SR 22

St. Peter's Engineering College (Autonomous) Dullapally (P), Medchal, Hyderabad – 500100. QUESTION BANK						Dept.	:	
						Academic Year 2024-25		
Subject Code	:	AS22 05PC02	Subject	:	Oops through Java			
Class/Section	:	B. Tech.	Year	:	II	Semester	:	III

BLOOMS LEVEL					
Remember	L1	Understand	L2	Apply	L3
Analyze	L4	Evaluate	L5	Create	L6

*****

Q. No	Question (s)	Marks	BL	CO
	UNIT - I			
1	**a) Define String.** A string is a series of characters stored in contiguous memory locations. It is terminated by a special character called the **null character**	**1M**	L2	C214.1
	b) List out the Java buzzwords. **1.**Object-Oriented, 2.Distributed, 3.Compiled and Interpreted, 4.Robust, 5.Secure, 6.Architecture-Neutral	**1M**	L1	C214.1
	c) Define Class with a syntax. public class MyClass { // Data members (variables) int myVariable; // Example variable // Constructor (optional) public MyClass() { // Initialize variables or perform other setup } // Methods (functions) public void myMethod() { // Define behavior for this method } }	**1M**	L1	C214.1
	d) Who is inventor of Java? The **Java programming language** was developed by **James Gosling**,	**1M**	L1	C214.1
	e) Define class and object. A class serves as a blueprint or template for creating objects	**1M**	L1	C214.1

	An object is an instance of a class.			
2	**a) Discuss about polymorphism and its types with an example?** Compile-Time Polymorphism (Static Polymorphism): Achieved through function overloading. Function Overloading: When multiple functions have the same name but different parameters (number or type). Example: Java class Helper { static int Multiply(int a, int b) { return a * b; } static double Multiply(double a, double b) { return a * b; } } System.out.println(Helper.Multiply(2, 4)); // Output: 8 System.out.println(Helper.Multiply(5.5, 6.3)); // Output: 34.65 Run-Time Polymorphism (Dynamic Polymorphism): Resolved by the Java Virtual Machine (JVM) during runtime. Achieved through method overriding (when a subclass provides a specific implementation for a method defined in its superclass). Example: Java class Bicycle { void ride() { System.out.println("Riding a bicycle"); } } class MountainBike extends Bicycle { void ride() { System.out.println("Riding a mountain bike"); } } Bicycle bike1 = new MountainBike(); bike1.ride(); // Output: "Riding a mountain bike"	**3M**	L2	C214.1
	b) Discuss about Constructor and its types with an example? Constructor Basics: A constructor in Java is a special method that gets called when an object of a class is created. Constructors have the same name as the class and do not have a return type.	**3M**	L2	C214.1

They initialize the object's state (data members) and perform any necessary setup.
Constructors are essential for creating and initializing objects.
Types of Constructors:
Default Constructor:
A constructor with no parameters.
Automatically provided by Java if no other constructors are defined.
Example:
Java

```
class Person {
    String name;

    // Default constructor (no parameters)
    Person() {
        name = "Unknown";
    }
}
```

Example:
Java

```
class Student {
    String name;
    int age;

    // Parameterized constructor
    Student(String studentName, int studentAge) {
        name = studentName;
        age = studentAge;
    }
}
```

Example:
Java

```
class Book {
    String title;

    // Copy constructor
    Book(Book originalBook) {
        title = originalBook.title;
    }
}

public class Main {
    public static void main(String[] args) {
        // Default constructor
        Person person = new Person();
```

	System.out.println("Person name: " + person.name); // Parameterized constructor Student student = new Student("Alice", 20); System.out.println("Student name: " + student.name + ", Age: " + student.age); // Copy constructor Book originalBook = new Book(); originalBook.title = "Java Programming"; Book copiedBook = new Book(originalBook); System.out.println("Copied book title: " + copiedBook.title); } }			
	c) Discuss the use of class and object in Java with syntax and example? Class: A class serves as a blueprint or template for creating objects. It defines the state (fields or variables) and behavior (methods) that objects of that class will have. Think of a class as a sketch of a house – it describes the structure, but you need to build actual houses (objects) based on that sketch. Syntax for creating a class: Java class ClassName { // Fields (variables) // Methods } Example: Java class Bicycle { private int gear = 5; // State (field) public void braking() { System.out.println("Working of Braking"); // Behavior (method) } } Object: An object is an instance of a class. It represents a specific entity based on the class definition. For example, if Bicycle is a class, MountainBicycle, SportsBicycle, and TouringBicycle can be objects of that class. Creating an object: Java	**3M**	L1	C214.1

	Bicycle sportsBicycle = new Bicycle(); Bicycle touringBicycle = new Bicycle(); AI-generated code. Review and use carefully. More info on FAQ. Here, sportsBicycle and touringBicycle are object names. We use the new keyword along with the class constructor (which has the same name as the class) to create objects. Accessing Members: You can use the object name followed by the dot (.) operator to access fields and methods of the class. Example: Java sportsBicycle.braking(); // Accessing the braking method int gearValue = sportsBicycle.gear; // Accessing the gear field			
	d) Define Data encapsulation in Java with an example? **Data encapsulation** in Java is a fundamental concept in object-oriented programming (OOP). It involves bundling data (instance variables) and methods (functions) that operate on that data within a single unit (class). The goal is to hide the implementation details of a class from outside access and expose a public interface for interaction. Here's an example demonstrating Java encapsulation: **Java** `class Person {` `  private String name;` `  private int age;` `  public String getName() {` `  return name;` `  }` `  public void setName(String name) {` `  this.name = name;` `  }` `  public int getAge() {` `  return age;` `  }` `  public void setAge(int age) {` `  this.age = age;` `  }` `}`	**3M**	L1	C214.1

	`` public class Main { public static void main(String[] args) { Person person = new Person(); person.setName("John"); person.setAge(30); System.out.println("Name: " + person.getName()); System.out.println("Age: " + person.getAge()); } } ```			
	e) List out the various operator supports in Java with an example. f) **Arithmetic Operators**: a. These operators perform common mathematical operations. b. Examples: i. Addition: `int sum = 100 + 50;` (result: `sum` is 150) ii. Subtraction: `int difference = 100 - 50;` (result: `difference` is 50) iii. Multiplication: `int product = 10 * 5;` (result: `product` is 50) iv. Division: `double quotient = 20.0 / 3;` (result: `quotient` is approximately 6.6667) v. Modulus (remainder): `int remainder = 20 % 3;` (result: `remainder` is 2) g) **Assignment Operators**: a. These operators assign values to variables. b. Example: i. `int x = 10;` (assigns the value 10 to variable `x`) c. Other assignment operators include `+=`, `-=` etc. h) **Comparison Operators**: a. Used to compare values or variables. b. Examples: i. Equal to: `boolean isEqual = x == y;` (result: `isEqual` is `true` if `x` equals `y`) ii. Greater than: `boolean isGreater = x > y;` (result: `isGreater` is `true` if `x` is greater than `y`) i) **Logical Operators**: a. Used to determine logic between variables or values. b. Examples:	**3M**	L1	C214.1

	i. Logical AND (`&&`): `boolean result = (x < 5) && (y < 10);` (result: `result` is `true` if both conditions are true) ii. Logical OR (`||`): `boolean result = (x < 5) || (y < 10);` (result: `result` is `true` if at least one condition is true)			
3	**a) Explain in detail about operators with an example.** b) **Arithmetic Operators**: a. These operators perform simple arithmetic operations on primitive data types: i. `+`: Addition ii. `-`: Subtraction iii. `*`: Multiplication iv. `/`: Division v. `%`: Modulo (remainder) b. Example: **Java** `int a = 10;` `int b = 3;` `System.out.println("a + b = " + (a + b)); // Output: 13` `System.out.println("a - b = " + (a - b)); // Output: 7` `System.out.println("a * b = " + (a * b)); // Output: 30` `System.out.println("a / b = " + (a / b)); // Output: 3` `System.out.println("a % b = " + (a % b)); // Output: 1` AI-generated code. Review and use carefully. More info on FAQ. c) **Unary Operators**: a. These operators work with a single operand: i. `-`: Unary minus (negates the value) ii. `+`: Unary plus (indicates positive value) iii. `++`: Increment operator (post-increment and pre-increment) iv. `--`: Decrement operator (post-decrement and pre-decrement) v. `!`: Logical NOT (inverts a boolean value) b. Example: **Java** `int x = 10;`	**5M**	L2	C214.1

	`int y = 10;` `System.out.println("Postincrement: " + (x++)); // Output: 10` `System.out.println("Preincrement: " + (++x)); // Output: 12` `System.out.println("Postdecrement: " + (y--)); // Output: 10` `System.out.println("Predecrement: " + (--y)); // Output: 8` AI-generated code. Review and use carefully. More info on FAQ. d) **Assignment Operator**: a. The = operator assigns a value to a variable. Example: **Java** `int result = 0;` `result = a + b; // Assign the sum of a and b to 'result'`			
	b) What is meant by type casting and its types with a syntax? Type casting in Java refers to converting a value from one data type to another. There are two main types of type casting: **Widening Casting (Automatic)**: This occurs when you convert a smaller data type to a larger one. The order of widening casting is: `byte` → `short` → `char` → `int` → `long` → `float` → `double`. Example: **Java** `int myInt = 9;` `double myDouble = myInt; // Automatic casting: int to double` AI-generated code. Review and use carefully. More info on FAQ. **Narrowing Casting (Manual)**: This involves converting a larger data type to a smaller one. You need to explicitly specify the target type using parentheses. The syntax is: `targetDataType variableName = (targetDataType) originalVariable;`	**5M**	L1	C214.1

	Example: **Java** `double myDouble = 9.78;` `int myInt = (int) myDouble; // Manual casting: double to int` Here's a real-life example where we calculate a user's percentage score in a game using type casting: **Java** `int maxScore = 500;` `int userScore = 423;` `float percentage = (float) userScore / maxScore * 100.0f;` `System.out.println("User's percentage is " + percentage);`			
	c) What is meant by method overloading with an example. **Method overloading** in Java allows you to define multiple methods with the **same name**, but they **differ in parameters** (such as the number or types of parameters). Here's how it works: **Overloading by Changing the Number of Parameters**: You can create overloaded methods with different parameter counts. Example: **Java** `class Calculator {` `  public static int add(int a, int b) {` `  return a + b;` `  }` `  public static double add(double a, double b) {` `  return a + b;` `  }` `}` **Overloading by Changing the Data Type of Parameters**: Overloaded methods can accept different data types. Example: **Java** `class HelperService {`	**5M**	L1	C214.1

	`` private String formatNumber(int value) { return String.format("%d", value); } private String formatNumber(double value) { return String.format("%.3f", value); } // Other overloaded methods... ```			
	e) Explain the characteristics of Java buzzwords. f) **Simple**: a. Java is designed to be easy for both beginners and professional programmers. b. It removes unnecessary complexities, such as explicit pointers and operator overloading. c. If you're familiar with basic Object-Oriented Programming (OOP) concepts, Java's simplicity makes it accessible. g) **Object-Oriented**: a. Java is a true object-oriented programming language. b. All code and data reside within objects and classes. c. The basic OOP concepts—like inheritance, polymorphism, abstraction, and encapsulation—are integral to Java. h) **Distributed**: a. Java enables the creation of distributed applications across networks. b. It allows seamless access to remote objects on the Internet. c. Multiple programmers in different locations can collaborate on a single project. i) **Compiled and Interpreted**: a. Java combines both compilation and interpretation. b. It compiles programs into an intermediate representation called Java Bytecode. c. The Bytecode is then interpreted by the Java Virtual Machine (JVM) to generate machine code. j) **Robust**: a. Java emphasizes reliability and error handling. b. Strict typing checks code at compile time and runtime. c. Memory management is handled through garbage collection.	**5M**	L2	C214.1

	d. Exception handling captures serious errors. k) **Secure**: a. Java provides a "firewall" between networked applications and your computer. b. It confines Java programs to the execution environment, preventing access to other parts of the system.			
	a) Explain the principles of Object Oriented Programming. Object-Oriented Programming (OOP) is a powerful paradigm used in Java and other programming languages. Let's delve into the key principles of OOP in Java: Abstraction: o Definition: Abstraction simplifies complex systems by focusing on essential features while ignoring unnecessary details. o In Java: You create abstract classes or interfaces that define common attributes and methods. These serve as blueprints for creating objects. Encapsulation: o Definition: Encapsulation bundles data (attributes) and methods (functions) into a single unit (class). It protects data by restricting direct access. o In Java: You use access modifiers (public, private, protected) to control visibility. Private instance variables can only be accessed through public methods (getters and setters). Inheritance: o Definition: Inheritance allows creating new classes based on existing ones (parent-child relationship). o In Java: Child classes (subclasses) inherit properties and behaviors from their parent class (superclass). It promotes code reuse and hierarchy. Polymorphism: Definition: Polymorphism enables treating objects of different	**5M**	L2	C214.1

	classes uniformly. o In Java: You achieve polymorphism through method overloading (same method name, different parameters) and method overriding (redefining a method in a subclass).			
4	**a) Explain the concepts of constructor and constructor overloading with an example.** the concepts of constructors and constructor overloading in Java, along with an example: Constructors: A constructor is a special method in a Java class that gets called when an object of that class is created. It initializes the object by setting its initial state (assigning values to instance variables). Constructors have the same name as the class and no return type (not even void). If you don't explicitly define a constructor, Java provides a default no-argument constructor. Constructor Overloading: Constructor overloading allows a class to have multiple constructors with different parameter lists. Each constructor can perform a different task based on the provided arguments. The compiler differentiates these constructors by considering the number and types of parameters. Example: Let's create a Box class with constructor overloading: Java class Box { double width, height, depth; // Constructor with three arguments Box(double w, double h, double d) { width = w; height = h; depth = d; } // Default constructor (no arguments) Box() { width = height = depth = 0; } // Constructor for a cube (one argument) Box(double len) { width = height = depth = len;	**10M**	L2	C214.1

	} double volume() { return width * height * depth; } } public class Test { public static void main(String args[]) { Box mybox1 = new Box(10, 20, 15); Box mybox2 = new Box(); // Default constructor Box mycube = new Box(7); // Cube constructor double vol; vol = mybox1.volume(); System.out.println("Volume of mybox1 is " + vol); vol = mybox2.volume(); System.out.println("Volume of mybox2 is " + vol); vol = mycube.volume(); System.out.println("Volume of mycube is " + vol); } } output: Volume of mybox1 is 3000.0 Volume of mybox2 is 0.0 Volume of mycube is 343.0			
	b) Define string. How to handle strings operations with an example. In Java, a string is a sequence of characters. It represents text data. For example, "hello" is a string containing the characters ‘h’, ‘e’, ‘l’, ‘l’, and ‘o’. Strings are widely used in Java programming for tasks like storing user input, manipulating text, and more. Creating a String: There are two ways to create a string object in Java: String Literal: You can create a string using double quotes. Example: Java String s = "welcome"; When you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the existing instance is returned. Otherwise, a new	**10M**	L2	C214.1

	string instance is created and placed in the pool. Using new Keyword: You can create a string using the new keyword. Example: Java String s = new String("Welcome"); AI-generated code. Review and use carefully. More info on FAQ. In this case, a new string object is created in the normal heap memory, and the literal “Welcome” is placed in the string constant pool. Common String Operations: Concatenation: You can concatenate strings using the + operator or the concat() method. Example: Java String firstName = "John"; String lastName = "Doe"; String fullName = firstName + " " + lastName; Getting a Character by Index: Use the charAt(index) method to retrieve a character at a specific position. Example: Java char firstChar = fullName.charAt(0); // Gets the first character ('J') Replacing Substrings: Use the replace(oldChar, newChar) or replace(oldStr, newStr) methods. Example: Java String modifiedName = fullName.replace("John", "Jane"); Finding Length: Use the length() method to get the length of a string. Example: Java int nameLength = fullName.length(); // Gets the length of the full name Splitting a String: Use the split(delimiter) method to split a string into an array of substrings. Example: Java			

	String[] parts = fullName.split(" "); // Splits by space			
	c) What is meant by array? How can create an array in java with an example. An array in Java is a data structure that allows you to store multiple values of the same type in a single variable. It's like a collection of similar-typed variables accessed by a common name. Here are some key points about Java arrays: Fixed Size: The number of values in a Java array is always fixed when you create it. Once you define the size, it cannot change dynamically. Declaration: To declare an array, use the following syntax: Java dataType[] arrayName; AI-generated code. Review and use carefully. More info on FAQ. dataType can be primitive types (e.g., int, char, double, etc.) or Java objects. arrayName is an identifier. Initialization: You can initialize arrays during declaration: Java int[] age = {12, 4, 5, 2, 5}; AI-generated code. Review and use carefully. More info on FAQ. The Java compiler automatically determines the size based on the number of elements. Accessing Elements: Use the index number to access elements of an array: Java System.out.println("First Element: " + age[0]); System.out.println("Second Element: " + age[1]); // ... AI-generated code. Review and use carefully. More info on FAQ. Example: Creating and Accessing an Array Java public class ArrayExample { public static void main(String[] args) { // Declare and initialize an array int[] numbers = {1, 2, 3, 4, 5}; // Access array elements System.out.println("First Element: " + numbers[0]);	**10M**	L1	C214.1

	System.out.println("Second Element: " + numbers[1]); // ... // Modify an element numbers[2] = 10; System.out.println("Modified Third Element: " + numbers[2]); } } AI-generated code. Review and use carefully. More info on FAQ. Output: First Element: 1 Second Element: 2 Modified Third Element: 10			

Q. No	Question (s)	Marks	BL	CO
	UNIT – II			
1	**a) What is the use of final keyword in Java?**	**1M**	L1	C214.2
	b) Differentiate Method overloading and overriding?	**1M**	L3	C214.2
	c) Define Package?	**1M**	L1	C214.2
	d) List out any two benefits of Inheritance.	**1M**	L1	C214.2
	e) List out the forms of Inheritance.	**1M**	L1	C214.1
2	**a) Explain about Access Modifier?**	**3M**	L1	C214.2
	b) Explain about single and multiple-level inheritance with an example?	**3M**	L2	C214.2
	c) List out the benefits and costs of Inheritance in Java?	**3M**	L2	C214.2
	d) Explain about generalization in Java?	**3M**	L2	C214.2
	e) Write a Java program to define final with inheritance concept?	**3M**	L1	C214.2
3	**a) Define Interface. How to extending the interface with an example?**	**5M**	L1	C214.2
	b) What is the difference between Interface and Class?	**5M**	L1	C214.2
	c) How to access package from another package?	**5M**	L2	C214.2
	d) What is the use of Super keyword? Give an example?	**5M**	L1	C214.2
	e) What is Abstraction in Java?	**5M**	L1	C214.2
4	**a) Explain in detail about abstract class with an example.**	**10M**	L2	C214.2
	b) What are the different forms of inheritance with an	**10M**	L1	C214.2

	example?			
	c) Explain in detail about polymorphism and its types.	**10M**	L2	C214.2

1. What is the use of final keyword in Java? 1m

a) The `final` keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override).

b) The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

c) The `final` keyword is called a "modifier".

2. Differentiate Method overloading and overriding? 1m

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
Method overloading helps to increase the readability of the program.	Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.

3. Define Package? 1m

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee

Making searching/locating and usage of classes, interfaces, enumerations and annotations easierProviding controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.

4. List out any two benefits of Inheritance. 1m

Inheritance in object-oriented programming (OOP) provides several benefits that help in code organization and reuse. Here are two key benefits:

1. **Code Reusability**:
   - **Description**: Inheritance allows you to create new classes that reuse the code of existing classes. This means you can define common functionality in a base class and extend it in derived classes without having to duplicate code.
   - **Example**: If you have a base class `Animal` with a method `makeSound()`, you can create derived classes like `Dog` and `Cat` that inherit from `Animal` and use the `makeSound()` method. This avoids the need to rewrite the `makeSound()` method for each derived class.
2. **Enhanced Maintainability**:
   - **Description**: Inheritance promotes a hierarchical organization of code, which makes it easier to maintain and update. Changes made to the base class automatically propagate to all derived classes, provided the changes don't break the contract established by the base class.
   - **Example**: If you update the `Animal` class to add a new method or modify an existing method, all classes derived from `Animal` will inherit these changes. This centralized update reduces the risk of inconsistencies and ensures that all derived classes benefit from improvements or bug fixes in the base class

5. List out the forms of Inheritance 1m

Single Inheritance: A class inherits from only one base class. This is the simplest form of inheritance.

Multiple Inheritances: A class inherits from more than one base class. This form of inheritance is not directly supported in Java due to ambiguity issues, but can be achieved through interfaces.

Multilevel Inheritance: A class inherits from a derived class, creating a chain of inheritance. This means a class acts as a base class for another class, which in turn acts as a base class for yet another class.

Hierarchical Inheritance: Multiple classes inherit from a single base class. This allows multiple subclasses to share common functionality from the base class.

Hybrid Inheritance :A combination of two or more types of inheritance. This can be complex and is not directly supported in some languages like Java due to potential ambiguity and complexity.

6. **Explain about Access Modifier? 3m**

access modifiers control the visibility and accessibility of classes, methods, and variables. They define how and where the members of a class (such as methods and fields) can be accessed from other classes.

Public: modifier allows the class, method, or field to be accessible from any other class in any package.

Protected: modifier allows access to the class members from within the same package and from subclasses (even if they are in different packages). However, it does not allow access from non-subclass classes in other packages.

Private: modifier restricts access to the class members so that they are only accessible within the same class. This is the most restrictive access level.

Default : If no access modifier is specified, the default access level is applied. This level is also known as package-private. Members with default access are accessible only within the same package.

7. **Explain about single and multiple inheritances with an example? 3m**

Single Inheritance: A class inherits from only one base class. This is the simplest form of inheritance.

```
class Animal {
   void eat() {
      System.out.println("This animal eats food.");
   }
}
class Dog extends Animal {
   void bark() {
      System.out.println("The dog barks.");
   }
class singleint{
public static void main(String agrs[]){
Dog d=new Dog();
d.bark();
}
```

Multiple Inheritances: A class inherits from more than one base class. This form of inheritance is not directly supported in Java due to ambiguity issues, but can be achieved through interfaces

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

```
class Dog implements Animal, Pet {
   public void eat() {
      System.out.println("The dog eats.");
   }

   public void play() {
      System.out.println("The dog plays.");
   }
}
```

8. **List out the benefits and costs of Inheritance in Java? 3m**

Benefits of Inheritance:

1. **Code Reusability**
2. **Ease of Maintenance**:
3. **Hierarchical Classification**:
4. **Polymorphism**:
5. **Encapsulation**:

Costs of Inheritance:

1. Tight Coupling
2. Increased Complexity
3. Fragile Base Class Problem
4. Inheritance Overuse
5. Performance Overheads

9. **Explain about generalization in Java? 3m**

generalization refers to the concept of designing a more general or abstract class from which more specific subclasses can be derived. This process is a key aspect of object-oriented design and plays a crucial role in abstraction, code reusability, and hierarchy management. Here's an in-depth look at generalization:

Key Aspects of Generalization:

1. Abstract Classes and Methods
2. **Super class and Subclass Relationships**:
3. **Polymorphism**:
4. **Code Reusability**:
5. Design Flexibility

10. **Write a Java program to define final with inheritance concept? 3M**

the `final` keyword can be used with classes, methods, and variables, and it plays a significant role in inheritance. Here's how it applies to each:

- **`final` class**: A class declared as `final` cannot be subclassed. This means no other class can extend a `final` class.

- **`final` method**: A method declared as `final` cannot be overridden by subclasses. This means that the behavior of the method is fixed and cannot be changed.
- **`final` variable**: A variable declared as `final` can only be assigned once. This makes the variable a constant

```
// Define a final class

final class Vehicle {

    // Final variable

    private final String brand;

    // Constructor to initialize the final variable

    Vehicle(String brand) {

        this.brand = brand;

    }

    // Final method

    final void displayInfo() {

        System.out.println("Vehicle brand: " + brand);

    }

    // Getter for brand

    public String getBrand() {

        return brand;

    }

}

// Attempt to extend the final class (This will cause a compile-time error)

// class Car extends Vehicle {

//     // This code will not compile because Vehicle is final

// }
```

```
public class Main {

    public static void main(String[] args) {

        // Create an instance of the final class Vehicle

        Vehicle myVehicle = new Vehicle("Toyota");



        // Display vehicle information

        myVehicle.displayInfo();

        // Accessing final variable through getter method

        System.out.println("Brand accessed through getter: " + myVehicle.getBrand());

    }

}
```

11. Define Interface. How to extending the interface with an example? 5M

interface is a reference type that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces are used to specify a set of methods that a class must implement, without providing the method implementations. Interfaces are a way to achieve abstraction and multiple inheritance in Java.

Key Points About Interfaces

1. **Method Signatures**: An interface defines a contract by specifying method signatures (method names, return types, and parameters) but does not provide implementations for these methods.
2. **Implementation**: A class implements an interface and provides concrete implementations for all of its methods.
3. **Multiple Inheritance**: Interfaces support multiple inheritance, meaning a class can implement multiple interfaces.
4. **Default Methods**: Java 8 introduced default methods, which allow interfaces to provide method implementations.
5. **Static Methods**: Interfaces can also have static methods with implementations.

Extending Interfaces

When you extend an interface, you create a new interface that inherits the methods of the parent interface. An interface can extend multiple other interfaces. This is similar to extending classes but involves multiple inheritance of method signatures.

Example of Extending Interfaces

Here's a detailed example to illustrate how to define and extend interfaces in Java:

```
// Define the base interface
interface Animal {
    void eat(); // Abstract method
    void sleep(); // Abstract method
}

// Define another interface that extends the base interface
interface DomesticAnimal extends Animal {
    void play(); // Additional method
}

// Implement the extended interface in a class
class Dog implements DomesticAnimal {
    @Override
    public void eat() {
        System.out.println("The dog eats.");
    }

    @Override
    public void sleep() {
        System.out.println("The dog sleeps.");
    }

    @Override
    public void play() {
        System.out.println("The dog plays.");
    }
}

// Main class to test the implementation
public class Main {
    public static void main(String[] args) {
        // Create an instance of Dog
        DomesticAnimal myDog = new Dog();

        // Call methods implemented in the Dog class
        myDog.eat();
        myDog.sleep();
        myDog.play();
    }
}
```

Output: The dog eats. The dog sleeps. The dog plays

12. What is the difference between Interface and Class? 5M

Class	Interface

Class	Interface
The keyword used to create a class is “class”	The keyword used to create an interface is “interface”
A class can be instantiated i.e., objects of a class can be created.	An Interface cannot be instantiated i.e. objects cannot be created.
Classes do not support multiple inheritance.	The interface supports multiple inheritances.
It can be inherited from another class.	It cannot inherit a class.
It can be inherited by another class using the keyword ‘extends’.	It can be inherited by a class by using the keyword ‘implements’ and it can be inherited by an interface using the keyword ‘extends’.
It can contain constructors.	It cannot contain constructors.
It cannot contain abstract methods.	It contains abstract methods only.

13. How to access package from another package? 5m

Accessing a package from another package in Java involves importing the classes or interfaces from the desired package into the class where you want to use them. Here are the general steps to achieve this in Java:

1. **Structure Your Project**: Ensure your project is structured correctly. For example:

```
project/
├── src/
│   ├── package1/
│   │   ├── MyClass1.java
│   ├── package2/
│   │   ├── MyClass2.java
```

2. **Declare the Packages**: In your Java files, declare the packages at the top of each file.

```
// Inside src/package1/MyClass1.java
package package1;

public class MyClass1 {
```

```
    public void display() {
        System.out.println("Hello from MyClass1 in package1");
    }
}

// Inside src/package2/MyClass2.java
package package2;

import package1.MyClass1;

public class MyClass2 {
    public static void main(String[] args) {
        MyClass1 obj = new MyClass1();
        obj.display();
    }
}
```

3.Compile the Packages: Compile the packages using the Java compiler (`javac`). Navigate to the `src` directory and compile the classes:

```
javac package1/MyClass1.java
javac package2/MyClass2.java
```

4.Run the Program: Run the program by specifying the fully qualified name of the class with the `java` command:

```
java package2.MyClass2
```

14. What is the use of Super keyword? Give an example? 5m

The `super` keyword in Java is used to refer to the immediate parent class object. It can be used for various purposes, such as calling parent class methods, accessing parent class constructors, and accessing parent class fields when they are hidden by subclasses.

Here are the primary uses of the `super` keyword:

1. **Accessing Parent Class Constructor**: The `super()` call can be used to invoke the constructor of the parent class.
2. **Accessing Parent Class Methods**: The `super` keyword can be used to call methods of the parent class that are overridden in the subclass.
3. **Accessing Parent Class Fields**: The `super` keyword can be used to access fields of the parent class when they are hidden by fields in the subclass.

```
// Parent class
class Animal {
    String name;

    // Constructor of parent class
    Animal(String name) {
        this.name = name;
    }
```

```
    // Method in parent class
    void display() {
        System.out.println("I am an animal. My name is " + name);
    }
}

// Child class
class Dog extends Animal {
    String breed;

    // Constructor of child class
    Dog(String name, String breed) {
        // Call the constructor of the parent class
        super(name);
        this.breed = breed;
    }

    // Method in child class
    void display() {
        // Call the display method of the parent class
        super.display();
        System.out.println("I am a dog. My breed is " + breed);
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", "Golden Retriever");
        dog.display();
    }
}
```

15. What is Abstraction in Java? 5m

Abstraction in Java is a fundamental concept of object-oriented programming that involves hiding the complex implementation details of a system and exposing only the necessary and relevant parts to the user. It helps in reducing complexity and allows the programmer to focus on interacting with the objects at a higher level.

Key Points about Abstraction in Java:

1. **Abstract Classes**: These are classes that cannot be instantiated on their own and can include abstract methods (methods without a body) that must be implemented by subclasses.

2. **Interfaces**: These are a collection of abstract methods that any class can implement, providing a way to achieve full abstraction (as interfaces cannot have any method implementations in Java 7 and earlier; Java 8 introduced default and static methods).

```
abstract class Animal {
    // Abstract method (does not have a body)
    abstract void sound();

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

class Dog extends Animal {
    // Provide implementation of the abstract method
    void sound() {
        System.out.println("Woof");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // Output: Woof
        dog.sleep(); // Output: This animal is sleeping.
    }
}
```

16. Explain in detail about abstract class with an example 10m

An abstract class in Java is a class that cannot be instantiated directly and is intended to be subclassed. It can contain abstract methods (methods without an implementation) as well as concrete methods (methods with an implementation). The abstract methods must be implemented by subclasses, ensuring that the subclasses provide specific behaviors while the abstract class can provide common functionality.

Key Points about Abstract Classes

1. **Cannot be Instantiated**: You cannot create an instance of an abstract class directly. It must be subclassed.
2. **Can Have Abstract Methods**: These methods do not have a body and must be implemented by the subclasses.
3. **Can Have Concrete Methods**: These methods have an implementation and can be inherited by subclasses.
4. **Can Have Constructors**: Even though you cannot instantiate an abstract class, you can have constructors in it, which are called when a subclass is instantiated.

Example

Here is a detailed example to illustrate the concept of abstract classes:

```
// Abstract class
abstract class Animal {
    String name;

    // Constructor
    Animal(String name) {
        this.name = name;
    }

    // Abstract method (no implementation)
    abstract void makeSound();

    // Concrete method
    void sleep() {
        System.out.println(name + " is sleeping.");
    }
}

// Subclass of Animal
class Dog extends Animal {
    // Constructor
    Dog(String name) {
        super(name);
    }

    // Implementation of the abstract method
    @Override
    void makeSound() {
        System.out.println(name + " says: Woof!");
    }
}

// Another subclass of Animal
class Cat extends Animal {
    // Constructor
    Cat(String name) {
        super(name);
    }

    // Implementation of the abstract method
    @Override
    void makeSound() {
        System.out.println(name + " says: Meow!");
    }
}

// Main class to test the abstract class and its subclasses
public class Main {
    public static void main(String[] args) {
        // Create instances of Dog and Cat
        Dog dog = new Dog("Buddy");
        Cat cat = new Cat("Whiskers");

        // Call the methods
        dog.makeSound(); // Output: Buddy says: Woof!
        dog.sleep();     // Output: Buddy is sleeping.
```

```
        cat.makeSound(); // Output: Whiskers says: Meow!
        cat.sleep();     // Output: Whiskers is sleeping.
    }
}
```

17. What are the different forms of inheritance with an example? 10m

nheritance in Java is a key aspect of object-oriented programming, allowing classes to inherit properties and methods from other classes. Here are the different forms of inheritance in Java along with examples:

1. **Single Inheritance**: A class inherits from one base class.

```
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: Bark
    }
}
```

2. **Multilevel Inheritance**: A class inherits from a derived class, forming a chain.

```
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}

class Puppy extends Dog {
    void sound() {
```

```
        System.out.println("Yap");
    }
}

public class Main {
    public static void main(String[] args) {
        Puppy puppy = new Puppy();
        puppy.sound();  // Output: Yap
    }
}
```

3. **Hierarchical Inheritance**: Multiple classes inherit from a single base class.

```
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        dog.sound();  // Output: Bark
        cat.sound();  // Output: Meow
    }
}
```

4. **Multiple Inheritance (using interfaces)**: Java does not support multiple inheritance with classes to avoid the diamond problem. However, multiple inheritance is possible using interfaces.

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

interface Vehicle {
    void move();
}
```

```
class RobotDog implements Animal, Vehicle {
    public void sound() {
        System.out.println("Beep");
    }

    public void move() {
        System.out.println("Rolling");
    }
}

public class Main {
    public static void main(String[] args) {
        RobotDog robotDog = new RobotDog();
        robotDog.sound();  // Output: Beep
        robotDog.move();   // Output: Rolling
    }
}
```

5. **Hybrid Inheritance**: Java does not directly support hybrid inheritance due to the lack of support for multiple inheritance with classes. However, a combination of interfaces and classes can achieve similar results.

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

interface Fish {
    void swim();
}

class Bird implements Animal {
    public void sound() {
        System.out.println("Chirp");
    }
}

class FlyingFish extends Bird implements Fish {
    public void swim() {
        System.out.println("Swimming");
    }
}

public class Main {
    public static void main(String[] args) {
        FlyingFish flyingFish = new FlyingFish();
        flyingFish.sound();  // Output: Chirp
        flyingFish.swim();   // Output: Swimming
    }
}
```

18. Explain in detail about polymorphism and its types. 10m

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common super class. It enables one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

There are two main types of polymorphism in OOP: **Compile-time polymorphism** and **Run-time polymorphism**.

1. Compile-time Polymorphism (Static Binding or Method Overloading)

Compile-time polymorphism is achieved through method overloading and operator overloading. This type of polymorphism is resolved during compile time.

Method Overloading: Method overloading occurs when multiple methods in the same class have the same name but different parameters (different type or number of parameters).

```
class MathOperation {
    // Method with 2 int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        MathOperation operation = new MathOperation();
        System.out.println(operation.add(2, 3));        // Output: 5
        System.out.println(operation.add(2, 3, 4));     // Output: 9
        System.out.println(operation.add(2.5, 3.5));    // Output: 6.0
    }
}
```

2.Run-time Polymorphism (Dynamic Binding or Method Overriding)

Run-time polymorphism is achieved through method overriding. This type of polymorphism is resolved during run time.

Method Overriding: Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass should have the same name, return type, and parameters as the method in the superclass.

```
class Animal {

    void sound() {

        System.out.println("Animal makes a sound");

    }

}

class Dog extends Animal {

    @Override

    void sound() {

        System.out.println("Dog barks");

    }

}

class Cat extends Animal {

    @Override

    void sound() {

        System.out.println("Cat meows");

    }

}



public class Main {

    public static void main(String[] args) {

        Animal myAnimal = new Animal();
```

```
        Animal myDog = new Dog();
        Animal myCat = new Cat();


        myAnimal.sound();  // Output: Animal makes a sound
        myDog.sound();     // Output: Dog barks
        myCat.sound();     // Output: Cat meows
    }
}
```

Types of Polymorphism

1. **Compile-time Polymorphism (Static Binding or Method Overloading)**
   - Resolved during compile time.
   - Achieved through method overloading.
   - Multiple methods with the same name but different signatures (parameter list).
2. **Run-time Polymorphism (Dynamic Binding or Method Overriding)**
   - Resolved during runtime.
   - Achieved through method overriding.
   - Methods in subclass override methods in superclass with the same signature.

Q. No	Question (s)	Marks	BL	CO
	UNIT – III			
1	**a) What is Exception handling?**	**1M**	L1	C214.3
	b) What are all the keywords required for defining exception handling?	**1M**	L1	C214.3
	c) Name one method from the String class that is used to concatenate strings?	**1M**	L1	C225.3
	d) List out the some pre-defined exceptions?	**1M**	L1	C214.3
	e) Write the syntax of try block?	**1M**	L1	C214.1
2	**a) How the exceptions are handled in exception handling with an example?**	**3M**	L1	C214.3
	b) Write a Java program with nested try block in the exception handling?	**3M**	L1	C214.3
	c) Explain Arithmetic Exception, IO Exception, and NullPointer Exception, with syntax and example	**3M**	L2	C214.3

	d) How finally block used in the Java application?	**3M**	L1	C214.3
	e) What is meant by runtime exception?	**3M**	L1	C214.3
3	**a) What is meant by exception handling?**	**5M**	L1	C214.3
	b) How to implement the nested try block in the exception handling mechanism?	**5M**	L2	C214.3
	c) How to implement the nested try block in the exception handling mechanism with explanation?	**5M**	L2	C214.3
	d) Differentiate between Multitasking and Multithreading?	**5M**	L2	C214.3
	e) Explain String Handling with proper example?	**5M**	L2	C214.3
4	**a)What is package? Explain about built-in packages in java?**	**10M**	L3	C214.3
	b) Explain the thread life cycle in detail?	**10M**	L2	C214.3
	c) Explain the Java thread by Extending thread class and implementing Runnable interfaces with an example?	**10M**	L2	C214.3

a) What is Exception handling?

Ans: Exception handling is a programming technique used to manage errors and exceptional conditions that occur during the execution of a program. It allows a program to continue running or terminate gracefully instead of crashing. This is achieved by using constructs like try, catch, throw, throws, and finally to detect and handle exceptions in a controlled manner.

b) What are all the keywords required for defining exception handling?

Ans:

The keywords required for defining exception handling in Java are:

try: Used to specify a block of code that might throw an exception.

catch: Used to handle the exception that occurs in the associated try block.

finally: Used to define a block of code that will always be executed after the try block, regardless of whether an exception was thrown or not.

throw: Used to explicitly throw an exception.

throws: Used in method signatures to declare that a method can throw exceptions.

c) Name one method from the String class that is used to concatenate strings?

Ans: One method from the String class used to concatenate strings is concat(). For example, str1.concat(str2) joins str2 to the end of str1.

d) List out the some pre-defined exceptions?

Ans: Any three are sufficient :

Here are some pre-defined exceptions in Java:

ArithmeticException: Thrown when an arithmetic operation, such as division by zero, occurs.

ArrayIndexOutOfBoundsException: Thrown when an array has been accessed with an illegal index.

NullPointerException: Thrown when an application attempts to use null in a case where an object is required.

ClassCastException: Thrown when an attempt is made to cast an object to a subclass of which it is not an instance.

IllegalArgumentException: Thrown to indicate that a method has been passed an illegal or inappropriate argument.

IllegalStateException: Thrown to signal that a method has been invoked at an illegal or inappropriate time.
IndexOutOfBoundsException: Thrown to indicate that an index of some sort is out of range.
NumberFormatException: Thrown to indicate that an attempt to convert a string to a numeric type failed.
IOException: Thrown when an I/O operation has failed or been interrupted.
FileNotFoundException: Thrown when an attempt to open the file denoted by a specified pathname has failed.
e) Write the syntax of try block?
Ans:
```
try {
   // Code that might throw an exception
} catch (ExceptionType1 e1) {
   // Code to handle ExceptionType1
} catch (ExceptionType2 e2) {
   // Code to handle ExceptionType2
} finally {
   // Code that will always execute, regardless of whether an exception is thrown or not
}
```
Explanation:

- The try block contains code that might throw an exception.
- The catch blocks handle specific exceptions that are thrown in the try block.
- The finally block contains code that executes after the try block, regardless of whether an exception was thrown or caught. The finally block is optional.

Section B

a) How the exceptions are handled in exception handling with an example?
Ans:
Exceptions in Java are handled using a combination of try, catch, and optionally finally blocks. Here's an example to illustrate how exceptions are handled:

```
public class ExceptionHandlingExample {
   public static void main(String[] args) {
      try {
         // Code that may throw an exception
         int[] numbers = {1, 2, 3};
         System.out.println("The fourth number is: " + numbers[3]);
      } catch (ArrayIndexOutOfBoundsException e) {
         // Code to handle the exception
         System.out.println("An error occurred: " + e.getMessage());
      } finally {
         // Code that will always execute
         System.out.println("This is the finally block.");
      }

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

Here, when the exception is thrown, the control transfers to the catch block, the error message is printed, and then the finally block executes. The program then continues executing the code after the try-catch-finally construct.

b) Write a Java program with nested try block in the exception handling?

Ans:
```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            // Outer try block
            System.out.println("Outer try block started.");

            try {
                // Inner try block
                System.out.println("Inner try block started.");
                int result = 10 / 0; // This will throw ArithmeticException
                System.out.println("This line will not be executed.");
            } catch (ArithmeticException e) {
                // Inner catch block
                System.out.println("Inner catch block: " + e.getMessage());
            } finally {
                // Inner finally block
                System.out.println("Inner finally block executed.");
            }

            // Code after inner try-catch-finally
            System.out.println("Code after inner try-catch-finally.");

        } catch (Exception e) {
            // Outer catch block
            System.out.println("Outer catch block: " + e.getMessage());
        } finally {
            // Outer finally block
            System.out.println("Outer finally block executed.");
        }

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

Explanation:

Outer try block: Starts execution of the outer block of code.

Inner try block: Nested within the outer try block. It contains code that may throw an exception. In this case, int result = 10 / 0; will throw an ArithmeticException.

Inner catch block: Catches exceptions thrown by the inner try block. It handles the ArithmeticException and prints an appropriate message.

Inner finally block: Executes after the inner try and catch blocks, regardless of whether an exception was thrown or caught.

Outer catch block: Catches any exceptions that are not handled by the inner catch block. In this example, it won’t catch anything since the inner catch block already handled the exception.

Outer finally block: Executes after the outer try block, regardless of whether an exception was thrown or not.
c) Explain Arithmetic Exception, IO Exception, and NullPointer Exception, with syntax and example?
Ans: explanation of ArithmeticException, IOException, and NullPointerException with syntax and examples:
1. ArithmeticException
Description: This exception is thrown when an exceptional arithmetic condition occurs, such as division by zero.
Example:

```
public class ArithmeticExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Caught an ArithmeticException: " + e.getMessage());
        }
    }
```

IOException:

```
import java.io.FileReader;
import java.io.IOException;

public class IOExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("nonexistentfile.txt"); // This will throw IOException
        } catch (IOException e) {
            System.out.println("Caught an IOException: " + e.getMessage());
        }
    }
}
```

NullPointerException:

```
public class NullPointerExceptionExample {
    public static void main(String[] args) {
        String str = null;
        try {
            int length = str.length(); // This will throw NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Caught a NullPointerException: " + e.getMessage());
        }
    }
}
```

d) How finally block used in the Java application?
Ans: In Java, the finally block is used to ensure that a certain piece of code is always executed, regardless of whether an exception was thrown or not. This is useful for cleanup activities such as closing files, releasing resources, or performing any other necessary final steps.
Synatx:

```
try {
    // Code that may throw an exception
```

```
} catch (ExceptionType e) {
   // Code to handle the exception
} finally {
   // Code that will always execute
}
```

Point to remember:

1. Always Executes: The finally block will execute whether an exception is thrown or not. Even if the try or catch block contains a return statement, the finally block will still be executed.
2. Resource Cleanup: It is commonly used for closing resources like files or database connections that need to be closed regardless of whether an exception occurred.
3. Optional: The finally block is optional. If it is not included, you simply omit it.

e) What is meant by runtime exception?

Ans: A runtime exception in Java refers to exceptions that are thrown during the execution of a program, as opposed to compile-time exceptions that are checked by the compiler. These are a type of unchecked exceptions, meaning they are not required to be caught or declared in the method signature.

Key Characteristics:

Unchecked Exceptions: Runtime exceptions are a subclass of RuntimeException, which is itself a subclass of Exception. They do not need to be declared in the throws clause of a method or be caught in a try-catch block.

Occurrence: These exceptions occur at runtime, often due to programming bugs or issues with the code logic, such as trying to access an array element with an invalid index or performing arithmetic operations with invalid values.

Common Examples:

NullPointerException: Thrown when an application attempts to use a null reference where an object is required.
ArrayIndexOutOfBoundsException: Thrown when an array has been accessed with an illegal index.
ArithmeticException: Thrown when an arithmetic operation, such as division by zero, occurs.
ClassCastException: Thrown when an attempt is made to cast an object to a subclass of which it is not an instance.

Example:

```
public class RuntimeExceptionExample {
   public static void main(String[] args) {
      try {
         // Example of NullPointerException
         String str = null;
         System.out.println(str.length()); // This will throw NullPointerException

         // Example of ArrayIndexOutOfBoundsException
         int[] numbers = {1, 2, 3};
         System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException
      } catch (Exception e) {
         // Catching all exceptions (including runtime exceptions)
```

```
            System.out.println("Caught an exception: " + e.getMessage());
        }

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

Section -C

a) What is meant by exception handling with proper example with program?
Ans: Definition:

Exception handling in Java is a mechanism to manage runtime errors, allowing a program to continue executing or terminate gracefully instead of crashing. It involves using specific constructs to detect, catch, and handle exceptions that occur during the execution of a program. The primary constructs used are try, catch, finally, and throw.

Components of Exception Handling:

try Block: Contains code that may throw an exception. It is used to specify a block of code in which exceptions might occur.
catch Block: Handles the exception thrown by the try block. It catches the exception and provides a response or message.
finally Block: Contains code that will always execute, regardless of whether an exception was thrown or caught. It's typically used for cleanup activities.
throw Statement: Used to explicitly throw an exception.
throws Keyword: Declares that a method can throw one or more exceptions. It is used in the method signature.
Example Program:

```
public class ExceptionHandlingDemo {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};

        try {
            // Code that may throw exceptions
            System.out.println("Accessing element at index 2: " + numbers[2]); // Valid access

            System.out.println("Accessing element at index 5: " + numbers[5]); // This will throw
ArrayIndexOutOfBoundsException

            int result = 10 / 0; // This will throw ArithmeticException

        } catch (ArrayIndexOutOfBoundsException e) {
            // Handling ArrayIndexOutOfBoundsException
            System.out.println("Caught ArrayIndexOutOfBoundsException: " + e.getMessage());
        } catch (ArithmeticException e) {
            // Handling ArithmeticException
            System.out.println("Caught ArithmeticException: " + e.getMessage());
        } finally {
            // Code that will always execute
```

```
            System.out.println("This is the finally block. It executes regardless of whether an
exception was thrown or not.");
        }

        System.out.println("Program continues executing after exception handling.");
    }
}
```

Here is a Java program demonstrating exception handling by handling different types of exceptions including ArithmeticException, ArrayIndexOutOfBoundsException, and using the finally block for resource cleanup.
b) How to implement the nested try block in the exception handling mechanism?
Ans: In Java, nested try blocks allow you to handle exceptions at multiple levels, providing more granular control over error handling. You can nest try blocks within each other to catch and handle exceptions at different levels of the code.

Here's how to implement nested try blocks in the exception handling mechanism:

Structure of Nested try Blocks
Outer try Block: The outermost try block surrounds a section of code that might throw an exception.
Inner try Block: Nested within the outer try block, this block contains code that may also throw exceptions. The inner try block can have its own catch blocks and finally block.
Catch Blocks: Each try block can have its own catch blocks to handle exceptions thrown by the respective try block.
Finally Blocks: Each try block can have its own finally block, which will execute regardless of whether an exception was thrown or caught.
Give one example:---
c) Explain Daemon Thread with an example?
Ans: Daemon Threads in Java are special types of threads that run in the background and do not prevent the JVM from exiting. They are typically used for tasks that should run continuously in the background, such as garbage collection or monitoring services. The primary characteristic of daemon threads is that they do not block the termination of the JVM when all non-daemon threads have finished execution.

Key Characteristics of Daemon Threads:
Background Tasks: Daemon threads are often used for background tasks that need to run as long as the application is running but should not prevent the application from shutting down.
JVM Termination: The JVM will exit when all non-daemon threads have finished execution, regardless of whether daemon threads are still running.
Set Daemon Status: Threads can be set to daemon status using the setDaemon(true) method before they are started.
Example:
```
public class DaemonThreadExample {
    public static void main(String[] args) {
        // Creating a non-daemon thread
        Thread mainThread = Thread.currentThread();
        System.out.println("Main thread: " + mainThread.getName());
```

```
        // Creating a daemon thread
        Thread daemonThread = new Thread(() -> {
            try {
                while (true) {
                    System.out.println("Daemon thread running...");
                    Thread.sleep(1000); // Sleep for 1 second
                }
            } catch (InterruptedException e) {
                System.out.println("Daemon thread interrupted.");
            }
        });

        // Set daemon status to true
        daemonThread.setDaemon(true);

        // Start the daemon thread
        daemonThread.start();

        // Main thread sleeps for 5 seconds and then exits
        try {
            Thread.sleep(5000); // Sleep for 5 seconds
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }

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

d) Differentiate between Multitasking and Multithreading?

Ans:

a) Explain the thread life cycle in detail?

Ans: Life Cycle of a Thread

Threads, the smallest unit of a process, have a life cycle. In Java, this life cycle features six main states that any thread can occupy at a given point in time:

1. New

A thread is in this state when you've created an instance of the Thread class but haven't invoked the start() method yet. It remains in this state until the program starts the thread.

2. Active

This state consists of two sub-states, Runnable and Running. Runnable implies that the thread is ready for execution and is waiting for resource allocation by the thread scheduler. Running means the thread scheduler has selected the thread and is currently executing its run() method.

3. Blocked / Waiting

A thread enters this state when it is temporarily inactive and waiting for a signal to proceed due to reasons like waiting for a resource to become available (Blocked) or waiting for another thread to perform a specific action (Waiting).

4. Timed Waiting
In this state, a thread is waiting for a specified period. A thread might enter this state through methods like Thread.sleep(long millis) or Object.wait(long timeout) where it waits for a particular duration before resuming its activities.
5. Terminated
This is the final state in the thread life cycle. The thread arrives here when it has completed its execution, i.e., its run() method has been completed, or it has been abruptly terminated due to an unhandled exception. Once in this state, the thread cannot be resumed.
As Java developers, having a profound understanding of these states and the transitioning nuances between them provides us with the power to harness threads effectively, optimizing the execution of our concurrent programs.

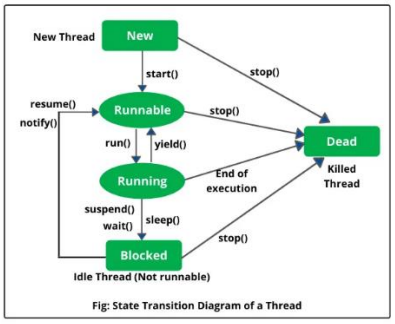

Fig: State Transition Diagram of a Thread

b) Explain the Java thread by Extending thread class and implementing Runnable interfaces with an example?
Ans:

Aspect	**Extending Thread Class**	**Implementing Runnable Interface**
Inheritance	Extends Thread class, which is not possible if your class already extends another class	Implements Runnable interface, allowing for multiple inheritance of classes
Flexibility	Less flexible due to single inheritance limitation	More flexible as it allows for implementation of other interfaces or inheritance from other classes
Code Separation	Code for task execution and thread management is combined	Code for task execution is separated from thread management, promoting cleaner design
Reusability	Limited reusability if you need to extend another class	Can reuse the Runnable implementation with different Thread instances
Example Usage	Suitable for simple cases where a class needs to perform threading	Preferred for complex scenarios where threading and business logic are better separated

Example for both has to be given

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
```

```
            System.out.println("Thread " + Thread.currentThread().getId() + " is running, i=" + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted.");
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();

        thread1.start(); // Start thread1
        thread2.start(); // Start thread2
    }
}
```

Example :

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable " + Thread.currentThread().getId() + " is running, i=" + i);
            try {
                Thread.sleep(500); // Sleep for 500 milliseconds
            } catch (InterruptedException e) {
                System.out.println("Runnable interrupted.");
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();

        Thread thread1 = new Thread(myRunnable);
        Thread thread2 = new Thread(myRunnable);

        thread1.start(); // Start thread1
        thread2.start(); // Start thread2
    }
}
```

Q. No	Question (s)	Marks	BL	CO
	UNIT – IV			
1	a) Define 'Events' in Java with respect to event handling.	1M	L2	C214.5
	b) What are 'Event sources' in Java?	1M	L1	C214.5
	c) What is an event listener?	1M	L1	C214.5
	d) What is a dialog box?	1M	L1	C214.5
	e) What is a layout manager?	1M	L1	C214.5
2	a) Explain different types of 'Event sources' in Java?	3M	L2	C214.5
	b) Explain 'Event Listeners' and their role in Java.	3M	L2	C214.5
	c) Describe the different ways to handle events in Java.	3M	L2	C214.5
	d) Explain about Handling Mouse Events.	3M	L2	C214.5
	e) Explain about "Button" component.	3M	L2	C214.5
3	a) Explain Delegation Event Model in Java.	5M	L2	C214.5
	b) Discuss the advantages and disadvantages of using adapter classes.	5M	L2	C214.5
	c) Explain Handling Mouse Events in java with example.	5M	L2	C214.5
	d) Write a GUI program in Java that containing a Button labeled "Click Here." On clicking the button, the program should display "Button clicked!" in the console.	5M	L2	C214.5
	e) What is the Difference Between TextField and TextArea in Java	5M	L1	C214.5
4	a) Describe the hierarchy of AWT user interface components.	10M	L2	C214.5
	b) Explain the complete event handling mechanism in Java, covering event sources, listeners, event classes, and their interactions in detail.	10M	L2	C214.5
	c) Discuss in detail 5 types of Layout Managers.	10M	L2	C214.5

Q. No	Question (s)	Marks	BL	CO
	UNIT – V			
1	a) Which container uses the boarder layout as their default layout.	1M	L2	C214.6
	b) What are untrusted applets?	1M	L1	C214.5
	c)What type of program is embedded in webpage to generate the dynamic content	1M	L1	C214.6
	d)Which method is called when every time applet receives focus	1M	L1	C214.5
	e) What method is called to clear the screen and calls paint() method.	1M	L1	C214.5

2	**a) Write syntax and example of drawPolygon(), drawRect(), drawArc()**	**3M**	L1	C214.5
	b) What is an event and what are the modules available for event handling	**3M**	L1	C214.6
	c) Difference between component and container	**3M**	L3	C214.6
	d)What are the restrictions imposed on java applets	**3M**	L1	C214.5
	e) What is the source and listener	**3M**	L1	C214.6
3	**a) Differentiate between Applet and Application**	**5M**	L3	C214.5
	b) Explain the following methods of applet class, drawRect(), drawPolygon(), drawArc(), drawRoundRect().	**5M**	L2	C214.5
	c)Provide the syntax of the following methods of graphics class and explain with program. a) drawOval() b)drawaPolygon()	**5M**	L2	C214.5
	d) Define applet and write a program to display message "Welcome to Java"	**5M**	L1	C214.5
	e) Explain any 4 applet tag with proper explanation.	**5M**	L1	C214.5
4	**a) Explain in detail MVC architecture.**	**10M**	L1	C214.6
	b) Explain applet lifecycle with diagram.	**10M**	L1	C214.6
	c) Explain different swing components.	**10M**	L1	C214.6