Interface in Java

An **interface in Java** is syntactically similar to a class but can have only abstract methods declaration and constants as members.

In other words, an interface is a collection of abstract methods and constants (i.e. static and final fields). It is used to achieve complete abstraction.

Every interface in java is abstract by default. So, it is not compulsory to write abstract keyword with an interface.

Once an interface is defined, we can create any number of separate classes and can provide their own implementation for all the abstract methods defined by an interface.

A class that implements an interface is called **implementation class**. A class can implement any number of interfaces in Java.

Every implementation class can have its own implementation for abstract methods specified in the interface

Since the implementation classes will have all the methods with a body, it is possible to create an instance of implementation classes

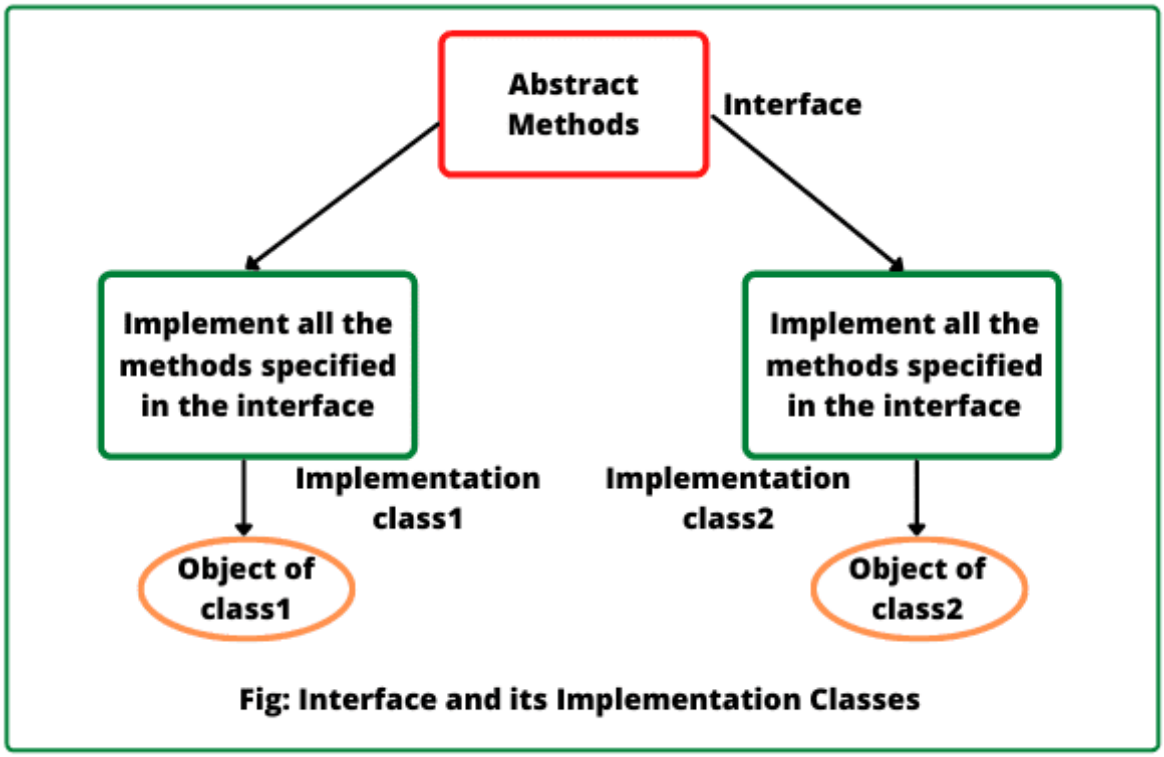


Fig: Interface and its Implementation Classes

Why do we use Interface?

There are mainly five reasons or purposes of using an interface in Java. They are as follows:

1.In industry, architect-level people create interfaces, and then it is given to developers for writing classes by implementing interfaces provided.

2. Using interfaces is the best way to expose our project's API to some other projects. In other words, we can provide interface methods to the third-party vendors for their implementation.

3. Programmers use interface to customize features of software differently for different objects.

4. It is used to achieve full abstraction in java.

5. By using interfaces, we can achieve the functionality of multiple **inheritance**.

How to Declare Interface in Java?

```
public abstract interface MyInterfac
{
  int x = 10; // public static final keyword invisibly present.
  void m1(); // public and abstract keywords invisibly present.
  void m2();// public and abstract keywords invisibly present.
}
```

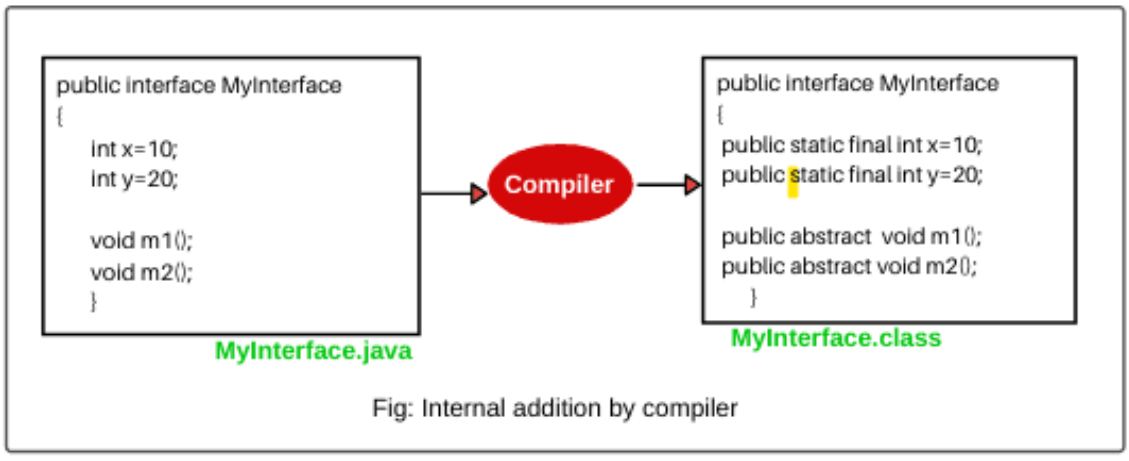


Fig: Internal addition by compiler

Features of Interface

There are the following features of an interface in Java. They are as follows:

1. Interface provides pure **abstraction in java**. It also represents the **Is-A** relationship.

2. It can contain three types of methods: abstract, default, and static **methods**.

3.All the (non-default) methods declared in the interface are by default abstract and public. So, there is no need to write abstract or public modifiers before them.

4. The fields (data members) declared in an interface are by default public, static, and final. Therefore, they are just public constants. So, we cannot change their value by implementing
class once they are initialized.

5. Interface cannot have constructors.

6. The interface is the only mechanism that allows achieving multiple inheritance in java.

7. A Java class can implement any number of interfaces by using keyword implements.

8. Interface can extend an interface and can also extend multiple interfaces.

Rules of Interface in Java

Here are some key points for defining an interface in java that must be kept in mind. The rules are as follows:

1.An interface cannot be instantiated directly. But we can create a reference to an interface that can point to an object of any of its derived types implementing it.

2. An interface may not be declared with final keyword.

3.It cannot have instance variables. If we declare a variable in an interface, it must be initialized at the time of declaration.

4. A class that implements an interface, must provide its own implementations of all the methods defined in the interface.

5.We cannot reduce the visibility of an interface method while overriding. That is, when we implement an interface method, it must be declared as public.

6. It can also be declared with an empty body (i.e. without any members). For example, java.util package defines EventListener interface without a body.

7. An interface can be declared within another interface or class. Such interfaces are called nested interfaces in java.

8.A top-level interface can be public or default with the abstract modifier in its definition. Therefore, an interface declared with private, protected, or final will generate a compile-time error.

9.All non-default methods defined in an interface are abstract and public by default. Therefore, a method defined with private, protected, or final in an interface will generate compile-time error.

10. If you add any new method in interface, all concrete classes which implement that interface must provide implementations for newly added method because all methods in interface are by default abstract.

Extending Interface in Java with Example

Like classes, an interface can also extend another interface. This means that an interface can be sub interfaces from other interfaces.

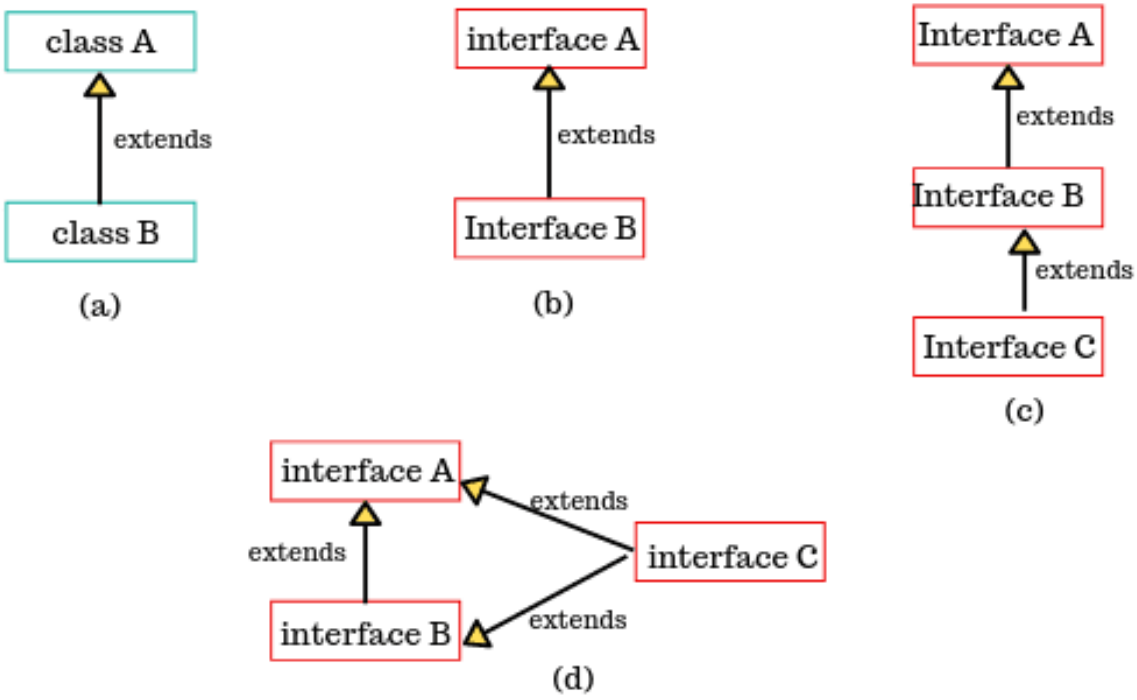


Fig: Various forms of extending Interface in Java

1. We can define all the constants in one interface and methods in another interface. We can use constants in classes where methods are not required.

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

2. We can also extend various interfaces together by a single interface.

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

Key points:

1. An interface cannot extend classes because it would violate rules that an interface can have only abstract methods and constants.
2. An interface can extend Interface1, Interface2.

Implementing Interface in Java

An interface is used as “superclass” whose properties are inherited by a class. A class can implement one or more than one interface by using a keyword implements followed by a list of interfaces separated by commas.

When a class implements an interface, it must provide an implementation of all methods declared in the interface and all its super interfaces.

Otherwise, the class must be declared abstract.

1. All methods of interfaces when implementing in a class must be declared as public otherwise, you will get a compile-time error if any other modifier is specified.

2. Class extends class implements interface.

3. Class extends class implements Interface1, Interface2…

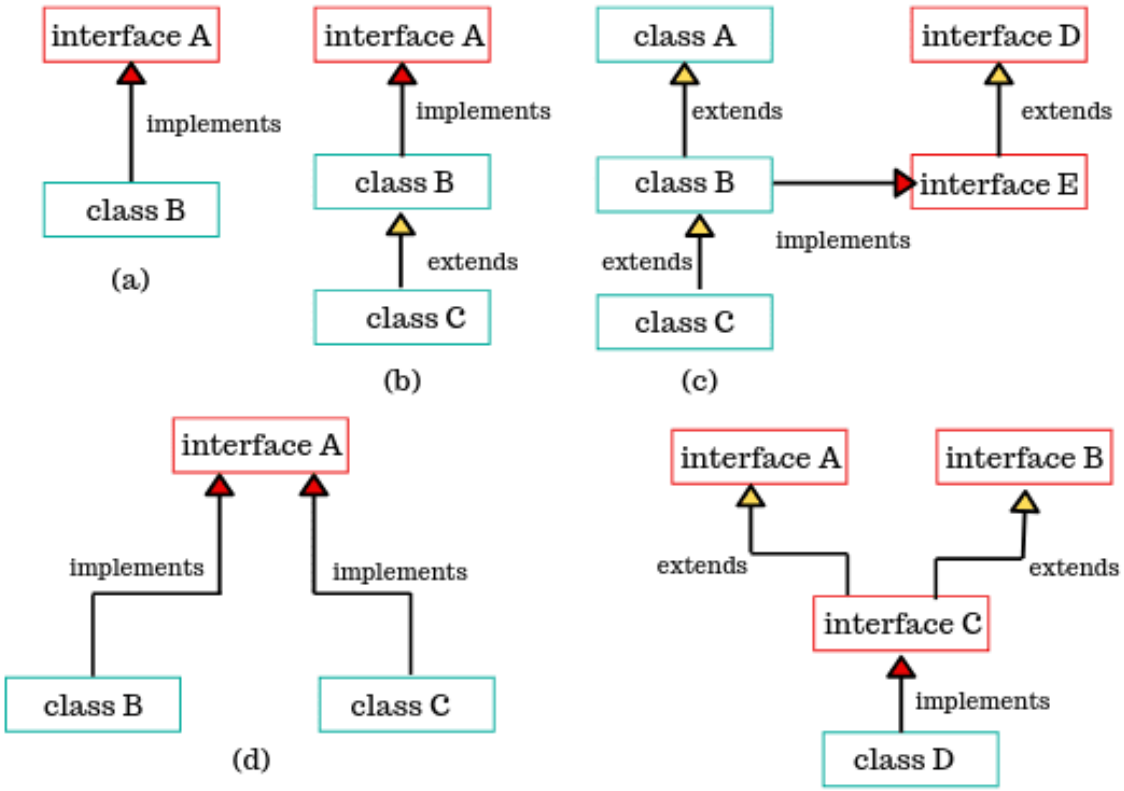


Fig: Various forms of interface implementation

Accessing Interface Variable in Java

The interface is also used to declare a set of constants that can be used in multiple classes. The constant values will be available to any classes that implement interface because it is by default public, static, and final.

We can use values in any method as part of the variable declaration or anywhere in the class.

multiple classes implement the same interface to use constant values

```
package interfacePrograms;
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 Main
{
  public static void main(String[] args)
  {
    Add a = new Add();
     a.m1();
     a.sum();
    Sub s = new Sub();
     s.sub();
   }
}
```

class B implements an interface A

```
package interfacePrograms;
public interface A
{
  void msg(); // No body.
}
public class B implements A
{
 // Override method declared in interface.
    public void msg()
    {
      System.out.println("Hello Java");
    }
    void show()
   {
      System.out.println("Welcome you");
   }
public static void main(String[] args)
{
```

```
   B b = new B();
    b.msg();
    b.show(); // A reference of interface is pointing to objects of class B.

   A a = new B();
    a.msg();
 // a.show(); // Compile-time error because a reference of interface can only
 //call methods declared in it and implemented by implementing class.

// show() method is not part of interface. It is part of class B.
// When you will call this method, the compiler will give a compile-time error.
//It can only be called when you create an object reference of class B.
   }
 }
```

Polymorphism in Java Interface

When two or more classes implement the same interface with different implementations, then through the object of each class, we can achieve polymorphic behavior for a given interface. This is called polymorphism in interface.

```
package interfacePrograms;
public interface Rectangle
{
  void calc(int l, int b);      // no body.
}
public class RecArea implements Rectangle {
public void calc(int l, int b)
{
  int area = l * b;             // Implementation.
  System.out.println("Area of rectangle = "+area);
 }
}
public class RecPer implements Rectangle {
public void calc(int l, int b)
{
  int perimeter = 2 * (l + b);   // Implementation.
  System.out.println("Perimeter of rectangle = "+perimeter);
 }
}
public class Execution {
public static void main(String[] args)
{
 Rectangle rc;           // Creating an interface reference.
  rc = new RecArea();    // Creating object of RecArea.
   rc.calc(20, 30);      // calling method.
  rc = new RecPer();     // Creating an object of RecPer.
   rc.calc(20, 30);
 }
}
```

Multilevel Inheritance by Interface

```
package interfacePrograms;
public interface Continent
{
  void showContinent();
 }
public interface Country extends Continent
{
  void showCountry();
 }
public interface State extends Country
{
  void showState();
}
public class City implements State
{
 public void showContinent()
  {
    System.out.println("Asia");
  }
 public void showCountry()
 {
   System.out.println("India");
 }
 public void showState()
 {
   System.out.println("Jharkhand");
 }
void showCity()
{
   System.out.println("Dhanbad");
}
public static void main(String[] args)
{
  City c = new City();
   c.showContinent();
   c.showCountry();
   c.showState();
   c.showCity();
  }
}
```

Multiple Inheritance in Java by Interface

When a class implements more than one interface, or an interface extends more than one interface, it is called **multiple inheritance**.

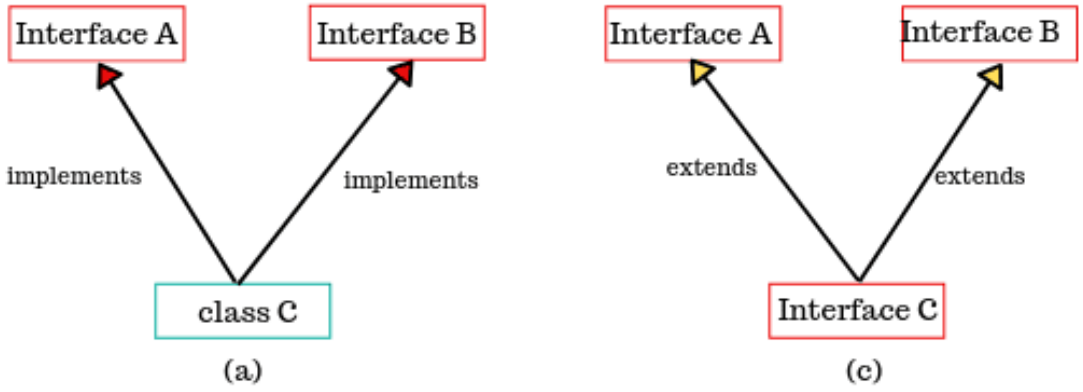


Fig: Various forms of Multiple inheritance by Interface in Java

```
package multipleInheritancebyInterfaces;
public interface Home
{
  void homeLoan();
 }
public interface Car
{
  void carLoan();
}
public interface Education
{
  void educationLoan();
 }
public class Loan implements Home, Car, Education
{
// Multiple inheritance using multiple interfaces.
  public void homeLoan()
  {
     System.out.println("Rate of interest on home loan is 8.5%");
  }
 public void carLoan()
 {
   System.out.println("Rate of interest on car loan is 9.25%");
 }
 public void educationLoan()
 {
   System.out.println("Rate of interest on education loan is 10.45%");
 }
public static void main(String[] args)
{
   Loan l = new Loan();
    l.homeLoan();
    l.carLoan();
    l.educationLoan();
  }
}
```

In Java, Multiple Inheritance is not supported through Class, but it is possible by Interface. Why?

If two superclasses have the same method name, then which method is inherited into subclass is the main confusion in multiple inheritance.

That's why Java does not support multiple inheritance in the case of class. But, it is supported through an interface because there is no confusion. This is because its implementation is provided by the implementation class.

```
package multipleInheritancebyInterface;
public interface AA
{
  void m1();
 }
public interface BB
{
  void m1();
}
public class Myclass implements AA, BB
{
  public void m1()
  {
     System.out.println("Hello Java");
  }
public static void main(String[] args)
{
   Myclass mc = new Myclass();
    mc.m1();
  }
}
```

Can we have an Interface without any Methods or Fields?

An interface without any fields or methods is called **marker interface** in java. There are several built-in Java interfaces that have no method or field definitions.

For example, Serializable, Cloneable, and Remote all are marker interfaces.They act as a form of communication among class objects.

Why interface methods are public and abstract by default?

Interface methods are public and abstract because their implementation is left for third-party vendors.

Abstract Class vs Interface in Java

	abstract class	interface
Keyword(s) used:	Two keywords abstract and class are used to define an abstract class.	Only one keyword interface is used to define an interface.
Keyword used by implementing class:	To inherit the abstract class, we use the extends keyword.	To implement an interface, we can use the implements keyword.
Variables:	Abstract class can have final, non-final, static, and non-static variables.	Interface cannot have any instance variables. It can have only static variables.
Initialization:	The abstract class variable does not require performing initialization at the time of declaration.	Interface variable must be initialized at the time of declaration otherwise we will get compile-time error.
Method:	An abstract class can have both abstract and non-abstract (concrete) methods.	Every method present inside an interface is always public and abstract, whether we are declaring or not. That's why interface is also known as a pure (100%) abstract class.
Constructors:	Inside an interface, we cannot declare/define a constructor because the purpose of constructor is to perform initialization of instance variable but inside interface every variable is always static. Therefore, inside the interface, the constructor concept is not applicable and does not require.	Since an abstract class can have instance variables. Therefore, we can define constructors within the abstract class to initialize instance variables.
Static and Instance blocks:	We can declare instance and static blocks inside abstract class.	We cannot declare instance and static blocks inside an interface. If you declare them, you will get compile time error.
Access modifiers:	There is no restriction in declaring private or protected members inside an abstract class.	We cannot define any private or protected members in an interface. All members are public by default.

	abstract class	interface
Single vs Multiple inheritance:	A class can extend only one class (which can be either abstract or concrete class).	A class can implement any number of interfaces.
Default Implementation:	An abstract class can provide a default implementation of a method. So, subclasses of an abstract class can just use that definition but subclasses cannot define that method.	An interface can only declare a method. All classes implementing interface must define that method.
Difficulty in making changes:	It is easy to make changes to the implementation of the abstract class. For example, we can add a method with default implementation and the existing subclass cannot define it.	It is difficult to make changes in an interface if many classes already implementing that interface. For example, suppose you declare a new method in interface, all classes implementing that interface will stop compiling because they do not define that method.
Uses:	If you know nothing about the implementation. You have just requirement specification then you should go to use interface.	If you know about implementation but not completely (i.e. partial implementation) then you should go for using abstract class.

```
package com.abstractProgram;
// Two keywords used: Abstract & class.
public abstract class AbstractClass {

// Declaration of final, non-final, static, and instance variables.
   int a; // Not require initialization.
   final int b = 20; // Final variable.
   static int c = 30; // static variable.

// Declaration of abstract and non-abstract methods.
   abstract void m1();
   static void m2()
   {
      System.out.println("Static method in abstract class");
   }
```

```
// Default implementation of instance method.
   void m3() { // Concrete method.
     System.out.println("Instance method in abstract class");
    }
// Declaration of constructors to initialization of instance variable.
    AbstractClass()
    {
      int a = 10;
      System.out.println("Value of a; "+a);
    }
// Declaration of static blocks.
   static {
      System.out.println("Static block in abstract class");
   }
// Declaration of non-static blocks.
  {
    System.out.println("Instance block in abstract class");
  }
// Declaration of private & protected members.
   private void m4()
   {
      System.out.println("Private method");
    }
   protected void m5()
   {
     System.out.println("Protected method");
   }
}
public class A extends AbstractClass
{
  void m1()
  {
     System.out.println("Implementation of abstract method");
  }
}
public class AbstractTest
{
  public static void main(String[] args)
  {
    A a = new A();
    System.out.println("Value of b: " +a.b);
    System.out.println("Value of c: " +AbstractClass.c);
     a.m1();
    AbstractClass.m2();
     a.m3();
     a.m5();
   }
}
```

```
package interfaceProgram;

public interface AA { // One keyword: interface.

 int x = 20; // Interface variable must be initialized at the time of declaration. By d
efault, interface variable is public, static, and final.
```

```
 void m1(); // By default, interface method is public and static.

// Here, we cannot declare instance variables, instance methods, constructors, static,
 and non-static block.
  }


public interface BB
{
  int y = 20;
  void m2();
 }

public class CC implements AA, BB { // Multiple Inheritance.

 public void m1()
 {
   System.out.println("Value of x: " +x);
   System.out.println("m1 method");
  }
public void m2()
{
   System.out.println("Value of y: " +y);
   System.out.println("m2 method");
 }
}
public class MyClass
{
  public static void main(String [] args)
  {
    CC c = new CC();
     c.m1();
     c.m2();
  }
}
```

Marker Interface in Java

An interface that does not contain methods, fields, and constants is known as **marker interface**.

In other words, an empty interface is known as **marker interface** or **tag interface.**It delivers the run-time type information about an object. It is the reason that the JVM and compiler have additional information about an object.

This marker interface tells the compiler that the objects of the class that implement the marker interface are different and that they should be treated differently.

Each marker interface in Java indicates that it represents something special to JVM or compiler.

In Java, we have three interfaces that are Marker interfaces as shown below:

#1) Serializable interface: Serializable is a marker interface present in the java.io package. We can serialize objects using this interface i.e. save the object state into a file.

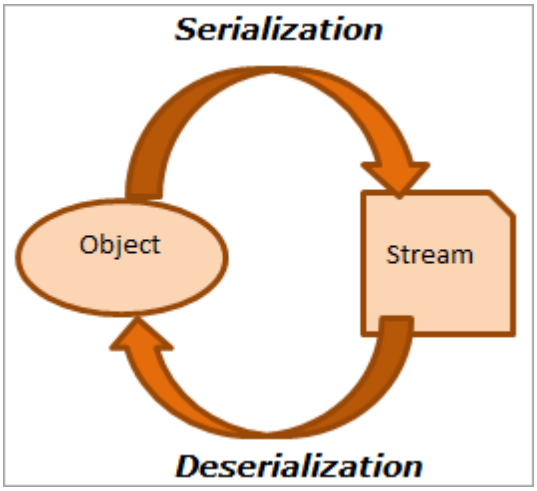
#2) Cloneable interface: The cloneable interface is a part of the java.lang package and allows the objects to be cloned.

#3) Remote interface: The remote interface is a part of the java.RMI package and we use this interface to create RMI applications. This interface mainly deals with remote objects.

Serialization In Java

Serialization can be defined as a process by which we convert the object state into its equivalent byte stream to store the object into the memory in a file or **persist** the object.

When we want to retrieve the object from its saved state and access its contents, we will have to convert the byte stream back to the actual Java object and this process is called deserialization.



We should fulfill the following condition for an object to be successfully serialized:

1. The class whose objects are serialized must implement java.io.Serializable interface.
2. All the member fields of the class must be serializable. If a particular field is not serializable then we should mark it as transient.

```
package interfaceArea;

import java.io.*;
import java.io.Serializable;

public class Student implements Serializable {
  int id;
  String name;

  public Student(int id, String name) {
    this.id = id;
    this.name = name;
  }

  public static void main(String[] args) {
    try {
      // Create the object of student class
      Student s1 = new Student(108, "Debasish");

      // Write the object to the stream by creating a output stream
      FileOutputStream Fout = new FileOutputStream("Debasish.txt");
      ObjectOutputStream Oout = new ObjectOutputStream(Fout);
      Oout.writeObject(s1);
      Oout.flush();
      // close the stream
      Oout.close();
      System.out.println("Object successfully written to the file");

      // Create a stream to read the object
      ObjectInputStream Oin = new ObjectInputStream(new FileInputStream("Debasish.txt"));
      Student s = (Student) Oin.readObject();
      // print the data of the deserialized object
      System.out.println("Student object: " + s.id + " " + s.name);
      // close the stream
      Oin.close();

    } catch (Exception e) {
      System.out.println(e);
    }
  }

}
```

Java Transient Keyword

A transient keyword is used to make a data member transient i.e. we do not want to serialize it.

```
package interfaceArea;

import java.io.*;
import java.io.Serializable;
```

```
public class Student implements Serializable {
  transient int id;
  String name;

  public Student(int id, String name) {
    this.id = id;
    this.name = name;
  }

  public static void main(String[] args) {
    try {
      // Create the object of student class
      Student s1 = new Student(108, "Debasish");

      // Write the object to the stream by creating a output stream
      FileOutputStream Fout = new FileOutputStream("Debasish.txt");
      ObjectOutputStream Oout = new ObjectOutputStream(Fout);
      Oout.writeObject(s1);
      Oout.flush();
      // close the stream
      Oout.close();
      System.out.println("Object successfully written to the file");

      // Create a stream to read the object
      ObjectInputStream Oin = new ObjectInputStream(new FileInputStream("Debasish.txt"));
      Student s = (Student) Oin.readObject();
      // print the data of the deserialized object
      System.out.println("Student object: " + s.id + " " + s.name);
      // close the stream
      Oin.close();

    } catch (Exception e) {
      System.out.println(e);
    }
  }

}
```

Java.io.NotSerializableException In Java

The exception of java.io.NotSerializableException is an exception that is thrown when the class is not eligible for serialization. The class that does not implement the Serializable interface becomes ineligible for serializatio

```
package interfaceArea;

import java.io.*;
public class Student{
  transient int id;
  String name;

  public Student(int id, String name) {
```

```
    this.id = id;
    this.name = name;
  }

  public static void main(String[] args) {
    try {
      // Create the object of student class
      Student s1 = new Student(108, "Debasish");

      // Write the object to the stream by creating a output stream
      FileOutputStream Fout = new FileOutputStream("Debasish.txt");
      ObjectOutputStream Oout = new ObjectOutputStream(Fout);
      Oout.writeObject(s1);
      Oout.flush();
      // close the stream
      Oout.close();
      System.out.println("Object successfully written to the file");

      // Create a stream to read the object
      ObjectInputStream Oin = new ObjectInputStream(new FileInputStream("Debasish.txt"));
      Student s = (Student) Oin.readObject();
      // print the data of the deserialized object
      System.out.println("Student object: " + s.id + " " + s.name);
      // close the stream
      Oin.close();

    } catch (Exception e) {
      System.out.println(e);
    }
  }

}
```

Cloneable Interface In Java

Cloning of objects means making a copy of the objects. Java supports object cloning using the “**Cloneable**” interface. The cloneable interface is a marker interface and is a part of the java.lang package.

When a class implements the Cloneable interface, then it implies that we can clone the objects of this class. The Object class of Java contains the ‘**clone**()’ method. So Cloneable interface
implemented by a particular class authorizes the clone () method to make copies of class instances.

If a class does not implement a Cloneable interface and still invokes the clone () method, then the exception **CloneNotSupportedException** is thrown by the Java compiler.

Classes implementing the Cloneable interface should override the clone () method.

So what is Object Cloning?

Object cloning is a process using which we create an exact copy of the object using the clone () method of the Object class. For the clone () method to be overridden and invoked, the class needs to implement the Cloneable interface.

```
package interfaceArea;

public class Student2 implements Cloneable {
  int rollno;
  String name;

  // class constructor
  Student2(int rollno, String name) {
    this.rollno = rollno;
    this.name = name;
  }

  // clone method
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }

  public static void main(String[] args) {
    try {
      Student2 s1 = new Student2(108, "Debasish");
      // clone the s1 object
      Student2 s2 = (Student2) s1.clone();

      System.out.println("Original Student object: " + s1.rollno + " " + s1.name);
      System.out.println("Cloned Student object: " + s2.rollno + " " + s2.name);

    } catch (CloneNotSupportedException c) {
      System.out.println(c);
    }

  }

}
```