**MODULE BANK**

**Question 1:**

**A) Implement a simple banking system with a class: "BankAccount" with the following attributes:**

* **Variables**: accountNumber, balance, ownerName
* **Constructor**: BankAccount(int accountNumber, double balance, String ownerName)
* **Methods**: double getBalance(), void deposit(double amount), void withdraw(double amount)
* **Exceptions**: “InsufficientFundsException” for throwing

**B) Write a program to create a BankAccount object, deposit some initial amount, and then perform the following operations:**

* Withdraw an amount less than the available balance.
* Withdraw an amount greater than the available balance (should throw InsufficientFundsException).
* Print the final balance after the withdrawals.
* Ensure that the program handles the InsufficientFundsException gracefully and displays an appropriate error message.

**Answer:**

**A) Implementing the BankAccount Class**

In this question, we will implement a BankAccount class that models a simple bank account with the attributes accountNumber, balance, and ownerName. We will also define methods to get the balance, deposit money, and withdraw money. The withdraw method will throw an InsufficientFundsException if the withdrawal amount exceeds the available balance.

**Code:**

class InsufficientFundsException extends Exception {

public InsufficientFundsException(String message) {

super(message);

}

}

class BankAccount {

private int accountNumber;

private double balance;

private String ownerName;

public BankAccount(int accountNumber, double balance, String ownerName) {

this.accountNumber = accountNumber;

this.balance = balance;

this.ownerName = ownerName;

}

public double getBalance() {

return balance;

}

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) throws InsufficientFundsException {

if (amount > balance) {

throw new InsufficientFundsException("Insufficient funds for withdrawal");

}

balance -= amount;

}

}

**B) Demonstrating the BankAccount Class**

Here, we will create a BankAccount object, deposit an initial amount, and then perform several withdrawal operations, demonstrating how the InsufficientFundsException is handled.

java

Copy code

public class Main {

public static void main(String[] args) {

BankAccount account = new BankAccount(123456, 500.00, "John Doe");

try {

account.deposit(100.00);

System.out.println("Current Balance: $" + account.getBalance());

account.withdraw(50.00); // Withdraw less than the balance

System.out.println("After withdrawal, Balance: $" + account.getBalance());

account.withdraw(600.00); // Attempt to withdraw more than the balance

} catch (InsufficientFundsException e) {

System.out.println(e.getMessage());

} finally {

System.out.println("Final Balance: $" + account.getBalance());

}

}

}

**Explanation:**

* The BankAccount class is designed with attributes accountNumber, balance, and ownerName.
* The deposit method adds the specified amount to the account balance.
* The withdraw method checks if the withdrawal amount is less than or equal to the current balance. If not, it throws an InsufficientFundsException.
* In the main program, the account balance is manipulated using the deposit and withdrawal methods. The program also demonstrates how to handle exceptions when attempting to withdraw more money than is available.

**Question 2:**

**Develop an inventory management system for a retail store. The system should allow users to perform various operations on the inventory management system using arrays and implement complex functionality.**

**Requirements:**

1. Each product should have the following attributes: productID, name, quantity, and price.
2. Implement a method to display the current inventory with all product details.
3. Implement a method to add a new product to the inventory.
4. Implement a method to search for a product by its productID and display its details.
5. Implement a method to update the quantity of a product in the inventory.
6. Implement a method to calculate the total value of the inventory (quantity multiplied by price for each product).
7. Implement a method to sort the inventory by product name in alphabetical order.
8. Implement a method to find the product(s) with the highest price and display their details.
9. Implement a method to remove a product from the inventory.

**Answer:**

**A) Implementing the Inventory Management System**

We will create a Product class to represent individual products and an Inventory class to manage the array of products and perform various operations like adding, searching, and updating products.

**CODE:**

class Product {

private int productID;

private String name;

private int quantity;

private double price;

public Product(int productID, String name, int quantity, double price) {

this.productID = productID;

this.name = name;

this.quantity = quantity;

this.price = price;

}

public int getProductID() {

return productID;

}

public String getName() {

return name;

}

public int getQuantity() {

return quantity;

}

public void setQuantity(int quantity) {

this.quantity = quantity;

}

public double getPrice() {

return price;

}

public double getTotalValue() {

return quantity \* price;

}

@Override

public String toString() {

return "Product ID: " + productID + ", Name: " + name + ", Quantity: " + quantity + ", Price: $" + price;

}

}

class Inventory {

private Product[] products;

private int count;

public Inventory(int size) {

products = new Product[size];

count = 0;

}

public void addProduct(Product product) {

if (count < products.length) {

products[count++] = product;

} else {

System.out.println("Inventory is full!");

}

}

public void displayInventory() {

for (int i = 0; i < count; i++) {

System.out.println(products[i]);

}

}

public Product searchProduct(int productID) {

for (int i = 0; i < count; i++) {

if (products[i].getProductID() == productID) {

return products[i];

}

}

return null;

}

public void updateProductQuantity(int productID, int newQuantity) {

Product product = searchProduct(productID);

if (product != null) {

product.setQuantity(newQuantity);

} else {

System.out.println("Product not found!");

}

}

public double calculateTotalInventoryValue() {

double totalValue = 0.0;

for (int i = 0; i < count; i++) {

totalValue += products[i].getTotalValue();

}

return totalValue;

}

public void sortInventoryByName() {

for (int i = 0; i < count - 1; i++) {

for (int j = i + 1; j < count; j++) {

if (products[i].getName().compareTo(products[j].getName()) > 0) {

Product temp = products[i];

products[i] = products[j];

products[j] = temp;

}

}

}

}

public void findMostExpensiveProduct() {

if (count == 0) {

System.out.println("Inventory is empty!");

return;

}

Product mostExpensive = products[0];

for (int i = 1; i < count; i++) {

if (products[i].getPrice() > mostExpensive.getPrice()) {

mostExpensive = products[i];

}

}

System.out.println("Most Expensive Product: " + mostExpensive);

}

public void removeProduct(int productID) {

for (int i = 0; i < count; i++) {

if (products[i].getProductID() == productID) {

for (int j = i; j < count - 1; j++) {

products[j] = products[j + 1];

}

products[--count] = null;

System.out.println("Product removed successfully!");

return;

}

}

System.out.println("Product not found!");

}

}

**B) Demonstrating the Inventory Management System**

We will create an Inventory object, add products, and demonstrate the various operations like searching, updating, sorting, and calculating the total inventory value.

**CODE:**

public class Main {

public static void main(String[] args) {

Inventory inventory = new Inventory(10);

Product product1 = new Product(1, "Laptop", 5, 800.00);

Product product2 = new Product(2, "Smartphone", 10, 500.00);

Product product3 = new Product(3, "Tablet", 8, 300.00);

inventory.addProduct(product1);

inventory.addProduct(product2);

inventory.addProduct(product3);

System.out.println("Current Inventory:");

inventory.displayInventory();

System.out.println("\nSearching for Product with ID 2:");

Product searchedProduct = inventory.searchProduct(2);

System.out.println(searchedProduct != null ? searchedProduct : "Product not found!");

System.out.println("\nUpdating Quantity of Product with ID 1:");

inventory.updateProductQuantity(1, 7);

inventory.displayInventory();

System.out.println("\nTotal Inventory Value: $" + inventory.calculateTotalInventoryValue());

System.out.println("\nSorting Inventory by Product Name:");

inventory.sortInventoryByName();

inventory.displayInventory();

System.out.println("\nFinding the Most Expensive Product:");

inventory.findMostExpensiveProduct();

System.out.println("\nRemoving Product with ID 3:");

inventory.removeProduct(3);

inventory.displayInventory();

}

}

**Explanation:**

* The Product class models the attributes and behaviors of individual products.
* The Inventory class manages an array of products, providing methods to add, display, search, update, sort, and remove products.
* The main program demonstrates the functionality of the inventory system, showing how to manage a retail store's inventory effectively.

**Question 3:**

**Develop a music player application. The application should utilize an abstract class to represent a generic media player and provide functionality for specific types of media players such as audio and video players.**

**Requirements:**

1. Design an abstract class MediaPlayer that represents the common properties and methods for all media players. Include attributes such as name, duration, and volume, as well as methods for playing, pausing, and stopping the media.
2. Implement at least two subclasses that inherit from the MediaPlayer class. One subclass should represent an audio player and the other subclass should represent a video player. Each subclass should provide specific methods and attributes related to their respective media types.
3. Each subclass should override the appropriate methods from the MediaPlayer class to implement the functionality specific to their media type (e.g., audio player should only support audio playback).
4. Implement a class called MusicPlayerApplication that provides functionality to interact with the media players. This class should contain an array or collection to store instances of different media players.
5. Include methods in the MusicPlayerApplication class to add new media players, display information about all media players, play, pause, and stop the media, and adjust the volume.
6. Encourage creativity by allowing users to customize media player settings (e.g., by providing options for repeat mode, shuffle mode, equalizer settings).

**Answer:**

**A) Implementing the MediaPlayer and Subclasses**

We will create an abstract MediaPlayer class with common attributes like name, duration, and volume, along with abstract methods to play, pause, and stop the media. Subclasses like AudioPlayer and VideoPlayer will extend MediaPlayer and implement these methods.

**CODE:**

abstract class MediaPlayer {

protected String name;

protected int duration; // in seconds

protected int volume;

public MediaPlayer(String name, int duration, int volume) {

this.name = name;

this.duration = duration;

this.volume = volume;

}

public abstract void play();

public abstract void pause();

public abstract void stop();

public void setVolume(int volume) {

this.volume = volume;

System.out.println("Volume set to: " + volume);

}

public void displayInfo() {

System.out.println("Name: " + name + ", Duration: " + duration + " sec, Volume: " + volume);

}

}

class AudioPlayer extends MediaPlayer {

public AudioPlayer(String name, int duration, int volume) {

super(name, duration, volume);

}

@Override

public void play() {

System.out.println("Playing audio: " + name);

}

@Override

public void pause() {

System.out.println("Pausing audio: " + name);

}

@Override

public void stop() {

System.out.println("Stopping audio: " + name);

}

public void enableEqualizer() {

System.out.println("Equalizer enabled for audio: " + name);

}

}

class VideoPlayer extends MediaPlayer {

public VideoPlayer(String name, int duration, int volume) {

super(name, duration, volume);

}

@Override

public void play() {

System.out.println("Playing video: " + name);

}

@Override

public void pause() {

System.out.println("Pausing video: " + name);

}

@Override

public void stop() {

System.out.println("Stopping video: " + name);

}

public void enableSubtitles() {

System.out.println("Subtitles enabled for video: " + name);

}

}

**B) Implementing the MusicPlayerApplication Class**

The MusicPlayerApplication class will manage an array of media players, allowing users to add, display, and control different media players.

**CODE:**

import java.util.ArrayList;

import java.util.List;

class MusicPlayerApplication {

private List<MediaPlayer> mediaPlayers;

public MusicPlayerApplication() {

mediaPlayers = new ArrayList<>();

}

public void addMediaPlayer(MediaPlayer mediaPlayer) {

mediaPlayers.add(mediaPlayer);

}

public void displayAllMediaPlayers() {

for (MediaPlayer player : mediaPlayers) {

player.displayInfo();

}

}

public void playAll() {

for (MediaPlayer player : mediaPlayers) {

player.play();

}

}

public void pauseAll() {

for (MediaPlayer player : mediaPlayers) {

player.pause();

}

}

public void stopAll() {

for (MediaPlayer player : mediaPlayers) {

player.stop();

}

}

}

public class Main {

public static void main(String[] args) {

MusicPlayerApplication app = new MusicPlayerApplication();

MediaPlayer audioPlayer = new AudioPlayer("Song.mp3", 300, 75);

MediaPlayer videoPlayer = new VideoPlayer("Movie.mp4", 7200, 50);

app.addMediaPlayer(audioPlayer);

app.addMediaPlayer(videoPlayer);

System.out.println("Displaying all media players:");

app.displayAllMediaPlayers();

System.out.println("\nPlaying all media:");

app.playAll();

System.out.println("\nPausing all media:");

app.pauseAll();

System.out.println("\nStopping all media:");

app.stopAll();

audioPlayer.setVolume(80);

videoPlayer.setVolume(60);

((AudioPlayer) audioPlayer).enableEqualizer();

((VideoPlayer) videoPlayer).enableSubtitles();

}

}

**Explanation:**

* The MediaPlayer class is abstract and serves as a blueprint for different types of media players.
* The AudioPlayer and VideoPlayer classes inherit from MediaPlayer and implement media-specific behaviors.
* The MusicPlayerApplication class allows managing multiple media players, providing functions to play, pause, stop, and display information about each player.

**Question 4:**

**A) Implement a banking system, design a class hierarchy that includes a base class representing a bank account and derived classes for different account types, such as savings account, checking account, and credit card account. Utilize inheritance to inherit common attributes and methods from the base class while implementing specific features in the derived classes.**

**B) Develop a program that represents a messaging application. Create an interface called Messenger with methods for sending and receiving messages. Implement the Messenger interface in different classes representing different messaging protocols (e.g., SMS, Email) and demonstrate the usage of interfaces in the messaging application.**

**Answer:**

**A) Implementing the Bank Account Hierarchy**

We will design a base class BankAccount and derived classes SavingsAccount, CheckingAccount, and CreditCardAccount. Each derived class will inherit common attributes like accountNumber and balance and will have additional features specific to that account type.

**CODE:**

class BankAccount {

protected int accountNumber;

protected double balance;

public BankAccount(int accountNumber, double balance) {

this.accountNumber = accountNumber;

this.balance = balance;

}

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) throws Exception {

if (amount > balance) {

throw new Exception("Insufficient funds");

}

balance -= amount;

}

public double getBalance() {

return balance;

}

@Override

public String toString() {

return "Account Number: " + accountNumber + ", Balance: $" + balance;

}

}

class SavingsAccount extends BankAccount {

private double interestRate;

public SavingsAccount(int accountNumber, double balance, double interestRate) {

super(accountNumber, balance);

this.interestRate = interestRate;

}

public void applyInterest() {

balance += balance \* interestRate;

}

}

class CheckingAccount extends BankAccount {

private double overdraftLimit;

public CheckingAccount(int accountNumber, double balance, double overdraftLimit) {

super(accountNumber, balance);

this.overdraftLimit = overdraftLimit;

}

@Override

public void withdraw(double amount) throws Exception {

if (amount > balance + overdraftLimit) {

throw new Exception("Exceeds overdraft limit");

}

balance -= amount;

}

}

class CreditCardAccount extends BankAccount {

private double creditLimit;

public CreditCardAccount(int accountNumber, double balance, double creditLimit) {

super(accountNumber, balance);

this.creditLimit = creditLimit;

}

@Override

public void withdraw(double amount) throws Exception {

if (amount > balance + creditLimit) {

throw new Exception("Exceeds credit limit");

}

balance -= amount;

}

}

**B) Implementing the Messaging Application with Interfaces**

We will create an interface Messenger with methods for sending and receiving messages. Classes SMSMessenger and EmailMessenger will implement this interface to provide functionality for sending and receiving SMS and Email messages, respectively.

**CODE:**

interface Messenger {

void sendMessage(String recipient, String message);

String receiveMessage();

}

class SMSMessenger implements Messenger {

private String phoneNumber;

private String receivedMessage;

public SMSMessenger(String phoneNumber) {

this.phoneNumber = phoneNumber;

}

@Override

public void sendMessage(String recipient, String message) {

System.out.println("Sending SMS to " + recipient + ": " + message);

}

@Override

public String receiveMessage() {

receivedMessage = "You have a new SMS message!";

return receivedMessage;

}

}

class EmailMessenger implements Messenger {

private String emailAddress;

private String receivedMessage;

public EmailMessenger(String emailAddress) {

this.emailAddress = emailAddress;

}

@Override

public void sendMessage(String recipient, String message) {

System.out.println("Sending Email to " + recipient + ": " + message);

}

@Override

public String receiveMessage() {

receivedMessage = "You have a new Email!";

return receivedMessage;

}

}

**Explanation:**

* The BankAccount class serves as a base class with common methods like deposit and withdraw.
* Derived classes (SavingsAccount, CheckingAccount, CreditCardAccount) inherit from BankAccount and implement specific features like interest rate, overdraft limit, and credit limit.
* The Messenger interface defines methods for sending and receiving messages. Classes SMSMessenger and EmailMessenger implement this interface, providing concrete behavior for sending and receiving messages via SMS and Email.

**Question 5:**

**Design a real-time gaming system involving different types of characters like warriors, mages, and archers, each with unique abilities and attributes. The system should allow characters to interact in a game environment, engage in combat, and utilize their abilities. Use polymorphism and inheritance to design a flexible and extendable gaming system.**

**Answer:**

**A) Implementing the Gaming System**

We will create a base class Character with attributes like name, health, and strength. Derived classes Warrior, Mage, and Archer will inherit from Character and implement specific abilities like melee attack, magic attack, and ranged attack.

**CODE:**

abstract class Character {

protected String name;

protected int health;

protected int strength;

public Character(String name, int health, int strength) {

this.name = name;

this.health = health;

this.strength = strength;

}

public abstract void attack(Character opponent);

public void takeDamage(int damage) {

health -= damage;

if (health < 0) health = 0;

System.out.println(name + " takes " + damage + " damage, remaining health: " + health);

}

public boolean isAlive() {

return health > 0;

}

@Override

public String toString() {

return "Name: " + name + ", Health: " + health + ", Strength: " + strength;

}

}

class Warrior extends Character {

public Warrior(String name, int health, int strength) {

super(name, health, strength);

}

@Override

public void attack(Character opponent) {

System.out.println(name + " performs a melee attack on " + opponent.name);

opponent.takeDamage(strength);

}

}

class Mage extends Character {

private int magicPower;

public Mage(String name, int health, int strength, int magicPower) {

super(name, health, strength);

this.magicPower = magicPower;

}

@Override

public void attack(Character opponent) {

System.out.println(name + " casts a spell on " + opponent.name);

opponent.takeDamage(magicPower);

}

}

class Archer extends Character {

private int range;

public Archer(String name, int health, int strength, int range) {

super(name, health, strength);

this.range = range;

}

@Override

public void attack(Character opponent) {

System.out.println(name + " shoots an arrow at " + opponent.name);

opponent.takeDamage(strength + range);

}

}

**B) Demonstrating the Gaming System**

We will create characters and simulate a battle, showing how they interact with each other using their unique abilities.

**CODE:**

public class Main {

public static void main(String[] args) {

Character warrior = new Warrior("Conan", 100, 20);

Character mage = new Mage("Gandalf", 80, 10, 30);

Character archer = new Archer("Legolas", 70, 15, 25);

System.out.println("Battle Begins!");

warrior.attack(mage);

mage.attack(archer);

archer.attack(warrior);

System.out.println("\nFinal States:");

System.out.println(warrior);

System.out.println(mage);

System.out.println(archer);

}

}

**Explanation:**

* The Character class is an abstract base class with common attributes like name, health, and strength, and an abstract method attack.
* Derived classes Warrior, Mage, and Archer implement specific attack methods based on their abilities.
* The main program simulates a battle where characters interact and use their abilities, demonstrating polymorphism and inheritance in action.

Question 6

A)

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Input student information

        String name = scanner.nextLine();

        String id = scanner.nextLine();

        double initialGpa = scanner.nextDouble();

        Student student = new Student(name, id, initialGpa);

        student.displayInfo();

        int testCases = scanner.nextInt();

        for (int i = 0; i < testCases; i++) {

            double newGpa = scanner.nextDouble();

            if (student.updateGPA(newGpa)) {

                System.out.println("GPA updated successfully.");

            } else {

                System.out.println("Invalid GPA. Please provide a value between 0 and 4.0.");

            }

            student.displayInfo();

        }

        scanner.close();

    }

}

class Student {

    private String name;

    private String id;

    private double gpa;

    public Student(String name, String id, double gpa) {

        this.name = name;

        this.id = id;

        this.gpa = (gpa >= 0 && gpa <= 4.0) ? gpa : 0;

    }

    public boolean updateGPA(double newGpa) {

        if (newGpa >= 0 && newGpa <= 4.0) {

            this.gpa = newGpa;

            return true;

        }

        return false;

    }

    public void displayInfo() {

        System.out.println("Student Name: " + name);

        System.out.println("Student ID: " + id);

        System.out.println("GPA: " + gpa);

    }

}

B)

import java.util.ArrayList;

import java.util.List;

import java.lang.ref.WeakReference;

public class Main {

    private static List<Object> memoryLeakList = new ArrayList<>();

    private static List<WeakReference<Object>> weakRefList = new ArrayList<>();

    public static void main(String[] args) {

        System.out.println("Simulating memory leak...");

        simulateMemoryLeak();

        System.out.println("Memory leak simulation completed.");

        detectMemoryLeak();

        System.out.println("Number of objects remaining in memoryLeakList: " + memoryLeakList.size());

        suggestSolution();

    }

    private static void simulateMemoryLeak() {

        for (int i = 0; i < 100000; i++) {

            Object obj = new Object();

            memoryLeakList.add(obj);

            weakRefList.add(new WeakReference<>(obj));

            if (i % 10 == 0) {

                memoryLeakList.remove(memoryLeakList.size() - 1);

            }

        }

    }

    private static void detectMemoryLeak() {

        System.gc(); // Request garbage collection

        int leakedObjects = 0;

        for (WeakReference<Object> weakRef : weakRefList) {

            if (weakRef.get() != null) {

                leakedObjects++;

            }

        }

        System.out.println("Detected " + leakedObjects + " leaked objects.");

    }

    private static void suggestSolution() {

        System.out.println("\nSuggested solution to fix the memory leak:");

        System.out.println("1. Clear the memoryLeakList when objects are no longer needed:");

        System.out.println("   memoryLeakList.clear();");

        System.out.println("2. Set individual elements to null if partial clearing is needed:");

        System.out.println("   memoryLeakList.set(index, null);");

        System.out.println("3. Use WeakReference or SoftReference for caching scenarios.");

        System.out.println("4. Implement a custom cache with size limits and eviction policies.");

    }

}

Question 7

A)

public class Main {

    public static void main(String[] args) {

        Mammal lion = new Mammal("Simba", "Lion", "Golden");

        Bird sparrow = new Bird("Chirpy", "Sparrow", 20.5);

        Reptile snake = new Reptile("Slither", "Python", true);

        Animal[] animals = {lion, sparrow, snake};

        for (Animal animal : animals) {

            System.out.println("\nAnimal: " + animal.getName() + " (" + animal.getSpecies() + ")");

            animal.eat();

            animal.makeSound();

            animal.move();

            if (animal instanceof Mammal) {

                Mammal mammal = (Mammal) animal;

                System.out.println("Fur color: " + mammal.getFurColor());

            } else if (animal instanceof Bird) {

                Bird bird = (Bird) animal;

                System.out.println("Wingspan: " + bird.getWingspan() + " cm");

            } else if (animal instanceof Reptile) {

                Reptile reptile = (Reptile) animal;

                System.out.println("Cold-blooded: " + reptile.isColdBlooded());

            }

        }

    }

}

// Abstract base class Animal

abstract class Animal {

    protected String name;

    protected String species;

    public Animal(String name, String species) {

        this.name = name;

        this.species = species;

    }

    public abstract void eat();

    public abstract void makeSound();

    public abstract void move();

    public String getName() {

        return name;

    }

    public String getSpecies() {

        return species;

    }

}

// Mammal subclass

class Mammal extends Animal {

    private String furColor;

    public Mammal(String name, String species, String furColor) {

        super(name, species);

        this.furColor = furColor;

    }

    @Override

    public void eat() {

        System.out.println(name + " is eating meat.");

    }

    @Override

    public void makeSound() {

        System.out.println(name + " is roaring.");

    }

    @Override

    public void move() {

        System.out.println(name + " is running.");

    }

    public String getFurColor() {

        return furColor;

    }

}

// Bird subclass

class Bird extends Animal {

    private double wingspan;

    public Bird(String name, String species, double wingspan) {

        super(name, species);

        this.wingspan = wingspan;

    }

    @Override

    public void eat() {

        System.out.println(name + " is eating seeds.");

    }

    @Override

    public void makeSound() {

        System.out.println(name + " is chirping.");

    }

    @Override

    public void move() {

        System.out.println(name + " is flying.");

    }

    public double getWingspan() {

        return wingspan;

    }

}

// Reptile subclass

class Reptile extends Animal {

    private boolean isColdBlooded;

    public Reptile(String name, String species, boolean isColdBlooded) {

        super(name, species);

        this.isColdBlooded = isColdBlooded;

    }

    @Override

    public void eat() {

        System.out.println(name + " is eating insects.");

    }

    @Override

    public void makeSound() {

        System.out.println(name + " is hissing.");

    }

    @Override

    public void move() {

        System.out.println(name + " is crawling.");

    }

    public boolean isColdBlooded() {

        return isColdBlooded;

    }

}

B)

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Get customer details

        System.out.println("Enter customer name:");

        String name = scanner.nextLine();

        System.out.println("Enter customer email:");

        String email = scanner.nextLine();

        System.out.println("Enter customer address:");

        String address = scanner.nextLine();

        Customer customer = new Customer(name, email, address);

        // Get number of products

        System.out.println("Enter the number of products:");

        int numProducts = scanner.nextInt();

        scanner.nextLine(); // Consume newline

        Order order = new Order(customer);

        // Get product details and add to order

        for (int i = 0; i < numProducts; i++) {

            System.out.println("Enter product details (ID, name, price, quantity):");

            String[] productDetails = scanner.nextLine().split(" ");

            int id = Integer.parseInt(productDetails[0]);

            String productName = productDetails[1];

            double price = Double.parseDouble(productDetails[2]);

            int quantity = Integer.parseInt(productDetails[3]);

            Product product = new Product(id, productName, price, quantity);

            order.addProduct(product);

        }

        // Display order summary

        System.out.println("\nOrder Summary:");

        System.out.println("Customer: " + customer.getName());

        System.out.println("Address: " + customer.getAddress());

        System.out.println("Total Price: $" + order.getTotalPrice());

        scanner.close();

    }

}

class Customer {

    private String name;

    private String email;

    private String address;

    public Customer(String name, String email, String address) {

        this.name = name;

        this.email = email;

        this.address = address;

    }

    public String getName() {

        return name;

    }

    public String getEmail() {

        return email;

    }

    public String getAddress() {

        return address;

    }

}

class Product {

    private int id;

    private String name;

    private double price;

    private int quantity;

    public Product(int id, String name, double price, int quantity) {

        this.id = id;

        this.name = name;

        this.price = price;

        this.quantity = quantity;

    }

    public int getId() {

        return id;

    }

    public String getName() {

        return name;

    }

    public double getPrice() {

        return price;

    }

    public int getQuantity() {

        return quantity;

    }

    public void setQuantity(int quantity) {

        this.quantity = quantity;

    }

}

class Order {

    private static int nextOrderId = 1;

    private int orderId;

    private Customer customer;

    private List<Product> products;

    public Order(Customer customer) {

        this.orderId = nextOrderId++;

        this.customer = customer;

        this.products = new ArrayList<>();

    }

    public void addProduct(Product product) {

        products.add(product);

    }

    public double getTotalPrice() {

        return products.stream()

                .mapToDouble(p -> p.getPrice() \* p.getQuantity())

                .sum();

    }

    public int getOrderId() {

        return orderId;

    }

    public Customer getCustomer() {

        return customer;

    }

    public List<Product> getProducts() {

        return new ArrayList<>(products);

    }

}

Question 8

A)

public class Main {

    public static void main(String[] args) {

        // Create an array of Shape objects

        Shape[] shapes = new Shape[3];

        // Alex creates a rectangle

        shapes[0] = new Rectangle("Alex's Rectangle", 5, 4);

        // Bella creates a circle

        shapes[1] = new Circle("Bella's Circle", 3);

        // They create another shape together

        shapes[2] = new Rectangle("Alex and Bella's Square", 6, 6);

        // Calculate and display areas using polymorphism

        for (Shape shape : shapes) {

            System.out.println(shape.getName() + " - Area: " + shape.calculateArea());

        }

    }

}

// Base class Shape

abstract class Shape {

    private String name;

    public Shape(String name) {

        this.name = name;

    }

    public String getName() {

        return name;

    }

    // Abstract method to be overridden by subclasses

    public abstract double calculateArea();

}

// Rectangle subclass

class Rectangle extends Shape {

    private double length;

    private double width;

    public Rectangle(String name, double length, double width) {

        super(name);

        this.length = length;

        this.width = width;

    }

    @Override

    public double calculateArea() {

        return length \* width;

    }

}

// Circle subclass

class Circle extends Shape {

    private double radius;

    public Circle(String name, double radius) {

        super(name);

        this.radius = radius;

    }

    @Override

    public double calculateArea() {

        return Math.PI \* radius \* radius;

    }

}

B)

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        Calculator calculator = new Calculator();

        System.out.print("Enter the Dividend Number: ");

        double dividend = scanner.nextDouble();

        System.out.print("Enter the Divisor Number: ");

        double divisor = scanner.nextDouble();

        try {

            double result = calculator.divide(dividend, divisor);

            System.out.println("Result: " + result);

        } catch (ArithmeticException e) {

            System.out.println("Error: " + e.getMessage());

        }

        scanner.close();

    }

}

class Calculator {

    public double divide(double dividend, double divisor) throws ArithmeticException {

        if (divisor == 0) {

            throw new ArithmeticException("Division by zero is not allowed.");

        }

        return dividend / divisor;

    }

}

Question 9

A)

import java.util.ArrayList;

import java.util.List;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class Main {

    public static void main(String[] args) {

        String input = "The quick brown fox jumps over the lazy dog";

        AdvancedString advStr = new AdvancedString(input);

        System.out.println("Original: " + advStr.getString());

        // Operation 1: Replace "fox" with "cat"

        advStr.replace("fox", "cat");

        System.out.println("After replace: " + advStr.getString());

        // Operation 2: Extract all words starting with "q" and ending with "k"

        List<String> extracted = advStr.extractPattern("\\bq\\w\*k\\b");

        System.out.println("Words starting with 'q' and ending with 'k': " + extracted);

        // Operation 3: Delete all the words starting with "q"

        advStr.deletePattern("\\bq\\w+");

        System.out.println("After deleting words starting with 'q': " + advStr.getString());

    }

}

class AdvancedString {

    private String str;

    public AdvancedString(String str) {

        this.str = str;

    }

    public String getString() {

        return str;

    }

    public void replace(String target, String replacement) {

        str = str.replace(target, replacement);

    }

    public List<String> extractPattern(String regex) {

        List<String> result = new ArrayList<>();

        Pattern pattern = Pattern.compile(regex);

        Matcher matcher = pattern.matcher(str);

        while (matcher.find()) {

            result.add(matcher.group());

        }

        return result;

    }

    public void deletePattern(String regex) {

        str = str.replaceAll(regex, "").replaceAll("\\s+", " ").trim();

    }

}

B)

**Analysis of Pattern Matching Procedures in Search Engines**

Search engines like Google, Yahoo, and Bing use sophisticated pattern matching procedures to provide relevant results to user queries. Here's an analysis of some key techniques:

1. **Exact String Matching**

   - Used for finding exact matches of query terms in documents.

   - Algorithms: Boyer-Moore, Knuth-Morris-Pratt (KMP)

   - Efficient for short patterns and long texts

2. **Regular Expressions**

   - Allow for more flexible pattern matching

   - Used for handling wildcards, variations in spelling, and complex patterns

   - Implementation often uses Finite Automata

3. **Inverted Index**

   - A data structure that maps words to their locations in documents

   - Enables quick lookup of documents containing specific words

   - Crucial for efficient full-text search

4. **TF-IDF (Term Frequency-Inverse Document Frequency)**

   - Measures the importance of a word in a document relative to a collection

   - Used for ranking search results

   - Helps in identifying relevant documents

5. **N-gram Models**

   - Break text into sequences of N characters or words

   - Used for approximate string matching and handling misspellings

   - Useful for suggestions and "Did you mean?" features

6. **Latent Semantic Analysis (LSA)**

   - Analyzes relationships between documents and terms

   - Helps in understanding context and semantics

   - Useful for handling synonyms and related concepts

7. **PageRank Algorithm**

   - Used by Google to rank web pages

   - Based on the link structure of the web

   - Not strictly a pattern matching procedure, but crucial for result ranking

8. **Machine Learning Models**

   - Used for understanding user intent and context

   - Includes techniques like Neural Networks and Support Vector Machines

   - Helps in handling natural language queries

9. **Phonetic Algorithms**

   - Like Soundex and Metaphone

   - Used for matching words that sound similar

   - Helpful for handling name variations and misspellings

10. **Locality-Sensitive Hashing (LSH)**

    - Used for finding similar items in large datasets

    - Helps in duplicate detection and "related searches" features

11. **Stemming and Lemmatization**

    - Reduce words to their root form

    - Helps in matching variations of words (e.g., "run", "running", "ran")

12. **Query Expansion**

    - Adds synonyms or related terms to the original query

    - Improves recall by finding relevant documents that don't contain exact query terms

These techniques are often combined and optimized in search engines to provide fast, accurate, and relevant results. The exact implementation details are proprietary and continuously evolving.

C)

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the combined string: ");

        String combinedString = scanner.nextLine().toLowerCase();

        System.out.print("Enter the name: ");

        String name = scanner.nextLine().toLowerCase();

        String surname = extractSurname(combinedString, name);

        System.out.println("Name: " + capitalizeFirstLetter(name));

        System.out.println("Surname: " + capitalizeFirstLetter(surname));

        scanner.close();

    }

    public static String extractSurname(String combinedString, String name) {

        int nameIndex = combinedString.indexOf(name);

        String prefix = combinedString.substring(0, nameIndex);

        String suffix = combinedString.substring(nameIndex + name.length());

        return prefix + suffix;

    }

    public static String capitalizeFirstLetter(String str) {

        if (str == null || str.isEmpty()) {

            return str;

        }

        return str.substring(0, 1).toUpperCase() + str.substring(1);

    }

}

Question 10

A)

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

public class Main {

    private static final List<Customer> customers = new ArrayList<>();

    private static final Random random = new Random();

    public static void main(String[] args) {

        while (true) {

            addCustomer();

            printMemoryUsage();

            try {

                Thread.sleep(10);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

    private static void addCustomer() {

        customers.add(new Customer(generateLargeData()));

    }

    private static byte[] generateLargeData() {

        byte[] data = new byte[1024 \* 1024]; // 1MB of data

        random.nextBytes(data);

        return data;

    }

    private static void printMemoryUsage() {

        Runtime runtime = Runtime.getRuntime();

        long usedMemory = runtime.totalMemory() - runtime.freeMemory();

        System.out.println("Used Memory: " + usedMemory / (1024 \* 1024) + " MB");

    }

}

class Customer {

    private byte[] data;

    public Customer(byte[] data) {

        this.data = data;

    }

}

B)

**Analyzing and Addressing Memory Issues with Java Garbage Collection**

**Analysis**

1. The issue stems from continuously adding Customer objects to a List, which prevents them from being garbage collected.

2. Each Customer object holds a large byte array, quickly consuming memory.

3. The ArrayList keeps growing, maintaining references to all Customer objects.

**Addressing the Issue**

1. Use Weak References:

   import java.lang.ref.WeakReference;

   import java.util.ArrayList;

   import java.util.List;

   List<WeakReference<Customer>> customers = new ArrayList<>();

   // When adding:

   customers.add(new WeakReference<>(new Customer(data)));

2. Periodically clean up null references:

   customers.removeIf(ref -> ref.get() == null);

3. Use a custom cache with size limit:

   public class LimitedCache<T> {

       private final int maxSize;

       private final LinkedHashMap<String, T> map;

       public LimitedCache(int maxSize) {

           this.maxSize = maxSize;

           this.map = new LinkedHashMap<String, T>(maxSize, 0.75f, true) {

               protected boolean removeEldestEntry(Map.Entry<String, T> eldest) {

                   return size() > maxSize;

               }

           };

       }

       public void put(String key, T value) {

           map.put(key, value);

       }

       public T get(String key) {

           return map.get(key);

       }

   }

4. Use Java's built-in SoftReference for memory-sensitive caching:

   import java.lang.ref.SoftReference;

   import java.util.HashMap;

   import java.util.Map;

   Map<String, SoftReference<Customer>> customerCache = new HashMap<>();

5. Tune JVM garbage collection:

   - Use `-XX:+UseG1GC` for G1 Garbage Collector

   - Adjust heap size: `-Xms4g -Xmx8g`

   - Set New Generation size: `-XX:NewRatio=3`

6. Implement Object Pooling:

   public class CustomerPool {

       private final Queue<Customer> pool = new ConcurrentLinkedQueue<>();

       private final int maxSize;

       public CustomerPool(int maxSize) {

           this.maxSize = maxSize;

       }

       public Customer acquire() {

           Customer customer = pool.poll();

           return (customer != null) ? customer : new Customer();

       }

       public void release(Customer customer) {

           if (pool.size() < maxSize) {

               pool.offer(customer);

           }

       }

   }

7. Use off-heap memory for large data:

   import java.nio.ByteBuffer;

   class Customer {

       private ByteBuffer data;

       public Customer(int size) {

           this.data = ByteBuffer.allocateDirect(size);

       }

   }

8. Implement proper cleanup in a `close()` method:

   class Customer implements AutoCloseable {

       private byte[] data;

       @Override

       public void close() {

           data = null; // Helps GC

       }

   }

By implementing these techniques, you can significantly improve memory management and reduce the likelihood of memory leaks in your Java application.

C)

**Step-by-Step Approach to Investigate Memory Issues in Java**

**1. Reproduce the Issue**

- Run the program with a fixed set of inputs

- Monitor memory usage over time

- Identify the point where memory usage becomes problematic

**2. Enable Verbose GC Logging**

- Add JVM flags: `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps`

- Analyze GC logs for frequency and duration of collections

**3. Use Memory Profiling Tools**

- Attach VisualVM or Java Mission Control to the running process

- Take heap dumps at different stages of execution

**4. Analyze Heap Dumps**

- Use tools like Eclipse Memory Analyzer (MAT)

- Identify objects consuming most memory

- Look for unexpected object retention

**5. Review Code for Common Issues**

- Check for proper resource closing (e.g., streams, connections)

- Look for static collections that grow unbounded

- Identify long-lived objects that might be candidates for weak references

**6. Implement Logging for Object Lifecycle**

class Customer {

    private static final AtomicInteger counter = new AtomicInteger();

    private final int id;

    public Customer() {

        this.id = counter.incrementAndGet();

        System.out.println("Customer " + id + " created");

    }

    @Override

    protected void finalize() {

        System.out.println("Customer " + id + " finalized");

    }

}

**7. Use JConsole to Monitor MBeans**

- Connect JConsole to the running application

- Monitor memory pools and GC activity in real-time

**8. Implement Memory-Aware Data Structures**

class MemoryAwareList<T> extends ArrayList<T> {

    private final int maxSize;

    public MemoryAwareList(int maxSize) {

        this.maxSize = maxSize;

    }

    @Override

    public boolean add(T t) {

        if (size() >= maxSize) {

            remove(0);

        }

        return super.add(t);

    }

}

**9. Use Weak References for Caching**

Map<String, WeakReference<Customer>> cache = new ConcurrentHashMap<>();

public void addToCache(String key, Customer customer) {

    cache.put(key, new WeakReference<>(customer));

}

public Customer getFromCache(String key) {

    WeakReference<Customer> ref = cache.get(key);

    return (ref != null) ? ref.get() : null;

}

**10. Implement Periodic Cleanup**

ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();

executor.scheduleAtFixedRate(() -> {

    System.gc();

    cache.entrySet().removeIf(entry -> entry.getValue().get() == null);

}, 1, 1, TimeUnit.MINUTES);

**11. Profile Different GC Algorithms**

- Test with different GC algorithms:

  - `-XX:+UseSerialGC`

  - `-XX:+UseParallelGC`

  - `-XX:+UseConcMarkSweepGC`

  - `-XX:+UseG1GC`

- Compare performance and memory usage

**12. Implement Off-Heap Storage for Large Objects**

class OffHeapStorage {

    private final ByteBuffer buffer;

    public OffHeapStorage(int size) {

        this.buffer = ByteBuffer.allocateDirect(size);

    }

    public void write(byte[] data) {

        buffer.put(data);

    }

    public byte[] read(int length) {

        byte[] data = new byte[length];

        buffer.get(data);

        return data;

    }

}

**13. Review Third-Party Libraries**

- Check if any libraries are known for memory leaks

- Update to latest versions if possible

- Consider alternatives if issues persist

**14. Implement Custom Memory Management**

class CustomMemoryManager {

    private final byte[] memoryPool;

    private final BitSet allocatedBlocks;

    public CustomMemoryManager(int size) {

        this.memoryPool = new byte[size];

        this.allocatedBlocks = new BitSet(size);

    }

    public int allocate(int size) {

        // Implementation for memory allocation

    }

    public void free(int offset) {

        // Implementation for memory deallocation

    }

}

By following these steps and implementing the suggested strategies, you can systematically investigate and address the memory issues in your Java application.