EXPERIMENT-9

Aim: Implement Multiple Inheritance.

Theory:

Multiple inheritance is a programming concept in object-oriented languages that allows a class to inherit properties and behaviors from more than one class. In Java, multiple inheritance is achieved through interfaces and is a topic of considerable depth. In this essay, we will explore multiple inheritance in Java in detail, covering the following aspects:

1. **Inheritance in Object-Oriented Programming (OOP):**

To understand multiple inheritance in Java, we should first grasp the fundamental concept of inheritance in OOP. Inheritance is one of the four main principles of OOP, alongside encapsulation, abstraction, and polymorphism. It enables the creation of new classes by reusing existing class properties and behaviors.

2. **Single Inheritance in Java:**

Java, like many other programming languages, supports single inheritance, which means a class can inherit from only one superclass. This constraint is designed to avoid the ambiguities and complexities associated with multiple inheritance.

3. **Need for Multiple Inheritance:**

Multiple inheritance becomes necessary when we want to create a new class that inherits properties and behaviors from more than one class. Consider real-world examples where objects exhibit multiple characteristics. For instance, a "FlyingCar" class may need to inherit properties from both "Car" and "Aircraft" classes.

4. **Problems with Multiple Inheritance:**

Multiple inheritance introduces several challenges and ambiguities, including the diamond problem. The diamond problem occurs when a class inherits from two classes that have a common ancestor, leading to ambiguity in method resolution. Java avoids these issues by not allowing multiple inheritance through classes.

5. **Interfaces in Java:**

Java provides a solution to the problems associated with multiple inheritance through the use of interfaces. An interface is a contract that defines a set of methods without providing their implementations. A class can implement multiple interfaces, thereby inheriting method signatures without causing conflicts.

6. **Defining Interfaces:**

In Java, you define an interface using the `interface` keyword. Interfaces can contain method declarations, but they are implicitly public, abstract, and do not contain instance variables. Classes implement interfaces using the `implements` keyword.

7. **Implementing Multiple Interfaces:**

A class can implement multiple interfaces by separating them with commas. This allows the class to inherit the method signatures of all the interfaces it implements. For example:

```
   class FlyingCar implements Car, Aircraft {
       // Implement methods from Car and Aircraft interfaces
   }
```

8. **Method Overriding in Interfaces:**

When a class implements multiple interfaces, it might inherit methods with the same name from different interfaces. In such cases, the class is required to provide implementations for these methods, avoiding ambiguity.

9. **Default Methods in Interfaces:**

Java 8 introduced the concept of default methods in interfaces. A default method provides a default implementation for a method defined in the interface. This allows for backward compatibility when new methods are added to interfaces.

10. **Multiple Inheritance of Behavior:**

Through multiple interface implementations, a class can inherit behavior from multiple sources. This enables a class to act like it belongs to multiple categories, making Java's type system more flexible.

11. **Interfaces vs. Abstract Classes:**

While interfaces provide a form of multiple inheritance in Java, it's essential to understand the differences between interfaces and abstract classes. Abstract classes can have constructors and instance variables, while interfaces cannot. This difference impacts how they are used and when to choose one over the other.

12. **Use Cases of Multiple Inheritance in Java:**

Multiple inheritance through interfaces is widely used in Java to solve real-world problems. For example, in graphical user interface (GUI) libraries, classes may implement interfaces like `MouseListener` and `ActionListener` to handle various events.

13. **Ambiguity Resolution:**

While Java provides a mechanism for multiple inheritance through interfaces, there can still be situations where method names clash, causing ambiguity. In such cases, explicit qualification (i.e., specifying the interface name) can resolve the ambiguity.

14. **Preventing and Handling Ambiguities:**

To prevent ambiguities, it's essential to design interfaces with care. When ambiguities occur, the Java compiler requires the programmer to resolve them explicitly. This may involve choosing a specific method implementation or using the `super` keyword to select the superclass's method.

15. **Multiple Inheritance in Modern Java:**

Java continues to evolve, and newer versions, like Java 9 and beyond, introduce features and enhancements related to interfaces and multiple inheritance. These updates make working with multiple inheritance in Java even more powerful and flexible.

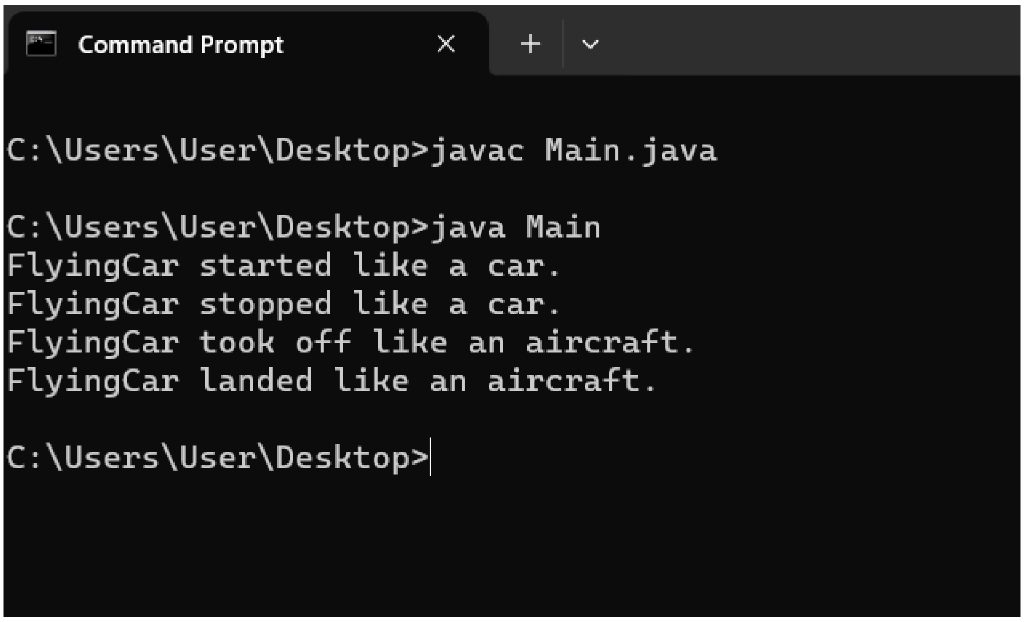
16. **Best Practices and Design Patterns:**

Effective use of multiple inheritance through interfaces requires adherence to best practices and design patterns. For instance, the Composite and Adapter patterns often involve classes implementing multiple interfaces to achieve specific goals.

CODE:

```
// Interface for the Car behavior
interface Car {
    void start();
    void stop();
}

// Interface for the Aircraft behavior
interface Aircraft {
    void takeOff();
    void land();
}

// Class that implements both Car and Aircraft interfaces
class FlyingCar implements Car, Aircraft {
    public void start() {
        System.out.println("FlyingCar started like a car.");
    }
    public void stop() {
        System.out.println("FlyingCar stopped like a car.");
    }
    public void takeOff() {
        System.out.println("FlyingCar took off like an aircraft.");
    }
    public void land() {
        System.out.println("FlyingCar landed like an aircraft.");
    }
}

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

        // Using methods from the Car interface
        flyingCar.start();
        flyingCar.stop();

        // Using methods from the Aircraft interface
        flyingCar.takeOff();
        flyingCar.land();
    }
}
```

OUTPUT:



```
C:\Users\User\Desktop>javac Main.java

C:\Users\User\Desktop>java Main
FlyingCar started like a car.
FlyingCar stopped like a car.
FlyingCar took off like an aircraft.
FlyingCar landed like an aircraft.

C:\Users\User\Desktop>
```

Conclusion:

Multiple inheritance in Java is achieved through interfaces, allowing a class to inherit method signatures from multiple sources without the issues associated with multiple inheritance through classes. This approach provides flexibility and promotes the design of more modular and reusable code.

In conclusion, multiple inheritance in Java is a fundamental concept that allows classes to inherit behavior from multiple sources through the use of interfaces. While Java avoids the complexities of multiple inheritance through classes to prevent the diamond problem, interfaces offer a flexible and robust solution to achieve the benefits of multiple inheritance in a controlled and safe manner. By understanding the principles, best practices, and design patterns related to multiple inheritance in Java, programmers can effectively harness the power of this feature to create modular, extensible, and maintainable software.