Chapter 5: Developing a package for mathematical illustration

In the last two chapters, we have seen how to use Java2D’s graphics primitives—lines, ellipses, rectangles and general paths—along with some finer features of the rendering engine. Our aim is now to create some tools to help us construct general mathematical illustrations with relative ease. In particular, we will build mathematical primitives such as graphs, grids and axes and bundle them together into a package. The point here is mainly to demonstrate what can be done and how to begin. You are encouraged to develop additional features as you see fit.

We will begin by revisiting the graph of the function \( y = x^3 - x \) that we constructed in the last chapter and modifying it to become more general.

1: Breaking the graph into different objects

You will remember that we displayed a grid and the graph of the function in a particular viewing window. Our first step will be to create objects out of these three things. By doing so, we will create code that we can more easily reuse. However, you will see another practical advantage as it allows us to concentrate on one job at a time.

Let’s begin. One vital piece of information is the region of the plane in which we will look at the graph and the grid. We will begin by storing this information in its own object.

```java
public class BoundingBox {
    public double llx, lly, urx, ury;
    public BoundingBox(double lowerLeftX, double lowerLeftY,
                        double upperRightX, double upperRightY) {
        llx = lowerLeftX; lly = lowerLeftY;
        urx = upperRightX; ury = upperRightY;
    }
}
```

This class does nothing more than store the coordinates of the lower left corner of our viewing region \((llx, lly)\) as well as those of the upper right \((urx, ury)\). Having these points recorded in an object gives us a simple way to pass the information around.

Let’s now design a new class CubicPanel into which we will place our grid and graph. We will begin by considering the fields we need to include in the class.

Certainly, we would like for such an object to represent a region in the plane. In other words, an instance of CubicPanel should have a BoundingBox and an AffineTransform that maps the region in the plane represented by the BoundingBox into the usual graphical coordinate system. As a part of this, the panel should know its own dimensions in pixels.

Finally, we would like the panel to include a grid and a graph of the cubic function. We will eventually design Grid and CubicFunction objects; however, let’s ignore most of the details for the time being and concentrate on the panel.
Here are the fields in the class `CubicPanel`.

```java
BoundingBox bbox;
AffineTransform transform;
int width = 0, height = 0;  // Dimensions of the panel in pixels
Grid grid;
CubicFunction function;
```

To instantiate a `CubicPanel`, we will describe the viewing window by giving the coordinates of the `BoundingBox`. Here is the constructor:

```java
public CubicPanel(double llx, double lly, double urx, double ury) {
    setBackground(Color.white);
    bbox = new BoundingBox(llx, lly, urx, ury);
    grid = new Grid(this);
    function = new CubicFunction(this);
}
```

You can see here that we have made some choices about how the `Grid` and `CubicFunction` will be instantiated. At this point, we will call on these objects to plot their respective elements of the figure. However, to do this, they will need access to the `BoundingBox` and `AffineTransform` of the `CubicPanel`. Therefore, both of these objects will have instance fields that contain the `CubicPanel` onto which they will plot. The meaning of this is simply to pass the instance of `CubicPanel` on which they will plot to the `Grid` and `CubicFunction`.

We will want for the `Grid` and `CubicFunction` to be able to retrieve the `BoundingBox` and `AffineTransform` of `CubicPanel` so we include methods to do this:

```java
public BoundingBox getBoundingBox() {
    return bbox;
}
public AffineTransform getTransform() {
    return transform;
}
```

Finally, we need to give instructions for how the `CubicPanel` should paint itself by defining the method `paintComponent`. The work here should be pretty simple since we are passing the bulk of the job onto the `Grid` and `CubicFunction` instances. Before doing this, however, we need to make sure that the `AffineTransform` is set up correctly.

Remember that this `AffineTransform` should convert our mathematical coordinates into graphical coordinates. You might therefore expect that the `AffineTransform` only needs to be set up once. However, it may happen that the viewer resizes the panel; this changes the graphical coordinate system and hence requires that we update the `AffineTransform`. We will therefore check to see if the user has resized the panel and if so, we will reconstruct the `AffineTransform`. Initially the size of the panel is set to $0 \times 0$. 

public void paintComponent(Graphics gfx) {
    super.paintComponent(gfx);
    Graphics2D g = (Graphics2D) gfx;
    Rectangle size = getBounds();
    if (size.width != width || size.height != height) {
        width = size.width; height = size.height;
        float ratioX = (float) (width / (bbox.urx - bbox.llx));
        float ratioY = (float) (-height /(bbox.ury - bbox.lly));
        transform = new AffineTransform();
        transform.scale(ratioX, ratioY);
        transform.translate(-bbox.llx, -bbox.ury);
    }
    grid.plot(g);
    function.plot(g);
}

If we ignore the resizing issue, this is really quite a simple method. All it does is call on the Grid and the CubicFunction to plot themselves on the Graphics2D object. The CubicPanel does not need to worry about how they do their job. You can see that this will make it easier to reuse this class: perhaps we want to add some axes or a label. We can simply add those features into the CubicPanel in the same way.

The CubicPanel can now be used like this:

    public static void main(String[] args) {
        DrawFrame frame = new DrawFrame("CubicPanel");
        CubicPanel cubic = new CubicPanel(-2, -3, 2, 3);
        frame.getContentPane().add(cubic, BorderLayout.CENTER);
        cubic.setPreferredSize(new Dimension(300, 300));
        frame.pack();
        frame.show();
    }

Let’s now do the work of designing the Grid class. For instance fields, it only needs a reference back to the CubicPanel in which it will draw the grid.

    CubicPanel panel;

The constructor will simply store the instance of the CubicPanel passed to it in this field.

    public Grid(CubicPanel cp) {
        panel = cp;
    }
Now we define the method that plots the grid and we’re done:

```java
public void plot(Graphics2D g) {
    BoundingBox bbox = panel.getBoundingBox();
    GeneralPath path = new GeneralPath();
    int minX = (int) Math.floor(Math.min(bbox.llx, bbox.urx));
    int maxX = (int) Math.ceil(Math.max(bbox.llx, bbox.urx));
    for (int x = minX; x <= maxX; x++) {
        path.moveTo(x, (float) bbox.lly);
        path.lineTo(x, (float) bbox.ury);
    }
    int minY = (int) Math.floor(Math.min(bbox.lly, bbox.ury));
    int maxY = (int) Math.ceil(Math.max(bbox.lly, bbox.ury));
    for (int y = minY; y <= maxY; y++) {
        path.moveTo((float) bbox.llx, y);
        path.lineTo((float) bbox.urx, y);
    }
    g.setPaint(Color.lightGray);
    AffineTransform transform = panel.getTransform();
    g.draw(transform.createTransformedShape(path));
}
```

The `CubicFunction` also has an instance field for the `CubicPanel` and a similar constructor:

```java
CubicPanel panel;
public CubicFunction(CubicPanel cp) {
    panel = cp;
}
```

The only real difference is in the `plot` method:

```java
public void plot(Graphics2D g) {
    BoundingBox bbox = panel.getBoundingBox();
    int steps = 50;
    double stepsize = (bbox.urx - bbox.llx)/steps;
    GeneralPath path = new GeneralPath();
    path.moveTo((float) bbox.llx, (float) function.valueAt(bbox.llx));
    for (int i = 1; i <= steps; i++) {
        double x = bbox.llx + i*stepsize;
        path.lineTo((float) x, (float) function.valueAt(x));
    }
    g.setPaint(Color.black);
    AffineTransform transform = panel.getTransform();
    g.draw(transform.createTransformedShape(path));
}
```

Notice that we have used a method to model the specific function we are looking at:

```java
public double valueAt(double x) {
    return x*x*x - x;
}
```
Well, we’ve worked a lot harder to draw the same figure we drew in the last chapter. What have we gained? For one, it is much easier to change the viewing window: all we need to do is modify the instantiation of CubicPanel. More importantly, we will see that we can, after some slight modifications, reuse the classes we have defined here to create more general illustrations.

**Exercise 1:** The grid we have drawn above will always draw a $1 \times 1$ grid. Modify the class we gave above to allow the user to select a different grid size $a \times b$. You will want to include instance fields for the horizontal and vertical spacing that are initially set to 1.

**Exercise 2:** Following this model, design a class Axes that draws a pair of lines representing the $x$ and $y$ axes.

**Exercise 3:** Following this model, design a class GraphicalPoint that draws a point at a specified location. Modify the CubicPanel class to include points at the local extrema of the cubic.

**Exercise 4:** Add the graph of the derivative of the cubic to the figure.

### 2: Using abstract classes

In our previous example, we had a class CubicPanel, in many ways like the canvas of a painting, that contained two graphical elements, a grid and the graph of a function. Now we would like to generalize this class so that it can hold an arbitrary number of more general graphical elements such as points, lines, axes and so forth. Moreover, in an effort to reuse this class as much as possible, we would like to be able for another class to add these features into the panel. We will call our new class FigurePanel.

When we think about the kinds of graphical elements we want to display, we see there are many common features. Each will have a color and each will sit inside a FigurePanel. Moreover, each will have a method plot that plots the graphical information. However, the implementation of this method will differ from element to element. To help us here, we will use a feature of Java known as an abstract class. An abstract class is a class containing fields and methods as usual only some methods may declared abstract and the body left undefined.

Here is an example:

```java
public abstract class Plotable {
    FigurePanel panel;
    Color color = Color.black;
    public abstract void plot(Graphics2D g);
    public void setFigurePanel(FigurePanel fp) { panel = fp; }
    public void setColor(Color c) { color = c; }
}
```

This defines an abstract class called Plotable with instance fields holding a FigurePanel and a Color. In addition, there are methods to set the FigurePanel and Color. Notice that the method plot is declared to be abstract and a definition is not given for this method.

Java does not allow us to instantiate an abstract class. It’s just as well since we would have no need for an instance of Plotable. However, we can extend Plotable by defining a new class with the
method plot defined. Here is an example of a new Grid class:

```java
public class Grid extends Plotable {
    public void plot(Graphics2D g) {
        BoundingBox bbox = panel.getBoundingBox();
        GeneralPath path = new GeneralPath();
        int minX = (int) Math.floor(Math.min(bbox.llx, bbox.urx));
        int maxX = (int) Math.ceil(Math.max(bbox.llx, bbox.urx));
        for (int x = minX; x <= maxX; x++) {
            path.moveTo(x, (float) bbox.lly);
            path.lineTo(x, (float) bbox.ury);
        }
        int minY = (int) Math.floor(Math.min(bbox.lly, bbox.ury));
        int maxY = (int) Math.ceil(Math.max(bbox.lly, bbox.ury));
        for (int y = minY; y <= maxY; y++) {
            path.moveTo((float) bbox.llx, y);
            path.lineTo((float) bbox.urx, y);
        }
        g.setPaint(color);
        AffineTransform transform = panel.getTransform();
        g.draw(transform.createTransformedShape(path));
    }
}
```

Notice that this class appears to have only one method plot. However, it inherits the methods setFigurePanel and setColor from Plotable along with the instance fields panel and color.

Now it seems clear how to build the new FigurePanel class to hold these graphical elements. We will simply add the elements to an instance of FigurePanel, which stores them in a Vector and asks for each of them to be plotted in its paintComponent method.
public class FigurePanel extends JPanel {
    AffineTransform transform;
    int width = 0, height = 0;
    BoundingBox bbox;
    Vector plotables;
    public FigurePanel(double llx, double lly, double urx, double ury) {
        setBackground(Color.white);
        bbox = new BoundingBox(llx, lly, urx, ury);
        plotables = new Vector();
    }
    public void paintComponent(Graphics gfx) {
        super.paintComponent(gfx);
        Rectangle size = getBounds();
        Graphics2D g = (Graphics2D) gfx;
        if (size.width != width || size.height != height) {
            width = size.width; height = size.height;
            float ratioX = (float) (width / (bbox.urx - bbox.llx));
            float ratioY = (float) (-height / (bbox.ury - bbox.lly));
            transform = new AffineTransform();
            transform.scale(ratioX, ratioY);
            transform.translate(-bbox.llx, -bbox.ury);
        }
        for (int i = 0; i < plotables.size(); i++)
            ((Plotable) plotables.elementAt(i)).plot(g);
    }
    public BoundingBox getBoundingBox() {
        return bbox;
    }
    public AffineTransform getTransform() {
        return transform;
    }
    public void add(Plotable p) {
        plotables.addElement(p);
        p.setFigurePanel(this);
    }
}
Let’s define two more Plotables and use our classes to create an illustration. First, a set of Axes:

```java
public class Axes extends Plotable {
    public void plot(Graphics2D g) {
        BoundingBox bbox = panel.getBoundingBox();
        GeneralPath path = new GeneralPath();
        path.moveTo((float) bbox.llx, 0); path.lineTo((float) bbox.urx, 0);
        path.moveTo(0, (float) bbox.lly); path.lineTo(0, (float) bbox.ury);
        g.setPaint(color);
        AffineTransform transform = panel.getTransform();
        g.draw(transform.createTransformedShape(path));
    }
}
```

and then a GraphicalPoint:

```java
public class GraphicalPoint extends Plotable {
    public static final int CIRCLE = 0;
    public static final int SQUARE = 1;
    int style = CIRCLE;
    double size = 2;
    public double x, y;
    public GraphicalPoint(double xp, double yp) {
        x = xp; y = yp;
    }
    public void setSize(double s) { size = s; }
    public void setStyle(int s) { style = s; }
    public void plot(Graphics2D g) {
        Point2D.Double point = new Point2D.Double(x, y);
        toPixels(point);
        if (style == CIRCLE) shape =
            new Ellipse2D.Double(point.x - size, point.y - size,
                2*size, 2*size);
        else shape =
            new Rectangle2D.Double(point.x - size, point.y - size,
                2*size, 2*size);
        g.setPaint(color);
        g.fill(shape);
        g.setPaint(Color.black);
        g.draw(shape);
    }
}
```

This class demonstrates another reason why we wish to transform the shapes individually, rather than modifying Graphics2D’s internal transformation, since we also wish to work with pixels when plotting the point.
Here is a simple illustration using these features.

```java
public class SimpleFigure {
    public static void main(String[] args) {
        JFrame frame = new JFrame("SimpleFigure");
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        FigurePanel panel = new FigurePanel(-2, -3, 2, 3);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        panel.setPreferredSize(new Dimension(300, 300));
        Grid grid = new Grid();
        grid.setColor(Color.lightGray);
        panel.add(grid);
        Axes axes = new Axes();
        panel.add(axes);
        GraphicalPoint p1 = new GraphicalPoint(1, 2);
        p1.setColor(Color.red);
        p1.setStyle(GraphicalPoint.SQUARE);
        panel.add(p1);
        GraphicalPoint p2 = new GraphicalPoint(0, -2);
        p2.setColor(Color.green);
        p2.setSize(10);
        panel.add(p2);
        frame.pack();
        frame.show();
    }
}
```
Now that the Plotables are constructed as well as FigurePanel, the actual coding of the figure can be done at a reasonably high level. The coordinate system is automatically taken care of for us and the details of plotting are hidden as well. Also notice that the elements are drawn in the order in which they are added to the FigurePanel. This means that the grid appears to lie below the axes which appear to lie below the points.

Exercise 5: Construct a Plotable called Line that represents a line between two specified points. It should have a boolean instance field infinite that controls whether to draw a line segment or the portion of an infinite line seen in the BoundingBox and a method setInfinite to set this field. You may wish to add a method setStroke to allow the line to be drawn at a specified thickness or perhaps dashed.

Exercise 6: Construct a Plotable called Ellipse that draws ellipses. The input should be the center of the ellipse and the length of the major and minor semi-axes. You may also want to include a method setRotation that causes the ellipse to be rotated.

Exercise 7: Illustrate Proposition 1 of Book I of *The Elements* again using the new drawing library we have created.

Exercise 8: In the SimpleFigure shown above, you probably noticed that the grid was not displayed at the bottom and the right. This is because they are drawn outside of the JPanel by one pixel. Construct a class Insets, with public instance fields left, right, top and bottom, and add a method setInsets to FigurePanel to add margins into a FigurePanel.
3: Packaging it together

We have now built a number of classes that work together with the aim to facilitate the creation of mathematical figures.

Axes     FigurePanel     Plottable
BoundingBox GraphicalPoint SimpleFigure
DrawFrame  Grid

This would be a good time to bundle these classes together in a package. It’s quite simple, really.

We will call this package figure. To create it, we need to:

- create a directory called figure and place all of our files in it.
- add a line at the beginning of each class file that says
  ```java
  package figure;
  ```
  This line identifies the class defined as belonging to the package.
- set our CLASSPATH environment variable to point to the parent directory of figure.
- recompile all the classes, and finally
- use the command
  ```java
  java figure.SimpleFigure
  ```
  to run the application.

Also, if we are defining a class in another directory and want to use the classes in the figure package, we need to include the import statement `import figure.*;`

4: Interfaces

It remains for us to convert the CubicFunction class, the class responsible for drawing the graphs of functions, over to our new, more general scheme. There is something fundamentally different about this task.

For the classes we have looked at so far, we only needed to set some instance fields; for instance, for GraphicalPoint we need to specify the $x$ and $y$ coordinates of the point. However, to allow graphing general functions, we need to set the method `valueAt` that returns the $y$ value for a input $x$. Some other programming languages, such as C, allow programmers to pass methods as arguments to other methods. However, Java does not; we are only allowed to pass primitive data types or objects.

Java provides an elegant means for something similar through a feature of object-oriented programming called an interface. An interface is not so much a class as it is a template for a class. Here is the example we will use:

```java
public interface Function {
    public double valueAt(double x, double[] params);
}
```

As with public class definitions, this code should be in a file called `Function.java`. The only thing this definition contains is the signature of one method and the method itself is not defined. Notice that
the method is slightly different than what we used above: another argument, an array of doubles is given so that we might pass parameters into the function.

We now say that a method implements the Function interface if it has a method with the given signature. For example,

```java
public class Cubic implements Function {
    public double valueAt(double x, double[] params) {
        return x*x*x - x;
    }
}
```

Notice that we use the implements keyword to indicate that this class implements the Function interface.

Let’s now rename and modify our function graphing class to GraphicalFunction and allow more general functions. We will include as an instance field a class implementing the Function interface and an array of doubles that give the parameters. The values for these fields are passed to the GraphicalFunction object through its constructor.

```java
public class GraphicalFunction extends Plottable {
    Function function;
    double[] params;
    Stroke stroke;
    int steps = 50;
    public GraphicalFunction(Function f, double[] p) {
        function = f; params = p;
        stroke = new BasicStroke(1f);
    }
    public void setSteps(int s) { steps = s; }
    public void setStroke(Stroke s) { stroke = s; }
    public void plot(Graphics2D g) {
        BoundingBox bbox = panel.getBoundingBox();
        AffineTransform transform = panel.getTransform();
        double stepsize = (bbox.urx - bbox.llx)/steps;
        GeneralPath path = new GeneralPath();
        path.moveTo((float) bbox.llx,
                     (float) function.valueAt(bbox.llx, params));
        for (int i = 1; i <= steps; i++) {
            double x = bbox.llx + i*stepsize;
            path.lineTo((float) x, (float) function.valueAt(x, params));
        }
        g.setPaint(color);
        Stroke savedStroke = g.getStroke();
        g.setStroke(stroke);
        g.draw(transform.createTransformedShape(path));
        g.setStroke(savedStroke);
    }
}
```
This works since we can guarantee that if a class implements `Function`, then it must have a method called `valueAt` that returns a `double` when given a `double` and an array of doubles.

We have noted, you’ll recall, that a class can extend at most one class. However, it is possible for a given class to implement an arbitrary number of interfaces. This will be especially useful when we look at events in the next chapter.

The Java API is replete with interfaces. In fact, `Shape` is an interface and which several of our graphics primitives, such as `Line2D` and `Rectangle2D`, implement.

There are several ways to use this, a few of which are illustrated in the following example:

```java
public class FunctionFigure implements Function {
    public double valueAt(double x, double[] params) {
        return x*x*x + params[0]*x;
    }

    public static void main(String[] args) {
        JPEGDrawFrame frame = new JPEGDrawFrame("FunctionFigure");
        FigurePanel panel = new FigurePanel(-2, -3, 2, 3);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        panel.setPreferredSize(new Dimension(300, 300));
        Color[] colors = { new Color(0.3f, 0.3f, 1f),
                         new Color(1f, 0.3f, 0.3f),
                         new Color(0.3f, 1f, 0.3f) };
        Grid grid = new Grid();
        grid.setColor(Color.lightGray);
        panel.add(grid);
        Axes axes = new Axes();
        panel.add(axes);
        FunctionFigure function = new FunctionFigure();
        for (int i = -1; i < 2; i++) {
            GraphicalFunction f1 =
                new GraphicalFunction(function, new double[] {i-1});
            f1.setColor(colors[i]);
            panel.add(f1);
            GraphicalFunction f2 =
                new GraphicalFunction(new Function() {
                    public double valueAt(double x, double[] params) {
                        return 3*x*x - params[0];
                    }
                }, new double[] {i});
            f2.setStroke(new BasicStroke(1, BasicStroke.CAP_SQUARE,
                                         BasicStroke.JOIN_MITER, 10f,
                                         new float[] {4, 4}, 0));
            f2.setColor(colors[i]);
            panel.add(f2);
        }
        frame.pack();
        frame.show();
    }
}
```
This example shows two ways in which we might implement the Function interface and pass it to the constructor of a GraphicalFunction. First, we can simply define a new class, like FunctionFigure above, that contains the mathematical function we wish to pass to an instance of GraphicalFunction. The second way is to use a so-called anonymous class. This is, in essence, a definition of a class constructed on the fly. The syntax is illustrated in the constructor for GraphicalFunction f2 above.

Exercise 9: Remember that a parabola is the set of points that are equidistant from one fixed point (the focus) and a line (the directrix). Create an illustration like the one below.
**Exercise 10:** Add a class to the \texttt{figure} package that draws parametric curves.

**Exercise 11:** Draw the image of a family of concentric circles centered at the origin under the complex mapping $z \mapsto e^z$. 