```
 1 // Abstract base class for shapes
 2 //
 3 // CONSTRUCTION: is not allowed; Shape is abstract
 4 //     one-parameter constructor provided for derived classes
 5 //
 6 // ******************PUBLIC OPERATIONS*********************
 7 // double area( )          --> Return the area (abstract)
 8 // boolean lessThan( rhs ) --> Compare 2 Shape objects by area
 9 // String toString( )      --> Standard toString method
10
11 public abstract class Shape
12 {
13     abstract public double area( );
14
15     public Shape( String shapeName )
16       { name = shapeName; }
17
18     final public boolean lessThan( Shape rhs )
19       { return area( ) < rhs.area( ); }
20
21     final public String toString( )
22       { return name + " of area " + area( ); }
23
24     private String name;
25 }
```

Figure 4.5 Abstract base class `Shape`

4.3 Example: Expanding the Shape Class

This section implements the derived `Shape` classes and shows how they are used in a polymorphic manner. The following problem is used:

> ***SORTING SHAPES***
>
> *Read N shapes (circles, squares, or rectangles) and output them sorted by area.*

The implementation of the derived classes, shown in Figure 4.6, is completely straightforward and illustrates almost nothing that we have not already seen. The only new item is that `Square` is derived from `Rectangle`, which itself is derived from `Shape`. This derivation is done exactly like all the others. In implementing these classes, we must do the following:

1. Provide a new constructor.
2. Examine each method that is not final or abstract to decide if we are willing to accept its defaults. For each method whose defaults we do not like, we must write a new definition.