Unit 3 java

21)a). List the difference between an abstract class and an interface.

b). Write a Java program to find the square and cube of a given number using hierarchical inheritance.

a) Difference between Abstract Class and Interface

Abstract Class:

- Can have both abstract and non-abstract methods.
- Can have instance variables (fields).
- Can have constructors.
- Supports single inheritance (extends only one class).
- Used to define a base class for other classes to extend.

Interface:

- Can only have abstract methods (Java 8+ allows default and static methods).
- Cannot have instance variables (fields) (except for constants).
- Cannot have constructors.
- Supports multiple inheritance (implements multiple interfaces).
- Used to define a contract for classes to implement.

Key Differences Summarized:

- **Methods:** Abstract class can have abstract and non-abstract methods. Interface can only have abstract methods (and default/static methods in Java 8+).
- **Variables:** Abstract class can have instance variables. Interface can only have constants (public static final fields).
- **Constructors:** Abstract class can have constructors. Interface cannot have constructors.
- **Inheritance:** Abstract class supports single inheritance (extends one class). Interface supports multiple inheritance (implements multiple interfaces).

b) Java Program Using Hierarchical Inheritance to Find Square and Cube

Here's a Java program demonstrating hierarchical inheritance to find the square and cube of a given number:

```
// Base class Number
class Number {
    protected int num;

    public Number(int num) {
        this.num = num;
    }

    // Method to get the number
    public int getNumber() {
```

```
        return num;
    }
}

// Subclass Square extending Number
class Square extends Number {
    public Square(int num) {
        super(num);
    }

    // Method to calculate square
    public int calculateSquare() {
        return num * num;
    }
}

// Subclass Cube extending Number
class Cube extends Number {
    public Cube(int num) {
        super(num);
    }

    // Method to calculate cube
    public int calculateCube() {
        return num * num * num;
    }
}

public class HierarchicalInheritanceExample {
    public static void main(String[] args) {
        int number = 5;

        Square square = new Square(number);
        Cube cube = new Cube(number);

        System.out.println("Number: " + number);
        System.out.println("Square: " + square.calculateSquare());
        System.out.println("Cube: " + cube.calculateCube());
    }
}
```

Explanation:

1. **Base Class `Number`:**
   - Defines a constructor to initialize `num`.
   - Provides a method `getNumber()` to retrieve the number.
2. **Subclasses `Square` and `Cube`:**
   - Extend `Number` and inherit `num`.
   - Each subclass provides a specific method (`calculateSquare()` for square and `calculateCube()` for cube) to compute the respective values.
3. **Main Class `HierarchicalInheritanceExample`:**
   - Creates instances of `Square` and `Cube` for a given number (`5` in this case).
   - Calls respective methods (`calculateSquare()` and `calculateCube()`) to compute and print the square and cube of the number.

Output:

```
Number: 5
Square: 25
Cube: 125
```

22)Define interface? Explain the implementation of an interface in Java with an example.

Define Interface

An interface in Java is a reference type that defines a set of abstract methods (methods without a body) and constants (public static final fields). It can also include default methods (methods with a body, introduced in Java 8) and static methods (methods that can be called without an instance of the class).

Implementation of an Interface in Java

To implement an interface in Java, a class must provide concrete implementations for all the abstract methods defined in the interface. This allows the class to fulfill the contract specified by the interface, ensuring that any instance of the class can be treated as an instance of the interface.

Example of Interface Implementation

Here's an example demonstrating the implementation of an interface in Java:

```
// Interface Shape
interface Shape {
    // Abstract method to calculate area
    double calculateArea();
}

// Class Circle implementing Shape interface
class Circle implements Shape {
    private double radius;

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

    // Implementing interface method to calculate area
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Class Rectangle implementing Shape interface
class Rectangle implements Shape {
    private double length;
    private double width;

    // Constructor
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
```

```
    }

    // Implementing interface method to calculate area
    @Override
    public double calculateArea() {
        return length * width;
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        // Creating instances of Circle and Rectangle
        Circle circle = new Circle(5);
        Rectangle rectangle = new Rectangle(4, 6);

        // Calculating and displaying areas
        System.out.println("Area of Circle: " + circle.calculateArea());
        System.out.println("Area of Rectangle: " +
rectangle.calculateArea());
    }
}
```

Explanation:

1. **Interface `Shape`:**
   - Declares an abstract method `calculateArea()` that any class implementing `Shape` must define.
2. **Class `Circle` and `Rectangle`:**
   - Implement `Shape` interface and provide specific implementations for `calculateArea()` based on circle and rectangle formulas.
3. **Main Class `InterfaceExample`:**
   - Creates instances of `Circle` and `Rectangle`.
   - Calls `calculateArea()` on each instance, demonstrating polymorphism as both objects are treated as `Shape`.

Output:

```
Area of Circle: 78.53981633974483
Area of Rectangle: 24.0
```

23)

Explain the relationship between classes and interfaces with an example of implementing interfaces with different classes.

Relationship Between Classes and Interfaces in Java

In Java, the relationship between classes and interfaces is fundamental to achieving abstraction, multiple inheritance, and polymorphism. Here's a detailed explanation along with an example demonstrating how classes implement interfaces.

Understanding Interfaces

- **Interface Definition:** An interface in Java defines a contract for classes to implement. It contains method signatures (without method bodies), default methods (with method bodies since Java 8), static methods, and constant variables.
- **Purpose:** Interfaces provide a way to achieve abstraction, allowing you to specify what a class should do (method signatures) without specifying how it should be done (implementation details).

Implementing Interfaces with Classes

To implement an interface in Java, a class must use the `implements` keyword followed by the interface name. The class then provides concrete implementations for all the abstract methods declared in the interface. This allows the class to fulfill the contract specified by the interface.

Example: Implementing Interfaces with Different Classes

Let's consider an example where we have an interface `Animal` with methods `sound()` and `move()`, and two classes `Dog` and `Bird` implementing this interface:

```
// Interface Animal
interface Animal {
    void sound(); // Abstract method for producing sound
    void move();  // Abstract method for movement
}

// Class Dog implementing Animal interface
class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }

    @Override
    public void move() {
        System.out.println("Dog runs");
    }
}

// Class Bird implementing Animal interface
class Bird implements Animal {
    @Override
    public void sound() {
        System.out.println("Bird chirps");
    }

    @Override
    public void move() {
        System.out.println("Bird flies");
    }
}

public class InterfaceImplementationExample {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal bird = new Bird();

        // Polymorphic behavior
        dog.sound();
```

```
        dog.move();

        bird.sound();
        bird.move();
    }
}
```

Explanation:

1. **Interface `Animal`:**
   - Defines two abstract methods `sound()` and `move()` that any class implementing `Animal` must implement.
2. **Classes `Dog` and `Bird`:**
   - Implement `Animal` interface and provide specific implementations for `sound()` and `move()` methods according to their behavior (`Dog` barks and runs, `Bird` chirps and flies).
3. **Main Class `InterfaceImplementationExample`:**
   - Demonstrates polymorphic behavior by creating instances of `Dog` and `Bird` as `Animal` references.
   - Calls `sound()` and `move()` methods on each instance, treating them uniformly through the `Animal` interface.

Output:

```
Dog barks
Dog runs
Bird chirps
Bird flies
```

24)Briefly explain Java Accessing Implementations through Interface References with an example of extending interface-interface inheritance.

Java: Accessing Implementations through Interface References

In Java, you can use interface references to access and manipulate objects of classes that implement those interfaces. This approach allows for flexibility, polymorphism, and code reusability. Additionally, interfaces can extend other interfaces, forming a hierarchy of contracts that classes can implement.

Example: Extending Interface-Interface Inheritance

Let's illustrate how interfaces can extend each other and how classes implement these interfaces:

```
// Interface Animal
interface Animal {
    void sound(); // Abstract method for producing sound
    void move();  // Abstract method for movement
}

// Subinterface Bird extending Animal
interface Bird extends Animal {
    void fly(); // Abstract method specific to birds
```

```
}

// Class Sparrow implementing Bird interface
class Sparrow implements Bird {
    @Override
    public void sound() {
        System.out.println("Sparrow chirps");
    }

    @Override
    public void move() {
        System.out.println("Sparrow hops");
    }

    @Override
    public void fly() {
        System.out.println("Sparrow flies short distances");
    }
}

public class InterfaceInheritanceExample {
    public static void main(String[] args) {
        Bird sparrow = new Sparrow();

        // Polymorphic behavior through interface reference
        sparrow.sound();
        sparrow.move();
        sparrow.fly();
    }
}
```

Explanation:

1. **Interface `Animal`:**
   - Defines two abstract methods `sound()` and `move()`.
2. **Interface `Bird` (extends `Animal`):**
   - Extends `Animal` interface and adds another abstract method `fly()` specific to birds.
3. **Class `Sparrow` (implements `Bird`):**
   - Implements `Bird` interface and provides concrete implementations for `sound()`, `move()`, and `fly()` methods.
4. **Main Class `InterfaceInheritanceExample`:**
   - Creates an instance of `Sparrow` and assigns it to a `Bird` reference (`sparrow`).
   - Calls `sound()`, `move()`, and `fly()` methods through the `sparrow` reference, demonstrating polymorphic behavior.

Output:

```
Sparrow chirps
Sparrow hops
Sparrow flies short distances
```

25)Define multiple inheritance in Java by interacting with program code.

In Java, multiple inheritance refers to a situation where a class inherits properties and behavior from more than one superclass. Unlike some other object-oriented programming

languages (like C++), Java does not support multiple inheritance through class inheritance (i.e., a class cannot extend more than one class). However, Java allows multiple inheritance through interfaces, where a class can implement multiple interfaces.

Example of Multiple Inheritance using Interfaces

```
// Interface 1: Vehicle
interface Vehicle {
    void start(); // Abstract method
    void stop();  // Abstract method
}

// Interface 2: Engine
interface Engine {
    void accelerate(); // Abstract method
    void brake();      // Abstract method
}

// Class Car implementing both Vehicle and Engine interfaces
class Car implements Vehicle, Engine {
    @Override
    public void start() {
        System.out.println("Car started");
    }

    @Override
    public void stop() {
        System.out.println("Car stopped");
    }

    @Override
    public void accelerate() {
        System.out.println("Car accelerating");
    }

    @Override
    public void brake() {
        System.out.println("Car braking");
    }
}

public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Car myCar = new Car();

        // Calling methods from both interfaces
        myCar.start();
        myCar.accelerate();
        myCar.brake();
        myCar.stop();
    }
}
```

Explanation:

1. **Interfaces `Vehicle` and `Engine`:**
    - Define sets of abstract methods (`start()`, `stop()` in `Vehicle`; `accelerate()`, `brake()` in `Engine`).

2. **Class `Car` implementing both interfaces:**
    - Implements `Vehicle` and `Engine`, providing concrete implementations for all methods defined in both interfaces.
3. **Main Class `MultipleInheritanceExample`:**
    - Creates an instance of `Car`.
    - Calls methods `start()`, `accelerate()`, `brake()`, and `stop()` on the `Car` instance, demonstrating the use of multiple inheritance through interfaces.

Output:

```
Car started
Car accelerating
Car braking
Car stopped
```

26)Explain the Java Nested Interface with an example of a nested interface that is declared within the class.

In Java, a nested interface is an interface that is declared within another class or interface. Nested interfaces are used to logically group related constants, methods, or other nested types within a class or interface scope. Here's an explanation along with an example demonstrating a nested interface declared within a class.

Example of Nested Interface within a Class

```
// Outer class
class Outer {

    // Nested interface within the Outer class
    interface Inner {
        void display(); // Abstract method
    }

    // Main method to demonstrate the nested interface
    public static void main(String[] args) {

        // Instantiating the nested interface
        Inner inner = new Inner() {
            // Implementation of the display method
            public void display() {
                System.out.println("This is the nested interface method");
            }
        };

        // Calling the display method using the object of the nested
interface
        inner.display();
    }
}
```

Explanation:

1. **Outer Class `Outer`:**
    - Contains a nested interface `Inner`.
2. **Nested Interface `Inner` within `Outer` class:**

    - Declares an abstract method `display()`.
3. **Main Method (`main`):**
    - Inside the `main` method, an instance of the nested interface `Inner` is created using an anonymous class implementation.
    - The `display()` method of the nested interface `Inner` is overridden and its implementation prints a message.
4. **Output:**
    - When `inner.display()` is called, it prints: `This is the nested interface method`.

27)Define Java Default Methods? Mention the different types of inner classes and explain anyone with an example.

Java Default Methods

In Java, default methods were introduced in Java 8 as a feature of interfaces. Default methods allow interfaces to provide a default implementation of a method. This enables the addition of new methods to interfaces without breaking existing implementations of those interfaces.

Characteristics of Default Methods:

- They are declared using the `default` keyword.
- They provide a default implementation that can be overridden by classes implementing the interface.
- They allow interfaces to evolve by adding new methods without forcing existing implementations to provide an implementation.

Example of Default Method:

```
// Interface with a default method
interface Vehicle {
    // Abstract method
    void start();

    // Default method
    default void stop() {
        System.out.println("Vehicle stopped");
    }
}

// Class implementing the Vehicle interface
class Car implements Vehicle {
    @Override
    public void start() {
        System.out.println("Car started");
    }

    // stop() method inherited from Vehicle interface
}

public class DefaultMethodExample {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start(); // Output: Car started
```

```
        myCar.stop();  // Output: Vehicle stopped (default implementation)
    }
}
```

Explanation:

- **Interface `Vehicle`:**
  - Declares an abstract method `start()` and a default method `stop()` with a default implementation.
- **Class `Car` implementing `Vehicle`:**
  - Implements the `start()` method as required by the `Vehicle` interface.
  - Inherits the default implementation of the `stop()` method from the `Vehicle` interface.
- **Main Class `DefaultMethodExample`:**
  - Creates an instance of `Car` (`myCar`).
  - Calls `start()` and `stop()` methods on `myCar`, demonstrating the use of the default method `stop()`.
- **Output:**

```
Car started
Vehicle stopped
```

Types of Inner Classes

In Java, there are several types of inner classes, which are classes defined within another class. These inner classes can be broadly categorized into four types:

1. **Member Inner Class**
2. **Static Nested Class**
3. **Local Inner Class (Method Local Inner Class)**
4. **Anonymous Inner Class**

Example of Member Inner Class:

```
// Outer class
class Outer {
    private int outerVar = 10;

    // Member Inner Class
    class Inner {
        void display() {
            System.out.println("Value of outerVar: " + outerVar);
        }
    }
}

public class InnerClassExample {
    public static void main(String[] args) {
        // Creating an instance of Outer class
        Outer outer = new Outer();

        // Creating an instance of Inner class using Outer class instance
        Outer.Inner inner = outer.new Inner();
```

```
        // Calling method of Inner class
        inner.display(); // Output: Value of outerVar: 10
    }
}
```

Explanation of Member Inner Class:

- **Outer Class `Outer`:**
  - Defines a private variable `outerVar`.
  - Contains a member inner class `Inner`.
- **Member Inner Class `Inner` within `Outer` class:**
  - Accesses the `outerVar` variable of the outer class.
- **Main Class `InnerClassExample`:**
  - Creates an instance of the `Outer` class (`outer`).
  - Creates an instance of the inner class `Inner` using `outer`.
  - Calls the `display()` method of the `Inner` class, which accesses `outerVar`.
- **Output:**

```
Value of outerVar: 10
```

28)a. What is a Java package? Explain the types of packages in detail.

b. Define user-defined packages. With a suitable example.

Java Packages

In Java, a package is a mechanism for organizing classes and interfaces into namespaces. It helps in preventing naming conflicts and provides access protection. Packages are also used for categorizing the classes/interfaces logically, making it easier to locate and use them.

Types of Packages in Java:

1. **Built-in Packages (Standard Packages):**
   - These are predefined packages that come with the Java Development Kit (JDK), such as `java.lang`, `java.util`, `java.io`, etc.
   - They provide fundamental classes and interfaces for tasks like input/output operations, collections, networking, etc.
2. **User-defined Packages:**
   - These are packages created by Java developers to organize their classes and interfaces according to their application's requirements.
   - User-defined packages help in grouping related classes/interfaces together, promoting modularity and code reusability.

User-defined Packages

User-defined packages in Java are packages created by developers to organize their classes and interfaces logically. They are declared using the `package` keyword at the beginning of the Java source file.

Example of User-defined Package:

Let's create a simple example to illustrate how to define and use a user-defined package:

1. **Create a directory structure:**
   - Create a directory named `myPackage` (or any name of your choice) to hold the package files.
   - Inside `myPackage`, create two files: `Greeting.java` (package class) and `Main.java` (main class).
2. **Greeting.java (Package Class):**

```
// Define the package
package myPackage;

// Class within the package
public class Greeting {
    public void greet() {
        System.out.println("Hello, welcome to my package!");
    }
}
```

3. **Main.java (Main Class):**

```
java
Copy code
// Main class outside the package
public class Main {
    public static void main(String[] args) {
        // Creating an instance of Greeting from the myPackage package
        myPackage.Greeting greeting = new myPackage.Greeting();
        greeting.greet(); // Output: Hello, welcome to my package!
    }
}
```

4. **Compile and Run:**
   - Compile the `Greeting.java` and `Main.java` files. Make sure they are in the correct directory structure (`myPackage` folder containing `Greeting.java` and `Main.java`).
   - Use the following commands:

     ```
     css
     Copy code
     javac myPackage/Greeting.java
     javac Main.java
     ```

   - Run the `Main` class:

     ```
     css
     Copy code
     java Main
     ```

Explanation:

- **Package Declaration (`Greeting.java`):**
  - The line `package myPackage;` declares that the `Greeting` class belongs to the `myPackage` package.

- **Class `Greeting` (within `myPackage`):**
  - Defines a simple method `greet()` that prints a greeting message.
- **Class `Main` (outside any package):**
  - Creates an instance of `Greeting` from the `myPackage` package and calls its `greet()` method.
- **Output:**
  - When you run `Main`, it prints: `Hello, welcome to my package!`, demonstrating that the `Main` class can access and use the `Greeting` class from the `myPackage` package.

29)a. Explain how to compile and run a Java package.

b. List various ways to access a package from another package and explain at least one way with an example code.

Compiling and Running a Java Package

Compiling a Java Package:

To compile a Java package, you need to follow these steps:

1. **Directory Structure:** Ensure your directory structure reflects the package hierarchy. For example, if you have a package `com.example`, create a directory structure like `com/example/`.
2. **Package Declaration:** In each Java source file (`*.java`), include a `package` statement at the beginning specifying the package name.
3. **Compilation Command:** Use the `javac` command followed by the path to the source file(s). If you have multiple source files in different directories, use the `-d` option to specify the output directory for compiled `.class` files.

   Example:

   ```
   javac -d . Main.java
   ```

   This command compiles `Main.java` and places the compiled `.class` file in the current directory (`.`).

Running a Java Package:

Once compiled, you can run the Java program from the root directory (where the package structure begins):

Example:

```
java com.example.Main
```

Here, `com.example.Main` specifies the fully qualified name of the class `Main` located in the `com.example` package.

Accessing a Package from Another Package

Java provides several ways to access classes/interfaces from one package to another:

1. **Import Statement:**
   - Use the `import` statement to specify the fully qualified class name or wildcard `*` to import all classes/interfaces from a package.
   - This makes it easier to access classes/interfaces from other packages without specifying the full package path each time.

Example Code:

Let's demonstrate how to access a class from another package using the `import` statement:

1. **Directory Structure:**

   Assume you have two packages: `com.example` and `com.utils`.

2. **Class in `com.example` Package:**

```
// File: com/example/ExampleClass.java
package com.example;

public class ExampleClass {
    public void display() {
        System.out.println("Hello from ExampleClass");
    }
}
```

3. **Class in `com.utils` Package:**

```
// File: com/utils/UtilsClass.java
package com.utils;

import com.example.ExampleClass; // Importing ExampleClass from
com.example package

public class UtilsClass {
    public static void main(String[] args) {
        ExampleClass example = new ExampleClass();
        example.display(); // Calling method from ExampleClass
    }
}
```

4. **Compiling and Running:**
   - Compile both classes specifying their respective source paths:

```
javac com/example/ExampleClass.java
javac com/utils/UtilsClass.java
```

   - Run the `UtilsClass` which accesses `ExampleClass` from `com.example` package:

```
java com.utils.UtilsClass
```

Explanation:

- **Package Structure:**
  - `ExampleClass` is located in the `com.example` package, and `UtilsClass` is in the `com.utils` package.
- **Import Statement:**
  - In `UtilsClass.java`, `import com.example.ExampleClass;` imports `ExampleClass` from `com.example` package.
- **Accessing ExampleClass:**
  - Inside `UtilsClass`, `ExampleClass example = new ExampleClass();` creates an instance of `ExampleClass` and calls its `display()` method.
- **Output:**
  - When you run `UtilsClass`, it prints: `Hello from ExampleClass`, demonstrating successful access and interaction between classes from different packages using the `import` statement.

30)Explain a subpackage in Java with a suitable example code.

In Java, a subpackage is simply a package within another package. This allows for further organization and hierarchy in the package structure. Subpackages help in organizing related classes and interfaces even more granularly, making the codebase more modular and easier to navigate.

Example of Subpackage in Java

Let's create a simple example to demonstrate how subpackages work:

1. **Directory Structure:**

   Assume we have a main package `com.company` and within it, we create a subpackage `com.company.utilities`.

2. **Class in Subpackage:**

```
// File: com/company/utilities/StringUtils.java
package com.company.utilities;

public class StringUtils {
    public static boolean isNullOrEmpty(String str) {
        return str == null || str.trim().isEmpty();
    }
}
```

3. **Main Class in Main Package:**

```
// File: com/company/MainApp.java
package com.company;

import com.company.utilities.StringUtils;

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

```
        String text = "Hello, World!";

        // Using method from StringUtils in the subpackage
        if (StringUtils.isNullOrEmpty(text)) {
            System.out.println("Text is null or empty");
        } else {
            System.out.println("Text is not null or empty");
        }
    }
}
```

4. **Compiling and Running:**
   - Compile both classes specifying their respective source paths:

     ```
     javac com/company/utilities/StringUtils.java
     javac com/company/MainApp.java
     ```

   - Run the `MainApp` class:

     ```
     java com.company.MainApp
     ```

Explanation:

- **Package Structure:**
  - `StringUtils` class is placed in the `com.company.utilities` subpackage, which is a subpackage of the `com.company` package.
- **Import Statement:**
  - In `MainApp.java`, `import com.company.utilities.StringUtils;` imports `StringUtils` from the `com.company.utilities` subpackage.
- **Using Subpackage Class:**
  - Inside `MainApp`, `StringUtils.isNullOrEmpty(text)` checks if the `text` variable is null or empty, demonstrating the use of a method from a class in a subpackage.
- **Output:**
  - Depending on the value of `text`, it will print either `Text is not null or empty` or `Text is null or empty`, showing the functionality of the `StringUtils` class method.