


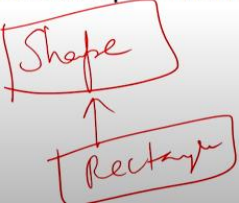
FUNDAMENTALS OF OBJECT ORIENTED PROGRAMMING

Lecture25

Lecture 25: Interface in Java

Understanding the real scenario of abstract class

```
1 abstract class Shape{  
2     abstract void draw();  
3 }  
4 //In real scenario, implementation is provided by others  
5 // i.e. unknown by end user  
6 class Rectangle extends Shape{  
7     void draw(){System.out.println("drawing rectangle");}  
8 }  
9
```



Welcome to lecture 25, Encapsulation and Abstraction. So in the previous lecture, we talked about the abstract class. So, in continuation, right? So, we will see one more example, right? So, exactly the real scenario.

Suppose I have an abstract class called Shape, right? So, under the abstract class Shape, I have an abstract method called draw, right? Void draw. So, now let us assume, right? In the real scenario, right?

Assume that the implementation is being provided by others, right? Unknown by the end user. So, now I have the class Rectangle, right? So, assume that I have class Rectangle which is extending Shape, right? So, you have a Shape as a base class, right?

So, assume that you have Shape as a, right? Which is an abstract class, and then you have the derived class, which is Rectangle, right? Correct. So now you can see you have an abstract method called draw, and then that is being defined over here in the derived class. Right.

```

10 ▾ class Circle1 extends Shape{
11   void draw(){System.out.println("drawing circle");}
12 }
13
14 //In real scenario, method is called by programmer or user
15 ▾ class TestAbstraction1{
16 ▾ public static void main(String args[]){
17   Shape s=new Circle1();
18   s.draw();
19 }
20 }

```

stdout
 drawing circle



You call it a superclass. Shape is a superclass, and Rectangle is a subclass, child class, or derived class. So, use extends; all these things we have seen. So, now it is printing 'drawing rectangle'; it is printing 'drawing rectangle'. So, now you assume that you have class Circle1, another class, right?

So, you have another class which is extending Shape, right? So, you have another class called Circle, let us say Circle1, right? So, Circle1, which is extending Shape, right? So, here you go. So, there also you have draw.

There, also, you have a draw. Right. So, let us have a TestAbstraction1. Right. You have a class which is a driver class.

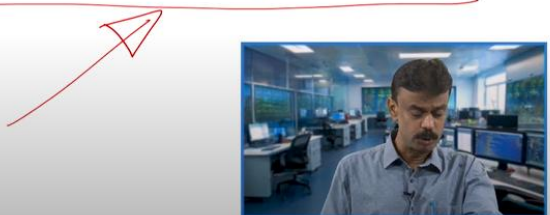
Right. You have the main. So, since it is a main, you call TestAbstraction1 as the driver class. Let us create an object s. Let us create an object s under Shape. Right.

So, let us create an object small s under the Shape class. Right. And then you have the constructor Circle1. Right. Exactly like the example we had seen.

Right. Bike and Honda4. Right. Under Bike, you have created an object obj. So when you have the constructor s, it will work.

Abstract class having constructor, data member, methods

```
1 //example of abstract class that have method body
2 abstract class Bike{
3     Bike(){System.out.println("bike is created");}
4     abstract void run();
5     void changeGear(){System.out.println("gear changed");}
6 }
7
```



Right. So you are calling s.draw. Right. The object s is invoking draw. Right.

So when it is happening, s.draw. Right. So we know which method will be called.

Right. So, we know which method will be called.

So, here you have the derived class, correct. So, here you have the derived class drawCircle1. So, System.out.println, you have drawing circle, right. So, here the drawing circle will be called, right. So, in this case, the drawing circle will be called.

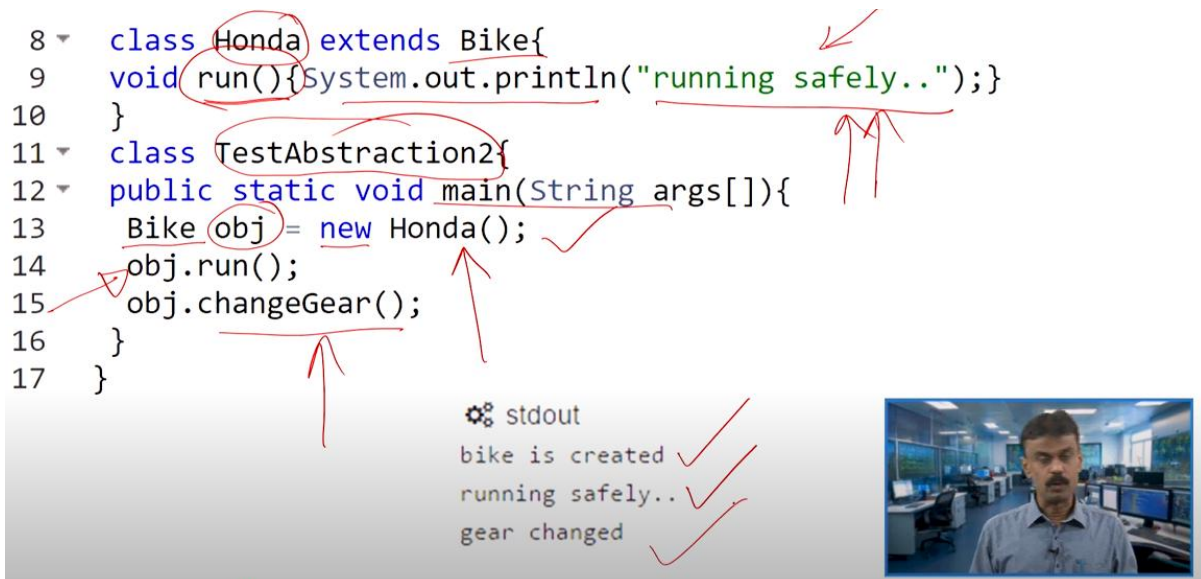
So, this will be called, correct. So, when I run the code, So, you can see drawing circle will be called, alright. So, your object s is under Shape, but the reference is in Circle1, right. So, upcasting, you are doing upcasting.

You have to do upcasting. So, suppose I put Shape s is equal to new, and then here also let us assume it is Shape, right. The constructor is also Shape, we know, right. You cannot instantiate an object s under the reference Shape. Right.

So, which will throw an error. Correct. So, therefore, you are upcasting this, and when you are upcasting, and moreover, we know that Shape is the abstract class, and it contains the abstract method as well. So, therefore, it will go to the derived class called Circle1, or you call it as a subclass Circle1. Under subclass Circle1, you have the method draw.

Right. So, that will print drawing circle. So, when you run the code, you will get the output drawing circle. So, another example, you have an abstract class that can contain a constructor, data member, methods, etc. So, for example, here is an abstract class Bike, and here is a constructor, and here is a constructor.

```
8 class Honda extends Bike{
9 void run(){System.out.println("running safely..");}
10 }
11 class TestAbstraction2{
12 public static void main(String args[]){
13     Bike obj = new Honda();
14     obj.run();
15     obj.changeGear();
16 }
17 }
```



stdout
bike is created ✓
running safely.. ✓
gear changed ✓

So, the constructor has Bike is created. Right. System.out.println Bike is created. Right. So, this is a constructor.

Bike is a constructor. And in line number four, you have abstract void run. All right. So that means it's an undefined function. Right.

An undefined method. Yes. This is an abstract method. So you can define it like this. So in C++, we saw pure virtual functions.

Correct. So you use the keyword virtual void run equals zero semicolon. Equivalent. Right. So in Java, we have abstract void run.

Right. With a semicolon. And then you have a usual method. Why is changeGear a usual method? So the usual method is your System.out.println and gear changed.

All right. So that is why we still have not achieved 100% abstraction. Right. 100% abstraction has not been achieved. The reason is this is an example.

So you have, let us say, two methods. One is a constructor. Correct. So one method is an abstract method. Another one is a non-abstract method.

So here we have 50 percent abstraction in terms of methods. Right. So now when we have the abstract class Bike. So it should be inherited. Right.

So that means you have class Honda, which means you have to extend. So, the class Honda, which is extending Bike. So, now you have the method called run, and

you have `System.out.println`, we call it as SOP, and running safely. You are printing running safely, right? So, now go to your driver class.

So, here you have the driver class, let us say `TestAbstraction2`. So, you have the main method. Under this main method, you have `obj`, which is an object, right? `obj` is under `Bike`. `Bike` is an abstract class allocating memory with a constructor `Honda`.

You are upcasting, right? Which is possible. This is absolutely fine. So this will work. Now `obj`, right?

So this object `obj` will invoke the function, the method `run`, right? `obj` will invoke the method `run`, right? So, when it is invoking, so obviously, right, this is an abstract method, right. So, that is being defined in the derived class or the subclass. So, the subclass method `run` will be invoked and running safely will be printed, right.

Running safely will be printed. Again, `obj` is invoking another method called `changeGear`, right. So, `obj` is invoking another method called `changeGear`. Right. So, when it is invoking `changeGear`, so `system.out.println`, you are getting gear changed, right.

So, you are getting gear, `changeGear`, all right. So, in this case, it will go to the base class. The reason is you are creating an object `obj`. So, it is overridden, right. So, it will go to the base class, all right.

And in the base class, you have the gear changed, right. So, that will be printed, all right. So, this will be printed. So, you are getting the output.

So, when you are creating an object `obj`, right? When you are creating an object `obj`, you have the constructor. So, you have the constructor. Constructor, you have one print statement. So, 'bike is created' will be printed. Right.

Bike is created will be printed. I hope it is clear. Bike is created will be printed. So that is why you are getting the first output: bike is created. Right.

So next, when `obj` is invoking `run`, running safely will be printed. So we are getting the next output: running safely. Right. And in the third case, when `obj` is invoking `changeGear`. Right.

When `obj` is invoking `changeGear`, gear changed will be printed, right. So, gear changed will be printed, right. I hope it is clear now, in the sense that in the abstract class, we can use the constructor data member, right. Obviously, we have seen several examples of abstract methods, right. So, here you go, we have abstract methods and then the usual method also, right, the non-abstract method.

So here, changeGear is the non-abstract method. So when we run the code, we get the output like this. So now, the next concept is the interface in Java. So when you want to achieve 100% abstraction, it is exactly the abstract that you define in a research paper. So we can do it with the help of an interface.

So it is exactly like a design plan. So what am I going to do in this paper? So in a similar way, what am I going to do in this program, right? So I need, let us say, 15 functions, right? 15 methods.

So these are all the methods I require, right? Assume that you are writing code or competing in a programming contest, right? So you have to plan. So when you are planning, you will have a set of methods, right? So this set of methods is what I want to use.

So, that is called the blueprint of a class. So, that means I will have the design plan. So, it contains, that means the interface contains the static constants. So, it contains only static constants and abstract methods only. So, it should have the constant.

Interface in Java

- An **interface in java** is a blueprint (a design plan) of a class. It has static constants and abstract methods only.
- The interface in java is **a mechanism to achieve fully abstraction**.
- There can be only abstract methods in the java interface not method body.
- It is used to achieve fully abstraction and **multiple inheritance** in Java.

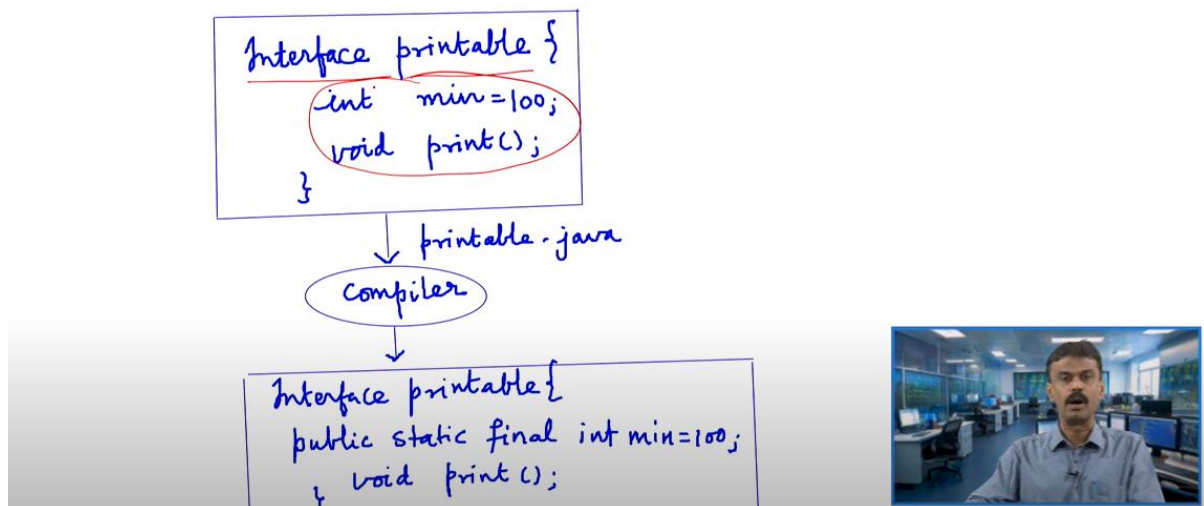


Suppose you are defining int min is equal to 100. It is nothing but static constant int min according to the compiler. So, once even though you are not defining that, we are going to see right, even though we are not defining. So, that is considered as constant, the data members min is equal to 5, int min equal to 5 in the sense. So, that will be constant, and then any method you are defining which will be the abstract method, one you can define only the abstract methods, right.

So, when I am talking about the interface. So, this is the mechanism you can achieve the full abstraction, right. So, the interface in Java. So, you can achieve 100% abstraction. So, when I am talking, obviously, when I am talking about abstract methods in the Java interface, so you do not have any method body.

So we are not defining anything, right? So inside, you are not defining anything. When I have the abstract, obviously, the method will also be abstract. So you do not have any method body, right? So since I said you can achieve 100% abstraction,

Interface



So now this is going to support your concept of multiple inheritance. So when we talked in terms of class, we have single inheritance or simple multi-level inheritance. But whereas we do not have multiple inheritance. When you are using only class, we do not have multiple inheritance. Hierarchical we have, but we do not have multiple inheritance.

So obviously you do not have hybrid inheritance either. Right. So this is going to support the concept of interface, which will support multiple inheritance in Java. So the main use of this is to achieve 100% abstraction. Right.

So, what is the use of an interface, you may ask. So, this will achieve full abstraction, or you can call it 100% abstraction. So, we can support the functionality of multiple inheritance. So, we are going to see. How this is going to support multiple inheritance.

If you recall, we have done multiple inheritance in C++, right? So, we have done multiple inheritance in C++ and Let us consider, in the case of Java, we are going to see how this is going to support multiple inheritance. So, suppose, let us consider we are going to have the keyword called Interface, right? We are going to have the keyword called Interface, and Printable is the name of the interface. All right.

There is a text. You have to use the keyword Interface and Printable. All right. So, which is the name of the interface. And then, suppose you are having.

All right. So suppose you have int min equal to 100 and void print. So this is a printable dot Java program. So according to the compiler. Alright.

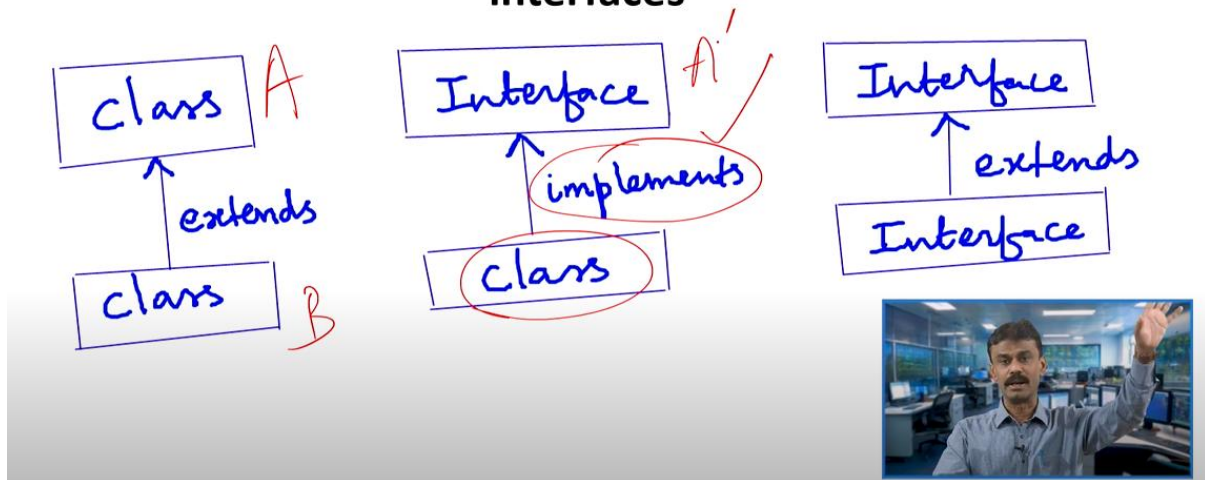
So according to the compiler. So this is static final int min equal to 100. That means it is a constant. Even if you do not define it. And similarly, void print will be nothing but the abstract.

Right. So this is nothing but the abstract void print. Alright. This is an abstract method according to the compiler. So that is what this diagram is depicting.

So, you may have the data member. Assume that you are not putting final. Keyword final. Keyword final is nothing but a constant. Const.

In C++, right. So, in the case of an interface, because this is 100% abstraction. So, here when you are writing int min is equal to 100, meaning it is public static final int min equal to 100, and void print function, which is nothing but the abstract function, right. So, in the compiler, it will be considered like this, even though if you do not mention. Even though if you do not mention in the compiler, it will be considered like this.

Understanding relationship between classes and interfaces



So, with this in mind, we will see how we are going to use the keywords. So, in fact, when we talk about class, one is the superclass, and another one is a subclass. Class A, let us say it is a superclass. Class B is a subclass. B is extending A. Class B extends A. So, class B extends A.

So assume that you have an interface and a class is going to inherit the interface. Possible, right? So in that case, you have to use the keyword called implements. You have to use the keyword called implements. So assume that you have the superclass equivalent.

You say super interface. Right. Or simply interface, which is the equivalent to a superclass. And then you have, let us say, subclass B dash. Right.

So from the inheritance point of view, you use the keyword implements. So B dash implements A dash. Right. And suppose your base is also an interface and derived is also an interface. That means the superclass is the interface A double dash and the derived class.

That means a subclass, which is also interface B double dash. Now you can say that B double dash extends A double dash. Right. So whenever you have the same. Right.

Interface

```
1 interface printable{
2     void print();
3 }
4
5 class A6 implements printable{
6     public void print(){System.out.println("Learning Interface @ IITR");}
7
8     public static void main(String args[]){
9         A6 obj = new A6();
10        obj.print();
11    }
12 }
```



Class. You use extends. Whenever it is different. You use implements. Right.

So we are going to have implements. So now. We will see with an example. So that will be very easier. Right.

So let us say printable. All right. The interface name of the interface is printable. Right. So how do you write it?

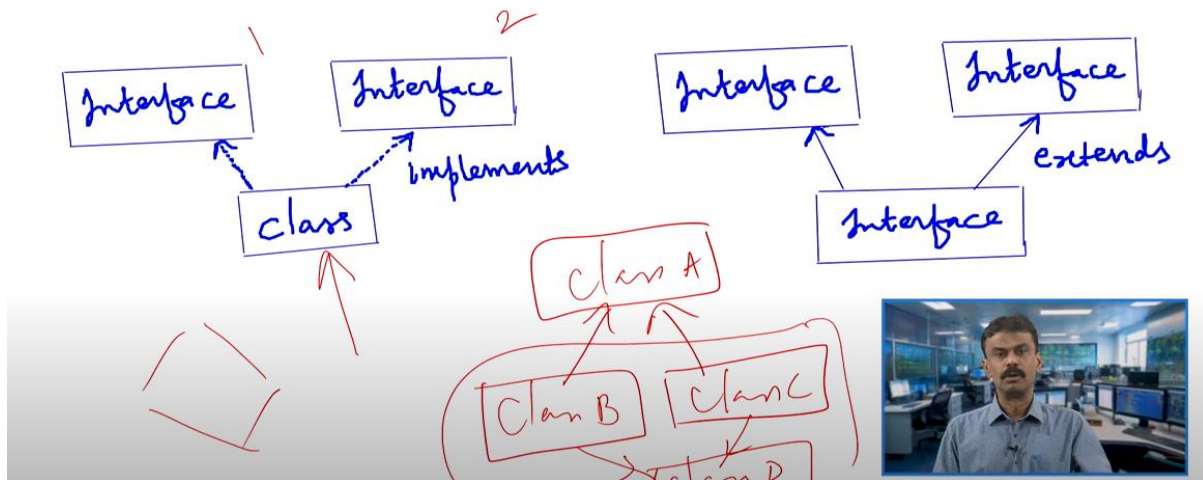
You have the keyword called interface. Right. You have the keyword called interface. And then you have the name of the interface, which is printable. So here you have void print.

Right. So here you have void print. So now class A6. Right. So now you have one class.

So one is an interface, another one is a class, all right. So the scenario will be like this, all right. So the scenario will be like this. So in that case, right, you have interface A-dash and class B-dash. So A-dash is nothing but printable, and B-dash is nothing but A6, right.

So class A6 implements printable. Class A6 is, class A6 implements printable. So here you have, you have a print method, right? So obviously, when I put void print, which is nothing but abstract, right? It is equivalent to abstract.

Multiple inheritance in Java by interface



Multiple inheritance in Java by interface

```
1 interface Printable{
2 void print();
3 }
4
5 interface Showable{
6 void show();
7 }
8
9 class A7 implements Printable, Showable{
10
11 public void print(){System.out.print("Hi, We are learning");}
12 public void show(){System.out.println(" Multiple Inheritance through JAVA interface @ IITR");}
13
14 public static void main(String args[]){
15 A7 obj = new A7();
16 obj.print();
17 obj.show();
18 }
19 }
```

A hand-drawn diagram shows a box labeled 'A7' with two arrows pointing to boxes labeled 'P' and 'S'. The 'P' box is labeled 'I1' and the 'S' box is labeled 'I2'.

So any method you write inside the interface is an abstract method. We know the definition of an abstract method, right? So now, based on that, let us consider you have in the subclass print. So your System.out.println learning interface, let us say at IITR. Learning interface at IITR, that is the print statement you are going to give.

So now I have created an object obj right under the class A6, right under the class A6, dynamically allocating memory, and my constructor is A6, right. So now the object obj under class A6 is invoking print, right. So when it is invoking print, so naturally So, this is the interface, 100% abstraction. Obviously, line number 2 should be overridden.

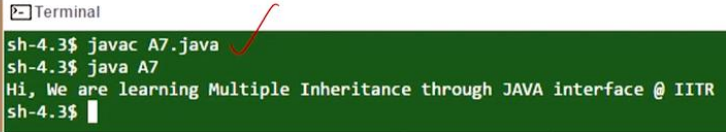
So, when it is overriding, it is going to this particular method, right? Void print method, and it is printing 'learning interface at IITR,' right? It is printing 'learning interface at IITR.' So, when I run this code, Right, when I run this code, so

compilation A6.java is the name of the function, and when I run this code, I will get 'learning interface at IITR,' right?


So you are getting the output, right? So this is how a simple case, what we have studied from this diagram, right? So you see this scenario, right? Printable is a dash, and A6 is b dash, right? So class b dash implements a dash. That is what you are writing. Class A6 implements printable. So, here you have the interface keyword, right?

Multiple inheritance in Java by interface

```
1 interface Printable{
2     void print();
3 }
4
5 interface Showable{
6     void show();
7 }
8
9 class A7 implements Printable,Showable{
10
11     public void print(){System.out.print("Hi, We are learning");}
12     public void show(){System.out.println(" Multiple Inheritance through JAVA interface @ IITR");}
13
14     public static void main(String args[]){
15         A7 obj = new A7();
16         obj.print();
17         obj.show();
18     }
19 }
```



```
Terminal
sh-4.3$ javac A7.java
sh-4.3$ java A7
Hi, We are learning Multiple Inheritance through JAVA interface @ IITR
sh-4.3$
```



So, here you have the interface keyword, and printable is the name of the interface. It contains void print, and then here, line number 5 is a syntax class A6 implements printable, right? So, when you have this, you can see how you achieve multiple inheritance in Java, right? How do you achieve multiple inheritance in Java, alright. So, based on the previous example, now you are going one step ahead.

So, suppose I have, let us say, an interface like this, and then the class is going to use or going to implement interface 1 and interface 2, alright. So, we can achieve multiple inheritance. So, when I define in terms of interface, let us say one interface I am defining, another interface, so I can have this multiple inheritance. In fact, in Java, the diamond problem will be solved. Assume that hierarchical is not a problem. Suppose I have class A, so this can be inherited by class B and class So, this was not a problem.

Whereas this is a problem when you have class C and class D. This is possible in C++, but this is not possible. So, that means this particular part. So, the shape is

looking like a diamond. So, you call it the diamond problem. So, the interface will solve this diamond problem.

So, when you have interface 1 and interface 2, then the class can implement them. The class implements interface 1, interface 2. And if you are using all interfaces, the interface extends. We use 'extends' if they are the same. If both are the same, yes, all are interfaces.

So, interface extends interface 1 and interface 2. So that is the meaning here. Let us see this with an example. So, we call it multiple inheritance in Java. So, let us have an interface called Printable.

So, let us have one interface called Printable. I simply write P. P stands for Printable. And another interface called Showable. Those are interfaces. So, I will write I1 and I2.

One is I1, interface 1. Another one is I2, showable. So, now the class A7. So, now the class A7 implements Printable and Showable, right? Class A7 implements Printable and Showable.

Class A7 implements Printable and Showable, right? So, in fact, when you look, interface Printable, you have one abstract method, void print. You have one abstract method called void print. And showable, you have a method called show. Right.

Abstract method. Right. So, an interface means that the method is abstract. Print and show. So this has to be extended in A7.

Class A7. That means we have to define this. So, public void print. You have System.out.println. One statement.

Let us say the statement is 'Hi.' We are learning. All right. Hi. We are learning.

And another statement. Show another function. Another method. Show, right? So, the show method has a print statement: 'Multiple inheritance through Java interface at IIT R, right? Multiple inheritance through Java interface at IIT R, right?'

So, now we are creating an object obj. Right now, the scenario is clear, right? So, here you have the print function, the print method. Which is the abstract method and show method, and A7 contains both, and then you are defining also, alright. So, you are defining show also. You are defining show also.

So, now assume that you are creating an object obj, right? So, let us assume that you are creating an object obj. So, obj is under A7. Right, I am creating an object obj whose reference is also A7. Right, so now this obj is invoking print.

Right, obviously it will be overridden, and this should be printed first. Hi, we are learning. Right, and then next obj is invoking show. Right, so next obj is invoking show. Right, so when it is invoking show, if you look at the class A7.

Right, if you look at class A7, so I am finding the right one print statement multiple inheritance to Java interface at IITR, right. So, this will be called, and when I run the code, I have to get the output like this, right? So, I have to get the output like this. So, I am compiling javac A7.java, then next running. So, you can see the output over here, right. We are learning multiple inheritance through Java interface at IITR. So, that will be printed, right.

So, the code is working fine, and you can see, right. So, the scenario we had seen. So, we have Printable and Showable. Now, in line number 9, the first time we are seeing class A7 implements Printable, Showable. So, in fact, if you look at C++, we have done this, right?


So, in C++, I mean, we can write both Printable and Showable as a class, right? So, assume that, what do you do? You replace interface with class or abstract class, right? And similarly, this, you will get an error, right? So, the overridden is not possible.

Whether you put simply class or abstract class, right? It will give an error because of line number 9. So, how do you achieve this? Put interface. Instead of class, use interface.

So, when you are using interface, right? So, now I can write the syntax in line number 9. So, the class A7 implements Printable and Showable, right? The class A7 implements Printable and Showable, right? So, we know that, right?

Multiple inheritance

```
1 interface Printable{
2     void print();
3 }
4
5 interface Showable{
6     void print();
7 }
8
9 class A8 implements Printable, Showable{
10
11     public void print(){System.out.println("CSN-103 @ IITR");}
12
13     public static void main(String args[]){
14         A8 obj = new A8();
15         obj.print();
16     }
17 }
```



So, we know when you are using only class, you cannot have multiple inheritance, but whereas It is possible by Interface only. Right. So the reason is.

The. Interface. Right. Which is supporting. Right.

The. I mean to say. The multiple inheritance. Concept. Is supported in the case of.

Interface. Because. I mean here there is no ambiguity. Right. As implementation is being provided.

By the implementation class. Right. So, therefore. You can have the multiple inheritance concept. So, what we had seen like this, okay.

So, maybe I will conclude with this particular example. So, it is similar, right? You have a printable interface and a showable interface, and then A8, right. So, this is I1, and this is I2, and in fact, this is a class. So, this class implements printable and showable. All right.

So here, I have both print statements. In the previous example, I had one print and one show. Right. So here, I have both as print statements. Right.

And then here it is implementing both Printable and Showable. A8 is implementing both Printable and Showable. And you have a void print statement:

System.out.println('CSN-103 at IITR is being printed'). So now I am creating an object obj under A8. Right, dynamically allocating memory with the constructor A8, and when object obj under A8 invokes the print method, right, so that will print 'CSN-103 at IITR', right.

So, this you will get as an output. When I run the code, I will get this as an output, right. So, in this chapter, we talked extensively a little bit about abstract class, and now you know what you mean by interface, right? So, in the whole chapter, we have seen encapsulation, right? How do you use getter and setter methods, how do you use access modifiers, right? And then how to prevent, how to give security for certain data members,

and how can I access those private data members from outside the class with the help of setter and getter methods, right? So then later we talked about abstraction, all right, which is 0 to 100 percent we can achieve, right, with the help of abstract class in Java, pure virtual function in C++, and finally, we concluded with the interface, right. With this, I conclude this class. Thank you.