 'root'
    password = 'root'

    processor = PaymentProcessor(host, database, user, password)
    processor.open_connection()

    # Example: Record payment
    order_id = 50
    payment_method = 'Credit Card'
    amount = 100.00
    processor.record_payment(order_id, payment_method, amount)

    processor.close_connection()

if __name__ == "__main__":
    main()
```

Output:

```
Connected to MySQL database
Payment recorded successfully
MySQL connection is closed
```

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.

```
1 usage
def search_products(self, keyword):
    try:
        cursor = self.connection.cursor(dictionary=True)
        search_query = """
        SELECT *
        FROM products
        WHERE ProductName LIKE %s OR Description LIKE %s
        """
        cursor.execute(search_query, (f"%{keyword}%", f"%{keyword}%"))
        products = cursor.fetchall()
        return products
    except Error as e:
        print(f"Error while searching products: {e}")
        return []

1 usage
def recommend_products(self, category):
    try:
        cursor = self.connection.cursor(dictionary=True)
        recommend_query = """
        SELECT *
        FROM products
        WHERE Categories = %s
        """
        cursor.execute(recommend_query, (category,))
```

```
    1 usage
    def recommend_products(self, category):
        try:
            cursor = self.connection.cursor(dictionary=True)
            recommend_query = """
            SELECT *
            FROM products
            WHERE Categories = %s
            """
            cursor.execute(recommend_query, (category,))
            products = cursor.fetchall()
            return products
        except Error as e:
            print(f"Error while recommending products: {e}")
            return []

# Example usage:
 usage
def main():
    host = 'localhost'
    database = 'Techshop'
    user = 'root'
    password = 'root'

    engine = ProductSearchEngine(host, database, user, password)
```

```
# Example usage:
1 usage
def main():
    host = 'localhost'
    database = 'Techshop'
    user = 'root'
    password = 'root'

    engine = ProductSearchEngine(host, database, user, password)
    engine.open_connection()

    # Example: Search for products
    keyword = 'smartphone'
    results = engine.search_products(keyword)
    print("Search Results:")
    for product in results:
        print(product)

    # Example: Get product recommendations
    category = 'Mobile phone'
    recommendations = engine.recommend_products(category)
    print("\nProduct Recommendations:")
    for product in recommendations:
        print(product)
```

```
    keyword = 'smartphone'
    results = engine.search_products(keyword)
    print("Search Results:")
    for product in results:
        print(product)

    # Example: Get product recommendations
    category = 'Mobile phone'
    recommendations = engine.recommend_products(category)
    print("\nProduct Recommendations:")
    for product in recommendations:
        print(product)

    engine.close_connection()

if __name__ == "__main__":
    main()
```

Output:

```
Connected to MySQL database
Search Results:
{'ProductID': 50, 'ProductName': 'Smartphone X', 'Description': 'Latest model with advanced features', 'Price': Decimal('1000.00'), 'Categories': 'Mobile phone'}
{'ProductID': 51, 'ProductName': 'Smartphone', 'Description': 'Latest model with advanced features', 'Price': Decimal('2200.00'), 'Categories': 'Mobile phone'}

Product Recommendations:
{'ProductID': 50, 'ProductName': 'Smartphone X', 'Description': 'Latest model with advanced features', 'Price': Decimal('1000.00'), 'Categories': 'Mobile phone'}
{'ProductID': 51, 'ProductName': 'Smartphone', 'Description': 'Latest model with advanced features', 'Price': Decimal('2200.00'), 'Categories': 'Mobile phone'}
MySQL connection is closed
```