Java Programming (4343203) Winter-2024 Solutions

Question 1(a): List out various Primitive data types in Java. (Marks: 03)

Java Primitive Data Types:

• **byte**: 8-bit integer (-128 to 127)
• **short**: 16-bit integer (-32,768 to 32,767)
• **int**: 32-bit integer (-2^31 to 2^31-1)
• **long**: 64-bit integer (-2^63 to 2^63-1)
• **float**: 32-bit floating-point number
• **double**: 64-bit floating-point number
• **char**: 16-bit Unicode character
• **boolean**: true or false value

📝 **Remember as "CHILDREN"**:

- **C**har, **H**ealthy boolean, **I**nteger, **L**ong, **D**ouble, **R**obust float, **E**fficient byte, **N**ifty short

Question 1(b): Explain Structure of Java Program with suitable example. (Marks: 04)

Java Program Structure:

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

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

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

    // 6. Additional methods
    public void displayMessage() {
        System.out.println("Welcome!");
    }
}
```

Key Components:
• **Package Declaration**: Groups related classes
• **Import Statements**: Accesses external classes
• **Class Declaration**: Blueprint for objects
• **Main Method**: Program execution starts here
• **Variables/Statements**: Program logic
• **Additional Methods**: Reusable code blocks

📝 **Remember as "PICMVA"**:

- **P**ackage, **I**mports, **C**lass, **M**ain method, **V**ariables/statements, **A**dditional methods

Question 1(c): List arithmetic operators in Java. Develop a Java program using any three arithmetic operators and show the output of program. (Marks: 07)

Arithmetic Operators in Java:
• **+** : Addition
• **-** : Subtraction
• ***** : Multiplication
• **/** : Division
• **%** : Modulus (remainder)
• **++** : Increment
• **--** : Decrement

```
public class ArithmeticDemo {
    public static void main(String[] args) {
        // Variable declaration and initialization
        int a = 10, b = 3;

        // Using three arithmetic operators
        int sum = a + b;       // Addition
        int product = a * b;   // Multiplication
        int remainder = a % b;  // Modulus

        // Display results
        System.out.println("Values: a = " + a + ", b = " + b);
        System.out.println("Sum: " + a + " + " + b + " = " + sum);
        System.out.println("Product: " + a + " * " + b + " = " + product);
        System.out.println("Remainder: " + a + " % " + b + " = " + remainder);
    }
}
```

Output:

```
Values: a = 10, b = 3
Sum: 10 + 3 = 13
Product: 10 * 3 = 30
Remainder: 10 % 3 = 1
```

📝 **Remember arithmetic operators as "ASMDIP"**:

- **A**ddition, **S**ubtraction, **M**ultiplication, **D**ivision, **I**ncrement, **P**ercentage (modulus)

Question 1(c OR): Write syntax of Java for loop statement. Develop a Java program to find out prime number between 1 to 10. (Marks: 07)

Java for loop syntax:

```
for (initialization; condition; update) {
    // code to be executed
}
```

Program to find prime numbers between 1 to 10:

```
public class PrimeNumbers {
    public static void main(String[] args) {
        System.out.println("Prime numbers between 1 and 10:");

        // Loop through numbers 1 to 10
        for (int i = 1; i <= 10; i++) {
            boolean isPrime = true;

            // Check if number is prime
            if (i == 1) {
                isPrime = false;  // 1 is not prime
            } else {
                // Check for factors
                for (int j = 2; j < i; j++) {
                    if (i % j == 0) {
                        isPrime = false;
                        break;
                    }
                }
            }

            // Print if prime
            if (isPrime) {
                System.out.print(i + " ");
            }
        }
    }
}
```

Output:

```
Prime numbers between 1 and 10:
2 3 5 7
```

Key Steps:
• **Iterate** through numbers 1 to 10
• For each number, **assume** it's prime initially
• Handle **special case** for 1 (not prime)
• **Check** if divisible by any number from 2 to (n-1)
• If divisible, number is **not prime**
• **Print** all prime numbers

📝 **Remember algorithm as "IACHP"**:

- **I**terate numbers, **A**ssume prime, **C**heck divisors, **H**andle special cases, **P**rint results

Question 2(a): List the differences between Procedure-Oriented Programming (POP) and Object-Oriented Programming (OOP). (Marks: 03)

POP vs OOP:

Procedure-Oriented Programming	Object-Oriented Programming
Focuses on procedures/functions	**Focuses** on objects
Divides program into functions	**Divides** program into objects
Data and functions are separate	**Encapsulates** data and functions together
Less secure (global data)	**More** secure (data hiding)
Examples: C, FORTRAN	**Examples**: Java, C++, Python

📝 **Remember as "FDSEM"**:

- **F**ocus (procedures vs objects), **D**ivision (functions vs objects), **S**tructure (separate vs combined), **E**ncapsulation (none vs present), **M**odels (procedural vs object)

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

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

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

    // Constructor
    Counter() {
        count++;  // Increments the shared counter
    }

    // Static method
    static void displayCount() {
        System.out.println("Total objects created: " + count);
```

```
    }

    public static void main(String[] args) {
        // Create objects
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        // Call static method (without object)
        Counter.displayCount();  // Output: Total objects created: 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 static uses as "SCM"**:

- **S**hared across objects, **C**lass-level access, **M**ethod calling without instances

Question 2(c): Define Constructor. List types of Constructors. Develop a java code to explain Parameterized constructor. (Marks: 07)

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

Types of Constructors:
• **Default Constructor**: No parameters
• **Parameterized Constructor**: Takes parameters
• **Copy Constructor**: Creates object from another object

```
public class Student {
    // Instance variables
    int id;
    String name;

    // Default constructor
    Student() {
        id = 0;
        name = "Unknown";
        System.out.println("Default constructor called");
    }

    // Parameterized constructor
    Student(int studentId, String studentName) {
        id = studentId;
        name = studentName;
        System.out.println("Parameterized constructor called");
    }

    // Display method
```

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

    public static void main(String[] args) {
        // Creating objects using constructors
        Student s1 = new Student();             // Default constructor
        Student s2 = new Student(101, "Alex");  // Parameterized constructor

        // Display object data
        s1.display();  // Output: ID: 0, Name: Unknown
        s2.display();  // Output: ID: 101, Name: Alex
    }
}
```

Parameterized Constructor Features:
• **Accepts parameters** during object creation
• **Initializes object** with provided values
• Has **same name as class**
• Has **no return type**
• **Called automatically** when object is created

📝 **Remember constructor types as "DPC"**:

- **D**efault (no params), **P**arameterized (with params), **C**opy (clone objects)

Question 2(a OR): List the basic OOP concepts in Java and explain any one. (Marks: 03)

Basic OOP Concepts in Java:

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

Encapsulation Example:

```
class Person {
    // Private variables (hidden)
    private String name;
    private int age;

    // Public methods (accessible)
    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
}
```

📝 **Remember as "EIPA"**:

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

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

final keyword makes elements **unchangeable or non-extendable**.

```
// Final variable (constant)
public class FinalDemo {
    final double PI = 3.14159;  // Cannot be changed

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

// Final class (cannot be extended)
final class SecureClass {
    void show() {
        System.out.println("This class cannot be extended");
    }
}

// This would cause error: Can't extend final class
// class ChildClass extends SecureClass {}
```

Uses of final:
• **final variable**: Creates **constants** (cannot be changed)
• **final method**: Prevents **method overriding** in subclasses
• **final class**: Prevents **class inheritance** (no subclasses)

📝 **Remember as "CMV"**:

- **C**onstant variables, **M**ethod cannot be overridden, **V**eto inheritance

Question 2(c OR): Write scope of java access modifier. Develop a java code to explain public modifier. (Marks: 07)

Access Modifiers Scope in Java:

Modifier	Class	Package	Subclass	World
private	✓	✗	✗	✗
default (no modifier)	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

```
// File: PublicDemo.java
package demo;
```

```java
// Public class - accessible from anywhere
public class PublicDemo {
    // Public variable
    public String message = "Hello World";

    // Public method
    public void display() {
        System.out.println(message);
    }
}

// File: MainApp.java
package application;

// Importing from another package
import demo.PublicDemo;

public class MainApp {
    public static void main(String[] args) {
        // Creating object of class from different package
        PublicDemo obj = new PublicDemo();

        // Accessing public member variable
        System.out.println("Message: " + obj.message);

        // Calling public method
        obj.display();
    }
}
```

Public Modifier Features:
• Accessible from **any class**
• Accessible from **any package**
• No **access restrictions**
• Used for classes, methods, and variables meant to be **widely accessible**
• Classes with **public interfaces** use public methods

📝 **Remember access scopes as "CDPS"**:

- **C**lass-only (private), **D**efault package, **P**rotected inheritance, **S**ystem-wide (public)

Question 3(a): List out different types of inheritance and explain any one with example. (Marks: 03)

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

Single Inheritance Example:

```
// Parent class
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

// Child class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking");
    }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();  // Inherited method
        d.bark(); // Own method
    }
}
```

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

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

Question 3(b): Explain any two String buffer class methods with suitable example. (Marks: 04)

StringBuffer Class Methods:

```
public class StringBufferDemo {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");

        // 1. append() - adds content at the end
        sb.append(" World");
        System.out.println("After append: " + sb);  // Hello World

        // 2. insert() - inserts content at specified position
        sb.insert(5, " Beautiful");
        System.out.println("After insert: " + sb);  // Hello Beautiful World

        // Other common methods:
        // 3. replace() - replaces text
        // 4. delete() - removes characters
        // 5. reverse() - reverses the content
    }
}
```

Key StringBuffer Methods:
• **append(String s)**: Adds text at the **end** of buffer
• **insert(int offset, String s)**: Inserts text at **specified position**
• **replace(int start, int end, String s)**: **Replaces** text from start to end
• **delete(int start, int end)**: **Removes** characters from start to end
• **reverse()**: **Reverses** all characters

📝 **Remember as "AIRDR"**:

- **A**ppend text, **I**nsert at position, **R**eplace content, **D**elete portions, **R**everse all

Question 3(c): Define Interface. Write a java program to demonstrate multiple inheritance using interface. (Marks: 07)

Interface is a collection of **abstract methods** that classes can implement.

```
// First interface
interface Printable {
    void print();  // Abstract method
}

// Second interface
interface Showable {
    void show();   // Abstract method
}

// Class implementing multiple interfaces
class Document implements Printable, Showable {
    // Implementing all methods from both interfaces
    public void print() {
        System.out.println("Document printing");
    }

    public void show() {
        System.out.println("Document showing");
    }

    public static void main(String[] args) {
        Document doc = new Document();

        // Calling implemented methods
        doc.print();
        doc.show();

        // Using interface reference
        Printable p = new Document();
        p.print();

        Showable s = new Document();
        s.show();
    }
```

```
}
```

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

📝 **Remember interface features as "PAVIA"**:

- **P**ublic methods, **A**bstract only, **V**ariables are constants, **I**mplementation required, **A**llows multiple inheritance

Question 3(a OR): Give differences between Abstract class and Interface. (Marks: 03)

Abstract Class vs Interface:

Abstract Class	Interface
Uses **abstract** keyword	Uses **interface** keyword
Can have **both** abstract and concrete methods	Contains **only** abstract methods (before Java 8)
Can have **constructors**	**Cannot** have constructors
Supports **variables** of any type	Variables are **public static final** only
Supports **single inheritance** only	Enables **multiple inheritance**
Can have **access modifiers**	Methods are **implicitly public**

📝 **Remember as "KCVIAM"**:

- **K**eyword differs, **C**oncrete methods allowed/disallowed, **V**ariable restrictions, **I**nheritance type, **A**ccess modifiers, **M**ethod implementation

Question 3(b OR): Explain any two String class methods with suitable example. (Marks: 04)

String Class Methods:

```java
public class StringMethodsDemo {
    public static void main(String[] args) {
        String str = "Hello Java Programming";

        // 1. length() - returns string length
        int len = str.length();
        System.out.println("Length: " + len);  // 23

        // 2. substring() - extracts part of string
```

```
        String sub = str.substring(6, 10);
        System.out.println("Substring: " + sub);  // Java

        // Other common methods:
        // 3. indexOf() - finds position of text
        // 4. equals() - compares strings
        // 5. replace() - replaces text
    }
}
```

Key String Methods:
• **length()**: Returns the **number of characters**
• **substring(int start, int end)**: Returns **portion** of string
• **indexOf(String str)**: Returns **position** of first occurrence
• **equals(Object obj)**: **Compares** string content
• **replace(char old, char new)**: **Replaces** characters

📝 **Remember as "LASER"**:

- **L**ength counting, **A**cquire substring, **S**earch with indexOf, **E**quals comparison, **R**eplace text

Question 3(c OR): Explain package and list out steps to create package with suitable example. (Marks: 07)

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 -d option
3. **Import** to use in other classes

```
// Step 1: Declare package (Calculator.java)
package mathutil;

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

    public int subtract(int a, int b) {
        return a - b;
    }
}

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

// Step 3: Use the package in another file (TestCalculator.java)
import mathutil.Calculator;

public class TestCalculator {
    public static void main(String[] args) {
```

```
        // Creating object from package
        Calculator calc = new Calculator();

        // Using methods
        System.out.println("5 + 3 = " + calc.add(5, 3));
        System.out.println("5 - 3 = " + calc.subtract(5, 3));
    }
}
```

Package Benefits:
• **Organizes** classes by functionality
• Prevents **naming conflicts**
• Provides **access control**
• Enables **data encapsulation**
• Makes locating classes **easier**

📝 **Remember package creation steps as "DCIE"**:

- **D**eclare package name, **C**ompile with -d, **I**mport in other classes, **E**xecute your program

Question 4(a): List types of errors in Java. (Marks: 03)

Types of Errors in Java:

• **Syntax Errors**: Grammar mistakes in code detected during **compilation**
• **Runtime Errors**: Problems that occur during **program execution**
• **Logical Errors**: Code runs but produces **incorrect results**
• **Compilation Errors**: Errors found by **compiler** (e.g., missing semicolons)
• **LinkageErrors**: Problems in **linking** classes at runtime

Examples:

```
// Syntax Error
if (x > 5) // Missing curly braces

// Runtime Error
int[] arr = new int[5];
arr[10] = 50;  // ArrayIndexOutOfBoundsException

// Logical Error
int sum = a - b;  // Should be addition but using subtraction
```

📝 **Remember as "SRCLL"**:

- **S**yntax, **R**untime, **C**ompilation, **L**ogical, **L**inkage errors

Question 4(b): Explain try catch block with example. (Marks: 04)

try-catch blocks handle exceptions in Java programs.

```
public class TryCatchDemo {
```

```
    public static void main(String[] args) {
        try {
            // Code that might cause exception
            int result = 10 / 0;  // ArithmeticException
            System.out.println("Result: " + result); // Won't execute
        }
        catch (ArithmeticException e) {
            // Exception handler
            System.out.println("Error: " + e.getMessage());

            // Optional: print stack trace
            // e.printStackTrace();
        }
        finally {
            // Always executes
            System.out.println("This always executes");
        }

        System.out.println("Program continues...");
    }
}
```

try-catch Components:
• **try block**: Contains code that might **throw exceptions**
• **catch block**: Handles the **specific exception** types
• **finally block**: Code that **always executes** (optional)
• Multiple catch blocks can handle **different exceptions**
• Without try-catch, exceptions cause **program termination**

📝 **Remember as "TECH"**:

- **T**ry risky code, **E**xception is caught, **C**atch handles problems, **H**ealing continues after

Question 4(c): List out any four differences between method overloading and overriding. Write a java code to explain method overriding. (Marks: 07)

Method Overloading vs Method Overriding:

Method Overloading	**Method Overriding**
Same class	**Parent-child** classes
Different parameters	**Same parameters**
Compile-time polymorphism	**Runtime** polymorphism
Return type can be **different**	Return type must be **same or covariant**
Static methods can be overloaded	**Static** methods cannot be overridden
Access modifiers can vary	**Cannot restrict** accessibility

```java
// Method overriding example
class Animal {
    // Parent class method
    void makeSound() {
        System.out.println("Animal makes sound");
    }
}

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

class Cat extends Animal {
    // Another overridden method
    @Override
    void makeSound() {
        System.out.println("Cat meows");
    }

    public static void main(String[] args) {
        // Create objects
        Animal a = new Animal();
        Animal d = new Dog();
        Animal c = new Cat();

        // Method behavior depends on actual object
        a.makeSound();  // Animal makes sound
        d.makeSound();  // Dog barks
        c.makeSound();  // Cat meows
    }
}
```

Method Overriding Rules:
• Method must have **same name**
• Method must have **same parameters**
• Must be **IS-A relationship** (inheritance)
• Return type can be **same or covariant**
• Cannot reduce **access level**

📝 **Remember overriding rules as "NISRA"**:

- **N**ame must match, **I**nheritance required, **S**ignature must match, **R**eturn type same/covariant, **A**ccess modifier same/wider

Question 4(a OR): List any four inbuilt exceptions. (Marks: 03)

Inbuilt Exceptions in Java:

• **ArithmeticException**: Math errors (e.g., division by zero)
• **NullPointerException**: Accessing null object reference
• **ArrayIndexOutOfBoundsException**: Invalid array index
• **NumberFormatException**: Invalid number format in conversion
• **ClassCastException**: Invalid casting between incompatible classes
• **IllegalArgumentException**: Method receives improper argument
• **IOException**: Input/output operation failures
• **FileNotFoundException**: File access issues

Examples:

```
// ArithmeticException
int x = 10 / 0;

// NullPointerException
String str = null;
int length = str.length();

// ArrayIndexOutOfBoundsException
int[] arr = new int[5];
int value = arr[10];

// NumberFormatException
int num = Integer.parseInt("abc");
```

📝 **Remember common exceptions as "ANNIE"**:

- **A**rithmetic, **N**ullPointer, **N**umberFormat, **I**ndexOutOfBounds, **E**xception

Question 4(b OR): Explain "throw" keyword with suitable example. (Marks: 04)

throw keyword explicitly throws an exception.

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

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

            // This won't execute if exception occurs
            System.out.println("After validation");
```

```
        } catch (ArithmeticException e) {
            // Handle the thrown exception
            System.out.println("Exception caught: " + e.getMessage());
        }

        System.out.println("Program continues...");
    }
}
```

Uses of throw:
• **Manually throws** an exception
• Used for **custom validations**
• Can throw **standard or custom** exceptions
• Required for **control flow** in exception handling
• Must be handled with **try-catch or throws**

📝 **Remember as "MCCTH"**:

- **M**anually create exception, **C**ustom validations, **C**ontrol flow changes, **T**hrow within methods, **H**andling required

Question 4(c OR): Compare 'this' keyword Vs 'Super' keyword. Explain super keyword with suitable Example. (Marks: 07)

'this' vs 'super' Keywords:

'this' Keyword	'super' Keyword
References **current class** object	References **parent class** object
Used to access **current class** members	Used to access **parent class** members
Used with **constructor chaining** in same class	Used to call **parent constructor**
Resolves **variable shadowing**	Accesses **overridden methods/variables**
Used in **current class** context	Used in **inheritance** context

super Keyword Example:

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

    void eat() {
        System.out.println("Animal is eating");
    }
}

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

```
    void eat() {
        System.out.println("Dog is eating");
    }

    void printColor() {
        // Access current class variable
        System.out.println("Dog color: " + color);

        // Access parent class variable
        System.out.println("Animal color: " + super.color);
    }

    void doActions() {
        // Call current class method
        eat();

        // Call parent class method
        super.eat();
    }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.printColor();  // Prints both colors
        d.doActions();   // Calls both eat methods
    }
}
```

Uses of super Keyword:
• Access **parent class variables**
• Call **parent class methods**
• Invoke **parent class constructor**
• **Differentiate** between overridden members
• Essential in **method overriding**

📝 **Remember super uses as "VMCDO"**:

- **V**ariables of parent, **M**ethods of parent, **C**onstructor calling, **D**istinguish overridden members, **O**verride implementation

Question 5(a): List Different Stream Classes. (Marks: 03)

Java Stream Classes:

Byte Stream Classes (Binary Data):
• **FileInputStream**: Reads bytes from files
• **FileOutputStream**: Writes bytes to files
• **BufferedInputStream**: Buffered input for efficiency
• **BufferedOutputStream**: Buffered output for efficiency
• **DataInputStream**: Reads primitive data types
• **DataOutputStream**: Writes primitive data types

Character Stream Classes (Text Data):
• **FileReader**: Reads characters from files
• **FileWriter**: Writes characters to files
• **BufferedReader**: Buffered reading with readLine()
• **BufferedWriter**: Buffered writing with newLine()
• **PrintWriter**: Enhanced writing capabilities

📝 **Remember as "FBI-CRP"**:

- **F**ile streams, **B**uffered streams, **I**nput/Output streams, **C**haracter streams, **R**eaders, **P**rinters

Question 5(b): Write a java program to develop user defined exception for "Divide by zero" error. (Marks: 04)

```
// Custom exception
class DivideByZeroException extends Exception {
    // Constructor
    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 {
            // Try some divisions
            System.out.println("10/2 = " + divide(10, 2));  // Works fine
            System.out.println("20/0 = " + divide(20, 0));  // Throws exception
        } catch (DivideByZeroException e) {
            System.out.println("Custom Exception: " + e.getMessage());
        }
        System.out.println("Program continues...");
    }
}
```

Custom Exception Steps:
• **Create** a class extending Exception
• Define **constructor** passing message to parent
• **Use throw** to raise the exception
• **Handle** with try-catch or declare with throws

📝 **Remember as "CETH"**:

- **C**reate custom class, **E**xtend Exception, **T**hrow when needed, **H**andle properly

Question 5(c): Write a program in Java that reads the content of a file byte by byte and copy it into another file. (Marks: 07)

```java
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileCopyByteByByte {
    public static void main(String[] args) {
        // Define source and destination files
        String sourceFile = "source.txt";
        String destFile = "destination.txt";

        // Declare streams
        FileInputStream fis = null;
        FileOutputStream fos = null;

        try {
            // Initialize streams
            fis = new FileInputStream(sourceFile);
            fos = new FileOutputStream(destFile);

            System.out.println("Copying file...");

            int byteData;
            // Read file byte by byte (-1 means end of file)
            while ((byteData = fis.read()) != -1) {
                // Write byte to destination file
                fos.write(byteData);
            }

            System.out.println("File copied successfully!");

        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Close streams in finally block
            try {
                if (fis != null) fis.close();
                if (fos != null) fos.close();
                System.out.println("Streams closed.");
            } catch (IOException e) {
                System.out.println("Error closing streams: " + e.getMessage());
            }
        }
    }
}
```

Key Steps:
• **Create** FileInputStream for source file
• **Create** FileOutputStream for destination file
• **Read** source file byte by byte with read()
• **Write** each byte to destination with write()
• **Close** both streams when finished
• **Handle** potential IOException

📝 **Remember as "CREW-CH"**:

- **C**reate streams, **R**ead bytes, **E**xamine for EOF, **W**rite bytes, **C**lose streams, **H**andle exceptions

Question 5(a OR): List different file operations in Java. (Marks: 03)

File Operations in Java:

• **File Creation**: Creating new files
• **Reading**: Reading data from files
• **Writing**: Writing data to files
• **Appending**: Adding data to existing files
• **Deleting**: Removing files
• **Renaming**: Changing file names
• **Copying**: Duplicating files
• **Checking Existence**: Verifying if file exists
• **Getting File Info**: Size, path, permissions, etc.
• **Directory Operations**: Creating, listing, navigating

Example Code:

```
File file = new File("test.txt");
boolean exists = file.exists();     // Check existence
boolean created = file.createNewFile(); // Create new file
long size = file.length();          // Get file size
boolean deleted = file.delete();    // Delete file
```

📝 **Remember as "CRWADCEG"**:

- **C**reate, **R**ead, **W**rite, **A**ppend, **D**elete, **C**opy, **E**xistence check, **G**et info

Question 5(b OR): Write a java program to explain finally block in exception handling. (Marks: 04)

```
import java.io.FileInputStream;
import java.io.IOException;

public class FinallyBlockDemo {
    public static void main(String[] args) {
        FileInputStream fis = null;

        try {
```

```
            // Code that might throw exception
            fis = new FileInputStream("nonexistent.txt");
            int data = fis.read();
            System.out.println("Data read: " + data);

        } catch (IOException e) {
            // Exception handler
            System.out.println("Exception caught: " + e.getMessage());

        } finally {
            // This always executes - ideal for cleanup
            System.out.println("Finally block executed");

            // Close resources regardless of exception
            try {
                if (fis != null) fis.close();
                System.out.println("Stream closed");
            } catch (IOException e) {
                System.out.println("Error closing stream");
            }
        }

        System.out.println("Program continues...");
    }
}
```

finally Block Purpose:
• **Always executes** regardless of exception
• Used for **cleanup operations** (closing files/connections)
• Executes even if there's a **return statement** in try/catch
• Doesn't execute only if **System.exit()** is called
• Critical for **resource management**

📝 **Remember as "ACER"**:

- **A**lways executes, **C**leanup operations, **E**nsures resource release, **R**uns after try-catch

Question 5(c OR): Write a java program to create a file and perform write operation on this file. (Marks: 07)

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

public class FileWriteDemo {
    public static void main(String[] args) {
        // Declare resources
        Scanner scanner = null;
        FileWriter writer = null;

        try {
```

```
            // Create file object
            File file = new File("sample.txt");

            // Create new file if it doesn't exist
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }

            // Initialize file writer
            writer = new FileWriter(file);

            // Get user input
            scanner = new Scanner(System.in);
            System.out.println("Enter text to write to file (type 'exit' to stop):");

            String line;
            while (true) {
                line = scanner.nextLine();
                if (line.equalsIgnoreCase("exit")) break;

                // Write to file
                writer.write(line + "\n");
            }

            System.out.println("Content written to file successfully!");

        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Close resources
            try {
                if (writer != null) writer.close();
                if (scanner != null) scanner.close();
            } catch (IOException e) {
                System.out.println("Error closing resources: " + e.getMessage());
            }
        }
    }
}
```

File Write Operation Steps:
• **Create** File object
• **Check** if file exists or create new one
• **Initialize** FileWriter
• **Get** content to write (from user/source)
• **Write** content to file
• **Close** resources (writer, scanner)

📝 **Remember as "CCIGWC"**:

- **C**reate file object, **C**heck existence, **I**nitialize writer, **G**et content, **W**rite to file, **C**lose resources