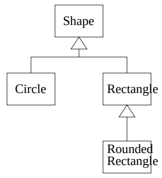
Lecture 25

This lecture primarily contains a worked example that serves as a way of revising & consolidating the material we have covered on inheritance.

So that there is something new, we also discuss abstract classes and abstract methods. We see how these enable us (a) to model the world better and (b) to impose a common interface on subclasses.

The worked example



- All shapes have a position (x, y), a `move` method for updating their position, a `toString` method and an `equals` method
- Additionally, a circle has a radius; a rectangle has a length & height; and a rounded rectangle has a length, height & arc radius
- Shapes also have areas, for which a getter is needed


```
public class Shape
{
   public Shape(int theX, int theY)
   {  x = theX;
      y = theY;
   }

   public void move(int theXChange,
      int theYChange)
   {  x += theXChange;
      y += theYChange;
   }

   public String toString()
   {  return "Shape at (" + x +
         ", " + y + ")";
   }
```



```
   public boolean equals(Object anObject)
   {  if (this == anObject)
      {  return true;
      }
      else if (anObject == null ||
         getClass() != anObject.getClass())
      {  return false;
      }
      else
      {  Shape other = (Shape) anObject;
         return x == other.x &&
            y == other.y;
      }
   }

   protected int x;
   protected int y;
}
```



```
public class Circle extends Shape
{
   public Circle(int theX, int theY,
      int theRadius)
   {  super(theX, theY);
      radius = theRadius;
   }

   public String toString()
   {  return "Circle of radius " + radius
         + " at (" + x + ", " + y + ")";
   }

   public boolean equals(Object anObject)
   {  if (! super.equals(anObject))
      {  return false;
      }
      Circle other = (Circle) anObject;
      return radius == other.radius;
   }

   protected int radius;
}
```



```
public class Rectangle extends Shape
{  public Rectangle(int theX, int theY,
      int theLength, int theHeight)
   {  super(theX, theY);
      length = theLength;
      height = theHeight;
   }

   public String toString()
   {  // body as you would expect
   }

   public boolean equals(Object anObject)
   {  if (! super.equals(anObject))
      {  return false;
      }
      Rectangle other = (Rectangle) anObject
      return length == other.length &&
         height == other.height;
   }

   protected int length;
   protected int height;
}
```

```
public class RoundedRectangle
   extends Rectangle
{  public RoundedRectangle(int theX,
      int theY, int theLength,
      int theHeight, int theArcRadius)
   {  super(theX, theY, theLength,
         theHeight);
      arcRadius = theArcRadius;
   }

   public String toString()
   {  // body as you would expect
   }

   public boolean equals(Object anObject)
   {  if (! super.equals(anObject))
      {  return false;
      }
      RoundedRectangle other =
         (RoundedRectangle) anObject;
      return arcRadius == other.arcRadius;
   }
   protected int arcRadius;
}
```


Some client code

```
Shape s = new Shape(100, 200);

Circle c1 = new Circle(10, 20, 5);
System.out.println(c1);
c1.move(2, 3);

Circle c2 = new Circle(12, 23, 5);
System.out.println(c1.equals(c2));

RoundedRectangle r1 =
   new RoundedRectangle(30, 40, 10, 5, 1);
RoundedRectangle r2 =
   new RoundedRectangle(30, 40, 10, 5, 2);
System.out.println(r1.equals(r2));
```


Abstract classes

- As we move up a class hierarchy, classes typically change from being concrete to being abstract
- A concrete class is meant to have objects instantiated from it
- An abstract class cannot be instantiated, and is defined using the keyword `abstract` (and we use italics in diagrams)
- We could choose to make `Shape` an abstract class:

  ```
  public abstract class Shape
  ```

- The following will now give a compiler-error:

  ```
  Shape s = new Shape(100, 200);
  ```


Adding a getter for areas to *all* subclasses

- Consider adding a `getArea` method to the subclasses
- How can we *ensure* that all subclasses (current & future) will have a `getArea` method?
  - We could define a ‘dummy’ version in `Shape`, which will be inherited, e.g.:

    ```
    public double getArea()
    {  return 0;
    }
    ```

  - This isn’t satisfactory: (a) it’s a hack, and (b) it doesn’t force the programmer to override with a sensible definition

Abstract methods

- Concrete classes must have implementations (bodies) for all methods
- Abstract classes do not have to provide bodies for all methods
- Methods without bodies are abstract methods, e.g.:

  ```
  public abstract double getArea();
  ```

- We use abstract methods as placeholders for methods that can sensibly be mentioned at superclass level, but which are going to be properly implemented in a variety of ways at subclass level
- The programmer must override abstract methods in *every* concrete subclass; compiler-error if not

Summary: The usefulness of `abstract` classes

- Classes can be abstract if they are ones we don’t ever want to instantiate (don’t ever want to create objects from them)
- They can be used to ensure that all their subclasses have a common core `public` interface:
  - They provide a resting place for concrete methods & variables inherited into (and, therefore common to) all subclasses
  - If there are `abstract` methods, concrete subclasses must override the `abstract` methods
- (The `abstract` classes may give us a useful type for polymorphic variables — see later)