OO Ideas and Plotting Algorithms

James W. Cooper

The process of learning OO programming at first involves a complete shifting of mental gears, sometimes called a *paradigm shift*. You have to start thinking about objects that conatin both data and methods rather than just arrays or structures. This is more true than ever when you write programs in Java, because you absolutely have to write all of your programs using objects.

Recently, a swim coach asked me if I could devise a program to plot swimmers' progress over time, where they could compare several strokes or distances at once. As I began to consider the design for this program, I realized that it was an ideal example of how object-oriented programming actually *simplifies* an otherwise complicated project.

The simple example of plotting multiple curves that I outline below would be much more difficult to write if we didn't use OO techniques to encapsulate some of the information we want to plot.

The Plot Outline

Now let's consider the problem we want to solve. A coach provides us with a set of times for one or more swimmers and wants us to plot these times versus the dates they were achieved. He would also like to see several different strokes or several distances of the same stroke on the same chart, to compare improvements. Now, in general swimmers do not swim all events equally frequently or at the same meets, so the number of data points for each stroke/distance will differ and the dates when these times were achieved will differ.

While we would prefer to use some standard plotting Bean to plot these data, there are a couple of reasons why we might choose to simply draw the lines ourselves inside a panel. First, it is difficult to find good, inexpensive plotting beans, and second, most plotting algorithms in standard packages (like Excel) require that one or both of the axes be integral, rather than both being values that can vary continuously. Both the dates and the times in this problem are in fact effectively floating point single precision numbers.

The problem we have to solve is that we need to plot

- A variable number of arrays
- Each array has an unpredictable number of data points.

Clearly it is difficult to represent these data as two dimensional arrays, since we don't know how may of them there will be or how long they will be. While we could use a series of arrays to represent these data and redimension them with *new* to change their size as needed, this can become very complex to keep track of.

The Plot Thickens

Rather than using arrays, let's consider using *objects* to represent these data and encapsulate this unpredictability. The result of using objects like this can turn a potential jungle of arrays into an incredibly simple, elegant little program that is easy to understand and easy to maintain.

Remember that an object in OO programming is a combination of a structure which holds data and the subroutines or *methods* that can operate on that data. Just as important, you can have any number of *instances* of an object, with each holding different data, but utilizing the same methods.

The objects we'll use in this program are

- A time-date pair object
- A time-date pair vector
- A plotting display object.

The DataPoint Object

The advantage of doing things this way is that each object knows about its data and how it must be handled and converted to a convenient format. For example, Table I shows the form of the date and time data point pairs in our input file:

100 yd Freestyle	
103.66	10-28-95
101.53	11-02-95
100.95	12-09-95
59.37	12-15-95
109.49	1-06-96
100.23	1-20-96

200 yd Freestyle	
216.80	11-02-95
209.18	12-01-95
215.27	12-09-95
209.65	1-20-96
211.45	1-01-96
207.47	2-10-96
206.11	2-23-96
209.87	3-23-96

Table 1 - Typical swimmer data

You see that the minutes and seconds are represented in these datafiles *without* the interve ning colon. However, whether the colon is present or not, we have to convert the data to seconds in order to plot it linearly. Likewise, we need to convert the date values into some sort of linear form. These conversions are best done inside a simple **DataPoint** object which stored the data and returns the values in a convenient linear form, as shown below.

```
public class DataPoint {
  private float time;
```

```
  private floatDate date;

  public DataPoint(float times, String dat)   {
     time = makeSeconds(times);
     date = new floatDate(dat);
  }
//--------------------------------
private float makeSeconds(float t) {
   int mins;
   float secs;
   //convert time to seconds and store it
   mins = (int)t / 100;
   secs = t - (mins * 100);
   return mins * 60 + secs;
}
//--------------------------------
public float getTime() {
   return time;      //return time in seconds
}
//--------------------------------
public float getDate() {
   return date.getFloatDate();
 }
}
```

Dates in Java

Dates in Java are best handled using the GregorianCalendar class, since most of the Date class methods have been deprecated. Further, to convert data from a string like "1-20-96" to an actual date representation, we use the SimpleDateFormat class. Finally, since we want to return dates as fractions along a year timeline axis, we create a floating point representation where the year is the integer and the fraction is the number of 366ths of the way through the year. This is shown in the *floatDate* class:

```
public class floatDate extends Date  {
   GregorianCalendar date;
   //-------------------------------
   public floatDate(String dat)    {
      date = new GregorianCalendar();
      SimpleDateFormat fmt = new SimpleDateFormat("MM-dd-yyyy");
      date.setTime(fmt.parse(dat, new ParsePosition(0)));
    }
   //-------------------------------
   public float getFloatDate()    {
   float day = date.get(Calendar.DAY_OF_YEAR);
   float year = date.get(Calendar.YEAR) + (day/366.0f);
   return year;
   }
}
```

The StrokeSet Object

Now we can create the next more complex object, the *StrokeSet* object, where each instance of this object contains one array of *DataPoint* objects. Since the array may be of

any size, we implement the array using the Vector class. But rather than just using a vector alone, we create the StrokeSet object which keeps track of the maximum and minimum values of the time and date axes as we add new points into the object. Our calling routine creates an instance of the StrokeSet object and allows it to read from a data file. It then creates an instance of a DataPoint object for each line it reads from the file:

```
public class StrokeSet {
  private Vector dataList;
  private float minTime, maxTime;
  private float minDate, maxDate;
  //---------------------------------------------
  public StrokeSet()   {
  SimpleDateFormat  fmt = new SimpleDateFormat();
  dataList = new Vector();
  minTime = 10000;
  maxTime = 0;
  minDate =  new floatDate("01-01-2100").getFloatDate();
  maxDate = new floatDate("01-01-1900").getFloatDate();
  }
  //---------------------------------------------
  public void readFile(String file)   {
     String s = "";
     InputFile f = new InputFile(file);
     s = f.readLine();
     while(s != null)      {
        int i = s.indexOf(",");
        if (i > 0)         {
        String dt = s.substring(0, i);
        float time = new Float(s.substring(i+1).trim()).floatValue();
        addPoint(time, dt);
        }
        s = f.readLine();
     }
     f.close();
   }
```

As we add each dataPoint object into the strokeSet collection, it recalculates the x and y maxima and minima:

```
  public void addPoint(float times, String dat)   {
    DataPoint dp;
    dp = new DataPoint(times, dat);
    dataList.addElement(dp);

    //recalculate mins and maxes
    if (dp.getTime() > maxTime) maxTime = dp.getTime();
    if (dp.getTime() < minTime) minTime = dp.getTime();
    if (dp.getDate() < minDate) minDate = dp.getDate();
    if (dp.getDate() > maxDate) maxDate = dp.getDate();

  }
```

Note that we initialize the max and min variables when we create each instance of the class so that we are comparing the array values to something meaningful:

So now you see the crux of this OO design: each *instance* of the *StrokeSet* object contains one array of DataPoints. Each array can be of a different size and we can get the maxima and minima of each one by asking each StrokeSet instance for it. We can get the size of each array from the object and we can get the values of each datapoint from the object using the enumeration methods of the Vector object. The advantage is that they can all be different sizes (or even empty!) without any real modification to our design.

The Climax of the Plot

Our final object is the form that displays the plotted data. We create a StrokePlot object which will plot data from a set of StrokeSet objects. It has an *add* method which recalculates the overall scale of the plot whenever we add in another StrokeSet:

```
public void add(StrokeSet st)   {
     if (xmin > st.getMinDate()) xmin = st.getMinDate();
     if (xmax < st.getMaxDate()) xmax = st.getMaxDate();
     if (ymin > st.getMinTime()) ymin = st.getMinTime();
     if (ymax < st.getMaxTime()) ymax = st.getMaxTime();
     strokes.addElement(st);
     rescale();
  }
```

Inside our plot module, then, we just store these strokeSet objects in another Vector: Then to plot the data, we just find the maxima and minima for all the strokeSets and set the scaling factors accordingly.

```
private void rescale()   {
     Dimension sz = getSize();
     height = sz.height;
     width = sz.width;
     calcXscale();
     calcYscale();
  }
  //---------------------------------------
  private void calcXscale()   {
     xoffset = xmin-(0.05f*(xmax-xmin));
     xscale = 1.1f * (xmax - xmin)/ width;
  }
  //---------------------------------------
  private void calcYscale()   {
     yoffset = ymin- (0.05f *(ymax-ymin));
     yscale = 1.1f * (ymax - ymin)/ height;
  }
```

And finally, our plotting takes place in a simple loop:

```
private int calcX(float x)   {
 return (int)((x - xoffset)/xscale);
}
//---------------------------------------
private int calcY(float y)   {
  return height - (int)((y - yoffset)/yscale);
}
```

```
//-------------------------------------
public void paint(Graphics g)
  {
     int x, y;

     g.drawRect(1, 1, width-2, height-2);
     for(int s=0; s < strokes.size(); s++)
     {
        StrokeSet st = (StrokeSet)strokes.elementAt(s);
        Enumeration e = st.elements();
        DataPoint dp = (DataPoint)e.nextElement();
        int oldx = calcX(dp.getDate());
        int oldy = calcY(dp.getTime());

        while(e.hasMoreElements())
        {
           dp = (DataPoint)e.nextElement();
           x = calcX(dp.getDate());
           y = calcY(dp.getTime());
           g.drawLine(oldx, oldy, x, y);
           g.fillArc(oldx, oldy, 4,4,0, 360);
           oldx = x;
           oldy = y;
        }
        g.fillArc(oldx, oldy, 4,4,0, 360);
      }

  }
}
```

The display for two strokes is illustrated in Figure 1.

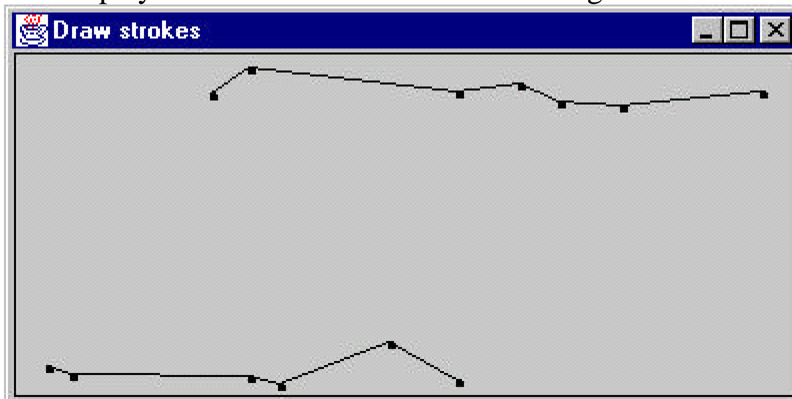

Figure 1 - A plot of a young swimmer's time improvements in the 100 and 200 yd freestyle.

Plotting of axis tick marks and labels is also the responsibility of this StrokePlot object, but is beyond the scope of this example. We'll take up some of the details of date handling and axis manipulation in another article.

Summary

You'll see here that by putting a Vector inside another object, you have all of the benefits of OO programming. Rather than worrying about separately named arrays or Vectors for each stroke data set, you can just create new instances of the StrokeSet object and keep one stroke in each instance. Further, instead of a 2-dimensional array (or two vectors) inside each StrokeSet instance, we keep a single collection of DataPoint instances, where each one contains one point pair.

So we see here that there is a little more to OO programming than just creating a couple of classes that we call from *main()* In fact, creating objects that contain other objects is frequently the key to simpler and easier to understand programs. That is the real shift in thinking that will make you’re a better Java programmer.

James W. Cooper is the author of 12 books on computing, and is working on a book on Java Design Patterns for Addison-Wesley. He also writes swimming software in his spare time.