Sri SAIRAM
COLLEGE OF ENGINEERING
Anekal, Bengaluru

Accredited by NAAC
ISO 9001:2015 Certified Institution
Approved by AICTE, New Delhi
Affiliated to Visvesvaraya Technological University
www.sairamce.edu.in

Object Oriented Programming with JAVA

(BCS306A)

ACADEMIC YEAR 2024 - 25

Lecture notes

Name Of the Programme	**B.E -AIML**
Scheme	**2022**
Year and Semester	**II Year III Semester**
Subject Code	**BCS306A**
Name of the Faculty	**KALIDASS M**

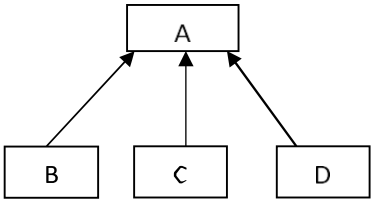
MODULE III

Inheritance: Inheritance Basics, Using super, Creating a Multilevel Hierarchy, When Constructors Are Executed, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, Local Variable Type Inference and Inheritance, The Object Class.

Interfaces: Interfaces, Default Interface Methods, Use static Methods in an Interface, Private Interface Methods.

INHERITANCE

- Inheritance is the process by which one class acquires the properties (instance variables, methods) of another class.
- Inheritance creates an IS-A relationship , meaning the child is a more specific version of the parent.



Super class / Base class/ Parent class

Sub class / Derived class / Child class

Fig: Inheritance

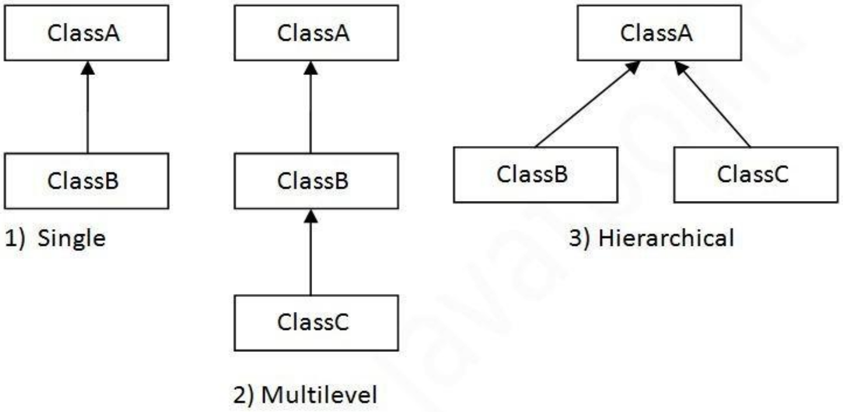
Main purpose of Inheritance:

1. Reusability.
2. Abstraction.

- “extends” keyword is used to inherit the properties from one class to another class .

➢ TYPES OF INHERITANCE:

- Single Inheritance.
- Multilevel Inheritance.
- Hierarchical Inheritance.
- Multiple Inheritance
  - J AVA does not support , need to use Interface.
- Hybrid Inheritance



- **Program for Single Inheritance :**

```
/* single inheritance */
import java.io.*;
class A
 {
   public int i;
   public A()
    {
      System.out.println(" \n \t default constructor A() is called");
      i=10;
    }
   public void Adisplay()
    {
      System.out.println(" \n \t in A class i= " +i);
    }
 }
class B extends A
 {
   public int j;
   public B()
    {
      System.out.println(" \n \t default constructor B() is called");

      j=20;
    }

   public void Bdisplay()

    {

      j=i+1;

      System.out.println(" \n \t in B class j= "+j);

    }

 }

class Simple
```

```
{
  public static void main(String arr[])
   {
     System.out.println(" \n \t start of main()");


     B b=new B();
     b.Adisplay();
     b.Bdisplay();
     System.out.println(" \n \t end of main()");
   }
}
```

OUTPUT :

```
start of main()
default constructor A() is called
default constructor B() is called
in A class i= 10

in B class j= 11
end of main()
```

- Program for Multilevel Inheritance :

```
/* multilevel inheritance */
import java.io.*;
class A
 {
    public int i;
    public A()
     {
        System.out.println("\n \t constructor A() is called");
        i=10;
     }
    public void Adisplay()
     {
        System.out.println(" \n \t in A class i= "+i);
     }
 }
class B extends A
 {
    public int j;
    public B()
     {
        System.out.println("\n \t constructor B() is called");
        j=20;
     }
    public void Bdisplay()
     {
        j=i+1;
        System.out.println(" \n \t in B class j= "+j);
     }

 }
class C extends B
 {
    public int k;
    public C()
     {
        System.out.println("\n \t constructor C() is called");
        k=30;
```

```
      }
     public void Cdisplay()
      {
         k=i+j;
         System.out.println(" \n  \t  in  C  class  k=  "+k);
      }
  }
  class MulLevel
   {
      public static void main(String arr[])
       {
         System.out.println(" \n  \t start of main()");

         C c=new C();
         c.Adisplay();
         c.Bdisplay();
         c.Cdisplay();
         System.out.println(" \n  \t end of main()");
       }
       }
```

OUTPUT :

```
   start of main()
   constructor A() is called
   constructor B() is called
   constructor C() is called

   in A class i= 10
   in B class j= 11
   in C class k= 21
   end of main()
```

When Constructors are called

When class hierarchy is created (multilevel inheritance), the constructors are called in the order of their derivation. That is, the top most super class constructor is called first, and then its immediate sub class and so on.

```
class A
{
A()
{
System.out.println("A's constructor.");
}
}
class B extends A
{
B()
{
System.out.println("B's constructor.");
}
}
class C extends B
{
C()
{
System.out.println("C's constructor.");
}
}
     class CallingCons
     {
       public static void main(String args[])


       {
         C c = new C();

       }

      }
```

Output:

```
     A's constructor
     B's constructor
     C's constructor
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to ***override*** the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

```
class A
{
     int i, j;
     A(int a, int b)
     {
       i = a;

       j = b;

}

void show() //suppressed

{

System.out.println("i and j: " + i + " " + j);
}

}

class B extends A
{
     int k;
     B(int a, int b, int c)
     {
        super(a, b);

        k = c;

     }

void show() //Overridden method

{

System.out.println("k: " + k)
}

}

class Override
```

```
{
    public static void main(String args[])
    {
     B subOb = new B(1, 2, 3);
      subOb.show();
    }

}
```

Output:
k: 3

A Super class Variable Can Reference a Subclass Object

- A reference variable of a super class can be assigned a reference to any subclassderived from that super class.
- You will find this aspect of inheritance quite useful in a variety ofsituations .For example, consider the following:

```
class Base
{
 void dispB()
 {
      System.out.println("Super class " );
 }
}
class Derived extends Base
{
  void dispD()
 {
    System.out.println("Sub class ");
 }
}
class Demo
{
  public static void main(String args[])
  {
    Base b = new Base();
    Derived d=new Derived();
    b=d; //superclass reference is holding subclass object
    b.dispB();
   //b.dispD(); error!!
   }
 }
```

➢ SUPER KEYWORD:

- ‘super’ is used when a subclass wants to refer to its **immediate** superclass members.
- ‘super’ has two general forms.
  - ✓ To make a call to the super class constructor from sub classconstructor.
  - ✓ The second is used to access a member of the superclass that has been hidden by a member of a subclass i.e To access superclass member (variable or method) when there is a duplicate member name in the subclass

program for super keyword :

```
/* super keyword */
import java.io.*;

class A
 {
    public int i;
    public A()
     {
        System.out.println(" \n \t constructor A() is called");
        i=10;
     }
    public void display()
     {
        System.out.println(" \n \t in A class i= " +i);
     }
}
class B extends A
 {
    public int i;

    public B()
     {
        super();
        System.out.println(" \n \t constructor B() is called");
        this.i=20;
        super.i=30;              // points the super class instance variable.
     }
    public void display()
     {
        super.display();         // calling super class method.
        this.i=this.i+super.i;
        System.out.println(" \n \t in B class subofi+supofi = "+this.i);
     }
}
 class Super
  {
     public static void main(String arr[])
      {
        System.out.println(" \n \t start of main()");
        B b=new B();
        b.display();
        System.out.println(" \n \t end of main()");
      }
  }
```

OUTPUT :

```
start of main()
constructor A() is called
constructor B() is called
in A class i= 30
in B class
subofi+supofi = 50end
of main
```

Using Abstract Classes

```
abstract class A

{

            abstract void
            callme();void
            callmetoo()
            {
               System.out.println("This is a concrete method.");

            }

}

class B extends A
     {

void callme() //overriding abstract method

{

               System.out.println("B's implementation of callme.");

}

}

     class AbstractDemo
     {
           public static void main(String args[])
           {
           B b = new B(); //subclass object b.callme(); //calling abstract
```

```
        method b.callmetoo(); //callingconcretemethod
}

}
```

Example: Write an abstract class *shape,* which has an abstract method *area()*. Derive three classes *Triangle, Rectangle* and *Circle* from the *shape* class and to override *area()*. Implement run-time polymorphism by creating array of references to supeclass. Compute area of different shapes and display the same.

Solution:

```
    abstract class Shape
    {
          final double PI=
          3.1416;abstract
          double area();

    }

    class Triangle extends Shape
    {
          int b, h;
          Triangle(int x, int y)
          {//constr

               uct

               or

               b=x

               ;

               h=y

               ;

          }

          double area()
          {
          //method overriding
             System.out.print("\nArea of Triangle

             is:");return 0.5*b*h;

          }

     }
     class Circle extends Shape
     {
     int r;

       Circle(int rad)
       {

           r=rad;

       }

       double area()
       {
         //constructor
         //overriding
         System.out.print("\nArea of Circle
         is:");return PI*r*r;

        }

      }

class Rectangle extends Shape
{
     int a, b;
     Rectangle(int x, int y)
     {
     a=x;
     b=y;
      }

     double area()
     {
       //constructor
       //overriding
       System.out.print("\nArea of Rectangle
       is:");return a*b;
     }
}
     class AbstractDemo
     {
        public static void main(String args[])
       {

        Shape r[]={new Triangle(3,4), new Rectangle(5,6),new Circle(2)};

           for(int i=0;i<3;i++)
                System.out.println(r[i].area());
        }

     }
```

Output:
Area of Triangle

is:6.0 Area of

Rectangle is:30.0

Area of Circle

is:12.5664

Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch.* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Java implements run-time polymorphism using dynamic method dispatch. We know that, a superclass reference variable can refer to subclass object. Using this fact, Java resolves the calls to overridden methods during runtime. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

```
class A

{

  void callme()

   {

      System.out.println("Inside A");

   }
}
 class B extends A
 {
void callme()
{
   System.out.println("Inside B");
}
}

     class C extends A
     {
           void callme()

           {
              System.out.println("Inside C");
           }
     }
     class Dispatch
     {
        public static void main(String args[])
        {
           A a = new A();
           B b = new B();
           C c = new C();

           A r; //Superclass reference
           r = a; //holding subclass object

           r.callme();r = b;
           r.callme();r = c;
           r.callme();

      }

     }
```

The Object Class

There is one special class, ***Object***, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array. Object defines the following methods, which means that they are available in every object.

METHOD	**PURPOSE**
Object clone()	Creates a new object that is same as the object being cloned
boolean equals(Object ob)	Determines whether one object is equal to another
protected void finalize()	Called before an unused object is recycled
final class getClass()	Obtains the class of an object at runtime
int hashCode()	Returns the hashcode associated with the invoking object
void notify()	Resumes execution of a thread waiting on the invoking object

void notifyAll()	Resumes execution of all threads waiting on the invoking object
String toString()	Returns a string that describes the object

void wait()	Waits on another thread of execution
void wait(long milliseconds)	
void wait(long milliseconds, int nanoseconds)	

The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may override the others. The**equals( )** method compares the contents of two objects. It returns **true** if the objects are equivalent, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automaticallycalled when an object is output using **println( )**. Many classes override this method.

Interfaces

Interface is an abstract type that can contain only the declarations of methods and constants. Interfaces are syntactically like classes, but they do not contain instance variables, and their methods are declared without any body. Any number of classes can implement an interface. One class may implement many interfaces. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. Interfaces are alternative means for multiple inheritance in Java.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access-specifier interface <interface-name>
{
datatype varname1 = value;
datatype varname2 = value;
…………………
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
…………………
}
```

Few key-points about interface:

- When no access specifier is mentioned for an interface, then it is treated as default and the interface is only available to other members of the package in which it is declared. When an interface is declared as public, the interface can be used by any other code.

- All methods and variables are implicitly public.

- All the methods declared are abstract methods and hence are not defined inside interface. But a class implementing an interface should define all the methods declared inside the interface.

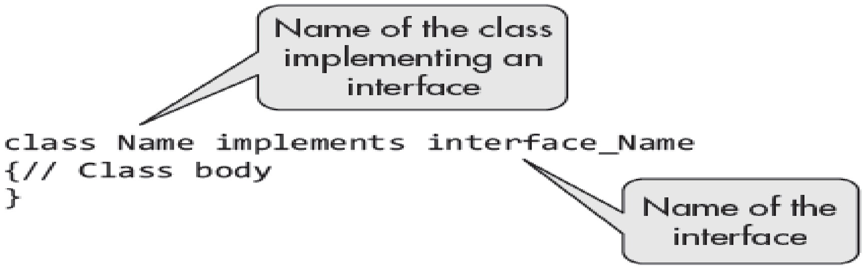
- Variables declared inside of interface are implicitly final and static, meaning they cannot be changed by the implementing class.

- All the variables declared inside the interface must be initialized.

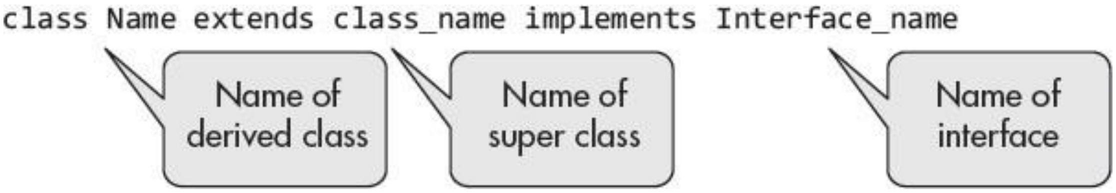
- An interface cannot implement itself; it must be implemented by a class.

Implementing Interface

To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The class that implements the interface has to have full definitions of all the abstract methods in the interface. The general form of a class that includes the implements clause looks like this:



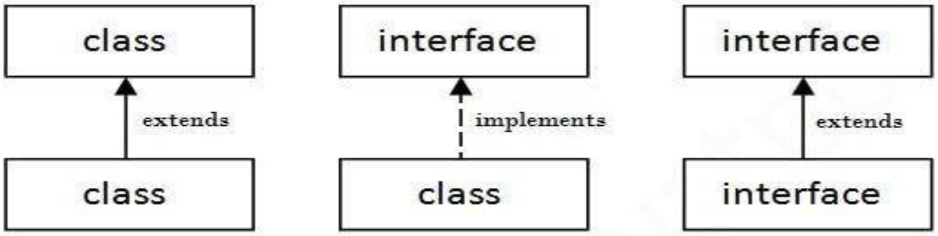
```
class Name implements interface_Name
{// Class body
}
```



```
class Name extends class_name implements Interface_name
```

class classname extends superclass implements interface1, interface2...
{
// class-body
}

The relationship between classes and interfaces



Consider the following example:

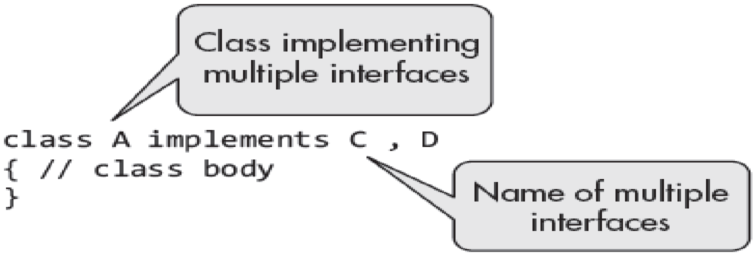
```
interface it1
{
        int x=10, y=20;
        public void add(int a, int b);
        public void sub(int a,  int b);
}

class demo implements it1
{
        public  void add(int s, int w)
        {
                System.out.println(“ Addition=” + (s+w));
        }
        public void sub(int s, int w)
        {
                System.out.println  (“Subtraction=  ” + (s-w));
        }
 public static void main(String args[])
 {
 demo obj=new demo( );
 obj.add(3,4);
 obj.sub(5,2);
 System.out.println(obj.x  +  obj.y);

 //obj.x=70;         // error since x is final variable in interface

}
```

Multiple Inheritance using Interfaces



```
class A implements C , D
{ // class body
}
```

```
interface X
{
     int x = 10;
     public void display();
}

interface Y
{
     int y  = 20;
     public void add();
}
class A implements X,Y
{
     public void display()
     {
     System.out.println("Class-B Method : x = "+ x + " y = " + y);
     }
     public void add()
     {
     System.out.println("Class-B Method : x+y = "+ (x+y));
     }
}

class InterfaceRef
{
     public static void main(String args[])
     {
     //Reference of X
     X objX = new A();
     objX.display();

     //Reference of Y
     Y objY = new A();
     objY.add();
      }
}
```

Output:

```
C:\ >javac InterfaceRef.java

C:\ >java InterfaceRef
Class-B Method : x = 10 y = 20
Class-B Method : x+y = 30
```

If a class includes an interface but does not fully implement the methods defined by that interface, then the class becomes abstract class and must be declared as abstract in the first line of its class definition.

Example:

```
interface it1
{
        int x=10, y=20;
        public void add(int a, int b);
        public void sub(int a, int b);
}
abstract class It2 implements it1
{
        public void add(int s, int w)
        {
                  System.out.println(“Addition=” + (s+w));

        }
}
class It3 extends It2
{
        public void sub(int s, int w)
        {
                System.out.println  (“Subtraction=” + (s-w));
        }
        public static void main(String args[])
         {
                It3 obj=new It2( );
                obj.add(5,6);
        }
}
```

Note: Interfaces may look like abstract classes. But there are lot of differences between them as shown in the following table:

Abstract Class	Interface
Abstract class can **have abstract and non- abstract** methods.	Interface can have **only abstract** methods. Since Java8, it can have default and static methods also.
Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
An **abstract class** can be extended using keyword extends.	An **interface class** can be implemented using keyword implements.
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

Interfaces can be extended / Inheritance of Interfaces

One interface can inherit another interface by using the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
interface A
{
void meth1();
void meth2();
}

interface B extends A
{
void meth3();
}
class MyClass implements B
{
public void meth1()
{
System.out.println("Implement meth1().");
}
public void meth2()
{
System.out.println("Implement meth2().");
}
public void meth3()
{
System.out.println("Implement meth3().");
}
}
class IFExtend
{
public static void main(String arg[])
{
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Default Interface Methods, Use static Methods in an Interface

Important Points:

- Interfaces can have default methods with implementation in Java 8 on later.
- Interfaces can have static methods as well, similar to static methods in classes.
- Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.
- Default methods are also known as defender methods or virtual extension methods.

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface. In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.

Default interface methods are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Therefore, we don’t need to modify the implementing classes.

Like regular interface methods, default methods are implicitly public; there’s no need to specify the public modifier.

Unlike regular interface methods, we declare them with the default keyword at the beginning of the method signature, and they provide an implementation.

```
// A simple program to Test Interface default methods in java
interface TestInterface
{
    public void square(int a);      // abstract method
    default void show()             // default method
    {
    System.out.println("Default Method Executed");
    }
}
class TestClass implements TestInterface
{
    public void square(int a)
    {
            System.out.println(a*a);
    }
    public static void main(String args[])
    {
            TestClass d = new TestClass();
            d.square(4);
            d.show();
    }
}
```

Output:

```
16

Default Method Executed
```

Static Methods:

The interfaces can have static methods as well which is similar to static method of classes.

```
// A simple Java program to demonstrate static methods in java
interface TestInterface
{
        public void square (int a);             // abstract method
        static void show()                      // static method
        {
                System.out.println("Static Method Executed");
        }
}
class TestClass implements TestInterface
{
        public void square (int a)
        {
                System.out.println(a*a);
        }
        public static void main(String args[])
        {
                TestClass d = new TestClass();
                d.square(4);
                TestInterface.show();
        }
}
```

Output:

```
 16

 Static Method Executed
```

Default Methods and Multiple Inheritance

In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

```
// A simple Java program to demonstrate multiple
// inheritance through default methods.
interface TestInterface1
{
	// default method
	default void show()
	{
		System.out.println("Default TestInterface1");
	}
}
interface TestInterface2
{
	// Default method
	default void show()
	{
		System.out.println("Default TestInterface2");
	}
}

// Implementation class code
class TestClass implements TestInterface1, TestInterface2
{
	// Overriding default show method
	public void show()
	{
		// use super keyword to call the show
		// method of TestInterface1 interface
```

```
        TestInterface1.super.show();
        // use super keyword to call the show
        // method of TestInterface2 interface
        TestInterface2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

Output:

```
Default TestInterface1
Default TestInterface2
```

Private Interface Methods

Rules For using Private Methods in Interfaces

- Private interface method cannot be abstract and no private and abstract modifiers together.
- Private method can be used only inside interface and other static and non-static interface methods.
- Private non-static methods cannot be used inside private static methods.
- We should use private modifier to define these methods and no lesser accessibility than private modifier.

```
// Java 9 program to illustrate default, static private methods in interfaces
public interface TempI
{
    public abstract void mul(int a, int b);
    public default void add(int a, int b)
    {
      sub(a, b);    // private method inside default method
      div(a, b);    // static method inside other non-static method
```

```
            System.out.print("Answer by Default method = ");
            System.out.println(a + b);
        }
        public static void mod(int a, int b)
        {
            div(a, b); // static method inside other static method
            System.out.print("Answer by Static method = ");
            System.out.println(a % b);
        }
        private void sub(int a, int b)
        {
            System.out.print("Answer by Private method = ");
            System.out.println(a - b);
        }
        private static void div(int a, int b)
        {
            System.out.print("Answer by Private static method = ");
            System.out.println(a / b);
        }
    }
    class Temp implements TempI
    {
        public void mul(int a, int b)
        {
            System.out.print("Answer by Abstract method = ");
            System.out.println(a * b);
        }
        public static void main(String[] args)
        {
            TempI in = new Temp();
            in.mul(2, 3);
            in.add(6, 2);
            TempI.mod(5, 3);
         }
   }
```

```
OUTPUT : Answer by Abstract method = 6            // mul(2, 3) = 2*3 = 6
        Answer by Private method = 4              // sub(6, 2) = 6-2 = 4
        Answer by Private static method = 3       // div(6, 2) = 6/2 = 3
        Answer by Default method = 8              // add(6, 2) = 6+2 = 8
        Answer by Private static method = 1       // div(5, 3) = 5/3 = 1
        Answer by Static method = 2               // mod(5, 3) = 5%3 = 2
```

Example for abstract method / default method / static method / private method / private static method

```
public interface CustomInterface
{
   public abstract void method1();
   public default void method2()
  {
      method4(); //private method inside default method
      method5(); //static method inside other non-static method
      System.out.println("default method");
   }
   public static void method3()
   {
      method5(); //static method inside other static method
      System.out.println("static method");
   }
   private void method4()
  {
        System.out.println("private method");
    }
    private static void method5(){
        System.out.println("private static method");
    }
}
public class CustomClass implements CustomInterface
{
    @Override
    public void method1()
   {
        System.out.println("abstract method");
    }
    public static void main(String[] args)
   {
        CustomInterface instance = new CustomClass();
        instance.method1();
        instance.method2();
        CustomInterface.method3();
    }
}
```

Output:

```
abstract method
private method
private static method
default method
private static method
static method
```

Develop a JAVA program to create an interface Resizable with methods resizeWidth(int width) and resizeHeight(int height) that allow an object to be resized. Create a class Rectangle that implements the Resizable interface and implements the resize methods.

```
interface Resizable
{
    void resizeWidth(int width);
    void resizeHeight(int height);
}

class Rectangle implements Resizable
{
    private int width;
    private int height;

    public Rectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    public void resizeWidth(int width)
    {
        this.width = width;
    }

    public void resizeHeight(int height)
    {
        this.height = height;
    }

    public int getWidth()
    {
        return width;
    }

    public int getHeight()
    {
        return height;
    }
}

public class ResizableDemo
{
    public static void main(String[] args)
    {
        Rectangle rectangle = new Rectangle(10, 20);

        System.out.println("Original Rectangle: Width=" +
rectangle.getWidth() + ", Height=" + rectangle.getHeight());

        rectangle.resizeWidth(15);
        rectangle.resizeHeight(30);

        System.out.println("Resized Rectangle: Width=" +
rectangle.getWidth() + ", Height=" + rectangle.getHeight());
    }
}
```

Output:

```
Original Rectangle: Width=10, Height=20
Resized Rectangle: Width=15, Height=30
```