Interface

An interface is not a class but it is a blueprint of a class. its definition is similar to a class definition except that it uses the interface keyword. All methods in an interface are *abstract methods or default method (java 8)*, that is, they are declared without the implementation part since they are to be implemented in the subclasses that use them. It can also include a static constant declaration. Writing an interface is like writing a class, but they are two different concepts:

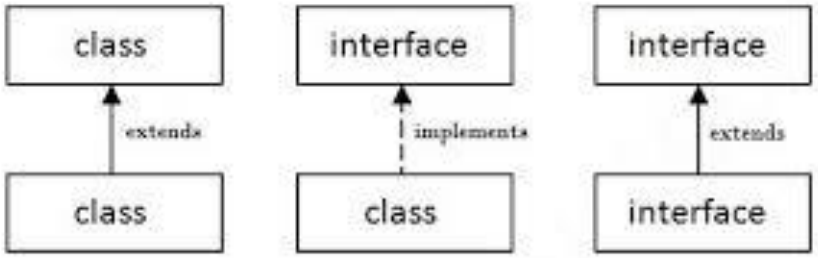
A class describes the attributes and behaviors of an object while, an interface contains behaviors that a class implements. **Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.**

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 bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

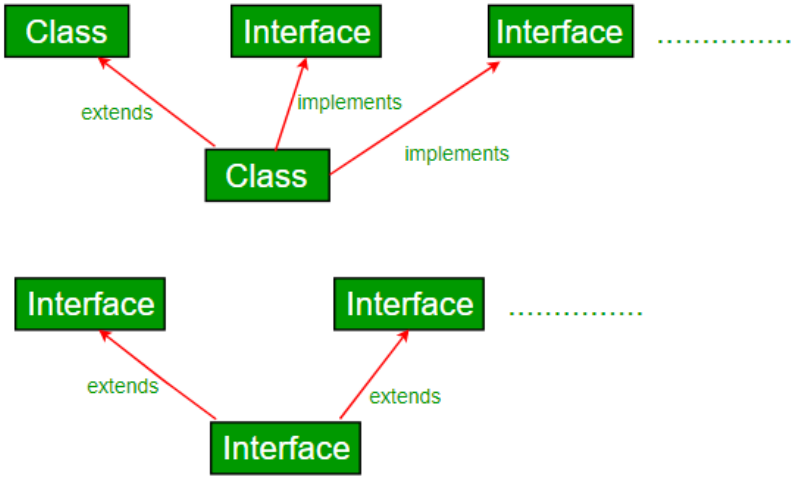
However, an interface is **different** from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.



When implementation interfaces there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface itself can extend another interface



Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface. Let us look at an example that depicts encapsulation:

```
public interface NameOfInterface {
  //Any number of final, static fields
  //Any number of abstract method declarations\
  }
```

Interfaces have the following properties:

- An interface is implicitly (ضمنيا) abstract. You 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.
- No static methods within Interface.

Example 1

```
public interface NewInterface {
int x=10;◄———— implicitly static and final (constant)
void print();◄———— implicitly public
public NewInterface();———— E R R O R no constructor within interface
 }
```

Example 2

```
/* File name : Animal.java */
interface Animal {

        public void eat();
        public void travel();
}
```

Attributes in an Interface

Data attributes declared in an interface construct are always static and final. They are implicitly declared as static and final in an interface definition, these keywords need not precede their declaration.

Methods in an Interface

All methods in an interface are abstract methods and any class that uses the interface must provide an implementation for them. It does not have to explicitly declare its methods abstract using the keyword abstract. Similarly, interface methods are always public, and the access modifier public keyword is not required since it is implied in the interface declaration. However, in contrast with data attributes in an interface, **methods may not be static since static methods**, being class specific, are never abstract.

Implementing Interfaces

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. When a class implements an interface, you can think of the class as signing a contract, agreeing 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**.

Example 3

```
/* File name : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){      System.out.println("Mammal eats");   }
   public void travel(){      System.out.println("Mammal travels");   }
   public int noOfLegs(){      return 0;   } }

public class Main {
   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();      m.travel();   }}
```

This would produce following result:

```
Mammal eats
Mammal travels
```

Example 4 (Trace)

Below is an example of a Shape interface

```
interface Shape {
        public double area();
        public double volume();}
```

Below is a Point class that implements the Shape interface.

```
public class Point implements Shape {

        static int x, y;
        public Point() {// we can define rect or circle…etc
                x = 0;
                y = 0;    }
        public double area() {              return 0;          }
        public double volume() {                    return 0;          }
        public static void print() { System.out.println("point: " + x + "," + y);}
        public static void main(String args[]) {
                Point p = new Point();
                p.print();          } }
```

Abstract Class and Interface

A class implementing an interface must implement all the abstract methods declared in an interface; otherwise, the class is considered as an *abstract* class and must be declared using the abstract keyword as follows:

```
abstract class ColourTest implements Colourable {
int i;
ColourTest() {}
public void setColour (int c) {
i=c;}
public static void main(String args[]) {
...
}}
```

The class ColourTest is declared abstract since the getColour() method of the Colourable interface is not implemented. Note that the setColour() method has to be declared public as it is a method of the Colourable interface.

There are main differences between an abstract class and an interface. **These differences are summarized as follows**:

Abstract Class	Interface
May have some methods declared `abstract`.	Can only have abstract methods.
May have `protected` properties and `static` methods.	Can only have `public` methods with no implementation.
May have `final` and nonfinal data attributes.	Limited to only constants.

Default Methods In Java 8

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.

Example 4 (Trace)

```
interface TestInterface  {
public void square(int a);
default void show(){System.out.println("Default Method Executed");
  } }

class TestClass implements TestInterface {
  public void square(int a)    {    System.out.println(a*a);    } }

 public class Main{
  public static void main(String args[])   {
     TestClass d = new TestClass();
     d.square(4);
      d.show();    } }
```

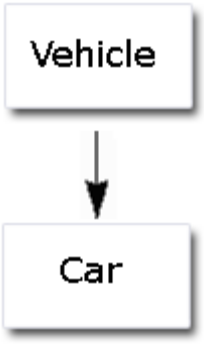
Output:

```
16
Default Method Executed
```

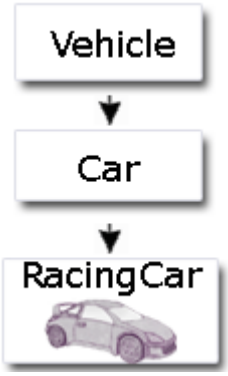
Multiple Inheritances Using Interface

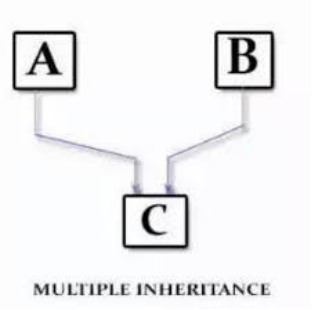
There are three types of inheritance in Java

Simple Inheritance, Multilevel Inheritance and multiple inheritances.



Simple Inheritance





Multilevel Inheritance

Multiple inheritance is the mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritances. Java does not support multiple inheritances, but the multiple inheritances can be achieved by **using the interface.**

Example 5

public class Employee extends Person, Employment { /// error
…… }

Here, Person is a concrete class that represents a person, while Employment is another concrete class that represents the details of a person who is employed. If you could only put them together, you would have everything necessary to define and implement an Employee class. Except in Java - you can't. Inheriting implementation from more than one superclass - multiple implementation inheritance - is not a feature of the language. Java allows a class to have a single superclass and no more. On the other hand, a class can implement multiple interfaces. In other words, Java supports multiple interface inheritance. **What is actually meant is that it does not support multiple implementation inheritance.**

Example 6 (Trace)

```
public interface I {
void x(); }
public class A implements I {
public void x() { System.out.println("in A.x"); }
public void y() { System.out.println("in A.y"); }}
public class B extends A {
void z() {
x(); y(); }
Public class Main{
public static void main(String args[]) {
A aa = new A();
B bb = new B();
bb.z(); } }
```

The following output
in A.x
in A.y

Suggesting that the methods x() and y() of class A have been invoked. Class B, being the subclass of class A, inherited not only method y() but also method x() which is a method of the interface I.

Multiple Inheritances Using Interface

Example 7 (Trace)

```
interface vehicleone{
    int  speed=90;
    public void distance(); }

interface vehicletwo{
    int distance=100;
    public void speed(); }

class Vehicle  implements vehicleone,vehicletwo{
    public void distance(){
            int  distance=speed*100;
            System.out.println("distance travelled is "+distance);      }
    public void speed(){                int speed=distance/100; }}

class Main{
    public static void main(String args[]){
            System.out.println("Vehicle");
            obj.distance();             obj.speed();      }}
```

The output is:

```
distance travelled is 9000
```

Extending Interface

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

Does a subclass of a class that implements an interface also inherit the methods of the interface?

```
public interface Hockey extends Sports

{

}
```

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritances is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list. For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports, Event

{

}
```

How can we use extending multipl

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

```
package java.util;
public interface EventListener
{ }
```

An interface with no methods in it is referred to as a **tagging** interface.

Example 8 (LAB)

Below is a java interfaces program showing the power of interface programming in java. Listing below shows 2 interfaces and 4 classes one being an abstract class.

Note:. The classes *B1* and *C1* satisfy the interface contract. But since the class **D1** does not define all the methods of the implemented interface *I2*, the class D1 is declared abstract.

Also, i1.methodI2() produces a compilation error as the method is not declared in *I1* or any of its super interfaces if present. Hence a **downcast** of interface reference I1 solves the problem as shown in the program. The same problem applies to i1.methodA1(), which is again resolved by a **downcast**.

((C1)o1).methodI1() compiles successfully, but produces a ClassCastException at runtime. This is because B1 does not have any relationship with C1 except they are “siblings”. You can’t cast siblings into one another.

When a given interface method is invoked on a given reference, the behavior that results will be appropriate to the class from which that particular object was instantiated. This is runtime polymorphism based on interfaces and overridden methods.

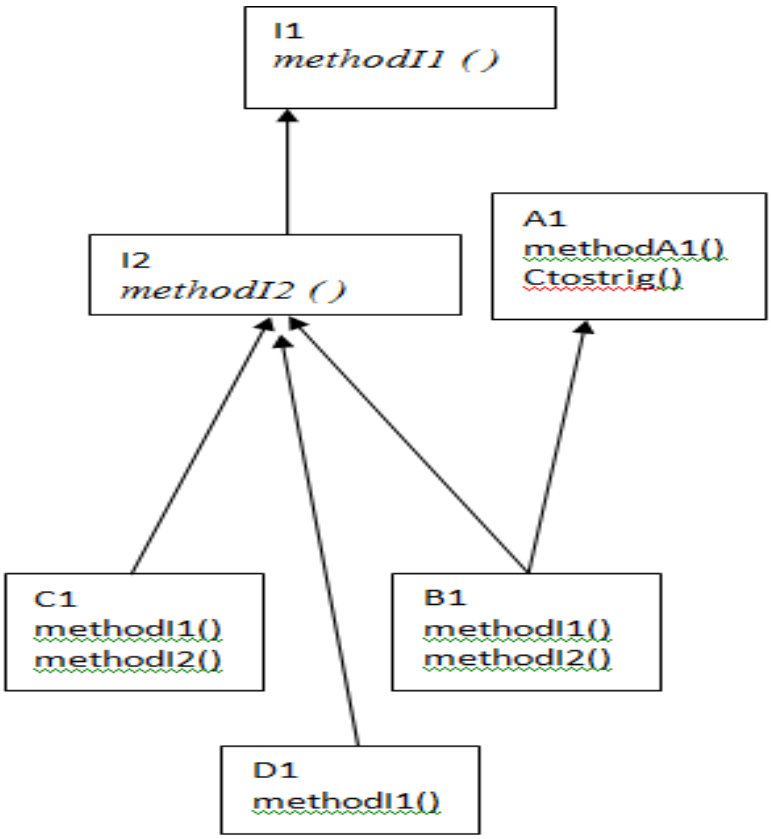
```
interface I1 {
        void methodI1(); // public  by default
}

interface I2 extends I1 {
        void methodI2(); // public  by default
}

class A1 {
        public String methodA1() {
                String strA1 = "I am in methodC1 of class A1";
                return strA1;
        }
        public String CtoString() {
                return "toString() method of class A1";
        }
}

class B1 extends A1 implements I2 {
        public void methodI1() {
                System.out.println("I am in methodI1 of class B1");
        }
        public void methodI2() {
                System.out.println("I am in methodI2 of class B1");        }}
class C1 implements I2 {
        public void methodI1() {
                System.out.println("I am in methodI1 of class C1");}
        public void methodI2() {
                System.out.println("I am in methodI2 of class C1");        } }

// Note that the class is declared as abstract as it does not
// satisfy the interface contract

public abstract class D1 implements I2 {
        public void methodI1() {      }
  // This class does not implement methodI2() hence declared abstract. }

public class InterFaceEx {
        public static void main(String[] args) {
                I1 i1 = new B1();
                i1.methodI1(); // OK as methodI1 is present in B1
                // i1.methodI2(); Compilation error as methodI2 not present in I1
                // Casting to convert the type of the reference from type I1 to type I2
                ((I2) i1).methodI2(); //WHY ? i1.method2() will cause error
```

```
                I2 i2 = new B1();
                i2.methodI1(); // OK
                i2.methodI2(); // OK
                // Does not Compile as methodA1() not present in interface reference I1
                // String var = i1.methodA1();
                // Hence I1 requires a cast to invoke methodA1
                String var2 = ((A1) i1).methodA1();
                System.out.println("var2 : " + var2);
                String var3 = ((B1) i1).methodA1();
                System.out.println("var3 : " + var3);
                String var4 = i1.toString();
                System.out.println("var4 : " + var4);
                String var5 = i2.toString();
                System.out.println("var5 : " + var5);
                I1 i3 = new C1();
                String var6 = i3.toString();
                System.out.println("var6 : " + var6); // It prints the Object toString() method
                Object o1 = new B1();
                // o1.methodI1(); does not compile as Object class does not define
                // methodI1()
                // To solve the probelm we need to downcast o1 reference. We can do it
                // in the following 4 ways
                ((I1) o1).methodI1(); // 1
                ((I2) o1).methodI1(); // 2
                ((B1) o1).methodI1(); // 3
                /*
                 *
                 * B1 does not have any relationship with C1 except they are "siblings (اشقاء)".
                 *
                 * Well, you can't cast siblings into one another.
                 *
                 */
                // ((C1)o1).methodI1(); Produces a ClassCastException(لماذا خطأ)
        }
}
```

Output

```
I am in methodI1 of class B1
I am in methodI2 of class B1
I am in methodI1 of class B1
I am in methodI2 of class B1
var2 : I am in methodC1 of class A1
var3 : I am in methodC1 of class A1
var4 : toString() method of class A1
var5 : toString() method of class A1
var6 : ?????????????  H.W.
I am in methodI1 of class B1
```

I am in methodI1 of class B1
I am in methodI1 of class B1



What is the default modifier in Interface?	

Answer	`public+abstract for methods`
	
	`public+abstract for interface declaration`
	
	`public+static+final for the interface declaration variable`