Plotting Engineering and Scientific Data using Java

Baldwin shows you how write a generalized plotting program that can be used to plot engineering and scientific data produced by any object that implements a very simple interface.

Published: December 17, 2002
By Richard G. Baldwin

Java Programming Notes # 1468

Preface

Excellent language for engineering computations

Because of its platform independence, Java provides an excellent programming language for engineering and scientific computational experiments, particularly where extreme execution speed is not a requirement. Programs developed for such experiments on one platform can be successfully executed on a variety of platforms without the need to rewrite or recompile.

A large Math library

Furthermore, because if its inherent simplicity, and the availability of a large **Math** library, Java provides an excellent programming language for engineers and scientists who want to do their own programming, but who have no desire to become programming experts. The code required to conduct an engineering or scientific computational experiment often consists of little more than the most rudimentary application of arithmetic in loops using data stored in arrays or read from disk files.

Now for the bad news

However, there is a downside to this happy story. When doing this sort of work, it is often very important to see the results of the experiments in the form of graphs or plots. Unfortunately, the programming required to produce graphical output from simple engineering and scientific computational experiments cannot be accomplished using rudimentary programming techniques. Rather, to do that job right requires considerable expertise in Java programming.

A generalized plotting program

This lesson develops a generalized plotting program, which is easy to connect to other programs, *(whether they are simple or complex)*, in order to display the output from those programs in two-dimensional Cartesian coordinates. The plotting program is specifically designed to be useful to persons having very little knowledge of Java programming.

> *(Actually, the lesson develops two very similar plotting programs each designed to display the data in a different format.)*

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

Figure 1 shows a typical display produced by one of the plotting programs that I will develop in this lesson. *(The other program superimposes all of the curves on the same set of axes instead of spacing them vertically as shown in Figure 1.)*

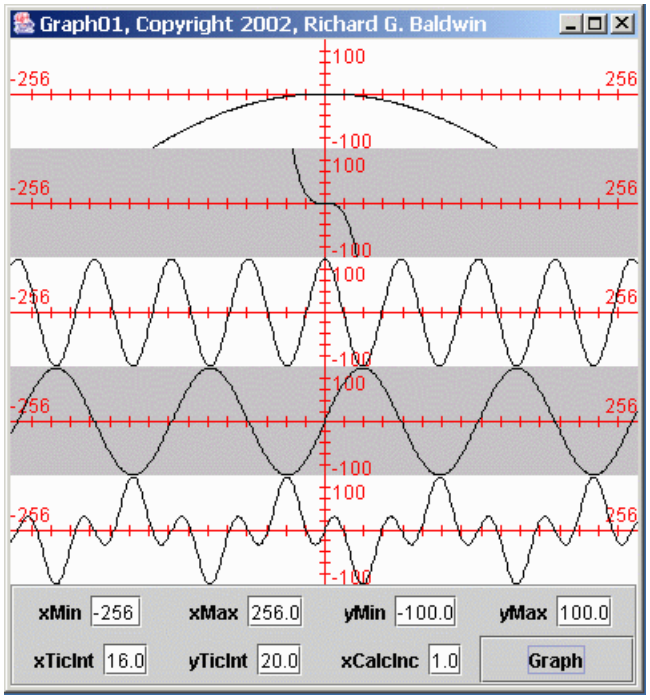

Figure 1 Sample Display

While the plotting program itself is quite complex, the code required to produce the data to be plotted can be very simple. For example, because of the use of the Java **Math** library, only fourteen lines of simple Java code were required to produce the data plotted in Figure 1.

The Graph01Demo program

The data displayed in Figure 1 was produced by a program named **Graph01Demo**. A listing of that program is shown in Listing 37 near the end of the lesson. I will explain that program in detail shortly.

A more substantive example

In addition, I will provide and discuss another sample program, which produces and plots data having considerably more engineering and scientific significance than the data shown in Figure 1. *(This will be a digital signal processing (DSP) example).* Even in that case, you will see that the program that produces the data is much less complex than the program used to plot the data.

Using the plotting program with your data

During the course of this lesson, I will explain everything that you will need to know to cause the output from your own engineering and scientific programs to be displayed by the plotting program.

The Graph01 program

The graphical display of the data shown in Figure 1 was produced by my generalized plotting program named **Graph01**. As you will see later, this is a long and fairly complex program.

A listing of the plotting program is shown in Listing 39 near the end of the lesson.

User needn't understand the plotting program

Fortunately, the user of the plotting program doesn't need to understand anything about the code that makes up the plotting program. All the user needs to understand is the interface to the program, which I will explain later.

However, for those of you who may be interested, I will also discuss and explain the plotting program later in this lesson.

Plotting format

As you can see in Figure 1, the plotting program allows for plotting up to five independent functions stacked vertically, each with the same vertical and horizontal axes. This vertical stacking format makes it easy to compare up to five plots at the same points on the horizontal axes.

If you need more than five functions, the number of functions can easily be increased with a few minor changes to the program.

> *(I will also provide, but will not discuss, another version of the program, named Graph02, which superimposes up to five plots on the same coordinate system. In some cases, that is a more useful form of display. You will find a complete listing of this program in Listing 40 near the end of the lesson.)*

Plotting parameters

As you can also see in Figure 1, a set of text fields and a button on the bottom of the frame make it possible for the user to modify the plotting parameters and to re-plot the same data with an entirely new set of plotting parameters.

> *(It is often true that important but subtle pieces of information can only be exposed by viewing the same data with different sets of plotting parameters.)*

Same data, different parameters

Figure 2 shows the same data as in Figure 1, but plotted with a different set of plotting parameters.

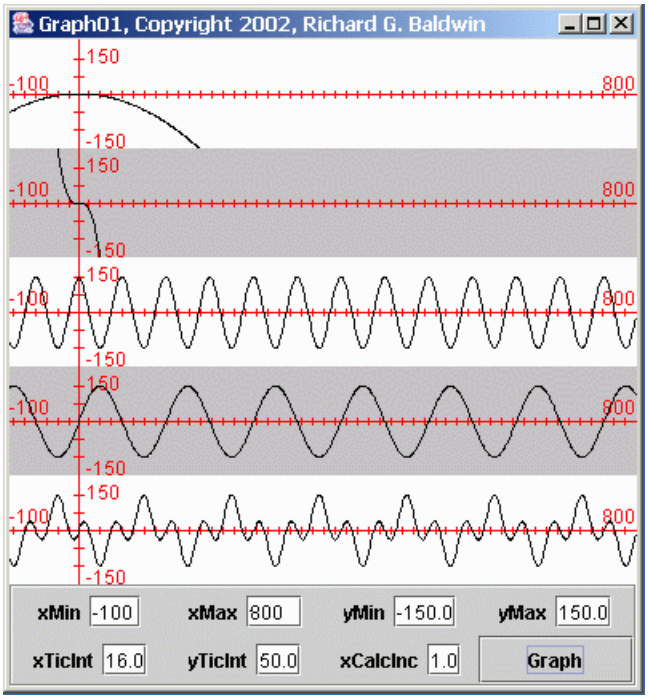

Figure 2 Sample Display for Same Data with Different Plotting Parameters

In the case of Figure 2, the origin was moved to the left, the total expanse of the horizontal axis was increased, and the space between the tic marks on the vertical axis was increased from 20 units to 50 units.

> *(It will help you to see the differences if you will position two browser windows side-by-side while viewing one display in one browser window and viewing the other display in the other browser window.)*

Discussion and Sample Code

Testing the plotting program

I am assuming that you have accomplished the minimal steps required to get the Java SDK that is available from Sun up and running.

To run the plotting program named **Graph01** in self-test mode, do the following:

- Copy the code in Listing 39 into a file named **Graph01.java**.
- Copy the code in Listing 1 into a file named **GraphIntfc.java**, and put that file in the same directory as the file named **Graph01.java** above.
- Compile the program named **Graph01.java** using the Java SDK. *(Note, you must be using SDK version 1.4 or later.)*

At this point, you should be able to execute the program named **Graph01** in self-test mode by entering the following command at the command prompt in the same directory where you compiled the program:

java Graph01

If everything has been done correctly up to this point, the display shown in Figure 4 should appear on your screen.

Using the plotting program

To use the plotting program with your own data generator program, do the following:

- Still working in the same directory, define and compile a data generator class that implements the interface named **GraphIntfc01**, shown in Listing 1.
- Start the plotting program named **Graph01** running by following the instructions that I will provide below.

Plotting your data using Graph01

Assume that your data-generator class is named **MyData**, and that you have successfully compiled it in the same directory as the compiled version of **Graph01**.

The next step is to enter the following command at the command prompt in the same directory. *(Note that this command differs from the command given earlier. This command provides the name of your class as a command-line parameter following the name of the plotting program.)*

java Graph01 MyData

When you do this, the plotting program should start pulling the necessary data from your data-generator program and plotting that data in the format shown in Figure 1.

Modifying plotting parameters

Once all the curves have been plotted, you can change any of the plotting parameter values in the text fields at the bottom of the display and press the button labeled **Graph**. When you press the button, the plotting program will re-plot your data using the new plotting parameters.

The plotting parameters

Here is the meaning of the plotting-parameter text fields shown in Figure 1:

- xMin and xMax - The values of the left and right ends of all horizontal axes.

- yMin and yMax - The values of the bottom and top of the vertical axis in each plotting area. *(Note that the different plotting areas are identified by alternating white and gray backgrounds.)*
- xTicInt - The distance between tic marks on the x-axis.
- yTicInt - The distance between tic marks on the y-axis.
- xCalcInc - The distance between the points on the x-axis where values for y are computed. *(Unless your data-generator program is taking too long to run, you should probably leave this set to 1.0 in order to get the best quality plots.)*

The labels on the axes

Each x-axis has a label at the left end and the right end. Similarly, each y-axis has a label at the bottom and the top. These labels represent the values at the extreme ends of the axes. For example in Figure 2, the label 800 appears at the right end of each x-axis. This is value of the x-axis where the axis intersects the border of the frame.

Keep the pixels in mind

When adjusting the plotting parameters, keep in mind that the total width of each of the plotting areas is slightly less than 400 pixels.

> *(You can easily increase this to full screen width by changing one value in the **Graph01** program and recompiling the program).*

While you can theoretically make the horizontal expanse of the x-axes as wide as you wish, because of the pixel limitation, you cannot see details that require a resolution of more than 400 points along the x-axis *(unless you modify the program as described above).*

The interface named GraphIntfc01

Regardless of its simplicity or its complexity, there are only two requirements for your data-generator program to operate successfully with the plotting program named **Graph01**:

1. It must implement the interface named **GraphIntfc01**.
2. It must have a constructor that doesn't require any parameters *(the default constructor will satisfy that requirement if you don't need another constructor).*

Implementing GraphIntfc01

All that is required to implement the interface is to define a class that provides a concrete definition for the six methods declared in Listing 1.

```
public interface GraphIntfc01{
  public int getNmbr();
  public double f1(double x);
  public double f2(double x);
  public double f3(double x);
```

```
  public double f4(double x);
  public double f5(double x);
}//end GraphIntfc01

Listing 1
```

The getNmbr method

On several occasions, I have stated that the plotting program can plot up to five functions. However, it doesn't have to plot all five functions. The plotting program can be used to plot any number of functions from one to five.

The method named **getNmbr** must return an integer value between 1 and 5 that specifies the number of functions to be plotted. The plotting program uses that value to divide the total plotting surface into the specified number of plotting areas, and plots each of the functions named **f1** through **fn** in one of those plotting areas.

The methods named f1, f2, f3, f4, and f5

As you can see in Listing 1, each of these methods receives a **double** value as an incoming parameter and returns a **double** value. In essence, each of these methods receives a value for x and returns the corresponding value for y.

One plotting area per method

Each of these methods provides the data to be plotted in one plotting area. The method named **f1** provides the data for the top plotting area, the method named **f2** provides the data for the first plotting area down from the top, and so forth.

> *(For example, if the **getNmbr** method returns a value of 4, the method named **f5** will never be invoked. If **getNmbr** returns 5, the method named **f5** will be invoked to provide the data for the bottom plotting area.)*

How does it work?

Each plotting area contains a horizontal axis. The plotting program moves across the horizontal axis in each plotting area one step at a time *(moving in incremental steps equal to the plotting parameter named xCalcInc).*

At each step along the way, the plotting program invokes the method associated with that plotting area, *(f1, f2, etc.),* passing the horizontal position as a parameter to the method.

The value returned by the method is assumed to be the vertical value associated with that horizontal position, and that is the vertical value that is plotted for that horizontal position.

Doesn't know and doesn't care

The plotting program doesn't know, and doesn't care how the method decides on the value to return for each value that it receives as an incoming parameter. The plotting program simply invokes the methods to get the data, and then plots the data.

Computed *"on the fly"*

For example, the returned values could be computed and returned *"on the fly,"* as is the case in the example named **Graph01Demo**, which we will look at shortly.

Returned from an array

On the other hand, the values could have been computed earlier and saved in an array, as will be the case in the DSP example that we will look at later.

From a disk file, a database, the internet, etc.

The returned values could be read from a disk file, obtained from a database on another computer, or obtained from any other source such as another computer on the internet.

All that matters is that when the plotting program invokes one of the five methods named **f1** through **f5**, passing a **double** value as a parameter, it expects to receive a **double** value as a return value, and it will plot the value that it receives.

It is up to you

It is up to you, the author of the data-generator program, to decide how you will implement the methods named **f1** through **f5**. In some cases, the implementation may be simple. In other cases, the implementation may be more complex. The first case that we will examine is very simple. A subsequent case involving DSP is not so simple.

The class named Graph01Demo

Although this is a very simple class definition, I am going to break it up and discuss it in fragments in order to help you focus your attention on the important points. A complete listing of the class definition is shown in Listing 37 near the end of the lesson.

Defining data-generator classes

This class is used to demonstrate how to write data-generator classes that will operate successfully with the program named **Graph01**.

> *(Figure 1 shows the display of the data produced by this class. You might want to refer to that figure while examining the code in this class.)*

Listing 2 shows the beginning of the class definition, which names the class, and specifies that the class implements the interface named **GraphIntfc01**.

```
class Graph01Demo
                implements
GraphIntfc01{

Listing 2
```

The number of functions to plot

Listing 3 shows the entire listing of the method named **getNmbr**.

```
  public int getNmbr(){
    return 5;
  }//end getNmbr

Listing 3
```

Recall from above that this method must return an integer value between 1 and 5, which tells the plotting program how many functions to plot.

This demonstration plots all five functions, as shown in Figure 1, so this method returns the value 5.

The topmost plotting area

Listing 4 shows the entire method named **f1**, whose output is plotted in the topmost plotting area of the display in Figure 1. *(This is the area at the top with the white background.)*

```
  public double f1(double x){
    return -(x*x)/200.0;
  }//end f1

Listing 4
```

This method receives an incoming parameter known locally as **x**. *(In all five methods defined in this class, the computations are performed on the fly.)* The method computes and returns the negative square of the incoming parameter *(divided by 200).* This produces the inverted bowl shape at the top of Figure 1.

The top-most plotting area with a gray background

The curve plotted in the top-most plotting area with the gray background in Figure 1 is produced by the method named **f2**, shown in Listing 5.

```
  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

Listing 5
```

As before, this function receives an incoming parameter known locally as **x**. The function computes and returns the negative cube of the incoming parameter *(divided by 200).* This produces the curve shown in the top-most gray area of Figure 1.

The middle white plotting area

The method named **f3**, shown in Listing 6, produces the curve shown in the center plotting area with the white background in Figure 1.

```
  public double f3(double x){
    return 100*Math.cos(x/10.0);
  }//end f3

Listing 6
```

This is a simple cosine curve, which is computed on the fly. Each time the method is invoked, the incoming parameter named **x**, is used to calculate the cosine of an angle in radians given by one-tenth the value of **x**. The cosine of that angle is multiplied by 100 and returned.

> *(Note that the cosine of the angle is computed using a static method of the standard Java class named **Math**. This is the class that contains the Java math library.)*

The bottom-most gray plotting area

The curve shown in the bottom-most gray plotting area of Figure 1 is produced by the method named **f4**, shown in Listing 7.

```
  public double f4(double x){
    return 100*Math.sin(x/20.0);
  }//end f4

Listing 7
```

The body of **f4** is similar to the body of **f3**, except that **f4** computes and returns sine values instead of cosine values. Also, the value of **x** is used differently so that the period of the curve produced by **f4** is twice the period of the curve produced by **f3**.

The bottom white plotting area

Finally, the bottom white plotting area in Figure 1 shows the output produced by the method named **f5**, shown in Listing 8.

```
  public double f5(double x){
    return 100*(Math.sin(x/20.0)
               *Math.cos(x/10.0));
  }//end f5
```

```
}//end sample class Graph01Demo

Listing 8
```

This method computes and returns the product of sine and cosine functions identical to those discussed above.

The end of the class definition

Listing 8 also shows the closing curly brace that signifies the end of the class definition for the class named **Graph01Demo**.

That's all you need to know

That's really all that you need to know to be able to make effective use of the generalized plotting program named **Graph01**. If you can define the methods named **f1** through **f5**, which will return the required values for your computational experiment, then you can make use of this program to plot your data.

A more substantive example

However, lest you go away believing that this is all too trivial to be interesting, I am going to show you another example that is far from trivial. In the next example, I will demonstrate two of the most important operations in the field commonly referred to as digital signal processing, or DSP for short.

Because many of you are unlikely to be familiar with the techniques and terminology involved, the discussion will of necessity be fairly shallow. However, I do want to show at least one example of how you can perform substantive computational experiments using this approach.

A DSP example

A DSP example showing convolution filtering and spectral analysis is shown in Figure 3.

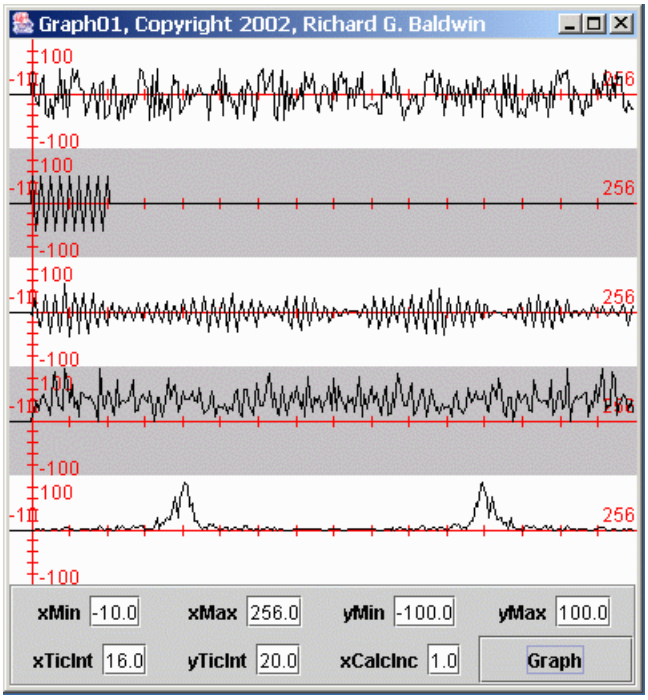

Figure 3 A Digital Signal Processing (DSP) Example

In the field of DSP, the five individual plots shown in the plotting areas of Figure 3 are commonly referred to as traces. I will use that terminology in this discussion.

White random noise

The top trace in the area with the white background shows about 256 samples of white random noise. When we get to the code, we will see that this data was created using a Java pseudo-random number generator.

A convolution filter

The second trace from the top shows a 33-point narrow-band convolution filter, which is simply a chunk taken out of a sinusoid whose frequency is one-fourth the sampling frequency. In other words, the sinusoid is represented by four samples per cycle.

The convolution filter output

The middle trace shows the result of applying the narrow-band convolution filter to the white noise. The output from the convolution process was amplified to bring it back into an appropriate amplitude range for visual analysis.

If you compare the middle trace with the top trace, you will notice that much of the high-frequency energy and much of the low-frequency energy has been removed. Most of the energy in the middle trace appears to be about the same frequency as the frequency of the convolution filter *(which is what we would expect).*

Time-domain vs. frequency-domain

The top three traces represent information in the time domain. The bottom two traces represent information in the frequency domain.

> *(Think of the frequency domain as the information that is visible on many audio systems, consisting of parallel vertical bars with lights that dance up and down. These lights are often associated with a device referred to as a frequency equalizer. When the music contains a lot of drums, or other sounds at the bass end, the lights at the low (usually left) end of the frequency spectrum are very active. When the music contains a lot of symbols, or sounds at the treble end, the lights at the high (right) end of the frequency spectrum are very active. That is a form of real-time spectrum analysis.)*

Frequency spectrum analysis

The two bottom traces in Figure 3 result from performing frequency spectrum analysis on the top trace and the middle trace respectively.

The white noise spectrum

The trace in the gray area immediately below the center is an estimate of the spectral distribution of the white noise in the top trace. The spectrum analysis was performed across the frequency range from zero frequency to the sampling frequency.

While not perfectly flat, as would be the case for perfectly *white* noise, you can see that the energy appears to be distributed across that entire range.

> *(If we wanted to improve our estimate, we could capture and analyze a much longer sample of the white noise.)*

If you examine this trace carefully, you might notice that there is a point of near symmetry in the middle. The values that you see above that point are a mirror-image of the values that you see below that point. *(I will have more to say about this later.)*

The filtered noise spectrum

The bottom trace shows an estimate of the spectral distribution of the filtered noise in the center trace. Again, the spectrum analysis was performed across the frequency range from zero frequency to the sampling frequency. Again also, there is a symmetry point in the middle with everything to the right of that point being a mirror image of everything to the left of that point.

Two spectral peaks are visible

Unlike the spectral analysis of the white noise, this spectral analysis shows two obvious peaks. One peak appears at one-fourth the sampling frequency, and the other peak appears at three-fourths the sampling frequency.

In other words, as we concluded from examining the center trace, the filtering process removed much of the energy above and below the design frequency of the convolution filter.

> *(By changing the design frequency of the convolution filter, and repeating the process, we could move this peak up or down along the frequency axis.)*

What does the symmetry mean?

Without getting into a lot of detail at this point, the point of symmetry that I identified above is known as the Nyquist folding frequency.

In order to be able to identify the frequency of a sine wave, you must have at least two samples per cycle of the sine wave. The Nyquist folding frequency is the frequency at which you have exactly two samples per cycle.

As the frequency of the sine wave continues to increase beyond that point, without a corresponding change in the sampling frequency, it is impossible to determine from the samples so obtained whether the frequency is increasing or decreasing.

An ambiguity in the spectrum analysis

As a result, the spectrum analysis process was unable to determine if the peak in the frequency spectrum was below or above the folding frequency. Thus, the bottom trace in Figure 3 shows two peaks which are mirror images of one another with the folding frequency being half way between the two peaks.

> *(As a practical matter, when doing spectrum analysis, there is no point in computing the values above the folding frequency. I did that here just to illustrate that there is a folding frequency, which is equal to one-half the sampling frequency.)*

Let's see some code

The class used to produce the data displayed in Figure 3 is named **Dsp002**. A complete listing of this class definition is shown in Listing 38 near the end of the lesson.

I will break this class up into fragments and briefly discuss it to show how you can define significant classes and easily connect them to the generalized plotting program named **Graph01**.

As before, having compiled the class named Dsp002, you would exercise it by entering the following at a command prompt:

java Graph01 Dsp002

Different from the previous example class

This class differs from the class named **Graph01Demo** in one very significant way. In that class, all the values returned by the methods named **f1** through **f5** were computed on the fly as the methods were called.

In this new class named **Dsp002**, all the data is generated and stored in array objects when an object of the class named **Dsp002** is instantiated. When the methods named **f1** through **f5** are invoked later, they simply retrieve the data from the array objects and return that data to the plotting program.

Basic operation of the program

As mentioned earlier, this program applies a narrow-band convolution filter to white noise, and then computes the amplitude spectrum of the filtered noise using a Discrete Fourier Transform (DFT) algorithm. The spectrum of the white noise is also computed. All of the processing occurs when an object of the class is instantiated, and the processed results are saved in arrays.

The input noise, the filter, the filtered output, and the two spectra are deposited in five arrays for later retrieval and display. The data in the five arrays are returned by the methods named f1, f2, f3, f4, and f5 respectively.

The values that are returned by the methods are scaled for appropriate display in the plotting areas provided by the program named **Graph01**.

Data and filter lengths

The code in Listing 9 establishes the data lengths for the white noise, the convolution filter, the filtered output, and the spectrum.

```
class Dsp002 implements GraphIntfc01{
  int operatorLen = 33;
  int dataLen = 256+operatorLen;
  int outputLen =
                 dataLen -
operatorLen;
  int spectrumPts = outputLen;

< operatorLen;

cnt++){
      //Note, the value of the
      // denominator in the argument
      // to the cos method specifies
      // the frequency relative to the
      // sampling frequency.
      operator[cnt] = Math.cos(
```

```
cnt*2*Math.PI/4);
    }//end for loop

    //Apply the operator to the data
    Convolve01.convolve(data,dataLen,

operator,operatorLen,output);

    //Compute DFT of the raw data and
    // save it in spectrumA array.
    Dft01.dft(data,spectrumPts,

spectrumA);

    //Compute DFT of the filtered data
    // and save it in spectrumB array.
    Dft01.dft(output,spectrumPts,

spectrumB);
    //All of the data has now been
    // produced and saved.  It may be
    // retrieved by invoking the
    // following methods named f1
    // through f5.

  }//end constructor

  //--------------------------------
//
  //The following six methods are
  // required by the interface named
  // GraphIntfc01.
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 5;
  }//end getNmbr
  //--------------------------------
//
  public double f1(double x){
    int index = (int)Math.round(x);
    //This version of this method
    // returns the random noise data.
    // Be careful to stay within the
    // array bounds.
    if(index < 0 ||
                 index >
```

Listing 9

Create data arrays

The code in Listing 10 creates the array objects that will be used to store the data until it is retrieved by the methods named **f1** through **f5**.

```
  double[] data = new double[dataLen];
  double[] operator =
                new
double[operatorLen];
  double[] output =
                  new
double[outputLen];
  double[] spectrumA =
                new
double[spectrumPts];
  double[] spectrumB =
                new
double[spectrumPts];
```

Listing 10

Generate and save the white noise

Most of the hard work is done by the constructor or by methods called by the constructor.

The code in Listing 11 generates and saves the white noise in the array object named **data**.

```
  public Dsp002(){//constructor
    Random generator = new Random(
                  new
Date().getTime());
    for(int cnt=0;cnt < data.length;

cnt++){
      //Get data, scale it, remove the
      // dc offset, and save it.
      data[cnt] = 100*generator.
                        nextDouble()-
50;
    }//end for loop
```

Listing 11

The random noise generator seed

Note that by virtue of the way this white noise is being generated, a different seed is passed to the constructor for the **Random** class each time an object of the **Dsp002** class is instantiated. Thus, each new object presents different random noise.

> *(In some cases, this may not be desirable and it may be preferable to use the same seed each time an object is instantiated.)*

Create the convolution operator

The code in Listing 12 creates the 33-point convolution operator, as a segment of a cosine wave, and saves it in the designated array.

```
    for(int cnt = 0; cnt <
operatorLen;

cnt++){
      operator[cnt] = Math.cos(

cnt*2*Math.PI/4);
    }//end for loop

Listing 12
```

Note that the constant value of 4 in the denominator of the argument to the **cos** method specifies the frequency of the cosine wave relative to the sampling frequency. *(In this case, the frequency of the cosine wave is one-fourth the sampling frequency.)*

Apply the convolution operator

The code in Listing 13 invokes a static method named **convolve** in a class named **Convolve01** to apply the convolution operator to the white noise and save the filtered result in the appropriate array. I will briefly discuss this method later.

```
    Convolve01.convolve(data,dataLen,

operator,operatorLen,output);

Listing 13
```

Compute the spectrum of the two traces

The code in Listing 14 invokes a static method named **dft** of a class named **Dft01** twice in succession to compute the spectra for the white noise and the filtered noise, and to save those spectra in the appropriate arrays.

```
    Dft01.dft(data,spectrumPts,

spectrumA);

    Dft01.dft(output,spectrumPts,

spectrumB);
  }//end constructor

Listing 14
```

All results have been computed and saved

That is the end of the constructor. At this point, all the results have been computed and saved in the appropriate arrays for later retrieval by the methods named **f1** through **f5**.

The object is simply setting in memory waiting to have its encapsulated data retrieved and plotted.

The getNmbr method

The **getNmbr** method for this class is exactly the same as for the class discussed earlier. As before, it returns the integer value 5, telling the plotting program that there are five plots to be generated.

```
  public int getNmbr(){
    return 5;
  }//end getNmbr

Listing 15
```

The method named f1

The method named **f1** is representative of all five of the methods named **f1** through **f5** in this class. Therefore, I will discuss only the first of the five methods in detail.

```
  public double f1(double x){
    int index = (int)Math.round(x);

    if(index < 0 ||
                index > data.length-
1){
      return 0;
    }else{
      return data[index];
    }//end else
  }//end f1

Listing 16
```

In all five cases, the purpose of the method is to fetch and return a value from an array, where the incoming parameter will be converted to an array index.

Convert incoming parameter to an index

The incoming parameter is received as type **double**. However, an array must be indexed using type **int**. The first statement in the method invokes the **round** method of the **Math** class to convert the **double** value to the nearest integer. That integer will be used as an array index.

Stay within array bounds

Following this, the method applies some logic to confirm that the index value is within the bounds of the array. If not, the method returns the value 0.

If the index is within the array bounds, the method retrieves and returns the value stored at that index location in the array.

And that's all there is to it.

Methods f2 through f5

Except for the scale factors applied to the data before returning it, the behavior of the methods named **f2** through **f5**, is essentially the same as the behavior of the method named **f1**. In each case, the method retrieves, scales, and returns a value previously stored in an array. Therefore, I won't discuss these other methods. You can view them in Listing 38 near the end of the lesson.

And that ends the definition of the class named **Dsp002**.

The class named Convolve01

The entire class named **Convolve01** is shown in Listing 17.

If you already understand convolution, you will probably find the code in this class straightforward. If not, the code will probably still be straightforward, but the reason for the code may be obscure.

```
class Convolve01{
  public static void convolve(
                      double[] data,
                      int dataLen,
                      double[]
operator,
                      int operatorLen,
                      double[] output){
    //Apply the operator to the data,
    // dealing with the index
    // reversal required by
    // convolution.
    for(int i=0;
          i < dataLen-
operatorLen;i++){
      output[i] = 0;
      for(int j=operatorLen-1;j>=0;
                                    j--
){
        output[i] +=

data[i+j]*operator[j];
      }//end inner loop
    }//end outer loop
  }//end convolve method
}//end Class Convolve01

Listing 17
```

Making a long story short

To make a long story short, the class named **Convolve01** provides a static method named **convolve**, which applies an incoming convolution operator to an incoming set of data and deposits the filtered data in an output array whose reference is received as an incoming parameter.

This class could easily be broken out and put in a library as a stand-alone class, or the **convolve** method could be added to a class containing a variety of DSP methods.

The discrete Fourier transform (DFT)

The entire class named **Dft01** is shown in Listing 18.

As with convolution, if you already understand the discrete Fourier transform, you will probably find the code in this class to be straightforward. If not, the code will probably still be straightforward, but the reasons for the code may be obscure. Since the purpose of this lesson is not to explain digital signal processing concepts, I won't attempt to provide a detailed explanation for the code in this method.

```
class Dft01{
  public static void dft(
                    double[] data,
                    int dataLen,
                    double[]
spectrum){
    //Set the frequency increment to
    // the reciprocal of the data
    // length.  This is convenience
    // only, and is not a requirement
    // of the DFT algorithm.
    double delF = 1.0/dataLen;
    //Outer loop iterates on frequency
    // values.
    for(int i=0; i < dataLen;i++){
      double freq = i*delF;
      double real = 0;
      double imag = 0;
      //Inner loop iterates on time-
      // series points.
      for(int j=0; j < dataLen; j++){
        real += data[j]*Math.cos(

2*Math.PI*freq*j);
        imag += data[j]*Math.sin(

2*Math.PI*freq*j);
        spectrum[i] = Math.sqrt(
                real*real +
imag*imag);
      }//end inner loop
    }//end outer loop
```

```
  }//end dft

Listing 18
```

Brief explanation

Once again, to make a long story short, this class provides a static method named **dft**, which computes and returns the amplitude spectrum of an incoming time series.

The amplitude spectrum is computed as the square root of the sum of the squares of the real and imaginary parts.

A DFT algorithm can compute any number of points in the frequency domain. In this case, the number of points computed in the frequency domain is equal to the number of samples in the incoming time series, which is a fairly common practice.

The method deposits the frequency data in an array whose reference is received as an incoming parameter.

As with convolution, this class could easily be broken out and put in a library as a stand-alone class, or the **dft** method could be added to a class containing a variety of DSP methods.

The plotting programs

Now that you have examined the examples, some of you may be interested in an explanation of the plotting program itself.

If you are interested only in how to use the plotting programs, and are not interested in the details of the plotting programs, skip ahead to the section entitled Run the Program.

If you are interested in learning how the plotting programs do what they do, keep reading.

Two plotting programs

Two very similar plotting programs are shown in the listings near the end of the lesson. The program named **Graph01**, shown in Listing 39, can be used to plot as many as five separate functions, each in its own plotting area. Examples of the display produce by this program are shown in Figures 1, 2, 3, and 4. I will briefly discuss this program in the paragraphs that follow.

The program named **Graph02**, shown in Listing 40, can also be used to plot as many as five separate functions. In this case, however, the graphs produced by the functions are superimposed in the same plotting area. This is simply an alternative display format. I won't discuss any of the particulars of this program, but if you understand the program named **Graph01**, you will have no difficulty understanding this program named **Graph02** well.

The program named Graph01

This program is designed to access a class file that implements the interface named **GraphIntfc01**, and to plot up to five functions defined in that class file.

The methods in the class corresponding to the functions to be plotted are named **f1**, **f2**, **f3**, **f4**, and **f5**.

As you learned in the earlier discussion, the class containing the functions must also define a static method named **getNmbr**. This method takes no parameters and returns the number of functions to be plotted. If this method returns a value greater than 5, a **NoSuchMethodException** will be thrown.

Separate plotting areas

The overall plotting surface is divided into the required number of equally sized plotting areas. One function is plotted on Cartesian coordinates in each plotting area.

A noarg constructor is required

The constructor for the class that implements **GraphIntfc01** must not require any parameters. This is because the **newInstance** method of the **Class** class is used to instantiate an object, based on a **String** provided as a command-line parameter. The **newInstance** method can only create objects using a noarg constructor.

Some methods may not be invoked

If the **getNmbr** method returns a value less than 5, then the methods that will not be invoked begin with **f5** and work down toward **f1**. For example, if the value returned by **getNmbr** is 3, then the program will invoke the methods named **f1**, **f2**, and **f3**. While the methods named **f4** and **f5** must exist in order to satisfy the interface, they won't be invoked. Therefore, it doesn't matter what those methods return as long as it is type **double**.

The visual appearance

As shown in Figure 1, the plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A Cartesian coordinate system with axes, tic marks, and labels is drawn in red in each plotting area.

The Cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

A test class

The program also compiles a test class named **junk**, which contains the five required methods plus the method named **getNmbr**. This makes it easy to compile and test the program in a stand-alone mode.

Usage instructions

At runtime, the name of the class that implements the **GraphIntfc01** interface must be provided as a command-line parameter.

If the command-line parameter is missing, the program instantiates an object from the internal test class named **junk** and plots the data provided by that object. Thus, you can test the program by running it with no command-line parameter. This will produce the display shown in Figure 4.

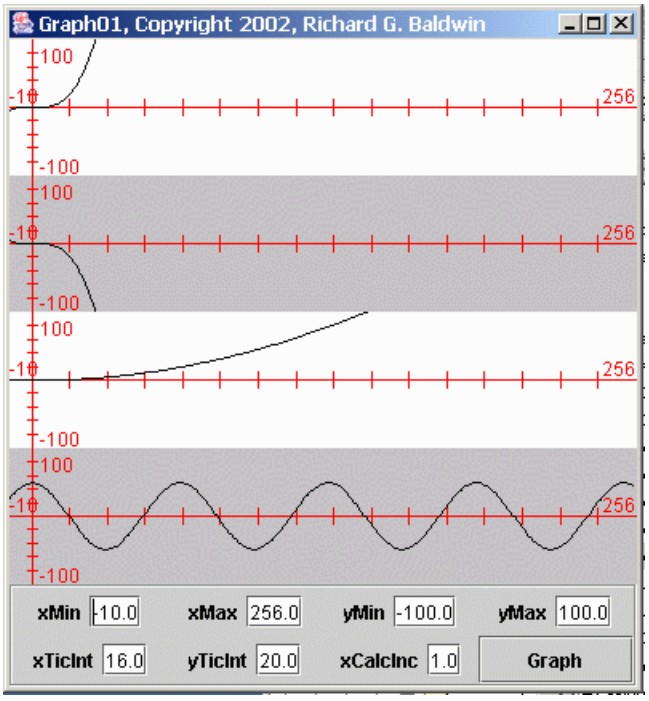

Figure 4 Graphic Display for Self-Test Class

If the command-line parameter is provided, the program instantiates an object of the class whose name matches the parameter, and plots the data provided by that object.

Plotting parameters

This program provides the following text fields for user input, along with a button labeled **Graph**. This allows the user to adjust the parameters and re-plot the graph as many times as needed with as many different plotting scales as may be needed:

- xMin = minimum x-axis value
- xMax = maximum x-axis value
- yMin = minimum y-axis value

- yMax = maximum y-axis value
- xTicInt = tic interval on x-axis
- yTicInt = tic interval on y-axis
- xCalcInc = calculation interval

The user can modify any of these parameters and then press the **Graph** button to cause the five functions to be re-plotted according to the new parameters.

A new object

Whenever the **Graph** button is pressed, the event handler for that button instantiates a new object of the class that implements the **GraphIntfc01** interface.

Depending on the nature of that class, this may be redundant in some cases. However, it is useful in those cases where it is necessary to refresh the values of instance variables defined in the class (such as a counter, for example).

> *(I will show you how to eliminate this feature from the plotting program if you decide that it is unnecessary for your data.)*

Requires Java SDK 1.4 or later

This program uses constants that were first defined in the **Color** class of v1.4.0. Therefore, the program requires v1.4.0 or later to compile and execute correctly.

Will discuss in fragments

I will discuss this program in fragments. As mentioned earlier, a complete listing of the program is provided in Listing 39 near the end of the lesson. You should be able to copy and paste that code into your Java source file, and then compile and execute it successfully.

The class named Graph01

The entire class, including the **main** method is shown in Listing 19.

```
class Graph01{
  public static void main(
           String[] args)
           throws
NoSuchMethodException,

ClassNotFoundException,

InstantiationException,

IllegalAccessException{
    if(args.length == 1){
      //pass command-line parameter
```

```
      new GUI(args[0]);
    }else{
      //no command-line parameter
given
      new GUI(null);
    }//end else
  }// end main
}//end class Graph01 definition

Listing 19
```

The primary purpose of **main** method is to instantiate an object of the class named **GUI**.

In addition, the **main** method checks to see if the user provided a command-line parameter, and if so, passes it along to the constructor for the **GUI** class.

The class named GUI

The beginning of the **GUI** class is shown in Listing 20.

```
class GUI extends JFrame
              implements
ActionListener{

  //Define plotting parameters and
  // their default values.
  double xMin = -10.0;
  double xMax = 256.0;
  double yMin = -100.0;
  double yMax = 100.0;

  //Tic mark intervals
  double xTicInt = 16.0;
  double yTicInt = 20.0;

  //Tic mark lengths.  If too small
  // on x-axis, a default value is
  // used later.
  double xTicLen = (yMax-yMin)/50;
  double yTicLen = (xMax-xMin)/50;

  //Calculation interval along x-axis
  double xCalcInc = 1.0;

  //Text fields for plotting
parameters
  JTextField xMinTxt =
             new JTextField("" +
xMin);
  JTextField xMaxTxt =
             new JTextField("" +
xMax);
  JTextField yMinTxt =
```

```
              new JTextField("" +
yMin);
  JTextField yMaxTxt =
              new JTextField("" +
yMax);
  JTextField xTicIntTxt =
           new JTextField("" +
xTicInt);
  JTextField yTicIntTxt =
           new JTextField("" +
yTicInt);
  JTextField xCalcIncTxt =
          new JTextField("" +
xCalcInc);

  //Panels to contain a label and a
  // text field
  JPanel pan0 = new JPanel();
  JPanel pan1 = new JPanel();
  JPanel pan2 = new JPanel();
  JPanel pan3 = new JPanel();
  JPanel pan4 = new JPanel();
  JPanel pan5 = new JPanel();
  JPanel pan6 = new JPanel();

  //Misc instance variables
  int frmWidth = 400;
  int frmHeight = 430;
  int width;
  int height;
  int number;
  GraphIntfc01 data;
  String args = null;

  //Plots are drawn on the canvases
  // in this array.
  Canvas[] canvases;
```

Listing 20

The code in Listing 20 declares and in some cases initializes several instance variables that are required later to support the plotting process. The comments and the names of the variables generally indicate the purpose of those variables.

The constructor for the GUI class

The beginning of the constructor for the GUI class is shown in Listing 21.

```
  GUI(String args)throws
                NoSuchMethodException,

ClassNotFoundException,
```

```
InstantiationException,

IllegalAccessException{

    if(args != null){
      //Save for use later in the
      // ActionEvent handler
      this.args = args;
      //Instantiate an object of the
      // target class using the String
      // name of the class.
      data = (GraphIntfc01)

Class.forName(args).

newInstance();
    }else{
      //Instantiate an object of the
      // test class named junk.
      data = new junk();
    }//end else
```

Listing 21

The main purpose of the code in Listing 21 is to instantiated the object that will provide the data to be plotted. If the user provided the name of a class as a command-line argument, an attempt will be made to create a **newInstance** of that class.

> *(In case you are unfamiliar with this approach, this is one way to create an object of a class whose name is specified as a **String** at runtime.)*

Otherwise, the code in Listing 21 will instantiate an object of the test class named **junk** *(to be discussed later).*

Array to hold Canvas objects

Each of the separate plotting areas in Figure 1 is an object of a class that extends the **Canvas** class. The code in Listing 22 invokes the **getNmbr** method on the new object to determine how many functions are to be plotted. Then it creates an array object to hold the requisite number of **Canvas** objects.

```
    //Create array to hold correct
    // number of Canvas objects.
    canvases =
              new
Canvas[data.getNmbr()];

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
```

```
      throw new NoSuchMethodException(
                  "Too many functions.
"
                    + "Only 5
allowed.");
    }//end if

Listing 22
```

Although the limit could easily be increased, this program is currently limited to plotting the output from five functions. The code in Listing 22 checks this limit and throws an exception if an attempt is made to plot more than five functions.

Routine GUI construction code

Although somewhat long and rather tedious, the code in Listing 23 is completely straightforward. This code continues with the construction of the GUI object, creating text fields, a button, etc.

```
    //Create the control panel and
    // give it a border for cosmetics.
    JPanel ctlPnl = new JPanel();
    ctlPnl.setLayout(//?rows x 4 cols
                     new
GridLayout(0,4));
    ctlPnl.setBorder(
                      new
EtchedBorder());

    //Button for replotting the graph
    JButton graphBtn =
                     new
JButton("Graph");
    graphBtn.addActionListener(this);

    //Populate each panel with a label
    // and a text field.  Will place
    // these panels in a grid on the
    // control panel later.
    pan0.add(new JLabel("xMin"));
    pan0.add(xMinTxt);

    pan1.add(new JLabel("xMax"));
    pan1.add(xMaxTxt);

    pan2.add(new JLabel("yMin"));
    pan2.add(yMinTxt);

    pan3.add(new JLabel("yMax"));
    pan3.add(yMaxTxt);

    pan4.add(new JLabel("xTicInt"));
    pan4.add(xTicIntTxt);
```

```
    pan5.add(new JLabel("yTicInt"));
    pan5.add(yTicIntTxt);

    pan6.add(new JLabel("xCalcInc"));
    pan6.add(xCalcIncTxt);

    //Add the populated panels and the
    // button to the control panel
with
    // a grid layout.
    ctlPnl.add(pan0);
    ctlPnl.add(pan1);
    ctlPnl.add(pan2);
    ctlPnl.add(pan3);
    ctlPnl.add(pan4);
    ctlPnl.add(pan5);
    ctlPnl.add(pan6);
    ctlPnl.add(graphBtn);
```

Listing 23

Because of the routine nature of the code in Listing 23, I will let the comments suffice as an explanation.

The Canvas objects

If you refer back to Figure 1, you will see that from one to five **Canvas** objects are stacked vertically in the center of a frame.

This is accomplished by placing a **JPanel** object in the center of the frame, and setting the layout manager on the **JPanel** to **GridLayout**. The grid is defined as having one column and an unspecified number of rows. Then one **Canvas** object is placed in each cell of the grid, beginning at the top and working downward from the top, until the required number of **Canvas** objects have been placed in the grid.

This is accomplished by the code in Listing 24.

```
    //Create a panel to contain the
    // Canvas objects.  They will be
    // displayed in a one-column grid.
    JPanel canvasPanel = new JPanel();
    canvasPanel.setLayout(//?rows,1
col
                    new
GridLayout(0,1));

    //Create a custom Canvas object
for
    // each function to be plotted and
    // add them to the one-column
grid.
```

```
    // Make background colors
alternate
    // between white and gray.
    for(int cnt = 0;
                  cnt < number;
cnt++){
      switch(cnt){
        case 0 :
          canvases[cnt] =
                       new
MyCanvas(cnt);
          canvases[cnt].setBackground(

Color.WHITE);
          break;
        case 1 :
          canvases[cnt] =
                       new
MyCanvas(cnt);
          canvases[cnt].setBackground(

Color.LIGHT_GRAY);
          break;
        case 2 :
          canvases[cnt] =
                       new
MyCanvas(cnt);
          canvases[cnt].setBackground(

Color.WHITE);
          break;
        case 3 :
          canvases[cnt] =
                       new
MyCanvas(cnt);
          canvases[cnt].setBackground(

Color.LIGHT_GRAY);
          break;
        case 4 :
          canvases[cnt] =
                       new
MyCanvas(cnt);
          canvases[cnt].

setBackground(Color.WHITE);
      }//end switch
      //Add the object to the grid.
      canvasPanel.add(canvases[cnt]);
    }//end for loop

Listing 24
```

The code in Listing 24:

- Creates the **JPanel** object, and sets its layout property to **GridLayout**.
- Creates the requisite number of objects of the **MyCanvas** class *(which extends Canvas)*, setting the background colors of the panels alternately to white and gray.
- Adds the **MyCanvas** objects to the cells in the grid. *(Note that the constructor for each **MyCanvas** object receives an integer that specifies its position in the stack of **MyCanvas** objects. We will see how that information is used later.)*

More routine construction code

The code in Listing 25 is simply more routine code required to:

- Finish the construction of the **GUI** object
- Set its location and size on the screen
- Make it visible

Once again, I will let the comments serve as the explanation for this code.

```
    //Add the sub-assemblies to the
    // frame.  Set its location, size,
    // and title, and make it visible.
    getContentPane().

add(ctlPnl,"South");
    getContentPane().

add(canvasPanel,"Center");

    setBounds(0,0,frmWidth,frmHeight);
    setTitle("Graph01, " +
                "Copyright 2002, " +
                "Richard G.
Baldwin");
    setVisible(true);

    //Set to exit on X-button click
    setDefaultCloseOperation(

EXIT_ON_CLOSE);

    //Get and save the size of the
    // plotting surface
    width = canvases[0].getWidth();
    height = canvases[0].getHeight();
```

Listing 25

Force a repaint

As you will see later, the actual plotting behavior of this program is defined by the code in an overridden version of the **paint** method in the **MyCanvas** class. I will discuss that code in some detail later.

One way to cause the code in the overridden **paint** method is to invoke the **repaint** method on a reference to a **MyCanvas** object.

The code in Listing 26 invokes the **repaint** method on each **MyCanvas** object in sequence, to guarantee that they are properly painted when the **GUI** object first becomes visible.

```
    for(int cnt = 0;
                    cnt < number;
cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end constructor

Listing 26
```

Similar code will be used again later to cause the graphs to be repainted each time the user presses the **Graph** button in the bottom right corner of Figure 1.

End of the constructor

The code in Listing 26 also ends the constructor for the **GUI** object. When the constructor finishes execution, the GUI appears on the screen with all plotting areas properly painted.

Re-plotting the data

Listing 27 shows the beginning of the event handler that is registered on the button to cause the functions to be re-plotted.

```
  public void actionPerformed(
                      ActionEvent
evt){
    //Re-instantiate the object that
    // provides the data
    try{
      if(args != null){
        data = (GraphIntfc01)Class.

forName(args).newInstance();
      }else{
        data = new junk();
      }//end else
    }catch(Exception e){
      //Known to be safe at this
point.
      // Otherwise would have aborted
      // earlier.
    }//end catch

Listing 27
```

The purpose of the event handler is to cause the functions to be re-plotted after the user changes the plotting parameters.

A new object of the target class

However, the code in Listing 27 goes beyond that. In particular, the code in Listing 27 creates a new object from which to get the data that is to be plotted.

In some cases, this may be required, depending on the nature of the class from which that object is instantiated. In other cases, it may not be necessary, and could slow down the re-plotting process.

If your class doesn't contain counters or other variables that need to be re-initialized whenever you re-plot, you could probably safely remove or disable the code in Listing 27. This will make the program run faster, although my may not be able to see the difference.

The rest of the event handler

The remaining code in the event handler is shown in Listing 28.

```
    //Set plotting parameters using
    // data from the text fields.
    xMin = Double.parseDouble(

xMinTxt.getText());
    xMax = Double.parseDouble(

xMaxTxt.getText());
    yMin = Double.parseDouble(

yMinTxt.getText());
    yMax = Double.parseDouble(

yMaxTxt.getText());
    xTicInt = Double.parseDouble(

xTicIntTxt.getText());
    yTicInt = Double.parseDouble(

yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(

xCalcIncTxt.getText());

    //Calculate new values for the
    // length of the tic marks on the
    // axes.  If too small on x-axis,
    // a default value is used later.
    xTicLen = (yMax-yMin)/50;
    yTicLen = (xMax-xMin)/50;

    //Repaint the plotting areas
```

```
    for(int cnt = 0;
                    cnt < number;
cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end actionPerformed

Listing 28
```

This code is very straightforward. It performs the following actions:

- Get new plotting parameters from the text fields.
- Perform some calculations.
- Cause each of the **MyCanvas** objects to be repainted using the new plotting parameters.

Again, I will let the comments provide any necessary explanations.

That brings us to the most interesting part of the program, the extended **Canvas** class.

The class named MyCanvas

The class named **MyCanvas** is an inner class of the **GUI** class, which is used to override the **paint** method in each of the plotting areas shown in Figure 1.

> *(Virtually all graphics operations in Java, other than those involving standard GUI components, are implemented by overriding the **paint** method on an object.)*

The beginning of this class definition is shown in Listing 29.

```
class MyCanvas extends Canvas{
  int cnt;//object number
  //Factors to convert from double
  // values to integer pixel
locations.
  double xScale;
  double yScale;

  MyCanvas(int cnt){//save obj number
    this.cnt = cnt;
  }//end constructor

Listing 29
```

Floating data vs. pixels

Most of the calculations in this program are performed on data of type **double**. However, graphics operations are ultimately performed in terms integer numbers of pixels. The code in

Listing 29 declares scale factors used later to convert from **double** values to integer pixel locations.

A simple constructor

The code in Listing 29 also defines the constructor, whose only purpose is to save an integer identifying the position of this object in the vertical stack of **MyCanvas** objects.

The overridden paint method

The beginning of the overridden **paint** method for the **MyCanvas** class is shown in Listing 30.

```
  //Override the paint method
  public void paint(Graphics g){
    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-
xMin)*xScale),
                (int)((0-
yMin)*yScale));
    drawAxes(g);//Draw the axes
    g.setColor(Color.BLACK);
```

Listing 30

The code in Listing 30:

- Calculates and saves the scale factors for converting from **double** coordinate values to integer values in pixels.
- Moves the plotting origin to the correct location.
- Invokes a method to draw the axes *(in red)* on the **MyCanvas** object.
- Sets the color to black for the remainder of the plotting activity on the object.

Get old values

The plotting process consists of drawing a straight line segment between two points. One of the points is defined by a pair of old coordinate values. The other point is defined by a pair of new coordinate values.

The code in Listing 31 initializes the beginning point for the plot. The initial value for the x-coordinate is the left edge of the plotting area.

```
    //Get initial data values
    double xVal = xMin;
    int oldX = getTheX(xVal);
```

```
    int oldY = 0;

    //Use the Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
      case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
      case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
      case 2 :
        oldY = getTheY(data.f3(xVal));
        break;
      case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
      case 4 :
        oldY = getTheY(data.f5(xVal));
    }//end switch
```

Listing 31

The initial y-coordinate value

The initial value for the y-coordinate depends on which function is being plotted on the **MyCanvas** object. Recall that each **MyCanvas** object contains an instance variable that identifies its position in the vertical stack of **MyCanvas** objects. The **switch** statement in Listing 31 uses that information to invoke one of the five methods named **f1** through **f5**. This gets the correct value for the y-coordinate based on the value of the x-coordinate.

The methods named **getTheX** and **getTheY** called by the code in Listing 31 convert the coordinate values from type **double** to integer values in pixels.

The method named **getTheY** also changes the sign on the data so that positive y-values go up the screen rather than down the screen.

> *(By default, positive vertical coordinate values go down the screen from top to bottom in Java.)*

Plot the points

The remainder of the overridden **paint** method is shown in Listing 32.

```
    //Now loop and plot the points
    while(xVal < xMax){
      int yVal = 0;
      //Get next data value.  Use the
      // Canvas obj number to
```

```
      // determine which method to
      // invoke to get the value for
y.
      switch(cnt){
        case 0 :
          yVal =

getTheY(data.f1(xVal));
          break;
        case 1 :
          yVal =

getTheY(data.f2(xVal));
          break;
        case 2 :
          yVal =

getTheY(data.f3(xVal));
          break;
        case 3 :
          yVal =

getTheY(data.f4(xVal));
          break;
        case 4 :
          yVal =

getTheY(data.f5(xVal));
      }//end switch1

      //Convert the x-value to an int
      // and draw the next line
segment
      int x = getTheX(xVal);
      g.drawLine(oldX,oldY,x,yVal);

      //Increment along the x-axis
      xVal += xCalcInc;

      //Save end point to use as start
      // point for next line segment.
      oldX = x;
      oldY = yVal;
    }//end while loop

  }//end overridden paint method
```

Listing 32

The code in Listing 32 is relatively straightforward. This code simply iterates from the minimum x-value to the maximum x-value, invoking the appropriate method *(from **f1** through **f5**)* to get the new y values. In the process, it invokes the **drawLine** method of the **Graphics** class to connect the points.

The drawAxes method

As it turns out, it is more difficult to draw and label the axes with tic marks than it is to plot the actual data.

The code to accomplish this is shown in Listing 33.

```
  void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Label left x-axis and bottom
    // y-axis.  These are the easy
    // ones.  Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                 getTheX(xMin),
                 getTheY(xTicLen/2)-
2);
    g.drawString("" + (int)yMin,

getTheX(yTicLen/2)+2,

getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis.  These are the hard
    // ones because the position must
    // be adjusted by the font size
and
    // the number of characters.
    //Get the width of the string for
    // right end of x-axis and the
    // height of the string for top of
    // y-axis
    //Create a string that is an
    // integer representation of the
    // label for the right end of the
    // x-axis.  Then get a character
    // array that represents the
    // string.
    int xMaxInt = (int)xMax;
    String xMaxStr = "" + xMaxInt;
    char[] array = xMaxStr.

toCharArray();

    //Get a FontMetrics object that
can
    // be used to get the size of the
    // string in pixels.
    FontMetrics fontMetrics =

g.getFontMetrics();
```

```
    //Get a bounding rectangle for the
    // string
    Rectangle2D r2d =

fontMetrics.getStringBounds(

array,0,array.length,g);
    //Get the width and the height of
    // the bounding rectangle.  The
    // width is the width of the label
    // at the right end of the
    // x-axis.  The height applies to
    // all the labels, but is needed
    // specifically for the label at
    // the top end of the y-axis.
    int labWidth =

(int)(r2d.getWidth());
    int labHeight =

(int)(r2d.getHeight());

    //Label the positive x-axis and
the
    // positive y-axis using the width
    // and height from above to
    // position the labels.  These
    // labels apply to the very ends
of
    // the axes at the edge of the
    // plotting surface.
    g.drawString("" + (int)xMax,
                 getTheX(xMax)-
labWidth,
                  getTheY(xTicLen/2)-
2);
    g.drawString("" + (int)yMax,
              getTheX(yTicLen/2)+2,

getTheY(yMax)+labHeight);

    //Draw the axes
    g.drawLine(getTheX(xMin),
                          getTheY(0.0),

getTheX(xMax),

getTheY(0.0));

    g.drawLine(getTheX(0.0),
                         getTheY(yMin),
                         getTheX(0.0),

getTheY(yMax));

    //Draw the tic marks on axes
```

```
    xTics(g);
    yTics(g);
  }//end drawAxes

Listing 33
```

The code in Listing 33 is fairly complex, particularly with respect to putting the labels on the ends of the axes. However, I doubt that many of you are interested in the details, so I will let the comments suffice to explain the code.

Drawing tic marks

Listing 34 shows the methods invoked from the code in Listing 33 to actually draw the tic marks on the axes.

```
  //Method to draw tic marks on x-axis
  void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
                    getTheY(-
xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small.  In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
      topEnd = 5;
      bottomEnd = -5;
    }//end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
      x = getTheX(xDoub);

g.drawLine(x,topEnd,x,bottomEnd);
      xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis
moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
      x = getTheX(xDoub);
```

```
g.drawLine(x,topEnd,x,bottomEnd);
      xDoub -= xTicInt;
    }//end while

  }//end xTics
  //-------------------------------
//

  //Method to draw tic marks on y-axis
  void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
      y = getTheY(yDoub);

g.drawLine(rightEnd,y,leftEnd,y);
      yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis
moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
      y = getTheY(yDoub);

g.drawLine(rightEnd,y,leftEnd,y);
      yDoub -= yTicInt;
    }//end while

  }//end yTics
```

Listing 34

Again, I am going to let the comments suffice to explain this code.

The getTheX and getTheY methods

As mentioned earlier, methods named **getTheX** and **getTheY** are used to convert coordinate values from type **double** to integer values in pixels. Those two methods are shown in Listing 35.

```
  //This method translates and scales
  // a double y value to plot properly
  // in the integer coordinate system.
  // In addition to scaling, it causes
  // the positive direction of the
```

```
  // y-axis to be from bottom to top.
  int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
  }//end getTheY
  //---------------------------------//

  //This method scales a double x value
  // to plot properly in the integer
  // coordinate system.
  int getTheX(double x){
    return (int)(x*xScale);
  }//end getTheX
  //---------------------------------//

}//end inner class MyCanvas
//===================================//

}//end class GUI
```

Listing 35

Listing 35 also marks the end of the inner class named **MyCanvas** and the end of the class named **GUI**.

The test class named junk

Listing 36 defines a test class named **junk** that implements the interface named **GraphIntfc01**.

```
class junk implements GraphIntfc01{
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 4;
  }//end getNmbr

  public double f1(double x){
    return (x*x*x)/200.0;
  }//end f1

  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
    return (x*x)/200.0;
  }//end f3

  public double f4(double x){
    return 50*Math.cos(x/10.0);
  }//end f4

  public double f5(double x){
```

```
    return 100*Math.sin(x/20.0);
  }//end f5

}//end sample class junk
```

Listing 36

This class defines the methods declared in the interface, and makes it possible to test the plotting program in a stand-alone mode without having access to another class that implements the interface.

Since I discussed the implementation of this interface in some detail earlier in the lesson, there should be no need for me to provide further discussion of the code in Listing 36. You might note, however, that since the method named **getNmbr** returns the value 4, the method named **f5** will not be invoked by the plotting program.

Run the Program

Copy the code for the plotting program from Listing 39 into a Java source file named Graph01.java.

Copy the code for the interface from Listing 1 into a Java source file named **GraphIntfc01.java**.

Compile and run the program named **Graph01** with no command-line parameters. This should use the internal test class named **junk** discussed earlier to produce the display shown in Figure 4.

Once you have the display on your screen, make changes to the plotting parameters in the text fields at the bottom and press the button labeled **Graph**. When you do, you should see the same functions being re-plotted with different plotting parameters.

Once that is successful, copy the code in Listing 37 into a file named **Graph01Demo.java**. Copy the code in Listing 38 into a file named **Dsp002.java**.

Compile these two files. Rerun the plotting program named **Graph01** providing **Graph01Demo** as a command-line parameter. Also rerun the plotting program providing **Dsp002** as a command-line parameter. This should produce displays similar to Figures 1 and 3.

Remember, however, that you must be running Java version 1.4 or later to successfully compile and execute this program.

Summary

I provided two generalized plotting programs in this lesson. One of the programs plots up to five functions in a vertical stack. The other program superimposes the plots for up to five functions on the same Cartesian coordinate system.

Each of these programs is capable of plotting the data produced by any object that implements a simple interface named **GraphIntfc01**.

I explained the interface named **GraphIntfc01.** I also explained how you can define classes of your own that implement the interface making them suitable for being plotted using either of the plotting programs.

I also provided two different sample classes that implement the interface for you to use as models as you come up to speed in defining your own classes.

Complete Program Listings

Complete listings of the programs discussed in this lesson are shown below.

```
/* File Graph01Demo.java
Copyright 2002, R.G.Baldwin

This class is used to demonstrate how
to write data-generator classes that
will operate successfully with the
program named Graph01.

Tested using JDK 1.4.0 under Win 2000.
**************************************/
class Graph01Demo
                implements GraphIntfc01{
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 5;
  }//end getNmbr

  public double f1(double x){
    //This is a simple x-squared
    // function with a negative
    // sign.
    return -(x*x)/200.0;
  }//end f1

  public double f2(double x){
    //This is a simple x-cubed
    // function
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
    //This is a simple cosine
    // function
    return 100*Math.cos(x/10.0);
  }//end f3

  public double f4(double x){
```

```
    //This is a simple sine
    // function
    return 100*Math.sin(x/20.0);
  }//end f4

  public double f5(double x){
    //This is function which
    // returns the product of
    // the above sine and cosines.
    return 100*(Math.sin(x/20.0)
                    *Math.cos(x/10.0));
  }//end f5

}//end sample class Graph01Demo
```

Listing 37

```
/* File Dsp002.java
Copyright 2002, R.G.Baldwin

Note:  This program requires access to
the interface named GraphIntfc01.

This is a sample DSP program whose
output is designed to be plotted
by the programs named Graph01 and
Graph02.  This requires that the
class implement GraphIntfc01.  It
also requires a noarg constructor.

This program applies a narrow-band
convolution filter to white noise, and
then computes the amplitude spectrum of
the filtered result using a simple
Discrete Fourier Transform (DFT)
algorithm.  The spectrum of the white
noise is also computed.

The program convolves a 33-point
sinusoidal convolution filter with
wide-band noise, and then computes the
amplitude spectrum of the raw data and
the filtered result.  The processing
occurs when an object of the class is
instantiated.

The input noise, the filter, the
filtered output, and the two spectra
are deposited in five arrays for
later retrieval and display.

The input noise, the filter, the
filtered output, the spectrum of the
```

```
noise, and the spectrum of the filtered
result are returned by the methods
named f1, f2, f3, f4, and f5
respectively.

The output values that are returned
are scaled for appropriate display in
the plotting areas provided by the
program named Graph01.

Tested using JDK 1.4.0 under Win 2000.
***********************************/
import java.util.*;

class Dsp002 implements GraphIntfc01{
  //Establish data and spectrum
  // lengths.
  int operatorLen = 33;
  int dataLen = 256+operatorLen;
  int outputLen =
                dataLen - operatorLen;
  int spectrumPts = outputLen;

  //Create arrays for the data and
  // the results.
  double[] data = new double[dataLen];
  double[] operator =
              new double[operatorLen];
  double[] output =
                new double[outputLen];
  double[] spectrumA =
              new double[spectrumPts];
  double[] spectrumB =
              new double[spectrumPts];

  public Dsp002(){//constructor
    //Generate and save some wide-band
    // random noise.  Seed with a
    // different value each time the
    // object is constructed.
    Random generator = new Random(
                new Date().getTime());
    for(int cnt=0;cnt < data.length;
                                  cnt++){
      //Get data, scale it, remove the
      // dc offset, and save it.
      data[cnt] = 100*generator.
                        nextDouble()-50;
    }//end for loop

    //Create a convolution operator and
    // save it in the array.
    for(int cnt = 0; cnt < operatorLen;
                                  cnt++){
      //Note, the value of the
      // denominator in the argument
```

```
      // to the cos method specifies
      // the frequency relative to the
      // sampling frequency.
      operator[cnt] = Math.cos(
                      cnt*2*Math.PI/4);
    }//end for loop

    //Apply the operator to the data
    Convolve01.convolve(data,dataLen,
          operator,operatorLen,output);

    //Compute DFT of the raw data and
    // save it in spectrumA array.
    Dft01.dft(data,spectrumPts,
                           spectrumA);

    //Compute DFT of the filtered data
    // and save it in spectrumB array.
    Dft01.dft(output,spectrumPts,
                           spectrumB);
    //All of the data has now been
    // produced and saved.  It may be
    // retrieved by invoking the
    // following methods named f1
    // through f5.

  }//end constructor

  //--------------------------------//
  //The following six methods are
  // required by the interface named
  // GraphIntfc01.
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 5;
  }//end getNmbr
  //--------------------------------//
  public double f1(double x){
    int index = (int)Math.round(x);
    //This version of this method
    // returns the random noise data.
    // Be careful to stay within the
    // array bounds.
    if(index < 0 ||
                index > data.length-1){
      return 0;
    }else{
      return data[index];
    }//end else
  }//end f1
  //--------------------------------//
  public double f2(double x){
    //Return the convolution operator
    int index = (int)Math.round(x);
    if(index < 0 ||
```

```
            index > operator.length-1){
      return 0;
    }else{
      //Scale for good visibility in
      // the plot
      return operator[index] * 50;
    }//end else
  }//end f2
  //---------------------------------//
  public double f3(double x){
    //Return filtered output
    int index = (int)Math.round(x);
    if(index < 0 ||
              index > output.length-1){
      return 0;
    }else{
      //Scale to approx same p-p as
      // input data
      return output[index]/6;
    }//end else
  }//end f3
  //---------------------------------//
  public double f4(double x){
    //Return spectrum of raw data
    int index = (int)Math.round(x);
    if(index < 0 ||
            index > spectrumA.length-1){
      return 0;
    }else{
      //Scale for good visibility in
      // the plot.
      return spectrumA[index]/10;
    }//end else
  }//end f4
  //---------------------------------//
  public double f5(double x){
    //Return the spectrum of the
    // filtered data.
    int index = (int)Math.round(x);
    if(index < 0 ||
            index > spectrumB.length-1){
      return 0;
    }else{
      //Scale for good visibility in
      // the plot.
      return spectrumB[index]/100;
    }//end else
  }//end f5

}//end sample class Dsp002
//===================================//

//This class provides a static method
// named convolve, which applies an
// incoming convolution operator to
// an incoming set of data and deposits
```

```
// the filtered data in an output
// array whose reference is received
// as an incoming parameter.
//This class could easily be broken out
// and put in a library as a stand-
// alone class, or the convolve method
// could be added to a class containing
// a variety of DSP methods.
class Convolve01{
  public static void convolve(
                     double[] data,
                     int dataLen,
                     double[] operator,
                     int operatorLen,
                     double[] output){
    //Apply the operator to the data,
    // dealing with the index
    // reversal required by
    // convolution.
    for(int i=0;
          i < dataLen-operatorLen;i++){
      output[i] = 0;
      for(int j=operatorLen-1;j>=0;
                                    j--){
        output[i] +=
                  data[i+j]*operator[j];
      }//end inner loop
    }//end outer loop
  }//end convolve method
}//end Class Convolve01
//===================================//

//This class provides a static method
// named dft, which computes and
// returns the amplitude spectrum of
// an incoming time series.  The
// amplitude spectrum is computed as
// the square root of the sum of the
// squares of the real and imaginary
// parts.
//Returns a number of points in the
// frequency domain equal to the number
// of samples in the incoming time
// series.  Deposits the frequency
// data in an array whose reference is
// received as an incoming parameter.
//This class could easily be broken out
// and put in a library as a stand-
// alone class, or the dft method
// could be added to a class containing
// a variety of DSP methods.
class Dft01{
  public static void dft(
                    double[] data,
                    int dataLen,
                    double[] spectrum){
```

```
    //Set the frequency increment to
    // the reciprocal of the data
    // length.  This is convenience
    // only, and is not a requirement
    // of the DFT algorithm.
    double delF = 1.0/dataLen;
    //Outer loop iterates on frequency
    // values.
    for(int i=0; i < dataLen;i++){
      double freq = i*delF;
      double real = 0;
      double imag = 0;
      //Inner loop iterates on time-
      // series points.
      for(int j=0; j < dataLen; j++){
        real += data[j]*Math.cos(
                      2*Math.PI*freq*j);
        imag += data[j]*Math.sin(
                      2*Math.PI*freq*j);
        spectrum[i] = Math.sqrt(
                real*real + imag*imag);
      }//end inner loop
    }//end outer loop
  }//end dft

}//end Dft01
```

Listing 38

```
/* File Graph01.java
Copyright 2002, R.G.Baldwin

Note:  This program requires access to
the interface named GraphIntfc01.

This is a plotting program.  It is
designed to access a class file, which
implements GraphIntfc01, and to plot up
to five functions defined in that class
file.  The plotting surface is divided
into the required number of equally
sized plotting areas, and one function
is plotted on Cartesian coordinates in
each area.

The methods corresponding to the
functions are named f1, f2, f3, f4,
and f5.

The class containing the functions must
also define a static method named
getNmbr(), which takes no parameters
and returns the number of functions to
```

be plotted. If this method returns a value greater than 5, a NoSuchMethodException will be thrown.

Note that the constructor for the class that implements GraphIntfc01 must not require any parameters due to the use of the newInstance method of the Class class to instantiate an object of that class.

If the number of functions is less than 5, then the absent method names must begin with f5 and work down toward f1. For example, if the number of functions is 3, then the program will expect to call methods named f1, f2, and f3. It is OK for the absent methods to be defined in the class. They simply won't be invoked.

The plotting areas have alternating white and gray backgrounds to make them easy to separate visually.

All curves are plotted in black. A Cartesian coordinate system with axes, tic marks, and labels is drawn in red in each plotting area.

The Cartesian coordinate system in each plotting area has the same horizontal and vertical scale, as well as the same tic marks and labels on the axes.

The labels displayed on the axes, correspond to the values of the extreme edges of the plotting area.

The program also compiles a sample class named junk, which contains five methods and the method named getNmbr. This makes it easy to compile and test this program in a stand-alone mode.

At runtime, the name of the class that implements the GraphIntfc01 interface must be provided as a command-line parameter. If this parameter is missing, the program instantiates an object from the internal class named junk and plots the data provided by that class. Thus, you can test the program by running it with no command-line parameter.

```
This program provides the following
text fields for user input, along with
a button labeled Graph.  This allows
the user to adjust the parameters and
replot the graph as many times with as
many plotting scales as needed:

xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these
parameters and then click the Graph
button to cause the five functions
to be re-plotted according to the
new parameters.

Whenever the Graph button is clicked,
the event handler instantiates a new
object of the class that implements
the GraphIntfc01 interface.  Depending
on the nature of that class, this may
be redundant in some cases.  However,
it is useful in those cases where it
is necessary to refresh the values of
instance variables defined in the
class (such as a counter, for example).

Tested using JDK 1.4.0 under Win 2000.

This program uses constants that were
first defined in the Color class of
v1.4.0.  Therefore, the program
requires v1.4.0 or later to compile and
run correctly.
***********************************/

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph01{
  public static void main(
           String[] args)
           throws NoSuchMethodException,
                  ClassNotFoundException,
                  InstantiationException,
                  IllegalAccessException{
    if(args.length == 1){
      //pass command-line parameter
```

```
      new GUI(args[0]);
    }else{
      //no command-line parameter given
      new GUI(null);
    }//end else
  }// end main
}//end class Graph01 definition
//===================================//

class GUI extends JFrame
             implements ActionListener{

  //Define plotting parameters and
  // their default values.
  double xMin = -10.0;
  double xMax = 256.0;
  double yMin = -100.0;
  double yMax = 100.0;

  //Tic mark intervals
  double xTicInt = 16.0;
  double yTicInt = 20.0;

  //Tic mark lengths.  If too small
  // on x-axis, a default value is
  // used later.
  double xTicLen = (yMax-yMin)/50;
  double yTicLen = (xMax-xMin)/50;

  //Calculation interval along x-axis
  double xCalcInc = 1.0;

  //Text fields for plotting parameters
  JTextField xMinTxt =
             new JTextField("" + xMin);
  JTextField xMaxTxt =
             new JTextField("" + xMax);
  JTextField yMinTxt =
             new JTextField("" + yMin);
  JTextField yMaxTxt =
             new JTextField("" + yMax);
  JTextField xTicIntTxt =
          new JTextField("" + xTicInt);
  JTextField yTicIntTxt =
          new JTextField("" + yTicInt);
  JTextField xCalcIncTxt =
         new JTextField("" + xCalcInc);

  //Panels to contain a label and a
  // text field
  JPanel pan0 = new JPanel();
  JPanel pan1 = new JPanel();
  JPanel pan2 = new JPanel();
  JPanel pan3 = new JPanel();
  JPanel pan4 = new JPanel();
  JPanel pan5 = new JPanel();
```

```
  JPanel pan6 = new JPanel();

  //Misc instance variables
  int frmWidth = 400;
  int frmHeight = 430;
  int width;
  int height;
  int number;
  GraphIntfc01 data;
  String args = null;

  //Plots are drawn on the canvases
  // in this array.
  Canvas[] canvases;

  //Constructor
  GUI(String args)throws
                NoSuchMethodException,
                ClassNotFoundException,
                InstantiationException,
                IllegalAccessException{

    if(args != null){
      //Save for use later in the
      // ActionEvent handler
      this.args = args;
      //Instantiate an object of the
      // target class using the String
      // name of the class.
      data = (GraphIntfc01)
                   Class.forName(args).
                         newInstance();
    }else{
      //Instantiate an object of the
      // test class named junk.
      data = new junk();
    }//end else

    //Create array to hold correct
    // number of Canvas objects.
    canvases =
            new Canvas[data.getNmbr()];

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
      throw new NoSuchMethodException(
                "Too many functions.  "
                  + "Only 5 allowed.");
    }//end if

    //Create the control panel and
    // give it a border for cosmetics.
    JPanel ctlPnl = new JPanel();
    ctlPnl.setLayout(//?rows x 4 cols
```

```
                    new GridLayout(0,4));
    ctlPnl.setBorder(
                     new EtchedBorder());

    //Button for replotting the graph
    JButton graphBtn =
                    new JButton("Graph");
    graphBtn.addActionListener(this);

    //Populate each panel with a label
    // and a text field.  Will place
    // these panels in a grid on the
    // control panel later.
    pan0.add(new JLabel("xMin"));
    pan0.add(xMinTxt);

    pan1.add(new JLabel("xMax"));
    pan1.add(xMaxTxt);

    pan2.add(new JLabel("yMin"));
    pan2.add(yMinTxt);

    pan3.add(new JLabel("yMax"));
    pan3.add(yMaxTxt);

    pan4.add(new JLabel("xTicInt"));
    pan4.add(xTicIntTxt);

    pan5.add(new JLabel("yTicInt"));
    pan5.add(yTicIntTxt);

    pan6.add(new JLabel("xCalcInc"));
    pan6.add(xCalcIncTxt);

    //Add the populated panels and the
    // button to the control panel with
    // a grid layout.
    ctlPnl.add(pan0);
    ctlPnl.add(pan1);
    ctlPnl.add(pan2);
    ctlPnl.add(pan3);
    ctlPnl.add(pan4);
    ctlPnl.add(pan5);
    ctlPnl.add(pan6);
    ctlPnl.add(graphBtn);

    //Create a panel to contain the
    // Canvas objects.  They will be
    // displayed in a one-column grid.
    JPanel canvasPanel = new JPanel();
    canvasPanel.setLayout(//?rows,1 col
                    new GridLayout(0,1));

    //Create a custom Canvas object for
    // each function to be plotted and
    // add them to the one-column grid.
```

```
    // Make background colors alternate
    // between white and gray.
    for(int cnt = 0;
                  cnt < number; cnt++){
      switch(cnt){
        case 0 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                          Color.WHITE);
          break;
        case 1 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                    Color.LIGHT_GRAY);
          break;
        case 2 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                          Color.WHITE);
          break;
        case 3 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].setBackground(
                    Color.LIGHT_GRAY);
          break;
        case 4 :
          canvases[cnt] =
                    new MyCanvas(cnt);
          canvases[cnt].
            setBackground(Color.WHITE);
      }//end switch
      //Add the object to the grid.
      canvasPanel.add(canvases[cnt]);
    }//end for loop

    //Add the sub-assemblies to the
    // frame.  Set its location, size,
    // and title, and make it visible.
    getContentPane().
                   add(ctlPnl,"South");
    getContentPane().
             add(canvasPanel,"Center");

    setBounds(0,0,frmWidth,frmHeight);
    setTitle("Graph01, " +
                 "Copyright 2002, " +
                 "Richard G. Baldwin");
    setVisible(true);

    //Set to exit on X-button click
    setDefaultCloseOperation(
                       EXIT_ON_CLOSE);
```

```
    //Get and save the size of the
    // plotting surface
    width = canvases[0].getWidth();
    height = canvases[0].getHeight();

    //Guarantee a repaint on startup.
    for(int cnt = 0;
                  cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end constructor
  //-----------------------------------//

  //This event handler is registered
  // on the JButton to cause the
  // functions to be replotted.
  public void actionPerformed(
                      ActionEvent evt){
    //Re-instantiate the object that
    // provides the data
    try{
      if(args != null){
        data = (GraphIntfc01)Class.
           forName(args).newInstance();
      }else{
        data = new junk();
      }//end else
    }catch(Exception e){
      //Known to be safe at this point.
      // Otherwise would have aborted
      // earlier.
    }//end catch

    //Set plotting parameters using
    // data from the text fields.
    xMin = Double.parseDouble(
                    xMinTxt.getText());
    xMax = Double.parseDouble(
                    xMaxTxt.getText());
    yMin = Double.parseDouble(
                    yMinTxt.getText());
    yMax = Double.parseDouble(
                    yMaxTxt.getText());
    xTicInt = Double.parseDouble(
                 xTicIntTxt.getText());
    yTicInt = Double.parseDouble(
                 yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(
                xCalcIncTxt.getText());

    //Calculate new values for the
    // length of the tic marks on the
    // axes.  If too small on x-axis,
    // a default value is used later.
```

```
    xTicLen = (yMax-yMin)/50;
    yTicLen = (xMax-xMin)/50;

    //Repaint the plotting areas
    for(int cnt = 0;
                  cnt < number; cnt++){
      canvases[cnt].repaint();
    }//end for loop

  }//end actionPerformed
  //--------------------------------//


//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
  int cnt;//object number
  //Factors to convert from double
  // values to integer pixel locations.
  double xScale;
  double yScale;

  MyCanvas(int cnt){//save obj number
    this.cnt = cnt;
  }//end constructor

  //Override the paint method
  public void paint(Graphics g){
    //Calculate the scale factors
    xScale = width/(xMax-xMin);
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                (int)((0-yMin)*yScale));
    drawAxes(g);//Draw the axes
    g.setColor(Color.BLACK);

    //Get initial data values
    double xVal = xMin;
    int oldX = getTheX(xVal);
    int oldY = 0;
    //Use the Canvas obj number to
    // determine which method to
    // invoke to get the value for y.
    switch(cnt){
      case 0 :
        oldY = getTheY(data.f1(xVal));
        break;
      case 1 :
        oldY = getTheY(data.f2(xVal));
        break;
      case 2 :
        oldY = getTheY(data.f3(xVal));
```

```
        break;
      case 3 :
        oldY = getTheY(data.f4(xVal));
        break;
      case 4 :
        oldY = getTheY(data.f5(xVal));
    }//end switch

    //Now loop and plot the points
    while(xVal < xMax){
      int yVal = 0;
      //Get next data value.  Use the
      // Canvas obj number to
      // determine which method to
      // invoke to get the value for y.
      switch(cnt){
        case 0 :
          yVal =
                getTheY(data.f1(xVal));
          break;
        case 1 :
          yVal =
                getTheY(data.f2(xVal));
          break;
        case 2 :
          yVal =
                getTheY(data.f3(xVal));
          break;
        case 3 :
          yVal =
                getTheY(data.f4(xVal));
          break;
        case 4 :
          yVal =
                getTheY(data.f5(xVal));
      }//end switch1

      //Convert the x-value to an int
      // and draw the next line segment
      int x = getTheX(xVal);
      g.drawLine(oldX,oldY,x,yVal);

      //Increment along the x-axis
      xVal += xCalcInc;

      //Save end point to use as start
      // point for next line segment.
      oldX = x;
      oldY = yVal;
    }//end while loop

  }//end overridden paint method
  //--------------------------------//

  //Method to draw axes with tic marks
  // and labels in the color RED
```

```
  void drawAxes(Graphics g){
    g.setColor(Color.RED);

    //Label left x-axis and bottom
    // y-axis.  These are the easy
    // ones.  Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                 getTheX(xMin),
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                        getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis.  These are the hard
    // ones because the position must
    // be adjusted by the font size and
    // the number of characters.
    //Get the width of the string for
    // right end of x-axis and the
    // height of the string for top of
    // y-axis
    //Create a string that is an
    // integer representation of the
    // label for the right end of the
    // x-axis.  Then get a character
    // array that represents the
    // string.
    int xMaxInt = (int)xMax;
    String xMaxStr = "" + xMaxInt;
    char[] array = xMaxStr.
                       toCharArray();

    //Get a FontMetrics object that can
    // be used to get the size of the
    // string in pixels.
    FontMetrics fontMetrics =
                    g.getFontMetrics();
    //Get a bounding rectangle for the
    // string
    Rectangle2D r2d =
           fontMetrics.getStringBounds(
               array,0,array.length,g);
    //Get the width and the height of
    // the bounding rectangle.  The
    // width is the width of the label
    // at the right end of the
    // x-axis.  The height applies to
    // all the labels, but is needed
    // specifically for the label at
    // the top end of the y-axis.
    int labWidth =
                 (int)(r2d.getWidth());
    int labHeight =
```

```
                (int)(r2d.getHeight());

    //Label the positive x-axis and the
    // positive y-axis using the width
    // and height from above to
    // position the labels.  These
    // labels apply to the very ends of
    // the axes at the edge of the
    // plotting surface.
    g.drawString("" + (int)xMax,
                 getTheX(xMax)-labWidth,
                  getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMax,
                getTheX(yTicLen/2)+2,
                getTheY(yMax)+labHeight);

    //Draw the axes
    g.drawLine(getTheX(xMin),
                         getTheY(0.0),
                         getTheX(xMax),
                         getTheY(0.0));

    g.drawLine(getTheX(0.0),
                        getTheY(yMin),
                        getTheX(0.0),
                        getTheY(yMax));

    //Draw the tic marks on axes
    xTics(g);
    yTics(g);
  }//end drawAxes

  //---------------------------------//

  //Method to draw tic marks on x-axis
  void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
                   getTheY(-xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small.  In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
      topEnd = 5;
      bottomEnd = -5;
    }//end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
```

```
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub -= xTicInt;
    }//end while

  }//end xTics
  //---------------------------------//

  //Method to draw tic marks on y-axis
  void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub -= yTicInt;
    }//end while

  }//end yTics

  //---------------------------------//

  //This method translates and scales
  // a double y value to plot properly
  // in the integer coordinate system.
  // In addition to scaling, it causes
  // the positive direction of the
  // y-axis to be from bottom to top.
```

```
  int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
  }//end getTheY
  //---------------------------------//

  //This method scales a double x value
  // to plot properly in the integer
  // coordinate system.
  int getTheX(double x){
    return (int)(x*xScale);
  }//end getTheX
  //---------------------------------//

}//end inner class MyCanvas
//===================================//

}//end class GUI
//===================================//

//Sample test class.  Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 4;
  }//end getNmbr

  public double f1(double x){
    return (x*x*x)/200.0;
  }//end f1

  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
    return (x*x)/200.0;
  }//end f3

  public double f4(double x){
    return 50*Math.cos(x/10.0);
  }//end f4

  public double f5(double x){
    return 100*Math.sin(x/20.0);
  }//end f5

}//end sample class junk
```

Listing 39

```
/* File Graph02.java
Copyright 2002, R.G.Baldwin

Note:  This program requires access to
the interface named GraphIntfc01.

This is a modified version of the
program named Graph01.  That program
plots up to five separate curves in
separate plotting areas.  This program
superimposes up to five separate curves
in different colors in the same
plotting area.

This is a plotting program.  It is
designed to access a class file, which
implements GraphIntfc01, and to plot up
to five functions defined in that class
file.

The methods corresponding to the
functions are named f1, f2, f3, f4,
and f5.

The class containing the functions must
also define a static method named
getNmbr(), which takes no parameters
and returns the number of functions to
be plotted.  If this method returns a
value greater than 5, a
NoSuchMethodException will be thrown.

Note that the constructor for the class
that implements GraphIntfc01 must not
require any parameters due to the
use of the newInstance method of the
Class class to instantiate an object
of that class.

If the number of functions is less
than 5, then the absent method names
must begin with f5 and work down toward
f1.  For example, if the number of
functions is 3, then the program will
expect to call methods named f1, f2,
and f3.  It is OK for the absent
methods to be defined in the class.
They simply won't be invoked.

Each curve is plotted in a different
color.  The correspondence between
colors and function calls is as
follows:

f1: BLACK
f2: BLUE
```

```
f3: RED
f4: MAGENTA
f5: CYAN

A Cartesian coordinate system with
axes, tic marks, and labels is drawn in
green.

The labels displayed on the axes,
correspond to the values of the extreme
edges of the plotting area.

The program also compiles a sample
class named junk, which contains five
methods and the method named getNmbr.
This makes it easy to compile and test
this program in a stand-alone mode.

At runtime, the name of the class that
implements the GraphIntfc01 interface
must be provided as a command-line
parameter.  If this parameter is
missing, the program instantiates an
object from the internal class named
junk and plots the data provided by
that class.  Thus, you can test the
program by running it with no
command-line parameter.

This program provides the following
text fields for user input, along with
a button labeled Graph.  This allows
the user to adjust the parameters and
replot the graph as many times with as
many plotting scales as needed:

xMin = minimum x-axis value
xMax = maximum x-axis value
yMin = minimum y-axis value
yMax = maximum y-axis value
xTicInt = tic interval on x-axis
yTicInt = tic interval on y-axis
xCalcInc = calculation interval

The user can modify any of these
parameters and then click the Graph
button to cause the five functions
to be re-plotted according to the
new parameters.

Whenever the Graph button is clicked,
the event handler instantiates a new
object of the class that implements
the GraphIntfc01 interface.  Depending
on the nature of that class, this may
be redundant in some cases.  However,
```

```
it is useful in those cases where it
is necessary to refresh the values of
instance variables defined in the
class (such as a counter, for example).

Tested using JDK 1.4.0 under Win 2000.

This program uses constants that were
first defined in the Color class of
v1.4.0.  Therefore, the program
requires v1.4.0 or later to compile and
run correctly.
***********************************/

import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;
import javax.swing.border.*;

class Graph02{
  public static void main(
           String[] args)
           throws NoSuchMethodException,
                  ClassNotFoundException,
                  InstantiationException,
                  IllegalAccessException{
    if(args.length == 1){
      //pass command-line parameter
      new GUI(args[0]);
    }else{
      //no command-line parameter given
      new GUI(null);
    }//end else
  }// end main
}//end class Graph02 definition
//===================================//

class GUI extends JFrame
             implements ActionListener{

  //Define plotting parameters and
  // their default values.
  double xMin = -10.0;
  double xMax = 325.0;
  double yMin = -100.0;
  double yMax = 100.0;

  //Tic mark intervals
  double xTicInt = 20.0;
  double yTicInt = 20.0;

  //Tic mark lengths.  If too small
  // on x-axis, a default value is
  // used later.
  double xTicLen = (yMax-yMin)/50;
```

```
double yTicLen = (xMax-xMin)/50;

//Calculation interval along x-axis
double xCalcInc = 1.0;

//Text fields for plotting parameters
JTextField xMinTxt =
           new JTextField("" + xMin);
JTextField xMaxTxt =
           new JTextField("" + xMax);
JTextField yMinTxt =
           new JTextField("" + yMin);
JTextField yMaxTxt =
           new JTextField("" + yMax);
JTextField xTicIntTxt =
        new JTextField("" + xTicInt);
JTextField yTicIntTxt =
        new JTextField("" + yTicInt);
JTextField xCalcIncTxt =
       new JTextField("" + xCalcInc);

//Panels to contain a label and a
// text field
JPanel pan0 = new JPanel();
JPanel pan1 = new JPanel();
JPanel pan2 = new JPanel();
JPanel pan3 = new JPanel();
JPanel pan4 = new JPanel();
JPanel pan5 = new JPanel();
JPanel pan6 = new JPanel();

//Misc instance variables
int frmWidth = 400;
int frmHeight = 430;
int width;
int height;
int number;
GraphIntfc01 data;
String args = null;

//Plots are drawn on theCanvas
Canvas theCanvas;

//Constructor
GUI(String args)throws
             NoSuchMethodException,
             ClassNotFoundException,
             InstantiationException,
             IllegalAccessException{

  if(args != null){
    //Save for use later in the
    // ActionEvent handler
    this.args = args;
    //Instantiate an object of the
    // target class using the String
```

```
      // name of the class.
      data = (GraphIntfc01)
                  Class.forName(args).
                        newInstance();
    }else{
      //Instantiate an object of the
      // test class named junk.
      data = new junk();
    }//end else

    //Throw exception if number of
    // functions is greater than 5.
    number = data.getNmbr();
    if(number > 5){
      throw new NoSuchMethodException(
                "Too many functions.  "
                  + "Only 5 allowed.");
    }//end if

    //Create the control panel and
    // give it a border for cosmetics.
    JPanel ctlPnl = new JPanel();
    ctlPnl.setLayout(//?rows x 4 cols
                  new GridLayout(0,4));
    ctlPnl.setBorder(
                   new EtchedBorder());

    //Button for replotting the graph
    JButton graphBtn =
                  new JButton("Graph");
    graphBtn.addActionListener(this);

    //Populate each panel with a label
    // and a text field.  Will place
    // these panels in a grid on the
    // control panel later.
    pan0.add(new JLabel("xMin"));
    pan0.add(xMinTxt);

    pan1.add(new JLabel("xMax"));
    pan1.add(xMaxTxt);

    pan2.add(new JLabel("yMin"));
    pan2.add(yMinTxt);

    pan3.add(new JLabel("yMax"));
    pan3.add(yMaxTxt);

    pan4.add(new JLabel("xTicInt"));
    pan4.add(xTicIntTxt);

    pan5.add(new JLabel("yTicInt"));
    pan5.add(yTicIntTxt);

    pan6.add(new JLabel("xCalcInc"));
    pan6.add(xCalcIncTxt);
```

```
    //Add the populated panels and the
    // button to the control panel with
    // a grid layout.
    ctlPnl.add(pan0);
    ctlPnl.add(pan1);
    ctlPnl.add(pan2);
    ctlPnl.add(pan3);
    ctlPnl.add(pan4);
    ctlPnl.add(pan5);
    ctlPnl.add(pan6);
    ctlPnl.add(graphBtn);

    //Create a custom Canvas object for
    // all functions to be plotted on.
    theCanvas = new MyCanvas();
    theCanvas.setBackground(
                          Color.WHITE);

    //Add the sub-assemblies to the
    // frame.  Set its location, size,
    // and title, and make it visible.
    getContentPane().add(
                        ctlPnl,"South");
    getContentPane().add(
                    theCanvas,"Center");

    setBounds(0,0,frmWidth,frmHeight);
    setTitle("Graph02, " +
                "Copyright 2002, " +
                "Richard G. Baldwin");
    setVisible(true);

    //Set to exit on X-button click
    setDefaultCloseOperation(
                         EXIT_ON_CLOSE);

    //Get and save the size of the
    // plotting surface
    width = theCanvas.getWidth();
    height = theCanvas.getHeight();

    //Guarantee a repaint on startup.
    theCanvas.repaint();

  }//end constructor
  //--------------------------------//

  //This event handler is registered
  // on the JButton to cause the
  // functions to be replotted.
  public void actionPerformed(
                      ActionEvent evt){
    //Re-instantiate the object that
    // provides the data
    try{
```

```
      if(args != null){
        data = (GraphIntfc01)Class.
           forName(args).newInstance();
      }else{
        data = new junk();
      }//end else
    }catch(Exception e){
      //Known to be safe at this point.
      // Otherwise would have aborted
      // earlier.
    }//end catch

    //Set plotting parameters using
    // data from the text fields.
    xMin = Double.parseDouble(
                    xMinTxt.getText());
    xMax = Double.parseDouble(
                    xMaxTxt.getText());
    yMin = Double.parseDouble(
                    yMinTxt.getText());
    yMax = Double.parseDouble(
                    yMaxTxt.getText());
    xTicInt = Double.parseDouble(
                  xTicIntTxt.getText());
    yTicInt = Double.parseDouble(
                  yTicIntTxt.getText());
    xCalcInc = Double.parseDouble(
                xCalcIncTxt.getText());

    //Calculate new values for the
    // length of the tic marks on the
    // axes.  If too small on x-axis,
    // a default value is used later.
    xTicLen = (yMax-yMin)/50;
    yTicLen = (xMax-xMin)/50;

    //Repaint the plotting area
    theCanvas.repaint();

  }//end actionPerformed
  //---------------------------------//


//This is an inner class, which is used
// to override the paint method on the
// plotting surface.
class MyCanvas extends Canvas{
  //Factors to convert from double
  // values to integer pixel locations.
  double xScale;
  double yScale;

  //Override the paint method
  public void paint(Graphics g){
    //Calculate the scale factors
    xScale = width/(xMax-xMin);
```

```
    yScale = height/(yMax-yMin);

    //Set the origin based on the
    // minimum values in x and y
    g.translate((int)((0-xMin)*xScale),
                (int)((0-yMin)*yScale));
    drawAxes(g);//Draw the axes

    //Draw each curve in a different
    // color.
    for(int cnt=0; cnt < number;
                                cnt++){

      //Get initial data values
      double xVal = xMin;
      int oldX = getTheX(xVal);
      int oldY = 0;
      //Use the curve number to
      // determine which method to
      // invoke to get the value for y.
      switch(cnt){
        case 0 :
          oldY= getTheY(data.f1(xVal));
          g.setColor(Color.BLACK);
          break;
        case 1 :
          oldY= getTheY(data.f2(xVal));
          g.setColor(Color.BLUE);
          break;
        case 2 :
          oldY= getTheY(data.f3(xVal));
          g.setColor(Color.RED);
          break;
        case 3 :
          oldY= getTheY(data.f4(xVal));
          g.setColor(Color.MAGENTA);
          break;
        case 4 :
          oldY= getTheY(data.f5(xVal));
          g.setColor(Color.CYAN);
      }//end switch

      //Now loop and plot the points
      while(xVal < xMax){
        int yVal = 0;
        //Get next data value.  Use the
        // curve number to determine
        // which method to invoke to
        // get the value for y.
        switch(cnt){
          case 0 :
            yVal = getTheY(
                        data.f1(xVal));
            break;
          case 1 :
            yVal = getTheY(
```

```
                        data.f2(xVal));
            break;
          case 2 :
            yVal = getTheY(
                        data.f3(xVal));
            break;
          case 3 :
            yVal = getTheY(
                        data.f4(xVal));
            break;
          case 4 :
            yVal = getTheY(
                        data.f5(xVal));
        }//end switch1

        //Convert the x-value to an int
        // and draw the next line
        // segment
        int x = getTheX(xVal);
        g.drawLine(oldX,oldY,x,yVal);

        //Increment along the x-axis
        xVal += xCalcInc;

        //Save end point to use as
        // start point for next line
        // segment.
        oldX = x;
        oldY = yVal;
      }//end while loop

    }//end for loop

  }//end overridden paint method
  //---------------------------------//

  //Method to draw axes with tic marks
  // and labels in the color GREEN
  void drawAxes(Graphics g){
    g.setColor(Color.GREEN);

    //Label left x-axis and bottom
    // y-axis.  These are the easy
    // ones.  Separate the labels from
    // the ends of the tic marks by
    // two pixels.
    g.drawString("" + (int)xMin,
                 getTheX(xMin),
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMin,
                  getTheX(yTicLen/2)+2,
                        getTheY(yMin));

    //Label the right x-axis and the
    // top y-axis.  These are the hard
    // ones because the position must
```

```
    // be adjusted by the font size and
    // the number of characters.
    //Get the width of the string for
    // right end of x-axis and the
    // height of the string for top of
    // y-axis
    //Create a string that is an
    // integer representation of the
    // label for the right end of the
    // x-axis.  Then get a character
    // array that represents the
    // string.
    int xMaxInt = (int)xMax;
    String xMaxStr = "" + xMaxInt;
    char[] array = xMaxStr.
                        toCharArray();

    //Get a FontMetrics object that can
    // be used to get the size of the
    // string in pixels.
    FontMetrics fontMetrics =
                    g.getFontMetrics();
    //Get a bounding rectangle for the
    // string
    Rectangle2D r2d =
           fontMetrics.getStringBounds(
               array,0,array.length,g);

    //Get the width and the height of
    // the bounding rectangle.  The
    // width is the width of the label
    // at the right end of the
    // x-axis.  The height applies to
    // all the labels, but is needed
    // specifically for the label at
    // the top end of the y-axis.
    int labWidth =
                 (int)(r2d.getWidth());
    int labHeight =
                (int)(r2d.getHeight());

    //Label the positive x-axis and the
    // positive y-axis using the width
    // and height from above to
    // position the labels.  These
    // labels apply to the very ends of
    // the axes at the edge of the
    // plotting surface.
    g.drawString("" + (int)xMax,
                getTheX(xMax)-labWidth,
                 getTheY(xTicLen/2)-2);
    g.drawString("" + (int)yMax,
              getTheX(yTicLen/2)+2,
              getTheY(yMax)+labHeight);

    //Draw the axes
```

```
    g.drawLine(getTheX(xMin),
                         getTheY(0.0),
                         getTheX(xMax),
                         getTheY(0.0));

    g.drawLine(getTheX(0.0),
                        getTheY(yMin),
                        getTheX(0.0),
                        getTheY(yMax));

    //Draw the tic marks on axes
    xTics(g);
    yTics(g);
  }//end drawAxes

  //--------------------------------//

  //Method to draw tic marks on x-axis
  void xTics(Graphics g){
    double xDoub = 0;
    int x = 0;

    //Get the ends of the tic marks.
    int topEnd = getTheY(xTicLen/2);
    int bottomEnd =
                   getTheY(-xTicLen/2);

    //If the vertical size of the
    // plotting area is small, the
    // calculated tic size may be too
    // small.  In that case, set it to
    // 10 pixels.
    if(topEnd < 5){
      topEnd = 5;
      bottomEnd = -5;
    }//end if

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive x-axis
    // moving to the right from zero.
    while(xDoub < xMax){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub += xTicInt;
    }//end while

    //Now do the negative x-axis moving
    // to the left from zero
    xDoub = 0;
    while(xDoub > xMin){
      x = getTheX(xDoub);
      g.drawLine(x,topEnd,x,bottomEnd);
      xDoub -= xTicInt;
    }//end while
```

```
  }//end xTics
  //---------------------------------//

  //Method to draw tic marks on y-axis
  void yTics(Graphics g){
    double yDoub = 0;
    int y = 0;
    int rightEnd = getTheX(yTicLen/2);
    int leftEnd = getTheX(-yTicLen/2);

    //Loop and draw a series of short
    // lines to serve as tic marks.
    // Begin with the positive y-axis
    // moving up from zero.
    while(yDoub < yMax){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub += yTicInt;
    }//end while

    //Now do the negative y-axis moving
    // down from zero.
    yDoub = 0;
    while(yDoub > yMin){
      y = getTheY(yDoub);
      g.drawLine(rightEnd,y,leftEnd,y);
      yDoub -= yTicInt;
    }//end while

  }//end yTics

  //---------------------------------//

  //This method translates and scales
  // a double y value to plot properly
  // in the integer coordinate system.
  // In addition to scaling, it causes
  // the positive direction of the
  // y-axis to be from bottom to top.
  int getTheY(double y){
    double yDoub = (yMax+yMin)-y;
    int yInt = (int)(yDoub*yScale);
    return yInt;
  }//end getTheY
  //---------------------------------//

  //This method scales a double x value
  // to plot properly in the integer
  // coordinate system.
  int getTheX(double x){
    return (int)(x*xScale);
  }//end getTheX
  //---------------------------------//

}//end inner class MyCanvas
//===================================//
```

```
}//end class GUI
//=================================//

//Sample test class.  Required for
// compilation and stand-alone
// testing.
class junk implements GraphIntfc01{
  public int getNmbr(){
    //Return number of functions to
    // process.  Must not exceed 5.
    return 5;
  }//end getNmbr

  public double f1(double x){
    return (x*x*x)/200.0;
  }//end f1

  public double f2(double x){
    return -(x*x*x)/200.0;
  }//end f2

  public double f3(double x){
    return (x*x)/200.0;
  }//end f3

  public double f4(double x){
    return 50*Math.cos(x/10.0);
  }//end f4

  public double f5(double x){
    return 100*Math.sin(x/20.0);
  }//end f5

}//end sample class junk
```

Listing 40

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of

Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-