Interfaces, Comparable
April 7, 2014
Interface for Shapes

• Example from section 9.5 of text: an interface to describe shapes.

• We've seen example using Car, Truck implementing the Vehicle interface.

• We'll get more practice with this example, then start looking at the Comparable interface, how to use it to sort a list.
Interface Mechanics

• Remember that an interface consists only of *abstract methods* and possibly constants.

• An *abstract method* is a method that is declared but not implemented (i.e. a method with a name and signature but no body).

• An interface **cannot** be instantiated.

• We **can** declare variables with a type of some interface (like Vehicle or Shape) that refer to any object that implements that interface.
Shape Interface

// A general interface for shape classes.
public interface Shape {
    public double getArea();
    public double getPerimeter();
}

- This interface declares that shapes have methods to compute their area and perimeter
Implementing Interfaces

- To connect a class to an interface:
  - Declare that the class implements the interface
  - Implement (i.e. provide a method body) each of the interface's methods in the class

- For example, a Rectangle class:

```java
public class Rectangle implements Shape{
    // we have promised implementations of
    // getArea() and getPerimeter()
}
```
public class Rectangle implements Shape {
    private double width;
    private double height;

    // Constructs a new rectangle with the given dimensions.
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
}
Rectangle – Implemented methods

// Returns the area of this rectangle.
public double getArea() {
    return width * height;
}

// Returns the perimeter of this rectangle.
public double getPerimeter() {
    return 2.0 * (width + height);
}
Circle Class

- Now we want to create a Circle class that also implements the Shape interface.
- What field(s) does a circle need?
- How do we implement the two abstract methods of the Shape interface?
// Represents circle shapes.
public class Circle implements Shape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }
}


Circle – implemented methods

// Returns the area of this circle.
public double getArea() {
    return Math.PI * radius * radius;
}

// Returns the perimeter of this circle.
public double getPerimeter() {
    return 2.0 * Math.PI * radius;
}
Benefits of Interfaces

- We could create a similar Triangle class
- These three classes now form a *type heirarchy*
- The Shape interface serves as a parent type for each class.
- We can now create an array of Shapes, pass a Shape as a parameter, return type Shape, etc.
- This is an example of *polymorphism*.
// Demonstrates shape classes.
public class ShapesMain {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[3];
        shapes[0] = new Rectangle(18, 18);
        shapes[1] = new Triangle(30, 30, 30);
        shapes[2] = new Circle(12);

        for (int i = 0; i < shapes.length; i++) {
            System.out.println("area=");
            System.out.println(shapes[i].getArea() + ", perimeter=");
            System.out.println(shapes[i].getPerimeter());
        }
    }
}
Extending Shape Hierarchy

- Could easily change this client code if another shape class (like Hexagon or Ellipse) were added to the hierarchy.

- We see that any type that implements Shape can be referred to by a Shape-type variable.

- Other benefit: a class may implement more than one interface. This is how the principle of multiple inheritance is achieved in Java.
The Comparable Interface

- The `Collections.sort` method can be used to sort an `ArrayList` (`Collections` is in the `java.util` package).
- Could use this to sort an `ArrayList` of `Strings`.
public class SortExample {
    public static void main(String[] args) {
        ArrayList<String> words = new ArrayList<String>() {
            words.add("four");
            words.add("score");
            words.add("and");
            words.add("seven");
            words.add("years");
            words.add("ago");

            // show list before and after sorting
            System.out.println("before sort, words = " + words);
            Collections.sort(words);
            System.out.println("after sort, words = " + words);
        }
    }
}
Collections.sort

- This will produce:
  - [ago, and, four, score, seven, years]
- This works on ArrayList<String> but not ArrayList<Point>. Why?
- The String class implements the Comparable interface, but Point does not.
Natural Ordering

- We are used to putting numbers in order from lowest to highest, or alphabetizing lists of Strings.
- Data types that can be sorted have a natural ordering of values.
- A type needs to have a well-defined comparison function to have an ordering of values.
Comparison Function

- A well-defined procedure for deciding, given a pair of values of the same type, the relative order of the two values.
- Not all data types have a comparison function (for example, Strings do, while ArrayLists do not).
- In Java, any type that has a natural ordering should implement the Comparable interface.
Comparable

```java
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- Here, the "T" is an example of a generic type. The "T" is short for the word "Type".
- The `compareTo` method is the comparison function for the Type that implements Comparable.
compareTo

- The convention for compareTo is that, when this method is called on an object, it should return one of:
  - A negative number to represent a less-than relationship
  - 0 to indicate equality
  - A positive number to represent a greater-than relationship
Integer Example

```java
Integer x = 7;
Integer y = 42;
Integer z = 7;
System.out.println(x.compareTo(y));
System.out.println(x.compareTo(z));
System.out.println(y.compareTo(x));
// Output: -1, 0, 1 (on diff. lines)
```
String Example

String x = "hello";
String y = "world";
String z = "hello";
System.out.println(x.compareTo(y));
System.out.println(x.compareTo(z));
System.out.println(y.compareTo(x));
// Gives -15, 0, 15 (on diff. lines)