This tutorial describes a Java package designed to assist in creating mathematical illustrations. My aims are to help users bypass some of the more technical aspects of the Java programming language so they might quickly begin building useful illustrations; create a uniform style for illustrations through a standardized collection of tools; provide high-level tools which model standard mathematical objects.

This tutorial assumes that the reader is comfortable with Java’s object-oriented nature and is familiar with subclasses and interfaces. It is not intended to teach you how to program in Java and it will not introduce you to every feature of the figure package. Please consult the figure API for more details.

We’ll see an example shortly, but first let me introduce you to a hierarchy of Java objects.

\[
\text{Plotables} \subset \text{Figures} \subset \text{FigureCanvas} \subset \text{Applet}
\]

The further you are to the left in this hierarchy, the more “mathematical” the objects are. For instance, Plotables are designed to model mathematical things such as axes and graphs of functions. As we move to the right, the objects become more like standard Java objects. The figure package is designed so that a user might write one class file (typically an Applet) which describes a mathematical situation through a collection of Plotables.

**Example 1: LineMover**

Let’s now look at an example to illustrate this hierarchy. Shown below is a sample applet demos.LineMover which is bundled with the figure package. The applet is designed so a user can move the points and line by clicking in one of the points and dragging.\(^1\)

\[\text{http://merganser.math.gvsu.edu/david/figs/figure.tgz.}\]
What is a Plotable? Well, something which can be plotted. The class Plotable is an abstract class which is subclassed by several objects in the figure package. Simply stated, everything you see in this diagram is a Plotable. For instance, there is a

**grid** given by an instance of Grid, a subclass of Plotable. It uses the default settings: the lines are parallel to the axes and one unit apart. It is described by the code:

```java
Grid grid = new Grid();
grid.setColor(new Color(0xcc, 0xcc, 0xcc));
```

**axes** given by an instance of Axes, a subclass of Plotable. Again, it uses the default settings to place the tick marks and labels. It is described by

```java
Axes axes = new Axes();
axes.setFont(new Font("Helvetica", Font.BOLD, 14));
```

**line** given by an instance of Line, a subclass of Plotable. It is specified, as we will see by declaring two points on the line.

```java
line = new Line(p1.x, p1.y, p2.x, p2.y);
line.setColor(Color.blue);
```

**points** given by two instances of FigurePoint, a subclass of Plotable.²

```java
p1 = new FigurePoint(0, 0, this);
p1.setSize(3);
p1.setFillColor(new Color(0xcc, 0x33, 0x66));

p2 = new FigurePoint(1, 1, this);
p2.setSize(3);
p2.setFillColor(new Color(0xcc, 0x33, 0x66));
```

Notice that Plotables have both mathematical and graphical information. For instance, the grid lines are drawn one unit apart in the coordinate system and the points appear at (0, 0) and (1, 1). In addition, they have graphical information, such as a color or font, which may be specified.

Let’s move to the next step of the hierarchy: Figure. The Plotables which we studied above are described relative to some coordinate system. An instance of Figure is simply a means of specifying a coordinate system and stating which Plotable elements are to be considered relative to that coordinate system.

In this example, we have only one instance of Figure since there is only one coordinate system. It is instantiated by the statement

```java
Figure f = new Figure(-4, -4, 4, 4);
```

²If there is a conflict with an existing Java object, the word “Figure” is prepended to the name. Thus a mathematical point is described by a FigurePoint so as not to conflict with java.awt.Point.
That is, to instantiate a Figure, we just specify a range for the horizontal and vertical coordinates. The format for doing this is:

```java
Figure f = new Figure(llx, lly, urx, ury);
```
where the lower-left corner of the Figure is the point \((llx, lly)\) and the upper-right corner is \((urx, ury)\).

![Coordinate system](image)

We tell the Figure which Plotables are to be considered relative to its coordinate system through its add methods.

```java
f.addToBackground(grid);
f.addToBackground(axes);

f.add(line);
f.add(p1);
f.add(p2);
```

This demonstrates that Figures have two drawing spaces: a background and foreground. Plotables in the background will rarely, if ever, be redrawn. It’s a good place to put elements which do not change (like our grid and axes) and we use the method `addToBackground(Plotable)` to do it.

By contrast, things in the foreground will be redrawn frequently so it holds the elements of the Figure which change, namely our points and line. Of course, everything may be put into the foreground at a (usually negligible) cost in performance. Things go into the foreground with the method `add(Plotable)`.

One more comment is important here. Remember that Java paint is opaque; this means that if two things overlap, the one drawn last will appear on top. This means that the order of drawing is important. It works like this: everything in the background is first drawn in the order in which it is added to the background. Then everything in the foreground is drawn in the order in which it is added to the foreground. There are methods in Figure to control this somewhat. For instance, a mark can be placed in the list of Plotables added to a Figure. Later Plotables can be added before or after the mark with the methods `addBeforeMark(Plotable)` and `addAfterMark(Plotable)`. One
may also specify the numerical position in the order of drawing using \texttt{addInPlace(Plotable, int)}.

So far we have instantiated all our \texttt{Plotables} and bound them together in a \texttt{Figure} which defines a relevant coordinate system. Somehow we need to control the way this \texttt{Figure} sits inside the \texttt{Applet}. Let’s take one more step to the right and discuss the \texttt{FigureCanvas} object.

\texttt{FigureCanvas} is a subclass of \texttt{java.awt.Canvas} designed to hold \texttt{Figures} inside an \texttt{Applet}. We’ll have more to say about this in a few minutes. For now, just notice the code

\begin{verbatim}
FigureCanvas fc = new FigureCanvas();
fc.add(f);
\end{verbatim}

instantiates a \texttt{FigureCanvas} and adds our \texttt{Figure} to it. We then add the \texttt{FigureCanvas} to our \texttt{Applet} in the usual way.

\begin{verbatim}
add("Center", fc);
\end{verbatim}

Let’s put everything together now and look at the whole code:

\begin{verbatim}
package demos;

import figure.*;
import java.awt.*;
import java.applet.*;

public class LineMover extends Applet implements Mover {
    Line line;
    FigurePoint p1, p2;

    public void init() {
        setBackground(Color.white);
        setLayout(new BorderLayout());

        FigureCanvas fc = new FigureCanvas();
        add("Center", fc);

        Figure f = new Figure(-4, -4, 4, 4);
        fc.add(f);

        Grid grid = new Grid();
        grid.setColor(new Color(0xcc, 0xcc, 0xcc));

        Axes axes = new Axes();
        axes.setColor(new Color(0x0c, 0x0c, 0x0c));
    }
}
\end{verbatim}

\footnote{The \texttt{Figure} package is written in Java 1.0.2 so that it might run in the widest possible environments. It is perfectly compatible with later versions of Java.}
axes.setFont(new Font("Helvetica", Font.BOLD, 14));

f.addToBackground(grid);
f.addToBackground(axes);

p1 = new FigurePoint(0, 0, this);
p1.setSize(3);
p1.setFillColor(new Color(0xcc, 0x33, 0x66));

p2 = new FigurePoint(1, 1, this);
p2.setSize(3);
p2.setFillColor(new Color(0xcc, 0x33, 0x66));

line = new Line(p1.x, p1.y, p2.x, p2.y);
line.setColor(Color.blue);

f.add(line);
f.add(p1);
f.add(p2);
}

public void moveElement(Moveable m, double x, double y) {
((FigurePoint) m).setPoint(x,y);
    line.setPoints(p1.x, p1.y, p2.x, p2.y);
}

You might notice that we have neglected one thing: how do we get the points to move when we click on them?
To understand this, it may be best to dig behind the scenes a little bit and discuss the three-step process by which adjustments to a Figure are handled by the figure package.

1. Mouse clicks, at least in Java 1.0.2, can be handled by Canvases. This means that when the mouse is clicked in a FigureCanvas, the FigureCanvas notices. It then determines whether the mouse click is an attempt to move any of the elements in a Figure in the FigureCanvas.

2. Elements which may be moved are subclasses of Moveable (itself a subclass of Plotable). An instance of FigureCanvas contains a list of all Moveables contained in any Figure inside the canvas. When the mouse is depressed, the FigureCanvas searches through this list to determine if any of the Moveables are being moved. If it determines that the mouse click is trying to move a Moveable, it notifies the Moveable of the attempt.

3. The way in which a Moveable moves may vary from instance to instance. For instance, one point may be constrained to move only horizontally, another only
vertically and a third may have no constraint at all. Therefore, a Moveable needs some instructions to follow when an attempt is made to move it. The instructions are provided by an instance of Mover, an interface with one method:

```
public void moveElement(Moveable m, double x, double y);
```

Every Moveable has a Mover associated to it; most likely, the Mover will be the Applet we are designing.

When notified of an attempt to move it, a Moveable calls the moveElement method of its Mover passing itself as the Moveable and the x and y coordinates to which the user wants to move the point.

That sounds complicated, but in practice it is quite easy. For instance, in our example, we need to do only a few things. First, we need to tell our FigurePoints that they are capable of being moved⁴ and we need to assign them their Movers. These two tasks are accomplished through the constructors

```
p1 = new FigurePoint(0, 0, this);
p2 = new FigurePoint(1, 1, this);
```

The first two arguments specify the initial point for the FigurePoints and the third argument is the associated Mover, in this case the Applet we are writing.

Finally, we need to make our Applet implement Mover by defining the method

```
public void moveElement(Moveable m, double x, double y);
```

Here, we simply set the point and adjust the line. It’s that simple. Repainting the FigureCanvas is automatically taken care of for us.

As this is the standard mechanism by which adjustments are made to a Figure by the user, this discussion may warrant your attention. You will also see a few more examples below to help make it clear.

By default, the motion of FigurePoints is constrained so that they cannot leave the Figure in which they belong. This may be turned on and off with the FigurePoint’s method

```
public void setConstrained(boolean)
```

Example 2: Integral

In our second example, we’ll see a different type of adjustable object though the mechanism by which it works is quite similar to what we saw above. Shown below is a diagram representing the Applet called demos.Integral.

---

⁴It is conceivable that you might wish to add a static FigurePoint into a Figure. You can do this through the constructor

```
p = new FigurePoint(a, b);
```

That is, you simply tell the FigurePoint where you would like it to be.
Notice that there is a slider in the upper left corner of the diagram which can be moved along an axis thus changing the number of rectangles used in the approximation of the definite integral. We’ll discuss this shortly. At this time, however, let’s meet a few new Plotables. The

**graph** is an instance of `FigureFunction`, a subclass of `Plotable`, designed to display the graph of a function. The constructor looks like

```java
FigureFunction function = new FigureFunction(this, null);
```

If a `FigureFunction` is to display the graph of a function, it needs to know how to evaluate the function. This information is passed to the constructor: `this` refers to an implementation of the interface `Evaluator` and `null` is a String which identifies the particular function we want from the `Evaluator`.

This interface `Evaluator` has one method with the signature

```java
public double valueAt(double x, String key);
```

What is the role of the String which is passed to `valueAt`? It is possible that the same `Evaluator` will be asked to perform the evaluation of many different functions. The `String key` is used as a tag to designate which function is used. Our example contains only one function so we simply set the tag to `null`; in other words, we don’t use it. In our example, we have

```java
public double valueAt(double x, String s) {
    return 1.8*Math.exp(-0.2*x)*Math.sin(x);
}
```

To illustrate how the tags work, suppose we want to graph both the sine and cosine functions. We could do it like this:
FigureFunction sin = new FigureFunction(this, "sin");
FigureFunction cos = new FigureFunction(this, "cos");

where the implementation of the Evaluator looks like:

```java
public double valueAt(double x, String s) {
    if (s.equals("sin")) return Math.sin(x);
    if (s.equals("cos")) return Math.cos(x);
    return 0;
}
```

By default, the domain of a FigureFunction is set to the horizontal range of the Figure in which it resides. This may be modified by the method

```java
setDomain(double x0, double x1)
```

rectangles are an instance of RiemannSum, a subclass of Plotable, consisting of a number of rectangles whose area represents a Riemann Sum. It is built like this:

```java
double initialRectangles = 20;
riemannSum = new RiemannSum(this, null);
riemannSum.setFillColors(new Color(0x66, 0x33, 0xcc),
    new Color(0xcc, 0x33, 0x66));
riemannSum.setRectangles((int) initialRectangles);
riemannSum.setStyle(RiemannSum.UPPER);
```

Here we specify that there will initially be 20 rectangles and then construct an instance of RiemannSum using a particular function, in this case, the same function displayed by function. We set two colors, one for rectangles with positive height, the other for rectangles with negative height and initialize the number of rectangles. There are several “styles” to use when building RiemannSums; here we have specified that the height of a rectangle is given by the largest value for the function over that sub-interval.

slider is an instance of Slider, a subclass of Plotable, designed so that the user may vary a parameter within a given range. It is built in the following way:

```java
Slider slider = new Slider(-3.5, -0.5, 3.7, 10, 50, this);
slider.setOrientation(Slider.HORIZONTAL);
slider.setStyle(FigurePoint.DIAMOND);
slider.setSize(3);
slider.setSpace(10);
slider.setFillColor(new Color(0x33, 0x33, 0x33));
slider.setLabel("");
slider.setValue(initialRectangles);
```
The constructor looks complicated so let’s break its arguments down. First, our slider is going to be oriented horizontally (the only alternative is to orient the slider vertically) so the first two arguments \((-3.5, -0.5)\) describe the Slider’s horizontal range in its containing Figure.

The third coordinate, \(3.7\), specifies the vertical coordinate along which the Slider resides. If the slider is to be oriented vertically, these arguments give the vertical range and the horizontal coordinate along which the slider moves.

The fourth and fifth coordinate specifies the allowed range for the parameter which is varied by the slider.

Finally, the sixth argument, \texttt{this}, specifies an implementation of the interface \texttt{SliderListener}, an object to be notified when the slider is moved. To implement this interface, we need to define the method:

```java
public void sliderMoved(Slider s, double val)
```

The way in which this works is similar to how the \texttt{FigurePoint}s in the previous example were adjusted. When a Slider is moved, it calls the \texttt{sliderMoved} method of the \texttt{SliderListener} associated to it and supplies itself as the Slider and the value \texttt{val} to which it has been moved.

In our example, the Slider is describing how many rectangles we use in the \texttt{RiemannSum} so the implementation looks like this:

```java
public void sliderMoved(Slider s, double val) {
    riemannSum.setRectangles((int) val);
}
```

Notice that the Slider has many attributes which can be set. In fact, you will notice that these attributes are similar to attributes for \texttt{FigurePoint}s and \texttt{Axes}. There is a good reason for this: the Slider is a subclass of \texttt{Figure} with two \texttt{Plotable}s, a \texttt{FigurePoint} and an \texttt{Axes}, added to it.

You may be thinking: how can Slider be both a subclass of \texttt{Plotable} and \texttt{Figure}. The answer is that \texttt{Figure} is itself a subclass of \texttt{Plotable}. Earlier, I said that everything you see is a \texttt{Plotable}. The converse, however, is false: you cannot see every \texttt{Plotable} because \texttt{Figures} are \texttt{Plotables} with no graphical information. The implication of this, however, is that \texttt{Figures} can be nested inside one another. (This is, in fact, what happens when a Slider is added to a Figure.) To add \texttt{Figure} inside to \texttt{Figure} outside, call

```java
outside.add(inside, llx, lly, urx, ury)
```

which specifies that \texttt{inside} lies inside \texttt{outside} within the region delimited by \((llx, lly)\) and \((urx, ury)\).

Here is the code for \texttt{Integral}:
package demos;
import java.awt.*;
import java.applet.*;
import figure.*;

public class Integral extends Applet implements SliderListener, Evaluator {
    RiemannSum riemannSum;

    public void init() {
        setBackground(Color.white);
        setLayout(new BorderLayout());

        FigureCanvas fc = new FigureCanvas();
        add("Center", fc);

        Figure f = new Figure(-4, -4, 4, 4);
        fc.add(f);

        double initialRectangles = 20;
        riemannSum = new RiemannSum(this, null);
        riemannSum.setFillColors(new Color(0x66, 0x33, 0xcc),
                                   new Color(0xcc, 0x33, 0x66));
        riemannSum.setRectangles((int) initialRectangles);
        riemannSum.setStyle(RiemannSum.UPPER);
        f.add(riemannSum);

        Axes axes = new Axes();
        f.add(axes);

        Slider slider = new Slider(-3.5, -0.5, 3.7, 10, 50, this);
        slider.setOrientation(Slider.HORIZONTAL);
        slider.setStyle(FigurePoint.DIAMOND);
        slider.setSize(3);
        slider.setSpace(10);
        slider.setFillColor(new Color(0x33, 0xcc, 0x33));
        slider.setLabel("");
        slider.setValue(initialRectangles);
        f.add(slider);
    }

    public double valueAt(double x, String s) {
        return 1.8*Math.exp(-0.2*x)*Math.sin(x);
    }
}
public void sliderMoved(Slider s, double val) {
    riemannSum.setRectangles((int) val);
}

Example 3: GridDemo

Our third example is similar in spirit to the first two but illustrates a few new features. It is shown below:

At the top is a matrix whose entries can be varied. The figure in the lower left is static while the figure in the lower right is the image of the left figure under the linear transformation represented by the matrix.

One difference should be immediately apparent: there is more than one Figure here. In the previous two examples, we have simply added Figures to a FigureCanvas with the method

    add(Figure)

This should be thought of as the default add method; it causes the Figure to fill out the entire FigureCanvas. However, there is another version of the add method:

    add(Figure, double llx, double lly, double urx, double ury)

which adds the Figure into the FigureCanvas so that it only takes up the rectangular region represented by \((llx, lly)\) and \((urx, ury)\) relative to the FigureCanvas’ coordinate system. We have not yet had occasion to refer to this coordinate system; every FigureCanvas has a coordinate system where \((llx, lly) = (0, 0)\) and \((urx, ury) = (1, 1)\).

Here is how the code looks:

    FigureCanvas fc = new FigureCanvas();
    Figure lowerLeft = new Figure(-4.2, -4.2, 4.2, 4.2);
Figure lowerRight = new Figure(-4.2, -4.2, 4.2, 4.2);

fc.add(lowerLeft, 0, 0, 0.47, 0.8);
fc.add(lowerRight, 0.53, 0, 1, 0.8);

The sliders representing the matrix are included by instantiating a Matrix object and adding it to the FigureCanvas. There are a few things to be said about the Matrix object.

First, notice that it is a subclass of Figure. This is not too surprising because it seems to be composed of four sliders and a few lines. This means that it may be added to the FigureCanvas in the same way that Figures are:

    matrix = new Matrix(this);
    fc.add(matrix, 0, 0.8, 1, 1);

By now, you should have a good idea what this represents in the constructor. When a Matrix is adjusted, it’s pretty pleased with itself and wants to tell someone about it. That someone is an implementation of the interface MatrixListener, an interface with one method whose signature is:

    public void matrixAdjusted(Matrix m);

In this example, the object which implements MatrixListener is the Applet itself. When it is notified that the Matrix has changed, the MatrixListener takes whatever action is appropriate; in this example, it adjusts the grid and polygon on the right.

Notice two more things about the Figures in this example. The code includes the lines:

    lowerLeft.setClipped(true);
    lowerLeft.setAspectRatio(1);

    lowerRight.setClipped(true);
    lowerRight.setAspectRatio(1);

The method call setClipped(true) causes the clipping rectangle to be set to the rectangular region of the Figure when the Figure is drawn. The result is that we only see what lies inside the Figure’s rectangular region. This is necessary because, otherwise, some of the Grid is drawn outside of the Figure in which it is contained.\(^5\)

Finally, the method call setAspectRatio(1) causes the scale for the horizontal and vertical axes to be the same. This is important if you want a square to look like a square or a circle to look like a circle. More generally, the aspect ratio may be set to any nonzero

\(^5\)Grids are drawn efficiently but without careful consideration for the boundary of the containing Figure.
number in which case it represents the ratio of the scale along the vertical axis to that along the horizontal axis.\footnote{You may wonder what would happen if it were set to a negative number. The answer is that the orientation of one of the axes would be reversed. The more general comment is worth noting too: when building a rectangular region using \((llx, lly)\) and \((urx, ury)\), nothing says that the first point is to the left and below the second. You should simply think that the region being described is the smallest rectangular region containing the two points. In this way, it is possible, for instance, to have quantities along the vertical axis increasing as you move down the screen.}

Here is the complete code:

```java
package demos;

import java.awt.*;
import java.applet.*;
import figure.*;

public class GridDemo extends Applet implements MatrixListener {
    Grid rightGrid;
    Matrix matrix;
    FigurePolygon fp;
    double[] x, y;
    Color gray = new Color(0x99, 0x99, 0x99);

    public void init() {
        setLayout(new BorderLayout());
        setBackground(Color.white);
        Figure lowerLeft = new Figure(-4.2, -4.2, 4.2, 4.2);
        lowerLeft.setClipped(true);
        lowerLeft.setAspectRatio(1);
        Grid leftGrid = new Grid();
        leftGrid.setColor(gray);
        Axes leftAxes = new Axes();
        FigureRectangle fr = new FigureRectangle(0, 0, 1, 1);
        fr.setColor(Color.blue);
        fr.setFillColor(gray);
        lowerLeft.add(leftGrid);
        lowerLeft.add(leftAxes);
        lowerLeft.add(fr);
        Figure lowerRight = new Figure(-4.2, -4.2, 4.2, 4.2);
        lowerRight.setClipped(true);
        lowerRight.setAspectRatio(1);
        rightGrid = new Grid();
```
rightGrid.setColor(gray);
Axes rightAxes = new Axes();

x = new double[4]; y = new double[4];
x[0] = 0; y[0] = 0;
x[1] = 1; y[1] = 0;
x[3] = 0; y[3] = 1;
fp = new FigurePolygon(x, y);
fp.setColor(Color.blue);
fp.setFillColor(new Color(0x99, 0x99, 0x99));
lowerRight.add(rightGrid);
lowerRight.add(fp);
lowerRight.add(rightAxes);

matrix = new Matrix(this);

FigureCanvas fc = new FigureCanvas();

fc.add(lowerLeft, 0, 0, 0.47, 0.8);
fcc.add(lowerRight, 0.53, 0, 1, 0.8);
fcc.add(matrix, 0, 0.8, 1, 1);
add("Center", fc);

}

private void matrixAdjusted(Matrix m) {
    double[] a = matrix.transform(1, 0);
double[] b = matrix.transform(0, 1);
rightGrid.setBasis(a[0], a[1], b[0], b[1]);
x[1] = a[0]; y[1] = a[1];
x[3] = b[0]; y[3] = b[1];
    }

Example 4: Feigenbaum

Our final example demonstrates the Feigenbaum diagram arising in discrete dynamical systems.
When you first look at this, you may think there are only two Figures, but in fact there are three. You see a quadratic function plotted in the leftmost Figure and a grid in the rightmost Figure. In addition, there is the cobweb diagram which connects features in the two Figures. Of course, nothing prohibits a Plotable from drawing a portion of itself outside of a containing Figure (unless clipping is turned on with the setClipped(boolean) method as seen earlier). The problem here is that the cobweb needs access to the coordinate systems from the two Figures. To accomplish this, we will add a third Figure and demonstrate how the coordinates in one Figure may be transformed into coordinates in another Figure lying in the same FigureCanvas.

First, we instantiate our Figures and add them to the FigureCanvas.

```java
big = new Figure(0, 0, 1, 1);
fcc.add(big);

left = new Figure(-0.5, -0.5, 1.5, 1.5);
right = new Figure(-0.5, -0.5, 5, 1.5);
fcc.add(left, 0, 0, 0.47, 1);
fcc.add(right, 0.53, 0, 1, 1);
```

The Figures called `left` and `right` are the ones you first notice when looking at the diagram. The Figure called `big` is designed to cover the entire FigureCanvas. (`big` is added first so that the cobweb is underneath the FigurePoint in `right`.) For this reason, it is added to the FigureCanvas without specifying a bounding box. Also, the particular coordinate system (0, 0, 1, 1) chosen for `big` is irrelevant since we will not be using it explicitly.

We transform the coordinates of `right` to those of `big` through the methods:

```java
public double rightCoordinatesToBigX(double x) {
    return big.toFigX(right.toPixX(x));
}
```
public double rightCoordinatesToBigY(double y) {
    return big.toFigY(right.toPixY(y));
}

Notice that Figure and FigureCanvas implement an interface called coordinatization which gives methods for converting coordinates to pixels and pixels to coordinates. In particular, right.toPixX(double) transforms a horizontal coordinate into a horizontal pixel location in the FigureCanvas. The method big.toFigX(int) transforms a horizontal pixel location into a horizontal coordinate in big’s coordinate system. Composing these two methods transforms horizontal coordinates in right into those in big.

One more comment is in order here: coordinate transformations are not typically set correctly until the first time the Figures are painted. The problem is that we want the cobweb to appear correctly the first time big is painted. For this reason, we include the following snippet at the end of the Applet’s init method:

    validate();
    fc.initialSetUp();
    setUpPolygon();

This has the effect first of laying out the FigureCanvas in the Applet; in particular, the FigureCanvas is told its dimensions. Next, we ask the FigureCanvas to set up its coordinate system and then we build the polygon.

Two other features of this example should be mentioned. First, there is a piece of text $y = r(x - x^2)$ appearing in left; this is an instance of Text, another subclass of Plotable. Text has some ability to format the text it is displaying. For instance, subscripts and superscripts are allowed (though they may not be nested) by using \_ and ^ . A single character following one of these symbols will be appropriately raised or lowered. For subscripts or superscripts of more than one character, simply enclose the desired text in curly braces. (This is modelled on \TeX ’s convention.)

Finally, the Applet exhibits different behavior when the mouse is clicked. If the Ctrl key is held down, the FigurePoint in right is horizontally constrained. Otherwise, it is vertically constrained. When asked to move, a Moveable retains the Event which makes the request. This can be retrieved with getEvent(). Here is the relevant code:

    public void moveElement(Moveable m, double x, double y) {
        if (m.getEvent().controlDown()) {
            r = x;
            point.setPoint(x, point.y);
        } else {
            point.setPoint(point.x, y);
        }
        setUpPolygon();
    }

Here is the complete code for Feigenbaum:

    package demos;
import java.awt.*;
import java.applet.*;
import figure.*;

/* This demonstration is designed to show a couple of things:
First, a Mover can access the Event which caused the moveElement
method to be called. This can be used to determine additional
information such as whether the Control key is being held.

Secondly, Figures are able to communicate their coordinates to
one another in useful ways.

*/

public class Feigenbaum extends Applet implements Mover, Evaluator {
    double r = 1;
    FigurePoint point;
    FigurePolygon cobweb;
    double[] xPoints, yPoints;
    int nPoints = 51;
    FigureCanvas fc;
    Figure big, left, right;
    boolean initialized = false;
    Thread mainThread;

    public void init() {
        setBackground(Color.white);
        setLayout(new BorderLayout());

        fc = new FigureCanvas();
        add("Center", fc);

        big = new Figure(0, 0, 1, 1); // A figure to cover everything
        fc.add(big);

        xPoints = new double[nPoints];
        yPoints = new double[nPoints];
        cobweb = new FigurePolygon(xPoints, yPoints);
        cobweb.setColor(new Color(0x99, 0x00, 0x99));
        big.add(cobweb);

        left = new Figure(-0.5, -0.5, 1.5, 1.5);
        right = new Figure(-0.5, -0.5, 5, 1.5);
        fc.add(left, 0, 0, 0.47, 1);
fc.add(right, 0.53, 0, 1, 1);

Axes leftAxes = new Axes();
leftAxes.setYLabel(""");
leftAxes.setXSpace(0.5);
left.addToBackground(leftAxes);

Text yLabel = new Text("y = r(x - x^2)", 0.05, 1.4);
yLabel.setAlignment(Text.WEST);
left.addToBackground(yLabel);

Axes rightAxes = new Axes();
rightAxes.setXLabel("r");
rightAxes.setYLabel("x");

FigureFunction parabola = new FigureFunction(this, null);
parabola.setColor(new Color(0x66, 0x33, 0xff));
left.add(parabola);

Line line = new Line(-0.5, -0.5, 1.5, 1.5);
left.addToBackground(line);

Grid grid = new Grid();
grid.setSpacing(0.5, 0.25);
grid.setColor(new Color(0x99, 0x99, 0x99));
right.addToBackground(grid);
right.addToBackground(rightAxes);

point = new FigurePoint(r, 0.25, this);
point.setFillColor(new Color(0xff, 0x33, 0x66));
point.setSize(3);
point.setStyle(FigurePoint.DIAMOND);
right.add(point);

validate();
fc.initialSetUp();
setUpPolygon();
}

public double valueAt(double x, String s) {
    return r*x*(1-x);
}

public void moveElement(Moveable m, double x, double y) {
    if (m.getEvent().controlDown()) {

```java
    r = x;
    point.setPoint(x, point.y);
} else {
    point.setPoint(point.x, y);
}
setUpPolygon();

public void setUpPolygon() {
    xPoints[0] = rightCoordinatesToBigX(point.x);
    yPoints[0] = rightCoordinatesToBigY(point.y);
    xPoints[1] = leftCoordinatesToBigX(point.y);
    yPoints[1] = leftCoordinatesToBigY(point.y);
    double leftX = point.y; double leftY = point.y;
    for (int i = 2; i < nPoints - 1; i += 2) {
        xPoints[i] = xPoints[i-1];
        leftY = valueAt(leftX, null);
        yPoints[i] = leftCoordinatesToBigY(leftY);
        leftX = leftY;
        xPoints[i+1] = leftCoordinatesToBigX(leftX);
        yPoints[i+1] = yPoints[i];
    }
    xPoints[nPoints - 1] = xPoints[nPoints - 2];
    yPoints[nPoints - 1] = leftCoordinatesToBigY(valueAt(leftX, null));
}

// These methods convert coordinates in one Figure into big's coords

public double rightCoordinatesToBigX(double x) {
    return big.toFigX(right.toPixX(x));
}

public double rightCoordinatesToBigY(double y) {
    return big.toFigY(right.toPixY(y));
}

public double leftCoordinatesToBigX(double x) {
    return big.toFigX(left.toPixX(x));
}

public double leftCoordinatesToBigY(double y) {
    return big.toFigY(left.toPixY(y));
}
```