**Multiple Inheritance**

Multiple inheritance is a feature of some object-oriented programming languages where a class can be inherited from multiple classes. For example if we have two classes Athlete and Student then an AthleteStudent class can be created by inheriting from both classes.

**Problem with Multiple Inheritance**

If two of the superclasses have the same method, which version of the method will be invoked by objects of the subclass?

**Java does not support multiple inheritance.** The Interface feature is provided in Java to provide some of the benefits of multiple inheritance.

**Interfaces**

An interface is a Java module which includes a collection of public abstract methods and public constants (final variables).

- A concrete class implementing an interface claims that it has implemented all the methods of the interface.
- At compile-time, the compiler will verify that claim and will issue an error if the class is concrete and has not implemented all of the interface methods.

**Motivation for Interfaces**

- **An alternative to multiple inheritance:** A class can extend one superclass and implement any number of interfaces
- **Polymorphism:** To process a collection of items which have common behavior but they are not related by an inheritance hierarchy.

**To Use Abstract Classes or Interfaces?**

- Use abstract classes if the collection of objects to be processed is of the same type, e.g. Dog, Cat, GoldFish are all types of Pets.
- Use interfaces if the collection of objects to be processed is not of the same type but have common behavior. For example, Pet, Vehicle and Planet are all Mobile, but we cannot think of one class that encompasses all objects of these classes.

**Using Interfaces**

- is-a relationship: similar to superclasses, a class that implements an interface has is-a relationship with the interface.
- Hence references (variables) of an interface type can be created and used to refer to objects of any class that implements the interface and invoke some of the methods of these classes.
- Only methods listed in the interface can be invoked directly from the interface reference. The reference must be down casted to the appropriate class before calling methods that are not declared in the interface.
- It is illegal to create object from an interface.
Example
Create an interface, Speaker, which has one method speak. Modify the Pet class to implement this interface. Create a class Philosopher which implements the interface Speaker. This class will have methods speak and lecture. The Philosopher speak method displays “Know thee self”. The philosopher lecture method displays “I speak words of wisdom …”. A third class that implements the Speaker interface is the Radio class. The speak method of the Radio class displays “This is the news from ABC radio”.

```java
Speaker [] speakers = new Speaker[3]; // creating an array with
    // 3 references not objects
speaker[0] = new Philosopher( ...); // OK a philosopher
    // is a speaker
speaker[1] = new Dog( ...);
speaker[2] = new Radio( ...);

Speaker s;
for (int i = 0; i < speakers.length; i++)
{
    s = speakers[i]; // OK - speaker[i] is-an object that
    // implements the Speaker interface
    s.speak( ); // OK - a speaker is guaranteed to have a method speak method
}

speaker[0].lecture( ); // compiler error
```

Note that the compiler only knows that speaker[0] is a Speaker, which does not declare a lecture method. Only at run-time the interpreter can determine the type of the Speaker.

To fix this, downcast:

```java
((Philosopher) speaker[0]).lecture( );
```

While this downcast can potentially cause a problem at run-time if speakers[0] is not a Philosopher, however the compiler will be satisfied with this conversion. It is up to the programmer to make sure that the object will be of the appropriate type at run-time. This can be done by using the instanceof operator to check that the type of speaker[0] is a Philosopher.