Lecture 25: Abstract Classes

- We're going to consider a group of classes that store information about various shapes
  - (show the code - Circle, Rectangle, Square)
  - If we were really going to use these, we might have a few more methods
  - But this is enough to explore the design issues
- (show the client program)
  - We have to declare the array as type Object[] because it stores a combination of different types of objects
  - (run the code)
  - When we run the code, it prints out the shapes but throws an exception when sorting the Square
  - Why?
    - Because we didn't specify how to compare shapes!
    - We don't implement Comparable
  - So let's fix this by having the Square implement Comparable
    - Change the class header - "implements Comparable<Square>"
    - How can we compare two shapes? *By AREA*
    - Writing this compareTo is kind of tricky, because our fields have type double

```
public int compareTo(Square other) {
    double difference = area() - other.area();
    if (difference < 0)
        return -1;
    else if (difference > 0)
        return 1;
    else // difference == 0
        return 0;
}
```

- (rerun) - This doesn't fix our problem! Now it's complaining about the Rectangle
  - What's the problem?
    - We only modified the Square, not the Rectangle or Circle
    - So we should make each of those comparable?
    - A bigger problem: We can only compare a Square with a Square, not a Circle or Rectangle
    - But we want a compareTo that can compare to any shape
  - What can we do?
    - We could have a Shape interface that all the shapes implement
    - And make each shape implement Comparable<Shape>

```
public interface Shape {
}
```

    - And change the compareTo so that it takes a Shape as a parameter
    - (doesn't compile)
    - We also have to say that we haven't told Java that Shape has an area() method

```
public interface Shape {
    public double area();
```

```
        }
```

  - We're still missing something very important
    - We haven't said that the Square **is-a** Shape
    - So we have to add "implements Shape"
    - Even better --> have Shape extend Comparable<Shape>, and then have Square implement Shape --> less to write overall
- Copy this to the other Shapes
  - Change the class headers to implement Shape
  - Copy the compareTo method
- (compile and run) - it works!
  - In fact, we can make an improvement to the client program
  - The array can be more specific now - instead of saying it's of type Object, we can say that everything inside it is a Shape
- But you should still feel very dirty right now - **what did we do wrong?**
  - We copy/pasted an identical compareTo method
  - This kind of redundancy is bad - it's more to manage if we ever want to change things
- We talked about inheritance and the 20-page employee handbook that all employees share
  - We want something like that here - SHARED BEHAVIOR
  - So what do we do?
  - Change the interface into a class
  - Move the compareTo method into the new Shape class

```
public class Shape implements Comparable<Shape> {
    public double area();

    public int compareTo(Shape other) {
        double difference = area() - other.area();
        if (difference < 0)
            return -1;
        else if (difference == 0)
            return 0;
        else // difference > 0
            return 1;
    }
}
```

  - But this doesn't compile :(

```
Error: missing method body, or declare abstract
```

  - What's the problem?
    - We have an area method, but it doesn't define any code!
    - The definitions are in the individual shape classes, and are each different
  - What can we do?
    - Delete that method from the Shape class
      - Doesn't work, because the compareTo needs the area() method
    - Sometimes we have "dummy" method stubs
      - Return a dummy value

```
public double area() {
    return 42.42;
```

```
            }
```
            - This allows us to compile, but it’s bad --> what if a subclass doesn’t override it?
- It’s better to leave the method unspecified, like an interface
    - So what’s the solution?
    - The error message we got actually tells us what to do!
    - We should “declare abstract”
    - It turns out that **abstract** is a modifier just like “public” and “static”
        - We can add it to the method header, and it just means that “I am not defining this method yet”
    - But it still doesn’t compile!
        - It says that the class isn’t declared abstract
        - If you want to have an abstract method, the entire class has to be declared abstract
- We have to change the rest of the classes to extend this class, rather than implementing an interface
- So now we see today’s topic: abstract classes
    - There is a continuum of class types in Java

```
concrete <---+-----------------------+------------------------+----> abstract
             |                       |                        |
      concrete class          abstract class              interface
```

   - We have normal classes, which declare ALL methods
    - We have interfaces, which are ONLY method headers - declare NO methods
    - And in between is the abstract class, which defines SOME methods
- What if we try to do something like this?

```
            Shape s = new Shape();  // illegal
```

   - It won’t work, because some of the Shape’s methods are not declared yet
    - You can’t instantiate an instance of an abstract class
    - But you can use the abstract class as a variable type, just like with an interface

```
            Shape s = new Rectangle(20, 30);  // legal
```

- We can relate this to the idea of an Employee from last week
    - Everyone is an employee, and they have some common behaviors (the 20 page booklet), but you can’t have JUST an employee
    - Imagine if someone asked you “what do you do?” and you said “I’m an employee”
        - It’s true, but you must have a specialization
    - In our company, we’d probably have Employee be an abstract class
        - Because it’s a useful way to share behaviors
        - While it prevents anyone from being JUST an employee
- There’s still some redundancy in our classes
    - The toString method is very similar - with different names but same string otherwise
    - *What can we do?*
    - Copy paste the toString into the abstract class
    - But we need to distinguish the name
    - We could add a field that stores the name of the shape, and pass it in to the constructor
    - Then the subclasses will call the super() constructor, passing in their name

```
public abstract class Shape implements Comparable<Shape> {
     private String name;
     public Shape(String name) {
         this.name = name;
     }

     ...
 }
```

- Finally, there's one more keyword of interest
  - The "final" keyword
  - Where have we seen it before?
    - In class constants
  - What does it mean?
    - It means that whatever it's describing cannot be changed
  - We can use it on methods, which means that subclasses CANNOT override the method
  - So if you're worried that a subclass might mess something up, make the method final
  - The toString and compareTo should both be final - we don't want a subclass overriding the compareTo and always returning 1.
  - If it overrode the toString it might be able to pretend to be something that it's not (i.e. a Triangle could seem to be a Square)
- What are the benefits/disadvantages of using abstract classes (compared with normal classes and interfaces)?
  - Pro (compared with interface) - we can reduce redundancy
  - Con (compared with interface) - we use up our inheritance relationship - the class cannot extend any other class
  - Pro (compared with normal class) - we don't have to have "dummy" methods
- Another application
  - Does anyone have any ideas where else we can use this idea of an abstract class?
  - In the ArrayList and LinkedList