Laboratory 9: Implementing a Curve Class

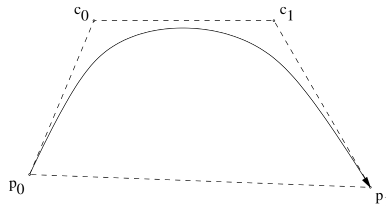
Objective. To consider the various issues associated with developing a new class.

Discussion. The `element` package, as distributed, does not have any objects that support the drawing of smooth curves or *splines*. In this lab we will develop a new class, called a `Curve` that implements Bézier curves. We will accomplish this in two steps. First we will construct a bare-bones class with only a few methods. We will then provide all the methods required by the `Drawable` interface and develop a full-fledged `Drawable` object that could be included as an extension to the `element` graphic package. This process is similar to the incremental construction of most objects in Java.

Procedure.

Initially, you are to build a small class, `Curve`, that has three constructors, drawing methods called `drawOn`, `fillOn`, and `clearOn`, as well as a `toString` method that would be called if a `Curve` was printed to an output stream.

1. Recall that the code supporting the drawing of Bézier curves is described in Section **??**. A natural method for representing these curves is a group of four points:



2. Once you determine how you will support your `Curve` with instance variables, you may start development of your class by filling out the details of the following class definition (you may implement `fillOn` by calling `drawOn`):

```
public class Curve
{
    private Pt p0,c0,c1,p1;

    public Curve(Curve c)
    // post: constructs curve with the same control points as c

    public void drawOn(DrawingWindow d)
    // post: draws the curve on window d

    public void fillOn(DrawingWindow d)
    // post: draws (like draw) curve on window d
```

```
    public void clearOn(DrawingWindow d)
    // post: erases curve from window d

    private Pt bez(Pt p0, Pt c0, Pt c1, Pt p1, double t)
    // pre: p0, p1 are endpoints; c0, c1 are control points
    //      0 <= t <= 1
    // post: returns the point along Bezier curve determined
    //       by p0, c0, c1, p1, and t

    public String toString()
    // post: constructs a string representation of curve
}
```

3. Write a program that tests the functionality of the `Curve` class. You should be able to draw a `Curve` in the `DrawingWindow` with the `Curve`'s `drawOn` method, but it should be impossible to use the `DrawingWindow`'s `draw` method, which takes a `Drawable` object.

4. Once you have verified that your `Curve` is drawing correctly, implement the remaining features of the `Drawable` interface:

```
public Object clone()
// post: returns a carbon copy of this curve

public int height()
// post: returns the height of the curve

public int width()
// post: returns the width of the curve

public int left()
// post: returns the left coordinate of the bounding box

public int right()
// post: returns the right coordinate of the bounding box

public int top()
// post: returns the top coordinate of the bounding box

public int bottom()
// post: returns the bottom coordinate of the bounding box

public Pt center()
// post: returns the center of the bounding box

public void center(Pt c)
// post: re-center the curve at point c
```

5. Demonstrate that your class is, indeed, a drawable class by indicating that it implements the `Drawable` interface. Test your code by drawing a curve in the `DrawingWindow` with the `DrawingWindow`'s `draw` method.

Thought questions. Consider the following questions as you complete the lab:

1. What changes in the interface would be necessary to allow the adjustment of the various control points of a `Curve`?
2. Suppose you could fill in a triangle. How would you fill in a `Curve`?
3. The drawing of a `Curve` takes considerably longer than drawing a `Line`. How would you measure the time needed?
4. Suppose you wished to make a new `MultiCurve` object, which is a series of `Curves` spliced together with *smooth* joins. What constraints would be put on the individual curves?