Super keyword in Java

super keyword

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

➢ **Usage of Java super Keyword**

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

super is used to refer immediate parent class instance variable.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
black
white
```

super can be used to invoke parent class method

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
eating...
barking...
```

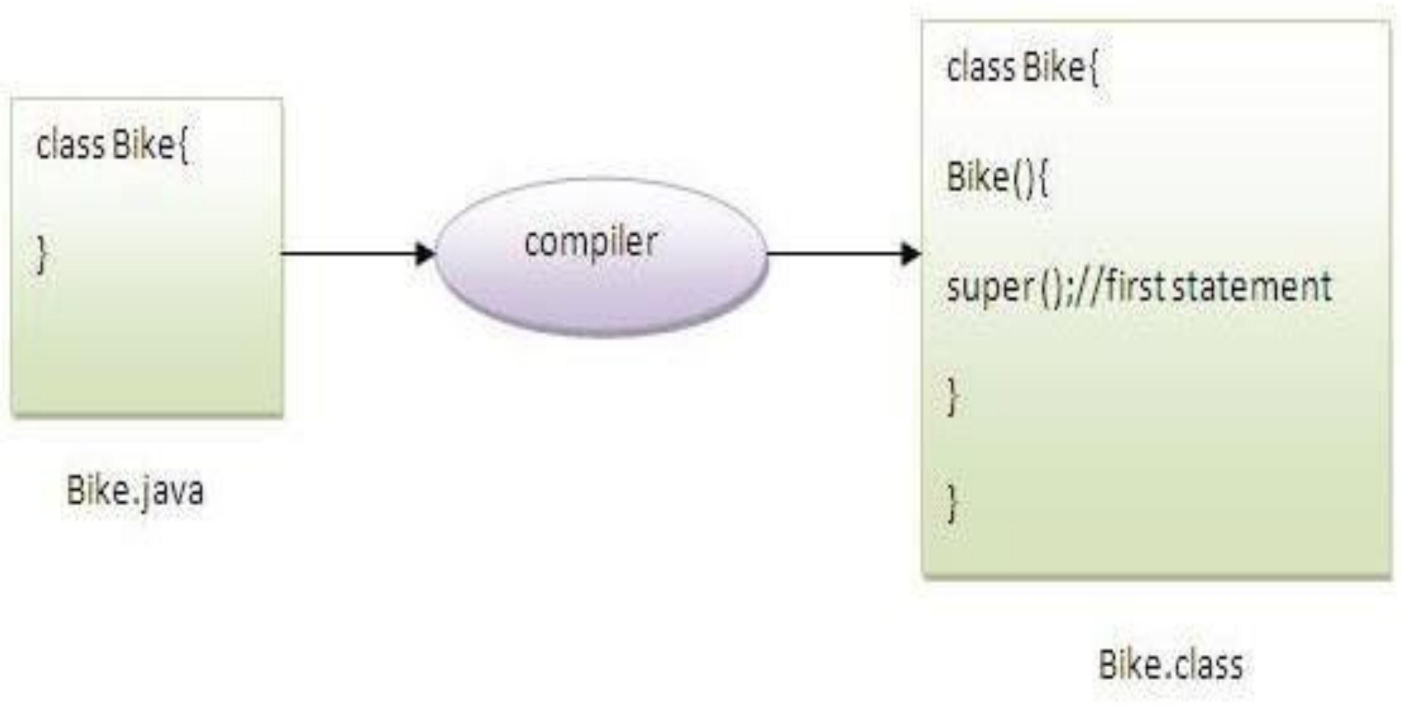
super is used to invoke parent class constructor.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

The super keyword can also be used to invoke the parent class constructor.

```
animal is created
dog is created
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



example of super keyword where super() is provided by the compiler implicitly.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
System.out.println("dog is created");
}
}
class TestSuper4{
public static void main(String args[]){
Dog d=new Dog();
}}
```

```
animal is created
dog is created
```

Example

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary)
{
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}
}
```

```
1 ankit 45000
```

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

➢Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Understanding the problem without method overriding

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
  }
}
```

```
Vehicle is running
```

Example of method overriding

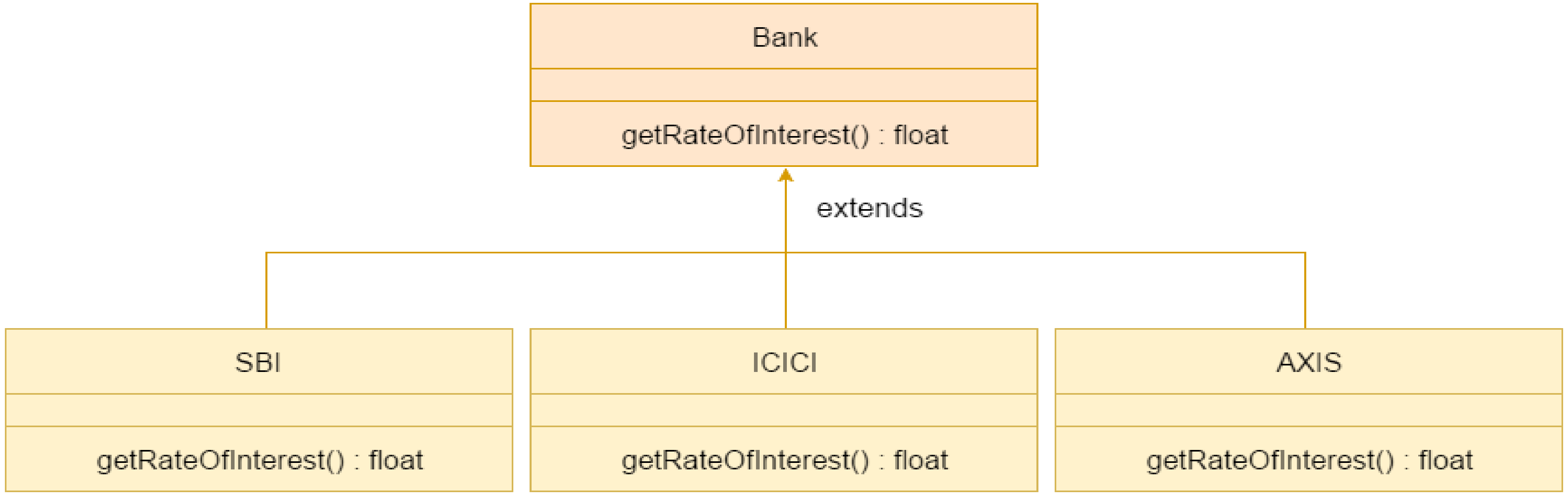
```
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

```
Bike is running safely
```

A real example of Java Method Overriding

- Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

Example

```
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
```

```
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRate
OfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRate
OfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRat
eOfInterest());
}
}
```

```
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Interview Questions

- ➢ Can we override static method?
- No, a static method cannot be overridden. It can be proved by runtime polymorphism.
- Why can we not override static method?
- It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

- Can we override java main method?
- No, because the main is a static method.

Cyclic Inheritance

Cyclic inheritance :

Cyclic inheritance is not allowed in java.

Example 1:

```
class A extends B{}  } (invalid)
class B extends A{}  } C.E:cyclic inheritance involving A
```

$$\begin{tikzcd} & B \arrow[dd, bend left] \\ & \\ & A \arrow[uu, bend left] \end{tikzcd}$$

Example 2:

class A extends A{} $\xrightarrow{C.E}$ cyclic inheritance involving A

Abstract Classes

- The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes).
- We use the abstract keyword to declare an abstract class.
- **Example**

```
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

- An abstract class can have both the regular methods and abstract methods.
- **Example:**

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Abstract Method

- A method that doesn't have its body is known as an abstract method.
- We use the same abstract keyword to create abstract methods.
- **Example:**

```
abstract void display();
```

- If a class contains an abstract method, then the class should be declared as abstract. Otherwise, it will generate an error.
- **Example**

```
// class should be abstract
class Language {

  // abstract method
  abstract void method1();
}
```

- Though abstract classes cannot be instantiated, we can create subclasses from it.
- We can then access members of the abstract class using the object of the subclass.

Example:

```
abstract class Language {
  // method of abstract class
  public void display() {
    System.out.println("This is Java Programming"); }
  }
}
class Main extends Language {
  public static void main(String[] args) {

    // create an object of Main
    Main obj = new Main();

    // access method of abstract class
    // using object of Main class
    obj.display();
  }
```

```
This is Java programming
```

Implementing Abstract Methods

- If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method.

Example:

```java
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}
class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}
class Main {
  public static void main(String[] args) {

    // create an object of Dog class
    Dog d1 = new Dog();
    d1.makeSound();
    d1.eat();
  }
}
```

```
Bark bark
I can eat.
```

Note:

- If the Dog class doesn't provide the implementation of the abstract method makeSound(), Dog should also be declared as abstract.
- This is because the subclass Dog inherits makeSound() from Animal.

Abstraction Advantages:

- The major use of abstract classes and methods is to achieve abstraction in Java.
- Abstraction is an important concept of object-oriented programming that allows us to hide unnecessary details and only show the needed information.
- This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea.

Static Methods in Abstract Class Can be called Independently

```
abstract class Helper {

        static void demofun()
        {
        System.out.println(“RVR & JC");
        }
}
public class GFG extends Helper {

        public static void main(String[] args)
        {
        Helper.demofun();
        }
}
```

```
RVR & JC
```

Abstract Classes Vs Abstract Methods

Abstract classes	**Abstract methods**
Abstract classes can’t be instantiated.	**Abstract method bodies must be empty.**
Other classes extend abstract classes.	**Sub-classes must implement the abstract class’s abstract methods.**
Can have both abstract and concrete methods.	**Has no definition in the class.**

Constructors in Abstract class

```
abstract class Content {
        int a;

        public Content(int a)
        {

        this.a = a;
        }
        abstract int multiply(int val);
}

class GFG extends Content {
        GFG()
        {

        super(2);
        }

        public int multiply(int val)
        {
        return this.a * val;
        }
}

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

                Content c = new GFG();

                System.out.println(c.multiply(3));
        }
}
```

```
6
```

Example:

```
abstract class Shape{
abstract void draw();
}

class Rectangle extends Shape{
void draw()
{System.out.println
("drawing rectangle");
}
}
class Circle1 extends Shape{
void draw()
{
System.out.println
("drawing circle");
}
}
class TestAbstraction1{
public static void main(String args[])
{
Shape s=new Circle1();
s.draw();
}
}
```

Properties of an abstract class:

- An abstract can have an abstract and a non-abstract method.
- It must be declared with an abstract keyword.
- It can have a constructor, static method.
- It can have a final method that prevents child class of abstract class not to change the body of the method
- The abstract method contains no-body or in simple words, you can say that you can’t define an abstract method inside an abstract class. We can define an abstract method inside the derived class of its abstract class.
- The object of the abstract class can’t be instantiated it means you can’t create an abstract class object directly but you can create its object by reference to its child class.

Interface in Java

- An **interface in Java** is a blueprint of a class.
- It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables.
- It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.

- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

Why to use Java interface?

- There are mainly three reasons to use interface.

1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling.

Realtime Example 1:

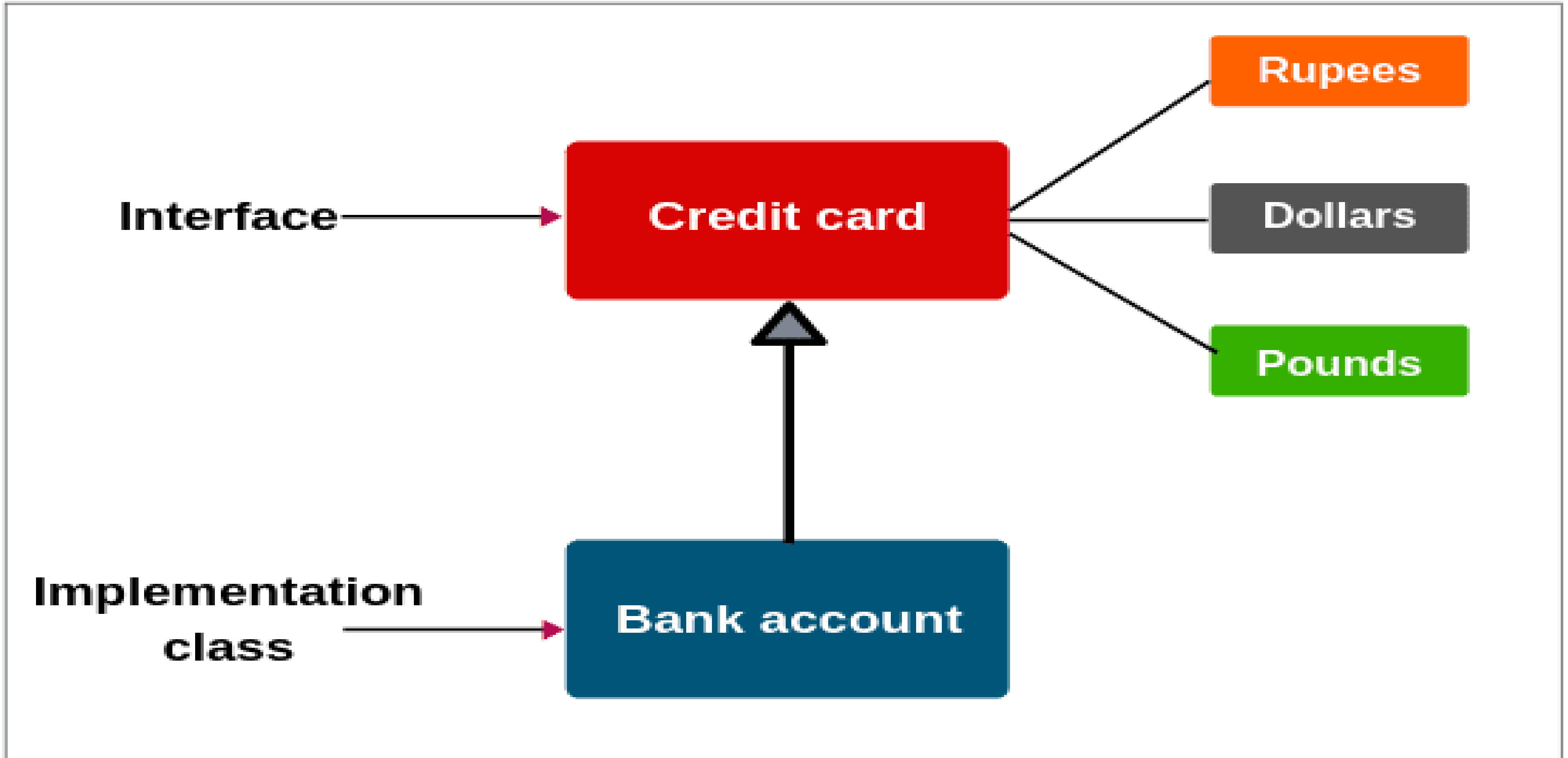


Fig: Interface and Implementation class

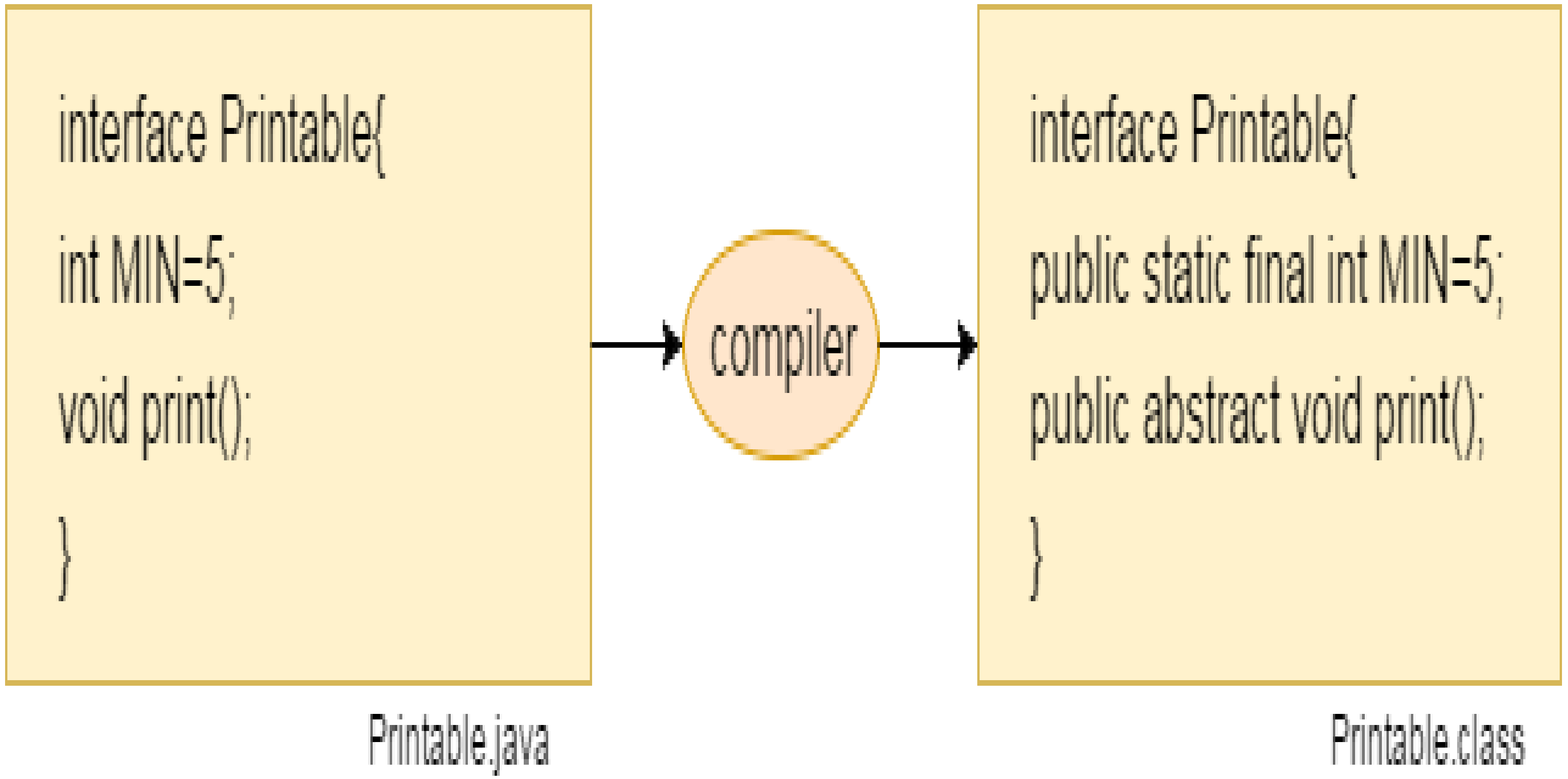
How to declare an interface?

- 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, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.
- **Syntax:**

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Interface fields are public, static and final by default, and the methods are public and abstract.

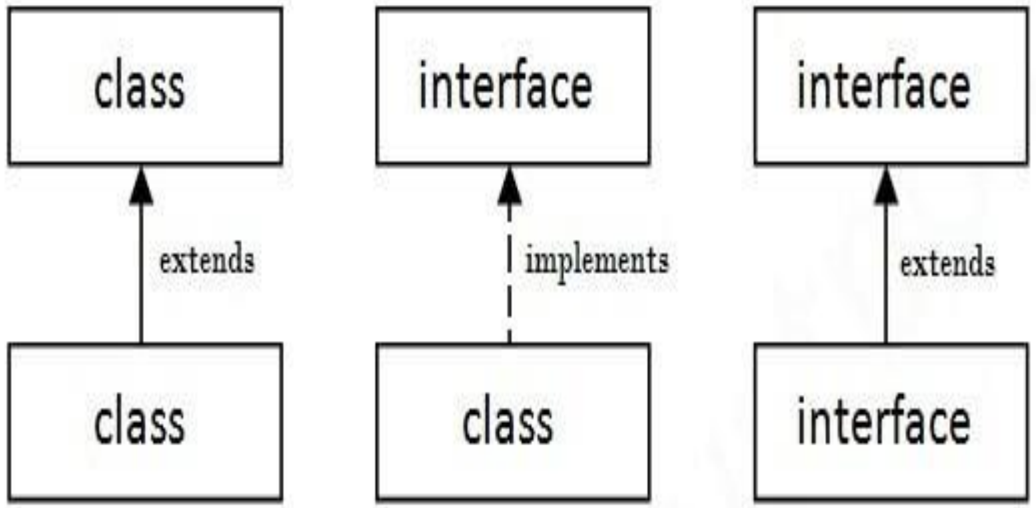


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

The relationship between classes and interfaces

- As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



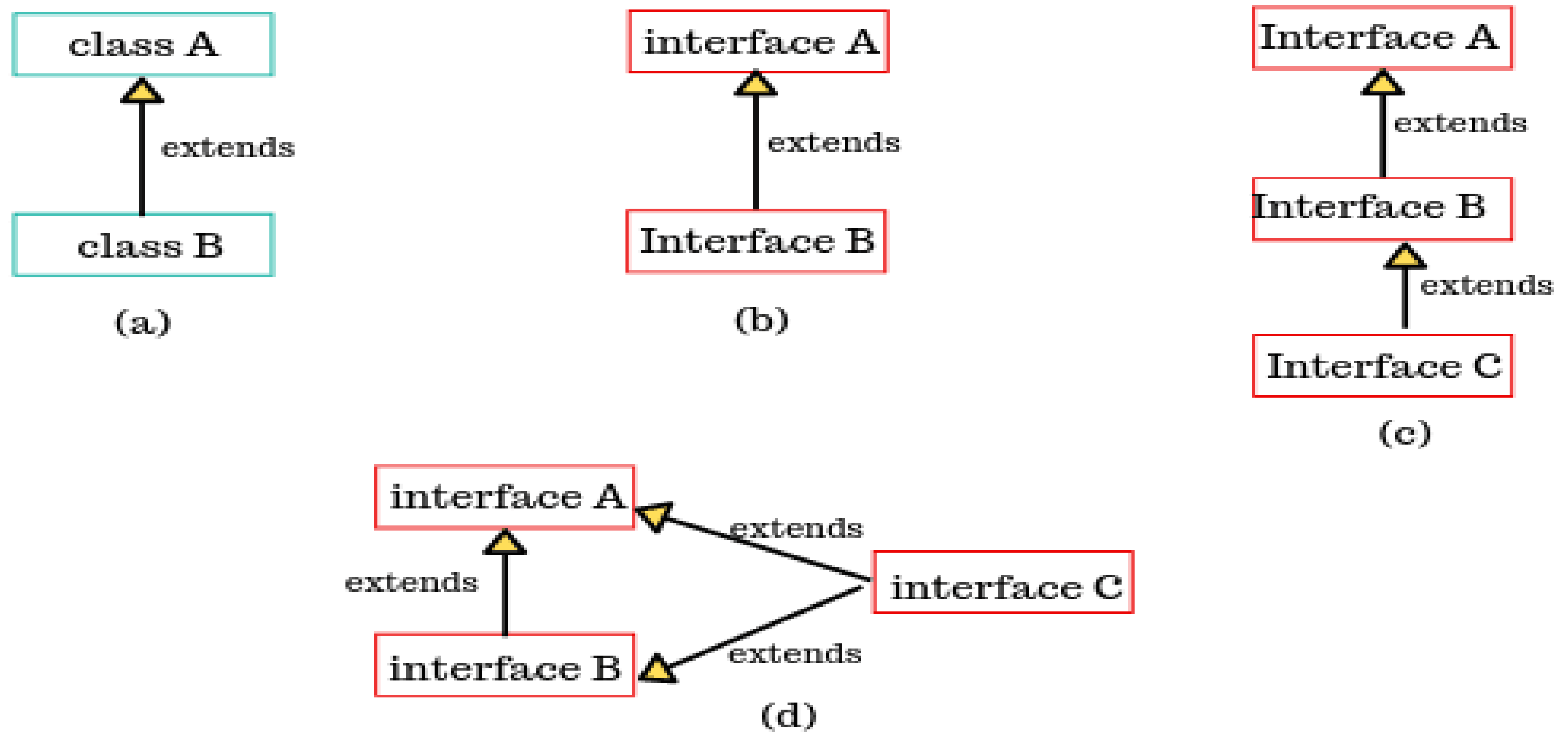
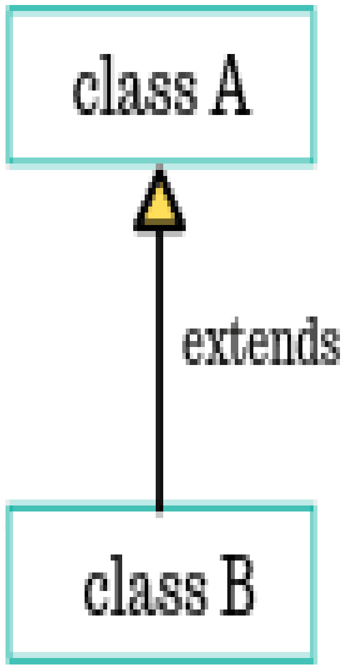


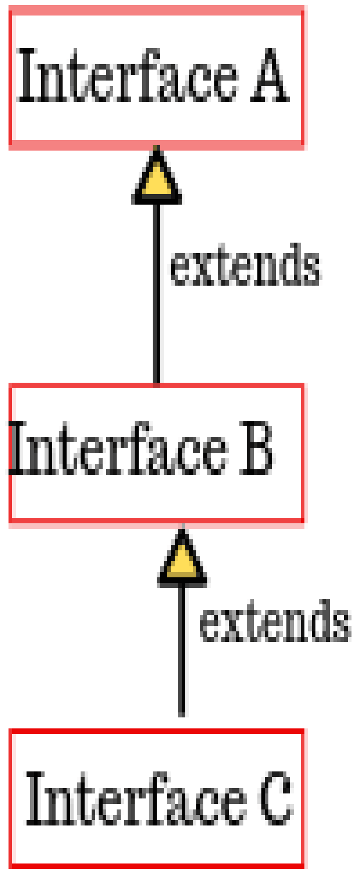
Fig: Various forms of extending Interface in Java

Examples:



```
interface A
{
  int x = 10;
  int y = 20;
 }
interface B extends A
{
  void show();
}
```

Example:



```
interface A
{
 int x = 20;
 int y = 30;
}
interface B extends A
{
 void show();
}
interface C extends A, B
{
 . . . . . . . .
}
```

Various forms of interface implementations

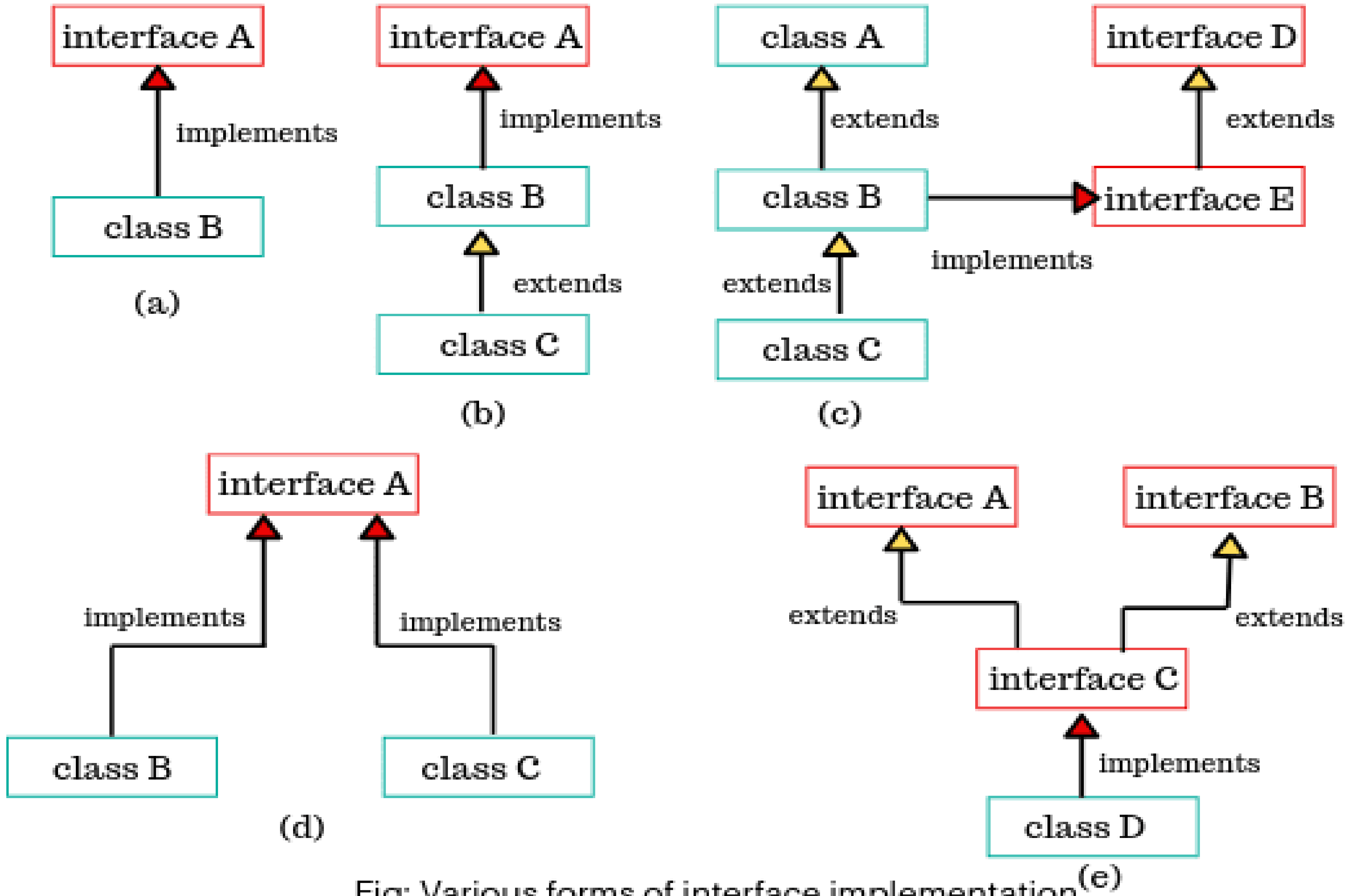


Fig: Various forms of interface implementation

Interface Example

```
interface printable{
void print();
}
class A implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A obj = new A();
obj.print();
 }
}
```

```
Hello
```

Example2:

```
interface Drawable{
void draw();
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();
d.draw();
}}
```

```
drawing circle
```

How to extend Interfaces in Java?

```
interface A {
  void funcA();
}
interface B extends A {
  void funcB();
}
class C implements B {
  public void funcA() {
    System.out.println("This is funcA");
  }
  public void funcB() {
    System.out.println("This is funcB");
  }
}
public class Demo {
  public static void main(String args[]) {
    C obj = new C();
    obj.funcA();
    obj.funcB();
  }
}
```

```
This is funcA
This is funcB
```

multiple classes implement the same interface

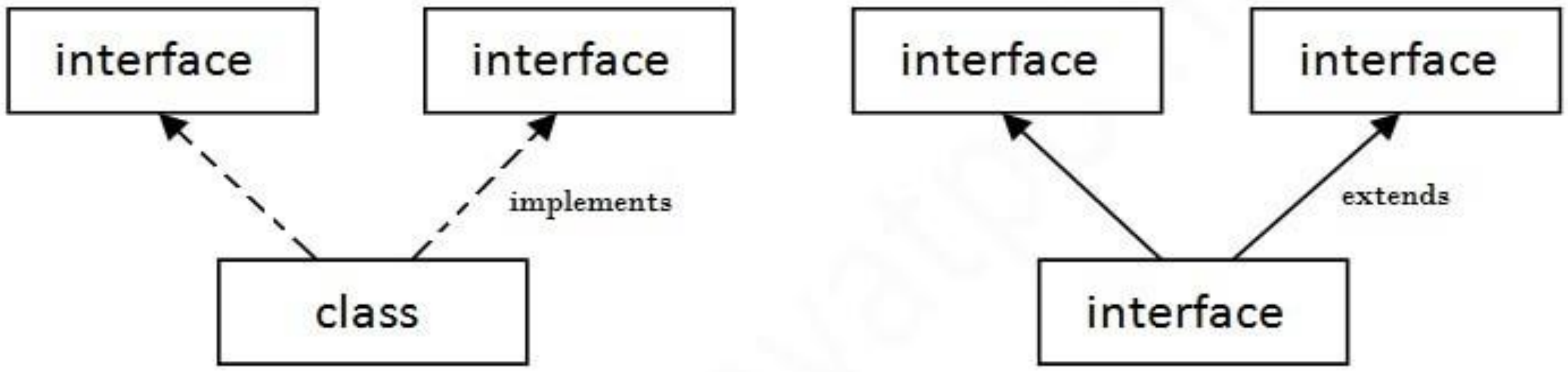
```
public interface ConstantValues
{
// Declaration of interface variables.
   int x = 20;
   int y = 30;
 }
public class Add implements ConstantValues
{
  int a = x;
  int b = y;
 void m1()
 {
   System.out.println("Value of a: " +a);
   System.out.println("Value of b: " +b);
  }
void sum()
{
  int s = x + y;
  System.out.println("Sum: " +s);
 }
}
```

```
public class Sub implements ConstantValues
{
  void sub()
  {
    int p = y - x;
    System.out.println("Sub: " +p);
  }
}
public class Test
{
  public static void main(String[] args)
  {
    Add a = new Add();
     a.m1();
     a.sum();
   Sub s = new Sub();
     s.sub();
   }
}
```

```
Value of a: 20
  Value of b: 30
      Sum: 50
      Sub: 10
```

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

Multiple Inheritance Example

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A obj = new A();
obj.print();  obj.show();
 }
}
```

```
Hello
Welcome
```

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

- multiple inheritance is not supported in the case of class because of ambiguity.
- However, it is supported in case of an interface because there is no ambiguity.
- It is because its implementation is provided by the implementation class.

Example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

```
Hello
```

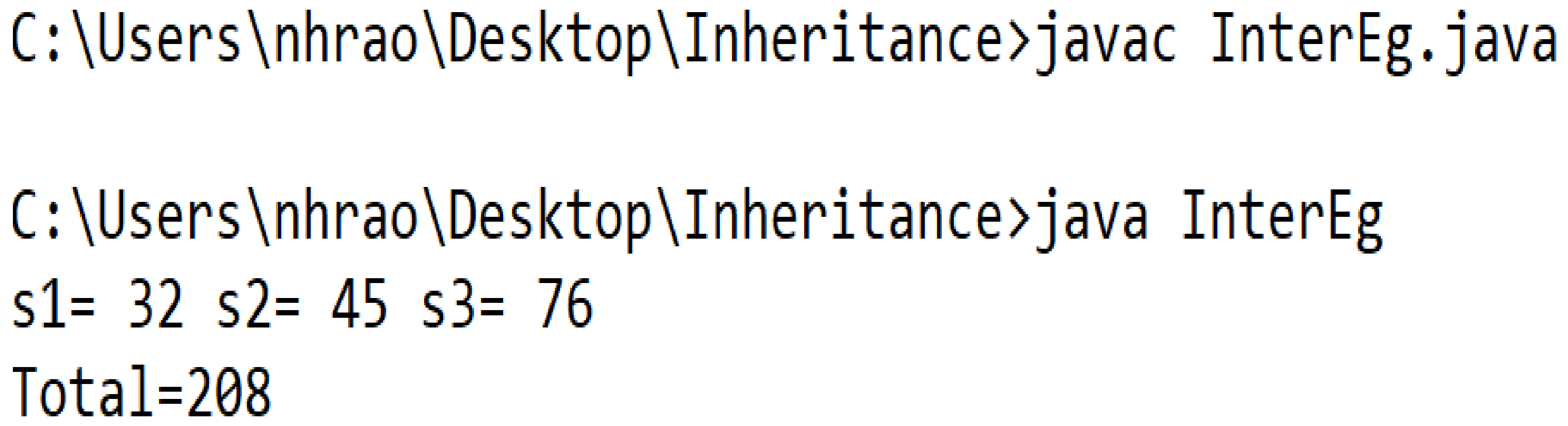
Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
7) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.

8)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Multiple Inheritance with one base class & interface Example

```
class Test
{
int s1,s2,s3;
void readdata1(int x,int y,int z)
{ s1=x; s2=y;s3=z; }
void printdata1()
{
System.out.println("s1= "+s1+" s2= "+ s2+" s3= "+s3);
}
}
interface Sports
{
int smarks=55;
public void printdata2();
}
class Result extends Test implements Sports
{
int sum;
public void printdata2()
{
printdata1();
sum=s1+s2+s3+smarks;
System.out.println("Total="+sum);
}
}
class MulInh
{
public static void main(String args[])
{
Result obj=new Result();
obj.readdata1(32,45,76);
obj.printdata2();
}
}
```

output



```
C:\Users\nhrao\Desktop\Inheritance>javac InterEg.java

C:\Users\nhrao\Desktop\Inheritance>java InterEg
s1= 32 s2= 45 s3= 76
Total=208
```