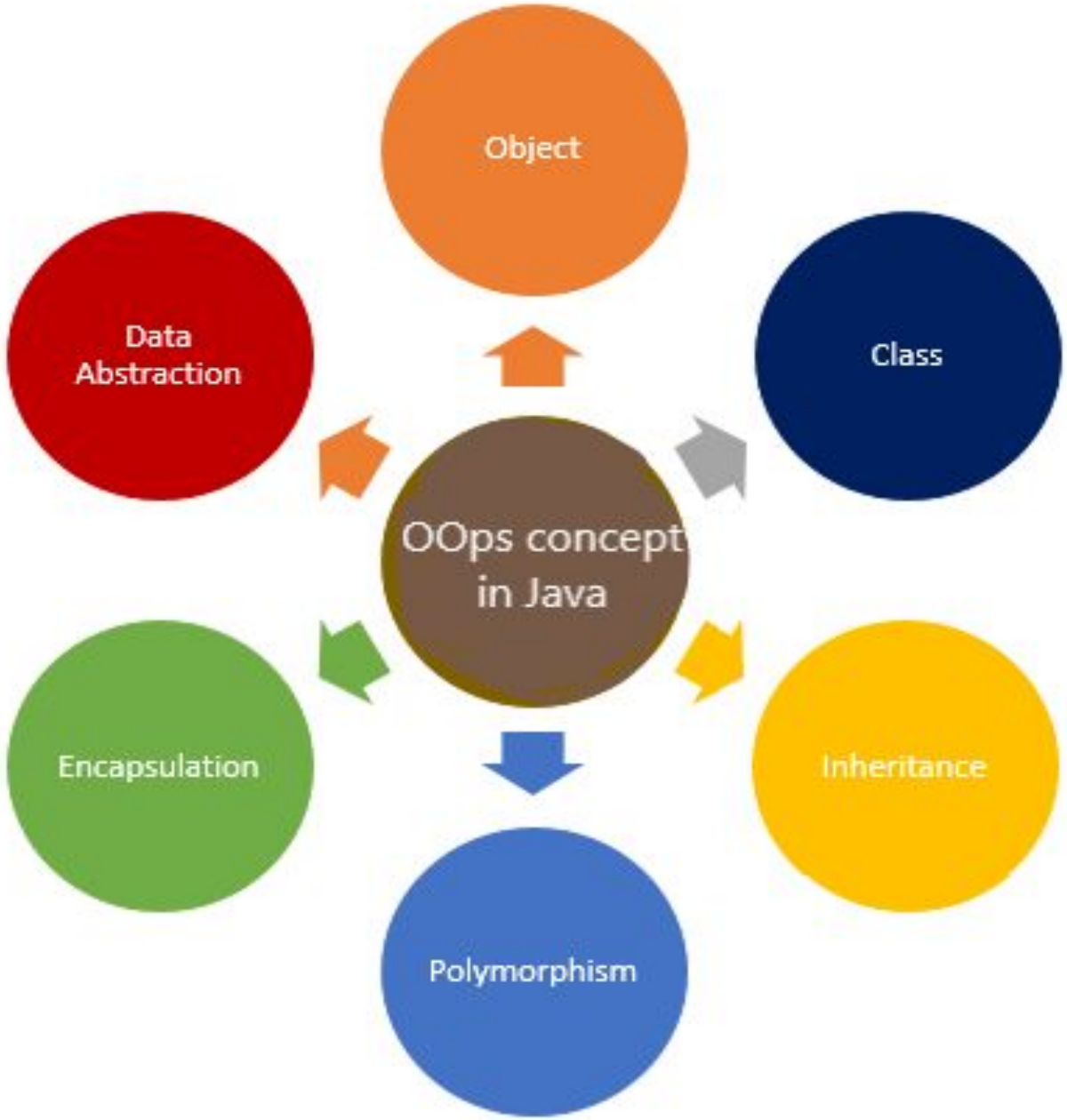
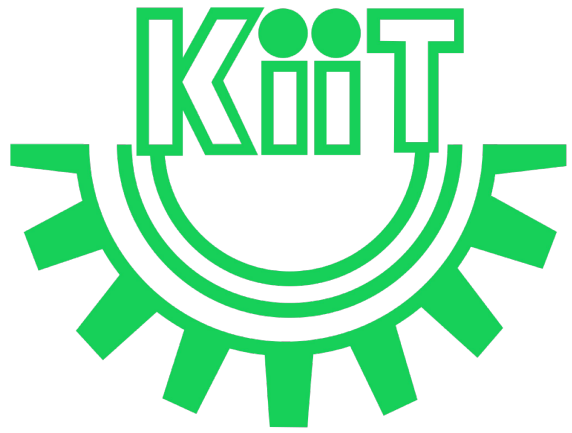


CS20004: Object Oriented Programming using Java

Lec-16

In this Discussion . . .

- Interfaces
  - Relationship between a class and an interface
  - Interface Declaration
  - Intuition behind the idea of Interfaces
  - Interfaces Vs. Abstract classes
  - Why use Java Interfaces
  - Interface Example
  - Implementing Multiple Interfaces
  - Extending Interfaces
  - Multiple Inheritance
- References



Interface in Java

keyboard interface glide class ways since angry
another certain class cake class java new
declared implementations java since new since will
baker new glide keyboard
extend java new angry new high glide certain using class
methods
since java glide example interface multiple
able class interface extends bird glide
assume glide class canary provide java

Interface

- In Java, ***an interface specifies the behavior of a class by providing an abstract type.***

> *In other words, an interface comprises a set of abstract methods that we would want our class to implement.*
>
> - These methods are public and abstract by default(i.e., we don't have to explicitly use the "abstract" keyword), and
> - Any class implementing the interface will need to provide implementations of those methods.

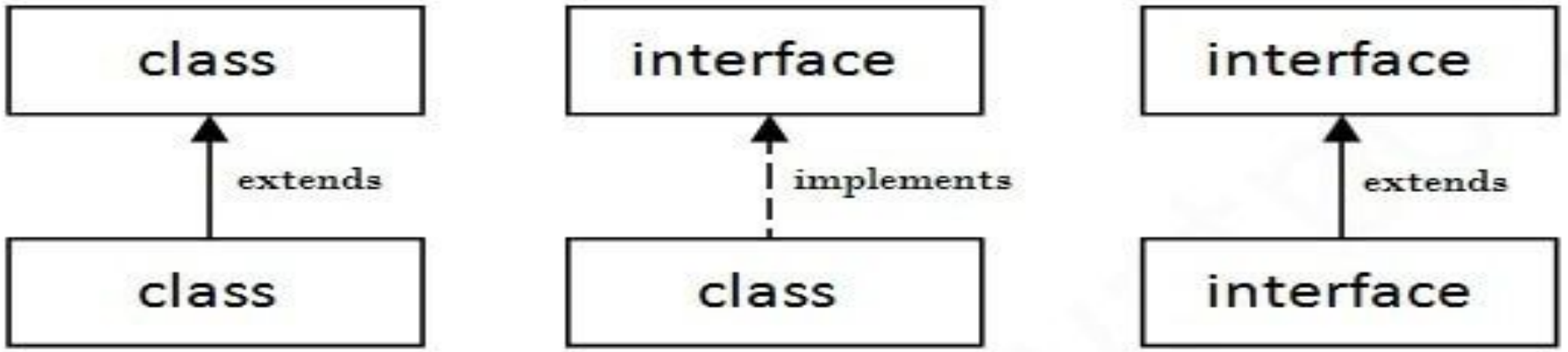
- As one of Java's core concepts, abstraction, polymorphism, and multiple inheritance are supported through the use of interfaces.

Interface

- **Interfaces are used in Java to achieve abstraction.**
- In general terms, an interface can be defined as a container that stores the signatures of the methods to be implemented in the code segment.
- It improves the levels of Abstraction.

Relation between a class and an Interface

- A class extends another class, an interface extends another interface, but a class implements an interface.



Interface declaration

- An interface is declared by using the **interface** keyword.

- **It provides total abstraction; means all the**
  - **methods in an interface are declared with the empty body with all of them declared as public and abstract,**
  - **while variables being public, static and final implicitly by the compiler.**

Syntax:

```
interface Interface_Name
{
        //Declare Constant Fields;
        //Declare Methods;
        //Default Methods;
}
```

Interface declaration

- An interface is declared by using the **interface** keyword.
- We will often find interfaces being named with an **able** suffix. For example, we have Runnable, Callable, Comparable, etc. in Java, but this is not a mandatory convention.

```
interface Interface_Name
{
        //Declare Constant Fields;
        //Declare Methods;
        //Default Methods;
}
```

Interface declaration

- An interface is declared by using the **interface** keyword.
- **Note:-** A class that implements an interface must implement all the methods declared in the interface, i.e., the class must implement it using the *implements* keyword and *override* (provide its own implementations) all the methods declared in the interface.

```
//using the "interface" keyword for declaration
interface Car {
  //some properties of a car
  public void start();
  // some other methods
}
```

```
class DieselCar implements Car{
   public void start(){
    System.out.println("Diesel Car starting...Vrooooom!");
   }
}
```

```
class ElectricCar implements Car{
   public void start(){
    System.out.println("Electric Car starting quietly...Zzzzzz!");
   }
}
```

Interface declaration

- **Note:-** A class that implements an interface must implement all the methods declared in the interface, i.e., the class must implement it using the *implements* keyword and *override* (provide its own implementations) all the methods declared in the interface.

```
//using the "interface" keyword for declaration
interface Car {
  //some properties of a car
  public void start();
  // some other methods
}
```

```
class DieselCar implements Car{
   public void start(){
    System.out.println("Diesel Car starting...Vroooooom!");
   }
}
```

```
class ElectricCar implements Car{
   public void start(){
    System.out.println("Electric Car starting quietly...Zzzzzz!");
   }
}
```

Although we cannot create objects of an interface, it can be used to reference a class that implements the interface.

```
Car tesla = new ElectricCar();
Car mercedes = new DieselCar();
```

Intuition behind the idea of Interfaces

- Scenario building

Intuition behind the idea of Interfaces

- Scenario building
    - Assume you are building out a version of the Angry Birds game.
    - Angry Birds is a fun game where the birds can be put on a slingshot to protect their eggs from certain pigs.
    - *Your task:* you have a few different types of birds, and you need to make them fly.

Intuition behind the idea of Interfaces

- Scenario building

    - *Your task:* you have a few different types of birds, and you need to make them fly.
    - We could implement it in the following way- using a **Bird** class as a parent class, and extending specific bird classes from the parent **Bird** class

Intuition behind the idea of Interfaces

- Scenario building
  - We could implement it in the following way- using a **Bird** class as a parent class, and extending specific bird classes from the parent **Bird** class.

```
class Bird
{
 //some properties of a bird
 String colour;
 String size;

 public void fly()
 {
   System.out.println("I can fly!");
 }
}
```

```
class Bluebird extends Bird
{
public void fly()
 {
 System.out.println("I can fly very far and high!");
 }
}
```

```
class Canary extends Bird{
public void fly()
 {
    System.out.println("I can't fly all that high..");
 }
}
```

Intuition behind the idea of Interfaces

- Scenario building

    - This seems alright so far, but suppose now we decide to make a space-themed version of Angry Birds, where you introduce a rocket, and you want to make the rocket fly.
    - **Oops!** The rocket is not a bird, so you cannot extend the Bird class from the Rocket class. **And this is where Interfaces come in.**

Intuition behind the idea of Interfaces

- Scenario building
  - **Oops!** The rocket is not a bird, so you cannot extend the Bird class from the Rocket class. **And this is where Interfaces come in.**
  - With an interface, we can abstract out the *fly()* method to a new class-like object (the interface) and make all the relevant classes implement this interface.

```
interface Flyable
{
        public void fly();
}
```

```
class Canary extends Bird implements Flyable
{
public void fly()
 {
    System.out.println("I can't fly all that high..");
 }
}
```

```
class Rocket implements Flyable
{
 public void fly()
 {
   System.out.println("I am zooming out into space!");
 }
}
```

This way, we can abstract out the flying property of an object from its definition by using an Interface.

Interface Vs. Abstract Class

- Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated. However,

Abstract Class	Interface
Abstract class can **have abstract and non-abstract methods.**	Interface can **have only abstract methods.** Since Java 8, it can have default & static methods also.
doesn't support multiple inheritance.	supports multiple inheritance.
can have final, non-final, static and non-static variables.	only static and final variables.
Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.

Why Use Java Interfaces?

- **Better polymorphism:** With interfaces, we don't have to make everything fit into one family of classes, offering much more flexibility.
  - Let us look at how we can leverage polymorphism in our cars example

```
class Helper{
  // creating a static method which can be called without instantiating an object
  public static void startCar(Car car){
      car.start();
  }
}

Helper.startCar(tesla); // Electric Car starting quietly...Zzzzzz!
Helper.startCar(mercedes); // Diesel Car starting...Vroooooom!
```

We can take any object that implements the Car interface as a parameter and call the start method on it. How powerful is that!

Why Use Java Interfaces?

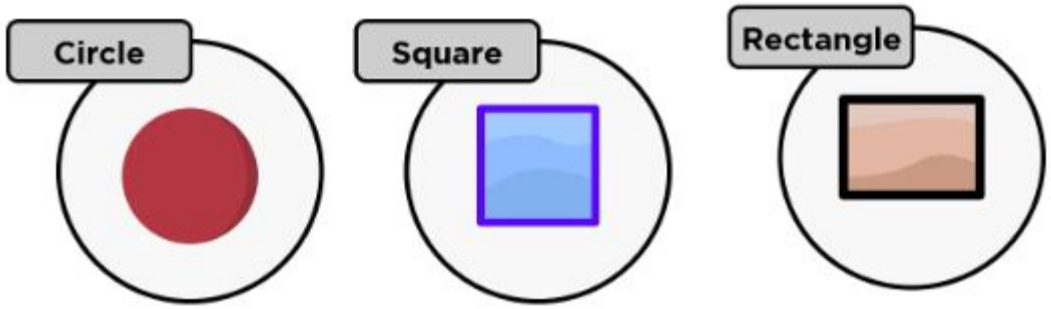
- **Multiple inheritances:** In Java, we cannot extend multiple classes because of the famous **Diamond problem**.
  - To solve this, we can use a Java interface to give child classes the freedom to extend whatever parent class they wish (and implement our interface) rather than forcing them to extend a specific base class to use our functionality.

Why Use Java Interfaces?

- **For abstraction:** Interfaces help to achieve security, hide certain details, and only show relevant and necessary information.
  - For example, when we start our car, we just plug in the key and start the car without worrying about the nitty-gritty details of what happens underneath when the car starts.
  - This is an example of abstraction! We carry out our task, that is, starting the car, but we don’t need to take care of anything else that happens in the background when starting the car.

Interface Example

- Let’s try to calculate the area of geometrical shapes using interfaces. For each shape, we have different methods.
- And all the methods are defined, independent of each other. Only method signatures are written in the Interface.



Interface Example

- And all the methods are defined, independent of each other. Only method signatures are written in the Interface.

```
package mypackage;
public interface Area
{
	public void Square();
	public void Circle();
	public void Rectangle();
}
```

```
package mypackage;
import java.util.Scanner;
public class ShapeArea implements
Area
{
	public void Circle()
	{
		Scanner kb = new
Scanner(System.in);
		System.out.println("Enter
the radius of the circle");
	double r = kb.nextInt();
	double areaOfCircle = 3.142 * r *
r;
		System.out.println("Area of
the circle is " + areaOfCircle);

	}
```

```
//Override
	public void Square()
	{
	Scanner kb2 = new
Scanner(System.in);
		System.out.println("Ent
er the length of the side of the
square");
	double s = kb2.nextInt();
double areaOfSquare = s * s;
		System.out.println("Are
a of the square is " +
areaOfSquare);

}
```

```
//Override
	public void Rectangle()
	{
		Scanner kb3 = new
Scanner(System.in);
		System.out.println("Ent
er the length of the Rectangle");
		double l =
kb3.nextInt();
		System.out.println("Ent
er the breadth of the Rectangle");
	double b = kb3.nextInt();
double areaOfRectangle = l * b;
		System.out.println("Are
a of the Rectangle is " +
areaOfRectangle);
}
```

Interface Example

- And all the methods are defined, independent of each other. Only method signatures are written in the Interface.

```
public class ShapeArea implements Area
{
	public void Circle()
	{
		Scanner kb = new Scanner(System.in);
		System.out.println("Enter the radius of the circle");
	double r = kb.nextInt();
	double areaOfCircle = 3.142 * r * r;
		System.out.println("Area of the circle is " + areaOfCircle);

	}
```

```
//Override
	public void Square()
	{
	Scanner kb2 = new Scanner(System.in);
		System.out.println("Enter the length of the side of the square");
	double s = kb2.nextInt();
double areaOfSquare = s * s;
		System.out.println("Area of the square is " + areaOfSquare);

}
```

```
//Override
	public void Rectangle()
	{
		Scanner kb3 = new Scanner(System.in);
		System.out.println("Enter the length of the Rectangle");
		double l = kb3.nextInt();
		System.out.println("Enter the breadth of the Rectangle");
	double b = kb3.nextInt();
double areaOfRectangle = l * b;
		System.out.println("Area of the Rectangle is " + areaOfRectangle);
}
```

```
public static void main(String args[])
	{
ShapeArea geometry = new ShapeArea();
		geometry.Circle();
		geometry.Square();
		geometry.Rectangle();
	}
}
```

Interface Example

- **Saving the files:**
  - Save the Interface Area in a separate java file called **Area.java**
  - Save the java class ShapeArea in a separate file called **ShapeArea.java**
- **Compiling the files:**
  - Start with compiling the Interface file first:
    - javac -d . Area.java
  - Next start with the ShapeArea.java file:
    - javac -d . ShapeArea.java
- **Finally, run the file with the main method():**
  - java mypackage.ShapeArea

Implementing Multiple Interfaces

- Assume that in our space-themed Angry Birds example, we implement a multi-stage Rocket, and we want the different modules of a rocket to separate at some point.
- In this scenario, we can have another interface, **Separable**, that can be implemented by the **Rocket** class along with the **Flyable** interface. This is how a class can implement multiple interfaces, and this is extremely useful when we want to achieve multiple inheritances.

```
class Rocket implements Flyable, Separable{
 void fly()
 {
    System.out.println("I am zooming out into space!");
 }

 void separate()
 {
    System.out.println("Carrying out stage separation..");
 }
}
```

Extending Interfaces

- Just like how a class can extend another class, an interface can also extend another.
- The **extends** keyword is used here as well, much like in a class.
- Suppose, in our Angry Birds example, we want our birds to be able to glide along with flying.

```
interface Glidable
{
        void glide();
}
```

Extending Interfaces

- Now our Bird classes can implement this interface as well.
- But since Birds that can glide can fly anyway, instead of having each child class implement two interfaces separately, we can extend the **Glidable** interface from the **Flyable** interface as below-

```
interface Glidable extends Flyable
{
                        void glide();
}
```

Extending Interfaces

- Now, when a Bird class implements the **Glidable** interface, it must also implement the fly method from the **Flyable** interface. So now our **Canary** bird class will look like this-

```
class Canary extends Bird implements Glidable
{
        void fly()
        {
                System.out.println("I can't fly all that high..");
         }
         void glide()
         {
                System.out.println("I am gliding through the sky!");
          }
}
```

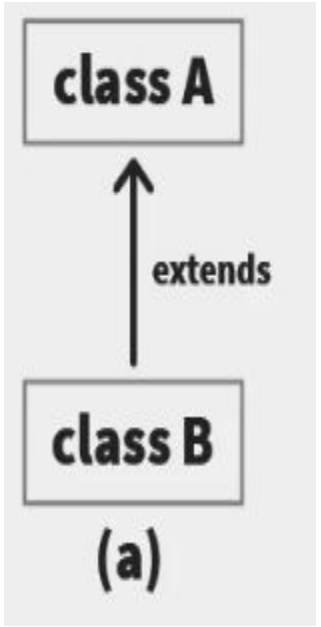
Extending Interfaces

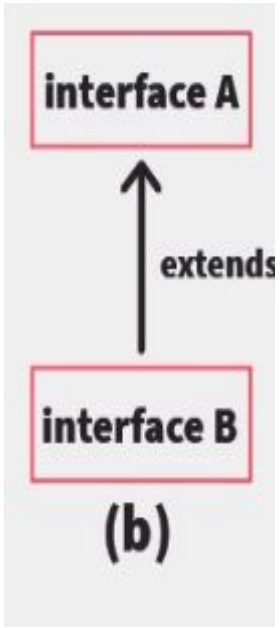
- So we can see that instead of having our **Canary** class implement both the interfaces separately, we can extend the **Flyable** interface from **Glidable** and make our **Canary** class simply implement the **Glidable** interface.

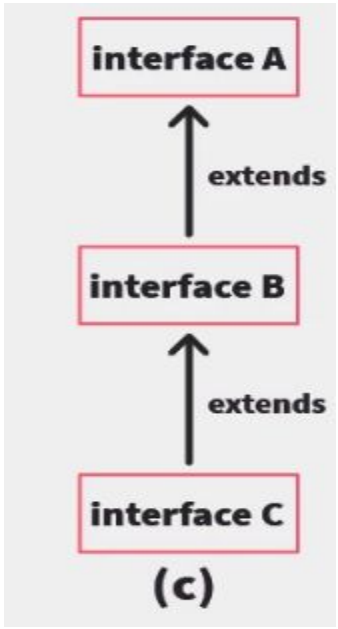
```
class Canary extends Bird implements Glidable
{
        void fly()
        {
                System.out.println("I can't fly all that high..");
         }
         void glide()
         {
                System.out.println("I am gliding through the sky!");
          }
}
```

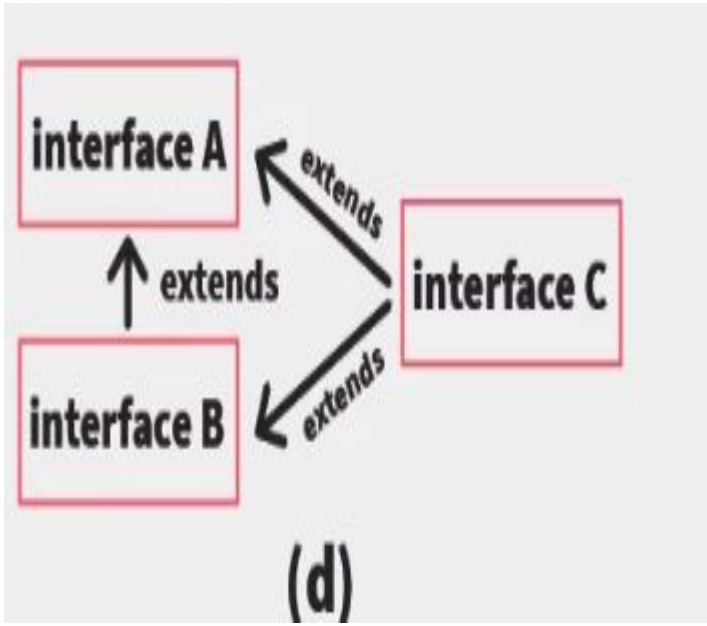
Extending Interfaces

- This concept can also be “extended” to multiple interfaces, where an interface can extend multiple interfaces. The different ways in which an interface can extend other interfaces are given below



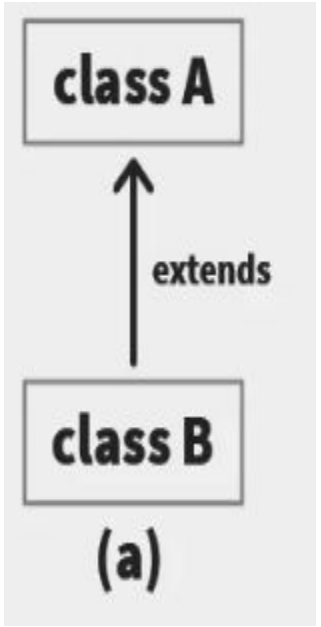


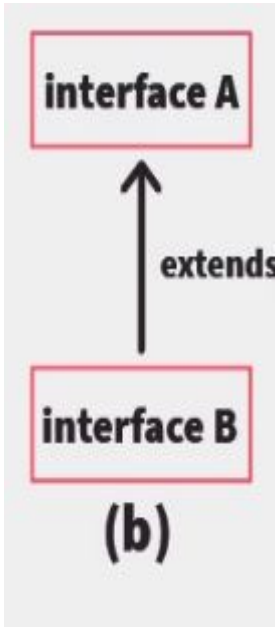


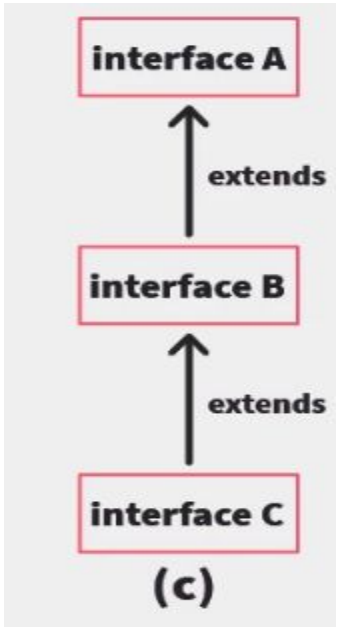


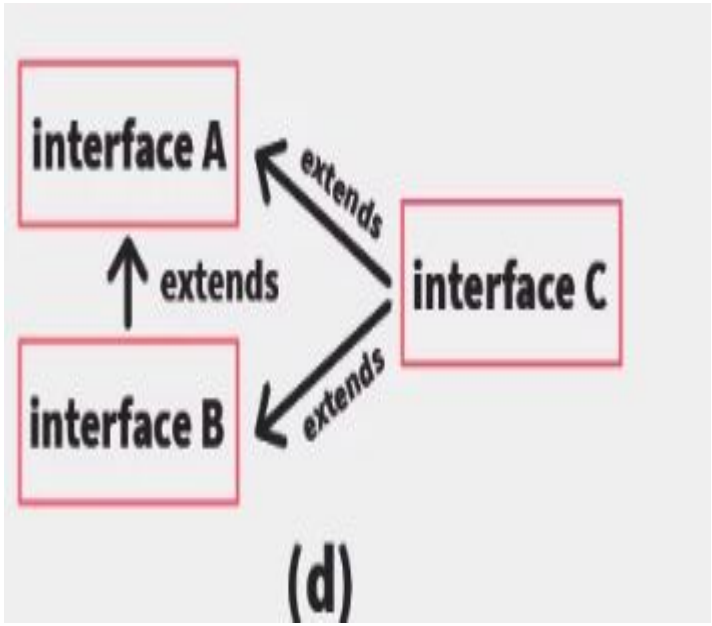
Extending Interfaces

- **Note:-** Interfaces cannot extend a class because this would violate the property of an interface that it must contain only abstract methods.









Multiple Inheritance using Interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

Multiple Inheritance in Java

References

1. https://www.simplilearn.com/tutorials/java-tutorial/java-interface
2. https://www.scaler.com/topics/java/interface-in-java/
3. https://www.javatpoint.com/interface-in-java
4. https://www.tutorialspoint.com/multiple-inheritance-by-interface-in-java
5. https://www.guru99.com/java-interface.html
6. https://www.webucator.com/article/how-to-implement-an-interface-in-java/#:~:text=Open%20a%20command%20prompt%20and,implemented%20by%20a%20Java%20program.&text=The%20Java%20program%20declares%20that,by%20using%20the%20implements%20keyword.
7.