INTERFACES

In Java (and other object-oriented programming languages) it sometimes becomes important to be formal about what functionality a class object will provide. Sometimes we want to have a group of classes that all provide access to similar methods and behavior, but actually represent different things and need different implementations for their methods. Sometimes we will even want to be able to use objects instantiated from these classes interchangeably with each other in our code.

Java provides a few different approaches to this question. Here we discuss the idea of using an interface to accomplish this functionality. An interface is like a kind of contract with the user of a class – any class that implements a particular interface guarantees that it will provide code for the methods defined by that interface.

Example 1: The Shape Interface

```java
public interface Shape {
    public double getArea();
    public double getPerimeter();
    public void setColor(int color);
    public int getColor();
}
```

Example 1 shows a very simple interface definition. Notice that it looks very similar to a class definition, with a few modifications. The first is the use of the `interface` keyword in place of the `class` keyword in the interface declaration. This instructs the Java compiler that it is dealing with an interface rather than a class.

The second thing to note is that the methods in the interface do not have any code associated with them – the interface has only the method signatures. This underscores the first key point about interfaces – **interfaces are not classes**. They are contracts that provide guarantees about what a class will provide.

In Example 1 above there are four method signatures contained in the interface description. Any class that implements this interface must implement these four methods. By implementing the interface the class is explicitly guaranteeing that these methods will be available for other code to use.

Note again that **interfaces are not classes**. They contain no method information. You cannot instantiate an object using an interface – the interface does not even contain constructor methods!
Example 2: The Square class

```java
public class Square implements Shape {

    private double length;
    private int color;

    public Square(double length) {
        this.length=length;
        color=0;
    }

    public double getArea() {
        return length*length;
    }

    public int getColor() {
        return color;
    }

    public double getPerimeter() {
        return 4*length;
    }

    public void setColor(int color) {
        this.color=color;
    }

    public void setLength(double length) {
        this.length=length;
    }

    public double getLength() {
        return length;
    }
}
```

Example 2 shows a class that implements the Shape interface. This class is just like any other Java class except for the addition of the “implements” keyword in the class declaration. “implements Shape” tells the Java compiler that this class is implementing the Shape interface and guarantees that the methods advertised by the Shape interface exist in the class – right down to the exact same method signature.

Notice that the class is not restricted to just the methods required by the interface it is implementing – additional methods that are not part of the interface may be defined in the class. (See below, however, for a discussion of some complications that can arise when adding methods outside those defined by the interface.)

Example 3 shows a second example of a class implementing the same interface. Notice that the internal code is in some places quite different from the code for the Square class above. Despite this the same methods are implemented in both classes, and the basic behavior of those methods is different only in the sense that what those methods are doing is different for Squares and Circles.
Example 3: The Circle class

```java
public class Circle implements Shape {

    private double radius;
    private int color;

    public Circle(double radius) {
        this.radius = radius;
        color = 0;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public int getColor() {
        return color;
    }

    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }

    public void setColor(int color) {
        this.color = color;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getCircumference() {
        return getPerimeter();
    }
}
```

Why use Interfaces?

Why would a programmer ever want to use an interface? There are times when there is a need in code to find a general solution to a problem. Suppose we have a drawing program that puts various shapes onto the screen, where each shape is stored internally as an object. We might try to create a general “Shape” class that can handle any kind of shape that we might want to come up with – creating many if/else if/.../else conditions in our methods to handle the various differences between types of shapes.

A more natural Object Oriented approach suggests that we might want to eliminate those ugly conditions in our methods and have each class encapsulate the idea of a single type of shape – keep the code specific to circles in a Circle shape, to rectangles in a Rectangle shape, to hexagons in a Hexagon shape, and so on. Or maybe at least have a few classes for shapes whose behavior is exceptional – such as classes for Circles, Rectangles, or Triangles.
However, if we create separate classes like this we run into a new problem - now we have situations where we need to move that ugly conditional logic out into our main program. Imagine you wanted to create an ArrayList of the various shapes you want to have on the screen – how would you declare that ArrayList if every shape could potentially be of a different class? However, if we use an interface for these classes we can get around this problem. Example 4 shows a simple bit of code that iterates over an ArrayList of shapes and prints their areas to the screen.

**Example 4: ShapeTest**

```java
public class ShapeTest {

    public static void main(String[] args) {
        ArrayList<Shape> myShapes = new ArrayList<Shape>();

        myShapes.add(new Square(10.0));
        myShapes.add(new Circle(5.0));
        myShapes.add(new Rectangle(5.0, 6.0));
        myShapes.add(new Circle(1.0));
        myShapes.add(new Square(1.0));

        Iterator<Shape> myIter = myShapes.iterator();
        while (myIter.hasNext()) {
            Shape tmpShape = myIter.next();
            System.out.println(tmpShape.getArea());
        }
    }
}
```

The above code outputs the following to the console:

```
100.0
78.53981633974483
30.0
3.141592653589793
1.0
```

(Notice that the above code uses the Rectangle class which we have not defined in this document. The Rectangle class is another class that implements Shape and computes its area and perimeter as expected. The implementation of the Rectangle class is left as an exercise for the reader, but it is similar to the implementation of the Square class.)

The test in Example 4 above makes use of a property of object-oriented programming languages known as **polymorphism**. Polymorphism means that a reference variable (such as tmpShape in the code above) can point to objects of a different type than its own so long as those objects have a type that is a subclass of the type of the reference variable. So if the reference variable has a type (like Shape) that is considered a superclass of the object (like an object of type Square), then the reference variable can treat that object as if it had the superclass type instead of the subclass type.
In other words, looking at Example 4, the reference “tmpShape” has a type of Shape. Because of polymorphism, the object that “tmpShape” points at does not have to exclusively be of type Shape – it can be of any type that is a child class of type Shape. And any class that implements an interface is considered to be a child class of that interface. So the actual object pointed at by “tmpShape” can be a Square, a Circle, a Rectangle, or any other object instantiated from a class that implements the interface Shape.

Note that this does not work in reverse. We cannot create an object of type “Shape” and have a reference of type “Square” point at it. First of all, polymorphism does not work that way – it only allows references of superclass types (also known as parent class types) to point at objects with subclass types (aka child class types). Second of all, interfaces are not classes. We cannot instantiate an object of type Shape because Shape is defined only by an interface – there is no constructor to run and nothing to instantiate.

Note also that polymorphism does not work “across” children – we cannot declare a reference to a Square object and point it at a Circle object. Polymorphism does not work that way – it only allows parent references to point at child objects.

Potential Pitfalls When Using Polymorphism

Notice a key phrase in the description of polymorphism above: “the reference variable can treat that object as if it had the superclass type instead of the subclass type.” This is important. When you exploit polymorphism to allow a parent class reference to point at a child class object, you are restricted to only using the methods in the parent class. For all intents and purposes, Java considers the object being pointed at by the reference to be of the parent class, even if the object being pointed at really is not of that type.

For example, consider the above code in Example 4. Suppose we wanted to use the getLength() method for the Square object in position 0 of the myShapes ArrayList. We might think we could do something like this:

```java
Shape tmpShape=myShapes.get(0);
System.out.println(tmpShape.getLength());
```

But if we attempt to perform those operations, Java responds with an error: “The method getLength() is undefined for the type Shape”. Even though the object really is a Square, we have told Java that we want to treat it as a Shape object – and so only the methods available to a Shape object are available for our use.

There is a way around this, but it involves reintroducing the ugly conditional testing we were trying to avoid. The following snippet of code can be used to get our Square object back out of the ArrayList of Shapes we created:

```java
Shape tmpShape=myShapes.get(0);
System.out.println(tmpShape.getLength());
```
if (myShapes.get(0) instanceof Square) {
  Square tmpSquare=(Square) myShapes.get(0);
  System.out.println(tmpSquare.getLength());
}

Two things of note are happening in this code snippet. The first is the test in the if-condition. “instanceof” tests to see if the object on the left is an instance of the class or interface on the right. If it is the expression evaluates to true, otherwise it evaluates to false.

The second item of note is what is happening in the line immediately following the if-statement. Remember that the ArrayList myShapes is an ArrayList of Shapes. This means that the get method will return back a Shape object. But we want it to be a Square object, and we know that it’s safe to assume that it IS a Square object because the “instanceof” test above has just checked it to make sure. So we use a process known as casting to change the type of the object from Shape to Square. Casting is performed by putting the type we want to change our object to in parentheses right after the equals and before the object reference.

Casting is not just something we do with ArrayLists – the most basic example of casting would be the following code snippet:

Shape myShape = new Circle(2.0);
Circle myCircle = (Circle) myShape;

Here we’ve created a new Circle object that is pointed at by the Shape reference “myShape”. If we want to get the Circle object back out, we cast it from a Shape object (which is what the Java compiler thinks it is) to a Circle object (which we know it really is). Now all of the Circle methods are available through the myCircle reference. But note that the “myShape” reference remains a Shape – a call to myShape.getCircumference() will still fail. You need to call myCircle.getCircumference() to get access to that method.