Abstract class
and
Abstract method

Abstract class

- An ***abstract class*** is a class that is declared abstract.
- A *Java abstract class* is a class which cannot be instantiated, meaning you cannot create new instances of an abstract class. The purpose of an abstract class is to function as a base for subclasses.

Declaring an Abstract Class in Java

In Java you declare that a class is abstract by adding the `abstract` keyword to the class declaration. Here is a Java abstract class example:

```
public abstract class MyAbstractClass {

}
```

That is all there is to declaring an abstract class in Java. Now you cannot create instances of `MyAbstractClass`. Thus, the following Java code is no longer valid:

```
MyAbstractClass myClassInstance =
    new MyAbstractClass();  //not valid
```

If you try to compile the code above the Java compiler will generate an error, saying that you cannot instantiate `MyAbstractClass` because it is an abstract class.

Abstract method

- An ***abstract method*** is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- An abstract method has no implementation. It just has a method signature.

An abstract class can have abstract methods. You declare a method abstract by adding the `abstract` keyword in front of the method declaration. Here is a Java abstract method example:

```
public abstract class MyAbstractClass {

    public abstract void abstractMethod();
}
```

If a class has an abstract method, the whole class must be declared abstract. Not all methods in an abstract class have to be abstract methods. An abstract class can have a mixture of abstract and non-abstract methods.

Subclasses of an abstract class must implement (override) all abstract methods of its abstract superclass. The non-abstract methods of the superclass are just inherited as they are. They can also be overridden, if needed.

- If a class includes abstract methods, the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject
        {       // declare fields
                // declare non-abstract methods
                abstract void draw();
        }
```

- When an abstract class is sub-classed, the sub-class usually provides implementations for all of the abstract methods in its parent class.

- However, if it does not, the sub-class must also be declared abstract.
- Abstract class may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be sub-classed.

Example 1
(abstract class)

```
abstract class Person
{
    String name;
    int age;
}

class Student extends Person
{
    int regNo;
    String deptt;
    void show()
    {
        System.out.println("Name is: " + name);
        System.out.println("Age is: " + age);
        System.out.println("RegNo is: " + regNo);
        System.out.println("Department is: " + deptt);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Student s = new Student();
        s.name = "Max";      s.age = 29;
        s.regNo = 4545;      s.deptt = "CSE";
        s.show();
    }
}
```

Example 2
(abstract method)

```
abstract class Shape
{
    abstract double area();
    abstract double perimeter();
}

class Rectangle extends Shape
{
    double length, width;

    Rectangle(double length, double width)
    {
        this.length = length;     this.width = width;
    }

    double area()        //implementing abstract method
    {   return length * width;  }

    double perimeter()  //implementing abstract method
    {   return 2 * (length + width);    }
}

public class Test
{
    public static void main(String[] args)
    {
        Rectangle obj = new Rectangle(10, 20);
        System.out.println("Area: " + obj.area());
        System.out.println("Perimeter: " + obj.perimeter());
    }
}
```

Exercise 1

Write a program in which we define a class named **Calculate** with one abstract method:

int add(int a, int b)

Now create another class name **Welcome** containing main() method where you have to execute this method.

Note: Take values from user using command-line arguments.

Exercise 2

- Write a program to create class **Shape** having abstract method: **double area()**
- Make another class **Square** having private attribute: side and this class inherits the Shape class.
- Now make a class **Test** having main() method in which you have to create an object of Square class and print its area.

Exercise 3

- Write a program to create class **Person** having private attributes: name & age and abstract method: **void display()**
- Make another class **Employee** having private attributes: id, companyName & salary.
- Now make a class **Test** having main() method in which you have to create at least 3 Employee objects and store them in arrayList. Also this class should have another method: **void search(int id)** which will be used to search and display record of an employee based on the id provided by user at run-time.

Interfaces

interface

- An **interface** in java is a blueprint of a class.
- It has static constants and abstract methods.
- It cannot be instantiated just like abstract class.
- It is used to achieve abstraction and multiple inheritance in Java.
- Since Java 8, interface can have default and static methods.

Why use Java interface?

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

Properties of **interfaces**

- The **interface** keyword is used to declare an interface.

- Interfaces have the following properties:

  - An interface is implicitly abstract. We do not need to use the **abstract** keyword when declaring an interface.

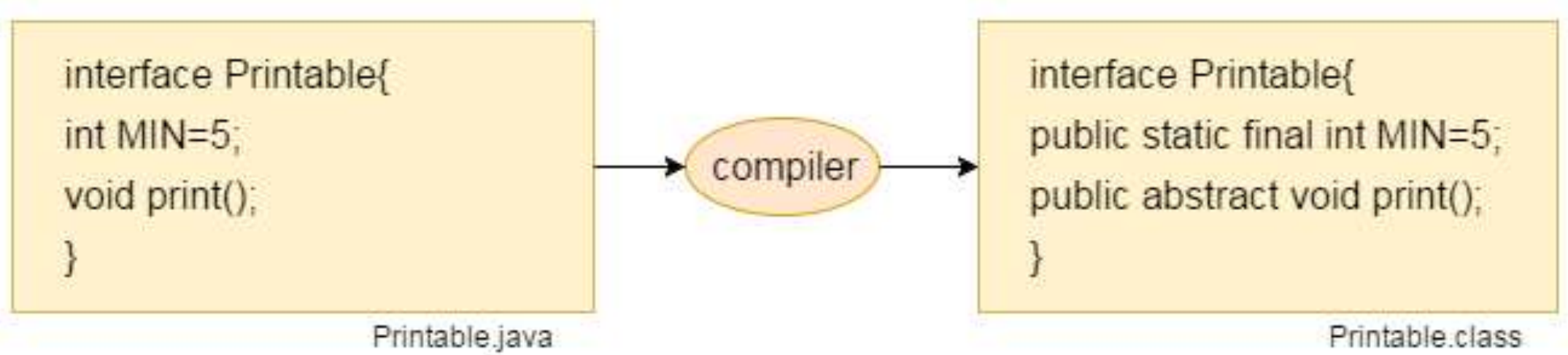
  - Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.

  - **Methods in an interface are implicitly public**.

  - **Variable in an interface are implicitly public, static & final** (and have to be assigned values there).

interface

- **interface** fields are public, static and final by default, and methods are public and abstract.



Implementing interfaces

- When a class implements an interface, then it has to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the **implements** keyword to implement an interface.
- The implements keyword appears in the class declaration following the extends portion of the declaration.

Example 1 (implementing interface)

```
interface Shape
{
    void color();
}

class Circle implements Shape
{
    // void color(){ }  //error, since can't reduce visibility
    public void color()
    {
        System.out.println("Red color");
    }
}

public class Test
{
    public static void main(String args[])
    {
        Circle c = new Circle();
        c.color();
    }
}
```

Example 2 (members accessibility)

```
interface Shape
{
    int r = 10;
    void area();
}

class Circle implements Shape
{
    int r = 20;     //hides the interface variable
    public void area()
    {
        System.out.println(3.14 * r * r);               //accesses instance variable "r"
        System.out.println(3.14 * Shape.r * Shape.r);  //accesses interface variable "r"
    }
}

public class Test
{
    public static void main(String args[])
    {
        Circle c = new Circle();
        c.area();
    }
}
```

Example 3 (Bank)

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Exercise

- Wap to create an interface named **Shape** containing method:
- **double getArea()**

- Now create a class named **Rectangle** which inherits **Shape** and contains instance variables: length, width which are initialized using parameterized constructor.

- Also create another class named **Circle** which also inherits **Shape** and contains instance variable: radius which is initialized using parameterized constructor.

- Now create another class **Test** containing main() method where you have to create an object of both **Rectangle** and **Circle** (and initialize their values using parameterized constructor).
- This class also contains another method named **calculate()** which will be used to print the area of any object passed to it.
- Call this calculate() method inside main() twice such that during the first call you have to pass **Circle** object and during the second call you have to pass **Rectangle** object.

Solution

```
interface Shape
{
    public double getArea();
}

class Rectangle implements Shape
{
    double length, width;
    Rectangle(double length, double width)
    {
        this.length = length;     this.width = width;
    }

    public double getArea()       //implementing function of interface
    {   return (length * width);   }
}

class Circle implements Shape
{
    double radius;
    Circle(double radius)
    {
        this.radius = radius;
    }

    public double getArea()     //implementing function of interface
    {   return (3.14 * radius * radius);    }
}
```

```java
public class Test
{
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(3.1, 4.8);
        calculate(r);      //passing Rectangle object to compute area

        Circle c = new Circle(10.2);
        calculate(c);      //passing Circle object to compute area
    }

    static void calculate(Shape obj)  //function carrying interface reference variable
    {
        System.out.println("Area: " + obj.getArea());
    }
}
```

Exercise

- Write a program to implement interface **Colors** having methods:

  void setColor(String color)

  String getColor()

- Create a class **Square** which implements the Colors interface and having private attributes: side & color and methods:

  void setSide(double side)

  double getSide()

  double getArea()

- Create a class **Test** having main() method in which you have to create an object of Square class and display its area & details after taking input from user at runtime.

interface vs. **class**

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte-code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding byte-code file must be in a directory structure that matches the package name.

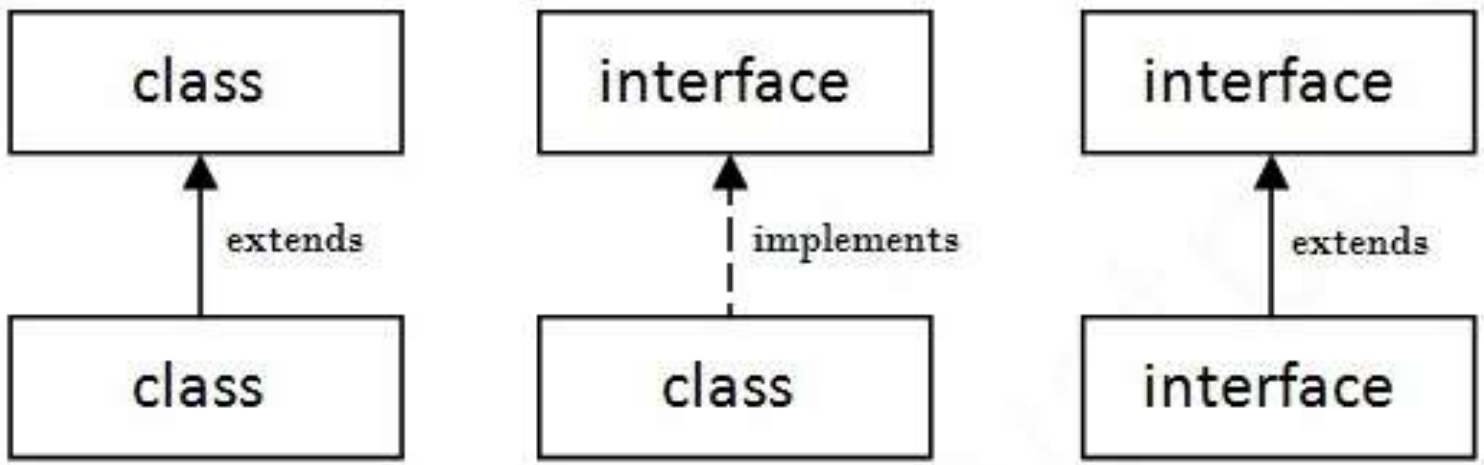
interface vs. **class**

An interface is different from a class in several ways, including:

- We cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

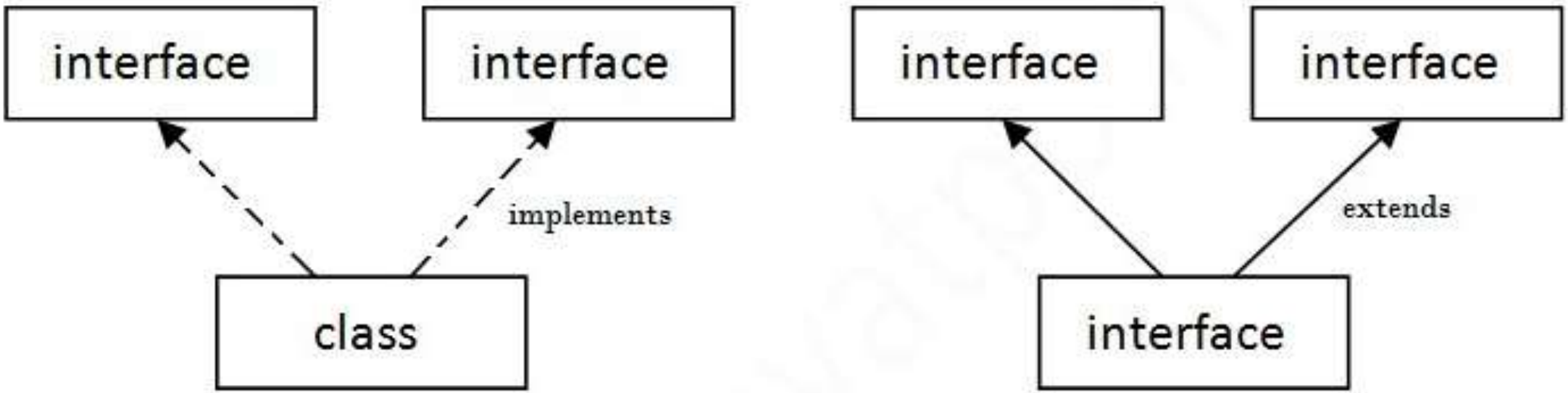
Understanding relationship between classes and interfaces

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



Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

Example 1 (multiple inheritance using interfaces)

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Example 2 (inheritance in interfaces)

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Example 3 (multiple inheritance using class & interface)

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

interface A
{
    void show();
}
interface B
{
    void display();
}

class Demo extends Shape implements A, B
{
    void fun(){
        System.out.println("class-Shape method");
    }
    public void show(){
        System.out.println("Interface-A method");
    }
    public void display() {
        System.out.println("Interface-B method");
    }
}
```

```
public class Test
{
    public static void main(String a[])
    {
        Demo d = new Demo();
        d.fun();
        d.show();
        d.display();
    }
}
```

Example 4 (anonymous class using interface)

```
interface Shape
{
    void showColor();
}

public class Test
{
    public static void main(String args[])
    {
        Shape s = new Shape(){
                        public void showColor()
                        {   System.out.println("Red");  }
                    };

        s.showColor();
    }
}
```

Difference between abstract class and interface

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. But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
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) **Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Default method in interface

Default methods

- Default methods are defined inside the interface and tagged with **default** are known as default methods.
- These methods are non-abstract methods.
- Since Java 8, we can have method body in interface. But we need to make it **default**.

Before Java 8 Java interfaces could not contain an implementation of the methods, but only contain the method signatures. However, this results in some problems when an API needs to add a method to one of its interfaces. If the API just adds the method to the desired interface, all classes that implements the interface must implement that new method. That is fine if all implementing classes are located within the API. But if some implementing classes are located in client code of the API (the code that uses the API), then that code breaks.

Example 1

```
interface Shape
{
    // String color{ }  //error, since method not declared default
    default void color(){
        System.out.println("Red");
    }
    double area();
}

class Circle implements Shape
{
    double radius;
    public double area(){
        return 3.14*radius*radius;
    }
}

public class Test
{
    public static void main(String args[]){
        Circle c = new Circle();
        System.out.println(c.area());
        c.color();
    }
}
```

Here **default** is not an access-specifier, but just a keyword. Hence, the method is by default **public** and can also be defined as follows:

default public void color() {

}

Example 2 (multiple inheritance using interfaces with default methods)

```
interface Shape1
{
    default void area()
    {
        System.out.println("Shape-1 method");
    }
}

interface Shape2
{
    default void area()
    {
        System.out.println("Shape-2 method");
    }
}

class Circle implements Shape1, Shape2
{
    //error, cannot use duplicate default methods from multiple interfaces
}
```

In order to remove ambiguity, override the default method (of the interface) in the class as:

```
public void area() {
        System.out.println("overridden method");
        // if you want to access parent method here, then use:    Shape1.super.area();
}
```

Key point - 1

```
interface Circle {
    default public void area() {
        System.out.println("circle-area");
    }
}

class Square {
    public void area() {
        System.out.println("square-area");
    }
}

class Calculate extends Square implements Circle {

}

public class Temp {
    public static void main(String args[])
    {
        Calculate c = new Calculate();
        c.area(); //accesses area-method of class
    }
}
```

When the super-types of a class or interface provide multiple default methods with the same signature,
Instance methods are preferred over interface default methods.

Exercise

Create a interface Colors having default method: **void showColor()**

Create another interface Shape having abstract method: **void area()**

Create a class Show having abstract method: **void display()**

Create a class Circle which inherits Colors, Shape and Show and have instance variable: radius.

Make a class Test having main() method in which you have to create object of Circle class and invoke all its methods.

static method in interface

static method

- Since Java 8, you can define static methods in interfaces.

- A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.

- **A class inherits from its direct super-class all concrete methods (both static and instance).**
- **A class inherits from its direct super-class and direct super-interfaces all abstract and default methods.**
- **A class does not inherit static methods from its super-interfaces.**

- A static method of interface is part of interface and hence unlike static method of class, it is never inherited. Therefore, we can't access static methods of interface using its reference variable and has to be accessed by using its name.

- Java interface static method helps us in providing security by not allowing implementation classes to override them.

Example 1 (Parent-class static method accessibility)

```java
class Parent {
    static void show() {
        System.out.println("parent");
    }
}

class Child extends Parent {

}

public class Temp {
    public static void main(String args[])
    {
        Child c = null;
        c.show();   //accesses parent-class method
    }
}
```

```java
class Parent {
    static void show() {
        System.out.println("parent");
    }
}

class Child extends Parent {
    static void show() {    //hides parent-static method
        System.out.println("child");
    }
}

public class Temp {
    public static void main(String args[])
    {
        Child c = null;
        c.show();   //accesses child-class method
    }
}
```

Hiding: Parent class methods that are static are not part of a child class (although they are accessible), so there is no question of overriding it. Even if you add another static method in a subclass, identical to the one in its parent class, this subclass static method is unique and distinct from the static method in its parent class.

Example 2 (Parent-interface static method accessibility)

```java
interface Parent {
    static void show() {
        System.out.println("parent");
    }
}

class Child implements Parent {

}

public class Temp {
    public static void main(String args[])
    {
        Child c = null;
        c.show();  //error, since static-method not inherited
        Parent.show();
    }
}
```

static method of interface is not inherited in child class

Example 3 (Parent-interface static method accessibility)

```
interface Shape
{
    static void color(){
        System.out.println("Red");
    }
    double area();
}

class Circle implements Shape
{
    double radius;
    public double area(){
        return 3.14*radius*radius;
    }
}

public class Test
{
    public static void main(String args[]){
        Circle c = new Circle();
        System.out.println(c.area());
        //c.color();  //error, since static method is exclusive to Shape
        Shape.color();
    }
}
```

Example 4 (multiple inheritance using interfaces with static methods)

```
interface Shape1
{
    static void area()
    {
        System.out.println("Shape-1 method");
    }
}

interface Shape2
{
    static void area()
    {
        System.out.println("Shape-2 method");
    }
}
```

```
class Circle implements Shape1, Shape2
{
    void fun()
    {
        //area();      //error, method not defined for Circle class
        Shape1.area();
        Shape2.area();
    }
}

public class Test
{
    public static void main(String args[])
    {
        Circle c = new Circle();
        c.fun();
    }
}
```

Example 5 (variable accessibility)

```
interface A {
    int a = 10;
}

class B {
    static int a = 20;
}

class C extends B implements A
{
    int a = 30;
}

class Temp {
      public static void main(String args[])
      {
          C obj = new C();
          System.out.println(obj.a);    //accesses variable of C
          System.out.println("Variable of A: " + A.a);
          System.out.println("Variable of B: " + B.a);
      }
}
```

Exercise

Write a program in which we define:

i. A class named **A** with one abstract method int add(int a, int b) and a variable a with value 10

ii. An interface X with at least one abstract method void test(), one default method void demo() and one static method void show() and a data member a with value 20

iii. An interface Y with one abstract method void test() and one default method void demo() and a data member a with value 30.

Create a class Z which inherits the class A and interfaces X and Y. Now invoke all the 5 methods of the class and interface and also display the values of a.

Solution

```
abstract class A {
    int a = 10;
    abstract int add(int a, int b);
}

interface X {
    int a = 20;
    void test();
    default void demo() {
        System.out.println("X default-method");
    }
    static void show() {
        System.out.println("X static-method");
    }
}

interface Y {
    int a = 30;
    void test();
    default void demo() {
        System.out.println("Y default-method");
    }
}
```

```
class Z extends A implements X, Y {
    int add(int a, int b) {
        return a + b;
    }
    public void test() {
        System.out.println("test method implemented");
    }
    public void demo() {
        X.super.demo();
        Y.super.demo();
        System.out.println("demo default-method overridden");
    }
}
```

```java
class Temp
{
	public static void main(String args[])
	{
		A obj = new Z();  //Parent reference holding child-object
		System.out.println("Variable of A: " + obj.a);
		System.out.println("Add method: " + obj.add(5, 6));

		System.out.println("Variable of X: " + X.a);
		X.show();

		System.out.println("Variable of Y: " + Y.a);
		((Z)obj).test();  //downcast to access new members of child-class
		((Z)obj).demo();  //downcast to access new members of child-class
	}
}
```