243 Week 3 Lecture - Interfaces

One problem, many solutions

Suppose we want to implement a set of classes to represent various shapes such as circles and rectangles. We want to be able to calculate the area and perimeter of each shape, regardless of its type and characteristics.

One approach would be to simply implement each class independently. The limitation is that we have not formally defined a relationship between the classes. Nothing would stop someone from adding another class, perhaps a square class, that does not implement the area and perimeter methods.

This week, we look at one mechanism Java provides to formally define relationships between objects, the interface.

One benefit of an interface is that it enforces standardization between objects. In our shape example, an interface will enforce that any class that implements the interface provides the methods to calculate the area and perimeter. Further, the interface will ensure that any methods for area and perimeter use the exact method signature. This requirement is very important, as we will see later.

Here is a simple interface:

```java
public interface Shape {

    /**
     * Calculate the area of this shape
     * @return the calculated area
     */
    public int area();

    /**
     * Calculate the perimeter of this shape
     * @return the calculated perimeter
     */
    public int perimeter();
}
```

Notice the syntax in this interface. It is not designated as a class, and the method bodies are absent. It will be the responsibility of any class that implements this interface to provide the method implementations – and that will depend on what the class is intended to do.
Note that we can not do the following with the Shape interface,

```
Shape myShape = new Shape();
```

.. because there are no implementations for the methods in the interface.

Instead, we must complete the Circle, Rectangle and Square classes that will each implement the Shape interface. The implementation of the Circle class is below:

```java
public class Circle implements Shape {
    int radius;       // radius of this circle object

    /**
     * Constructor for Circle objects.
     * @param radius the radius of this circle.
     */
    public Circle (int radius) {
        this.radius = radius;
    }

    /**
     * Calculate the area of this circle
     * @return the calculated area
     */
    public int area () {
        // Forced to cast this to an int since the
        // Interface requires an int return value.
        // PI is a double.

        return (int) (Math.PI * radius * radius);
        // or this:
        // return (int) (Math.PI * Math.pow (radius, 2))
    }

    /**
     * Calculate the perimeter of this circle
     * @return the calculated perimeter
     */
    public int perimeter() {
        int diameter = 2 * radius;
    }
}
```
Now that we have implemented the true class, we can instantiate a Circle object:

```java
Shape myCircle = new Circle(10);
```

So, why would we go to all this bother? When we define a Circle as shown in the above constructor call, it creates an is_a relationship with Shape. We can do the same thing with Rectangle and Square.

You already know that Java does not allow you to put data of different types into an array. For example, an int array can only contain primitive int values; a String array can only contain String objects (with exceptions we will talk about when we introduce inheritance). Using the Shape interface, we can create instances of Circle, Rectangle and Square and put them into a common Shape array. Why? Because Circle, Rectangle and Square each implement the same interface so they have an is_a relationship with Shape. We will see that this is a very powerful feature of interfaces. This same is_a relationship also allows us to create an ArrayList (or any other data structure) and insert any of the 3 objects we have defined:

```java
ArrayList <Shape> myArrayList = new ArrayList <Shape>();
myArrayList.add (new Circle (10));
myArrayList.add (new Rectangle (10,4));
myArrayList.add (new Square (3));
```

Another feature of interfaces is that we can easily make changes. In lab you will see an interface for a queue. If we implement the queue using an array as the underlying data structure, then later decide that an ArrayList would be a better choice, we can simply drop it in place and will not need to make other revisions to our code.

**Polymorphism**

Let’s go back to our Shape example. Suppose we create this array:

```java
Shape[] items = new Shape[5];
```

Since Circles, Rectangles and Squares share an is_a relationship with Shape, we can put them into our items array:

```java
items[0] = new Circle(10);
```
items[1] = new Circle(12);
items[2] = new Rectangle(3, 12);
items[3] = new Square(7);
...

for (int loop = 0; loop < 4; loop++) {
    System.out.println("Area is " +
                   items[loop].area());
    System.out.println("Perimeter is " +
                   items[loop].perimeter());
}

Look closely at the loop. The way the area and perimeter are calculated is very different for each object type. By using an interface, each class can implement the methods however they are needed. At runtime, Java will resolve the actual type of the object (as created when the constructor was called) and call the correct method. This is referred to as polymorphism (from the Greek for “many forms”) — we are calling a single method name, but do not get the same behavior each time.

Other Uses for Interfaces

There is another reason for using an interface in Java. Consider the problem of sorting objects of a given class. Suppose we have Vehicle objects, and each vehicle has an instance variable representing the odometer reading on the vehicle. If we cannot use < and > to compare Vehicle objects, how could we order them?

We could write a specialized version of a standard sorting algorithm such as mergesort that works for Vehicles, but it would be much better if we could use the same sorting implementation for each new class of objects to be sorted.

In this case, we can use an interface to represent a particular capability that a class may have, in addition to its main “job”. Then, a piece of code such as a sorting routine can be written to work for any class with that capability. In Java, a sorting routine is already provided, and works for any class that implements the Comparable interface.

public class Vehicle implements Comparable<Vehicle> {  
   public int compareTo(Vehicle other) {  
       // code here  
   }  
}

Actually, since we will be comparing Vehicles only to other Vehicles, the interface is specified as Comparable<Vehicle>.

It is important to note that a class may have several capabilities, and may need to implement several interfaces. This is allowed in Java, and might look like (from the Java
API):

```java
public class LinkedList<E> implements List<E>,
    Deque<E>, Cloneable, Serializable {
    ...
}
```

Iterable is not there, because List<E> in turn implements Iterable<E>, so a List must also already be Iterable!

**How to write a generic class**

Java allows us to implement data structures that can accommodate any data type. Let’s start with a simple interface for a Queue. The interface defines the basic operations for the queue. Note that this particular interface contains some methods we may not need, depending on how we decide to implement the actual queue. For example, if we are using a linked data structure `isFull()` may not apply (since we often assume a dynamically allocated structure can be of any size). However, if we implement the queue using an array, the queue can get full.

```java
public interface Queue<E> { ...
```

Here, E is not the name of any particular type. Rather, we use it as a placeholder. Once it is mentioned in the header, we can use it anywhere we would use a regular type.

We have provided an implementation of this interface as `MyGenericQueue` that uses Java’s ArrayList as the underlying data structure:

```java
public class MyGenericQueue <E> implements Queue <E> {
```

We can now create a `MyGenericQueue` that contains any object type (E); for example, a queue of `Circle` objects.

```java
    private MyGenericQueue <Circle> circleQueue =
        new MyGenericQueue <Circle>();
```

Within the generic class, Java effectively does a cut-and-paste for the generic parameter, giving us a queue of whatever data type we choose to instantiate. (More accurately, the compiler knows how and when to check types on the way in and out of the class.) This is an important design principle that we should try to achieve whether or not it is a situation calling for generics.

We have now implemented two different ideas of generalizing our solutions: interfaces and generic classes. Both of these will support modularity and code reuse in the future.