Plotting Large Quantities of Data using Java

Learn how to use Java to plot millions of multi-channel data values in an easy-to-view format with very little programming effort.

Published: August 23, 2005
By Richard G. Baldwin

Java Programming, Notes # 1492

Preface

In one of my earlier lessons entitled Plotting Engineering and Scientific Data using Java, I published a generalized 2D plotting class that makes it easy to cause other programs to display their outputs in 2D Cartesian coordinates. I have used that plotting class in numerous lessons since I published it. Hopefully, some of you have been using it as well.

In another of my earlier lessons entitled Plotting 3D Surfaces using Java, I presented and explained a 3D surface plotting class that is also very easy to use. I have used that class in several lessons since then, and I will be using it in many future lessons as well.

Plotting large quantities of data

One of the common requirements of engineering, technical, and scientific computing is to be able to plot and to examine very large quantities of data. This is particularly true in time-series analysis, spectral analysis, and digital signal processing *(DSP)*. I will present and explain four separate Java classes in this lesson, which make it very easy to plot and to examine large quantities of data in Java programs.

How do you use these classes?

All that's necessary to use these classes to plot large quantities of data is to:

1. Instantiate a plotting object of type **PlotALot01**, **PlotALot02**, **PlotALot03** or **PlotALot04**.

2. Feed the data values that need to be plotted to the plotting object as they become available.
3. Invoke a method named **plotData** on the plotting object when all of the data has been fed to the object.

It couldn't be easier

The choice among the four classes listed above depends on whether you need to plot one, two, or three channels of data, and the format in which you want to plot the data. The class named **PlotALot01** is used to plot single-channel data. The classes named **PlotALot02** and **PlotALot03** are used to plot two-channel data in two different formats. The class named **PlotALot04** is used to plot three-channel data.

Will use in subsequent lessons

These plotting classes will be used in numerous future lessons involving such complex topics as adaptive signal processing using Java.

A free plotting class

If you arrived at this page seeking a free Java class for plotting your data, you are in luck. Just copy the source code for the classes in Listing 35 through Listing 38 near the end of this lesson and feel free to use them as described in the comments in the source code.

On the other hand, if you would like to learn how the classes do what they do, and perhaps use your programming skills to improve them, keep reading. Hopefully, once you have finished the lesson, you will have learned quite a lot about plotting large quantities of data using Java.

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 figures and listings 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

The four classes that I will present and explain in this lesson are designed to make it easy for you to plot and examine large quantities of data.

Sample output for PlotALot01class

The first class named **PlotALot01** is designed for the plotting of large quantities of single-channel data. Figure 1 shows an example of the plotted output from this class when used to plot a *small amount* of data.

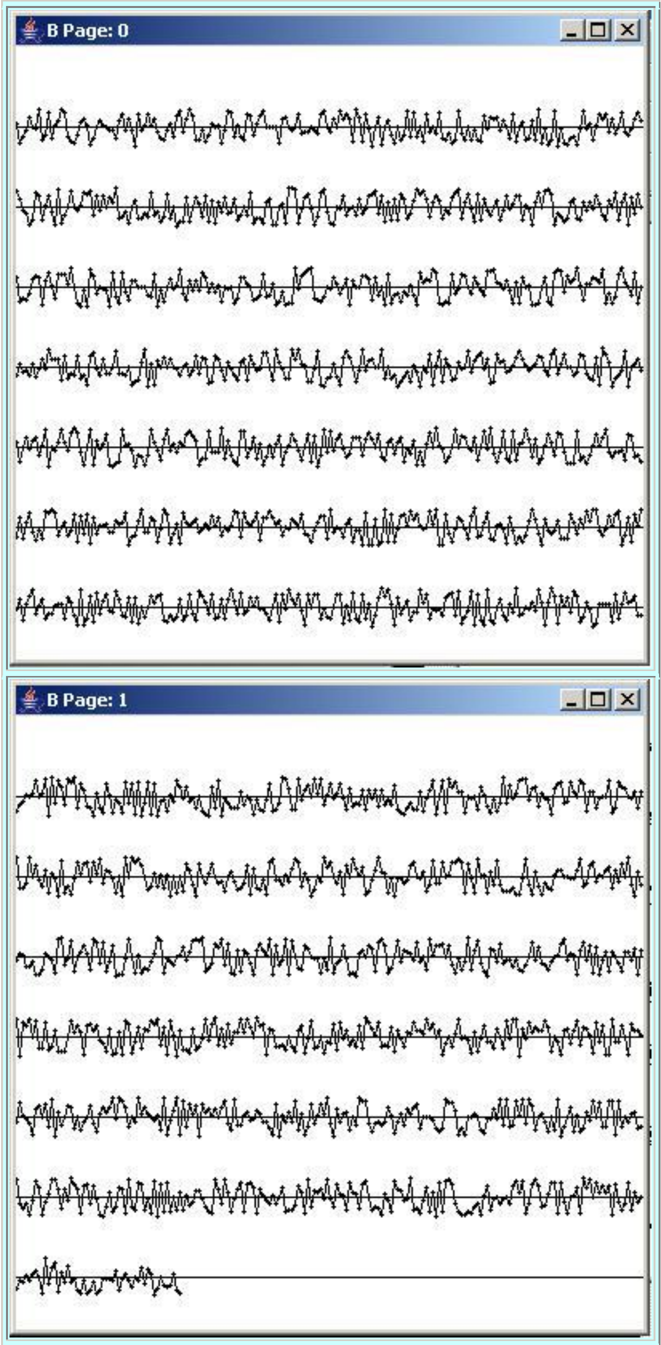


Figure 1

Usage information

To use the class named **PlotALot01** to plot data, you first instantiate an object of the class, and then you feed data to it as the data becomes available within your program.

When all of the data has been fed to the plotting object, you invoke a method named **plotData** on the object. This causes the object to produce one or more pages of plotted data in a stack on the screen. The page containing the earliest data is on the top of the stack and the page containing the latest data on the bottom of the stack.

Plotting format

The first data sample is plotted on the left end of the top trace on the top page *(entitled Page: 0).* Successive data values are plotted from left to right across the page. When the data for the first trace reaches the right end of the trace, the next data sample is plotted at the left end of a new trace that is created below the current trace. Hence, the chronological order of the data is from left to right, top to bottom.

A horizontal axis is drawn for each trace. Positive data values are plotted above the axis and negative values are plotted below the axis.

On to the next page

When the bottom trace on a page is filled, a new page is created automatically. The next data sample is plotted on the left end of the top trace on the new page and the process described above is repeated until that page also become full. Then a new page is created, etc.

Nearly unlimited plotting capacity

You can cause the page size to be as large as you want up to the full size of the screen on your computer. You can create as many pages as you want and you can place as many traces on each page as you want.

Other than the amount of memory that is available to the Java virtual machine, *(and perhaps some limit on the number of **Page** objects allowed by the operating system),* there is almost no limit to the number of pages that can be produced and the amount of data that can be plotted.

Millions of data values plotted

I have successfully plotted two million data values in 141 full screen pages on a modest laptop computer with no difficulty whatsoever. When I pushed that total up to eight million data values in 563 full screen pages, the plotting process slowed down, but I was still able to display and examine the plots. The practical limit on my computer seems to be somewhere between two million and eight million data values.

Two sample pages

Figure 1 shows two pages that were physically removed from the stack and arranged with the page containing the earliest data above the page containing the latest data for publication in this lesson.

Two overloaded constructors

Two overloaded constructors are provided for the class. One constructor plots the data using a set of default plotting parameters. This constructor is provided for extreme ease of use. The only information that you must provide to this constructor is a string that becomes part of the title for each page.

> *(The pages in Figure 1 were plotted using default plotting parameters with a title string of "B". The amount of data that was fed to the plotting object for Figure 1 filled Page 0 and almost filled Page 1.)*

Can change default plotting parameters

I set the values of the default plotting parameters to make the results suitable for use in this narrow publication format. If you don't like my choice of default plotting parameters, you can change them to values that you find more useful. For example, you could cause the default size of the **Frame** object to fill your screen, allowing you to plot quite a lot of data on each page.

Control over the plotting parameters

The other overloaded constructor takes seven parameters that allow you to control all aspects of the plotting format including:

- Page title
- Frame width and hence plotted data width
- Frame height, spacing between traces, and hence the number of traces per page
- Spacing between samples, number of traces per page, and hence the number of samples per page
- Width and height of an oval that is used to mark each sample on the plot

The plotted sample values are connected by a straight line. Each sample is marked with an oval. You can specify the width and the height of the oval in pixels. If you set the width and height to zero, the oval simply disappears from the plot.

Default plotting parameters in Figure 1

The plots in Figure 1 were produced using the constructor that applies default plotting parameters. For example, the data was plotted using the default value of two pixels per sample. Hence the lines connecting the sample values in Figure 1 are very short.

The ovals in Figure 1 had a default width and height of two pixels each. At this small size, the ovals end up looking more like plus characters than ovals.

The overall parameters governing the plot in Figure 1 were:

```
Title: B
Frame width: 400
Frame height: 410
Page width: 392
Page height: 383
Trace spacing: 50
Sample spacing: 2
Traces per page: 7
Samples per page: 1372
```

Explanation of terms

Some explanation of the terminology in the above list is probably in order. The *Frame width* and *Frame height* are the actual width and height of the **Frame** objects shown in Figure 1.

The *Page width* and *Page height* are the width and height of a **Canvas** object contained in the **Frame** object, upon which the plotting is performed. The width of the **Canvas** actually controls the number of samples that can be plotted in each trace.

The *Trace spacing* is the number of pixels that separate each of the horizontal axes on a page in Figure 1.

The *Sample spacing* specifies the number of pixels that are dedicated to each sample horizontally. In Figure 1, that value is 2. This means that every other black pixel in Figure 1 indicates the value of a data sample. The pixels in between are fillers.

The *Traces per page* specifies the number of horizontal axes on each page against which the data values are plotted.

Samples per page gives the actual number of data values that are plotted on each page. This is determined from the values of *Traces per page, Sample spacing*, and *Page width.*

Location of the stack of plots

There are also two overloaded versions of the method named **plotData**. One version lets you specify the location of the upper left corner of the stack of pages relative to the upper left corner of the screen. The other version simply places the stack of pages in the upper left corner of the screen by default.

Sample output for PlotALot02 class

The class named **PlotALot01** is designed for the plotting of large quantities of data from a single channel as described above. The classes named **PlotALot02** and **PlotALot03** are each designed to plot two channels of data. These two classes plot the two-channel data in different formats.

Superimposed data

The class named **PlotALot02** provides all of the features described above for the class named **PlotALot01**, such as overloaded constructors, overloaded **plotData** methods, etc. In addition, it provides the capability to superimpose two sets of data on the same axes with one set being plotted in black and the other being plotted in red. This is illustrated in Figure 2.

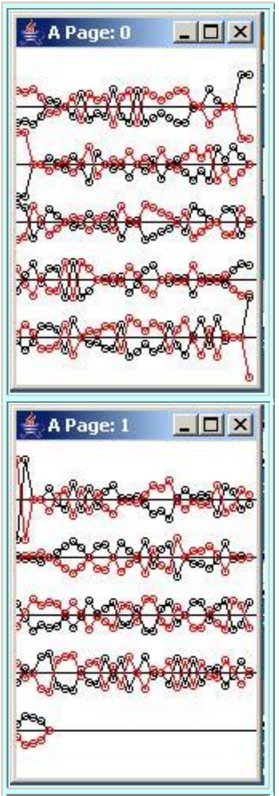


Figure 2

Same data, different sign

The plots in Figure 2 were produced by plotting two versions of the same data. The algebraic sign of each of the data values was inverted in one set of data relative to the other. Thus, the red plot in Figure 2 is an upside down version of the black plot. This makes it easy to confirm that both plotting processes are behaving the same way.

Plotting parameters were controlled

The plots in Figure 2 were produced using the version of the constructor that allows the user to control the plotting parameters. The overall plotting parameters for Figure 2 are shown below:

```
Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36
Sample spacing: 5
```

```
Traces per page: 5
Samples per page: 150
```

Larger ovals

As you can see, the ovals that were used to mark the sample values in Figure 2 were larger than in Figure 1. With a height and a width of four pixels, each oval turned out to be a circle centered on the sample value.

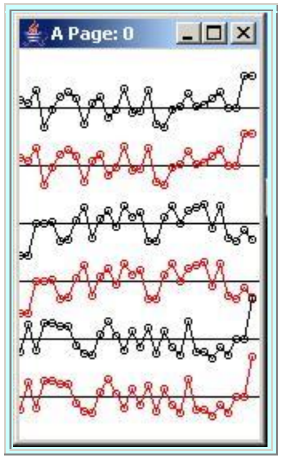
Horizontal scaling was greater

Also, the horizontal scaling in Figure 2 was five pixels per sample as opposed to two pixels per sample in Figure 1. As a result, the circles marking the samples were further apart, and the straight lines connecting the circles are often visible.

Sample output for PlotALot03 class

The classes named **PlotALot02** and **PlotALot03** are each designed to plot two channels of data. These two classes plot the two-channel data in different formats. Whereas **PlotALot02** superimposes the two sets of data on the same horizontal axes using color to provide visual separation, **PlotALot03** plots the two sets of data on alternating horizontal axes as shown in Figure 3. **PlotALot03** also uses color to provide visual separation between the two sets of data. One set is plotted on the odd numbered axes in black. The other set is plotted on the even numbered axes in red.

The class named **PlotALot03** also provides all of the general capabilities described earlier for the class named **PlotALot01** that are appropriate for a two-channel plotting system.



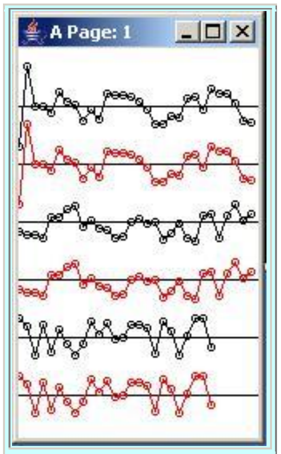


Figure 3

Same data, two colors

The two sets of data plotted in Figure 3 consisted of exactly the same values. Thus, the plots on the even numbered axes look just like the plots on the odd numbered axes except that one plot is red and the other is black. Using the same values for each set of data makes it easy to confirm that both plotting processes are behaving the same way.

The plotting parameters

The overall plotting parameters for Figure 3 are shown below:

```
Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 90
```

Because **PlotALot03** doesn't superimpose the two sets of data, twice as many pages would be required for **PlotALot03** to plot a given amount of data as would be required by **PlotALot02** for the same **Page** size.

PlotALot03 will refuse to plot data for a set of plotting parameters that result in an odd number of traces on the page.

Sample output for PlotALot04 class

The class named **PlotALot04** plots three sets of data on separate horizontal axes as shown in Figure 4. The first set of data is plotted in black. The second set of data is plotted in red. The third set of data is plotted in blue. This class is particularly useful for displaying the input, output, and error signals involved in adaptive signal processing.

The class named **PlotALot04** also provides all of the general capabilities described earlier for the class named **PlotALot01** that are appropriate for a three-channel plotting system.

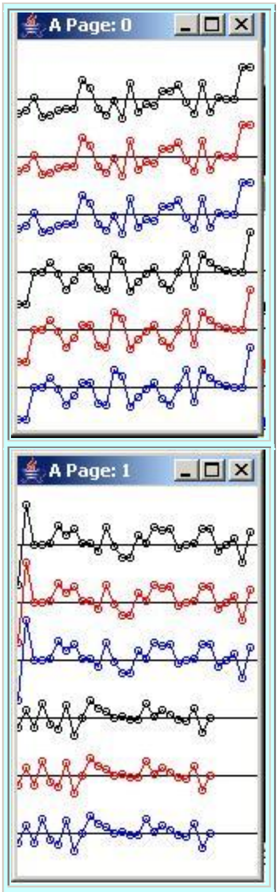


Figure 4

Same data, three colors

The three sets of data plotted in Figure 4 consisted of exactly the same values. Thus, the plots on the three different axes look just alike except that the first plot is black, the second plot is red and the third is blue. Using the same values for each set of data makes it easy to confirm that all three plotting processes are behaving the same way.

The plotting parameters

The overall plotting parameters for Figure 4 are shown below:

```
Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 60
```

PlotALot04 will terminate if the number of traces per page is not evenly divisible by 3

Sample Programs

The class named PlotALot01

Now that you know where we are heading, it's time to examine these four classes in detail. I will begin with the class named **PlotALot01**.

Purpose of the class

This class is designed to plot large amounts of data for a single channel. The class is particularly useful for plotting time series data. Also, by carefully adjusting the plotting parameters, this class can be used to plot large quantities of spectral data in a waterfall display with each new spectral estimate being plotted immediately below the previous estimate.

Usage information

The class provides a **main** method so that the class can be run as an application to test itself. The **main** method also illustrates how to use the class.

There are three steps involved in the use of this class for plotting large quantities of data:

1. Instantiate a plotting object of type **PlotALot01** using one of two overloaded constructors.
2. Feed the data that is to be plotted to the plotting object by invoking the **feedData** method once for each data value.
3. Invoke one of two overloaded **plotData** methods on the plotting object once all of the data has been fed to the object. This causes all of the data to be plotted and causes the pages to be stacked in a particular location on the screen with page 0 on the top of the stack.

Different plotting objects

A program that uses this class for plotting can instantiate as many different plotting objects as are needed to plot all of the different sets of data that need to be plotted independently of one another.

> *(For example, a program that uses this class could instantiate one plotting object to plot time series data and a different plotting object to plot spectral data.)*

Can plot a large number of data values

Each plotting object can be used to plot as many data values as needed *(unless the program runs out of memory).*

> *(As mentioned earlier, I have successfully plotted two million data values in 141 full screen pages on a modest laptop computer with no difficulty whatsoever. When I pushed that total up to eight million data values in 563 full screen pages, the plotting process slowed down, but I was still able to display and examine the plots. The practical limit on my computer seems to be somewhere between two million and eight million data values.)*

Multiple Page objects

A plotting object of type **PlotALot01** owns one or more **Page** objects that extend the **Frame** class. The plotting object can own as many **Page** objects as are necessary to plot all of the data that is fed to the plotting object.

A stack of Page objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen, with the data plotted on a **Canvas** object contained by each **Page** object. The **Page** showing the earliest data *(page 0)* is on the top of the stack and the **Page** showing the latest data is on the bottom of the stack.

> *(The* ***Page*** *objects on the top of the stack must be physically moved in order to see the Page objects on the bottom of the stack.)*

Multiple traces on each Page object

As shown in Figure 1, each **Page** object contains one or more horizontal axes on which the data is plotted. The earliest data is plotted on the axis nearest the top of the **Page** moving from left to right across the **Page**. Positive data values are plotted above the axis and negative values are plotted below the axis.

When the right end of an axis is reached, the next data value is plotted on the left end of the axis immediately below it. When the right end of the last axis on the **Page** is reached, a new **Page** object is automatically created and the next data value is plotted at the left end of the top axis on the new **Page** object.

Two overloaded constructors

There are two overloaded versions of the constructor for the **PlotALot01class**. One overloaded version accepts several incoming parameters allowing the user to control various aspects of the plotting format. *(An example of the use of this constructor is shown in Figure 5.)* A second overloaded version accepts a title string only and sets all of the plotting parameters to default values. *(An example of the use of this constructor is shown in Figure 1.)*

> *(You can easily modify the default values and recompile the class if you prefer different default values.)*

Constructor parameters

The parameters for the version of the constructor that accepts plotting parameters are:

- **String title**: Title for the **Frame** object. This title is concatenated with the page number and the result appears in the banner at the top of the **Page** as shown in Figure 1.
- **int frameWidth**: The **Frame** width in pixels.
- **int frameHeight**: The **Frame** height in pixels.
- **int traceSpacing**: Distance between trace axes in pixels.
- **int sampSpace**: Number of pixels dedicated to each data sample in pixels per sample. *(Must be 1 or greater.)*
- **int ovalWidth**: Width of an oval that is used to mark the sample value on the plot. *(See Figure 5 for a good example of the ovals. Set the oval width and height parameters to zero to eliminate the ovals altogether.)*
- **int ovalHeight**: Height of an oval that is used to mark the sample value on the plot.

Two plotting objects for test purposes

For test purposes, the **main** method instantiates and feeds two independent plotting objects. Plotting parameters are specified for the first plotting object and the stack of pages for this plotting object is located 401 pixels to the right of the upper left corner of the screen. The output produced by this plotting object is shown in Figure 5 below. *(The two pages in the screen shot in Figure 5 were manually relocated for reasons that I will explain later.)*

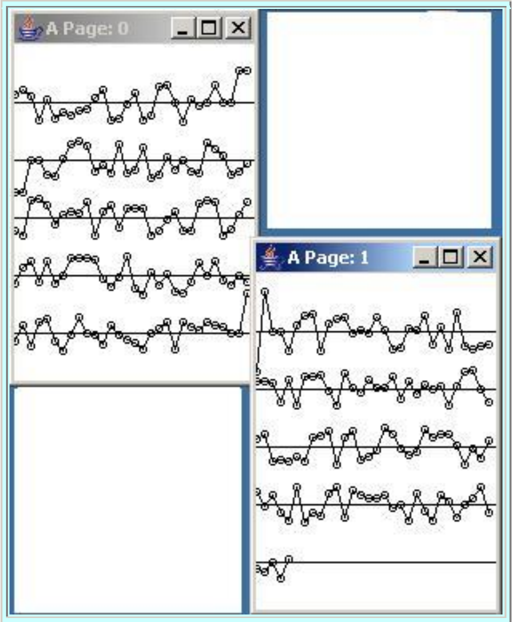


Figure 5

Default plotting parameters are used for the second plotting object and the stack of pages is located in the default location at the upper left corner of the screen. The output produced by this plotting object was shown earlier in Figure 1.

The data to be plotted

Most of the data that is fed to each plotting object is white random noise produced by a random noise generator. However, fifteen of the data values fed to the first plotting object are not random.

Transition from trace to trace on the same page

Eight of the data values for the first plotting object are set to 0,0,20,20,-20,-20,0,0. The result can be seen at the end of the first trace and the beginning of the second trace in Page 0 in Figure 5. Note that the last four plotted points for the first trace have values of 0,0,20, and 20. Then note that the first four plotted points on the second trace have values of -20, -20, 0, and 0. This confirms the proper transition from one trace to the next on the same page with no loss of data values in the transition.

Transition from page to page

Seven of the values for the first plotting object are set to values of 0,0,25,-25,25,0,0. The result can be seen at the end of the last trace on Page 0 and the beginning of the first trace on Page 1 in Figure 5. Note that the last three plotted points in the last trace on Page 0 have values of 0, 0, and 25. Then note that the first four plotted points in the first trace on Page 1 have values of -25, 25, 0, and 0. This confirms the proper transition from one page to the next with no loss of data in the transition.

(The two pages in Figure 5 were manually arranged as shown before capturing the screen shot to emphasize the transition of the data from one page to the next. The large white rectangles in Figure 5 are the result of removing the background clutter in the image caused by icons on the desktop.)

Proper locations for AWT Frame under WinXP

These specific values and the locations in the data where they are placed provide visible confirmation that the transitions mentioned above are handled correctly by the plotting object. These are the correct locations for an AWT **Frame** object for Java running under WinXP. Note however that a **Frame** may have different **inset** values under other operating systems, which may cause these specific locations to be incorrect for that operating system. In that case, the values will be plotted but they won't necessarily confirm the proper transition.

Information about plotting parameters

Information about the plotting parameters for each plotting object is displayed on the command line screen when this class is used for plotting. The values shown below result from the execution of the **main** method of the **PlotALot01** class for test purposes. One of the plotting objects instantiated by the **main** method is entitled "A" and the other is entitled "B".

```
Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36
Sample spacing: 5
Traces per page: 5
Samples per page: 150

Title: B
Frame width: 400
Frame height: 410
Page width: 392
Page height: 383
Trace spacing: 50
Sample spacing: 2
Traces per page: 7
Samples per page: 1372
```

The graphic output produced for the object entitled "A" is shown in Figure 5. This output was based on plotting format parameters that were passed to the constructor. The graphic output produced for the object entitled "B" is shown in Figure 1. This output was based on default plotting parameters.

Overloaded plotData method

There are two overloaded versions of the **plotData** method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. This version requires two parameters, which are coordinate values in pixels. The first parameter specifies the horizontal coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen. The second parameter specifies the vertical coordinate of the upper left corner of the stack of pages relative to the upper left corner of the screen.

> *(Specifying coordinate values of 0,0 causes the stack to be located in the upper left corner of the screen. Positive vertical coordinates progress down the screen.)*

The other overloaded version of **plotData** places the stack of pages in the upper left corner of the screen by default.

A WindowListener for program termination

Each page has a **WindowListener** that will terminate the program if the user clicks the close button on the **Frame**.

J2SE 5.0 is required

The class was tested using J2SE 5.0 and WinXP. J2SE 5.0 is required because the class uses generics with an **ArrayList** object.

Let's see some code!

I will present and explain this class in fragments. A complete listing of the class is provided in Listing 35 near the end of the lesson.

The main method

As mentioned earlier, this class contains a **main** method. The **main** method is provided so that the class can be run as an application for self-test purposes. The **main** method also illustrates the proper use of the class.

The beginning of the class and the beginning of the **main** method is shown in Listing 1.

```
public class PlotALot01{
  public static void main(String[]
args){
    PlotALot01 plotObjectA =
            new
PlotALot01("A",158,237,36,5,4,4);
    PlotALot01 plotObjectB = new
PlotALot01("B");
```

Listing 1

Instantiate two plotting objects

Listing 1 instantiates two independent plotting objects. The first plotting object, referred to by **plotObjectA** is instantiated by invoking the constructor that accepts plotting parameters. A description of each of the constructor parameters was provided earlier. You may find it useful to compare the values shown in Listing 1 with the overall plotting parameters listed earlier to confirm how they are related.

The second plotting object, referred to by **plotObjectB** is instantiated by invoking the constructor that accepts only the page title as a parameter and uses default values for all of the plotting parameters. You will see those default values later in the code.

Feed the plotting object entitled "A"

Listing 2 contains a **for** loop that feeds 275 values to the plotting object entitled "A". Most of the code in Listing 2 is required to set fifteen specific values to test for proper transitions as described earlier. This code is straightforward and shouldn't require further explanation.

> *(I was able to determine the correct locations for these values by knowing the size of the* ***Frame****,* ***inset*** *values for the* ***Frame****, the space between traces, the number of pixels dedicated to each sample, etc.)*

```
    for(int cnt = 0;cnt < 275;cnt++){
      if(cnt == 147){
        plotObjectA.feedData(0);
      }else if(cnt == 148){
        plotObjectA.feedData(0);
      }else if(cnt == 149){
        plotObjectA.feedData(25);
      }else if(cnt == 150){
        plotObjectA.feedData(-25);
      }else if(cnt == 151){
        plotObjectA.feedData(25);
      }else if(cnt == 152){
        plotObjectA.feedData(0);
      }else if(cnt == 153){
        plotObjectA.feedData(0);
      }else if(cnt == 26){
        plotObjectA.feedData(0);
      }else if(cnt == 27){
        plotObjectA.feedData(0);
      }else if(cnt == 28){
        plotObjectA.feedData(20);
      }else if(cnt == 29){
        plotObjectA.feedData(20);
      }else if(cnt == 30){
        plotObjectA.feedData(-20);
      }else if(cnt == 31){
        plotObjectA.feedData(-20);
      }else if(cnt == 32){
        plotObjectA.feedData(0);
      }else if(cnt == 33){
        plotObjectA.feedData(0);
      }else{
        plotObjectA.feedData(
                       (Math.random()
- 0.5)*25);
      }//end else
    }//end for loop

Listing 2
```

White random noise

The final statement in Listing 2 uses a random number generator to feed white random noise to the plotting object for all data values other than the fifteen data values specified in the preceding statements. You can see the random values plotted and marked by round ovals in Figure 5.

Plot the data

The statement in Listing 3 invokes the overloaded **plotData** method to cause all of the pages belonging to the plotting object entitled "A" to be stacked in a location where the upper left corner of the stack is 401 pixels to the right of the upper left corner of the screen.

```
    plotObjectA.plotData(401,0);

Listing 3
```

As described earlier, page 0 containing the earliest data fed to the plotting object is on the top of the stack. Figure 1 shows the two pages belonging to this plotting object after they have been manually rearranged to make them both visible.

Feed and plot the object entitled "B"

Listing 4 feeds 2600 random white noise values to the object entitled "B" and displays the pages in the default location in the upper left corner of the screen. Listing 4 also signals the end of the **main** method.

```
    for(int cnt = 0;cnt < 2600;cnt++){
      plotObjectB.feedData(
                       (Math.random()
- 0.5)*25);
    }//end for loop
    plotObjectB.plotData();

  }//end main

Listing 4
```

Listing 4 *(plus one of the statements in Listing 1)* is much more typical of the amount of code required to use this plotting class than was the case with Listing 2.

(Almost all of the code in Listing 2 was required to set the special data values used to test the transitions discussed earlier.)

The three steps

To recap, the three steps required to use this class for plotting nearly unlimited amounts of data are:

1. Instantiate a plotting object of the class named **PlotALot01**, as in Listing 1.
2. Invoke the **feedData** method once for each data value that is to be plotted, as in Listing 4.
3. Invoke the **plotData** method on the plotting object after all of the data has been fed to the plotting object, as in Listing 3 or Listing 4.

Some instance variables

Continuing with the class definition for the class named **PlotALot01**, Listing 5 shows several instance variables that belong to a plotting object instantiated from this class.

```
  String title;
  int frameWidth;
  int frameHeight;
  int traceSpacing;//pixels between
traces
  int sampSpacing;//pixels between
samples
  int ovalWidth;//width of sample
marking oval
  int ovalHeight;//height of sample
marking oval

  int tracesPerPage;
  int samplesPerPage;
  int pageCounter = 0;
  int sampleCounter = 0;
  ArrayList <Page> pageLinks =
                           new
ArrayList<Page>();
```

Listing 5

The purpose of each of these instance variables is indicated by the name of the variable, and in some cases by the comments following the variable declaration. In addition, I will have more to say about some of these variables later when I discuss the code that uses them.

*(Note the use of generics in the declaration and initialization of the variable named **pageLinks**. The use of generics dictates that this class requires J2SE 5.0 or later.)*

The first overloaded constructor

As mentioned earlier, there are two overloaded versions of the constructor for this class. The overloaded version that begins in Listing 6 accepts several incoming parameters allowing the user to control various aspects of the plotting format.

> *(A different overloaded version, which I will discuss later, accepts a title string only and sets all of the plotting parameters to default values.)*

```
  PlotALot01(String title,//Frame
title
             int frameWidth,//in
pixels
             int frameHeight,//in
pixels
             int traceSpacing,//in
pixels
             int sampSpace,//in pixels
per sample
             int ovalWidth,//sample
marker width
             int ovalHeight)//sample
marker hite
  {//constructor
```

Listing 6

Listing 6 shows the signature for this overloaded version of the constructor. The comments should make the code self explanatory.

Save the parameter values

With one exception, the code in Listing 7 simply saves the incoming parameter values in instance variables to make those values available to other members of the class.

```
    this.title = title;
    this.frameWidth = frameWidth;
    this.frameHeight = frameHeight;
    this.traceSpacing = traceSpacing;
    //Convert to pixels between
samples.
    this.sampSpacing = sampSpace - 1;
    this.ovalWidth = ovalWidth;
    this.ovalHeight = ovalHeight;
```

Listing 7

The exception

The exception has to do with the parameter named **sampSpace**. This parameter is provided by the user in units of pixels per sample, because that seems to be the most natural way for a human to specify this plotting parameter. However, for computational purposes, it is better to have the value of the number of pixels between samples, which is one less than the number of pixels per sample. This conversion is made during the saving of this parameter in Listing 7.

A temporary Page object

As you will see later, the **Page** class consists of a **Canvas** contained in an AWT **Frame**. Because an AWT **Frame** takes on the look and feel of the operating system under which the program is running, it may be constructed differently under different operating systems. Many important plotting parameters depend on the size of the **Canvas**, which depends on the values of the **insets** for the **Frame** for that particular operating system.

(A good exercise would be for you to convert this class to Swing using a look and feel that is independent of the operating system.)

The code in Listing 8 instantiates a temporary **Page** object solely for the purpose of obtaining information about the width and the height of the **Canvas** object. This information is used later to compute a variety of other important parameter values.

```
    Page tempPage = new Page(title);
    int canvasWidth =
tempPage.canvas.getWidth();
    int canvasHeight =

tempPage.canvas.getHeight();
```

Listing 8

Display some information

Listing 9 gets and displays information about the plotting object on the command line screen. An example of this output was shown earlier.

```
    //Display information about this
plotting
    // object.
    System.out.println("\nTitle: " +
title);
    System.out.println(
          "Frame width: " +
tempPage.getWidth());
    System.out.println(
        "Frame height: " +
tempPage.getHeight());
    System.out.println(
                   "Page width: " +
canvasWidth);
    System.out.println(
                 "Page height: " +
canvasHeight);
    System.out.println(
                "Trace spacing: " +
traceSpacing);
    System.out.println(
         "Sample spacing: " +
(sampSpacing + 1));
    if(sampSpacing < 0){

System.out.println("Terminating");
      System.exit(0);
    }//end if

Listing 9
```

Terminate on negative value for sampSpacing

In addition, Listing 9 tests to confirm that the number of pixels between samples is not a negative value. If it is a negative value, Listing 9 terminates the program immediately after the number of pixels per sample has been displayed

Dispose of the temporary Page object

Once the width and height of the **Canvas** has been determined, the temporary **Page** object is no longer needed. Listing 10 disposes of that object freeing all of the resources dedicated to the object.

```
    tempPage.dispose();

Listing 10
```

Compute and display the remaining plotting parameters

Having determined the width and height of the **Canvas**, Listing 11 computes and displays the remaining plotting parameters. The expressions used to compute these values are straightforward and shouldn't require further explanation.

```
    tracesPerPage =
                 (canvasHeight -
traceSpacing/2)/

traceSpacing;
    System.out.println("Traces per
page: "
                                +
tracesPerPage);
    if(tracesPerPage == 0){
      System.out.println("Terminating
program");
      System.exit(0);
    }//end if
    samplesPerPage = canvasWidth *
```

```
tracesPerPage/

(sampSpacing + 1);
    System.out.println("Samples per
page: "
                                + 
samplesPerPage);
```

Listing 11

Terminate on zero traces per page

In addition, Listing 11 terminates the program if it is determined that the number of traces per page is equal to zero. The reason for this should be obvious to the reader. If termination does occur, it occurs immediately after the number of traces per page has been displayed.

Instantiate first usable Page object

Listing 12 instantiates the first usable **Page** object. *(Recall that a temporary **Page** object was instantiated and disposed of earlier.)* This **Page** object will be entitled *title Page: 0* as indicated in Figure 1 and Figure 5.

```
    pageLinks.add(new Page(title));
  }//end constructor
```

Listing 12

Note that a reference to this **Page** object *(and all subsequently instantiated **Page** objects)* is saved in an object of type **ArrayList** referred to by **pageLinks**.

Listing 12 also signals the end of this constructor for the **PlotALot01** class.

The other overloaded constructor

A second overloaded constructor is provided for those who don't want to have to think about plotting parameters. This constructor, which is shown in its entirety in Listing 13, establishes a set of default plotting parameters.

> *(In case you are unfamiliar with the use of the keyword **this** to cause one constructor to invoke another constructor of the same class, you can learn about that topic in my earlier lesson entitled The Essence of OOP using Java, The this and super Keywords.)*

```
  PlotALot01(String title){
    this(title,400,410,50,2,2,2);
  }//end overloaded constructor
```

Listing 13

The default plotting parameter values

This is where the default plotting parameters are specified as having the following values:

- frameWidth: 400
- frameHeight: 410
- traceSpacing: 50
- sampSpace: 2
- ovalWidth: 2
- ovalHeight: 2

As mentioned earlier, these values were chosen mainly to be compatible with this narrow publication format. You should feel free to change the default values to a set of values that is more consistent with your needs. For example, if you plan to plot and examine very large amounts of data, you might want to consider setting the **frameWidth** and **frameHeight** to completely fill the screen on your computer. Then you can examine large amounts of data without the need to skip from one page to the next.

The feedData method

The **feedData** method must be invoked on the plotting object once for each data value that is to be plotted. This method is shown in its entirety in Listing 14.

```
  void feedData(double val){
    if((sampleCounter) ==
samplesPerPage){
      pageCounter++;
      sampleCounter = 0;
      pageLinks.add(new Page(title));
    }//end if

pageLinks.get(pageCounter).putData(

val,sampleCounter);
    sampleCounter++;
  }//end feedData
```

Listing 14

Scaling of data to be plotted

The **feedData** method receives an incoming data value of type **double**. This is probably a good time to point out that the data must be properly scaled for plotting before it is passed to this method.

The incoming **double** value will later be cast to type **int**. As you should already know, if the **double** value is too large to fit in type **int**, the value resulting from the cast will be indeterminate.

In reality, however, the cast shouldn't be a problem. I'm unaware of any computer monitor whose vertical dimension is greater than a few thousand pixels. Regardless of the size of the **Page** object, a data value whose magnitude is greater than a few thousand units will be completely off the screen when plotted. Therefore, depending on the resolution of the monitor, the maximum magnitudes of the incoming data values should probably have been scaled to 1000 or less to be suitable for plotting.

Is the page full?

Listing 14 first checks to see if the current page is full before attempting to plot the new data value. If the page is full, Listing 14 increments the page counter, resets the sample counter, and instantiates a new **Page** object.

Save the data value for plotting later

All of the data values are stored in array objects of type **double** as they are fed to the plotting object. Later, when it is time to display the plotted version of the data, an overridden **paint** method accesses that data and produces the plot.

The MyCanvas class, a preview

Each **Page** object contains an object of a class named **MyCanvas**, which extends the **Canvas** class. Each **MyCanvas** object owns an array object in which the **double** data values to be plotted on that page are stored.

The **MyCanvas** class overrides the **paint** method to cause it to plot the data stored in the array whenever the overridden version of the **paint** method is invoked.

> *(If you are familiar with graphics in Java, you will already know that the overridden **paint** method can be invoked for a variety of reasons, such as covering and later uncovering the page. If you are not familiar with graphics in Java, I discuss the overriding of the **paint** method in numerous earlier lessons including several lessons on animation in Java.)*

I will have much more to say about the class named **MyCanvas** later.

Invoke the putData method to store the data value

For now, simply be aware that the **feedData** method in Listing 14 invokes the **putData** method on the current **Page** object to cause the data value to be stored in the array object belonging to the corresponding **MyCanvas** object. The current value of the sample counter is also passed to the **putData** method to specify the array element into which the data value is to be stored.

Finally, the **feedData** method increments the sample counter and returns to await being invoked to receive the next data sample.

The plotData method

The **plotData** method must be invoked once when all of the data has been fed to the plotting object by way of the **feedData** method. The purpose of the **plotData** method is to rearrange the **Page** objects in a stack on the screen with page 0 *(containing the earliest data)* on the top of the stack.

Having rearranged the **Page** objects, the **plotData** method causes the object on the top of the stack to become visible. This, in turn, causes its overridden **paint** method to be invoked causing the data to be plotted as shown in Figure 1 and Figure 5.

Two overloaded versions

There are two overloaded versions of the **plotData** method. One version allows the user to specify the location on the screen where the stack of plotted pages will appear. The other version places the stack in the upper left corner of the screen by default.

Specify the location of the stack

The version of the **plotData** method that allows the user to specify the location begins in Listing 15. This version receives two incoming parameters. These parameters specify the coordinates of the upper left corner of the stack of **Page** objects relative to the upper left corner of the screen. The first parameter specifies the horizontal coordinate and the second parameter specifies the vertical coordinate, with positive vertical values going down the screen.

> *(Specifying both coordinate values as 0 will cause the stack to appear in the upper left corner of the screen.)*

Make certain that the last page is visible

As you will see later, each of the pages are displayed on the screen as they are produced. It is possible that this method could be called before the operating system has completed the process of making the last page visible. The **plotData** method delays until the last page has become visible on the screen.

```
  void plotData(int xCoor,int yCoor){
    Page lastPage =

pageLinks.get(pageLinks.size() - 1);
    while(!lastPage.isVisible()){
      //Loop until last page becomes
visible
    }//end while loop
```

Listing 15

At this point, the pages appear on the screen with the last page on the top of the stack. This order needs to be reversed to cause the first page to be on the top of the stack.

Make all pages invisible

The reversal of the order of the pages in the stack is accomplished by first making every page invisible, and then making them visible again in reverse order.

The code in Listing 16 iterates on the **ArrayList** object containing references to all of the pages. The reference to each **Page** object is obtained from the list, and its **visible** property value is set to **false**.

```
    Page tempPage = null;
    for(int cnt = 0;cnt <
(pageLinks.size());

cnt++){
      tempPage = pageLinks.get(cnt);
      tempPage.setVisible(false);
    }//end for loop
```

Listing 16

Make the pages visible in reverse order

The code in Listing 17 iterates on the **ArrayList** object again, accessing the references to the **Page** objects in reverse order and setting the value of the **visible** property for each **Page** object to **true**. This results in page 0 *(the page with the earliest data)* being on the top of the stack.

```
    for(int cnt = pageLinks.size() -
1;cnt >= 0;

cnt--){
      tempPage = pageLinks.get(cnt);

tempPage.setLocation(xCoor,yCoor);
      tempPage.setVisible(true);
    }//end for loop

  }//end plotData(int xCoor,int yCoor)
```

Listing 17

Set the location of the stack

In addition, the code in Listing 17 sets the **location** property of each **Page** object to the coordinate values received as incoming parameters to the **plotData** method. This causes the stack of **Page** objects to appear in the specified location on the screen.

The other overloaded version of the plotData method

The other overloaded version of the **plotData** method is shown in its entirety in Listing 18.

```
  void plotData(){
    plotData(0,0);//invoke overloaded
version
  }//end plotData()
```

Listing 18

This version receives no incoming parameters. The body of the method simply invokes the first overloaded version discussed above, passing coordinate values as 0,0 as parameters. As explained earlier, this causes the stack of **Page** objects to appear in the upper left corner of the screen.

The Page class

The **Page** class is a member class of the class named **PlotALot01**. As such, methods belonging to objects of the **Page** class have direct access to all of the other members of the enclosing **PlotALot01** object, including instance variables belonging to the **PlotALot01** object.

(If you are unfamiliar with member classes, see the lesson entitled The Essence of OOP using Java, Member Classes.)

Potentially many Page objects...

A **PlotALot01** object may own as many **Page** objects as are required to plot all of the data values that are fed to it.

*(The reference to each **Page** object is stored in an **ArrayList** object belonging to the **PlotALot01** object.)*

The **Page** class, which extends the **Frame** class begins, in Listing 19. The constructor for the **Page** class also begins in Listing 19.

```
  class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){//constructor
      canvas = new MyCanvas();
      add(canvas);
      setSize(frameWidth,frameHeight);
      setTitle(title + " Page: " +
pageCounter);
      setVisible(true);
```

Listing 19

The **Page** class begins by declaring two instance variables. The instance variable named **canvas** will hold a reference to an object of the **MyCanvas** class upon which the data will actually be plotted.

The other instance variable will hold the value of a sample counter.

The constructor for the Page class

The constructor that begins in Listing 19 instantiates an object of a member class named **MyCanvas** and stores its reference in the reference variable named **canvas**. Then it adds the **MyCanvas** object to the *center* location of the **Page *(Frame)***.

Following this, the constructor accesses the variables named **frameWidth** and **frameHeight** belonging to the enclosing **PlotALot01** object and uses those values to set the size of the **Page**.

Then the constructor accesses the **title** and **pageCounter** variables belonging to the enclosing **PlotALot01** object and uses those values to set the title for the **Page** object.

Finally, the code in Listing 19 causes the **Page** object to become visible on the screen.

An anonymous terminator for the Page class

Still inside the constructor, Listing 20 instantiates an anonymous class to terminate the program when the user clicks on the **close** button on the **Page**.

(In case you are unfamiliar with anonymous inner classes, see my earlier lesson entitled The Essence of OOP using Java, Anonymous Classes.)

```
      addWindowListener(
        new WindowAdapter(){
          public void windowClosing(

WindowEvent e){
            System.exit(0);//terminate
program
          }//end windowClosing()
        }//end WindowAdapter
      );//end addWindowListener
    }//end constructor

Listing 20
```

Listing 20 also signals the end of the constructor for the **Page** class.

The putData method of the Page class

This method, which is shown in its entirety in Listing 21, receives a sample value of type **double** and also receives the sample counter associated with that data value. It uses the value of the sample counter to store the data value in an array object belonging to the **MyCanvas** object.

```
    void putData(double sampleValue,
                 int sampleCounter){
```

```
      canvas.data[sampleCounter] =
sampleValue;
      this.sampleCounter =
sampleCounter;
    }//end putData
```

Listing 21

In addition, the **putData** method saves the sample counter value in an instance variable to make it available to the overridden **paint** method later. This value is needed by the **paint** method so it will know how many samples to plot on the final page which probably won't be full.

The MyCanvas class

The **MyCanvas** class, which begins in Listing 22, is a member class of the **Page** class. As such, methods belonging to an object of the **MyCanvas** class have direct access to the other members of the enclosing **Page** object, including instance variables of the **Page** object. In addition, methods belonging to an object of the **MyCanvas** class have direct access to the other members, including instance variables, of the enclosing **PlotALot01** object.

```
    class MyCanvas extends Canvas{
      double [] data =
                      new
double[samplesPerPage];
```

Listing 22

The class definition begins by creating a new array object of type **double** that will be used to store the data values belonging to the page.

The overridden paint method

The overridden **paint** method of the **MyCanvas** class begins in Listing 23.

```
      public void paint(Graphics g){
        for(int cnt = 0;cnt <
tracesPerPage;

cnt++){
          g.drawLine(0,

(cnt+1)*traceSpacing,
                     this.getWidth(),

(cnt+1)*traceSpacing);
        }//end for loop
```

Listing 23

Draw the horizontal axes

The code in Listing 23 draws a set of horizontal axes on the **MyCanvas** object, one for each trace that will be plotted on the object. These horizontal axes are shown in Figure 1 and Figure 5.

Plot the points

Listing 24 shows the beginning of the code that is used to plot the data values stored in the array that was created in Listing 22.

```
        if(sampleCounter > 0){
          for(int cnt = 0;cnt <=
sampleCounter;

cnt++){
            //Compute a vertical
offset
            int yOffset =
                   (1 +
cnt*(sampSpacing + 1)/

this.getWidth())*traceSpacing;

Listing 24
```

Listing 24 begins by testing the value of the sample counter to make certain that there are some points to be plotted. If so, it enters a **for** loop to plot each data value stored in the array. Because it uses the value of the sample counter to terminate the **for** loop, only those data values that have been stored in the array object will be plotted, even if the array object isn't full.

A vertical offset

The data values in the array are to be plotted on one or more horizontal axes on the page. Therefore, it is necessary to first determine where on the page each data value is to be plotted. The code in Listing 24 uses various pieces of information to determine the vertical location of the axis against which each data value will be plotted.

Draw an oval

The code in Listing 25 draws an oval centered on the sample value to mark the sample on the plot. It is best if the dimensions of the oval are evenly divisible by 2 for centering purposes. Otherwise, the ovals may appear to be a little off center.

```

g.drawOval(cnt*(sampSpacing + 1)%
                   this.getWidth() -
ovalWidth/2,
              yOffset - (int)data[cnt]
                                  -
ovalHeight/2,
              ovalWidth,
              ovalHeight);

Listing 25
```

Is positive vertical up or down?

Normally vertical coordinates increase going down the screen in Java graphics. However, this isn't what most of us are accustomed to seeing when we plot data. We prefer to see increasing vertical coordinates going up the page. The code in Listing 25 reverses the sign on the data values to cause positive data values to be plotted above the axis and negative data values to be plotted below the axis. Increasing values go up. Decreasing values go down.

Connect the points with straight lines

The code in Listing 26 connects the sample values with straight lines. Care is taken to avoid drawing a line from the last sample value on one trace to the first sample value on the next trace. That would really be ugly.

```
            if(cnt*(sampSpacing + 1)%

this.getWidth() >=

sampSpacing + 1){
              g.drawLine(
                (cnt - 1)*(sampSpacing
+ 1)%

this.getWidth(),
                yOffset -
(int)data[cnt-1],
                cnt*(sampSpacing + 1)%

this.getWidth(),
                yOffset -
(int)data[cnt]);
            }//end if
          }//end for loop
        }//end if for sampleCounter >
0
      }//end overridden paint method
    }//end inner class MyCanvas
  }//end inner class Page
}//end class PlotALot01

Listing 26
```

End of the PlotALot01 class

Listing 26 also signals the end of the overridden **paint** method, the end of the **MyCanvas** class, the end of the **Page** class, and the end of the **PlotALot01** class. In short, Listing 26 signals the end of the class under discussion.

The class named PlotALot02

Much of the code in this class is very similar to the code in the class named **PlotALot01**. Therefore, the discussion of this class will be much briefer than the earlier discussion of the class named **PlotALot01**.

Designed for two-channel data

This class is an update to the class named **PlotALot01**. This class is designed to plot large amounts of data for two channels on the same axes as shown in Figure 2. One set of data is plotted using the color black. The other set of data is plotted using the color red.

As is the case for the class named **PlotALot01**, this class provides a **main** method so that the class can be run as an application to test itself.

Three steps to use the class

As before, there are three steps involved in the use of this class for plotting data:

1. Instantiate a plotting object of type **PlotALot02**.
2. Feed pairs of data values to the plotting object by invoking the **feedData** method once for each pair of data values. The first value in the pair will be plotted in black. The second value in the pair will be plotted in red.
3. Invoke the **plotData** method on the plotting object after all of the data has been fed to the object. This causes all of the data to be plotted. It also causes the pages to be rearranged placing page 0 on the top of the stack.

A stack of Page objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen. Each Page object contains one or more horizontal axes on which the data is plotted as shown in Figure 2. The two data sets are superimposed on the same axes with the data from one data set being plotted in black and the data from the other data set being plotted in red.

Testing with the main method

For test purposes, the main method instantiates a single plotting object and feeds two data sets to that plotting object. As before, the data that is fed to the plotting object is white random noise. One of the data sets is the sequence of values obtained from a random number generator. The other data set is the same as the first except that the sign of each data values is reversed.

Some data is not random

Also as before, and for the same reason, fifteen of the data values for each data set are not random. The non-random data values are the same as in the **main** method for the class named **PlotALot01**. Figure 2 illustrates how these fifteen specific values are used to confirm the proper transition from the end of one trace to the beginning of the next trace, and also to confirm the proper transition from the end of one page to the beginning of the next page.

The class named PlotALot02 and the main method

As before, I will discuss this class in fragments. A complete listing of the class is provided in Listing 36 near the end of the lesson. However, because much of the code in this class is very similar to code that I explained for the class named **PlotALot01**, this discussion of the code will be much briefer. I will highlight those aspects of this code that are different from the code in **PlotALot01**.

The beginning of the class and an abbreviated version of the **main** method is provided in Listing 27. Much of the code has been deleted from Listing 27 for brevity.

```
public class PlotALot02{
  public static void main(String[]
args){
    PlotALot02 plotObjectA =
            new
PlotALot02("A",158,237,36,5,4,4);

    for(int cnt = 0;cnt < 275;cnt++){
      double valBlack = (Math.random()
- 0.5)*25;
      double valRed = -valBlack;
      //Feed pairs of values to the
plotting
      // object by invoking the
feedData method
      // once for each pair of data
values.
      if(cnt == 147){
        plotObjectA.feedData(0,0);

      //...code deleted for brevity

      }else{

plotObjectA.feedData(valBlack,valRed);
      }//end else
    }//end for loop
    //Cause the data to be plotted in
the default
    // screen location.
    plotObjectA.plotData();
  }//end main
```

Listing 27

Two data values are required

The important thing to note in Listing 27 is that two data values must be passed to the **feedData** method each time it is invoked. This consists of one data value from each channel of data being plotted.

The feedData method

The modified **feedData** method is shown in its entirety in Listing 28.

```
  void feedData(double valBlack,double
valRed){
    if((sampleCounter) ==
samplesPerPage){
      //if the page is full, increment
the page
      // counter, create a new empty
page, and
      // reset the sample counter.
      pageCounter++;
      sampleCounter = 0;
      pageLinks.add(new Page(title));
    }//end if
    //Store the sample values in the
MyCanvas
    // object to be used later to
paint the
    // screen.  Then increment the
sample
    // counter.  The sample values
pass through
    // the page object into the
current MyCanvas
    // object.

pageLinks.get(pageCounter).putData(

valBlack,valRed,sampleCounter);
    sampleCounter++;
  }//end feedData
```

Listing 28

The most significant things to note about the modified version of the **feedData** method are:

- The method receives two incoming data values as parameters instead of just one.
- The method passes the two data values, along with the sample counter value to the **putData** method of the **PlotALot02** class. Thus, the **putData** method has also been modified to require two incoming data values.

The putData method

The modified **putData** method is shown in its entirety in Listing 29. This modified version of the method receives a pair of data values and stores each of the data values in a different array object belonging to the **MyCanvas** object.

```
    void putData(double
valBlack,double valRed,
                              int
sampleCounter){
      canvas.blackData[sampleCounter]
= valBlack;
      canvas.redData[sampleCounter] =
valRed;
      this.sampleCounter =
sampleCounter;
    }//end putData

Listing 29
```

The MyCanvas class

The modified version of the **MyCanvas** class begins in Listing 30.

```
    class MyCanvas extends Canvas{
      double [] blackData =
                      new
double[samplesPerPage];
      double [] redData =
                      new
double[samplesPerPage];

Listing 30
```

The class begins by creating two different array objects in which incoming data is stored instead of just one array object. These are the objects that are populated by the **putData** method in Listing 29.

The overridden paint method

The modified version of the overridden **paint** method begins in Listing 31. Most of the code was deleted for brevity from Listing 31 because it is very similar to the code in the overridden **paint** method in the class named **PlotALot01**.

```
      public void paint(Graphics g){
        //Draw horizontal axes
        //... code deleted for brevity

        //Plot the points.
        if(sampleCounter > 0){
          for(int cnt = 0;cnt <=
sampleCounter;

cnt++){
            //Compute a vertical
offset.
            //...code deleted for
brevity

            //Begin by plotting the
values from
            // the blackData array
object.
            //Draw an oval.
            g.setColor(Color.BLACK);
            //...code deleted for
brevity

            //Connect the sample
values with
            // straight lines.
            //...code deleted for
brevity

Listing 31
```

Setting the drawing color

The most significant thing in Listing 31 is the invocation of the **setColor** method of the **Graphics** class to set the drawing color to black. Otherwise, the code is essentially the same as the code in the overridden paint method in **PlotALot01**. The code in Listing 31 draws the black traces shown in Figure 2.

New code in the overridden paint method

The overridden version of the **paint** method continues in Listing 32. The code in Listing 32 is essentially all new code that was created to plot the second data set in red. However, the only real difference between this code and code that I explained earlier with respect to the class named **PlotALot01** is:

- The drawing color has been set to red instead of the default color of black.
- The data being plotted is the second set of data. This data is stored in the array object referred to by **redData**.

Otherwise, this code is essentially the same as the code that was used to plot the single data set in the overridden paint method in the class named **PlotALot01**.

```
            //Now plot the data stored
in the
            // redData array object.
            g.setColor(Color.RED);
            //Draw the ovals as
described above.

g.drawOval(cnt*(sampSpacing + 1)%
                   this.getWidth() -
ovalWidth/2,
              yOffset -
(int)redData[cnt]
                                  -
ovalHeight/2,
              ovalWidth,
              ovalHeight);

            //Connect the sample
values with
            // straight lines as
described above.
            if(cnt*(sampSpacing + 1)%

this.getWidth() >=

sampSpacing + 1){
              g.drawLine(
                (cnt - 1)*(sampSpacing
+ 1)%

this.getWidth(),
                yOffset -
(int)redData[cnt-1],
                cnt*(sampSpacing + 1)%

this.getWidth(),
                yOffset -
(int)redData[cnt]);

            }//end if
          }//end for loop
        }//end if for sampleCounter >
0
      }//end overridden paint method
    }//end inner class MyCanvas
  }//end inner class Page
}//end class PlotALot02

Listing 32
```

The code in Listing 32 draws the red traces shown in Figure 2.

End of class PlotALot02

Listing 32 signals the end of the overridden **paint** method, the **MyCanvas** class, the **Page** class, and the **PlotALot02** class.

The class named PlotALot03

I will discuss the class named **PlotALot03** in fragments. A complete listing of the class is provided in Listing 37 near the end of the lesson.

Much of the code in the class named **PlotALot03** is very similar to the code in **PlotALot02**. Therefore, this discussion will be brief, simply highlighting the differences between the two classes.

Two-channel data on alternating axes

This class is an update to the class named **PlotALot02**. This class is designed to plot large amounts of data for two channels on alternating horizontal axes. One set of data is plotted using the color black. The other set of data is plotted using the color red.

Three steps for using the class

As before, there are three steps involved in the use of this class for plotting two-channel data:

1. Instantiate a plotting object of **typePlotALot03**.
2. Feed pairs of data values to the plotting object by invoking the **feedData** method once for each pair of data values. The first value in the pair will be plotted in black on one axis. The second value in the pair will be plotted in red on an axis below that one.
3. Invoke the **plotData** method on the plotting object when all of the data has been fed to the object. This causes all of the data to be plotted and also causes the **Page** objects to be rearranged so that page 0 is on the top of the stack.

A stack of Page objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen, with the data plotted on a **Canvas** object contained by the **Page** object.

Each **Page** object contains two or more horizontal axes on which the data is plotted. The class will terminate if the number of axes on the page is an odd number.

Alternating axes

The two data sets are plotted on alternating axes as shown in Figure 3 with the data from one data set being plotted in black on one axis and the data from the other data set being plotted in red on the axis below that axis.

The earliest data is plotted on the pair of axes nearest the top of the **Page** moving from left to right across the page. Positive data values are plotted above the axis and negative values are plotted below the axis.

When the right end of an axis is reached, the next data value is plotted on the left end of the second axis below it skipping one axis in the process. When the right end of the last pair of axes on the **Page** is reached, a new **Page** object is created and the next pair of data values are plotted at the left end of the top pair of axes on that new **Page**.

Testing with the main method

For test purposes, the **main** method instantiates a single plotting object and feeds two data sets to that plotting object. The data that is fed to the plotting object is white random noise. One of the data sets is the sequence of values obtained from a random number generator. The other data set is the same as the first. Thus, the pairs of black and red data sets that are plotted should have the same shape making it easy to confirm that the process of plotting the two data sets is behaving the same in both cases.

Some data is not random

Fifteen of the data values for each data set are not random for the same reasons discussed earlier. Figure 3 shows how these specific values confirm proper transition from one trace to the next on the same page and confirm the proper transition from one page to the next.

Modified constructor code

The first code that I will highlight as being different from the code in the class named **PlotALot02** is shown in Listing 33. This code appears in the modified constructor for the **PlotALot03** class.

```
    if((tracesPerPage == 0) ||

(tracesPerPage%2 != 0) ){
      System.out.println("Terminating
program");
      System.exit(0);
    }//end if

    samplesPerPage = canvasWidth *
tracesPerPage/

(sampSpacing + 1)/2;
```

Listing 33

The **if** statement in Listing 33 confirms that the number of traces per page is evenly divisible by two. If not, the program terminates.

The last statement in Listing 33 computes the value of **samplesPerPage** taking into account that only half as many samples from each data set can be plotted on a page as is the case when the plots of the two data sets are superimposed on the same axes in the class named **PlotALot02**.

The overridden paint method

Additional code that I will highlight as being different is in the overridden **paint** method of the **MyCanvas** class. This code is shown in Listing 34, and the code that is different is highlighted in boldface.

```
      public void paint(Graphics g){
        //Draw horizontal axes
        //...code deleted for brevity

        //Plot the points
        if(sampleCounter > 0){
          for(int cnt = 0;cnt <=
sampleCounter;

cnt++){

            //Plot values from the
blackData
            // array object.
            g.setColor(Color.BLACK);

            //Compute a vertical
offset to locate
            // the black data on the
odd numbered
            // axes on the page.
            int yOffset =
               ((1 + cnt*(sampSpacing
+ 1)/

this.getWidth())*2*traceSpacing)
                                  -
traceSpacing;

            //Draw an oval
            //...code deleted for
brevity
            //Connect the sample
values with
            // straight lines.
            //...code deleted for
brevity

            //Plot the data stored in
the
            // redData array object.
            g.setColor(Color.RED);
            //Compute a vertical
offset to locate
            // the red data on the
even numbered
            // axes on the page.
            yOffset = (1 +
cnt*(sampSpacing + 1)/
this.getWidth())*2*traceSpacing;

            //Draw the ovals
            //...code deleted for
brevity
            //Connect the sample
values with
            // straight lines
            //...code deleted for
brevity

          }//end for loop
        }//end if for sampleCounter >
0
      }//end overridden paint method
    }//end inner class MyCanvas
  }//end inner class Page
}//end class PlotALot02

Listing 34
```

Code was deleted for brevity

Most of the code in the overridden **paint** method is the same as the code that I discussed earlier and was deleted from Listing 34 for brevity.

The code that is different is the code that computes the vertical offset values to locate the black data on the odd numbered axes and to locate the red data on the even numbered axes as shown in Figure 3. I will let you work through the expressions in Listing 34 on your own and convince yourself that the code is correct.

End of class PlotALot03

Listing 34 signals the end of the overridden **paint** method, the **MyCanvas** class, the **Page** class, and the **PlotALot03** class.

The class named PlotALot04

This class is an update to the class named **PlotALot03**. This class is designed to plot large amounts of three-channel data on separate horizontal axes. One set of data is plotted using the color black. The second set of data is plotted using the color red. The third set of data is plotted using the color blue.

The class provides a main method so that the class can be run as an application to test itself.

Three steps for using the class

There are three steps involved in the use of this class for plotting data:

- Instantiate a plotting object of type **PlotALot04**.
- Feed triplets of data values to the plotting object by invoking the **feedData** method once for each triplet of data values. The first value in the triplet will be plotted in black on one axis. The second value in the triplet will be plotted in red on an axis below that axis. The third value in the triplet will be plotted in blue on an axis below that one.
- Invoke the **plotData** methods on the plotting object when all of the data has been fed to the object.

A stack of Page objects

The class produces a graphic output consisting of a stack of **Page** objects on the screen, with the data plotted on a **Canvas** object contained by the **Page** object. The Page showing the earliest data is on the top of the stack and the **Page** showing the latest data is on the bottom of the stack.

Each Page object contains three or more horizontal axes on which the data is plotted. The class will terminate if the number of axes on the page is not evenly divisible by 3.

The three data sets are plotted on separate axes as shown in Figure 4 with the data from one data set being plotted in black on one axis, the data from the second data set being plotted in red on the axis below that axis, and the data from the third data set being plotted in blue on the axis below that axis.

Testing with the main method

For test purposes, the main method instantiates a single plotting object and feeds three data sets to that plotting object producing the graphic output shown in Figure 4.

Won't discuss the code

The code in this class is so similar to the code in the class named **PlotALot03** that I'm not going to discuss the code. You will find a complete listing of the class in Listing 38 near the end of the lesson.

Run the Programs

I encourage you to copy, compile, and run the programs that you will find in Listing 35 through Listing 38 below.

Modify the programs and experiment with them in order to learn as much as you can about the use of Java for plotting large quantities of data. For example, you might want to modify the default plotting parameters to a different set of plotting parameters that are more to your liking. One possibility is to cause the default **Page** size to fill the entire screen on your computer.

Another good exercise would be for you to convert this class to Swing using a look and feel that is independent of the operating system.

What's Next?

I will be using these four classes in a variety of future lessons involving such complex topics as adaptive signal processing using Java.

Summary

In this lesson, I presented and explained four self-testing classes for plotting large quantities of data. One class plots a nearly unlimited amount of single-channel data using multiple traces on multiple pages.

(I have successfully plotted two million data values in 141 full screen pages on a modest laptop computer with no difficulty whatsoever. When I pushed that total up to eight million data values in 563 full screen pages, the plotting process slowed down, but I was still able to display and examine the plots. The practical limit on my computer seems to be somewhere between two million and eight million data values.)

A second class plots a large quantity of two-channel data superimposing the two data sets on the same axes with the plot of one data set being colored black and the plot of the other data set being colored red.

A third class also plots a large quantity of two-channel data, but with this class, the two sets of data are plotted on alternating horizontal axes. Again, one set of data is colored black and the other set is colored red.

A fourth class plots a large quantity of three-channel data on separate axes. In this case, one set is colored black, the second set is colored red, and the third set is colored blue.

Complete Program Listings

Complete listings of the four programs that I explained in this lesson are provided in Listing 35 through Listing 38 below.

```
/*File PlotALot01.java
Copyright 2005, R.G.Baldwin
This program is designed to plot large amounts of
time-series data for a single channel.  See
PlotALot02.java for a two-channel program.

Note that by carefully adjusting the plotting
parameters, this program could also be used to
plot large quantities of spectral data in a
waterfall display.

The class provides a main method so that the
class can be run as an application to test
itself.
```

```
There are three steps involved in the use of this
class for plotting time series data:
1. Instantiate a plotting object of type
   PlotALot01 using one of two overloaded
   constructors.
2. Feed data that is to be plotted to the
   plotting object by invoking the feedData
   method once for each data value.
3. Invoke one of two overloaded plotData methods
   on the plotting object once all of the data
   has been fed to the object.  This causes all
   of the data to be plotted.

A using program can instantiate as many
plotting objects as are needed to plot all of the
different time series that need to be plotted.
Each plotting object can be used to plot as many
data values as need be plotted until the program
runs out of available memory.

The plotting object of type PlotALot01 owns one
or more Page objects that extend the Frame class.
The plotting object can own as many Page objects
as are necessary to plot all of the data that is
fed to that plotting object.

The program produces a graphic output consisting
of a stack of Page objects on the screen, with
the data plotted on a Canvas object contained by
the Page object.  The Page showing the earliest
data is on the top of the stack and the Page
showing the latest data is on the bottom of the
stack.  The Page objects on the top of the stack
must be physically moved in order to see the
Page objects on the bottom of the stack.

Each Page object contains one or more horizontal
axes on which the data is plotted.  The earliest
data is plotted on the axis nearest the top of
the Page moving from left to right across the
axis.  Positive data values are plotted above
the axis and negative values are plotted below
the axis.  When the right end of an axis is
reached, the next data value is plotted on the
left end of the axis immediately below it.  When
the right end of the last axis on the Page is
reached, a new Page object is created and the
next data value is plotted at the left end of the
top axis on that Page object.

A mentioned above, there are two overloaded
versions of the constructor for the PlotALot01
class. One overloaded version accepts several
incoming parameters allowing the user to control
various aspects of the plotting format. A second
```

```
overloaded version accepts a title string only
and sets all of the plotting parameters to
default values. You can easily modify these
default values and recompile the class if you
prefer different default values.

The parameters for the version of the constructor
that accepts plotting format information are:

String title: Title for the Frame object. This
 title is concatenated with the page number and
 the result appears in the banner at the top of
 the Frame.
int frameWidth:The Frame width in pixels.
int frameHeight: The Frame height in pixels.
int traceSpacing: Distance between trace axes in
 pixels.
int sampSpace: Number of pixels dedicated to each
 data sample in pixels per sample.  Must be 1 or
 greater.
int ovalWidth: Width of an oval that is used to
 mark the sample value on the plot.
int ovalHeight: Height of an oval that is used to
 mark the sample value on the plot.

For test purposes, the main method instantiates
and feeds two independent plotting objects.
Plotting parameters are specified for the first
plotting object. Default plotting parameters are
accepted for the second plotting object.

The data that is fed to each plotting object is
white random noise. However, for the first
plotting object, fifteen of the data values are
not random.  Rather, seven of the values are set
to values of 0,0,25,-25,25,0,0 to confirm the
proper transition from the end of one page to the
beginning of the next page. In addition, eight of
the values are set to 0,0,20,20,-20,-20,0,0 in
order to confirm the proper transition from one
trace to the next trace on the same page.

These specific values and the locations in the
data where they are placed provide visible
confirmation that the transitions mentioned above
are handled correctly. Note, however that these
are the correct locations for an AWT Frame object
under WinXP. A Frame may have different inset
values under other operating systems, which may
cause these specific locations to be incorrect
for that operating system.  In that case, the
values will be plotted but they won't confirm
the proper transition.

The following information about the plotting
parameters for each plotting object is displayed
```

```
on the command line screen when the class is used
for plotting.  The values shown below result from
the execution of the main method of the class for
test purposes. One of the plotting objects
instantiated by the main method is entitled "A"
and the other is entitled "B".

Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36
Sample spacing: 5
Traces per page: 5
Samples per page: 150

Title: B
Frame width: 400
Frame height: 410
Page width: 392
Page height: 383
Trace spacing: 50
Sample spacing: 2
Traces per page: 7
Samples per page: 1372

There are two overloaded versions of the plotData
method. One version allows the user to specify
the location on the screen where the stack of
plotted pages will appear. This version requires
two parameters, which are coordinate values in
pixels.  The first parameter specifies the
horizontal coordinate of the upper left corner of
the stack of pages relative to the upper left
corner of the screen.  The second parameter
specifies the vertical coordinate of the upper
left corner of the stack of pages relative to the
upper left corner of the screen. Specifying
coordinate values of 0,0 causes the stack to be
located in the upper left corner of the screen.

The other overloaded version of plotData places
the stack of pages in the upper left corner of
the screen by default.

Each page has a WindowListener that will
terminate the program if the user clicks the
close button on the Frame.

The program was tested using J2SE 5.0 and WinXP.
Requires J2SE 5.0 to support generics.
************************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PlotALot01{
  //This main method is provided so that the
  // class can be run as an application to test
  // itself.
  public static void main(String[] args){
    //Instantiate two independent plotting
    // objects.  Control plotting parameters for
    // the first object.  Accept default plotting
    // parameters for the second object.
    PlotALot01 plotObjectA =
            new PlotALot01("A",158,237,36,5,4,4);
    PlotALot01 plotObjectB = new PlotALot01("B");

    //Feed the data to the first plotting object.
    for(int cnt = 0;cnt < 275;cnt++){
      //Plot some white random noise in the first
      // object using specified plotting
      // parameters. Note, that fifteen of the
      // following values are not random.  Seven
      // values are set to 0,0,25,-25,25,0,0
      // specifically to confirm the proper
      // transition from the end of one page to
      // the beginning of the next page.  Eight
      // values are set to 0,0,20,20,-20,-20,0,0
      // to confirm the proper transition from
      // one trace to the next trace on the same
      // page.  Note that these are the correct
      // values for an AWT Frame object under
      // WinXP.  However, a Frame may have
      // different inset values on other
      // operating systems, which may cause these
      // specific values to be incorrect.
      if(cnt == 147){
        plotObjectA.feedData(0);
      }else if(cnt == 148){
        plotObjectA.feedData(0);
      }else if(cnt == 149){
        plotObjectA.feedData(25);
      }else if(cnt == 150){
        plotObjectA.feedData(-25);
      }else if(cnt == 151){
        plotObjectA.feedData(25);
      }else if(cnt == 152){
        plotObjectA.feedData(0);
      }else if(cnt == 153){
        plotObjectA.feedData(0);
      }else if(cnt == 26){
        plotObjectA.feedData(0);
      }else if(cnt == 27){
        plotObjectA.feedData(0);
      }else if(cnt == 28){
        plotObjectA.feedData(20);
      }else if(cnt == 29){
        plotObjectA.feedData(20);
      }else if(cnt == 30){
        plotObjectA.feedData(-20);
      }else if(cnt == 31){
        plotObjectA.feedData(-20);
      }else if(cnt == 32){
        plotObjectA.feedData(0);
      }else if(cnt == 33){
        plotObjectA.feedData(0);
      }else{
        plotObjectA.feedData(
                       (Math.random() - 0.5)*25);
      }//end else
    }//end for loop
    //Cause the data to be plotted.
    plotObjectA.plotData(401,0);

    //Plot white random noise in the second
    // plotting object using default plotting
    // parameters.
    //Feed the data to the second plotting
    // object.
    for(int cnt = 0;cnt < 2600;cnt++){
      plotObjectB.feedData(
                       (Math.random() - 0.5)*25);
    }//end for loop
    //Cause the data to be plotted.
    plotObjectB.plotData();

  }//end main
  //---------------------------------------------//

  String title;
  int frameWidth;
  int frameHeight;
  int traceSpacing;//pixels between traces
  int sampSpacing;//pixels between samples
  int ovalWidth;//width of sample marking oval
  int ovalHeight;//height of sample marking oval

  int tracesPerPage;
  int samplesPerPage;
  int pageCounter = 0;
  int sampleCounter = 0;
  ArrayList <Page> pageLinks =
                         new ArrayList<Page>();

  //There are two overloaded versions of the
  // constructor for this class.  This
  // overloaded version accepts several incoming
  // parameters allowing the user to control
  // various aspects of the plotting format. A
  // different overloaded version accepts a title
  // string only and sets all of the plotting
  // parameters to default values.
  PlotALot01(String title,//Frame title
             int frameWidth,//in pixels
             int frameHeight,//in pixels
             int traceSpacing,//in pixels
             int sampSpace,//in pixels per sample
             int ovalWidth,//sample marker width
             int ovalHeight)//sample marker hite
  {//constructor
    //Specify sampSpace as pixels per sample.
    // Should never be less than 1.  Convert to
    // pixels between samples for purposes of
    // computation.
    this.title = title;
    this.frameWidth = frameWidth;
    this.frameHeight = frameHeight;
    this.traceSpacing = traceSpacing;
    //Convert to pixels between samples.
    this.sampSpacing = sampSpace - 1;
    this.ovalWidth = ovalWidth;
    this.ovalHeight = ovalHeight;

    //The following object is instantiated solely
    // to provide information about the width and
    // height of the canvas. This information is
    // used to compute a variety of other
    // important values.
    Page tempPage = new Page(title);
    int canvasWidth = tempPage.canvas.getWidth();
    int canvasHeight =
                     tempPage.canvas.getHeight();
    //Display information about this plotting
    // object.
    System.out.println("\nTitle: " + title);
    System.out.println(
          "Frame width: " + tempPage.getWidth());
    System.out.println(
        "Frame height: " + tempPage.getHeight());
    System.out.println(
                   "Page width: " + canvasWidth);
    System.out.println(
                 "Page height: " + canvasHeight);
    System.out.println(
               "Trace spacing: " + traceSpacing);
    System.out.println(
         "Sample spacing: " + (sampSpacing + 1));
    if(sampSpacing < 0){
      System.out.println("Terminating");
      System.exit(0);
    }//end if
    //Get rid of this temporary page.
    tempPage.dispose();
    //Now compute the remaining important values.
    tracesPerPage =
                 (canvasHeight - traceSpacing/2)/
                                   traceSpacing;
    System.out.println("Traces per page: "
                               + tracesPerPage);
    if(tracesPerPage == 0){
      System.out.println("Terminating program");
      System.exit(0);
    }//end if
    samplesPerPage = canvasWidth * tracesPerPage/
                                 (sampSpacing + 1);
    System.out.println("Samples per page: "
                                 + samplesPerPage);
    //Now instantiate the first usable Page
    // object and store its reference in the
    // list.
    pageLinks.add(new Page(title));
  }//end constructor
  //-------------------------------------------//

  PlotALot01(String title){
    //Invoke the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
  }//end overloaded constructor
  //-------------------------------------------//

  //Invoke this method for each point to be
  // plotted.
  void feedData(double val){
    if((sampleCounter) == samplesPerPage){
      //if the page is full, increment the page
      // counter, create a new empty page, and
      // reset the sample counter.
      pageCounter++;
      sampleCounter = 0;
      pageLinks.add(new Page(title));
    }//end if
    //Store the sample value in the MyCanvas
    // object to be used later to paint the
    // screen.  Then increment the sample
    // counter.  The sample value passes through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
                            val,sampleCounter);
    sampleCounter++;
  }//end feedData
  //-------------------------------------------//

  //There are two overloaded versions of the
  // plotData method.  One version allows the
  // user to specify the location on the screen
  // where the stack of plotted pages will
  // appear.  The other version places the stack
  // in the upper left corner of the screen.

  //Invoke one of the overloaded versions of
  // this method once when all of the data has
  // been fed to the plotting object in order to
  // rearrange the order of the pages with
```

```
// page 0 at the top of the stack on the
// screen.

//For this overloaded version, specify xCoor
// and yCoor to control the location of the
// stack on the screen.  Values of 0,0 will
// place the stack at the upper left corner of
// the screen.  Also see the other overloaded
// version, which places the stack at the upper
// left corner of the screen by default.
void plotData(int xCoor,int yCoor){
  Page lastPage =
           pageLinks.get(pageLinks.size() - 1);
  //Delay until last page becomes visible.
  while(!lastPage.isVisible()){
    //Loop until last page becomes visible
  }//end while loop

  Page tempPage = null;
  //Make all pages invisible
  for(int cnt = 0;cnt < (pageLinks.size());
                                        cnt++){
    tempPage = pageLinks.get(cnt);
    tempPage.setVisible(false);
  }//end for loop

  //Now make all pages visible in reverse order
  // so that page 0 will be on top of the
  // stack on the screen.
  for(int cnt = pageLinks.size() - 1;cnt >= 0;
                                        cnt--){
    tempPage = pageLinks.get(cnt);
    tempPage.setLocation(xCoor,yCoor);
    tempPage.setVisible(true);
  }//end for loop

}//end plotData(int xCoor,int yCoor)
//-------------------------------------------//

//This overloaded version of the method causes
// the stack to be located in the upper left
// corner of the screen by default
void plotData(){
  plotData(0,0);//invoke overloaded version
}//end plotData()
//-------------------------------------------//

//Inner class.  A PlotALot01 object may
// have as many Page objects as are required
// to plot all of the data values.  The
// reference to each Page object is stored
// in an ArrayList object belonging to the
// PlotALot01 object.
class Page extends Frame{
  MyCanvas canvas;
  int sampleCounter;
  Page(String title){//constructor
    canvas = new MyCanvas();
    add(canvas);
    setSize(frameWidth,frameHeight);
    setTitle(title + " Page: " + pageCounter);
    setVisible(true);

    //---------------------------------------//
    //Anonymous inner class to terminate the
    // program when the user clicks the close
    // button on the Frame.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(
                                  WindowEvent e){
          System.exit(0);//terminate program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
    //---------------------------------------//
  }//end constructor
  //=========================================//

  //This method receives a sample value of type
  // double and stores it in an array object
  // belonging to the MyCanvas object.
  void putData(double sampleValue,
               int sampleCounter){
    canvas.data[sampleCounter] = sampleValue;
    //Save the sample counter in an instance
    // variable to make it available to the
    // overridden paint method. This value is
    // needed by the paint method so it will
    // know how many samples to plot on the
    // final page which probably won't be full.
    this.sampleCounter = sampleCounter;
  }//end putData

  //=========================================//
  //Inner class
  class MyCanvas extends Canvas{
    double [] data =
                    new double[samplesPerPage];

    //Override the paint method
    public void paint(Graphics g){
      //Draw horizontal axes, one for each
      // trace.
      for(int cnt = 0;cnt < tracesPerPage;
                                        cnt++){
        g.drawLine(0,
                   (cnt+1)*traceSpacing,
                   this.getWidth(),
                   (cnt+1)*traceSpacing);
      }//end for loop
```

```
        //Plot the points if there are any to be
        // plotted.
        if(sampleCounter > 0){
          for(int cnt = 0;cnt <= sampleCounter;
                                          cnt++){
            //Compute a vertical offset to locate
            // the data on a particular trace.
            int yOffset =
                   (1 + cnt*(sampSpacing + 1)/
                   this.getWidth())*traceSpacing;
            //Draw an oval centered on the sample
            // value to mark the sample.  It is
            // best if the dimensions of the oval
            // are evenly divisable by 2 for
            // centering purposes.
            //Reverse the sign on sample value to
            // cause positive sample values to go
            // up on the screen
            g.drawOval(cnt*(sampSpacing + 1)%
                   this.getWidth() - ovalWidth/2,
              yOffset - (int)data[cnt]
                                 - ovalHeight/2,
              ovalWidth,
              ovalHeight);

            //Connect the sample values with
            // straight lines.  Do not draw a
            // line connecting the last sample in
            // one trace to the first sample in
            // the next trace.
            if(cnt*(sampSpacing + 1)%
                               this.getWidth() >=
                                sampSpacing + 1){
              g.drawLine(
                (cnt - 1)*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)data[cnt-1],
                cnt*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)data[cnt]);
            }//end if
          }//end for loop
        }//end if for sampleCounter > 0
      }//end overridden paint method
    }//end inner class MyCanvas
  }//end inner class Page
}//end class PlotALot01
//===============================================//
```

Listing 35

```
/*File PlotALot02.java
```

```
Copyright 2005, R.G.Baldwin
This program is an update to the program named
PlotALot01.  This program is designed to plot
large amounts of time-series data for two
channels on the same axes.  One set of data is
plotted using the color black.  The other set of
data is plotted using the color red.  See
PlotALot01.java for a one-channel program.

Note that by carefully adjusting the plotting
parameters, this program could also be used to
plot large quantities of spectral data in a
waterfall display.

The class provides a main method so that the
class can be run as an application to test
itself.

There are three steps involved in the use of this
class for plotting time series data:
1. Instantiate a plotting object of type
   PlotALot02 using one of two overloaded
   constructors.
2. Feed pairs of data values that are to be
   plotted to the plotting object by invoking the
   feedData method once for each pair of data
   values.  The first value in the pair will be
   plotted in black.  The second value in the
   pair will be plotted in red.
3. Invoke one of two overloaded plotData methods
   on the plotting object once all of the data
   has been fed to the object.  This causes all
   of the data to be plotted.

A using program can instantiate as many plotting
objects as are needed to plot all of the
different pairs of time series that need to be
plotted.  Each plotting object can be used to
plot as many pairs of data values as need be
plotted until the program runs out of available
memory.

The plotting object of type PlotALot02 owns one
or more Page objects that extend the Frame class.
The plotting object can own as many Page objects
as are necessary to plot all of the pairs of data
that are fed to that plotting object.

The program produces a graphic output consisting
of a stack of Page objects on the screen, with
the data plotted on a Canvas object contained by
the Page object.  The Page showing the earliest
data is on the top of the stack and the Page
showing the latest data is on the bottom of the
stack.  The Page objects on the top of the stack
must be physically moved in order to see the
```

```
Page objects on the bottom of the stack.

Each Page object contains one or more horizontal
axes on which the data is plotted.  The two time
series are superimposed on the same axes with the
data from one time series being plotted in black
and the data from the other time series being
plotted in red.

The earliest data is plotted on the axis nearest
the top of  the Page moving from left to right
across the horizontal axis.  Positive data values
are plotted above the axis and negative values
are plotted below the axis.  When the right end
of an axis is reached, the next data value is
plotted on the left end of the axis immediately
below it.  When the right end of the last axis on
the Page is reached, a new Page object is created
and the next data value is plotted at the left
end of the top axis on that new Page object.

A mentioned above, there are two overloaded
versions of the constructor for the PlotALot02
class. One overloaded version accepts several
incoming parameters allowing the user to control
various aspects of the plotting format. A second
overloaded version accepts a title string only
and sets all of the plotting parameters to
default values. You can easily modify these
default values and recompile the class if you
prefer different default values.

The parameters for the version of the constructor
that accepts plotting format information are:

String title: Title for the Frame object. This
 title is concatenated with the page number and
 the result appears in the banner at the top of
 the Frame.
int frameWidth:The Frame width in pixels.
int frameHeight: The Frame height in pixels.
int traceSpacing: Distance between trace axes in
 pixels.
int sampSpace: Number of pixels dedicated to each
 data sample in pixels per sample.  Must be 1 or
 greater.
int ovalWidth: Width of an oval that is used to
 mark each sample value on the plot.
int ovalHeight: Height of an oval that is used to
 mark each sample value on the plot.

For test purposes, the main method instantiates a
single plotting object and feeds two time series
to that plotting object.  Plotting parameters are
specified for the plotting object by using the
overloaded version of the constructor that
```

```
accepts plotting parameters.

The data that is fed to the plotting object is
white random noise. One of the time series is
the sequence of values obtained from a random
number generator.  The other time series is the
same as the first except that the sign of each
data values are reversed.

Fifteen of the data values for each time series
are not random.  Seven of the values for the
first time series are setto values of 0,0,25,-25,
25,0,0.  The corresponding seven values for the
second time series are set to the same values
with sign reversal.  This is done to confirm the
proper transition from the end of one page to the
beginning of the next page.

In addition, eight of the values for the first
time series are set to 0,0,20,20,-20,-20,0,0.
The corresponding values for the second time
series are set to the same values with sign
reversal.  This is done in order to confirm the
proper transition from one trace to the next
trace on the same page.

These specific values and the locations in the
data where they are placed provide visible
confirmation that the transitions mentioned above
are handled correctly. Note, however that these
are the correct locations for an AWT Frame object
under WinXP. A Frame may have different inset
values under other operating systems, which may
cause these specific locations to be incorrect
for that operating system.  In that case, the
values will be plotted but they won't confirm
the proper transition.

The following information about the plotting
parameters is displayed on the command line
screen when the class is used for plotting.  The
values shown below result from the execution of
the main method of the class for test purposes.

Title: A
Frame width: 158
Frame height: 237
Page width: 150
Page height: 210
Trace spacing: 36
Sample spacing: 5
Traces per page: 5
Samples per page: 150

There are two overloaded versions of the plotData
method. One version allows the user to specify
```

```
the location on the screen where the stack of
plotted pages will appear. This version requires
two parameters, which are coordinate values in
pixels.  The first parameter specifies the
horizontal coordinate of the upper left corner of
the stack of pages relative to the upper left
corner of the screen.  The second parameter
specifies the vertical coordinate of the upper
left corner of the stack of pages relative to the
upper left corner of the screen. Specifying
coordinate values of 0,0 causes the stack to be
located in the upper left corner of the screen.

The other overloaded version of plotData places
the stack of pages in the upper left corner of
the screen by default.  The main method in this
class uses the second version causing the stack
of pages to appear in the upper left corner of
the screen by default.

Each page has a WindowListener that will
terminate the program if the user clicks the
close button on the Frame.

The program was tested using J2SE 5.0 and WinXP.
Requires J2SE 5.0 to support generics.
*************************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PlotALot02{
  //This main method is provided so that the
  // class can be run as an application to test
  // itself.
  public static void main(String[] args){
    //Instantiate a plotting object using the
    // version of the constructor that allows for
    // controlling the plotting parameters.
    PlotALot02 plotObjectA =
            new PlotALot02("A",158,237,36,5,4,4);

    //Feed pairs of data values to the plotting
    // object.
    for(int cnt = 0;cnt < 275;cnt++){
      //Plot some white random noise Note that
      // fifteen of the values for each time
      // series are not random.  See the opening
      // comments for a discussion of the reasons
      // why.  Cause the values for the second
      // time series to be the negative of the
      // values for the first time series.
      double valBlack = (Math.random() - 0.5)*25;
      double valRed = -valBlack;
      //Feed pairs of values to the plotting
      // object by invoking the feedData method
      // once for each pair of data values.
      if(cnt == 147){
        plotObjectA.feedData(0,0);
      }else if(cnt == 148){
        plotObjectA.feedData(0,0);
      }else if(cnt == 149){
        plotObjectA.feedData(25,-25);
      }else if(cnt == 150){
        plotObjectA.feedData(-25,25);
      }else if(cnt == 151){
        plotObjectA.feedData(25,-25);
      }else if(cnt == 152){
        plotObjectA.feedData(0,0);
      }else if(cnt == 153){
        plotObjectA.feedData(0,0);
      }else if(cnt == 26){
        plotObjectA.feedData(0,0);
      }else if(cnt == 27){
        plotObjectA.feedData(0,0);
      }else if(cnt == 28){
        plotObjectA.feedData(20,-20);
      }else if(cnt == 29){
        plotObjectA.feedData(20,-20);
      }else if(cnt == 30){
        plotObjectA.feedData(-20,20);
      }else if(cnt == 31){
        plotObjectA.feedData(-20,20);
      }else if(cnt == 32){
        plotObjectA.feedData(0,0);
      }else if(cnt == 33){
        plotObjectA.feedData(0,0);
      }else{
        plotObjectA.feedData(valBlack,valRed);
      }//end else
    }//end for loop
    //Cause the data to be plotted in the default
    // screen location.
    plotObjectA.plotData();
  }//end main
  //---------------------------------------------//

  String title;
  int frameWidth;
  int frameHeight;
  int traceSpacing;//pixels between traces
  int sampSpacing;//pixels between samples
  int ovalWidth;//width of sample marking oval
  int ovalHeight;//height of sample marking oval

  int tracesPerPage;
  int samplesPerPage;
  int pageCounter = 0;
  int sampleCounter = 0;
  ArrayList <Page> pageLinks =
                         new ArrayList<Page>();
```

```
//There are two overloaded versions of the
// constructor for this class.  This
// overloaded version accepts several incoming
// parameters allowing the user to control
// various aspects of the plotting format. A
// different overloaded version accepts a title
// string only and sets all of the plotting
// parameters to default values.
PlotALot02(String title,//Frame title
           int frameWidth,//in pixels
           int frameHeight,//in pixels
           int traceSpacing,//in pixels
           int sampSpace,//in pixels per sample
           int ovalWidth,//sample marker width
           int ovalHeight)//sample marker hite
{//constructor
  //Specify sampSpace as pixels per sample.
  // Should never be less than 1.  Convert to
  // pixels between samples for purposes of
  // computation.
  this.title = title;
  this.frameWidth = frameWidth;
  this.frameHeight = frameHeight;
  this.traceSpacing = traceSpacing;
  //Convert to pixels between samples.
  this.sampSpacing = sampSpace - 1;
  this.ovalWidth = ovalWidth;
  this.ovalHeight = ovalHeight;

  //The following object is instantiated solely
  // to provide information about the width and
  // height of the canvas. This information is
  // used to compute a variety of other
  // important values.
  Page tempPage = new Page(title);
  int canvasWidth = tempPage.canvas.getWidth();
  int canvasHeight =
                 tempPage.canvas.getHeight();
  //Display information about this plotting
  // object.
  System.out.println("\nTitle: " + title);
  System.out.println(
        "Frame width: " + tempPage.getWidth());
  System.out.println(
      "Frame height: " + tempPage.getHeight());
  System.out.println(
               "Page width: " + canvasWidth);
  System.out.println(
             "Page height: " + canvasHeight);
  System.out.println(
           "Trace spacing: " + traceSpacing);
  System.out.println(
       "Sample spacing: " + (sampSpacing + 1));
  if(sampSpacing < 0){
    System.out.println("Terminating");
```

```
      System.exit(0);
    }//end if
    //Get rid of this temporary page.
    tempPage.dispose();
    //Now compute the remaining important values.
    tracesPerPage =
                 (canvasHeight - traceSpacing/2)/
                                   traceSpacing;
    System.out.println("Traces per page: "
                               + tracesPerPage);
    if(tracesPerPage == 0){
      System.out.println("Terminating program");
      System.exit(0);
    }//end if
    samplesPerPage = canvasWidth * tracesPerPage/
                              (sampSpacing + 1);
    System.out.println("Samples per page: "
                              + samplesPerPage);
    //Now instantiate the first usable Page
    // object and store its reference in the
    // list.
    pageLinks.add(new Page(title));
  }//end constructor
  //--------------------------------------------//

  PlotALot02(String title){
    //Invoke the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
  }//end overloaded constructor
  //--------------------------------------------//

  //Invoke this method once for each pair of data
  // values to be plotted.
  void feedData(double valBlack,double valRed){
    if((sampleCounter) == samplesPerPage){
      //if the page is full, increment the page
      // counter, create a new empty page, and
      // reset the sample counter.
      pageCounter++;
      sampleCounter = 0;
      pageLinks.add(new Page(title));
    }//end if
    //Store the sample values in the MyCanvas
    // object to be used later to paint the
    // screen.  Then increment the sample
    // counter.  The sample values pass through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
                  valBlack,valRed,sampleCounter);
    sampleCounter++;
  }//end feedData
  //--------------------------------------------//
```

```
//There are two overloaded versions of the
// plotData method.  One version allows the
// user to specify the location on the screen
// where the stack of plotted pages will
// appear.  The other version places the stack
// in the upper left corner of the screen.

//Invoke one of the overloaded versions of
// this method once when all data has been fed
// to the plotting object in order to rearrange
// the order of the pages with page 0 at the
// top of the stack on the screen.

//For this overloaded version, specify xCoor
// and yCoor to control the location of the
// stack on the screen.  Values of 0,0 will
// place the stack at the upper left corner of
// the screen.  Also see the other overloaded
// version, which places the stack at the upper
// left corner of the screen by default.
void plotData(int xCoor,int yCoor){
  Page lastPage =
           pageLinks.get(pageLinks.size() - 1);
  //Delay until last page becomes visible.
  while(!lastPage.isVisible()){
    //Loop until last page becomes visible
  }//end while loop

  Page tempPage = null;
  //Make all pages invisible
  for(int cnt = 0;cnt < (pageLinks.size());
                                        cnt++){
    tempPage = pageLinks.get(cnt);
    tempPage.setVisible(false);
  }//end for loop

  //Now make all pages visible in reverse order
  // so that page 0 will be on top of the
  // stack on the screen.
  for(int cnt = pageLinks.size() - 1;cnt >= 0;
                                        cnt--){
    tempPage = pageLinks.get(cnt);
    tempPage.setLocation(xCoor,yCoor);
    tempPage.setVisible(true);
  }//end for loop

}//end plotData(int xCoor,int yCoor)
//--------------------------------------------//

//This overloaded version of the method causes
// the stack to be located in the upper left
// corner of the screen by default
void plotData(){
  plotData(0,0);//invoke overloaded version
}//end plotData()
//--------------------------------------------//
```

```
//Inner class.  A PlotALot02 object may
// have as many Page objects as are required
// to plot all of the data values.  The
// reference to each Page object is stored
// in an ArrayList object belonging to the
// PlotALot02 object.
class Page extends Frame{
  MyCanvas canvas;
  int sampleCounter;

  Page(String title){//constructor
    canvas = new MyCanvas();
    add(canvas);
    setSize(frameWidth,frameHeight);
    setTitle(title + " Page: " + pageCounter);
    setVisible(true);

    //---------------------------------------//
    //Anonymous inner class to terminate the
    // program when the user clicks the close
    // button on the Frame.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(
                                 WindowEvent e){
          System.exit(0);//terminate program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
    //---------------------------------------//
  }//end constructor
  //=========================================//

  //This method receives a pair of sample
  // values of type double and stores each of
  // them in a separate array object belonging
  // to the MyCanvas object.
  void putData(double valBlack,double valRed,
                             int sampleCounter){
    canvas.blackData[sampleCounter] = valBlack;
    canvas.redData[sampleCounter] = valRed;
    //Save the sample counter in an instance
    // variable to make it available to the
    // overridden paint method. This value is
    // needed by the paint method so it will
    // know how many samples to plot on the
    // final page which probably won't be full.
    this.sampleCounter = sampleCounter;
  }//end putData

  //=========================================//
  //Inner class
  class MyCanvas extends Canvas{
    double [] blackData =
                      new double[samplesPerPage];
    double [] redData =
                    new double[samplesPerPage];

    //Override the paint method
    public void paint(Graphics g){
      //Draw horizontal axes, one for each
      // trace.
      for(int cnt = 0;cnt < tracesPerPage;
                                        cnt++){
        g.drawLine(0,
                   (cnt+1)*traceSpacing,
                   this.getWidth(),
                   (cnt+1)*traceSpacing);
      }//end for loop

      //Plot the points if there are any to be
      // plotted.
      if(sampleCounter > 0){
        for(int cnt = 0;cnt <= sampleCounter;
                                        cnt++){
          //Compute a vertical offset to locate
          // the data on a particular trace.
          int yOffset =
                 (1 + cnt*(sampSpacing + 1)/
                 this.getWidth())*traceSpacing;
          //Begin by plotting the values from
          // the blackData array object.
          //Draw an oval centered on the sample
          // value to mark the sample in the
          // plot. It is best if the dimensions
          // of the oval are evenly divisable
          // by 2 for  centering purposes.
          //Reverse the sign of the sample
          // value to cause positive sample
          // values to be plotted above the
          // axis.
          g.setColor(Color.BLACK);
          g.drawOval(cnt*(sampSpacing + 1)%
                 this.getWidth() - ovalWidth/2,
            yOffset - (int)blackData[cnt]
                                - ovalHeight/2,
            ovalWidth,
            ovalHeight);

          //Connect the sample values with
          // straight lines.  Do not draw a
          // line connecting the last sample in
          // one trace to the first sample in
          // the next trace.
          if(cnt*(sampSpacing + 1)%
                             this.getWidth() >=
                              sampSpacing + 1){
            g.drawLine(
              (cnt - 1)*(sampSpacing + 1)%
                               this.getWidth(),
              yOffset - (int)blackData[cnt-1],
              cnt*(sampSpacing + 1)%
                               this.getWidth(),
              yOffset - (int)blackData[cnt]);
          }//end if

          //Now plot the data stored in the
          // redData array object.
          g.setColor(Color.RED);
          //Draw the ovals as described above.
          g.drawOval(cnt*(sampSpacing + 1)%
                 this.getWidth() - ovalWidth/2,
            yOffset - (int)redData[cnt]
                                - ovalHeight/2,
            ovalWidth,
            ovalHeight);

          //Connect the sample values with
          // straight lines as described above.
          if(cnt*(sampSpacing + 1)%
                               this.getWidth() >=
                                sampSpacing + 1){
            g.drawLine(
              (cnt - 1)*(sampSpacing + 1)%
                               this.getWidth(),
              yOffset - (int)redData[cnt-1],
              cnt*(sampSpacing + 1)%
                               this.getWidth(),
              yOffset - (int)redData[cnt]);

          }//end if
        }//end for loop
      }//end if for sampleCounter > 0
    }//end overridden paint method
  }//end inner class MyCanvas
}//end inner class Page
}//end class PlotALot02
//===============================================//
```

Listing 36

```
/*File PlotALot03.java
Copyright 2005, R.G.Baldwin
This program is an update to the program named
PlotALot02.  This program is designed to plot
large amounts of time-series data for two
channels on alternating horizontal axes.  One set
of data is plotted using the color black.  The
other set of data is plotted using the color red.

See PlotALot02 for a class that plots two
channels of data in black and red superimposed on
the same axes.  See PlotALot01.java for a
```

```
one-channel program.

Note that by carefully adjusting the plotting
parameters, this program could also be used to
plot large quantities of spectral data in a
waterfall display.

The class provides a main method so that the
class can be run as an application to test
itself.

There are three steps involved in the use of this
class for plotting time series data:
1. Instantiate a plotting object of type
   PlotALot03 using one of two overloaded
   constructors.
2. Feed pairs of data values that are to be
   plotted to the plotting object by invoking the
   feedData method once for each pair of data
   values.  The first value in the pair will be
   plotted in black on one axis.  The second
   value in the pair will be plotted in red on an
   axis below that axis.
3. Invoke one of two overloaded plotData methods
   on the plotting object once all of the data
   has been fed to the object.  This causes all
   of the data to be plotted.

A using program can instantiate as many plotting
objects as are needed to plot all of the
different pairs of time series that need to be
plotted.  Each plotting object can be used to
plot as many pairs of data values as need be
plotted until the program runs out of available
memory.

The plotting object of type PlotALot03 owns one
or more Page objects that extend the Frame class.
The plotting object can own as many Page objects
as are necessary to plot all of the pairs of data
that are fed to that plotting object.

The program produces a graphic output consisting
of a stack of Page objects on the screen, with
the data plotted on a Canvas object contained by
the Page object.  The Page showing the earliest
data is on the top of the stack and the Page
showing the latest data is on the bottom of the
stack.  The Page objects on the top of the stack
must be physically moved in order to see the
Page objects on the bottom of the stack.

Each Page object contains two or more horizontal
axes on which the data is plotted.  The program
will terminate if the number of axes on the page
is an odd number.
```

```
The two time series are plotted on alternating
axes with the data from one time series being
plotted in black on one axis and the data from
the other time series being plotted in red on
the axis below that axis.

The earliest data is plotted on the pair of axes
nearest the top of the Page moving from left to
right across the page.  Positive data values
are plotted above the axis and negative values
are plotted below the axis.  When the right end
of an axis is reached, the next data value is
plotted on the left end of the second axis
below it skipping one axis in the process.  When
the right end of the last pair of axes on the
Page is reached, a new Page object is created and
the next pair of data values are plotted at the
left end of the top pair of axes on that new Page
object.

A mentioned above, there are two overloaded
versions of the constructor for the PlotALot03
class. One overloaded version accepts several
incoming parameters allowing the user to control
various aspects of the plotting format. A second
overloaded version accepts a title string only
and sets all of the plotting parameters to
default values. You can easily modify these
default values and recompile the class if you
prefer different default values.

The parameters for the version of the constructor
that accepts plotting format information are:

String title: Title for the Frame object. This
 title is concatenated with the page number and
 the result appears in the banner at the top of
 the Frame.
int frameWidth:The Frame width in pixels.
int frameHeight: The Frame height in pixels.
int traceSpacing: Distance between trace axes in
 pixels.
int sampSpace: Number of pixels dedicated to each
 data sample in pixels per sample.  Must be 1 or
 greater.
int ovalWidth: Width of an oval that is used to
 mark each sample value on the plot.
int ovalHeight: Height of an oval that is used to
 mark each sample value on the plot.

For test purposes, the main method instantiates a
single plotting object and feeds two time series
to that plotting object.  Plotting parameters are
specified for the plotting object by using the
overloaded version of the constructor that
```

```
accepts plotting parameters.

The data that is fed to the plotting object is
white random noise. One of the time series is
the sequence of values obtained from a random
number generator.  The other time series is the
same as the first.  Thus, the pairs of black and
red time series that are plotted should have the
same shape making it easy to confirm that the
process of plotting the two time series is
behaving the same in both cases.

Fifteen of the data values for each time series
are not random.  Seven of the values for each of
the time series are set to values of 0,0,25,-25,
25,0,0.  This is done to confirm the proper
transition from the end of one page to the
beginning of the next page.

In addition, eight of the values for each time
series are set to 0,0,20,20,-20,-20,0,0.  This
is done in order to confirm the proper transition
from one trace to the next trace on the same
page.

These specific values and the locations in the
data where they are placed provide visible
confirmation that the transitions mentioned above
are handled correctly. Note, however that these
are the correct locations for an AWT Frame object
under WinXP. A Frame may have different inset
values under other operating systems, which may
cause these specific locations to be incorrect
for that operating system.  In that case, the
values will be plotted but they won't confirm
the proper transition.

The following information about the plotting
parameters is displayed on the command line
screen when the class is used for plotting.  The
values shown below result from the execution of
the main method of the class for test purposes.

Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 90

There are two overloaded versions of the plotData
method. One version allows the user to specify
the location on the screen where the stack of
```

```
plotted pages will appear. This version requires
two parameters, which are coordinate values in
pixels.  The first parameter specifies the
horizontal coordinate of the upper left corner of
the stack of pages relative to the upper left
corner of the screen.  The second parameter
specifies the vertical coordinate of the upper
left corner of the stack of pages relative to the
upper left corner of the screen. Specifying
coordinate values of 0,0 causes the stack to be
located in the upper left corner of the screen.

The other overloaded version of plotData places
the stack of pages in the upper left corner of
the screen by default.  The main method in this
class uses the second version causing the stack
of pages to appear in the upper left corner of
the screen by default.

Each page has a WindowListener that will
terminate the program if the user clicks the
close button on the Frame.

The program was tested using J2SE 5.0 and WinXP.
Requires J2SE 5.0 to support generics.
*************************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PlotALot03{
  //This main method is provided so that the
  // class can be run as an application to test
  // itself.
  public static void main(String[] args){
    //Instantiate a plotting object using the
    // version of the constructor that allows for
    // controlling the plotting parameters.
    PlotALot03 plotObjectA =
            new PlotALot03("A",158,270,36,5,4,4);

    //Feed pairs of data values to the plotting
    // object.
    for(int cnt = 0;cnt < 175;cnt++){
      //Plot some white random noise Note that
      // fifteen of the values for each time
      // series are not random.  See the opening
      // comments for a discussion of the reasons
      // why.  Cause the values for the second
      // time series to be the same as the
      // values for the first time series.
      double valBlack = (Math.random() - 0.5)*25;
      double valRed = valBlack;
      //Feed pairs of values to the plotting
      // object by invoking the feedData method
      // once for each pair of data values.
      if(cnt == 87){
        plotObjectA.feedData(0,0);
      }else if(cnt == 88){
        plotObjectA.feedData(0,0);
      }else if(cnt == 89){
        plotObjectA.feedData(25,25);
      }else if(cnt == 90){
        plotObjectA.feedData(-25,-25);
      }else if(cnt == 91){
        plotObjectA.feedData(25,25);
      }else if(cnt == 92){
        plotObjectA.feedData(0,0);
      }else if(cnt == 93){
        plotObjectA.feedData(0,0);
      }else if(cnt == 26){
        plotObjectA.feedData(0,0);
      }else if(cnt == 27){
        plotObjectA.feedData(0,0);
      }else if(cnt == 28){
        plotObjectA.feedData(20,20);
      }else if(cnt == 29){
        plotObjectA.feedData(20,20);
      }else if(cnt == 30){
        plotObjectA.feedData(-20,-20);
      }else if(cnt == 31){
        plotObjectA.feedData(-20,-20);
      }else if(cnt == 32){
        plotObjectA.feedData(0,0);
      }else if(cnt == 33){
        plotObjectA.feedData(0,0);
      }else{
        plotObjectA.feedData(valBlack,valRed);
      }//end else
    }//end for loop
    //Cause the data to be plotted in the default
    // screen location.
    plotObjectA.plotData();
  }//end main
  //----------------------------------------------//

  String title;
  int frameWidth;
  int frameHeight;
  int traceSpacing;//pixels between traces
  int sampSpacing;//pixels between samples
  int ovalWidth;//width of sample marking oval
  int ovalHeight;//height of sample marking oval

  int tracesPerPage;
  int samplesPerPage;
  int pageCounter = 0;
  int sampleCounter = 0;
  ArrayList <Page> pageLinks =
                         new ArrayList<Page>();
  //There are two overloaded versions of the
  // constructor for this class.  This
  // overloaded version accepts several incoming
  // parameters allowing the user to control
  // various aspects of the plotting format. A
  // different overloaded version accepts a title
  // string only and sets all of the plotting
  // parameters to default values.
  PlotALot03(String title,//Frame title
             int frameWidth,//in pixels
             int frameHeight,//in pixels
             int traceSpacing,//in pixels
             int sampSpace,//in pixels per sample
             int ovalWidth,//sample marker width
             int ovalHeight)//sample marker hite
  {//constructor
    //Specify sampSpace as pixels per sample.
    // Should never be less than 1.  Convert to
    // pixels between samples for purposes of
    // computation.
    this.title = title;
    this.frameWidth = frameWidth;
    this.frameHeight = frameHeight;
    this.traceSpacing = traceSpacing;
    //Convert to pixels between samples.
    this.sampSpacing = sampSpace - 1;
    this.ovalWidth = ovalWidth;
    this.ovalHeight = ovalHeight;

    //The following object is instantiated solely
    // to provide information about the width and
    // height of the canvas. This information is
    // used to compute a variety of other
    // important values.
    Page tempPage = new Page(title);
    int canvasWidth = tempPage.canvas.getWidth();
    int canvasHeight =
                     tempPage.canvas.getHeight();
    //Display information about this plotting
    // object.
    System.out.println("\nTitle: " + title);
    System.out.println(
          "Frame width: " + tempPage.getWidth());
    System.out.println(
        "Frame height: " + tempPage.getHeight());
    System.out.println(
                   "Page width: " + canvasWidth);
    System.out.println(
                 "Page height: " + canvasHeight);
    System.out.println(
               "Trace spacing: " + traceSpacing);
    System.out.println(
         "Sample spacing: " + (sampSpacing + 1));
    if(sampSpacing < 0){
      System.out.println("Terminating");
      System.exit(0);
    }//end if
    //Get rid of this temporary page.
    tempPage.dispose();
    //Now compute the remaining important values.
    tracesPerPage =
                 (canvasHeight - traceSpacing/2)/
                                    traceSpacing;
    System.out.println("Traces per page: "
                                + tracesPerPage);
    if((tracesPerPage == 0) ||
                      (tracesPerPage%2 != 0) ){
      System.out.println("Terminating program");
      System.exit(0);
    }//end if
    samplesPerPage = canvasWidth * tracesPerPage/
                             (sampSpacing + 1)/2;
    System.out.println("Samples per page: "
                               + samplesPerPage);
    //Now instantiate the first usable Page
    // object and store its reference in the
    // list.
    pageLinks.add(new Page(title));
  }//end constructor
  //---------------------------------------------//

  PlotALot03(String title){
    //Invoke the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
  }//end overloaded constructor
  //---------------------------------------------//

  //Invoke this method once for each pair of data
  // values to be plotted.
  void feedData(double valBlack,double valRed){
    if((sampleCounter) == samplesPerPage){
      //if the page is full, increment the page
      // counter, create a new empty page, and
      // reset the sample counter.
      pageCounter++;
      sampleCounter = 0;
      pageLinks.add(new Page(title));
    }//end if
    //Store the sample values in the MyCanvas
    // object to be used later to paint the
    // screen.  Then increment the sample
    // counter.  The sample values pass through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
                 valBlack,valRed,sampleCounter);
    sampleCounter++;
  }//end feedData
  //---------------------------------------------//
```

```
//There are two overloaded versions of the
// plotData method.  One version allows the
// user to specify the location on the screen
// where the stack of plotted pages will
// appear.  The other version places the stack
// in the upper left corner of the screen.

//Invoke one of the overloaded versions of
// this method once when all data has been fed
// to the plotting object in order to rearrange
// the order of the pages with page 0 at the
// top of the stack on the screen.

//For this overloaded version, specify xCoor
// and yCoor to control the location of the
// stack on the screen.  Values of 0,0 will
// place the stack at the upper left corner of
// the screen.  Also see the other overloaded
// version, which places the stack at the upper
// left corner of the screen by default.
void plotData(int xCoor,int yCoor){
  Page lastPage =
           pageLinks.get(pageLinks.size() - 1);
  //Delay until last page becomes visible.
  while(!lastPage.isVisible()){
    //Loop until last page becomes visible
  }//end while loop

  Page tempPage = null;
  //Make all pages invisible
  for(int cnt = 0;cnt < (pageLinks.size());
                                        cnt++){
    tempPage = pageLinks.get(cnt);
    tempPage.setVisible(false);
  }//end for loop

  //Now make all pages visible in reverse order
  // so that page 0 will be on top of the
  // stack on the screen.
  for(int cnt = pageLinks.size() - 1;cnt >= 0;
                                        cnt--){
    tempPage = pageLinks.get(cnt);
    tempPage.setLocation(xCoor,yCoor);
    tempPage.setVisible(true);
  }//end for loop

}//end plotData(int xCoor,int yCoor)
//--------------------------------------------//

//This overloaded version of the method causes
// the stack to be located in the upper left
// corner of the screen by default
void plotData(){
  plotData(0,0);//invoke overloaded version
}//end plotData()
//--------------------------------------------//
```

```
//Inner class.  A PlotALot03 object may
// have as many Page objects as are required
// to plot all of the data values.  The
// reference to each Page object is stored
// in an ArrayList object belonging to the
// PlotALot03 object.
class Page extends Frame{
  MyCanvas canvas;
  int sampleCounter;

  Page(String title){//constructor
    canvas = new MyCanvas();
    add(canvas);
    setSize(frameWidth,frameHeight);
    setTitle(title + " Page: " + pageCounter);
    setVisible(true);

    //---------------------------------------//
    //Anonymous inner class to terminate the
    // program when the user clicks the close
    // button on the Frame.
    addWindowListener(
      new WindowAdapter(){
        public void windowClosing(
                                WindowEvent e){
          System.exit(0);//terminate program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
    //---------------------------------------//
  }//end constructor
  //=========================================//

  //This method receives a pair of sample
  // values of type double and stores each of
  // them in a separate array object belonging
  // to the MyCanvas object.
  void putData(double valBlack,double valRed,
                            int sampleCounter){
    canvas.blackData[sampleCounter] = valBlack;
    canvas.redData[sampleCounter] = valRed;
    //Save the sample counter in an instance
    // variable to make it available to the
    // overridden paint method. This value is
    // needed by the paint method so it will
    // know how many samples to plot on the
    // final page which probably won't be full.
    this.sampleCounter = sampleCounter;
  }//end putData

  //=========================================//
  //Inner class
  class MyCanvas extends Canvas{
    double [] blackData =
                     new double[samplesPerPage];
    double [] redData =
                    new double[samplesPerPage];

    //Override the paint method
    public void paint(Graphics g){
      //Draw horizontal axes, one for each
      // trace.
      for(int cnt = 0;cnt < tracesPerPage;
                                        cnt++){
        g.drawLine(0,
                   (cnt+1)*traceSpacing,
                   this.getWidth(),
                   (cnt+1)*traceSpacing);
      }//end for loop

      //Plot the points if there are any to be
      // plotted.
      if(sampleCounter > 0){
        for(int cnt = 0;cnt <= sampleCounter;
                                        cnt++){

          //Begin by plotting the values from
          // the blackData array object.
          g.setColor(Color.BLACK);

          //Compute a vertical offset to locate
          // the black data on the odd numbered
          // axes on the page.
          int yOffset =
             ((1 + cnt*(sampSpacing + 1)/
              this.getWidth())*2*traceSpacing)
                                - traceSpacing;

          //Draw an oval centered on the sample
          // value to mark the sample in the
          // plot. It is best if the dimensions
          // of the oval are evenly divisable
          // by 2 for  centering purposes.
          //Reverse the sign of the sample
          // value to cause positive sample
          // values to be plotted above the
          // axis.

          g.drawOval(cnt*(sampSpacing + 1)%
                 this.getWidth() - ovalWidth/2,
            yOffset - (int)blackData[cnt]
                                - ovalHeight/2,
            ovalWidth,
            ovalHeight);

          //Connect the sample values with
          // straight lines.  Do not draw a
          // line connecting the last sample in
          // one trace to the first sample in
          // the next trace.
          if(cnt*(sampSpacing + 1)%
                             this.getWidth() >=
                              sampSpacing + 1){
            g.drawLine(
              (cnt - 1)*(sampSpacing + 1)%
                                this.getWidth(),
              yOffset - (int)blackData[cnt-1],
              cnt*(sampSpacing + 1)%
                                this.getWidth(),
              yOffset - (int)blackData[cnt]);
          }//end if

          //Now plot the data stored in the
          // redData array object.
          g.setColor(Color.RED);
          //Compute a vertical offset to locate
          // the red data on the even numbered
          // axes on the page.
          yOffset = (1 + cnt*(sampSpacing + 1)/
               this.getWidth())*2*traceSpacing;

          //Draw the ovals as described above.
          g.drawOval(cnt*(sampSpacing + 1)%
                 this.getWidth() - ovalWidth/2,
            yOffset - (int)redData[cnt]
                            - ovalHeight/2,
            ovalWidth,
            ovalHeight);

          //Connect the sample values with
          // straight lines as described above.
          if(cnt*(sampSpacing + 1)%
                             this.getWidth() >=
                              sampSpacing + 1){
            g.drawLine(
              (cnt - 1)*(sampSpacing + 1)%
                                this.getWidth(),
              yOffset - (int)redData[cnt-1],
              cnt*(sampSpacing + 1)%
                                this.getWidth(),
              yOffset - (int)redData[cnt]);

          }//end if
        }//end for loop
      }//end if for sampleCounter > 0
    }//end overridden paint method
  }//end inner class MyCanvas
}//end inner class Page
}//end class PlotALot03
//=============================================//
```

Listing 37

```
/*File PlotALot04.java
```

```
Copyright 2005, R.G.Baldwin
This program is an update to the program named
PlotALot03.  This program is designed to plot
large amounts of time-series data for three
channels on separate horizontal axes.  One set
of data is plotted using the color black.  The
second set of data is plotted using the color
red.  The third set of data is plotted using the
color blue.

See PlotALot03 for a class that plots two
channels of data in black and red on alternating
axes.

See PlotALot02 for a class that plots two
channels of data in black and red superimposed on
the same axes.

See PlotALot01.java for a one-channel program.

The class provides a main method so that the
class can be run as an application to test
itself.

There are three steps involved in the use of this
class for plotting time series data:
1. Instantiate a plotting object of type
   PlotALot04 using one of two overloaded
   constructors.
2. Feed triplets of data values that are to be
   plotted to the plotting object by invoking the
   feedData method once for each triplet of data
   values.  The first value in the triplet will
   be plotted in black on one axis.  The second
   value in the triplet will be plotted in red on
   an axis below that axis.  The third value in
   the triplet will be plotted in blue on an axis
   below that one.
3. Invoke one of two overloaded plotData methods
   on the plotting object once all of the data
   has been fed to the object.  This causes all
   of the data to be plotted.

A using program can instantiate as many plotting
objects as are needed to plot all of the
different triplets of data that need to be
plotted.  Each plotting object can be used to
plot as many triplts of data values as need be
plotted until the program runs out of available
memory.

The plotting object of type PlotALot04 owns one
or more Page objects that extend the Frame class.
The plotting object can own as many Page objects
as are necessary to plot all of the triplets of
data that are fed to that plotting object.
```

```
The program produces a graphic output consisting
of a stack of Page objects on the screen, with
the data plotted on a Canvas object contained by
the Page object.  The Page showing the earliest
data is on the top of the stack and the Page
showing the latest data is on the bottom of the
stack.  The Page objects on the top of the stack
must be physically moved in order to see the
Page objects on the bottom of the stack.

Each Page object contains three or more
horizontal axes on which the data is plotted. The
program will terminate if the number of axes on
the page is not evenly divisable by 3.

The three time series are plotted on separate
axes with the data from one time series being
plotted in black on one axis, the data from
the second time series being plotted in red on
the axis below that axis, and the data from the
third time series being plotted in blue on the
axis below that axis.

The earliest data is plotted on the three axes
nearest the top of the Page moving from left to
right across the page.  Positive data values
are plotted above the axis and negative values
are plotted below the axis.  When the right end
of an axis is reached, the next data value is
plotted on the left end of the third axis
below it skipping two axes in the process.  When
the right end of the last triplet of axes on the
Page is reached, a new Page object is created and
the next triplet of data values are plotted at
the left end of the top three axes on that new
Page object.

A mentioned above, there are two overloaded
versions of the constructor for the PlotALot04
class. One overloaded version accepts several
incoming parameters allowing the user to control
various aspects of the plotting format. A second
overloaded version accepts a title string only
and sets all of the plotting parameters to
default values. You can easily modify these
default values and recompile the class if you
prefer different default values.

The parameters for the version of the constructor
that accepts plotting format information are:

String title: Title for the Frame object. This
 title is concatenated with the page number and
 the result appears in the banner at the top of
 the Frame.
```

```
int frameWidth:The Frame width in pixels.
int frameHeight: The Frame height in pixels.
int traceSpacing: Distance between trace axes in
 pixels.
int sampSpace: Number of pixels dedicated to each
 data sample in pixels per sample.  Must be 1 or
 greater.
int ovalWidth: Width of an oval that is used to
 mark each sample value on the plot.
int ovalHeight: Height of an oval that is used to
 mark each sample value on the plot.

For test purposes, the main method instantiates a
single plotting object and feeds three time
series to that plotting object.  Plotting
parameters are specified for the plotting object
by using the overloaded version of the
constructor that accepts plotting parameters.

The data that is fed to the plotting object is
white random noise. One of the time series is
the sequence of values obtained from a random
number generator.  The other two time series are
the same as the first.  Thus, the triplets of
black, red, and blue time series that are plotted
should have the same shape making it easy to
confirm that the process of plotting the three
time series is behaving the same in all three
cases.

Fifteen of the data values for each time series
are not random.  Seven of the values for each of
the time series are set to values of 0,0,25,-25,
25,0,0.  This is done to confirm the proper
transition from the end of one page to the
beginning of the next page.

In addition, eight of the values for each time
series are set to 0,0,20,20,-20,-20,0,0.  This
is done in order to confirm the proper transition
from one trace to the next trace on the same
page.

These specific values and the locations in the
data where they are placed provide visible
confirmation that the transitions mentioned above
are handled correctly. Note, however that these
are the correct locations for an AWT Frame object
under WinXP. A Frame may have different inset
values under other operating systems, which may
cause these specific locations to be incorrect
for that operating system.  In that case, the
values will be plotted but they won't confirm
the proper transition.

The following information about the plotting
```

```
parameters is displayed on the command line
screen when the class is used for plotting.  The
values shown below result from the execution of
the main method of the class for test purposes.

Title: A
Frame width: 158
Frame height: 270
Page width: 150
Page height: 243
Trace spacing: 36
Sample spacing: 5
Traces per page: 6
Samples per page: 60

There are two overloaded versions of the plotData
method. One version allows the user to specify
the location on the screen where the stack of
plotted pages will appear. This version requires
two parameters, which are coordinate values in
pixels.  The first parameter specifies the
horizontal coordinate of the upper left corner of
the stack of pages relative to the upper left
corner of the screen.  The second parameter
specifies the vertical coordinate of the upper
left corner of the stack of pages relative to the
upper left corner of the screen. Specifying
coordinate values of 0,0 causes the stack to be
located in the upper left corner of the screen.

The other overloaded version of plotData places
the stack of pages in the upper left corner of
the screen by default.  The main method in this
class uses the second version causing the stack
of pages to appear in the upper left corner of
the screen by default.

Each page has a WindowListener that will
terminate the program if the user clicks the
close button on the Frame.

The program was tested using J2SE 5.0 and WinXP.
Requires J2SE 5.0 to support generics.
*************************************************/

import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class PlotALot04{
  //This main method is provided so that the
  // class can be run as an application to test
  // itself.
  public static void main(String[] args){
    //Instantiate a plotting object using the
    // version of the constructor that allows for
    // controlling the plotting parameters.
    PlotALot04 plotObjectA =
            new PlotALot04("A",158,270,36,5,4,4);

    //Feed triplets of data values to the
    // plotting object.
    for(int cnt = 0;cnt < 115;cnt++){
      //Plot some white random noise. Note that
      // fifteen of the values for each time
      // series are not random.  See the opening
      // comments for a discussion of the reasons
      // why.
      double valBlack = (Math.random() - 0.5)*25;
      double valRed = valBlack;
      double valBlue = valBlack;
      //Feed triplets of values to the plotting
      // object by invoking the feedData method
      // once for each triplet of data values.
      if(cnt == 57){
        plotObjectA.feedData(0,0,0);
      }else if(cnt == 58){
        plotObjectA.feedData(0,0,0);
      }else if(cnt == 59){
        plotObjectA.feedData(25,25,25);
      }else if(cnt == 60){
        plotObjectA.feedData(-25,-25,-25);
      }else if(cnt == 61){
        plotObjectA.feedData(25,25,25);
      }else if(cnt == 62){
        plotObjectA.feedData(0,0,0);
      }else if(cnt == 63){
        plotObjectA.feedData(0,0,0);
      }else if(cnt == 26){
        plotObjectA.feedData(0,0,0);
      }else if(cnt == 27){
        plotObjectA.feedData(0,0,0);
      }else if(cnt == 28){
        plotObjectA.feedData(20,20,20);
      }else if(cnt == 29){
        plotObjectA.feedData(20,20,20);
      }else if(cnt == 30){
        plotObjectA.feedData(-20,-20,-20);
      }else if(cnt == 31){
        plotObjectA.feedData(-20,-20,-20);
      }else if(cnt == 32){
        plotObjectA.feedData(0,0,0);
      }else if(cnt == 33){
        plotObjectA.feedData(0,0,0);
      }else{
        plotObjectA.feedData(valBlack,
                             valRed,
                             valBlue);
      }//end else
    }//end for loop
    //Cause the data to be plotted in the default
    // screen location.
    plotObjectA.plotData();
  }//end main
  //--------------------------------------------//

  String title;
  int frameWidth;
  int frameHeight;
  int traceSpacing;//pixels between traces
  int sampSpacing;//pixels between samples
  int ovalWidth;//width of sample marking oval
  int ovalHeight;//height of sample marking oval

  int tracesPerPage;
  int samplesPerPage;
  int pageCounter = 0;
  int sampleCounter = 0;
  ArrayList <Page> pageLinks =
                         new ArrayList<Page>();

  //There are two overloaded versions of the
  // constructor for this class.  This
  // overloaded version accepts several incoming
  // parameters allowing the user to control
  // various aspects of the plotting format. A
  // different overloaded version accepts a title
  // string only and sets all of the plotting
  // parameters to default values.
  PlotALot04(String title,//Frame title
             int frameWidth,//in pixels
             int frameHeight,//in pixels
             int traceSpacing,//in pixels
             int sampSpace,//in pixels per sample
             int ovalWidth,//sample marker width
             int ovalHeight)//sample marker hite
  {//constructor
    //Specify sampSpace as pixels per sample.
    // Should never be less than 1.  Convert to
    // pixels between samples for purposes of
    // computation.
    this.title = title;
    this.frameWidth = frameWidth;
    this.frameHeight = frameHeight;
    this.traceSpacing = traceSpacing;
    //Convert to pixels between samples.
    this.sampSpacing = sampSpace - 1;
    this.ovalWidth = ovalWidth;
    this.ovalHeight = ovalHeight;

    //The following object is instantiated solely
    // to provide information about the width and
    // height of the canvas. This information is
    // used to compute a variety of other
    // important values.
    Page tempPage = new Page(title);
    int canvasWidth = tempPage.canvas.getWidth();
    int canvasHeight =
                   tempPage.canvas.getHeight();
    //Display information about this plotting
    // object.
    System.out.println("\nTitle: " + title);
    System.out.println(
          "Frame width: " + tempPage.getWidth());
    System.out.println(
        "Frame height: " + tempPage.getHeight());
    System.out.println(
                   "Page width: " + canvasWidth);
    System.out.println(
                 "Page height: " + canvasHeight);
    System.out.println(
               "Trace spacing: " + traceSpacing);
    System.out.println(
         "Sample spacing: " + (sampSpacing + 1));
    if(sampSpacing < 0){
      System.out.println("Terminating");
      System.exit(0);
    }//end if
    //Get rid of this temporary page.
    tempPage.dispose();
    //Now compute the remaining important values.
    tracesPerPage =
                 (canvasHeight - traceSpacing/2)/
                                    traceSpacing;
    System.out.println("Traces per page: "
                                + tracesPerPage);
    if((tracesPerPage == 0) ||
                        (tracesPerPage%3 != 0) ){
      System.out.println("Terminating program");
      System.exit(0);
    }//end if
    samplesPerPage = canvasWidth * tracesPerPage/
                             (sampSpacing + 1)/3;
    System.out.println("Samples per page: "
                               + samplesPerPage);
    //Now instantiate the first usable Page
    // object and store its reference in the
    // list.
    pageLinks.add(new Page(title));
  }//end constructor
  //--------------------------------------------//

  PlotALot04(String title){
    //Invoke the other overloaded constructor
    // passing default values for all but the
    // title.
    this(title,400,410,50,2,2,2);
  }//end overloaded constructor
  //--------------------------------------------//

  //Invoke this method once for each triplet of
  // data values to be plotted.
  void feedData(double valBlack,
                double valRed,
                double valBlue){
    if((sampleCounter) == samplesPerPage){
      //if the page is full, increment the page
      // counter, create a new empty page, and
      // reset the sample counter.
      pageCounter++;
      sampleCounter = 0;
      pageLinks.add(new Page(title));
    }//end if
    //Store the sample values in the MyCanvas
    // object to be used later to paint the
    // screen.  Then increment the sample
    // counter.  The sample values pass through
    // the page object into the current MyCanvas
    // object.
    pageLinks.get(pageCounter).putData(
                              valBlack,
                              valRed,
                              valBlue,
                              sampleCounter);
    sampleCounter++;
  }//end feedData
  //---------------------------------------------//

  //There are two overloaded versions of the
  // plotData method.  One version allows the
  // user to specify the location on the screen
  // where the stack of plotted pages will
  // appear.  The other version places the stack
  // in the upper left corner of the screen.

  //Invoke one of the overloaded versions of
  // this method once when all data has been fed
  // to the plotting object in order to rearrange
  // the order of the pages with page 0 at the
  // top of the stack on the screen.

  //For this overloaded version, specify xCoor
  // and yCoor to control the location of the
  // stack on the screen.  Values of 0,0 will
  // place the stack at the upper left corner of
  // the screen.  Also see the other overloaded
  // version, which places the stack at the upper
  // left corner of the screen by default.
  void plotData(int xCoor,int yCoor){
    Page lastPage =
             pageLinks.get(pageLinks.size() - 1);
    //Delay until last page becomes visible.
    while(!lastPage.isVisible()){
      //Loop until last page becomes visible
    }//end while loop

    Page tempPage = null;
    //Make all pages invisible
    for(int cnt = 0;cnt < (pageLinks.size());
                                         cnt++){
```

```
      tempPage = pageLinks.get(cnt);
      tempPage.setVisible(false);
    }//end for loop

    //Now make all pages visible in reverse order
    // so that page 0 will be on top of the
    // stack on the screen.
    for(int cnt = pageLinks.size() - 1;cnt >= 0;
                                            cnt--){
      tempPage = pageLinks.get(cnt);
      tempPage.setLocation(xCoor,yCoor);
      tempPage.setVisible(true);
    }//end for loop

  }//end plotData(int xCoor,int yCoor)
  //---------------------------------------------//

  //This overloaded version of the method causes
  // the stack to be located in the upper left
  // corner of the screen by default
  void plotData(){
    plotData(0,0);//invoke overloaded version
  }//end plotData()
  //---------------------------------------------//

  //Inner class.  A PlotALot04 object may
  // have as many Page objects as are required
  // to plot all of the data values.  The
  // reference to each Page object is stored
  // in an ArrayList object belonging to the
  // PlotALot04 object.
  class Page extends Frame{
    MyCanvas canvas;
    int sampleCounter;

    Page(String title){//constructor
      canvas = new MyCanvas();
      add(canvas);
      setSize(frameWidth,frameHeight);
      setTitle(title + " Page: " + pageCounter);
      setVisible(true);

      //---------------------------------------//
      //Anonymous inner class to terminate the
      // program when the user clicks the close
      // button on the Frame.
      addWindowListener(
        new WindowAdapter(){
          public void windowClosing(
                                  WindowEvent e){
            System.exit(0);//terminate program
          }//end windowClosing()
        }//end WindowAdapter
      );//end addWindowListener
      //---------------------------------------//
    }//end constructor
    //==========================================//

    //This method receives a triplet of sample
    // values of type double and stores each of
    // them in a separate array object belonging
    // to the MyCanvas object.
    void putData(double valBlack,
                 double valRed,
                 double valBlue,
                 int sampleCounter){
      canvas.blackData[sampleCounter] = valBlack;
      canvas.redData[sampleCounter] = valRed;
      canvas.blueData[sampleCounter] = valBlue;
      //Save the sample counter in an instance
      // variable to make it available to the
      // overridden paint method. This value is
      // needed by the paint method so it will
      // know how many samples to plot on the
      // final page which probably won't be full.
      this.sampleCounter = sampleCounter;
    }//end putData

    //==========================================//
    //Inner class
    class MyCanvas extends Canvas{
      double [] blackData =
                      new double[samplesPerPage];
      double [] redData =
                      new double[samplesPerPage];
      double [] blueData =
                      new double[samplesPerPage];

      //Override the paint method
      public void paint(Graphics g){
        //Draw horizontal axes, one for each
        // trace.
        for(int cnt = 0;cnt < tracesPerPage;
                                          cnt++){
          g.drawLine(0,
                     (cnt+1)*traceSpacing,
                     this.getWidth(),
                     (cnt+1)*traceSpacing);
        }//end for loop

        //Plot the points if there are any to be
        // plotted.
        if(sampleCounter > 0){
          for(int cnt = 0;cnt <= sampleCounter;
                                          cnt++){

            //Begin by plotting the values from
            // the blackData array object.
            g.setColor(Color.BLACK);

            //Compute a vertical offset to locate
            // the black data on every third axis
```

```
            // on the page.
            int yOffset =
               ((1 + cnt*(sampSpacing + 1)/
                this.getWidth())*3*traceSpacing)
                                - 2*traceSpacing;

            //Draw an oval centered on the sample
            // value to mark the sample in the
            // plot. It is best if the dimensions
            // of the oval are evenly divisable
            // by 2 for  centering purposes.
            //Reverse the sign of the sample
            // value to cause positive sample
            // values to be plotted above the
            // axis.

            g.drawOval(cnt*(sampSpacing + 1)%
                   this.getWidth() - ovalWidth/2,
              yOffset - (int)blackData[cnt]
                                  - ovalHeight/2,
              ovalWidth,
              ovalHeight);

            //Connect the sample values with
            // straight lines.  Do not draw a
            // line connecting the last sample in
            // one trace to the first sample in
            // the next trace.
            if(cnt*(sampSpacing + 1)%
                               this.getWidth() >=
                                sampSpacing + 1){
              g.drawLine(
                (cnt - 1)*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)blackData[cnt-1],
                cnt*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)blackData[cnt]);
            }//end if

            //Now plot the data stored in the
            // redData array object.
            g.setColor(Color.RED);
            //Compute a vertical offset to locate
            // the red data on every third axis
            // on the page.
            yOffset = (1 + cnt*(sampSpacing + 1)/
                  this.getWidth())*3*traceSpacing
                                  - traceSpacing;

            //Draw the ovals as described above.
            g.drawOval(cnt*(sampSpacing + 1)%
                   this.getWidth() - ovalWidth/2,
              yOffset - (int)redData[cnt]
                                  - ovalHeight/2,
              ovalWidth,
              ovalHeight);

            //Connect the sample values with
            // straight lines as described above.
            if(cnt*(sampSpacing + 1)%
                               this.getWidth() >=
                                sampSpacing + 1){
              g.drawLine(
                (cnt - 1)*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)redData[cnt-1],
                cnt*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)redData[cnt]);

            }//end if


            //Now plot the data stored in the
            // blueData array object.
            g.setColor(Color.BLUE);
            //Compute a vertical offset to locate
            // the blue data on every third axis
            // on the page.
            yOffset = (1 + cnt*(sampSpacing + 1)/
                 this.getWidth())*3*traceSpacing;

            //Draw the ovals as described above.
            g.drawOval(cnt*(sampSpacing + 1)%
                   this.getWidth() - ovalWidth/2,
              yOffset - (int)blueData[cnt]
                                  - ovalHeight/2,
              ovalWidth,
              ovalHeight);

            //Connect the sample values with
            // straight lines as described above.
            if(cnt*(sampSpacing + 1)%
                               this.getWidth() >=
                                sampSpacing + 1){
              g.drawLine(
                (cnt - 1)*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)blueData[cnt-1],
                cnt*(sampSpacing + 1)%
                                 this.getWidth(),
                yOffset - (int)blueData[cnt]);
            }//end if
          }//end for loop
        }//end if for sampleCounter > 0
      }//end overridden paint method
    }//end inner class MyCanvas
  }//end inner class Page
}//end class PlotALot04
//===============================================//
```

```
Listing 38
```

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 have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

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

Keywords
Java plot multi-channel Canvas horizontal vertical axis oval overridden paint

-end-