Java Programming (4343203) Summer-2024 Solutions

Question 1(a): Explain Garbage collection in java. (Marks: 03)

Garbage Collection is Java's automatic memory management system.

- **Purpose**: Automatically **deletes unused objects** to free memory
- **Working**: JVM identifies objects with **no references** and removes them
- **Benefits**: Prevents **memory leaks** and manual memory management errors

📝 **Remember as "GC-APB"**:

- **G**arbage collector **A**utomatically **P**revents memory leaks by **B**acking up memory

Question 1(b): Explain JVM in detail. (Marks: 04)

JVM (Java Virtual Machine) is the engine that runs Java programs.

Platform independence: Enables Java's "**Write Once, Run Anywhere**" capability
• **Components**:

- **Class Loader**: **Loads** class files into memory
- **Runtime Data Areas**: **Stores** program data during execution
- **Execution Engine**: **Interprets** bytecode into machine code
- **Garbage Collector**: **Removes** unused objects

📝 **Remember as "LERG"**:

- **L**oads classes, **E**xecutes bytecode, **R**uns anywhere, **G**arbage collects

Question 1(c): Write a program in java to print Fibonacci series for N terms. (Marks: 07)

```java
import java.util.Scanner;

public class FibonacciSeries {
    public static void main(String[] args) {
        // Get input from user
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of terms: ");
        int n = sc.nextInt();

        // Initialize first two terms
        int first = 0, second = 1;

        System.out.println("Fibonacci Series for " + n + " terms:");

        // Loop to print series
        for (int i = 1; i <= n; i++) {
            System.out.print(first + " ");

            // Calculate next term
            int next = first + second;
            first = second;
            second = next;
        }
        sc.close();
    }
}
```

Key Steps:
• **Initialize**: Start with **first=0, second=1**
• **Print**: Current **first** value
• **Calculate**: **next = first + second**
• **Update**: **first = second** and **second = next**
• **Repeat**: Until reaching N terms

📝 **Remember as "IPCUR"**:

- **I**nitialize, **P**rint, **C**alculate next, **U**pdate values, **R**epeat

Question 1(c OR): Write a program in java to find out minimum from any ten numbers using command line argument. (Marks: 07)

```
public class FindMinimum {
    public static void main(String[] args) {
        // Check if we have enough arguments
        if (args.length < 10) {
            System.out.println("Please enter 10 numbers");
            return;
        }

        // Convert first argument to integer and assume it's minimum
        int min = Integer.parseInt(args[0]);

        // Check remaining numbers
        for (int i = 1; i < 10; i++) {
            int num = Integer.parseInt(args[i]);
            // Update minimum if current number is smaller
            if (num < min) {
                min = num;
            }
        }

        System.out.println("Minimum number is: " + min);
    }
}
```

Key Steps:

- **Read**: Get numbers from **command line arguments**
- **Initialize**: Set **min = first number**
- **Compare**: Check if each number is **less than min**
- **Update**: If current number is smaller, **min = current number**
- **Output**: Print the smallest number

📝 **Remember as "RICUO"**:

- **R**ead arguments, **I**nitialize minimum, **C**ompare each number, **U**pdate if smaller, **O**utput result

Question 2(a): List out basic concepts of Java OOP. Explain any one in details. (Marks: 03)

Basic OOP Concepts in Java:

- **Encapsulation**: Wrapping data and methods in a **single unit (class)**
- **Inheritance**: Creating new classes that **reuse attributes** of existing classes
- **Polymorphism**: Objects of different classes responding to the **same method name**
- **Abstraction**: Hiding implementation details, showing only **essential features**

📝 **Remember as "EIPA"**:

- **E**ncapsulation, **I**nheritance, **P**olymorphism, **A**bstraction

Question 2(b): Explain final keyword with example. (Marks: 04)

final keyword makes Java elements **unchangeable/non-extendable**.

```java
// Final variable (constant)
final double PI = 3.14159;

// Final method (cannot be overridden)
public final void displayInfo() {
    System.out.println("This method cannot be overridden");
}

// Final class (cannot be extended)
final class SecureClass {
    // Class implementation
}
```

Uses of final:

- **final variable**: Creates **constants** (cannot be reassigned)
- **final method**: Prevents **method overriding** in subclasses
- **final class**: Prevents **class inheritance** (no subclasses allowed)

📝 **Remember as "VCM"**:

- **V**ariables become constants, **C**lasses cannot be extended, **M**ethods cannot be overridden

Question 2(c): What is constructor? Explain parameterized constructor with example. (Marks: 07)

Constructor is a special method that **initializes objects** when created.

```java
public class Student {
    int id;
    String name;

    // Parameterized constructor
    public Student(int studentId, String studentName) {
        id = studentId;         // Initialize id
        name = studentName;     // Initialize name
    }

    public void display() {
        System.out.println("ID: " + id + ", Name: " + name);
    }

    public static void main(String[] args) {
        // Create object using constructor
        Student s1 = new Student(101, "Ravi");
        s1.display();
    }
}
```

Parameterized Constructor:
• **Accepts parameters** when creating objects
• **Initializes object attributes** with provided values
• Has **same name as class** but with parameters
• **No return type**, not even void
• **Automatically called** when object is created using `new`

📝 **Remember as "PAINS"**:

- **P**arameters accepted, **A**ttributes initialized, **I**dentical name as class, **N**o return type, **S**ame-time execution as object creation

Question 2(a OR): Explain the Java Program Structure with example. (Marks: 03)

Java Program Structure:

```java
// 1. Package declaration
package myprogram;

// 2. Import statements
import java.util.Scanner;

// 3. Class declaration
public class HelloWorld {
    // 4. Main method
    public static void main(String[] args) {
        // 5. Program statements
        System.out.println("Hello World!");
    }
}
```

Key Components:

- **Package Declaration**: Organizing related classes
- **Import Statements**: Accessing classes from other packages
- **Class Declaration**: Blueprint for objects
- **Main Method**: Entry point of program
- **Program Statements**: Actual code instructions

📝 **Remember as "PICMS"**:

- **P**ackage, **I**mports, **C**lass, **M**ain method, **S**tatements

Question 2(b OR): Explain static keyword with suitable example. (Marks: 04)

static keyword creates elements that **belong to class** rather than objects.

```java
public class Counter {
    // Static variable (shared by all objects)
    static int count = 0;

    // Constructor
    Counter() {
        count++;  // Increment counter
    }

    // Static method
    static void displayCount() {
        System.out.println("Count: " + count);
    }

    public static void main(String[] args) {
        // Call static method without object
        Counter.displayCount();  // Output: Count: 0

        // Create objects
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        // Call static method again
        Counter.displayCount();  // Output: Count: 2
    }
}
```

Uses of static:
• **static variable**: Single copy **shared** by all objects
• **static method**: Can be called **without creating objects**
• **static block**: Executed when class is **loaded in memory**

📝 **Remember as "COS"**:

- **C**lass-level access, **O**ne copy shared, **S**ame for all objects

Question 2(c OR): Define Inheritance. List out types of it. Explain multilevel and hierarchical inheritance with suitable example. (Marks: 07)

Inheritance is a mechanism where a new class **acquires properties** of an existing class.

Types of Inheritance:
• **Single**: One subclass extends one superclass
• **Multilevel**: Chain of inheritance (A→B→C)
• **Hierarchical**: Multiple subclasses extend one superclass
• **Multiple**: One class extends multiple classes (supported via interfaces)
• **Hybrid**: Combination of inheritance types

Multilevel Inheritance Example:

```java
class Animal {
    void eat() { System.out.println("Eating..."); }
}

class Dog extends Animal {
    void bark() { System.out.println("Barking..."); }
}

class Puppy extends Dog {
    void weep() { System.out.println("Weeping..."); }

    public static void main(String args[]) {
        Puppy p = new Puppy();
        p.eat();   // From Animal
        p.bark();  // From Dog
        p.weep();  // From Puppy
    }
}
```

Hierarchical Inheritance Example:

```java
class Animal {
    void eat() { System.out.println("Eating..."); }
}

class Dog extends Animal {
    void bark() { System.out.println("Barking..."); }
}
```

```
class Cat extends Animal {
    void meow() { System.out.println("Meowing..."); }

    public static void main(String args[]) {
        Cat c = new Cat();
        c.eat();  // From Animal
        c.meow(); // From Cat

        Dog d = new Dog();
        d.eat();  // From Animal
        d.bark(); // From Dog
    }
}
```

📝 **Remember inheritance types as "SMHMH"**:

- **S**ingle, **M**ultilevel, **H**ierarchical, **M**ultiple, **H**ybrid

Question 3(a): Explain this keyword with suitable example. (Marks: 03)

this keyword refers to the **current object** in a method or constructor.

```
public class Student {
    int rollNo;
    String name;

    // Constructor with parameters
    Student(int rollNo, String name) {
        this.rollNo = rollNo;  // this refers to current object
        this.name = name;
    }

    void display() {
        System.out.println(rollNo + " " + name);
    }

    public static void main(String args[]) {
        Student s1 = new Student(111, "Karan");
        s1.display();
    }
}
```

Uses of this:
• **Differentiates** between instance variables and parameters
• **Invokes** current class methods/constructors
• **Returns** the current object

📝 **Remember as "DIR"**:

- **D**ifferentiates variables, **I**nvokes methods, **R**eturns current object

Question 3(b): Explain different access controls in Java. (Marks: 04)

Access Modifiers control visibility of classes, methods, and variables.

Modifier	Class	Package	Subclass	World
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

Access Levels:
• **private**: Accessible only **within class**
• **default**: Accessible within **same package**
• **protected**: Accessible within **package and subclasses**
• **public**: Accessible from **anywhere**

📝 **Remember with "PriDefProPub"**:

- **Pri**vate (class), **Def**ault (package), **Pro**tected (package+subclass), **Pub**lic (everywhere)

Question 3(c): What is interface? Explain multiple inheritance using interface with example. (Marks: 07)

Interface is a **contract** containing abstract methods and constants that classes must implement.

```java
// Define interfaces
interface Printable {
    void print();
}

interface Showable {
    void show();
}

// Implement multiple interfaces
class Magazine implements Printable, Showable {
    // Implement all methods from both interfaces
    public void print() {
        System.out.println("Printing magazine...");
    }

    public void show() {
        System.out.println("Showing magazine...");
    }

    public static void main(String args[]) {
        Magazine m = new Magazine();
        m.print();
        m.show();
    }
}
```

Interface Features:
• Enables **multiple inheritance** in Java
• Contains **abstract methods** (no implementation)
• Methods are implicitly **public abstract**
• Variables are implicitly **public static final**
• Classes **implement** interfaces (not extend)

📝 **Remember as "MAPLE"**:

- **M**ultiple inheritance, **A**bstract methods only, **P**ublic by default, **L**ike a contract, **E**asy implementation

Question 3(a OR): Explain super keyword with example. (Marks: 03)

super keyword refers to the **parent class** objects/methods.

```java
class Animal {
    String color = "white";

    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    String color = "black";

    void printColor() {
        System.out.println(color);        // prints black
        System.out.println(super.color);  // prints white
    }

    void eat() {
        super.eat();  // calls parent class method
        System.out.println("Eating bread...");
    }
}
```

Uses of super:
• Access **parent class variables**
• Call **parent class methods**
• Call **parent class constructor**

📝 **Remember as "VMC"**:

- Access **V**ariables, **M**ethods, and **C**onstructors of parent class

Question 3(b OR): What is package? Write steps to create a package and give example of it. (Marks: 04)

Package is a **namespace** that organizes related classes and interfaces.

Steps to Create Package:

1. **Declare** package at top of source file
2. **Compile** with javac -d option
3. **Import** package to use it in other classes

```
// Step 1: Declare package (save as Calculator.java)
package mathutils;

public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}

// Step 2: Compile with javac -d . Calculator.java

// Step 3: Use the package in another class
import mathutils.Calculator;

class TestCalculator {
    public static void main(String args[]) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20));
    }
}
```

Package Benefits:
• **Organizes** related classes
• Prevents **naming conflicts**
• Provides **access control**

📝 **Remember package creation as "DCI"**:

- **D**eclare package, **C**ompile with -d, **I**mport to use

Question 3(c OR): Define: Method Overriding. List out Rules for method overriding. Write a java program that implements method overriding. (Marks: 07)

Method Overriding is redefining a method in subclass that is already defined in parent class.

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
```

```java
}

class Dog extends Animal {
    // Overridden method
    @Override
    void makeSound() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        myAnimal.makeSound();  // Output: Animal makes a sound

        Dog myDog = new Dog();
        myDog.makeSound();     // Output: Dog barks

        // Polymorphism
        Animal animal = new Dog();
        animal.makeSound();    // Output: Dog barks
    }
}
```

Rules for Method Overriding:

- Method must have **same name** as parent class
- Method must have **same parameters** as parent class
- Must be **IS-A relationship** (inheritance)
- Access modifier should be **same or more permissive**
- Return type must be **same or covariant**
- Cannot override **final** or **static** methods

📝 **Remember rules as "SPIARS"**:

- **S**ame name, **P**arameters matching, **I**nheritance needed, **A**ccess same/wider, **R**eturn type same/subclass, **S**tatic/final can't be overridden

Question 4(a): Explain abstract class with suitable example. (Marks: 03)

Abstract Class is a **restricted class** that cannot be instantiated directly.

```java
abstract class Shape {
    // Abstract method (no body)
    abstract void draw();

    // Concrete method
    void resize() {
        System.out.println("Resizing shape");
    }
}

class Circle extends Shape {
    // Implementing abstract method
    void draw() {
        System.out.println("Drawing circle");
    }

    public static void main(String[] args) {
        // Shape s = new Shape(); // Error: Cannot instantiate
        Circle c = new Circle();
        c.draw();
        c.resize();
    }
}
```

Abstract Class Features:
• Cannot be **instantiated** directly
• Can have **abstract methods** (no body)
• Can have **concrete methods** (with body)
• Subclasses must **implement abstract methods**

📝 **Remember as "NACI"**:

- **N**o instantiation, **A**bstract methods allowed, **C**oncrete methods allowed, **I**mplementation required

Question 4(b): What is Thread? Explain Thread life cycle. (Marks: 04)

Thread is a **lightweight process** that enables concurrent execution.

Thread Life Cycle States:
• **New**: Thread created but not started
• **Runnable**: Thread ready to run
• **Running**: Currently executing
• **Blocked/Waiting**: Temporarily inactive
• **Terminated**: Completed execution

Transitions:
• **start()**: New → Runnable
• **run()**: Runnable → Running
• **sleep()/wait()**: Running → Waiting
• **notify()**: Waiting → Runnable
• **run() completes**: Running → Terminated

📝 **Remember as "NRWBT"**:

- **N**ew, **R**unnable, **R**unning, **W**aiting/Blocked, **T**erminated

Question 4(c): Write a program in java that creates the multiple threads by implementing the Thread class. (Marks: 07)

```
class MyThread extends Thread {
    private String threadName;
```

```java
    // Constructor
    public MyThread(String name) {
        this.threadName = name;
    }

    // Override run method
    @Override
    public void run() {
        try {
            for (int i = 1; i <= 3; i++) {
                System.out.println(threadName + ": Count " + i);
                Thread.sleep(1000); // Pause for 1 second
            }
        } catch (InterruptedException e) {
            System.out.println(threadName + " interrupted.");
        }
        System.out.println(threadName + " finished.");
    }
}

public class MultiThreadDemo {
    public static void main(String[] args) {
        // Create multiple threads
        MyThread thread1 = new MyThread("Thread-1");
        MyThread thread2 = new MyThread("Thread-2");

        // Start threads
        thread1.start();
        thread2.start();

        System.out.println("Main thread finished.");
    }
}
```

Creating Multiple Threads:
• **Extend Thread class**: Create a subclass that extends Thread
• **Override run() method**: Define what thread will do
• **Create thread objects**: Instantiate your thread subclass
• **Call start() method**: Begin thread execution

📝 **Remember as "ECOS"**:

- **E**xtend Thread, **C**reate the run method, **O**bject creation, **S**tart the thread

Question 4(a OR): Explain final class with suitable example. (Marks: 03)

final class is a class that **cannot be extended** (no subclasses allowed).

```java
final class FinalClass {
    void display() {
        System.out.println("This is a final class");
    }
}

// Error: Cannot extend final class
// class ChildClass extends FinalClass {
//     ...
// }

class FinalClassDemo {
    public static void main(String[] args) {
        FinalClass fc = new FinalClass();
        fc.display();
    }
}
```

Benefits of final class:
• **Security**: Prevents modification of sensitive classes
• **Immutability**: Ensures class behavior isn't altered
• **Optimization**: Compiler can optimize final classes

📝 **Remember as "SIO"**:

- **S**ecurity enhancement, **I**mmutability guarantee, **O**ptimization friendly

Question 4(b OR): Explain thread priorities with suitable example. (Marks: 04)

Thread Priority determines the **importance** of a thread's execution.

```
class PriorityThread extends Thread {
    PriorityThread(String name) {
        super(name);
    }

    public void run() {
        System.out.println("Running thread: " +
                           getName() +
                           ", Priority: " +
                           getPriority());
    }
}

public class ThreadPriorityDemo {
    public static void main(String[] args) {
        // Create threads
        PriorityThread t1 = new PriorityThread("Low Priority");
        PriorityThread t2 = new PriorityThread("Normal Priority");
        PriorityThread t3 = new PriorityThread("High Priority");

        // Set priorities
        t1.setPriority(Thread.MIN_PRIORITY);     // 1
        // t2 uses default priority                // 5
        t3.setPriority(Thread.MAX_PRIORITY);     // 10

        // Start threads
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Priority Details:
• Range from **1 (MIN_PRIORITY) to 10 (MAX_PRIORITY)**
• Default is **5 (NORM_PRIORITY)**
• Higher priority thread **gets preference** in execution
• Priority is only a **hint to scheduler**, not guaranteed

📝 **Remember as "RPH"**:

- **R**ange 1-10, **P**reference for higher values, **H**int for scheduler

Question 4(c OR): What is Exception? Write a program that shows the use of Arithmetic Exception. (Marks: 07)

Exception is an **abnormal condition** that disrupts the normal flow of program.

```
public class ArithmeticExceptionDemo {
    public static void main(String[] args) {
        try {
            // Code that may cause exception
            int a = 30, b = 0;
            System.out.println("Trying to divide: " + a + "/" + b);

            // This will throw ArithmeticException
            int result = a / b;

            // This won't execute if exception occurs
            System.out.println("Result: " + result);

        } catch (ArithmeticException e) {
            // Exception handler
            System.out.println("Exception caught: " + e.getMessage());
            System.out.println("Cannot divide by zero!");

        } finally {
            // Always executes
            System.out.println("Finally block executed");
        }

        System.out.println("Rest of the code continues...");
    }
}
```

Exception Handling Elements:

- **try**: Contains code that might throw exception
- **catch**: Handles the exception
- **finally**: Executes regardless of exception
- **throw**: Manually throws an exception
- **throws**: Declares exceptions method might throw

📝 **Remember as "TCFTTS"**:

- **T**ry risky code, **C**atch problems, **F**inally clean up, **T**hrow when needed, **T**hrows to declare, **S**afe execution

Question 5(a): Write a Java Program to find sum and average of 10 numbers of an array. (Marks: 03)

```java
public class ArraySumAverage {
    public static void main(String[] args) {
        // Initialize array
        int[] numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

        // Variables for sum and average
        int sum = 0;
        double average;

        // Calculate sum
        for (int i = 0; i < numbers.length; i++) {
            sum += numbers[i];
        }

        // Calculate average
        average = (double) sum / numbers.length;

        // Print results
        System.out.println("Sum = " + sum);
        System.out.println("Average = " + average);
    }
}
```

Steps:

- **Initialize array** with 10 values
- **Sum** all elements using loop
- **Average** = sum / number of elements
- **Display** results

📝 **Remember as "ISAD"**:

- **I**nitialize array, **S**um elements, **A**verage calculation, **D**isplay results

Question 5(b): Write a Java program to handle user defined exception for 'Divide by Zero' error. (Marks: 04)

```java
// Custom exception class
```

```
class DivideByZeroException extends Exception {
    public DivideByZeroException(String message) {
        super(message);
    }
}

public class CustomExceptionDemo {
    // Method that may throw custom exception
    static double divide(int a, int b) throws DivideByZeroException {
        if (b == 0) {
            throw new DivideByZeroException("Cannot divide by zero!");
        }
        return (double) a / b;
    }

    public static void main(String[] args) {
        try {
            System.out.println(divide(10, 2));  // Works fine
            System.out.println(divide(20, 0));  // Throws exception
        } catch (DivideByZeroException e) {
            System.out.println("Custom Exception: " + e.getMessage());
        }
    }
}
```

Creating Custom Exception:
• **Extend** Exception class
• Create **constructor** with message
• **throw** the exception when condition occurs
• **catch** the exception to handle it

📝 **Remember as "ETCW"**:

- **E**xtend Exception, make a **T**hrowable message, **C**reate and throw, **W**rite handler

Question 5(c): Write a java program to create a text file and perform read operation on the text file. (Marks: 07)

```
import java.io.File;
import java.io.FileWriter;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class FileReadWriteDemo {
    public static void main(String[] args) {
        try {
            // Create file
            File file = new File("sample.txt");
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            }
            // Write to file
            FileWriter writer = new FileWriter(file);
            writer.write("Hello World!\nThis is a sample text file.\nJava I/O is easy.");
            writer.close();
            System.out.println("Successfully wrote to the file.");

            // Read from file
            System.out.println("\nFile contents:");
            FileReader reader = new FileReader(file);
            BufferedReader buffReader = new BufferedReader(reader);

            String line;
            while ((line = buffReader.readLine()) != null) {
                System.out.println(line);
            }

            // Close reader
            buffReader.close();

        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

File Operations:
• **Create** file with File class
• **Write** content using FileWriter
• **Read** content using FileReader and BufferedReader
• **Close** resources after use
• **Handle** exceptions with try-catch

📝 **Remember as "CWRCH"**:

- **C**reate file, **W**rite content, **R**ead content, **Cl**ose resources, **H**andle exceptions

Question 5(a OR): Explain java I/O process. (Marks: 03)

Java I/O (Input/Output) provides classes for **reading and writing data**.

I/O Streams:
• **Byte Streams**: Handle binary data (FileInputStream, FileOutputStream)
• **Character Streams**: Handle text data (FileReader, FileWriter)
• **Buffered Streams**: Improve performance (BufferedReader, BufferedWriter)

Process Flow:
• **Open** stream → **Process** data → **Close** stream

📝 **Remember as "BCOP"**:

- **B**yte or character streams, **C**onnect to source/destination, **O**perate on data, **P**roperly close

Question 5(b OR): Explain throw and finally in Exception Handling with example. (Marks: 04)

throw explicitly throws an exception. **finally** ensures code always executes.

```
public class ThrowFinallyDemo {
    // Method that uses throw
    static void validateAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        } else {
            System.out.println("Welcome to vote!");
        }
    }

    public static void main(String[] args) {
        try {
            // Code that might throw exception
            validateAge(15);  // This will throw exception

        } catch (ArithmeticException e) {
            System.out.println("Exception: " + e.getMessage());

        } finally {
            // This will always execute
            System.out.println("Finally block executed");
        }

        System.out.println("Rest of the code...");
    }
}
```

Key Points:
• **throw**: Manually **throws specified exception** based on conditions
• **finally**: Block that **always executes** regardless of exception
• Use **throw** for custom validation logic
• Use **finally** for cleanup operations (closing files, connections)

📝 **Remember as "TFC"**:

- **T**hrow exceptions manually, **F**inally always runs, **C**leanup resources

Question 5(b OR): Explain throw and finally in Exception Handling with example. (Marks: 04)

throw explicitly throws an exception. **finally** ensures code always executes.

```
public class ThrowFinallyDemo {
    // Method that uses throw
    static void validateAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        } else {
            System.out.println("Welcome to vote!");
        }
    }

    public static void main(String[] args) {
        try {
            // Code that might throw exception
            validateAge(15);  // This will throw exception

        } catch (ArithmeticException e) {
            System.out.println("Exception: " + e.getMessage());

        } finally {
            // This will always execute
            System.out.println("Finally block executed");
        }

        System.out.println("Rest of the code...");
    }
}
```

Key Points:
• **throw**: Manually **throws specified exception** based on conditions
• **finally**: Block that **always executes** regardless of exception
• Use **throw** for custom validation logic
• Use **finally** for cleanup operations (closing files, connections)

📝 **Remember as "TFC"**:

- **T**hrow exceptions manually, **F**inally always runs, **C**leanup resources

Question 5(c OR): Write a java program to display the content of a text file and perform append operation on the text file. (Marks: 07)

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.IOException;

public class FileAppendDemo {
    public static void main(String[] args) {
        try {
            // Create file if it doesn't exist
            File file = new File("data.txt");
            if (!file.exists()) {
                // Create and write initial content
                FileWriter writer = new FileWriter(file);
                writer.write("Original content\n");
                writer.close();
                System.out.println("File created with initial content");
            }

            // Display current content
            System.out.println("Current file content:");
            FileReader reader = new FileReader(file);
            BufferedReader buffReader = new BufferedReader(reader);

            String line;
            while ((line = buffReader.readLine()) != null) {
                System.out.println(line);
            }
            buffReader.close();

            // Append new content (true flag enables append mode)
            FileWriter appendWriter = new FileWriter(file, true);
            appendWriter.write("This content is appended\n");
            appendWriter.close();
            System.out.println("\nContent appended successfully");

            // Display updated content
            System.out.println("\nUpdated file content:");
            reader = new FileReader(file);
            buffReader = new BufferedReader(reader);

            while ((line = buffReader.readLine()) != null) {
                System.out.println(line);
            }
            buffReader.close();

        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Append Operation Steps:
• **Display** current content first
• **Open** file in append mode (FileWriter with true parameter)
• **Write** new content at the end
• **Close** resources
• **Display** updated content

📝 **Remember as "DOWCD"**:

- **D**isplay original, **O**pen in append mode, **W**rite new content, **C**lose writer, **D**isplay updated content