Interfaces

Declaring an interface

[access] interface { 
    // list of method signatures
} 

eg,

public interface Shape { 
    public void draw();
    public void setSize(int height, int width);
    public int getArea();
} 

Interfaces cannot have constructors, fields or implemented methods (method signatures must end with a semi-colon).

Implementing an interface

[access] [abstract] class implements 
{ 
    // must implement all methods declared by interface
} 

eg,

public class Circle implements Shape { 
    public void draw() {
        ... // implementation
    }
    public void setSize(int height, int width) {
        ... // implementation
    }
    public int getArea() {
        ... // implementation
    }
} 

Concept

An interface is another way to specify that a class "is a" something else, or rather than an object is part of some conceptual category. Consider the following code snippet:

Circle c = new Circle(); 

An instance is implicitly in the category of its class type: c is a Circle. Because class Circle implements Shape, c is a Shape, as well. Thus, c may be used in contexts expecting a Circle, and in contexts expecting a Shape.

For example, suppose we had a kind of graphic component, Window.
The field `myShapes` may contain instances of `Circle` and `Square`, but each is uniformly treated as a `Shape` (only the `draw()` method is used). In fact, if a new type of `Shape` were introduced, such as a `Pentagon`, none of the existing code in `Window` (or in `Circle`, `Square` or `Shape`) would need to be modified. Any number of new classes may implement `Shape` and be used in code expecting `Shape`-like objects.

Conceptually, implementing an interface is the same as extending a class. They are both a form of subclassing. The implementing or extending class becomes an is a of the superclass or interface. Thus, it is important that the implementing class honestly implement the inherited methods (class `Cowboy` should not implement `Shape` and thus implement `draw` because then somebody's going to get hurt). See `Inheritance` for more on why subclasses ought to be behavioral subclasses.