CBCS SCHEME

USN | 1 | C | R | 2 | 3 | M | C | 0 | 8 | 0 | 22MCA22

Second Semester MCA Degree Examination, June/July 2024
Object Oriented Programming using Java

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Explain the key attributes of object oriented principles.	6	L2	CO1
	b.	What is narrowing and widening? Explain with example.	6	L1	CO1
	c.	Explain method overloading and constructor overloading with suitable example.	8	L2	CO1
		OR			
Q.2	a.	Differentiate procedural oriented programming and object oriented programming.	6	L2	CO1
	b.	How arrays are defined and initialized in Java? Explain with an example.	8	L2	CO1
	c.	Write a short note on 'this' keyword.	6	L1	CO1
		Module – 2			
Q.3	a.	Explain the usage of 'final' keyword with suitable example.	8	L2	CO2
	b.	Discuss 'super' keyword with an example.	8	L2	CO2
	c.	Write a Java program to list the factorial of the numbers 1 to 10. To calculate the factorial value, use while loop (Hint Fact of 4 = 4 * 3 * 2 * 1).	4	L3	CO2
		OR			
Q.4	a.	Define Inheritance. List and explain list of inheritance in Java.	8	L2	CO2
	b.	What is method overriding? Explain with example program.	6	L1	CO2
	c.	Explain 'abstract' keyword with an example.	6	L2	CO2
		Module – 3			
Q.5	a.	Define Interface. Write a java program for the implementation of multiple inheritance using interfaces to calculate the area of a rectangle and triangle.	10	L3	CO3
	b.	Write a Java program for the following. i) Create a package named 'shape' ii) Create some classes in the package representing some common shapes like 'square', 'triangle' and 'circle'. iii) Impact and compile these classes in other program.	10	L3	CO3

		OR			
Q.6	a.	Define package. Explain the access protection for class members with respect to package.	6	L2	CO3
	b.	Differentiae abstract class and interface.	6	L2	CO3
	c.	Explain with an example, how interfaces can be extended.	8	L3	CO3
		Module – 4			
Q.7	a.	What is an exception? Explain the exception handling mechanism with suitable example.	10	L3	CO4
	b.	Explain how to create your own exceptions. Give an example.	10	L3	CO4
		OR			
Q.8	a.	What is checked and unchecked exception? Write a Java program to illustrate nested try catch statement.	10	L3	CO4
	b.	Write a Java program to demonstrate a division by zero exception.	4	L3	CO4
	c.	Differentiate between throw and throws with example.	6	L2	CO4
		Module – 5			
Q.9	a.	Define AWT. List and explain types of containers in Java AWT.	6	L2	CO5
	b.	Write a Java program to create a window when we press, i) M or m the windows display Good Morning ii) A or a the windows display Good Afternoon iii) E or e the window display Good Evening iv) N or n the window display Good Night	10	L3	CO5
	c.	Write a short note on swings.	4	L2	CO5
		OR			
Q.10	a.	Define Applet. Explain life cycle of applets.	6	L2	CO5
	b.	Write a Java applet program, which handles keyword event.	10	L3	CO5
	c.	Write a short note on JFrames.	4	L1	CO5

* * * * *

1. a) The **key attributes of object-oriented programming (OOP)** principles are fundamental concepts that define how software can be structured and designed around objects. These principles are **Encapsulation**, **Abstraction**, **Inheritance**, and **Polymorphism**. Each plays a unique role in creating efficient, modular, and scalable systems.

1. Encapsulation

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit, known as an **object**. It also restricts direct access to some of the object's components to ensure data security and integrity.

Key Points:

Data Hiding: Internal details are hidden from the outside world.

Controlled Access: Access to data is typically provided through methods like **getters** and **setters**.

Ensures that objects maintain control over their state.

2. Abstraction

Abstraction focuses on **hiding complex implementation details** and exposing only the necessary functionality. It allows developers to interact with objects at a higher level without needing to understand their internal workings.

Key Points:

Simplifies Complex Systems: Users interact with simplified interfaces.

Focus on What, Not How: Only essential operations are exposed.

Often achieved through **abstract classes** and **interfaces**.

3. Inheritance

Inheritance is the ability to create a new class (child class) that derives or inherits the properties and behaviors of an existing class (parent class). This promotes code reuse and establishes a hierarchical relationship.

Key Points:

Code Reusability: Reduces duplication by reusing existing code.

Hierarchy: Models real-world relationships like **"is-a"** relationships.

The child class can **extend** or **override** parent class functionality.

4. Polymorphism

Polymorphism means **"many forms"** and allows methods in different classes to share the same name but exhibit different behaviors. It enables the same interface to be used with different underlying data types or objects.

Key Points:

Flexibility: A single interface can handle different types of objects.

Method Overloading and Overriding: Provides multiple ways to perform the same operation.

Promotes extensibility and adaptability.

5. Classes and Objects

While not a principle per se, **classes and objects** are the building blocks of OOP. A **class** serves as a blueprint, defining the structure and behavior, while an **object** is an instance of that class.

Key Points:

Classes: Define attributes (data) and methods (behavior).

Objects: Represent specific instances with their own data and state.

Multiple objects can be created from a single class.

Summary of OOP Attributes

Attribute	Description
Encapsulation	Combines data and methods into a single unit; restricts direct data access.
Abstraction	Hides implementation details; exposes only essential functionality.
Inheritance	Enables new classes to inherit and extend existing class properties and behavior.
Polymorphism	Allows methods or operations to behave differently depending on the object or data.
Classes & Objects	Classes define the structure, while objects are specific instances of those classes.

1. b)
   In Java, **narrowing** and **widening** refer to the conversion of one data type to another. These conversions occur between primitive data types or between object references and are governed by the size or compatibility of the data types.

   1. Widening Conversion (Implicit Type Casting)

- **Definition:** Widening occurs when a smaller data type is automatically converted to a larger data type.
  - **No loss of data** happens during this conversion.
  - Done **implicitly** by the compiler.
- **Example:** Converting int to long or float to double.

```
public class WideningExample {
   public static void main(String[] args) {
      int num = 100;       // Integer value
      long longNum = num;   // Automatically converted to long
      double doubleNum = num; // Automatically converted to double

      System.out.println("Integer: " + num);
      System.out.println("Long: " + longNum);
      System.out.println("Double: " + doubleNum);
   }
}
```

Output:

```
Integer: 100
Long: 100
Double: 100.0
```

Key Points:

- Widening is safe because the target type has a larger range.
- Examples:
  - byte → short → int → long → float → double.

2. Narrowing Conversion (Explicit Type Casting)

- **Definition:** Narrowing occurs when a larger data type is explicitly converted to a smaller data type.
  - **Data loss** or truncation may occur.
  - Requires an **explicit cast** by the programmer.
- **Example:** Converting double to int or long to byte.

```
public class NarrowingExample {
   public static void main(String[] args) {
      double num = 99.99;  // Double value
      int intNum = (int) num; // Explicitly cast to int
```

```
        long longNum = 1000L;
        byte byteNum = (byte) longNum; // Explicitly cast to byte

        System.out.println("Double: " + num);
        System.out.println("Converted to Int (truncated): " + intNum);
        System.out.println("Long: " + longNum);
        System.out.println("Converted to Byte (overflow): " + byteNum);
    }
}
```

Output:

Double: 99.99
Converted to Int (truncated): 99
Long: 1000
Converted to Byte (overflow): -24

Key Points:

- Narrowing can lead to **loss of precision** or **overflow**.
- Requires explicit casting because it's **unsafe**.
- Examples:
    - double → float → long → int → short → byte.

1. c)

   Method Overloading

   Definition:

   Method overloading in Java occurs when multiple methods in the same class have the same name but differ in their **parameter list** (type, number, or order of parameters). It allows methods to perform similar but slightly different tasks using the same name.

   Key Features of Method Overloading:

1. **Same method name**, but different parameter lists.
2. Return type **can be different**, but it alone cannot distinguish overloaded methods.
3. **Compile-time polymorphism**: The method to be invoked is determined at compile-time.

   Example: Method Overloading

   java
   Copy code

```
class Calculator {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two double values
    double add(double a, double b) {
        return a + b;
    }
}

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

```
        Calculator calc = new Calculator();

        System.out.println("Addition of two integers: " + calc.add(5, 10));      // Calls add(int, int)
        System.out.println("Addition of three integers: " + calc.add(5, 10, 15)); // Calls add(int, int, int)
        System.out.println("Addition of doubles: " + calc.add(5.5, 10.2));      // Calls add(double, double)
    }
}
```

Output:

```
Addition of two integers: 15
Addition of three integers: 30
Addition of doubles: 15.7
```

Constructor Overloading

Definition:

Constructor overloading occurs when a class has multiple constructors with the same name (the class name) but different parameter lists. It enables the creation of objects with different initializations.

Key Features of Constructor Overloading:

1. **Same constructor name**, but different parameter lists.
2. Allows different ways to initialize an object.
3. Achieves flexibility in object creation.

Example: Constructor Overloading

```
java
Copy code
class Student {
    String name;
    int age;
    String course;

    // Constructor with no parameters
    Student() {
        name = "Unknown";
        age = 0;
        course = "Not enrolled";
    }

    // Constructor with one parameter
    Student(String name) {
        this.name = name;
        this.age = 18;  // Default age
        this.course = "Not enrolled";
    }

    // Constructor with all parameters
    Student(String name, int age, String course) {
        this.name = name;
        this.age = age;
        this.course = course;
    }

    void displayDetails() {
        System.out.println("Name: " + name + ", Age: " + age + ", Course: " + course);
    }
}
```

```
public class ConstructorOverloadingExample {
   public static void main(String[] args) {
      // Using different constructors
      Student student1 = new Student();
      Student student2 = new Student("Alice");
      Student student3 = new Student("Bob", 22, "Computer Science");

      // Display student details
      student1.displayDetails();
      student2.displayDetails();
      student3.displayDetails();
   }
}
```

Output:

Name: Unknown, Age: 0, Course: Not enrolled
Name: Alice, Age: 18, Course: Not enrolled
Name: Bob, Age: 22, Course: Computer Science

Comparison: Method Overloading vs Constructor Overloading

Aspect	**Method Overloading**	**Constructor Overloading**
Definition	Same method name, different parameter list.	Same constructor name, different parameter list.
Purpose	Implements tasks with variations based on arguments.	Creates objects with different initializations.
Invocation	Called explicitly by the programmer using the method name.	Called implicitly when creating objects with the new keyword.
Example	add(int, int) vs add(double, double).	Student() vs Student(String) vs Student(String, int).

2. a)

Comparison Between Procedural-Oriented Programming (POP) and Object-Oriented Programming (OOP)

Aspect	**Procedural-Oriented Programming (POP)**	**Object-Oriented Programming (OOP)**
Definition	Focuses on functions and procedures to solve problems.	Focuses on objects and their interactions to model real-world problems.
Program Structure	Divided into **functions** and **procedures**.	Divided into **classes** and **objects**.
Data Management	Data is typically **global** or **shared** between functions.	Data is **encapsulated** within objects and accessed via methods.

Aspect	Procedural-Oriented Programming (POP)	Object-Oriented Programming (OOP)
Security	Data is less secure as it is accessible by any function.	Data is more secure due to **encapsulation** and controlled access.
Modularity	Functions are the building blocks.	Objects and classes are the building blocks.
Reusability	Limited reusability; code reuse is achieved via functions.	High reusability through **inheritance** and **polymorphism**.
Real-World Representation	Does not directly represent real-world scenarios.	Models real-world entities using **objects**.
Code Complexity	Can become complex as the program grows larger.	Easier to manage and extend due to modularity and abstraction.
Examples of Languages	C, Pascal, Fortran.	Java, Python, C++, Ruby.
Focus	Focuses on **how** tasks are done (step-by-step execution).	Focuses on **what** objects do (behavior) and their attributes.
Function Overloading	Not supported.	Supported (method overloading).
Inheritance	Not available.	Key feature that allows extending existing classes.
Polymorphism	Not available.	Supported (e.g., method overriding and overloading).

Key Differences Explained:

1. **Program Structure:**
   - In POP, the primary focus is on creating functions that perform operations. The data is passed between these functions.
   - In OOP, the program is built around classes that encapsulate data and methods, forming objects that interact with one another.
2. **Encapsulation and Security:**
   - In POP, data can be freely accessed and modified by any function, making it harder to track changes and ensure security.
   - In OOP, data is encapsulated within objects, and access is restricted using modifiers like private or protected.
3. **Modularity and Reusability:**
   - POP provides limited reusability. If a procedure needs to be reused, it must be explicitly called and written for the specific context.
   - OOP inherently supports code reuse through features like **inheritance** and **polymorphism**, enabling the creation of new classes from existing ones.
4. **Real-World Representation:**
   - POP does not model real-world entities effectively, as it lacks a direct way to represent objects.
   - OOP models real-world entities as objects with attributes (data) and behaviors (methods).

Examples:

Procedural-Oriented Programming:

```
#include <stdio.h>

void calculateArea(float radius) {
   printf("Area: %.2f\n", 3.14 * radius * radius);
}

int main() {
   float radius = 5.0;
   calculateArea(radius);
   return 0;
}
```

Object-Oriented Programming:

```
java
Copy code
class Circle {
   private double radius;

   Circle(double radius) {
      this.radius = radius;
   }

   void calculateArea() {
      System.out.println("Area: " + (3.14 * radius * radius));
   }
}

public class Main {
   public static void main(String[] args) {
      Circle circle = new Circle(5.0);
      circle.calculateArea();
   }
}
```

2. b) **Arrays in Java**

An **array** in Java is a container object that holds a fixed number of elements of the same data type. Arrays are used to store multiple values in a single variable, which makes them useful for managing collections of data.

Defining and Initializing Arrays

1. **Defining an Array** To define an array, you specify the data type of its elements, followed by square brackets [], and then the array name:

   java

   Copy code

   dataType[] arrayName;

   Alternatively:

   java

Copy code

dataType arrayName[];

2. **Initializing an Array** An array can be initialized in two ways:
    - **At Declaration:** Directly assigning values to the array.
    - **After Declaration:** Creating the array object and assigning values.

    Examples

    1. Declaring and Initializing an Array

- **At Declaration:**

    java

    Copy code

    int[] numbers = {10, 20, 30, 40, 50};

- **After Declaration:**

    java

    Copy code

    int[] numbers = new int[5]; // Array of size 5

    numbers[0] = 10;

    numbers[1] = 20;

    numbers[2] = 30;

    numbers[3] = 40;

    numbers[4] = 50;

    2. Accessing Array Elements

    Array elements are accessed using their **index**, starting from 0.

    java

    Copy code

    public class ArrayExample {

      public static void main(String[] args) {

        int[] numbers = {10, 20, 30, 40, 50}; // Declaring and initializing

        System.out.println("First element: " + numbers[0]); // Accessing the first element

        System.out.println("Third element: " + numbers[2]); // Accessing the third element

      }

    }

    Output:

    First element: 10

Third element: 30

3. Looping Through an Array

You can use a loop to iterate through array elements.

java

Copy code

```
public class ArrayIteration {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};

        System.out.println("Array elements:");
        for (int i = 0; i < numbers.length; i++) { // Using a for loop
            System.out.println(numbers[i]);
        }

        System.out.println("Using enhanced for loop:");
        for (int number : numbers) { // Using enhanced for loop
            System.out.println(number);
        }
    }
}
```

Output:

Array elements:

10

20

30

40

50

Using enhanced for loop:

10

20

30

40

50

4. Multidimensional Arrays

Java supports multidimensional arrays, such as 2D arrays.

```
public class MultiDimensionalArray {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        System.out.println("Element at [1][1]: " + matrix[1][1]); // Accessing element
    }
}
```

Output:

Element at [1][1]: 5

Key Points

1. Array elements are stored in **contiguous memory locations**.
2. The size of an array is **fixed** after initialization.
3. Arrays can store **primitive data types** or **objects**.
4. **Default values** for array elements:
    - Numeric types: 0
    - char: \u0000 (null character)
    - Boolean: false
    - Reference types: null

2. c)

The this Keyword in Java

The this keyword in Java is a reference variable that refers to the **current instance** of a class. It is used primarily in methods and constructors to refer to the current object of the class.

Key Uses of the this Keyword:

1. **Referring to Instance Variables:** this is used to differentiate between instance variables and local variables when they have the same name.

```
class Person {
    String name;  // Instance variable
    Person(String name) {
        this.name = name;  // 'this.name' refers to the instance variable, 'name' refers to the constructor parameter
```

```
    }
    void display() {
        System.out.println("Name: " + this.name);  // Using 'this' to refer to the instance variable
    }
}
```

In this example, the constructor parameter name is shadowing the instance variable name. Using this.name refers to the instance variable.

2. **Invoking Current Class Methods:** You can use this to call other methods within the same class.

```
class Calculator {
    void add() {
        this.display();  // Calling another method in the same class
    }
    void display() {
        System.out.println("Hello from display!");
    }
}
```

3. **Referring to the Current Object:** this can be used to pass the current object as an argument to another method or constructor.

```
java
Copy code
class Box {
    int length;
    int width;
    Box(int length, int width) {
        this.length = length;
        this.width = width;
    }
    void printBox(Box box) {
        System.out.println("Length: " + box.length + ", Width: " + box.width);
    }
    void display() {
        this.printBox(this);  // Passing current object 'this' to the method
    }
}
```

4. **Calling Another Constructor:** The this() constructor call can be used to invoke another constructor in the same class. This must be the first statement in the constructor.

```
java

Copy code

class Student {
   String name;
   int age;
   Student() {
      this("Unknown", 18);  // Calls another constructor with parameters
   }
   Student(String name, int age) {
      this.name = name;
      this.age = age;
   }
}
```

Key Points:

- this refers to the **current instance** of a class.
- It helps avoid confusion between instance variables and method parameters when they have the same name.
- It is used to call other methods and constructors within the same class.
- It can be used to pass the current object as a parameter to another method.

3. a)

The final Keyword in Java

The final keyword in Java is used to indicate that a variable, method, or class cannot be modified. It is used in different contexts to provide restrictions or enforce immutability.

Usage of final Keyword:

1. **Final Variables:**
   - When a variable is declared as final, its value cannot be modified once it is initialized. This makes the variable **constant**.
   - For **primitive types**, the value cannot be changed.
   - For **reference types**, the reference (address) cannot be changed, but the object it points to can be modified.

Example:

```
public class FinalVariableExample {
   public static void main(String[] args) {
      final int MAX_VALUE = 100;  // final primitive variable
      System.out.println(MAX_VALUE);

      // MAX_VALUE = 200;  // Error: cannot assign a value to a final variable
   }
}
```

Output:

Copy code

For **reference variables**, once assigned, the reference cannot point to a different object, but you can modify the object's fields (if they are not final).

```
public class FinalReferenceExample {
    public static void main(String[] args) {
        final StringBuilder sb = new StringBuilder("Hello");
        sb.append(" World");  // This is allowed, as the object's content can be modified.
        System.out.println(sb);  // Output: Hello World

        // sb = new StringBuilder("New String");  // Error: cannot assign a new object to final variable
    }
}
```

2. **Final Methods:**
    - When a method is declared as final, it cannot be overridden by subclasses. This ensures that the behavior of the method is **fixed** and cannot be changed.

Example:

```
class Parent {
    final void show() {
        System.out.println("This is a final method.");
    }
}

class Child extends Parent {
    // Error: Cannot override the final method from Parent
    // void show() {
    //     System.out.println("Overriding method.");
    // }
}

public class FinalMethodExample {
    public static void main(String[] args) {
        Parent obj = new Parent();
        obj.show();  // Output: This is a final method.
    }
}
```

Output:

```
This is a final method.
```

3. **Final Classes:**
    - A class declared as final cannot be subclassed or extended. This ensures that the class's implementation is **immutable** and cannot be changed.

Example:

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

// Error: Cannot subclass the final class FinalClass
// class SubClass extends FinalClass {
//     // Compilation error: Cannot subclass the final class FinalClass
// }
```

```
public class FinalClassExample {
   public static void main(String[] args) {
      FinalClass obj = new FinalClass();
      obj.display();  // Output: This is a final class.
   }
}
```

Output:

This is a final class.

Key Points:

1. **final variable:** Once initialized, its value cannot be changed.
    - For primitives, the value is immutable.
    - For references, the reference cannot point to another object, but the object's state can be changed unless the fields are final as well.
2. **final method:** Cannot be overridden by subclasses, ensuring the method's behavior remains unchanged.
3. **final class:** Cannot be subclassed or extended. This is often used to create immutable classes like String.

3. b)

The super Keyword in Java

In Java, the super keyword is used to refer to the **parent class** (superclass) of the current object. It is typically used in the context of **inheritance** to refer to parent class methods, constructors, and variables.

The super keyword helps to access members of the superclass that may be hidden or overridden by the subclass.

Key Uses of the super Keyword:

1. **Calling Parent Class Constructor:**
    - The super() keyword can be used to call the constructor of the parent class.
    - It must be the **first statement** in the subclass constructor.
2. **Accessing Parent Class Methods:**
    - If a subclass overrides a method of the parent class, the super keyword can be used to call the method of the parent class.
3. **Accessing Parent Class Variables:**
    - If the subclass has a variable with the same name as a parent class variable, super is used to refer to the parent class's variable.

Example 1: Calling Parent Class Constructor

In the following example, the super() keyword is used to call the constructor of the parent class Animal from the subclass Dog.

```
class Animal {
   Animal() {
      System.out.println("Animal constructor called");
   }
}
```

```
class Dog extends Animal {
    Dog() {
        // Calling the parent class constructor
        super();  // This is optional here as Java calls it automatically if not specified
        System.out.println("Dog constructor called");
    }
}
public class SuperKeywordExample {
    public static void main(String[] args) {
        Dog dog = new Dog();  // Creates an instance of Dog, which calls both Animal and Dog constructors
    }
}
```

Output:

```
Animal constructor called
Dog constructor called
```

- In this example, super() explicitly calls the constructor of the parent class Animal, which is executed before the Dog constructor.

Example 2: Accessing Parent Class Method

In the following example, the super keyword is used to access the overridden method of the parent class.

```
class Animal {
    void makeSound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {
    void makeSound() {
        System.out.println("Dog barks");
    }
    void callSuperMethod() {
        // Calling the parent class's method
        super.makeSound();  // Uses super to call the overridden method in the parent class
    }
}
public class SuperKeywordExample {
    public static void main(String[] args) {
```

```
        Dog dog = new Dog();
        dog.makeSound();      // Calls the Dog class's method
        dog.callSuperMethod(); // Calls the parent class's method using super
    }
}
```

Output:

```
Dog barks
Animal makes sound
```

- Here, makeSound() is overridden in the Dog class. When we call super.makeSound(), the method of the Animal class is invoked, despite being overridden in the Dog class.

Example 3: Accessing Parent Class Variable

In this example, the super keyword is used to access the variable name from the parent class Animal, even though it is overshadowed by a variable with the same name in the subclass Dog.

```
class Animal {
    String name = "Animal";
    void display() {
        System.out.println("Name: " + name);
    }
}
class Dog extends Animal {
    String name = "Dog";
    void display() {
        System.out.println("Name (in Dog class): " + name);
        System.out.println("Name (from Animal class): " + super.name);  // Accessing parent class variable using super
    }
}
public class SuperKeywordExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.display();
    }
}
```

Output:

```
Name (in Dog class): Dog
```

Name (from Animal class): Animal

- Here, name is defined in both the Animal and Dog classes. Using super.name, we are able to refer to the name variable from the parent class Animal.

Key Points:

- super is used to **refer to the parent class**.
- It is used to call the **parent class constructor**, **methods**, and access **variables**.
- **super()** is typically used to invoke the parent class constructor, and it must be the first statement in the subclass constructor.
- When a subclass **overrides** a parent class method, super can be used to access the method in the parent class.
- **Accessing parent class variables** with super is useful when there is a naming conflict with variables in the subclass.

3. c)

```
public class FactorialCalculator {
   public static void main(String[] args) {
      // Loop from 1 to 10
      for (int i = 1; i <= 10; i++) {
         // Calculate factorial of the current number
         long factorial = 1;
         if (i == 4) {
            // Use while loop to calculate the factorial of 4
            int j = 4;
            while (j > 0) {
               factorial *= j;
               j--;
            }
         } else {
            // For other numbers, use a for loop to calculate factorial
            for (int j = 1; j <= i; j++) {
               factorial *= j;
            }
         }
         // Print the factorial of the number
         System.out.println("Factorial of " + i + " is: " + factorial);
      }
   }
}
```

Output:
Factorial of 1 is: 1
Factorial of 2 is: 2
Factorial of 3 is: 6
Factorial of 4 is: 24
Factorial of 5 is: 120
Factorial of 6 is: 720
Factorial of 7 is: 5040
Factorial of 8 is: 40320
Factorial of 9 is: 362880
Factorial of 10 is: 3628800

4. a)

Inheritance is one of the core principles of **Object-Oriented Programming (OOP)** in Java. It allows one class (called a **subclass** or **child class**) to inherit the properties (fields) and behaviors (methods) of another class (called the **superclass** or **parent class**). This helps in code reuse, making it easier to maintain and extend the codebase.

In Java, inheritance enables the subclass to:

- **Inherit** the non-private fields and methods of the superclass.
- **Override** the methods of the superclass to provide specific implementations.
- **Extend** the functionality of the parent class without modifying the original class.

The **extends** keyword is used in Java to define inheritance. The subclass inherits all public and protected members (fields and methods) of the superclass, but it cannot access the private members.

Types of Inheritance in Java

There are different types of inheritance in Java, each representing how classes are related:

1. Single Inheritance:

In **single inheritance**, a class can inherit from only one superclass. This is the simplest form of inheritance, where a child class inherits the members (fields and methods) of only one parent class.

- **Example:**

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

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

public class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.eat();  // Inherited method
    dog.bark(); // Dog's own method
  }
}
```

Output:

```
Eating...
Barking...
```

2. Multilevel Inheritance:

In **multilevel inheritance**, a class inherits from another class, and then another class inherits from the child class. This forms a chain of inheritance.

- **Example:**

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

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

```
class Puppy extends Dog {  // Puppy inherits from Dog
    void play() {
        System.out.println("Playing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Puppy puppy = new Puppy();
        puppy.eat();  // Inherited from Animal
        puppy.bark(); // Inherited from Dog
        puppy.play(); // Own method in Puppy
    }
}
```

Output:

```
Eating...
Barking...
Playing...
```

3. Hierarchical Inheritance:

In **hierarchical inheritance**, multiple subclasses inherit from a single superclass. In this case, all subclasses share the common properties and behaviors of the superclass.

- **Example:**

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

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

class Cat extends Animal {  // Cat also inherits from Animal
    void meow() {
        System.out.println("Meowing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method
        dog.bark(); // Dog's own method

        Cat cat = new Cat();
        cat.eat();  // Inherited method
        cat.meow(); // Cat's own method
    }
}
```

Output:

Eating...
Barking...
Eating...
Meowing...

4. Multiple Inheritance (Not Supported in Java):

Java does not support **multiple inheritance** through classes, meaning a class cannot directly inherit from more than one class. This is done to avoid ambiguity issues (for example, if two superclasses define the same method). However, Java allows multiple inheritance through **interfaces** (explained below).

- **Note:** Java doesn't allow a class to inherit from more than one class:

```
// This will cause an error
class Animal {
  void eat() {
    System.out.println("Eating...");
  }
}

class Pet {
  void play() {
    System.out.println("Playing...");
  }
}

// Multiple inheritance is not allowed with classes
class Dog extends Animal, Pet { // Error: Class Dog cannot have multiple parents
}
```

However, **interfaces** can be used to achieve multiple inheritance in Java.

- **Example of multiple inheritance using interfaces:**

```
interface Animal {
  void eat();
}

interface Pet {
  void play();
}

class Dog implements Animal, Pet {
  public void eat() {
    System.out.println("Eating...");
  }

  public void play() {
    System.out.println("Playing...");
  }
}

public class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.eat();  // From Animal interface
    dog.play(); // From Pet interface
  }
}
```

Output:

Eating...
Playing...

4. b)

Method Overriding is a feature of **Object-Oriented Programming (OOP)** in Java where a subclass provides its own implementation of a method that is already defined in its superclass. The method in the subclass must have the **same name**, **same parameter list**, and **same return type** (or subtype) as the method in the parent class.

Method overriding is used to implement **runtime polymorphism**, which allows a subclass to modify the behavior of a method inherited from its superclass.

Key Points about Method Overriding:

1. **Same method signature**: The method in the subclass must have the same method signature (name, parameters, and return type) as in the parent class.
2. **Access modifiers**: The overridden method in the subclass cannot have a more restrictive access modifier than the method in the parent class. For example, if the parent class method is public, the overridden method in the subclass must also be public.
3. **Dynamic Method Dispatch**: The version of the overridden method that gets called is determined at runtime, not compile-time.
4. **The @Override annotation**: While not mandatory, it is good practice to use the @Override annotation to indicate that a method is intended to override a method in the parent class.

Example Program: Method Overriding

Let's consider an example where a Animal class has a method makeSound(), and we override that method in the Dog class and Cat class.

```
// Parent class
class Animal {
   // Method in parent class
   void makeSound() {
      System.out.println("Animal makes a sound");
   }
}

// Child class - Dog overrides the makeSound() method
class Dog extends Animal {
   // Overriding method in child class
   @Override
   void makeSound() {
      System.out.println("Dog barks");
   }
}

// Child class - Cat overrides the makeSound() method
class Cat extends Animal {
   // Overriding method in child class
   @Override
   void makeSound() {
      System.out.println("Cat meows");
   }
}

public class MethodOverridingExample {
   public static void main(String[] args) {
      // Creating instances of Dog and Cat classes
      Animal myDog = new Dog();  // Upcasting
      Animal myCat = new Cat();  // Upcasting

      // Calling overridden methods
```

```
        myDog.makeSound();  // Calls Dog's makeSound()
        myCat.makeSound();  // Calls Cat's makeSound()

        // Creating an instance of the parent class
        Animal myAnimal = new Animal();
        myAnimal.makeSound();  // Calls Animal's makeSound()
    }
}
```

Explanation:

1. **Parent Class (Animal)**: It defines a method makeSound() that prints "Animal makes a sound".
2. **Child Classes (Dog and Cat)**: Both override the makeSound() method to provide their own specific implementation. The Dog class prints "Dog barks", and the Cat class prints "Cat meows".
3. **Upcasting**: In the main method, we create objects of Dog and Cat, but assign them to references of type Animal. This is called **upcasting**, and it is possible because both Dog and Cat are subclasses of Animal.
4. **Runtime Polymorphism**: When we call makeSound() on an Animal reference that points to a Dog or Cat object, Java dynamically determines at runtime which version of the method to call based on the actual object type (not the reference type). This is **dynamic method dispatch**.

Output:

Dog barks
Cat meows
Animal makes a sound

- When myDog.makeSound() is called, the makeSound() method in the Dog class is executed, which prints "Dog barks".
- When myCat.makeSound() is called, the makeSound() method in the Cat class is executed, which prints "Cat meows".
- When myAnimal.makeSound() is called, it executes the method in the Animal class, printing "Animal makes a sound".

Benefits of Method Overriding:

1. **Polymorphism**: Overriding allows Java to make decisions at runtime about which method to invoke, enabling polymorphism.
2. **Customized Behavior**: It allows a subclass to provide a specific implementation for a method inherited from a parent class.
3. **Code Reusability**: While overriding, you can still reuse the code of the parent class by calling super.method() inside the overridden method if necessary.
4. **Cleaner and More Flexible Code**: Method overriding allows you to maintain the same method signature while providing new or more specific behavior in derived classes.

4. c) Explain 'abstract" keyword with an example,

In Java, the abstract keyword is used to define **abstract classes** and **abstract methods**.

- **Abstract Class**: An abstract class is a class that cannot be instantiated directly. It is meant to be extended by other classes. It can contain both abstract methods (methods without a body) and regular methods (methods with a body).
- **Abstract Method**: An abstract method is a method that is declared without an implementation. It does not have a body and is meant to be overridden in subclasses.

The abstract keyword provides the mechanism for creating a **template** for other classes. Abstract classes allow you to define **general behavior** that can be shared across multiple subclasses, but require the subclasses to provide **specific implementations**.

Key Points about Abstract Keyword:

1. **Abstract Class**:
    - Cannot be instantiated directly (i.e., you cannot create an object of an abstract class).
    - Can have both abstract and non-abstract methods.
    - Can have instance variables, constructors, and methods with code.
2. **Abstract Method**:
    - A method without a body (i.e., no implementation).
    - Must be implemented by the subclasses of the abstract class (unless the subclass is also abstract).
3. **Concrete Subclass**: A subclass that extends an abstract class must provide implementations for all the abstract methods of the parent class (unless the subclass is abstract).

Example: Using the abstract Keyword

Let's define an example where we have an abstract class Animal, which has an abstract method sound(). We will then create two concrete subclasses, Dog and Cat, which provide their specific implementations of the sound() method.

```
// Abstract class
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Regular method with a body
    void sleep() {
        System.out.println("This animal is sleeping");
    }
}

// Subclass Dog that extends Animal and implements the abstract method
class Dog extends Animal {
    // Providing implementation for the abstract method sound()
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Subclass Cat that extends Animal and implements the abstract method
class Cat extends Animal {
    // Providing implementation for the abstract method sound()
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        // Animal animal = new Animal(); // Error: cannot instantiate the abstract class Animal

        // Creating instances of Dog and Cat
        Animal dog = new Dog();
        Animal cat = new Cat();

        // Calling methods
        dog.sound(); // Calls Dog's implementation of sound
        dog.sleep(); // Calls the regular method sleep (inherited from Animal)

        cat.sound(); // Calls Cat's implementation of sound
        cat.sleep(); // Calls the regular method sleep (inherited from Animal)
    }
}
```

Explanation:

1. **Abstract Class (Animal)**:
    - The sound() method is declared as abstract. This means that any class that extends Animal must provide its own implementation of sound().
    - The sleep() method is a regular method with a body. It provides functionality that can be inherited by subclasses.
2. **Concrete Subclasses (Dog and Cat)**:

- Both Dog and Cat extend the Animal class.
- Each subclass provides its own implementation of the abstract sound() method. The Dog class implements it as "Dog barks", and the Cat class implements it as "Cat meows".

3. **In the Main method**:
   - We cannot instantiate an object of the Animal class directly, as it is abstract.
   - We create instances of the Dog and Cat classes and call both the overridden sound() method and the inherited sleep() method.

Output:

Dog barks
This animal is sleeping
Cat meows
This animal is sleeping

Key Benefits of Using the abstract Keyword:

1. **Template for Subclasses**: The abstract class serves as a template for its subclasses, enforcing a certain structure by requiring subclasses to implement abstract methods.
2. **Code Reusability**: The abstract class allows code sharing across subclasses, especially if there are common methods that can be used by all subclasses, like the sleep() method in the example.
3. **Flexibility**: Subclasses have the flexibility to provide specific implementations of the abstract methods, which enables polymorphism.
4. **Design Pattern**: Abstract classes are often used in design patterns (like Template Method pattern) where the abstract class provides a framework with certain steps that need to be completed by subclasses.

5. a) Define Interface. Write a java program for the implementation of multiple 10 inheritance using interfaces to calculate the area of a rectangle and triangle.

An **interface** in Java is a blueprint of a class that contains only abstract methods (until Java 8) and constants (static final fields). Starting from Java 8, interfaces can also have **default** and **static methods**. From Java 9 onwards, they can have **private methods** as well.
Interfaces are used to achieve **full abstraction** and **multiple inheritance**, as Java does not support multiple inheritance with classes due to the diamond problem.

Key Features of Interfaces

1. An interface is declared using the interface keyword.
2. It can contain abstract methods (implicitly public and abstract) and constants (public, static, and final by default).
3. A class implements an interface using the implements keyword and must override all its abstract methods.
4. A class can implement multiple interfaces, thus achieving multiple inheritance.

Java Program: Multiple Inheritance using Interfaces

The following program demonstrates multiple inheritance using interfaces to calculate the **area of a rectangle** and the **area of a triangle**.

```
// Interface to calculate the area of a rectangle
interface RectangleArea {
   void calculateRectangleArea(double length, double breadth);
}

// Interface to calculate the area of a triangle
interface TriangleArea {
   void calculateTriangleArea(double base, double height);
}

// Class implementing both interfaces
class AreaCalculator implements RectangleArea, TriangleArea {

   // Overriding method from RectangleArea interface
   @Override
   public void calculateRectangleArea(double length, double breadth) {
```

```
        double area = length * breadth;
        System.out.println("Area of Rectangle: " + area);
    }

    // Overriding method from TriangleArea interface
    @Override
    public void calculateTriangleArea(double base, double height) {
        double area = 0.5 * base * height;
        System.out.println("Area of Triangle: " + area);
    }
}

public class MultipleInheritanceExample {
    public static void main(String[] args) {
        // Create an object of AreaCalculator
        AreaCalculator calculator = new AreaCalculator();

        // Calculate and display areas
        calculator.calculateRectangleArea(10, 5);  // Rectangle with length=10, breadth=5
        calculator.calculateTriangleArea(6, 8);   // Triangle with base=6, height=8
    }
}
```

Explanation:

1. **Interfaces (RectangleArea and TriangleArea)**:
   - The RectangleArea interface declares an abstract method calculateRectangleArea.
   - The TriangleArea interface declares an abstract method calculateTriangleArea.
2. **Class (AreaCalculator)**:
   - Implements both interfaces using the implements keyword.
   - Provides concrete implementations for the methods declared in the interfaces.
3. **Main Method**:
   - An object of the AreaCalculator class is created.
   - The methods calculateRectangleArea and calculateTriangleArea are called to calculate and display the respective areas.

Output:

Area of Rectangle: 50.0
Area of Triangle: 24.0

Key Benefits of Interfaces in Multiple Inheritance

1. **Avoids Diamond Problem**: Unlike classes, multiple interfaces can be implemented without conflicts because interfaces provide only method declarations and no implementation.
2. **Promotes Abstraction**: Interfaces help define a contract that implementing classes must adhere to.
3. **Flexibility**: A class can implement multiple interfaces, making it versatile and reusable.

5. b)

File: Square.java

```
package shape;

public class Square {
    private double side;

    // Constructor
    public Square(double side) {
        this.side = side;
    }
```

```
    // Method to calculate area
    public double getArea() {
        return side * side;
    }
}
```

File: Triangle.java

```
package shape;

public class Triangle {
    private double base;
    private double height;

    // Constructor
    public Triangle(double base, double height) {
        this.base = base;
        this.height = height;
    }

    // Method to calculate area
    public double getArea() {
        return 0.5 * base * height;
    }
}
```

File: Circle.java

```
package shape;

public class Circle {
    private double radius;

    // Constructor
    public Circle(double radius) {
        this.radius = radius;
    }

    // Method to calculate area
    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

File: MainProgram.java

```
import shape.Square;
import shape.Triangle;
import shape.Circle;

public class MainProgram {
    public static void main(String[] args) {
        // Create objects of the shapes
        Square square = new Square(5);         // Square with side = 5
        Triangle triangle = new Triangle(4, 3); // Triangle with base=4, height=3
        Circle circle = new Circle(2);         // Circle with radius=2

        // Calculate and display the areas
        System.out.println("Area of Square: " + square.getArea());
        System.out.println("Area of Triangle: " + triangle.getArea());
        System.out.println("Area of Circle: " + circle.getArea());
    }
```

}

Output:
Area of Square: 25.0
Area of Triangle: 6.0
Area of Circle: 12.566370614359172

6. a) Define package. Explain the access protection for class numbers with respect to package.

A **package** in Java is a namespace that organizes classes and interfaces. It helps manage large codebases by grouping related classes together, promoting better modularity and reusability. Packages also prevent naming conflicts between classes by providing a unique namespace for each group.

- Packages are declared using the package keyword at the beginning of a Java file.
- Built-in packages like java.util, java.io, and java.lang provide pre-defined functionality.
- Custom packages can be created by developers to organize their code.

Types of Packages

1. **Built-in Packages**: Provided by Java, e.g., java.util, java.io.
2. **User-defined Packages**: Created by the user to organize their classes, e.g., com.mycompany.utils.

Access Protection in Java with Respect to Packages
Java uses **access modifiers** to control the visibility of classes, methods, and fields. The behavior of access modifiers changes based on whether the elements are in the **same package** or **different packages**.
Access Modifiers

1. **Public**:
   - Accessible everywhere, both within and outside the package.
2. **Protected**:
   - Accessible within the same package.
   - Accessible outside the package only through inheritance.
3. **Default (Package-Private)**:
   - Accessible only within the same package.
   - No modifier is specified (the absence of a modifier implies default access).
4. **Private**:
   - Accessible only within the same class. Not accessible outside the class, even within the same package.

Access Control Table

Modifier	Same Class	Same Package	Subclass (Different Package)	Outside Package
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

Example: Access Protection Across Packages
Package 1: mypackage
Create a file named TestClass.java in a package mypackage.

```
package mypackage;

public class TestClass {
   public int publicField = 10;
   protected int protectedField = 20;
   int defaultField = 30; // Default access
   private int privateField = 40;

   public void displayFields() {
```

```
        System.out.println("Public Field: " + publicField);
        System.out.println("Protected Field: " + protectedField);
        System.out.println("Default Field: " + defaultField);
        System.out.println("Private Field: " + privateField);
    }
}
```

Package 2: anotherpackage

Create another file named AccessTest.java in a package anotherpackage.

```
package anotherpackage;

import mypackage.TestClass;

public class AccessTest extends TestClass {
    public static void main(String[] args) {
        TestClass test = new TestClass();

        System.out.println("Public Field: " + test.publicField); // Accessible
        // System.out.println("Protected Field: " + test.protectedField); // Not accessible directly
        // System.out.println("Default Field: " + test.defaultField); // Not accessible
        // System.out.println("Private Field: " + test.privateField); // Not accessible

        // Accessing protected field through inheritance
        AccessTest access = new AccessTest();
        System.out.println("Protected Field (via inheritance): " + access.protectedField);
    }
}
```

Output

Public Field: 10
Protected Field (via inheritance): 20

6. b) Differentiae abstract class and interface

Feature	Abstract Class	Interface
Definition	A class that cannot be instantiated and may contain both abstract and concrete methods.	A blueprint for a class that contains only abstract methods (before Java 8) and static/final fields.
Keyword	Declared using the abstract keyword.	Declared using the interface keyword.
Method Implementation	Can have abstract methods (without a body) and concrete methods (with a body).	Can have: - Abstract methods (implicitly public and abstract). - Default and static methods (from Java 8).
Inheritance	A class can extend only one abstract class (single inheritance).	A class can implement multiple interfaces (multiple inheritance).
Fields	Can have instance variables (both final and non-final).	Can only have constants (implicitly public, static, and final).
Constructor	Can have a constructor to initialize fields.	Cannot have a constructor (interfaces cannot be instantiated).
Access Modifiers	Methods and fields can have any access modifier (public, protected, private).	Methods are implicitly public. Fields are implicitly public static final.

Feature	Abstract Class	Interface
Purpose	Used when classes share common behavior but may also need to define some unique methods.	Used to define a contract or blueprint for implementing classes.
Default Methods	Allowed (normal methods are used).	Supported from Java 8 using the default keyword.
Static Methods	Can have static methods (with or without a body).	Supported from Java 8 (must have a body).
Examples	Abstract classes like Shape, Animal, etc., share some common behaviors with subclasses.	Interfaces like Comparable, Runnable, and List define specific behaviors.

6. c) Explain with an example, how interfaces can be extended.

Rules for Extending Interfaces

1. An interface can extend one or more interfaces (multiple inheritance is supported).
2. A class that implements the child interface must provide implementations for all methods from both the child and parent interfaces.

Example: Extending Interfaces

Parent Interface

java
Copy code

```
interface Shape {
   double calculateArea();
   double calculatePerimeter();
}
```

Child Interface

java
Copy code

```
interface AdvancedShape extends Shape {
   double calculateVolume(); // New method added by the child interface
}
```

Implementing the Child Interface

A class that implements the AdvancedShape interface must implement methods from both AdvancedShape and Shape.

java
Copy code

```
class Cube implements AdvancedShape {
   double side;

   Cube(double side) {
      this.side = side;
   }

   // Implementing methods from Shape interface
   @Override
   public double calculateArea() {
      return 6 * side * side; // Surface area of a cube
   }

   @Override
   public double calculatePerimeter() {
      return 12 * side; // Total edge length of a cube
   }
```

```
    // Implementing method from AdvancedShape interface
    @Override
    public double calculateVolume() {
        return side * side * side; // Volume of a cube
    }
}
```

Testing the Implementation

```
java
Copy code
public class InterfaceExtensionExample {
    public static void main(String[] args) {
        Cube cube = new Cube(3); // Create a Cube with side length = 3

        System.out.println("Surface Area: " + cube.calculateArea());
        System.out.println("Perimeter: " + cube.calculatePerimeter());
        System.out.println("Volume: " + cube.calculateVolume());
    }
}
```

Output

```
Surface Area: 54.0
Perimeter: 36.0
Volume: 27.0
```

Key Points

1. **Inheritance in Interfaces**:
   - The AdvancedShape interface extends the Shape interface.
   - It inherits the calculateArea and calculatePerimeter methods and defines a new method calculateVolume.
2. **Implementation**:
   - The Cube class implements the AdvancedShape interface.
   - It provides implementations for all inherited and newly added methods.
3. **Multiple Inheritance**:
   - An interface can extend multiple interfaces. For example:

```
java
Copy code
interface ThreeDimensional extends Shape, AdvancedShape {
    void renderIn3D();
}
```

This feature of extending interfaces allows for designing flexible and modular systems in Java.

7. a) What is an exception? Explain the exception handling mechanism with suitable example.

What is an Exception in Java?

An **exception** in Java is an event that disrupts the normal flow of program execution. It usually occurs during runtime and is caused by unexpected conditions like invalid user input, hardware failures, or trying to access a file that doesn't exist.

Exceptions in Java are represented as objects, which are instances of the Throwable class or its subclasses.

Types of Exceptions

1. **Checked Exceptions**:
   - Exceptions checked at compile time.
   - Must be either caught or declared in the method using throws.
   - Example: IOException, SQLException.
2. **Unchecked Exceptions**:
   - Exceptions checked at runtime.
   - Subclasses of RuntimeException.
   - Example: NullPointerException, ArithmeticException.
3. **Errors**:

- Represent serious problems that are not meant to be handled by the application.
- Example: OutOfMemoryError, StackOverflowError.

Exception Handling Mechanism in Java

Java provides a robust mechanism to handle exceptions using five keywords:

- try: Defines a block of code to monitor for exceptions.
- catch: Handles the exception that occurs in the try block.
- finally: Defines a block of code that always executes, regardless of whether an exception occurs or not.
- throw: Used to explicitly throw an exception.
- throws: Declares the exceptions a method can throw.

Example: Exception Handling in Java

Example 1: Handling a Division by Zero Exception

java

Copy code

```
public class ExceptionExample {
   public static void main(String[] args) {
      try {
         int result = 10 / 0; // This will cause ArithmeticException
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: Division by zero is not allowed.");
      } finally {
         System.out.println("Execution completed.");
      }
   }
}
```

Output:

csharp

Copy code

```
Exception caught: Division by zero is not allowed.
Execution completed.
```

Explanation

1. The try block contains code that may throw an exception.
2. The catch block handles the ArithmeticException.
3. The finally block executes regardless of whether an exception occurs.

Example 2: Handling Multiple Exceptions

java

Copy code

```
public class MultipleExceptionExample {
   public static void main(String[] args) {
      try {
         int[] numbers = {1, 2, 3};
         System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException
      } catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Exception caught: Index is out of bounds.");
      } catch (Exception e) {
         System.out.println("General exception caught.");
      } finally {
         System.out.println("Execution completed.");
      }
   }
}
```

Output:

```
Exception caught: Index is out of bounds.
```

Execution completed.

Example 3: Using throw and throws

java

Copy code

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

public class CustomExceptionExample {
    public static void checkNumber(int num) throws CustomException {
        if (num < 0) {
            throw new CustomException("Negative numbers are not allowed.");
        }
        System.out.println("Number is: " + num);
    }

    public static void main(String[] args) {
        try {
            checkNumber(-5); // This will throw a CustomException
        } catch (CustomException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

Exception caught: Negative numbers are not allowed.

Key Points of Exception Handling

1. **Prevents Crashes**:
    - Helps prevent the program from terminating unexpectedly.
2. **Debugging**:
    - Makes it easier to debug runtime errors.
3. **Custom Exceptions**:
    - Developers can create custom exceptions by extending the Exception class.
4. **Best Practices**:
    - Use specific exceptions in catch blocks rather than a generic Exception.
    - Use finally for cleanup operations like closing files or releasing resources.

By using exception handling effectively, Java programs can gracefully recover from errors and maintain stability during runtime.

7. b) Explain how to create your own exceptions. Give an example.

In Java, you can create your own exceptions by defining a custom exception class. This is useful when you need to handle specific error conditions in a way that is more meaningful to your application. Custom exceptions are created by extending the Exception class or the RuntimeException class.

Steps to Create Your Own Exception

1. **Define a Class**:
    - Extend the Exception class for checked exceptions.
    - Extend the RuntimeException class for unchecked exceptions.
2. **Add Constructors**:
    - Include constructors to pass custom error messages or causes.
3. **Throw and Handle the Exception**:
    - Use the throw keyword to throw the custom exception.

- Handle it using try-catch blocks.

Example: Custom Exception
Custom Exception Class

```
class InvalidAgeException extends Exception {
  // Constructor with custom message
  public InvalidAgeException(String message) {
    super(message);
  }
}
```

Using the Custom Exception

```
public class CustomExceptionExample {
  // Method to validate age
  public static void validateAge(int age) throws InvalidAgeException {
    if (age < 18) {
      throw new InvalidAgeException("Age must be 18 or older to register.");
    }
    System.out.println("Registration successful for age: " + age);
  }

  public static void main(String[] args) {
    try {
      validateAge(16); // This will throw an InvalidAgeException
    } catch (InvalidAgeException e) {
      System.out.println("Exception caught: " + e.getMessage());
    }

    try {
      validateAge(20); // This will pass the validation
    } catch (InvalidAgeException e) {
      System.out.println("Exception caught: " + e.getMessage());
    }
  }
}
```

Output

```
Exception caught: Age must be 18 or older to register.
Registration successful for age: 20
```

Explanation

1. **Custom Exception Class**:
   - InvalidAgeException is a custom exception that extends the Exception class.
   - A constructor is defined to accept a custom error message.
2. **Throwing the Exception**:
   - In the validateAge method, the custom exception is thrown if the condition (age < 18) is met.
3. **Handling the Exception**:
   - The try-catch block in main catches and handles the InvalidAgeException.

Unchecked Custom Exception

If you want to create an unchecked custom exception, extend RuntimeException instead of Exception.

Example:

```
class InvalidScoreException extends RuntimeException {
  public InvalidScoreException(String message) {
    super(message);
  }
}
```

```
public class UncheckedExceptionExample {
   public static void checkScore(int score) {
      if (score < 0 || score > 100) {
         throw new InvalidScoreException("Score must be between 0 and 100.");
      }
      System.out.println("Valid score: " + score);
   }

   public static void main(String[] args) {
      checkScore(50); // Valid score
      checkScore(-10); // Will throw InvalidScoreException
   }
}
```

Output

```
Valid score: 50
Exception in thread "main" InvalidScoreException: Score must be between 0 and 100.
```

When to Use Custom Exceptions

- When existing exceptions do not sufficiently describe the error condition.
- To make error handling more specific and meaningful.
- To enforce business logic, such as validating application-specific conditions.

By defining and using custom exceptions, you can build more robust and readable error-handling mechanisms tailored to your application's needs.

8. a) What is checked and unchecked exception? Write a Java program to illustrate nested try catch statement.

In Java, exceptions are divided into two categories: **checked exceptions** and **unchecked exceptions**.

Checked Exceptions

1. **Definition**: Checked exceptions are exceptions that are checked at compile-time. The Java compiler requires the developer to handle these exceptions using try-catch blocks or by declaring them in the method signature using throws.
2. **Examples**:
   - IOException
   - SQLException
   - FileNotFoundException
3. **Characteristics**:
   - Must be explicitly handled in the code.
   - Ensures that the program handles the possibility of failure gracefully.

Unchecked Exceptions

1. **Definition**: Unchecked exceptions are exceptions that occur at runtime. They are subclasses of RuntimeException, and the compiler does not require the developer to handle them.
2. **Examples**:
   - ArithmeticException
   - NullPointerException
   - ArrayIndexOutOfBoundsException
3. **Characteristics**:
   - Not required to be explicitly handled.
   - Usually indicate programming bugs (e.g., accessing a null reference).

Example Program: Nested Try-Catch Statements

Nested try-catch statements are used when one try block is inside another try block. This is useful when a specific portion of the code requires additional exception handling.

Program: Nested Try-Catch

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

```
        try {
            // Outer try block
            int[] numbers = {10, 20, 30};
            System.out.println("Accessing array element: " + numbers[2]);

            try {
                // Inner try block
                int result = 10 / 0; // This will cause ArithmeticException
                System.out.println("Result: " + result);
            } catch (ArithmeticException e) {
                System.out.println("Inner catch: Division by zero is not allowed.");
            }

            System.out.println("Exiting inner try-catch.");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Outer catch: Array index is out of bounds.");
        } finally {
            System.out.println("Outer finally: Program execution completed.");
        }
    }
}
```

Output

Accessing array element: 30
Inner catch: Division by zero is not allowed.
Exiting inner try-catch.
Outer finally: Program execution completed.

Explanation

1. **Outer Try-Catch**:
    - Handles any ArrayIndexOutOfBoundsException that might occur while accessing the array.
2. **Inner Try-Catch**:
    - Handles any ArithmeticException that might occur while performing division.
3. **Finally Block**:
    - Executes regardless of whether an exception occurs or not.
4. **Execution Flow**:
    - First, the outer try block is executed.
    - If an exception occurs in the outer try block, it is handled by the outer catch.
    - If no exception occurs in the outer try block, the inner try block is executed.

This example demonstrates how nested try-catch blocks allow fine-grained control over exception handling for different portions of the code.

8. b) Write a Java program to demonstrate a division by zero exception.

In Java, dividing an integer by zero causes an ArithmeticException, which is an unchecked exception. The following program demonstrates how this exception occurs and how it can be handled gracefully.

Java Code

```
public class DivisionByZeroExample {
    public static void main(String[] args) {
        try {
            // Attempting division by zero
            int numerator = 10;
            int denominator = 0;
            int result = numerator / denominator; // This will throw ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
```

```
        // Handling the exception
        System.out.println("Exception caught: Division by zero is not allowed.");
    } finally {
        // Executing the finally block
        System.out.println("Execution completed.");
    }
  }
}
```

Explanation

1. **Try Block**:
    - The try block contains the code that attempts to divide a number by zero, which raises an ArithmeticException.
2. **Catch Block**:
    - The catch block handles the ArithmeticException and prints an appropriate error message.
3. **Finally Block**:
    - The finally block executes regardless of whether the exception is thrown or not, ensuring cleanup or final steps.

Output

Exception caught: Division by zero is not allowed.
Execution completed.

8. c) Differentiate between throw and throws with example.

Both throw and throws are related to exception handling in Java, but they serve different purposes. Here's a detailed comparison:

Feature	**Throw**	**throws**
Purpose	Used to explicitly throw an exception from within a method or block of code.	Used to declare that a method may throw an exception.
Usage	throw is used to throw an exception explicitly.	throws is used to declare exceptions that a method may throw.
Position	It is used inside a method to throw an exception.	It is used in the method signature to specify possible exceptions.
Exception Type	Can throw any type of exception (checked or unchecked).	Can only be used for **checked exceptions**.
Syntax	throw new ExceptionType("Message");	public void method() throws ExceptionType { }
Required Handling	The thrown exception can be handled by a try-catch block or passed to the caller using throws.	The exceptions declared with throws must either be caught or declared further in the calling methods.

1. throw Example

The throw keyword is used to explicitly throw an exception from a method or a block of code.

Example: Using throw

java
Copy code

```
public class ThrowExample {
  // Method to check age
  public static void checkAge(int age) {
    if (age < 18) {
      throw new IllegalArgumentException("Age must be 18 or older.");
    }
    System.out.println("Age is valid: " + age);
  }
```

```
    public static void main(String[] args) {
        try {
            checkAge(16);  // This will throw an IllegalArgumentException
        } catch (IllegalArgumentException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

Exception caught: Age must be 18 or older.

- In this example, the throw keyword is used to explicitly throw an IllegalArgumentException when the age is less than 18.

2. throws Example

The throws keyword is used in a method declaration to specify that the method may throw one or more exceptions, and the caller of the method must handle them (either by catching or declaring them further).

Example: Using throws

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

public class ThrowsExample {
    // Method that declares it may throw a CustomException
    public static void validateAge(int age) throws CustomException {
        if (age < 18) {
            throw new CustomException("Age must be 18 or older.");
        }
        System.out.println("Age is valid: " + age);
    }

    public static void main(String[] args) {
        try {
            validateAge(16);  // This will throw a CustomException
        } catch (CustomException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}
```

Output:

Exception caught: Age must be 18 or older.

- In this example, the throws keyword is used in the method signature (validateAge), which declares that the method can throw a CustomException. The caller (in this case, main) is responsible for handling the exception with a try-catch block.

9. a) Define A WT. List and explain types of containers in Java AWT

AWT (Abstract Window Toolkit) is a set of APIs provided by Java for building graphical user interfaces (GUIs) in Java applications. AWT provides a way to create and manage user interface components, such as buttons, text fields, and other controls, in a platform-independent manner.

AWT is a part of the java.awt package, and it allows developers to design graphical user interfaces that can work across different platforms. It is a platform-dependent toolkit because it relies on the underlying operating system for rendering GUI components, making the appearance of AWT components vary between platforms.

Types of Containers in AWT

A **container** is a component that can hold other components (called children or sub-components). Containers provide layout and arrangement functionality for organizing the components within them.

List of Containers in Java AWT

1. **Frame** (java.awt.Frame)
2. **Panel** (java.awt.Panel)
3. **Dialog** (java.awt.Dialog)
4. **Window** (java.awt.Window)
5. **Applet** (java.applet.Applet)
6. **ScrollPane** (javax.swing.JScrollPane)

Explanation of Each Container

1. Frame (java.awt.Frame)

- **Description**: A Frame is a top-level window with a title and borders, which can contain other components like buttons, text fields, etc.
- **Usage**: It is used to create the main window of a GUI application.
- **Example**:

```
import java.awt.*;

public class FrameExample {
   public static void main(String[] args) {
      Frame frame = new Frame("AWT Frame Example");
      frame.setSize(400, 300);
      frame.setVisible(true);
   }
}
```

2. Panel (java.awt.Panel)

- **Description**: A Panel is a container that is typically used to organize components inside a window or frame. It can hold multiple components, which are arranged in a layout.
- **Usage**: It is used for grouping components inside a frame or another container. Panels are commonly used when you want to organize components in specific regions of a window.
- **Example**:

```
import java.awt.*;

public class PanelExample {
   public static void main(String[] args) {
      Frame frame = new Frame("Panel Example");
      Panel panel = new Panel();
      panel.add(new Button("Click Me"));
      frame.add(panel);
      frame.setSize(400, 300);
      frame.setVisible(true);
   }
}
```

3. Dialog (java.awt.Dialog)

- **Description**: A Dialog is a pop-up window that is used to get input or show messages to the user. It can be modal (blocking) or non-modal (non-blocking).
- **Usage**: It is used for pop-up windows, where the user interacts with the dialog and then returns to the main window.
- **Example**:

```
import java.awt.*;

public class DialogExample {
   public static void main(String[] args) {
      Frame frame = new Frame("Dialog Example");
      Dialog dialog = new Dialog(frame, "Input Dialog", true);
      dialog.setSize(200, 150);
      dialog.setVisible(true);
```

```
    }
}
```

4. Window (java.awt.Window)

- **Description**: A Window is a top-level container similar to a Frame but without the title bar and border. It is used when you need a simple window that doesn't require a title or other decorations.
- **Usage**: It is used for creating special types of windows or sub-windows that don't require the traditional window features like a title bar.
- **Example**:

```
import java.awt.*;

public class WindowExample {
    public static void main(String[] args) {
        Window window = new Window(new Frame());
        window.setSize(400, 300);
        window.setVisible(true);
    }
}
```

5. Applet (java.applet.Applet)

- **Description**: An Applet is a container used for Java applications that run inside a web browser. Applets are a part of the java.applet package and were used for embedding Java code in web pages, although they are now considered obsolete.
- **Usage**: It is used for small Java applications that need to be embedded inside a browser.
- **Example**:

```
import java.applet.*;
import java.awt.*;

public class AppletExample extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, Applet!", 20, 30);
    }
}
```

6. ScrollPane (javax.swing.JScrollPane)

- **Description**: A ScrollPane is a container that allows a component to be scrolled. It is used when the component's content exceeds the visible area, and a scrollbar is needed to navigate the content.
- **Usage**: It is used to wrap components like TextArea, TextField, Panel, etc., and adds scrollbars automatically when the content size exceeds the visible area.
- **Example**:

```
import javax.swing.*;
import java.awt.*;

public class ScrollPaneExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ScrollPane Example");
        JTextArea textArea = new JTextArea(10, 30);
        JScrollPane scrollPane = new JScrollPane(textArea);
        frame.add(scrollPane);
        frame.setSize(400, 300);
        frame.setVisible(true);
    }
}
```

9. b) Write a Java program to create a window when we press. Mor m the windows display Good Morning A or a the windows dísplay Good Afternoon iii) Eor e the window display Good Evening iv) Nor n the window display Good Night.

```
import java.awt.*;
import java.awt.event.*;
```

```
public class GreetingWindow extends Frame implements KeyListener {

   private Label label;

   // Constructor to set up the Frame and add a Label
   public GreetingWindow() {
      setTitle("Greeting Window");
      setSize(400, 200);
      setLayout(new FlowLayout());

      // Create a label to display messages
      label = new Label("Press M, A, E, or N for greetings", Label.CENTER);
      add(label);

      // Add KeyListener to the frame
      addKeyListener(this);

      // Set the frame visibility
      setVisible(true);

      // Close the window on closing
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent we) {
            System.exit(0);
         }
      });
   }

   // Method to handle key typed events
   public void keyTyped(KeyEvent ke) {
      // No action needed here
   }

   // Method to handle key pressed events
   public void keyPressed(KeyEvent ke) {
      char key = ke.getKeyChar(); // Get the key pressed

      // Display different messages based on the key pressed
      if (key == 'M' || key == 'm') {
         label.setText("Good Morning");
      } else if (key == 'A' || key == 'a') {
         label.setText("Good Afternoon");
      } else if (key == 'E' || key == 'e') {
         label.setText("Good Evening");
      } else if (key == 'N' || key == 'n') {
         label.setText("Good Night");
      } else {
         label.setText("Press M, A, E, or N for greetings");
      }
   }

   // Method to handle key released events
   public void keyReleased(KeyEvent ke) {
      // No action needed here
   }
```

```
    // Main method to run the program
    public static void main(String[] args) {
        new GreetingWindow(); // Create and show the window
    }
}
```

9. c) Write a short note on swings.

Swing is a part of Java's Standard Library (javax.swing package) used to create graphical user interfaces (GUIs). It is an extension of the Abstract Window Toolkit (AWT), offering a richer set of GUI components and providing a more powerful and flexible toolkit for building desktop applications. Swing is completely written in Java, making it platform-independent and more customizable than AWT.

Key Features of Swing:

1. **Lightweight Components**: Swing components are lightweight (i.e., they do not rely on the underlying operating system for rendering), unlike AWT components that are heavyweight. Swing components are written entirely in Java, allowing for more flexibility in appearance and behavior.
2. **Pluggable Look and Feel**: Swing allows the "look and feel" (i.e., the appearance) of components to be changed or customized. You can switch between different looks (e.g., Windows look, Mac look, or custom looks), making Swing applications more visually consistent across platforms.
3. **Rich Set of Controls**: Swing provides a wide range of GUI components such as buttons, checkboxes, text fields, tables, and trees. It also provides more advanced components like JTable for displaying tabular data, JList for displaying lists, and JTree for hierarchical data.
4. **Event Handling**: Swing supports event-driven programming and provides an easy way to handle events like mouse clicks, keyboard inputs, and other user interactions. It uses the listener model for handling events.
5. **Customizable Components**: Swing provides flexibility in customizing the appearance and behavior of components. For example, you can create custom components or modify the properties of standard components, such as changing the background color, font, or adding animations.
6. **Double Buffering**: Swing uses a technique called double buffering, which reduces flickering by drawing the entire component in memory before displaying it to the screen. This ensures smooth graphical rendering, especially in complex UIs.

Commonly Used Swing Components:

1. **JFrame**: A top-level container used to create a window for the application.
2. **JButton**: A button used to trigger actions when clicked.
3. **JLabel**: A component used to display text or images.
4. **JTextField**: A single-line text input field for user input.
5. **JTextArea**: A multi-line text input field.
6. **JComboBox**: A dropdown menu for selecting an option.
7. **JList**: A list of items that can be selected by the user.
8. **JTable**: A table for displaying data in rows and columns.
9. **JScrollPane**: A container that provides scrolling capability for other components.
10. **JPanel**: A container that organizes components within a window.

Example of a Simple Swing Program:

```
import javax.swing.*;

public class SimpleSwingExample {
    public static void main(String[] args) {
        // Create a JFrame (a top-level container)
        JFrame frame = new JFrame("Swing Example");

        // Create a button
        JButton button = new JButton("Click Me!");

        // Add action listener to the button
        button.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Hello, Swing!"));
```

```
        // Add button to the frame
        frame.add(button);

        // Set frame properties
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Explanation:

- This simple Swing program creates a window (JFrame) with a button (JButton).
- When the button is clicked, a message is displayed using a dialog box (JOptionPane).

Advantages of Swing:

1. **Cross-Platform**: Swing components are platform-independent, making them suitable for creating Java applications that run across various operating systems without modification.
2. **Customizable**: Swing allows for greater control over the appearance of the components and the layout of the user interface.
3. **Rich Component Set**: Swing provides a rich set of components that allow for the development of sophisticated UIs, from simple buttons to complex tables and trees.
4. **Integrated Event Handling**: Swing integrates seamlessly with the event-driven programming model, providing various listeners to handle user actions.

Disadvantages of Swing:

1. **Performance**: Swing components can be slower than native components, especially on older machines or complex interfaces.
2. **Complexity**: Creating complex layouts and interfaces with Swing can be more challenging compared to simpler libraries like AWT or newer UI frameworks like JavaFX.
3. **Heavyweight**: Swing components, while lightweight in terms of native resources, may still consume significant CPU and memory, particularly in large applications.