Lesson 31: Interfaces

There are basically two viewpoints when considering interfaces in Java:

The implementation viewpoint:

Consider the following superclass:

```
public abstract class Parent
{
        public abstract void method1( );
        public abstract void method2( );
        public abstract int method3(double d);
}
```

…and now the subclass:

```
public class Child extends Parent
{
        public void method1( )
        { //some code…}

        public void method2( )
        { //some code…}

        public int method3(double c )
        { //some code…}

        public int statevar1;
}
```

****

Notice that the above superclass does **absolutely nothing**. All methods there are abstract. Also, there are **no** state variables. All it does is force us to implement its methods in the subclass. If this is all a particular superclass does, then it could be equivalently replaced with an **interface**. Alter the *Parent* class by converting it to an interface as follows:

```
public interface Parent
{
        void method1( ); //notice the semicolons at the ends of these signatures
        void method2( );
        int method3(double d);
}
```

Notice that with the methods above it would be legal to start their signatures with ***public abstract***; however, even if we leave them off, they are automatically *public* and *abstract*…all because this is an **interface**. It is conventional in interfaces **not** to use *public* and *abstract* in the signatures.

Now adjust the subclass as follows:

```
public class Child implements Parent
{
        public void method1( )
        { //some code…}

        public void method2( )
        { //some code…}

        public int method3(double c )
        { //some code…}

        … some other methods…

        public int statevar1;
}
```

Notice that all the interface does here is to **force** us to implement those methods in the subclass…..**big deal**. (Actually it's a very big deal. Take a look at a short essay presented in Appendix L for four compelling reasons to use interfaces.) The *Child* class above will **refuse to compile** until **all** methods in the *Parent* interface have been implemented in *Child*.

The object viewpoint:

Expect to eventually make an **object** out of the following interface…you probably thought we could only make objects out of classes! This interface describes some of the methods found in a robot related class. As with the previous viewpoint on interfaces, this one will also force other classes implementing *RobotArm* to provide code for each of these methods (i.e. to implement the methods):

```
public interface RobotArm
{
        void moveUp( double rate, double howFar );
        void moveDown( double rate, double howFar );
        void twistLeft(double deg);
        void twistRight(double deg);
}
```

What we really want to look at here is how the *Robot* interface provides the "glue" that holds together several **cooperating classes**, specifically two different industrial robots supplied by two different robot manufacturers. They are the Lexmar 234 and the General Robotics 56A.

Let’s suppose we have two classes (*Lexmar234* and *GR56A*) that each implement *RobotArm*:

```
//The implemented methods give detailed instructions on how to manipulate the
//“arm” of this particular robot.
public class Lexmar234 implements RobotArm
{
    public Lexmar234( ) { Constructor}
    public void moveUp(double rate, double howFar) { //some code }
    public void moveDown(double rate, double howFar) { //some code }
    public void twistLeft(double deg) { //some code }
    public void twistRight(double deg) { //some code }
}

public class GR56A implements RobotArm
{
    public GR56A( ) { Constructor}
    public void moveUp(double rate, double howFar) { //some code, different
    from above }
    public void moveDown(double rate, double howFar) {//some code,
    different from above}
    public void twistLeft(double deg) { //some code, different from above }
    public void twistRight(double deg) { //some code, different from above }
}
```

So far, this is no different from the **implementation** viewpoint we previously discussed. In other words, the interface forces us to write code for those methods in classes where we specifically say *implements RobotArm*.

Now let’s find out what’s different about the **object** viewpoint of interfaces. We will look at a *Tester* class with a *main* method in which we will **create objects** from the *RobotArm* interface.

```
public class Tester
{
    public static void main(String[] args)
    {
        RobotArm lx = new Lexmar234( );
        RobotArm gr = new GR56A( );

        // Do something with the Lexmar robot
        lx.moveDown(3, 27.87);
        lx.twistRight(22.0);

        // Do something with General Robotics machine
        gr.moveUp(16.1, -23.19);
```

```
                    gr.twistLeft(18);
            }
    }
```

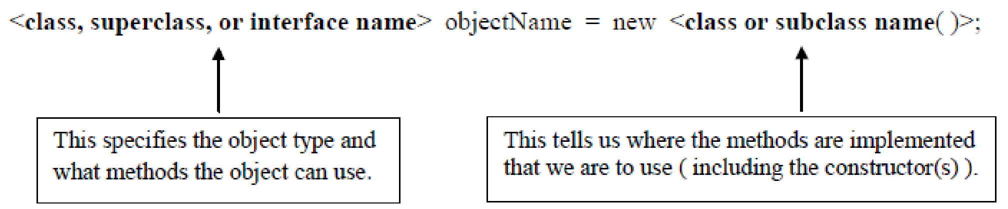
It is significant that nowhere in the above class did we say *implements* in the code. Also, notice, for example, that when we declare

```
RobotArm lx = new Lexmar234( );
```

that ***lx* is of type *RobotArm*** even though *RobotArm* is not a class; it's just an interface! This is specified by the **left side** of the above statement, and it means that *lx* can **only** use methods given in the *RobotArm* interface. The object *lx* will use these methods as **implemented in the *Lexmar234* class**. Notice this is specified on the **right side** of the above statement.

Important generalization:

All this brings us to an important generalization about classes and interfaces as illustrated in Fig 37-1.



<class, superclass, or interface name> objectName = new <class or subclass name( )>;
This specifies the object type and what methods the object can use.
This tells us where the methods are implemented that we are to use (including the constructor(s)).

Fig. 37-1 Determining object type, legal methods, and where the methods are implemented. If a method in the subclass overrides that of the superclass, then code in the subclass runs.

Miscellaneous facts concerning interfaces and implementations:

1. *instanceof* as applied to the example on the previous page:

   (gr instanceof RobotArm) returns a *true*, and

   (gr instanceof GR56A) also returns a *true*.

   Notice that the syntax is *anObject instanceof ClassOrInterface*: a *boolean* is returned.

2. **Polymorphism** is the property of being able to have methods **named the same**, while having possibly **different implementations**. For example, *twistLeft* is a method in **both** the *Lexmar234* and *GR56A* classes above, yet the implementations would likely be radically different because the two different manufacturers of these robots probably control their machines differently.

3. Saying that a class **realizes** an interface is the same as saying that it *implements* that interface.

4. It is possible to simultaneously extend another class and implement an interface. It is possible for a particular class to only extend a **single** class; however, it can implement as many interfaces as desired. Below, we show the *Redwood* class extending the *Tree* class and implementing both the *Limb* and *Leaf* interfaces.

```
public class Redwood extends Tree implements Limb, Leaf {
        //code not shown
        }
```

   Notice that when simultaneously extending and implementing, ***extends* must come first**.

5. An interface can have a **data member**. The implementing class **inherits this data member** just as if it were inheriting it from a superclass. It is permissible to override this variable in the implementing class with a variable of the same name and even of a different type.
   <**class, superclass, or interface name**> objectName = new <**class or subclass name**( )>;
   This specifies the object type and what methods the object can use.
   This tells us where the methods are implemented that we are to use (including the constructor(s)).

6. **Interfaces can be generic** in much the same ways as classes. (See Appendix AF for rules concerning how to create and use generic classes.) The following is an example of a generic interface and two implementing classes:

```
public interface MyInterface<T> { //methods signatures that use T }

public class MyClass1 <T> implements MyInterface<T> {
        //code might use T
        }

public class MyClass2 implements MyInterface<Integer> { //code }
```