Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-five)

5.2: Inheritance

In the last module we discussed a Circle class, which presumably was part of a drawing program of some kind. A real drawing program would also need a Square class, and a Triangle class, and many others. You might create a picture by arranging a collection of circles, squares, and triangles. What would the collection itself look like? Well, perhaps it could be an array of . . . hmm. An array of what? Arrays have to contain a specific type of data, don't they? The answer is yes, of course, but that doesn't mean we can't use an array for our collection. It just means that, one way or other, we need to arrange for circles, squares, and triangles to all be somehow related, so that when it is appropriate, we could treat each just as a shape. An array of shapes could then represent our picture, and everything would work.

This makes good sense, of course, because circles, squares, and triangles have a lot in common. Each has a position, and each can be moved, for example:

```
public class Shape
{
    public double m_x, m_y;
    public Shape(double x, double y }
    { m_x = x; m_y = y; }
    public void moveTo(double x, double y)
    { m_x = x; m_y = y; }
    public void moveBy(double x, double y)
    { m_x += x; m_y += y;}
}
```

Now we can define a circle as a shape that has a unchanging radius:

```
public class Circle
       extends Shape        // See Note #1 below
{
   private double m_r;      // Note #2
   public Circle(double r)
        {
      super(0.0, 0.0);      // Note #3
       m_r = r;
    }
    public double radius()
    { return m_r; }
}
```

```
super(0.0, 0.0);
```

actually invokes

```
Shape(0.0, 0.0);
```

If present, this must be the first line in the new class's constructor. If a constructor does not call one of the extended class's constructors, Java silently inserts "super();", a call to the default constructor.

Given the definitions above, the following code is all legal:

```
Circle c = new Circle(1.0);
c.moveTo(1.0, 1.0);       // Note #4
c.m_x = 3.0;              // Note #5
double r = c.radius();    // Note #6

Shape s = c;              // Note #7
s.m_y = 2.0;              // Note #8
s.moveBy(1.0, -1.0);      // Note #9
```

while the following is illegal:

```
//Note #10
Circle c2 = new Circle(1.0, 1.0);
double r2 = s.radius();   // Note #11
```

and the following is OK:

```
Circle c3 = (Circle) s;   // Note #12
double r3 = c3.radius();  // Note #13
double r4 = ((Circle) s).radius(); // Note #14
```

Note #1: The extends clause here means that a class is defined in terms of another class. A Circle therefore has m_x and m_y variables, and also has moveTo and moveBy methods. You may have at most one extends clause for a class; a class can extend only one other class. If you don't specify an extends clause, "extends java.lang.Object" is understood. This means that Circle inherits all the methods in the class java.lang.Object, since Shape inherits them by default.

Note #2: Note that we don't mention m_x and m_y, but they're understood. The definition of Circle explicitly mentions only the things that are new with this class.

Note #3: A class's constructor can call a constructor in the extended class using this special syntax: the call to

Note #4: We can call the Shape method moveTo( ) on the Circle object c because Circle inherits all of Shape's methods. A Circle is a Shape with some extra stuff added.

Note #5: The same goes for member variables: because m_x and m_y are public in Shape, they're also public in Circle, and we can access them.

Note #6: Of course, we can call the Circle method radius( ) on the Circle object c.

Note #7: This is perfectly legal, because a Circle is a Shape; it has all the methods and member variables of a Shape; so anything we can do to a Shape, we can do to a Circle. No type cast is needed since there is no possibility of anything going wrong (but see Note #9 about type cast below).

Note #8: While referring to the object originally pointed by c through the variable s, we can freely access the Shape part of the object, like the member m_y.

Note #9: Again, this function is visible through s because it is a method of Shape.

Note #10: Constructors, unlike ordinary member functions, are *not* inherited. This line appears to be an inspired (but incorrect!) attempt to construct a Circle with a given position by calling the Shape constructor with two arguments. The designer of the Circle class has not provided any means for the clients of Circle to access this constructor, so it cannot be used. Perhaps this is deliberate, but I doubt it. Remember this when designing your own classes: you must redeclare constructors in each subclass, adding any new initializations to the new version:

```
public Circle(double x, double y)
{
   super(x, y);
   m_r = 1.0;
}
```

Note #11: The variable s refers to the same object as the variable c. However, as far as the compiler is concerned, s is a Shape, not a Circle. We can only access Shape members through s. radius( ) belongs to the Circle class, and Shapes don't have such a member, so this is flagged as a syntax error by the compiler.

Note #12: We know that s refers to a Circle, so we can insert a *type cast* and assign the contents of s to a new Circle variable named c3. The type cast, simply a type name in a set of parentheses, tells the compiler to expect s to contain a circle, one out of presumably many kinds of shapes it could actually contain. Without the cast, this assignment will fail to compile (because the compiler does not know s is referring to a Circle; for all it knows, it could refer to a square or a triangle). Even though we added the cast, Java does not trust us! At runtime, the Java Virtual Machine checks to see that s truly does refer to a Circle, or a class that itself extends Circle. If this assumption is invalid, the JVM will throw a ClassCastException and halt (unless you catch the exception and act on it).

Note #13: Now that we have a Circle variable referring to the Circle object, we can call Circle methods again.

Note #14: Alternatively, we can combine the previous two lines into one in this fashion. Note the outer set of parentheses, which are necessary to make it clear that the variable s is being type cast, not the return value of theradius( ) method!