

MathToolKit Tutorial

Joe Yanik and Chuck Pheatt

Division of Mathematics and Computer Science

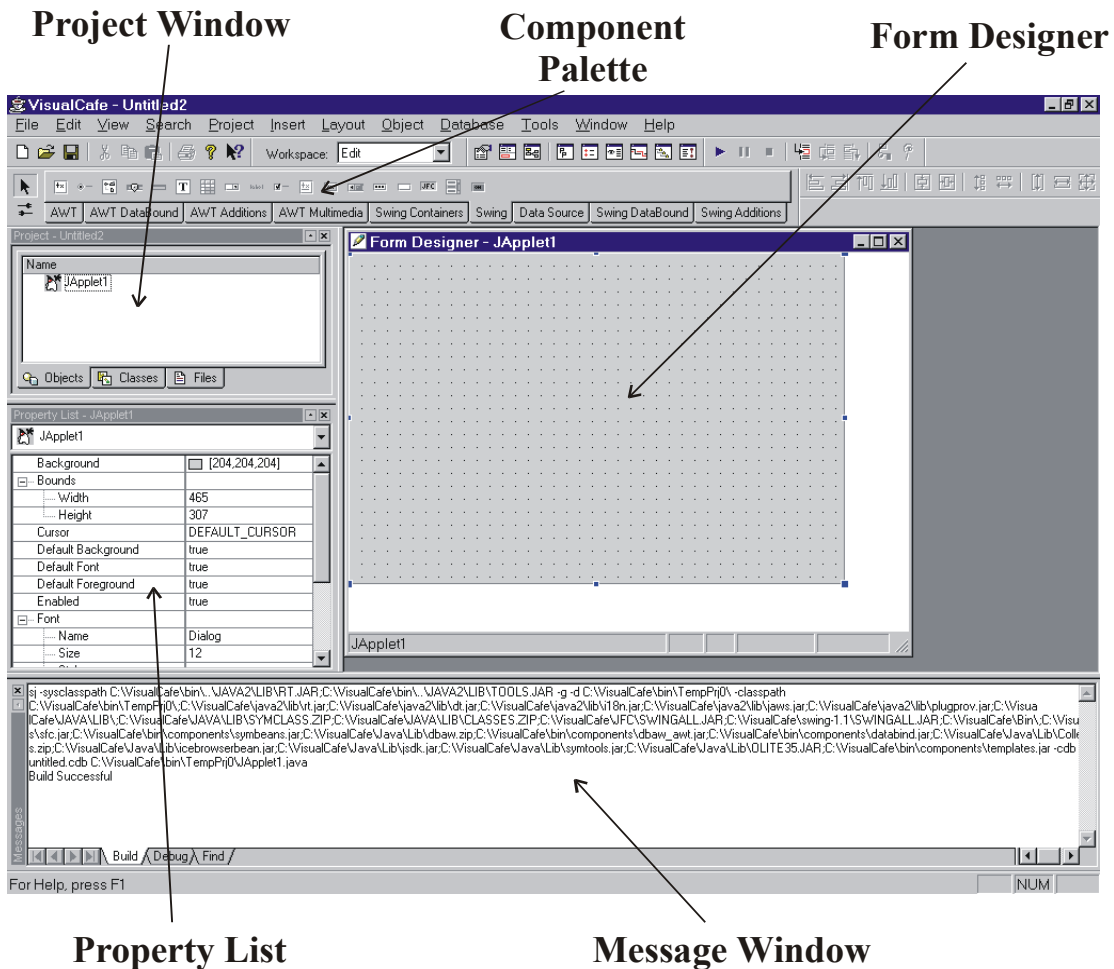
Emporia State University

Emporia, KS 66801

(yanikjoe@emporia.edu or pheattch@emporia.edu)

The main part of this tutorial is an overview of the main components in the *MathToolKit*. In addition there are two sections at the end that provide an introduction to VisualNumerics Java Numerical Library and to the java.math class. Although it does not assume any previous knowledges of Java Programming the tutorial does not attempt to teach Java. For an introduction to Java Programming *and* to the MathToolKit you should start with Tutorial01 and proceed through to Tutorial18. The tutorial assumes that you are using Visual Café and that you have the MathToolKit set up in the Visual Café environment. (For directions on setting up the MathToolKit, see the document *ToolKit Setup.*)

Begin by opening up Visual Café. Your desktop should look something like this:



The components can be arranged in different places on the screen, but be sure that you can identify the *Project Window*, the *Component Palette*, the *Form Designer*, the *Property List*, and the *Message Window*. We will be referring to these terms in the tutorials. If you don't see them, here is what you should do:

1. *If you don't see the Project Window:* Go to the **File** menu and select **New Project...**. From the resulting dialog box select an appropriate Project template (probably *JFC Applet*).
2. *If you don't see the Form Designer:* In the Project Window select the objects tab (it is probably already selected) and double-click on the icon for the Applet (it is probably called something like "JApplet1").
3. *If you don't see the Component Palette:* Right-click at the top of the Visual Café window and check the Component Palette option.
4. *If you don't see the Property List:* Go to the **View** menu and select **Property List**.
5. *If you don't see the Message Window:* Go to the **View** menu and select **Messages**.

MathGrapher, SymbolicFunction and ParametricCurve

In this part of the tutorial, we will explore the MathGrapher, SymbolicFunction, and SymbolicParametricCurve components. The MathGrapher will graph a function on a coordinate system while SymbolicFunction and SymbolicParametricCurve provide convenient ways to define a function or a parametric curve with a formula.

1. If you haven't done so already, start a new project by choosing **New Project...** from the **File** menu. (Make sure that you choose **New Project...** and not **New File**.)
2. In the resulting dialog box choose **JFCApplet** as the template for your project.
3. Make sure that the Project Window is active (click on it if necessary), then choose **Save...** from the **File** menu.
4. Visual Café is going to create a lot of files so it is a good idea to create a separate folder to contain them in. Do this from the Dialog box then give your project a name and save it into that folder
5. Select the Math tab on the component palette and find the SymbolicFunction icon. Click on it and drag it onto the Form Designer. This will place a copy of the icon there. (Since this is an “invisible” component it doesn't matter where you place it.)
6. Make sure that the SymbolicFunction icon is selected on the Form Designer and look at the Property List. It should contain three properties: “Formula”, “Name”, and “Title”. We will modify the Formula and the Name property..
7. Change the Name property to “f”.(without the quotes) and for the Formula property type in the formula for your favorite function (such as $\sin(x)$). [**Warning:** You must use x as your variable.]
8. Let's also create a Parametric Curve. Find the SymbolicParametricCurve icon on the Math palette, click on it, and drag it to the Form Designer.
9. With the SymbolicParametricCurve selected on the Form Designer look at the Property List. This time there are 7 properties: “Name”, “TDelta”, “TMax”, “TMin”, “Title”, “XFormula”, and “YFormula”. You can probably guess what most of these refer to. (Title is the title that will appear on the graph when the curve is active.)
10. Let's define an ellipse. Change the Name property to “ellipse” (without the quotes). (The names here aren't significant. You just need to be able to recognize them later on.) Define TDelta to be .05, TMax to be 6.3, TMin to be 0, XFormula to be $2*\cos(t)$ and YFormula to be $3*\sin(t)$. [**Warning:** You must use t as your variable.]
11. Now we want to graph our curves. For that we need a MathGrapher object. Find the icon on the Math Palette and drag it over to place a copy on the Form Designer. If you like, you can resize it after it is there.
12. With your MathGrapher object selected look at the Property List. You should see quite a few properties there. You can modify any of them. You won't notice the effect of some of the modifications until you execute the Applet, but others will have an immediate effect. Try modifying the following properties and observe the result:
 - AxesColor
 - GridColor
 - GridLines
 - Title
 - XLabel and YLabel
 - XMin, XMax, YMin, YMax

13. Now we want to add the graphs of our functions to the MathGrapher. We can do that from the Property List. Modify the F property to be f and the G property to be ellipse. Their graphs should appear on the coordinate system.
14. As you are doing all of this Visual Café is writing the appropriate code. To view the code you can right-click on the Form Designer and choose **Edit Source** from the resulting menu. See if you can find the code that corresponds to the changes that you have made.

Warning: It was important that we added the functions to the Form Designer *before* we added the MathGrapher. If we had put the MathGrapher on first, the code that Visual Café wrote would have put the instruction to add the function to the graph before the instruction to modify its formula. The result is that, at execution time, the graph of “ $y=0$ ” would be displayed no matter what your formula. (You can easily get around this by adding a call to the updateGraph method of the MathGrapher object at the end of the init method. We will investigate this more fully in the next section.)

15. It may be time to execute your Applet. From the **Project** menu choose **Execute**. Try out the trace feature by clicking on the graph (to make sure that the graph has the “focus”) then use the arrow keys to trace either curve. (If the trace feature doesn’t work you may have inadvertently set the TraceEnabled feature to false in the property list.)
16. When you have had about as much fun as you can with the trace feature shut down your applet by closing the applet window.
17. Now we can experiment with some other properties that will show up at execution time. Experiment with the various options of the ZoomMode property and its companion the ZoomFactor property. You can probably guess what each one does. (If you like, you can also shut off the Trace feature by setting TraceEnabled to false.)

Notes:

- As you experiment with zooming, you can always return to the original graph, by holding down the shift key while you click on the graph.
- When the trace feature is enabled, if you want to go to a specific x -value, you can hit the *Enter* key to bring up a dialog box that will allow you to enter in any value. (You can even enter in arithmetic expression or expressions involving Pi or e.)
- You can graph more than 2 curves, but not with the Property List. Just add a call to the addGraph method. (There is also a removeGraph method.) You can graph up to 20 at once.

Making Your Own Function

Sometimes you want to design a function that can't be described by a simple formula. You can do that by creating a subclass of the MathFunction class and overriding the functionValue method. (If you don't know what this means, that's OK. We'll walk through it below.) As an example of this we are going to define the function

$$f(x) = \begin{cases} x + 2 & \text{if } x \leq -1 \\ x^2 & \text{if } -1 < x \leq 1 \\ 1 & \text{if } x > 1 \end{cases}$$

1. If you still have the previous project open, leave it open and we can work with it. If not, create a new project, and put a MathGrapher on the Form Designer. Alter the name property of the MathGrapher so that it is "graph" (without the quotes).
2. From the **File** menu select **New File**. We are going to create a new class. Let's begin by typing in the skeleton of the class. Type in the following precisely as it appears below. (**Warning:** Java is case-sensitive, so match the capitalization.)

```
import EDU.emporiamathtools.*;

public class MyFunction extends MathFunction
{
    public double functionValue(double x)
    {
    }
}
```

What the words mean (You can skip this if you want)

- The first line imports the mathtools package, because that is where the MathFunction class resides.
- The first line announces that the name of the class is MyFunction and that it should be a subclass of MathFunction. This means that it will have at its disposal all of the data and methods of MathFunction. (Included among these are the ones that are necessary to graph a MathFunction object.)
- The opening and closing braces define the beginning and the end of the class.
- The other line of text announces that functionValue is a method that will accept as a x as a parameter. x is of type *double* which means that it will be a double-precision floating point number. This method, when called should return a number, also of type double. (As its name suggests, the number that is returned should be the function value of the function at x , or, in other words $f(x)$.)
- The inner opening and closing braces define the beginning and the end of the functionValue method. Here is where we need to put the code to define the function.

3. Make sure that the window that contains your source code for MyFunction is selected and select **Save As...** from the **File** menu. Save the file but it is *very important* that

(1) You save it into the same folder as your other project files and (2) you name it MyFunction.

4. At this point your MyFunction class should appear in the Project Window. It still doesn't do anything, though, so let's add the necessary code to define it. Insert the necessary lines so that the full class looks like this:

```
import EDU.emporia.mathtools.*;

public class MyFunction extends MathFunction
{
    public double functionValue(double x)
    {
        if(x <= -1)
            return x+2;
        else if (x <= 1)
            return x*x;
        else
            return 1;
    }
}
```

5. After typing in the above choose **Compile MathFunction.java** from the **Project** menu just to make sure that you don't have any typos.
6. Now we want to add a MyFunction object to our Applet. We can do this by simply selecting the MyFunction icon from the Project Window and dragging it onto the Form Designer. Do that now. Select the icon on the Form designer and edit the name on the property list so that it is "h" (without the quotes).
7. Now we are going to have to add a line of code to the Applet. Open up the source code for the Applet by right clicking on the Form Designer and selecting **Edit Source**. There will be a lot of code that Visual Café has written. Find the line that says

```
public void init()
```

This is the beginning of the init method. Below it is the opening curly brace of the init method. We want to insert our line at the end of this method so we need to find the closing curly brace. One way of doing this is to select the opening curly brace and press *control-]*. (**Warning:** The stuff that Visual Café writes is enclosed between the symbols `//{{` and `//}}`. Don't tamper with that and don't put anything in there yourself!) Just before the closing curly brace insert the following line:

```
graph.addGraph(h) ;
```

(**Warning:** This assumes that you have named your MathGrapher object *graph* and your MyFunction object *h*, as directed above.)

8. Execute the Applet to see the graph of your function.

PLFunction and New Functions from Old

We have seen two ways of creating a function. Any function created in this way will be a `MathFunction` object. `MathFunction` objects have access to methods that allow you to combine them in a number of ways to get new `MathFunction` objects. For example, you can form the sum, product, quotient, or composition of two `MathFunction` objects. You can add a constant to a `MathFunction` object or multiply one by a constant. In this part of the tutorial we will demonstrate some of these capabilities with some `PLFunction` objects. The `PLFunction` bean creates a randomly generated piecewise linear function that can be useful for generating examples. `PLFunction` objects are also `MathFunction` objects.

1. Close down any projects that you have open and start a new project. (Again, choose **New Project..**, from the **File** menu and take the **JFCApplet** template.) Save the project into a new folder.
2. Since we are going to put three `MathGraphers` in our Applet, resize the Form Designer so that it is large enough to hold them.
3. Using the `MathGrapher` icon from the Math palette put three different `MathGraphers` onto the Form Designer, positioning two of them next to each other at the top of the Form Designer and one at the bottom in the middle. By editing the Name in the Property List call the top two `fGraph` and `gGraph`, respectively, and call the bottom one `hGraph`.
4. Find the `RandomPLFunction` icon on the Math palette and use it to place two `RandomPLFunction` objects on the Form Designer. Since these are invisible, it doesn't matter where you put them, but to help keep them straight, you might put one of them near the `fGraph` and the other one near the `gGraph`.
5. From the Property List name the first one `f` and the second one `g`.
6. Using the Property List for `fGraph` set the `F` property to `f`. Similarly, set the `F` property of `gGraph` to `g`. You should see their graphs displayed.
7. Open up the source code for the Applet. (Right-click on the Form Designer and choose **Edit Source** from the resulting menu.) We are going to use the `MathFunction` class from the "mathtools" package of the `MathToolkit`. In order to do this we will have to import that package. At the beginning of the source code you should see a number of "import statements". Add the following one:

```
import EDU.emporia.mathtools.*;
```

8. We are going to use `hGraph` to display various combinations of `f` and `g`. First, we will need to write the code necessary to define `h`, the function that will be graphed on `hGraph`. We first need to "declare" `h` to be an object of type `MathFunction`. We will do that right at the beginning of the Applet. Right after the opening curly brace of the Applet (before the `init` method) insert the following line:

```
MathFunction h;
```

9. At the end of the `init` method insert the following two lines:

```
h = f.plus(g);  
hGraph.addGraph(h);
```

The first line calls the `plus` method of `f` and sends it the parameter `g`. The effect of it is to define `h` to be `f+g`.

10. Execute the Applet and convince yourself that the graph of h is the graph of $f+g$.
11. Try the following variations on the definition of h . You can probably guess what each one does:

- `h = f.times(g);`
- `h = f.minus(g)`
- `h = f.dividedBy(g);`
- `h = f.times(2);`
- `h = f.plus(-3);`
- `h = f.composedWith(g);`

11. Of course, you could combine many of those to get any number of variations. One combination is a little more difficult to do than it may appear at first. Suppose that we want to define h by $h(x) = f(x+1)$. As an aid in doing this there is a “static” variable called `IDENTITY` in the `MathFunction` class which is the function defined by $y = x$. See if you can parse the follow expression to see why it works to define h by $h(x) = f(x+1)$:

```
h = f.composedWith(MathFunction.IDENTITY.plus(1));
```

Note: The `MathFunction` class also has a `probablyEquals` method which can be used to check to see if two functions are the same. It compares them at 20 randomly selected x -values and returns true if they agree to within a very small tolerance at those 20 values. So, for example, if you have two `MathFunction` objects f and g , `f.probablyEquals(g)` would be true if they agreed on those 20 points.

The MathTable and Plotting Points

MathGrapher objects also have the capability of plotting individual points. One of the easiest ways to do this is in conjunction with the *MathTable*.

1. Close down any projects that you have open and start a new project. (Again, choose **New Project...**, from the **File** menu and take the **JFCApplet** template.) Save the project into a new folder.
2. On the Form Designer, place three components: a MathGrapher and a MathTable (from the Math palette) and a JButton (from the Swing tab.)
3. By editing the *Name* property of each one from the Property List, call the MathGrapher *graph*, the MathTable *table*, and the JButton *graphButton*.
4. Change the *Text* property of the JButton to “Graph”.
5. When the user clicks on the JButton we want the points to be graphed. In order to do this we will have to set up an *event handler* for the JButton. In order to do this we will have to open up the Source Code Window. Do this by right-clicking on the Form Designer and choosing **Edit Source** from the resulting menu.
6. Look at the top of the resulting window. You should see two choice boxes. One of them is labeled **Objects:** and the other is labeled **Events/Methods:**. From the **Objects** box select *graphButton*. Then go to the **Events/Methods** box and select *actionPerformed*. This will result in a lot of code being written. We are interested in the block of code corresponding to the *graphButton_actionPerformed* method. Any code that is in there will be executed whenever the user presses on the button.
7. In this *graphButton_actionPerformed* method (between the curly braces) put in the following line:

```
graph.addPoints(table.getPoints());
```

8. (**Warning:** the above line will work only if you have followed the instructions above in naming the MathGrapher *graph* and the MathTable *table*.) Notice what the above line is doing. It calls the *getPoints* method of *table*. *getPoints* which returns an array of *Point2d.Double* objects that corresponds to the data that has been entered into the table. That array is then passed as a parameter to the *addPoints* method of *graph* which displays them on the graph.
9. Execute the Applet and enter in some data, then press the button. (**Note:** Data entered into a cell isn't actually registered until you move out of the cell. So, if you push the *graphButton* while your cursor is still active in the last cell, the last point probably won't be graphed.)
10. Try entering in an arithmetic expression into one of the cells. Also, try going back and editing a cell that you have previously put data in to see what happens.

Tangent Lines and Secant Lines

In this section we will illustrate how to use the `TangentLine` and `SecantLine` components by setting up a sample mathematical activity that uses them. As usual let's begin by putting everything that we need onto the form designer.

1. Start a new project using the JFC Applet template.
2. On the Form Designer put a `SymbolicFunction` object (call it f) and a `TangentLine` object (call it $tangent$). (Both are on the Math Palette)
3. On the Property List for f set *formula* to be “sin(x)”.
4. On the Property List for $tangent$ set the *F* property to be f . This signals that $tangent$ will be the tangent to f . Set *xBase* to be 0.5. This indicates that it should be the tangent at $x=0.5$.
5. At this point we have a choice. If we leave it like it is, it will compute the slope by approximating the derivative. On the other hand, if we want to be precise, we can specify the property *fPrime* to be the derivative function. For the purposes of this activity it probably won't make that much difference, but let's go ahead and be precise. So, put another `SymbolicFunction` object on the Form Designer, call it $fPrime$, and specify it's formula to be “cos(x)”. Then set the *fPrime* property of $tangent$ to be $fPrime$.
6. Put a `SecantLine` object onto the Form Designer. Call it $secant$. Set the *xOne* property to be 0.5 and the *xTwo* property to be 2.
7. Add a `MathGrapher` object to the Form Designer and call it $graph$. To better show the curve use the Property List to set *xMin* to be -3, *xMax* to be 3, *yMin* to be -2 and *yMax* to be 2. Set the *F* property to be f and the *G* property to be $tangent$.
8. We will have to manually add the $secant$ graph. Open up the Source Code Window (by right-clicking on the Form Designer and choosing **Edit Source**). Find the *init* method. It begins with the line “public void init” and includes all the code between the opening and closing curly brace after that line. At the end of the *init* method (before the closing curly brace), insert the following line:

```
graph.addGraph(secant);
```

9. At this point another issue has arisen because of the order in which Visual Café has put in the instructions. The main issue is that it called the *setFPrime* method of $tangent$ before the formula for $fPrime$ had been defined. We could have avoided this by being more careful about the order in which we added objects to the Form Designer, but that probably requires more planning than is necessary. To correct this we will add two lines at the end of the *init* method.

```
tangent.updateSlope();  
graph.updateGraph();
```

10. There is also an *updateSlope* method for the `SecantLine` class which isn't needed in this case if you put things on in the order specified.
11. At this point you may want to execute your applet to see how it works.
12. Now let's add three `MathTextFields`. (The `MathTextFields` are on the Math tab of the palette.) By editing the *Name* property on the Property List of each, call the first one *inputTextField*, call the second one *tangentTextField* and the third one *secantTextField*. Next to *inputTextField* put a `JLabel` (which can be found on the

Swing tab of the palette) and set it to display the label “x2” by editing the *Text* property on the Property List. Similarly, label *tangentTextField* to be “Tangent Slope” and *secantTextField* to be “Secant Slope”.

13. We just want *tangentTextField* and *secantTextField* to display information so set the *editable* property for each of them to be *false*.
14. The value in the *inputTextField* is supposed to represent the x-coordinate for the second point used to draw the secant line. Since we have started our secant line off to go from 0.5 to 2, set the *text* property of *inputTextField* to be “2”.
15. Let’s initialize the other two text fields. Their values should reflect the slopes of the tangent line and the secant line, respectively, so add the following at the end of the *init* method.

```
tangentTextField.setMathValue(tangent.getSlope());  
secantTextField.setMathValue(secant.getSlope());
```

16. Notice the call to the *getSlope* methods of *secantLine* and *tangentLine*. Now set up an *ActionPerformed* event handler for *inputTextField*. (Do this by going to the Source Code window and choosing *inputTextField* from the **Objects** choice box at the top, then selecting *ActionPerformed* from the **Events/Methods** choice box.) In the resulting *inputTextField_actionPerformed* method, put the lines

```
secant.setXTwo(inputTextField.getMathValue());  
graph.updateGraph();  
secantTextField.setMathValue(secant.getSlope());
```

17. Execute your applet. Enter some values into *inputTextField* and observe what happens as *x2* gets closer to 0.5. (You fire the *ActionPerformed* event by hitting the Enter key while your cursor is in *inputTextField*.)

Note: There is one technical detail that can come up when working with the *SecantLine* class. The *setXTwo* and *setXOne* methods will ignore any attempts to set *xOne* or *xTwo* to values that would make them equal (because the slope would be undefined). This means that, for example, if you enter 0.5 into the box, the slope will just be whatever the last value was that was displayed. It also means that, for example, if, from the Property List, you want to set *xOne* to be 1, when *xTwo* is already 1, the changes may not be reflected in the graph. You can get around this by first changing *xTwo* then *xOne*, but, if you do this from the Property List, be sure to look at the order that Visual Café writes the instructions. It may be safer to do it by hand.

The FunctionTable and the ParametricTable

In this section we will illustrate how to use the FunctionTable and ParametricTable components.

1. Start a new project using the JFC Applet template.
2. On the Form Designer put two SymbolicFunction objects. Call one of them f and the other one g .
3. On the Property List for f set *formula* to be “ $\arctan(x)$ ”.
4. On the Property List for g set *formula* to be “ $2*x^2/(x^2+3*x+4)$ ”.
5. Drag a FunctionTable object onto the form designer and call it *table*.
6. Set the F property of *table* to be f and the G property to be g . Notice that it adds another column to accommodate two functions.
7. Experiment with editing the following properties until you understand what they all do: *autoXValue*, *xMin*, *xDelta*, *xLabel*, *yLabel*.
8. Execute your applet and click on the y -headings to see how the title changes. Try it both with *autoXValue* set at *true* and set at *false*.

Try a similar experiment with the ParametricTable and with two *SymbolicParametric* objects

Polygonal Curve

In this section we will illustrate how to use PolygonalCurve class. This class is not a component so it will not appear on your component palette. It is in the mathtools package. It constructs a parametrically defined polygonal curve between points that you specify.

1. Start a new project using the JFC Applet template.
2. Import the `EDU.emporia.mathtools` package in the usual way.
3. Declare and instantiate the variable *curve* to be of type PolygonalCurve by putting the following statement in the outer block:

```
PolygonalCurve curve = new PolygonalCurve();
```

4. Put a MathGrapher object on the Form Designer and call it *graph*.
5. At the end of the *init* method add *curve* to *graph* in the usual way with the statement:

```
graph.addGraph(curve);
```

6. Create a *mousePressed* event-handler for *graph*. (Do this from the Source Code Window by selecting *mousePressed* from the **Events/Methods** choice box while *graph* is in the **Objects** choice box.) In the resulting *graph_mousePressed* method insert the following code:

```
double x = graph.xPixelToMath(event.getX());
double y = graph.yPixelToMath(event.getY());

curve.addPoint(x,y);

graph.updateGraph();
```

7. Notice the *addPoint* method of *curve*. This will add a new point to the PolygonalCurve object (which will be parametrically defined to draw line segments between the points).
8. Execute your Applet. Click on the coordinate system several times to see what happens.

The SlopeField Component

In this section we will illustrate how to use the *SlopeField* component. It constructs a Slope Field that can be used to study a system of differential equations of the form:

$$\frac{dy}{dt} = f(x, y)$$
$$\frac{dx}{dt} = g(x, y)$$

1. Start a new project using the JFC Applet template. Remember to save your project into a separate folder.
2. Find the SlopeField component on the Math tab of the component palette and use it to place a SlopeField object on the Form Designer and call it *mySlopeField*.
3. Look at the property list for *mySlopeField*. Since the SlopeField class is an extension of the MathGrapher class all the properties of MathGrapher are there and should be familiar to you. There are some new ones, though. Experiment with the properties *FieldColor* and *SlopeRadius* until you understand what they do.
4. The properties *XDerivative* and *YDerivative* correspond to the formulas for $g(x, y)$ and $f(x, y)$ above. They should be algebraic expressions in the variables x and y . Experiment with those to see the effect on *mySlopeField*. (**Note:** If you want the Slope Field to correspond to a first order differential equation of the form $\frac{dy}{dx} = f(x, y)$ just make *XDerivative* equal to 1, which is the default value. In particular, if you want it to correspond to an antiderivative make *XDerivative* equal to 1 and make *YDerivative* an expression involving only x .)
5. There are some other properties that you will only see at run-time. Before experimenting with those, let's go ahead and execute the Applet to see how the SlopeField class works in action. After choosing appropriate values for *XDerivative* and *YDerivative*, execute the Applet. Click on the Slope Field with the mouse. It will then draw an approximation to a solution to the system of differential equations using Euler's method. (It draws a series of line segments that are linear approximations to the solution based on the derivatives at that point.)
6. Now shut down the applet and let's go back and look at some of the other properties.
 - a. *EulerDelta* determines the increment that is used to determine the next point in Euler's method. In general, the smaller this is the more accurate the approximation will be.
 - b. *EulerGraphColor* determines the color that will be used to draw the Euler approximation.
 - c. *EulerMouseEnabled* determines whether or not an Euler approximation is drawn when the mouse is clicked on the Slope Field.

- d. *EulerPointNumber* determines the number of Euler points that are computed in each direction.
7. Experiment with the above features until you feel comfortable with them. (In addition to the above new properties, there is one change in the behavior of the MathGrapher properties. Since the mouse click is used for the Euler graph, the zoom features, if enabled, require the user to press the Control key while clicking with the mouse.)

A Predator-Prey Example

As an illustration of the use of the *SlopeField* let's set up an Applet to investigate solutions to a differential system corresponding to a *Lotka-Volterra* predator-prey model. We will consider the system.

$$\frac{dy}{dt} = y - 0.5xy$$

$$\frac{dx}{dt} = -x + 0.7xy$$

Here x represents the number of foxes (in hundreds) and y represents the number of rabbits (in thousands).

1. You can use the same Applet if you like. Since negative values of x and y are not relevant for this model use the property list for *mySlopeField* to set $XMin = 0$, $YMin=0$, $XMax=5$, $YMax=5$.
2. Set the *XDerivative* property and the *YDerivative* property to correspond to the above system. (Don't forget to use * to indicate multiplication.)
3. It turns out that the default value of *EulerDelta* is too crude for this system, so set it to be 0.01.
4. Now that we have made *EulerDelta* smaller we will need to increase the value of *EulerPointNumber* in order to get a complete graph. Set it equal to 400.
5. Go ahead and execute the Applet. Click on the Slope Field to draw some graphs. (Interpret the graphs in terms of the relationship between the number of foxes and the number of rabbits as time progresses.)
6. We want to do an investigation to find the approximate *equilibrium point*, the point at which the populations of foxes and rabbits stays fixed over time. But, since our graph can get cluttered it might be useful to provide the capability to erase the graphs that are already there. Let's add a JButton to do that. From the Swing tab on the component palette select the JButton and place it on the Form Designer. Call it *clearButton* (by changing the *Name* property) and change the *Text* property to "Clear".
7. Set up an event handler to handle an *actionPerformed* event on *clearButton*. ((1)Go to the source code by right-clicking on the Form Designer and choosing **Edit Source**. (2)From the **Object** choice box at the top of the Source Code window select *clearButton*. and (3) select *actionPerformed* from the **Events/Methods** choice box.
8. When the *clearButton* is pressed we want to clear out all the graphs. Each graph corresponds to a *dataPoint* that is used to generate the graph. To clear the graphs we

need to remove all the *dataPoints*. To do this we will use the aptly named *removeAllDataPoints* method. In the resulting *clearButton_actionPerformed* method add the following line:

```
mySlopeField.removeAllDataPoints();
```

9. Execute the Applet to see how it works.
10. If you want to set up your Applet so that the user can add a specific *dataPoint* by typing it in, you would use the *addDataPoint* method. This has two versions. One accepts as a parameters two numbers of type *double* corresponding to the *x* and *y* coordinate of the point to be added. The second version accepts a *Point2D.Double* object.

Antiderivative Exploration

We will do one more example with the Slope Field that will take advantage of the fact that it is also a MathGrapher. We want to set up an applet that will let the students explore the concept of the antiderivative by experimenting with various functions to see which ones seem to be consistent with the Slope Field.

1. You can either create a new project or modify your old one. Make sure that there is a *SlopeField* object called *mySlopeField* and that the values of *xMin*, *xMax*, *yMin*, and *yMax* are set to the default values of -10,10,-10,and 10.
2. From the Swing tab of the component palette select the *JTextField* and create a *JTextField* object large enough to hold an algebraic formula. Call it *fPrimeTextField*.
3. Put a *JLabel* to the left of *fPrimeTextField*. Set the *HorizontalAlignment* property (under *Alignment/Position*) to *RIGHT*. Set the *Text* property of the *JLabel* to “ $f'(x) =$ ”.
4. Now create another *JTextField* and call it *guessTextField*. Put a label next to it so that it says “Guess $f(x)$ ”.
5. From the Math tab find the *SymbolicFunction* icon and place a *SymbolicFunction* object on the Form Designer. (Remember that this is invisible so it doesn't matter where you put it.) Set the *Name* property to *guessFunction*. This will represent the user's guess for the antiderivative.
6. From the Property List for *mySlopeField* set the *F* property to *guessFunction* so that the graph of *guessFunction* will be displayed on *mySlopeField*.
7. We are going to set it up so that the user can alter the formula for $f'(x)$ by typing into *fPrimeTextField* and hitting *Enter*. In order to do this we will need to set up an *actionPerformed* event handler for *fPrimeTextField*. Do that now.
8. Make sure that the *XDerivative* property for *mySlopeField* is set to 1. In the *actionPerformed_fPrimeTextField* method that was created above we need to put in the code to alter the value of *YDerivative* so that it is the current formula in *fPrimeTextField*. Type the following lines into the *actionPerformed_fPrimeTextField* method:

```
try{
    mySlopeField.setYDerivative(fPrimeTextField.getText());
}catch(EDU.emporia.mathtools.Graphable_error e){}
```

9. Notice the try/catch structure that is set up to handle the exception that will be thrown if the user types in an illegal formula. This is necessary every time the *setYDerivative* method is called.
10. Now we want to let the user guess some functions that might be consistent with the slope field. Set up an *actionPerformed* event-handler for the *guessTextField*. Type the following code into the resulting *actionPerformed_guessTextField* method and execute your applet.

```
try {
    guessFunction.setFormula(guessTextField.getText());
    mySlopeField.updateGraph();
}catch(EDU.emporia.mathtools.Graphable_error e) { }
```

The PiecewiseLinearFunction, SplineFunction, and SplineCurve

In this section we will illustrate how to use some classes that are useful for creating custom curves. These classes are not components so they will not appear on your component palette. They are in the `mathtools` package.

The PiecewiseLinearFunction

In a previous tutorial we worked with the *PolygonalCurve* class. This class defined a parametric curve that was made up of a series of line segments passing through some given points. The *PiecewiseLinearFunction* class is similar except that it defines a *function* that passes through a given set of points. (Of course, that means that no two of the points can have the same x -coordinate.)

1. Start a new project using the JFC Applet template. Remember to save your project into a separate folder. Open the source code window for the Applet. (You can do this by right-clicking on the Form Designer and selecting **Edit Source** from the resulting context menu.)
2. Import the `EDU.emporia.mathtools` package in the usual way by adding the statement

```
import EDU.emporia.mathtools.*;
```

to the import statements at the beginning of the Applet.

3. Declare and instantiate the variable f to be of type *PiecewiseLinearFunction* by putting the following statement in the outer block (right after the first curly bracket):

```
PiecewiseLinearFunction f = new PiecewiseLinearFunction ();
```

4. By selecting the `MathGrapher` component from the `Math` tab of the component palette put a `MathGrapher` object on the Form Designer. By editing the *Name* property in the property list call it *graph*.
5. Add f to *graph* in the usual way by putting the following statement at the end of the *init* method:

```
graph.addGraph(f);
```

6. Create a *mousePressed* event-handler for *graph*. (You do this in the source code window by choosing *graph* from the **Object** choice box at the top of the window and *mousePressed* from the **Events/Methods** choice box.) In the resulting *graph_mousePressed* method insert the following code:

```

double x = graph.xPixelToMath(event.getX());
double y = graph.yPixelToMath(event.getY());

try{
    f.addPoint(x,y);
}catch(IllegalPointError e){}

graph.updateGraph();

```

7. Most of the above is probably familiar to you. The first two lines get the x and y coordinates where the Mouse was pressed, convert them from pixel coordinates to math coordinates, and assign them to the variables x and y . The new thing here is that the *addPoint* method of the *PiecewiseLinearCurve* object is called to add the point to f . This will result in the definition of f being adjusted so that it passes through the new point. (Notice that when the *addPoint* method is called it is necessary to do it in the context of a *try/catch* structure in order to handle the exception that occurs when an attempt is made to add a point with the same x -coordinate as one that is already in place.) Finally, the graph is updated so that it will redraw to reflect the new curve.
8. Execute your Applet. Click on the coordinate system several times to see what happens.
9. Of course, in order to define a piecewise linear function for use in an example, you would probably not leave it up to the user to assign the points. You would probably call the *addPoints* method from within the Applet at the time of definition. You can do this as in the code above or you can pass an entire array of points to the *addPoint* method. (They should be an array of *EDU.mathtools.Point2D.Double* objects.) To illustrate this, let's use a *MathTable* to create an array of points.
10. Get a *MathTable* from the Math tab of the component palette and put it on the Form Designer. Edit the *Name* property so that it is called *table*.
11. We will want a button that will trigger the event that will move the points from the table to the curve so let's put that on now. Get a *JButton* from the Swing tab and call it *addButton*. Edit the *Text* property of *addButton* so that it says "Add Points".
12. Set up an *actionPerformed* event-handler for *addButton*. (Choose *addButton* from the **Objects** choice box and *actionPerformed* from the **Events/Methods** choice box at the top of the Source Code window.)
13. Now we want to add the code into the resulting *addButton_actionPerformed* method. (Recall that this is the code that will be executed when *addButton* is pressed.) All we want to do is to get the points from *table* and add them to f . We will also need to put in the necessary *try/catch* structure. Type in the following into the *addButton_actionPerformed* block of code.

```

try{
    f.addPoints(table.getPoints());
    graph.updateGraph();
}catch(IllegalPointError e){}

```

14. There is one refinement that we should add to this, but go ahead and execute the applet to see how it works. Type in some numbers into *table* and press the *addButton* to see them reflected in the graph.
15. To see how we can improve the applet, try adding some new points to the table and pressing the button a second time. You should find that nothing happens. Can you

guess why? The problem is that the second time that you pressed the button it tried to add the points from the table and some of them were duplicates of points that had already been added. This generated a dreaded *IllegalPointException* and, according to the way that we set up our exception handling, nothing happened. (We could go back and include an error message in the catch part of the *try/catch* structure, but we are going to just try to avoid the error all together.)

16. One way of avoiding this problem is to remove the previous points before adding new ones. Fortunately, there is a *removeAllPoints* method that does just that. Insert the following line right at the beginning of the *addButton_actionPerformed* method.

```
f.removeAllPoints();
```

17. Execute your applet to see how it works.

The SplineFunction

The *SplineFunction* is similar to the *PiecewiseLinear* function except that it will define a differentiable function passing through a given set of points. In addition you can specify the value of the derivative at any one of the points (or you can just let it automatically assign a derivative).

1. The beginning of this part will be pretty much like the previous one, but let's go ahead and start a new project using the JFC Applet template.
2. Import the *EDU.emporia.mathtools* package in the usual way by adding the statement

```
import EDU.emporia.mathtools.*;
```

to the import statements at the beginning of the Applet.

3. Declare and instantiate the variable *f* to be of type *SplineFunction* by putting the following statement in the outer block:

```
SplineFunction f = new SplineFunction ();
```

4. By selecting the *MathGrapher* component from the *Math* tab of the component palette put a *MathGrapher* object on the Form Designer. By editing the *Name* property in the property list call it *graph*.
5. Add *f* to *graph* in the usual way by putting the following statement at the end of the *init* method:

```
graph.addGraph(f);
```

6. Create a *mousePressed* event-handler for *graph*. (You do this in the source code window by choosing *graph* from the **Object** choice box at the top of the window and *mousePressed* from the **Events/Methods** choice box.) In the resulting *graph_mousePressed* method insert the following code:

```

double x = graph.xPixelToMath(event.getX());
double y = graph.yPixelToMath(event.getY());

try{
    f.addPoint(x,y);
}catch(IllegalPointError e){}

graph.updateGraph();

```

7. The above code is identical to the code for the *PiecewiseLinearFunction* example. Once again we need to handle the exception that occurs when a point is added with a duplicate *x*-value.
8. Execute your Applet. Click on the coordinate system several times to see what happens.
9. As you can see from the above there is an *addPoint* method that allows us to add a point to the curve just as with the *PiecewiseLinearFunction* class. In the case of the *SplineFunction* class you can also use the *addPoint* method to specify a point and the slope of the curve at that point. Let's see how that works by setting up 3 *MathTextFields* to be used to enter in a point and a slope.
10. On the Math tab find the *MathTextField* component and use it to put three identical *MathTextField* objects on the Form Designer. (Recall that one way of doing this is to set up one of them the way you want, then, with the Control key pressed, click and drag it to produce a copy.) Edit the *Name* property from the Property List so that the three *MathTextFields* are called *xTextField*, *yTextField*, and *slopeTextField*.
11. Next to each *MathTextField* put a JLabel to identify it. (JLabels are on the Swing tab.) Label *xTextField* "x", *yTextField* "y", and *slopeTextField* "slope". (Recall that you do this by editing the *Text* property in the property list. Also, you probably want to change the *Horizontal Alignment* property to RIGHT. In addition, if you are like me and find the default color of Navy Blue to be hard to read, you can change the *Foreground* property to Black.)
12. We are going to use these to add one point at a time to our *SplineFunction* so we will need a JButton to fire the event that will result in adding the point. Add a JButton (from the Swing tab) and call it *addButton* and edit the *Text* property so that it reads "Add Point".
13. Set up an actionPerformed event handler for *addButton* and, in the resulting *addButton_actionPerformed* method type in the following:

```

double x = xTextField.getMathValue();
double y = yTextField.getMathValue();
double m = slopeTextField.getMathValue();

try{
    f.addPoint(x,y,m);
    graph.updateGraph();
}catch(IllegalPointError e){}

```

14. The first three lines use the *getMathValue* method of the *MathTextFields* to get the current values stored in each. The last part is similar to what we have seen before except that we have passed three parameters to the *addPoint* method which results in the slope being specified. (You can specify the slopes at some points and make others automatic.)

15. Execute the Applet. Enter some values into the *MathTextFields* to see how they work. (You can still click on the graph to add more points with the mouse.)

Creating a Custom Function

Armed with these tools you should be able to create a variety of examples. As an illustration let's create an example of a function f such that

$$\lim_{x \rightarrow 2^-} f(x) = 3$$
$$\lim_{x \rightarrow 2^+} f(x) = -1$$
$$f(2) \text{ is undefined}$$

1. You can either start a new project or modify the old one. Just make sure that you have a *MathGrapher* object on your Form Designer and that it is called *graph*.
2. We are going to create a new function by forming a new class that is an extension of the *MathFunction* class. From the **File** menu choose **New File**. In the resulting window type in the following:

```
import EDU.emporia.mathtools.*;

public class CustomFunction extends MathFunction
{
}
}
```

3. This sets up the skeleton for our *CustomFunction* class. The import statement is necessary because the *MathFunction* class is part of the *mathtools* package. Save the file into your project folder. Be sure that you save it under the name *CustomFunction*. (Visual Café should offer that as the default choice.) It should now appear in your Project Window.
4. We are going to create our function by splicing together two functions. The one on the left will be a *SplineFunction* and the one on the right will be a *PiecewiseLinearFunction*. Let's declare and instantiate one of each. Put the following statements right at beginning of your *CustomFunction* class. (On the line right after the first curly bracket.)

```
SplineFunction f = new SplineFunction();
PiecewiseLinearFunction g = new PiecewiseLinearFunction();
```

5. Now we will create a Constructor for the *CustomFunction*. Recall that the constructor is the first method that is called whenever a *CustomFunction* object is created. Type in the following right below the lines from the step above (but before the closing curly bracket.) After you have typed it in, see if you can figure out what each line is doing.

```

CustomFunction()
{
    try{
        f.addPoint(2,3,0);
        f.addPoint(0,0,1);
        f.addPoint(-3,-2,0);
        g.addPoint(2,-1);
        g.addPoint(5,2);
        g.addPoint(8,-3);
    }catch(IllegalPointError e){}
}

```

- Notice that, since f is a *SplineCurve*, we are able to specify both a point and a slope. So for example, f will pass through the point (2,3) with slope 0.
- Finally, recall that the *MathFunction* class is set up so that, in order to define a function all you need to do is override the *functionValue* method. We will do that now. Type in the following after the code above

```

public double functionValue(double x)
{
    if(x < 2) return f.functionValue(x);
    else if (x == 2) return Double.NaN;
    else return g.functionValue(x);
}

```

- You can probably figure out what the above statements are doing. The one thing that may be new is *Double.NaN*. This is the expression for the *double* value that is “Not a Number”. This is our way of making the function undefined at $x=2$. When a *MathGrapher* object encounters this it essentially picks up the pen and stops graphing.
- We are now done with the *CustomFunction* class. Go ahead and compile it to make sure that you haven’t made any typing errors. (Choose **Compile CustomFunction.java** from the **Project** menu.)
- Finally, we need to graph it. Open up the source code for the Applet. At the beginning of the Applet insert the following line:

```

CustomFunction h = new CustomFunction();

```

- This will create a new *CustomFunction* object. Now all we need to do is add it to the graph. We’ll do that in the *init* method. Insert the following line just before the closing curly brace of the *init* method of the Applet. (Remember to avoid putting anything between the green double curly braces. This is the area reserved for Visual Café.)

```

graph.addGraph(h);

```

- This, of course, assumes that you have named your *MathGrapher* object *graph*. Now execute your applet. Try using the arrow keys to trace the curve. (Remember to click on the graph first to make sure that it has the focus.)

The SplineCurve

For the sake of completeness there is also a *SplineCurve* class. This constructs a parametrically defined curve through specified points in the order in which they are added. Unlike the *PolygonalCurve* the *SplineCurve* constructs a differentiable curve. You can specify the derivative at any point by giving two coordinates which will serve as a tangent vector. (Both the magnitude and the direction of the vector affect the definition of the curve.)

1. The steps will be familiar. You may want to just edit one of your previous projects, but in case you are creating a new one, we will repeat the steps. Start a new project using the JFC Applet template.

2. Import the `EDU.emporia.mathtools` package by adding the statement

```
import EDU.emporia.mathtools.*;
```

to the import statements at the beginning of the Applet.

3. Declare and instantiate the variable f to be of type `SplineCurve` by putting the following statement in the outer block:

```
SplineCurve f = new SplineCurve ();
```

4. Call the `MathGrapher` `graph`.

5. Add f to `graph` by putting the following statement at the end of the `init` method:

```
graph.addGraph(f);
```

6. Create a `mousePressed` event-handler for `graph`. In the resulting `graph_mousePressed` method insert the following code:

```
double x = graph.xPixelToMath(event.getX());
double y = graph.yPixelToMath(event.getY());

f.addPoint(x, y);

graph.updateGraph();
```

7. Notice that, unlike with `PiecewiseLinearFunction` and `SplineFunction` there is no `try/catch` structure since the `addPoint` method of `SplineCurve` does not throw an exception. (There is no problem created by adding duplicate points.)
8. Execute your Applet. Click on the coordinate system several times to see what happens.
9. To illustrate how to specify both a point and a slope put four identical `MathTextField` objects on the Form Designer. Edit the `Name` property from the Property List so that the four `MathTextFields` are called `xTextField`, `yTextField`, `xSlopeTextField`, and `ySlopeTextField`.
10. Next to each `MathTextField` put a `JLabel` to identify it. (`JLabels` are on the Swing tab.) Label `xTextField` "x", `yTextField` "y", `xSlopeTextField` "x slope", and `ySlopeTextField` "y slope".
11. Add a `JButton` (from the Swing tab) and call it `addButton` and edit the `Text` property so that it reads "Add Point".
12. Set up an `actionPerformed` event handler for `addButton` and, in the resulting `addButton_actionPerformed` method, type in the following:

```
double x = xTextField.getMathValue();
double y = yTextField.getMathValue();
double xSlope = xSlopeTextField.getMathValue();
double ySlope = ySlopeTextField.getMathValue();

f.addPoint(x, y, xSlope, ySlope);
graph.updateGraph();
```

13. The result of these lines is that the curve f will pass through the point (x,y) and have tangent vector $\langle xSlope, ySlope \rangle$.

14. Execute the Applet. Enter some values into the *MathTextFields* to see how they work.
(You can still click on the graph to add more points with the mouse.)

The Java Numerical Library and Working with Matrices

In this section we will illustrate how to use some of the classes from the Java Numerical Library (JNL). The JNL is from Visual Numerics and can be downloaded for free from <http://www.vni.com/products/wpd/jnl/>. It contains classes that perform numerical computations with Vectors, Complex Numbers, Matrices, and Statistics. We will illustrate the use of the library by constructing a primitive Matrix Calculator.

Setting up the Library

After downloading the library you should have a folder called *JNL* with two subfolders: *api* and *Classes*. The *api* folder contains the documentation for the library. You may want to browse through that to see what is available. Inside the *Classes* folder is a folder called *VisualNumerics*. In order to access the JNL from within Visual Café this folder must be in the classpath. The easiest way to do this is to put a copy of this folder into the **Visual Café|Java|Lib** folder. (**Note:** You want the *VisualNumerics* folder there, *not* the *Classes* folder.) Once you have done this you are ready to go.

Creating a Matrix class

The JNL basically consists of a library of static methods that perform numerical computations. (Recall that a *static* method is one that you can call directly without bothering to create an instance of the class.) It doesn't attempt to create much of a user interface. We will try to do this by creating a Matrix class. Although admittedly a work in progress this class may have the potential of eventually becoming a useful *mathbean*.

1. Start a new project using the JFC Applet template. Remember to save your project into a separate folder.
2. Choose **New File** from the **File** menu. Let's go ahead and create the skeleton for the class. We are going to represent the matrix as a grid of *MathTextFields* on a *JPanel*. So it will be an extension of *JPanel*. We are going to have to import a lot of packages, but let's begin with the *Swing* package where *JPanel* resides. Put the following as the first line of your new file.

```
import javax.swing.*;
```

3. Now we will put in the skeleton of the *Matrix* class.

```
public class Matrix extends JPanel  
{  
  
}
```

4. Save the file into your Project Folder making sure that you save it under the name *Matrix*. (Visual Café should offer that as the default choice.) It should now appear in your Project Window.
5. The entries of the matrix will appear in *MathTextFields* so that the user can edit them. So we will want a doubly subscripted array of *MathTextFields* to be added to our *JPanel*. Before doing this we will need to import the *mathbean* package where the *MathTextField* class lives. So add the following import statement to the beginning of the file:

```
import edu.emporia.mathbeans.*;
```

6. In addition we will want variables *rows* and *columns* to represent the number of rows and columns for the matrix so we will declare those variables, too. Add the following declarations at the beginning of the Class (after the first curly bracket).

```
MathTextField[] [] entry;
int rows, columns;
```

7. We are going to add a title to our JPanel so that we can label the matrix. To do that we will add a *TitledBorder*. But it is in still another package (javax.swing.border) so first we will have to import it. And we are going to use a *layout manager* which is in the java.awt package so we will need to import that, too. Add the following import statements to the others:

```
import javax.swing.border.TitledBorder;
import java.awt.*;
```

8. Now we can declare the TitledBorder. Add the following declaration after the ones for the *entry*, *rows*, and *columns*.

```
TitledBorder border = new TitledBorder("");
```

9. Notice that the version of the constructor for TitleBorder that we used required a String as a parameter. This String will be the title. We just gave it the empty string so that there will be no title until one is specified. Now we are ready to create a constructor for the Class. We could create a couple of versions, but let's just create the most general one. This one will specify the number of rows, columns, and the title for the Matrix. Type the following into the main body of the class. We'll explain what it all does afterwards:

```
public Matrix(int r, int c, String t)
{
    rows = (r>0 ? r : 1);
    columns = ( c>0 ? c : 1);

    setBorder(border);
    border.setTitle(t);

    setLayout(new GridLayout(rows, columns, 0, 0));

    entry = new MathTextField[rows][columns];

    for (int i = 0; i < rows ; i++)
    {
        for(int j = 0 ; j<columns ; j++)
        {
            entry[i][j] = new MathTextField();
            entry[i][j].setMathValue(0);
            add(entry[i][j]);
        }
    }
}
```

10. In the first two lines we want to set the value of *rows* and *columns* to the values specified by the constructor parameters. However, we want to make sure that they are positive integers. So we test to see if the numbers are positive and assign a value of 1 if they are not. The second two lines sets the Border to be the *TitledBorder border* that was declared above and sets the Title according to the specified parameter. Next we set the Layout manager for our *JPanel* to be a *GridLayout* with the number of rows

and columns of the grid determined by the value of the variables *rows* and *columns*. The result of this is that, as components are added to the *JPanel*, they will be laid out in a Grid.

11. Finally, the nested *for* loops create a new *MathTextField* object for each element of the *entry* array, set its value to 0, and add the *MathTextField* to the *JPanel*. The layout manager will then arrange them in a grid form. (Hopefully, this will make it look somewhat like a matrix.)

Testing it Out

1. Let's try it out to see what it looks like so far. We will create a *Matrix* object and add it to our Applet to see what it looks like. Open up the source code window for the *JApplet* of your project. We will declare and instantiate a *Matrix* object called *A* using the constructor that we just created. Insert the following line at the beginning of the main block of the applet:

```
Matrix A = new Matrix(3, 3, "A");
```

2. This is declaring *A* to be a 3 by 3 matrix with title "A". Now we want to display it in the Applet. We will do that at the end of the *init* method. Insert the following lines:

```
getContentPane().add(A);  
A.setBounds(10, 10, 200, 100);
```

3. The first line adds it to the *ContentPane* of the Applet. The second line specifies that it should be placed 10 pixels over and 10 pixels down from the upper left hand corner and that it should be 200 pixels wide and 100 pixels tall.
4. Execute the Applet to see what it looks like. Notice that, since the *JPanel* is made up of *MathTextFields* you can edit the entries and even use mathematical expressions such as *Pi* or *sqrt(2)* in there.

Adding Functionality

1. Unfortunately, our class creates something that is a matrix in appearance only. It has no functionality. We will add some methods to the *Matrix* class to provide that and we will use the JNL to do the computational work. But first we have to (you guessed it!) import the JNL into our class. So go back to the source code for the *Matrix* class and add the following import statement:

```
import VisualNumerics.math.*;
```

2. Before adding functionality we need to add some *get* and *set* methods that will allow us to get information from the *Matrix*. In particular we want to be able to query the *Matrix* about its entries and we want to be able to set the values of the entries. One issue that we should think about is how we want to handle a situation where someone tries to access or change a non-existent entry. For example, through one of the methods we might ask for the value of the (3,4) entry, in a 2 by 2 matrix. One way of handling this would be to throw an exception when this happens. This may be the best solution, but, in order to keep it as simple as possible, we will set ours up without exception handling.
3. Let's begin by defining a *setEntry* method. We will use this method to set the value of the entry in a given row and column. We will set it up to just ignore illegal attempts to

set the entry. Type in the following code at the end of the outer block of the *Matrix* class. (Before the last curly bracket.)

```
public void setEntry(double x, int r, int c)
{
    if ( (r < rows)&&(r >= 0)&&(c < columns)&&(c >= 0) )
        entry[r][c].setMathValue(x);
}
```

4. The above method sets the entry in the (r, c) spot to be x (provided there is an (r, c) spot). Now we will add an entry to allow a doubly subscripted array of values to be passed as values for the matrix. It is hoped that the dimensions of this array will match the dimensions of the matrix, but we will set it up so that a misguided attempt won't break our program. Insert the following method:

```
public void setEntries( double[][] x)
{
    for ( int i = 0 ; i < x.length ; i++)
        for ( int j = 0 ; j < x[i].length ; j++)
            setEntry(x[i][j], i, j);
}
```

5. Notice that any illegal values will be taken care of by the *setEntry* method that we defined in the previous step. Now we will add some corresponding *get* methods to allow us to get information from the matrix. Let's begin with a *getEntry* method. Insert the following:

```
public double getEntry(int r, int c)
{
    if ( (r < rows)&&(r >= 0)&&(c < columns)&&(c >= 0) )
        return entry[r][c].getMathValue();
    else
        return Double.NaN;
}
```

6. Under normal circumstances this will return the value in the r th row and the c th column. If an illegal value is called for it will return the double value *Double.NaN* which represents "Not a Number". Now add the *getEntries* method as defined below:

```
public double[][] getEntries()
{
    double x[][] = new double[rows][columns];
    for ( int i = 0 ; i < rows ; i++)
        for ( int j = 0 ; j < columns ; j++)
            x[i][j] = getEntry(i, j);

    return x;
}
```

7. Notice that the above will return a doubly subscripted array representing the values of the matrix. We are finally ready to add some functionality to our *Matrix* class. We will be using the static methods of the *DoubleMatrix* class of the JNL. These allow us to add, subtract, and multiply matrices. We can find the transpose, the inverse, the determinant, and the trace. To the *DoubleMatrix* class a matrix is just a doubly subscripted array of *double* values so we will be using the *getEntries* method and the *setEntries* methods that we defined above to get the required information. Let's start out by setting up a method in our *Matrix* class to allow us to add it to another *Matrix* object. Here it is:

```

public Matrix add(Matrix M)
{
    double x[] [] = getEntries();
    double y[] [] = M.getEntries();

    double[] [] sum = DoubleMatrix.add(x,y);

    Matrix answer = new Matrix(rows, columns, "Sum");

    answer.setEntries(sum);
    return answer;
}

```

8. The first two lines arrange that x contain the values for the current *Matrix* object while y contains the values for the matrix M passed as a parameter. The third line calls the static method *add* from the *DoubleMatrix* class and passes x and y as parameters. This method returns a doubly subscripted array that represents the sum of the matrices and we assign that to the variable *sum*. Finally, we create a new *Matrix* object called *answer* and set its entries so that they are the same as *sum*. We then return *answer*. (**Note:** The *add* method has the potential of throwing an *IllegalArgumentException* which will happen, for example, if you try to add two matrices of different dimensions. You have the option of throwing caution to the wind and not handling this exception, as we did here, in order to keep it simple.)
9. Let's add one other handy method to our *Matrix* class before giving it another trial run. We would like to be able to set the title of the matrix, so we need a *setTitle* method. Insert the following method into your *Matrix* class:

```

public void setTitle(String t)
{
    border.setTitle(t);
    repaint();
}

```

10. The first line sets the title, which is a property of *border*, and the second line calls for a *repaint* so that the new title will appear on the screen.

Another Test

1. Let's give this a trial run to see how it works. We will create two 3 by 3 matrices in our applet and another one to contain the answer. We will also put a *JTextArea* on the Form Designer to handle output. We want to leave some room for some buttons, too. So start out by making the Form Designer larger. By editing the *Width* and *Height* properties of the Applet in the Property List make it 600 pixels wide and 400 pixels high.
2. Now add the following lines at the beginning of the *Applet*. (Make sure that you are in the source code window for the Applet and not the *Matrix* class.) This assumes that you still have the declaration for A that we did above.

```

Matrix B = new Matrix(3,3,"B");
Matrix answer = new Matrix(3,3,"Answer");

```

3. Add the following code to the end of the *init* method. (Again, this assumes that the code for adding the *Matrix A* is still there.)

```

getContentPane().add(B);
B.setBounds(10, 150, 200, 100);

```

```

getContentPane().add(answer);
answer.setBounds(370,10,200,100);

```

4. This should position *B* in the lower left corner and *answer* in the upper right. (If you want to see where they are you will have to execute the applet.)
5. We are going to use a *JTextArea* to hold text output, but, in case the output should be larger than the size of the *JTextArea* we will put it on a *JScrollPane* which will add scrolling capabilities. Get a *JScrollPane* from the *Swing Containers* tab on the palette and put it in the lower right part of the Form Designer. Just to be sure that it is in the right place, edit the *Bounds* properties on the Property List so that *X* is 370, *Y* is 150, *Width* is 200, and *Height* is 140.
6. Now get a *JTextArea* and put it on the *JScrollPane*. Edit the *Name* property so that it is called *outputTextArea*.
7. We will use a *JButton* to trigger the addition of the matrices. Get a *JButton* from the *Swing* tab and put it on the Form Designer at the top in the middle. Again, to make sure that it is positioned correctly, edit the *Bounds* properties so that *X* is 240, *Y* is 25, *Width* is 90, and *Height* is 20. This button will signal addition of the matrices so edit the *Name* property to be *addButton* and edit the *Text* property to be "A+B".
8. Now set up an *actionPerformed* event-handler for *addButton* in the applet. (Choose *addButton* from the **Objects** choice box and *actionPerformed* from the **Events/Methods** choice box at the top of the Source Code window for the applet.)
9. In the resulting *addButton_actionPerformed* method insert the following code:

```

answer.setEntries(A.add(B).getEntries());
answer.setTitle("A+B");

```
10. All the action is taking place in the first line, so let's parse that. *A.add(B)* passes the parameter *B* to the *add* method of *A*. Recall that we set that up so that it would return a *Matrix* object that corresponds to the sum of *A* and *B*. We then call the *getEntries* method of that resulting matrix (which represents *A+B*) and that retrieves a doubly subscripted array corresponding to the values in the entries for *A+B*. Finally, we pass that array to the *setEntries* method for *answer* resulting in the values of *answer* being the same as those of *A+B*.
11. Execute your applet. Edit the entries of *A* and *B* then press the *addButton* to see the answer displayed there.

Exercise 1: The *DoubleMatrix* class of the JNL also has a *subtract* and a *multiply* method which have the same signature as the *add* method. Insert *subtract* and *multiply* methods into our *Matrix* class then test it out by adding a *subtractButton* and a *multiplyButton* to the Applet so that the answer will appear

The Inverse and the Transpose

Next we will use the *DoubleMatrix* class to add the capability of computing the inverse and the transpose. We'll do the inverse and leave the transpose as an exercise. The *DoubleMatrix* class has an *inverse* method which computes the inverse of a matrix (again represented as a doubly subscripted array). However, the *inverse* method throws two types of Exceptions. The first type is an *IllegalArgumentException* as in the previous methods. This is thrown, for example, if you try to get the inverse of a matrix that is not square. As we saw above, you can choose not to handle this exception. This might make

sense, for example, if your applet is designed, as ours was, to only deal with square matrices. However, it also throws a *MathException*. This occurs if the matrix is square, but not invertible. This Exception must be handled.

1. We begin, as before, by setting up the appropriate method in the *Matrix* class. Add the following method:

```
public Matrix inverse() throws MathException
{
    try{
        double[][] inverse = DoubleMatrix.inverse(getEntries());
        Matrix answer = new Matrix(rows, columns, "Inverse");
        answer.setEntries(inverse);
        return answer;
    }catch (MathException e){ throw e;}
}
```

2. We are setting up our method so that, if the *MathException* occurs we will just throw it from the *inverse* method and it can be handled in an appropriate manner in the program from which it is called.
3. Now go back to the applet and add an *inverseButton*. Edit its text property to be “inv(A)” and set up an *actionPerformed* event handler for it.
4. We are going to handle the *MathException* in the applet. Since this is in the *VisualNumerics* package we will need to import it. Add the following import statement to the beginning of the applet.

```
import VisualNumerics.math.*;
```

5. In the *inverseButton_actionPerformed* method insert the following code:

```
try{
    answer.setEntries(A.inverse().getEntries());
    answer.setTitle("inv(A)");
}catch(MathException e){
    outputTextArea.append("A is not invertible.\n");
}
```

6. The only really new thing here is the exception handling mechanism. We will attempt to compute the inverse of *A* and, if it is invertible, that will be displayed in *answer*. Otherwise we will put a message in *outputTextArea* alerting the user to the problem.
7. Execute your applet to see how it works. Try some examples where *A* is invertible and some where it is not.

Exercise 2: Add a *transpose* method to your *Matrix* class. Add a *transposeButton* to the Applet and set it up to compute the transpose of *A*. (Note: As you might expect, the *transpose* method of *DoubleMatrix* does not throw a *MathException* so you don’t need to worry about handling that.)

The Determinant and the Trace

The *DoubleMatrix* class also has methods for computing the determinant and the trace. Both the *determinant* method and the *trace* method throw *MathExceptions* so they must be handled. The *determinant* method throws a *MathException* when the matrix is singular. (This apparently is because it uses the *LU* factorization to compute the determinant.) The *trace* method throws it when the matrix is not square. (It would seem

to me to be more consistent with the other methods to throw an *IllegalArgumentException* in this case, but that is not how they did it.)

1. Again, let's do the determinant and leave the trace as an exercise. Add the following method to the *Matrix* class.

```
public double determinant() throws MathException
{
    try{
        return DoubleMatrix.determinant(getEntries());
    }catch(MathException e){throw e;}
}
```

2. Notice that since the determinant is a number and not a matrix the return type is *double*. Now go to the applet and insert a *determinantButton* and label it "det(A)". Create an *actionPerformed* event-handler and insert the following code into the *determinantButton_actionPerformed* method:

```
try{
    double d = A.determinant();
    outputTextArea.append("det(A)=" + d + "\n");
}catch(MathException e){
    outputTextArea.append("A is singular.\n");
}
```

3. Notice that we are reporting the determinant in *outputTextArea* since it is not a matrix. Execute your applet and give the *determinantButton* a whirl.

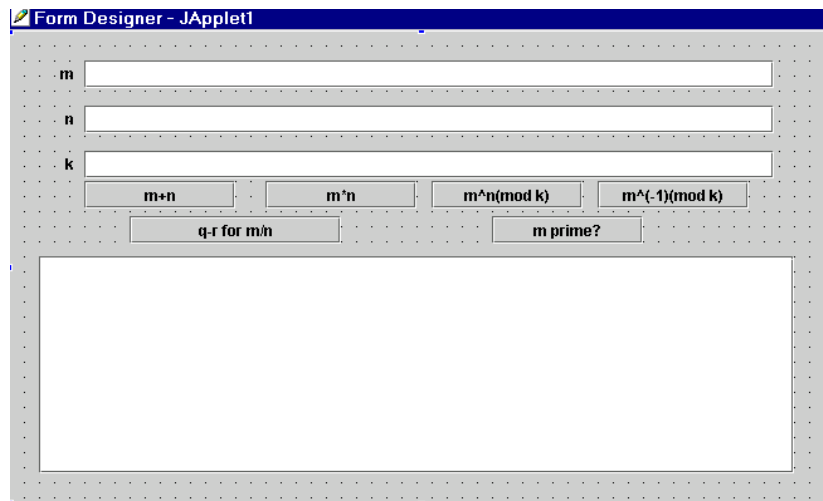
Exercise 3: Add a *transpose* method to the *Matrix* class and add a corresponding button to the applet.

The BigInteger Class

In the *java.math* package is a class called *BigInteger*. It was provided mainly for cryptographic capabilities, but it is ideal for doing Number Theory. *BigInteger* objects represent arbitrary-precision integers. They provide the same kind of operations as regular integers, but they never overflow. In addition, the *BigInteger* class provides methods for primality-testing, for finding the greatest common divisor, and some methods for doing modular arithmetic.

Setting up the User Interface

1. Start a new project in the usual manner, choosing the JFCApplet template.
2. We are going to be working with some big integers and we want some *JTextFields* to hold them so make your Form Designer wide and fairly tall.
3. Add three *JTextFields* to the top of the Form Designer one right under the other. Edit the *Name* properties on the Property Lists so that they are called, respectively, *mTextField*, *nTextField*, and *kTextField*.
4. Put *JLabels* next to each one to identify them. Label them, respectively, as “m”, “n”, and “k”.
5. Add a wide *JScrollPane* to the bottom of the Form Designer. Put a *JTextArea* on the *JScrollPane* and call it *outputTextArea*. To prevent our big numbers from running off the side of *outputTextArea* set the *LineWrap* property to *true*. Leave room between *outputTextArea* and the *JTextFields* to put some buttons. (See the picture below.)



6. Add 6 *JButtons* as indicated on the picture above. Call the first one *addButton* and label it “m+n”. Call the second one *multiplyButton* and label it “m*n”. Call the third one *modPowButton* and label it “mⁿ(mod k)”. Call the fourth one *modInverseButton* and label it “m⁽⁻¹⁾(mod k)”. Call the fifth one *qrButton* and label it “q-r for m/n”. Finally, call the sixth one *primeButton* and label it “m prime?”.
7. In the applet add *actionPerformed* event-handlers for each of the 6 buttons.

Making the Buttons Work

1. We will need to define *BigInteger* objects to correspond to each of m , n , and k . First we will need to import the *java.math* package. Add the following import statement to the top of the source code for the applet.

```
import java.math.*;
```

2. Add declarations for m , n , and k at the beginning of the outer block of code for the applet by inserting the following statement.

```
BigInteger m,n,k;
```

3. Now we want to make each Button do a computation. Let's begin with *addButton*. Insert the following code into the *addButton_actionPerformed* method:

```
m = new BigInteger(mTextField.getText());
n = new BigInteger(nTextField.getText());
BigInteger sum = m.add(n);
outputTextArea.append("m+n = " + sum.toString() + "\n\n");
```

4. The first two lines create new *BigInteger* objects based on the text in *mTextField* and *nTextField* and assign them to m and n respectively. (These have the potential of throwing a *NumberFormatException* if the String in one of the TextFields is not valid.) The second line calls the *add* method of the *BigInteger* object m to add it to n and return a new *BigInteger* object representing the sum. We assign that to sum . Finally we append it to *outputTextArea*. Execute your applet. Type some large integers into *mTextField* and *nTextField* and press *addButton* to see it work.

Exercise 1: Use the *multiply* method of the *BigInteger* class to write the corresponding code for the *multiplyButton* then test it out with some large integers.

5. Now let's put in the appropriate code for *modPowButton*. As its label suggests it should compute $m^n \pmod k$. Insert the following code into the *modPowButton_actionPerformed* method:

```
m = new BigInteger(mTextField.getText());
n = new BigInteger(nTextField.getText());
k = new BigInteger(kTextField.getText());

BigInteger modPow = m.modPow(n,k);
outputTextArea.append("m^n (mod k) = " + modPow.toString() + "\n\n");
```

6. The only thing new here is the call to the *modPow* method of the *BigInteger* class m . Notice that it accepts two *BigInteger* objects as parameters and that the first one is the exponent and the second one is the modulus. Go ahead and try it out.

Exercise 2: Use the *modInverse* method of the *BigInteger* class to set your applet up so that pressing the *modInverseButton* will compute the inverse of m modulo n . In other words it will compute the number r such that $mr \equiv 1 \pmod n$. (Hint: the *modInverse* method accepts one *BigInteger* object as a parameter and that object serves as the modulus.)

7. Now let's set up *primeButton*. Insert the following into the *primeButton_actionPerformed* method:

```
m = new BigInteger(mTextField.getText());
```

```
String s;

if ( m.isProbablePrime(20) ) s = "m is almost certainly prime.\n\n";
else s = "m is definitely not prime.\n\n";

outputTextArea.append(s);
```

8. Notice that the *isProbablePrime* method accepts an integer as a parameter. Because the primality test can only test to see if the number is probably prime, the integer determines the certainty of the result. If the test determines that the number is not prime then it is definitely not prime and it returns *false*. However, if the test determines that it is probably prime then it is prime with a probability of $1 - \frac{1}{2^r}$ where *r* is the integer that is passed as a parameter. In our case we are using *r*=20 which means that a return of *true* means that the number is prime with a probability of $1 - \frac{1}{2^{20}} = .999999046326$. Higher values give more certainty (and require more computation time.) Test it out on some small numbers that you can check and with some large numbers.
9. Finally, let's set up the *qrButton*. Insert the following into the *qrButton_actionPerformed* method:

```
m = new BigInteger(mTextField.getText());
n = new BigInteger(nTextField.getText());

BigInteger[] qr = m.divideAndRemainder(n);
outputTextArea.append("quotient of m/n = " + qr[0].toString() + "\n");
outputTextArea.append("remainder of m/n = " + qr[1].toString()+"\n\n");
```

10. Notice that the *divideAndRemainder* method of *BigInteger* returns a 2-element array of *BigInteger* objects. The first element of the array (the one at index 0) is the quotient and the second one is the remainder. (**Warning:** The *divideAndRemainder* method doesn't really handle negative integers the way that it should from a Number Theory perspective. If one of the numbers is negative and the other is positive both the quotient and the remainder will be negative. If it applied the Division Algorithm appropriately the remainder would always be positive.) Try it out.
11. The *math* package also contains a *BigDecimal* class that allows you to do decimal computations of arbitrary precision.