

## Chapter 4

# Input and Output: the Open Source Physics Library

©2002 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian  
16 February 2002

We introduce the Open Source Physics library to make graphical input and output easier and give some simple examples of the `Calculation` interface.

### 4.1 Graphical Input

So far we have “hard coded” the values of various parameters in our programs. For example, if we wish to change the values of `xmin` and `xmax` in Listing 3.2, we have to edit the `Derivative2App` program, recompile it, and then run it again. It would be much easier if we could input the data from the console (command line) and even better if we could use a graphical interface. The details of input and output are the most demanding part of the syntax in any programming language. Although languages such as Fortran and C have only console input and output, our experience is that once we have written a program that has the desired format for input and output, we simply copy and modify this format when we write a new program. The details of the syntax are not easy to remember.

Input and output in Java is both easier and more difficult than it is in traditional languages. The difficulty arises from the greater power of Java which allows us to input data graphically. Greater power means more syntax. In the following, we will introduce new vocabulary and syntax associated with `Container`, `JFrame`, `JButton`, and the `ActionListener` interface among many other terms. The new classes and interfaces will be confusing at first. But Java is also easier because the object oriented nature of Java allows us to “hide” much of this syntax from the user.

We have hidden most of the syntax associated with graphical input and output in the *Open Source Physics* (OSP) library or package of classes and interfaces. Our approach will be to first learn how to use this package to do graphical input (Section 4.4), plots (Section 4.6) and then

animations and visualizations (Chapter 6) and gradually learn how this package was constructed. This package is open source and you are encouraged to look at it and even change it. We suggest that you read the following to get an idea of what is involved in building a graphical interface, and then skip to pages 57 and 60 where we give examples of how to use the OSP package.

## 4.2 An Example

One of the great attractions of Java is that device and platform independent graphics is built directly into the language. Lines, circles, rectangles, images, and text can be drawn with just a few statements. However, creating even a simple graph can require a fair amount of programming. A scale needs to be established, axes need to be drawn, and data needs to be transformed. An additional complication arises due to the fact that a graph must be able to redraw itself whenever a window is covered or resized. But plotting is a fairly routine task. First we show a simple example that uses a `Drawable` interface and the `DrawingPanel` class that are part of the OSP library.

The `Drawable` interface is defined in the `org.opensourcephysics.display` class. It contains a single method, `draw`, that is invoked automatically from a method of the `DrawingPanel` class.

```
public interface Drawable {  
    public void draw(DrawingPanel drawingPanel, Graphics g);  
}
```

`Drawable` objects, that is, objects that implement the `Drawable` interface, are instantiated and then added to a drawing panel where the objects will draw themselves in the order that they are added. Listing 4.1 displays a program that creates a drawing panel containing a circle and an arrow.

Listing 4.1: A circle and an arrow in a drawing panel.

```
package org.opensourcephysics.sip.ch4;  
import org.opensourcephysics.display.*;  
import javax.swing.JFrame;  
  
public class DrawablesApp {  
  
    public static void main(String[] args) {  
        // create a drawing frame and a drawing panel  
        DrawingPanel panel = new DrawingPanel();  
        DrawingFrame frame = new DrawingFrame(panel);  
        panel.setSquareAspect(false);  
        // create a circle and an arrow  
        Circle circle = new Circle(0, 0);  
        panel.addDrawable(circle);  
        Arrow arrow = new Arrow(0, 0, 4, 3);  
        panel.addDrawable(arrow);  
        frame.show();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    }  
}
```

The frame used in the example, `DrawingFrame`, is a subclass of `JFrame`, which we will discuss in Section 4.3.

The OSP package also defines `PlottingPanel` as a subclass of `DrawingPanel`. This class contains the necessary axes and titles to produce linear, log-log, and semilog graphs. For example, Listing 4.2 plots the period  $T$  versus the semimajor axis  $a$  of the planets in the Sun's solar system (see Chapter 8). The data arrays, `a` and `T`, contain the semimajor axis of the planets in astronomical units (AU) and the period in years, respectively. Note that the plot automatically adjusts itself to fit the data because the `autoscale` parameter is set to `true` for both axes.

Listing 4.2: Program to plot the semimajor axis versus the period.

```

package org.opensourcephysics.sip.ch4;
import org.opensourcephysics.display.*;
import java.awt.event.*;
import java.awt.*;
public class PlotDataApp {

    public static void main(String[] args) {

        PlottingPanel plotPanel = new PlottingPanel("a", "T", "Kepler's Second Law");
        DrawingFrame drawingFrame = new DrawingFrame(plotPanel);
        Dataset dataset = new Dataset();
        double[] a = {0.241, 0.615, 1.0, 1.88, 11.86, 29.50, 84.0, 165, 248};
        double[] period = {0.387, 0.723, 1.0, 1.523, 5.202, 9.539, 19.18, 30.06, 39.44};
        dataset.setConnected(false);
        dataset.append(a, period);
        plotPanel.addDrawable(dataset);
        plotPanel.setAutoscaleX(true);
        plotPanel.setAutoscaleY(true);
        plotPanel.repaint();
        drawingFrame.show();
        drawingFrame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
    }
}

```

#### Exercise 4.1. Log-log plots

- Modify `PlotDataApp` so that it plots  $\log T$  versus  $\log a$ . What can you conclude from the plot (see Section 8.5)? Another way of making a log-log plot is to replace the statement `Dataset dataset = new Dataset()` by `LogDataset dataset = new LogDataset(Axis.LOG10, Axis.LOG10)`?
- What happens if you replace `dataset.setConnected(false)` by `dataset.setConnected(true)`?
- What happens if you add the statement `plotPanel.setSquareAspect(false)`? What happens when the aspect ratio is true?

## 4.3 Graphical Input

We now discuss in brief some of the considerations that go into developing a graphical interface. Our goal is not to give a thorough introduction to graphics in Java, but to give you a general idea of what is involved.

### 4.3.1 Components

Our first task is to create a window. An object of the `JFrame` class appears on the screen as a window with a border, a title, and icons for closing and resizing the window. The following example creates a window and displays it on the screen.

Listing 4.3: An example of a window using `JFrame`.

```
package org.opensourcephysics.sip.ch4;
import javax.swing.JFrame;

public class FrameApp {
    public static void main(String[] args) {
        // create a JFrame with title 'Frame Example'
        JFrame f = new JFrame("Frame Example");
        f.setSize(400,400); // set size of frame in pixels
        f.setVisible(true); // display frame on screen
        // exit when frame is closed
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

Note that `JFrame` has a number of methods associated with it.

Java has an extensive library of standard packages that do input and output, image handling, sound, among many other essential functions. These packages are part of the Java Application Programming Interface (API). The statement,

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

means that we are using a package of graphical user interface (GUI) classes known as “Swing.” (We have implicitly been accessing the classes of `java.lang` which is imported by default.)

*Exercise 4.2.* Opening a window

Run `FrameApp` and notice that a `JFrame` can be resized. What happens when you close the `JFrame`? Remove the last statement of the program. Now what happens when you close the `JFrame`?

A `JFrame` is an example of a *component*, which is a graphical object that can be displayed on the screen. We will soon introduce other examples of components including buttons, text labels, and text boxes. A component is any object of a class that is a subclass of the abstract class `Component`. A `Container`, which inherits from `Component`, can contain other `Components`. Figure 4.1 shows the hierarchy of some of the important graphics classes. The important thing to remember is that all graphical objects are derived from `Component`.

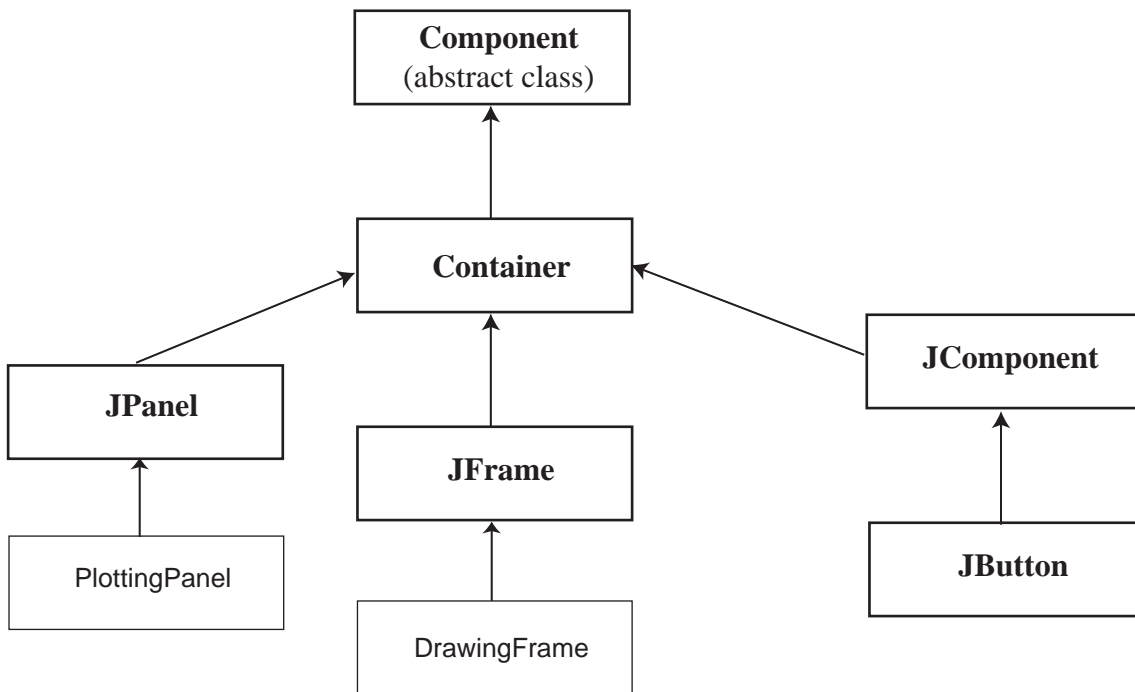


Figure 4.1: A part of the graphics class inheritance hierarchy. Classes that begin with J are part of Swing. `DrawingFrame` and `PlottingPanel` are part of the OSP library.

Now that we have created a main window using `JFrame`, how do we add other components such as buttons and text fields? Although a `JFrame` appears on the screen as a simple window, it is composed of multiple layers, each responsible for a different function. The layer of the `JFrame` that holds other components that are visible to the user is known as the **content pane**. The following code creates a frame and a button, retrieves the frame’s content pane, and adds the button to the content pane:

```

JFrame frame = new JFrame();
Container c = frame.getContentPane(); // get content pane layer of JFrame
JButton button = new JButton("Calculate");
c.add(button, BorderLayout.CENTER);

```

Add this code (and any necessary import statements) to class `FrameApp` and see what happens.

The method `getContentPane()` returns the content pane of `frame`, which is an object of type `Container`. When we add a button or text field to the content pane, what determines where the text field will be placed within the container and the size of the text field? These properties are determined by the content pane’s layout manager. By default the content pane uses a `BorderLayout` that places the objects it contains in one of five positions: `NORTH`, `SOUTH`, `CENTER`, `EAST`, or `WEST`. The `add` method adds a component such as a button to a container. The form of

the `add` method depends on the layout manager used by the container. For example, if we use the `FlowLayout` manager, no position information is needed in the `add` method.

In Listing 4.4 we construct a label, a text field, and a button for inputting `xmin`, `xmax`, and `dx` for the calculation of the derivative considered in Section 3.2.

Listing 4.4: First example of graphical input.

```

package org.opensourcephysics.sip.ch4;
import java.awt.Container;           // use abstract window (awt) toolkit
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.JButton;
import javax.swing.JLabel;

public class GUITest1App extends JFrame {
    protected JButton calculateButton;
    protected JTextField xminTextField;
    protected JTextField xmaxTextField;
    protected JTextField numberOfPointsTextField;

    public GUITest1App() {
        super("First example of graphical input");
        JLabel xminLabel = new JLabel("xmin = ");
        xminTextField = new JTextField(20); // 20 is number of columns
        JLabel xmaxLabel = new JLabel("xmax = ");
        xmaxTextField = new JTextField(20);
        JLabel numberOfPointsLabel = new JLabel("number of points = ");
        numberOfPointsTextField = new JTextField(20);
        calculateButton = new JButton("Calculate");
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(xminLabel);
        c.add(xminTextField);
        c.add(xmaxLabel);
        c.add(xmaxTextField);
        c.add(numberOfPointsLabel);
        c.add(numberOfPointsTextField);
        c.add(calculateButton);
        setSize(300,200);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        GUITest1App app = new GUITest1App();
    }
}

```

In this example we imported only the classes of `java.awt` and `javax.swing` that we needed. However, if we use many classes from the same package, it is more convenient to import the whole package by using an asterisk (\*) in place of a specific class name. For example, to import all the classes in the `awt` package, we write

```
import java.awt.*;
```

Swing is built on top of the Abstract Windowing Toolkit (`awt`) package, which is the reason that we usually have to import classes from `java.awt` as well as `javax.swing`.

*Exercise 4.3.* Textfields and buttons

Run `GUITest1App` and explain the function of the button, textfields, and textlabels. What happens when you click on the button?

### 4.3.2 Events

When the user clicks the button displayed in `GUITest1App`, it triggers an *event*. Events are typically triggered by mouse clicks and keyboard input. A GUI-based program must be able to respond to user-initiated events at any time and is said to be *event driven*.

An *event* is initiated by the user and invokes methods in objects that have registered their interest. To handle events an object must have methods that know what to do once an event is received. In `GUITest1App`, nothing happened when the button is clicked because no object registered an interest in receiving events for the button. Objects that register their interest in receiving events are known as *event listeners*, and must implement the appropriate listener interface for the event. An object is notified when a button is clicked by implementing the `ActionListener` interface and registering this object as an action listener for the button by using the `addActionListener` method. When a user clicks on the button, the `actionPerformed` method is invoked for the `ActionListeners` that registered themselves for the particular button. In Listing 4.5, we extend `GUITest1App` to create a graphical user interface that performs a calculation when the button is clicked.

Listing 4.5: Example of the implementation of the `ActionListener` interface.

```
package org.opensourcephysics.sip.ch4;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import org.opensourcephysics.numerics.Function;
import org.opensourcephysics.sip.extras.MyMath;
import org.opensourcephysics.sip.ch3.MyFunction;

public class GUITest2App extends GUITest1App implements ActionListener {

    public GUITest2App() {
        super(); // use constructor of GUITest1App
        calculateButton.addActionListener(this); // note additional statement
    }

    public void actionPerformed(ActionEvent e) { // need to add method
        String s = xmaxTextField.getText();
        double xmax = Double.parseDouble(s); // convert String to double
```

```

s = xminTextField.getText();
double xmin = Double.parseDouble(s);
s = numberOfPointsTextField.getText();
int numberOfPoints = Integer.parseInt(s);
double dx = (xmax - xmin)/numberOfPoints;
double x = xmin;
Function function = new MyFunction();
for (int i = 0; i < numberOfPoints; i++) {
    double derivative = MyMath.derivative(function,x,dx);
    System.out.println("x = " + x + " derivative = " + derivative);
    x += dx;
}

public static void main(String[] args) {
    GUITest2App app = new GUITest2App();
}
}

```

To emphasize the necessary changes, we have used inheritance to define the class `GUITest2App`. In this case it would have been just as easy to copy and paste. Note the inclusion of the `actionPerformed` method.

The method `calculateButton.addActionListener` needs to know the specified action listener that is to receive events from `calculateButton`. Recall that the keyword `this` always refers to the current object, which is the object within which the reference appears. Hence, the argument `this` in the `calculateButton.addActionListener(this)` statement notifies `GUITest2App`, which implements `ActionListener`.

## 4.4 Using the OSP Package for Graphical Input

We have now learned how to construct a program that allows us to input parameters graphically without having to recompile our program. We could continue to build a user interface that is appropriate for each program. For example, if we want to have graphical input for `ParticleMoverApp`, we could build a user interface with text fields corresponding to `dt` and `minimumHeight` and a Calculate button. However, this process is time consuming and not very object oriented because we would be doing more or less the same thing over and over again. For these reasons we have developed the `CalculationControl` class and the `Calculation` interface as part of the OSP package to allow us to easily define the input parameters. To see how to use the `Calculation` interface, look at the following example:

Listing 4.6: Example of the use of the `Calculation` interface.

```

package org.opensourcephysics.sip.ch4;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.sip.ch3.MyFunction;
import org.opensourcephysics.numerics.Function;
import org.opensourcephysics.sip.extras.MyMath;

```

```

public class Derivative3App implements Calculation {
    protected Control myControl;
    protected double xmin, xmax, dx;
    protected int numberOfPoints;
    protected Function function = new MyFunction();

    public void calculate() {
        xmax = myControl.getDouble("xmax");
        xmin = myControl.getDouble("xmin");
        numberOfPoints = myControl.getInt("numberOfPoints");
        calculateDerivative ();
    }

    public void resetCalculation() {
        myControl.setValue("xmin",0);    // default values
        myControl.setValue("xmax",1);
        myControl.setValue("numberOfPoints",10);
    }

    public void calculateDerivative() {
        double dx = (xmax - xmin)/numberOfPoints;
        double x = xmin;
        for (int i = 0; i < numberOfPoints; i++) {
            double derivative = MyMath.derivative(function,x,dx);
            System.out.println("x = " + x + " derivative = " + derivative);
            x += dx;
        }
    }

    // method used to give Calculation a reference to Control object
    public void setControl(Control c) {
        myControl = c;
        resetCalculation ();
    }

    public static void main(String[] args) {
        Derivative3App app = new Derivative3App();
        CalculationControl c = new CalculationControl(app);
        app.setControl(c);
    }
}

```

Run `Derivative3App` to see how it works.

`CalculationControl` implements the `Control` interface and is designed to control any calculation. `CalculationControl` has a text area where the user can enter parameters, a message area for output, and two buttons: calculate and reset. In our first example, the calculation simply prints the results. In Section 4.6 we will learn how to add graphical output.

Any class that implements `Calculation` must have a common set of methods that the control is guaranteed to find. These methods are listed in the code for the `Calculation` interface.

```

package org.opensourcephysics.templates;

public interface Calculation {
    public void calculate();
    public void resetCalculation();
    public void setControl(Control control);
}

```

When the calculate or reset buttons are clicked, the `CalculationControl` invokes a method with the corresponding name in a `Calculation` object. For example, when a user clicks the calculate button, the `CalculationControl` invokes the `calculate` method in the `Calculation`. The `calculate` method reads new values from the control, does the calculation, and displays the result. The `resetCalculation` method sets the default values of the input parameters for the calculation.

The important methods in the `Control` interface are listed in the following. The complete listing is given in the `OSP controls` package.

Listing 4.7: Important methods in the `Control` interface.

```

package org.opensourcephysics.templates;

public interface Control {
    // store name and value in control object
    public void setValue(String name, int val);
    public void setValue(String name, double val);
    public void setValue(String name, boolean val);
    public void setValue(String name, Object val);

    public int getInt(String name);    // get value from control object
    public double getDouble(String name);
    public String getString(String name);
    public boolean getBoolean(String name);

    public void println(String s);    // print string in control's message area
}

```

The `main` method in Listing 4.6 creates an instance of `Derivative3App` named `app`, which is the object that performs the calculation. In our example, the calculation merely print messages. The second statement creates the graphical user interface, an `Control` object named `c`. The `Control` constructor registers the `app` with the control so that the control can invoke the calculation's methods when the buttons are clicked. `statement` registers the control with the simulation. This action enables `app` to store and retrieve parameters from the control. The `main` method has done its job by creating two objects—the control object and the calculation object—and registering each object with the other. In this way we have allowed the possibility of changing the implementation of the `Control` interface without requiring any changes in the above code.

After the `main` method finishes its work, the program is running and the control is visible on the screen. The program will be terminated when the control's `JFrame` is closed.

## 4.5 Using the OSP Package to Plot a Function

In Appendix 4A we discuss additional graphics classes and their methods that allow us to make simple plots. There is a lot to digest and unless you have prior experience with Java, it is unlikely that you will be able to follow all the details. However, it is not necessary to understand these details because the OSP package “hides” them. We encourage you to look at the source code for this package as you gain more experience with Java.

We first distinguish between two levels of plotting. In the “quick and dirty” mode, we want to plot the results produced by our program to help check the program and to gain a preliminary understanding of how the system behaves. In the “presentation” mode, we want to do a careful graphical analysis of the results and make plots that we can present to others. In the latter mode we want to optimize the range of each axis, label the axes, have tick marks at convenient spacings, etc. For this second mode we recommend that you export your data to a separate program that is optimized for making graphs and fits to data. There are many open source, shareware, and commercial programs with these capabilities. These programs allow you to manipulate the data in many ways, do simple curve fitting, and to set up plots in many different formats.

We have developed the OSP display package to make plots directly from Java. The core functionality of this library is similar to more sophisticated graphics packages and provides a framework that not only plots  $(x, y)$  data sets, but that can also be used for animations and other visualizations. In this chapter we use it only to draw graphs.

As an example, we use the `Calculation` interface and `CalculationControl` class to plot the derivative of the function  $(x^3 - \sin x)e^{-x^2}$  that we computed in `Derivative3App`. We also use the `PlottingPanel` and `DrawingFrame` and `Dataset` classes that are part of the OSP package.

Listing 4.8: Example of a simple plot using the `Calculation` interface.

```

package org.opensourcephysics.sip.ch4;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.sip.ch3.MyFunction;
import org.opensourcephysics.numerics.Function;
import org.opensourcephysics.sip.extras.MyMath;
// explicitly include classes to emphasize ones that are used for the first time
import org.opensourcephysics.display.PlottingPanel;
import org.opensourcephysics.display.DrawingFrame;
import org.opensourcephysics.display.DatasetCollection;
import org.opensourcephysics.display.Dataset;

public class PlotExampleApp implements Calculation {
    Control myControl;
    double xmin, xmax;
    int numberOfPoints;
    Function function = new MyFunction();
    PlottingPanel plottingPanel;
    DrawingFrame drawingFrame;
    DatasetCollection datasetCollection;
    int datasetIndex;

    public PlotExampleApp() {

```

```

plottingPanel = new PlottingPanel("x", "y", "Derivative");
drawingFrame = new DrawingFrame(plottingPanel);
datasetCollection = new DatasetCollection();
plottingPanel.addDrawable(datasetCollection);
}

public void calculate() {
    xmax = myControl.getDouble("xmax");
    xmin = myControl.getDouble("xmin");
    numberOfPoints = myControl.getInt("number of points");
    double dx = (xmax - xmin)/(numberOfPoints - 1);
    double x = xmin;
    datasetIndex++;
    Dataset d = new Dataset();
    plottingPanel.addDrawable(d);
    for (int i = 0; i < numberOfPoints; i++) {
        double derivative = MyMath.derivative(function,x,dx);
        datasetCollection.append(datasetIndex, x, derivative);
        x += dx;
    }
    plottingPanel.repaint();
}

public void resetCalculation() {
    xmax = 1;
    xmin = 0;
    numberOfPoints = 10;
    datasetIndex = -1;
    datasetCollection.clear();
    plottingPanel.repaint();
}

// give Calculation a reference to Control object
public void setControl(Control c) {
    myControl = c;
    resetCalculation();
}

public static void main(String[] args) {
    PlotExampleApp app = new PlotExampleApp();
    CalculationControl c = new CalculationControl(app);
    app.setControl(c);
}
}

```

Note that the new graphics statements are

```

plottingPanel = new PlottingPanel("x", "y", "Derivative");
drawingFrame = new DrawingFrame(plottingPanel);

```

`PlottingPanel` is a subclass of `DrawingPanel` and contains  $x$  and  $y$  axis labels, a title, and axes

to produce linear, log-log, and semilog plots.

We store the  $x$  and  $y$  values that we want to plot in a `Dataset`, which contains an array of  $x$  and  $y$  points (see Appendix 4A). Because we might want to make multiple plots with different colors, we can add each data set to a `datasetCollection`.

We will use `PlotExampleApp` as a template for the programs in Chapter 5, where we consider again the motion of a particle in one and two dimensions.

In Table 4.1 we summarize the most common methods in the OSP display classes. The online documentation at <http://www.opensourcephysics.org/> has a complete listing and more extensive descriptions.

#### Dataset methods

<code>append</code>	Add data to the data set as $x, y$ coordinate pairs. The data can be either doubles or arrays of doubles.
<code>clear</code>	Remove all data.
<code>setConnected</code>	Connect the data points with straight lines. Default is <code>false</code> .
<code>setLineColor</code>	Set the color of the lines connecting the points. Ignored if points are not connected. Default is <code>black</code> .
<code>setMarkerColor</code>	Set the color of the data markers. Ignored if the data marker is set to <code>NO_MARKER</code> . Default is <code>black</code> .
<code>setMarkerShape</code>	Set the shape of the data markers. Valid shapes are <code>NO_MARKER</code> , <code>CIRCLE</code> , <code>SQUARE</code> , <code>FILLED_CIRCLE</code> , and <code>FILLED_SQUARE</code> . The default is <code>SQUARE</code> .

#### DrawingPanel methods

<code>addDrawable</code>	Add a drawable object to the panel. Objects will be drawn in the order that they are added.
<code>setAutoscaleX</code>	Autoscale the $x$ axis using max and min values from the data. Default is <code>true</code> .
<code>setAutoscaleY</code>	Autoscale the $y$ axis using max and min values from the data. Default is <code>true</code> .
<code>repaint</code>	Repaint all drawable objects. The repaint operation is done at the convenience of the operating system.
<code>removeAll</code>	Clear the panel by removing all the drawable objects.
<code>removeDrawable</code>	Remove one drawable object.
<code>setSquareAspect</code>	Set $x$ and $y$ scale to have equal pixels per unit.
<code>setPreferredMinMaxX</code>	Set scale on $x$ axis (set <code>autoscale</code> to <code>false</code> ).
<code>setPreferredMinMaxY</code>	Set scale on $y$ axis.
<code>setPreferredMinMax</code>	Set scale on $x$ and $y$ axes.

Table 4.1: Some of the important methods for making plots.

#### Exercise 4.4. Plotting options

- Use the OSP library to write a simple program that plots the sin function between 0 and  $2\pi$ . What is the minimum number of points that are necessary in order to recognize the sine curve?

- b. Change the marker style to `CIRCLE` and set the connected points option to `false`.

#### Exercise 4.5. Plotting two curves

Create a second data set in your program and use this data set to add a cosine curve to the plot.

```
DatasetCollection dataset2 = new DatasetCollection();
```

Plot the sine and a cosine curves simultaneously.

#### Exercise 4.6. Test of numerical derivatives

- a. Modify `PlotExampleApp` so that the function and its derivative can be plotted.
- b. Add the boolean variable, `derivativeMode` and read and store this variables from the control. Boolean variables can be read using the `getBoolean` method. Plot the derivative if `derivativeMode` is true. Test your code by taking derivatives of some analytical functions.
- c. Choose a function whose derivative can be calculated in terms of known functions. Plot the analytical and numerical derivatives on the same graph. One way to do so is to plot the (analytical) derivative (a function) with `derivativeMode` set true and then to plot the numerical result with `derivativeMode` set false. How do the two curves compare? What happens when you change  $\Delta x$ ?
- d. Plot the function  $\sin(1/x)$  and its numerical derivative on the interval  $(-1,1)$  using 10000 points. What happens to the plot of the derivative as  $\Delta x$  takes on the values 0.1, 0.01, 0.001, and 0.0001? Why?

## 4.6 Model-View-Control

Experienced programmers approach a programming project not as a coding task but as a design process. They look for data structures and behaviors that are common to other problems they have solved. Separating the physics from the user interface and the data visualization is one such approach. In keeping with computer science terminology, we refer to the user interface as the *Control*. It is responsible for handling events and passing them on to other objects. The plots that we have constructed presents a visual representation of the data, and is an example of a *View*. By using this design strategy it is possible to have multiple views of the same data. For example, we could show a plot and a table view of the same data. Finally, the physics can be thought of as a *Model* that maintains the data and provides the methods by which that data can change.

The Model-View-Control (MVC) design pattern that we have used for `PlotExampleApp` is one of the most successful software architectures ever devised. It is the basis for the `smallTalk` programming language and was used extensively in designing the Java Swing components. Java is an excellent language with which to implement the MVC pattern. Java allows us to isolate the model, the control, and the view in separate classes, and makes it easy to reuse these classes in various projects. And Java makes it easy to add new features to a class without having to change the existing code. These features are known as encapsulation, code reuse, and polymorphism, and are among the hallmarks of object oriented programming.

## Appendix A: Plotting a Function

We have seen that Java provides many components such as buttons, text fields, and labels and have learned how to use the `OSP` package to create buttons and text labels for graphical input. However, suppose we want to plot a function. No component is provided that is designed to display such an object. The solution is to do our own painting by subclassing a `JPanel`. In this Appendix we examine various Java classes and components by building a simple version of the `OSP` drawing package that allows us to draw a red square on the screen and make a very simple plot.

A `JPanel` is a blank component that has a surface on which we can draw. The painting code goes in the method `paintComponent`, which we override from `JComponent`, the parent class of `JPanel`. Before we do any painting, we invoke `super.paintComponent`, which paints the background of the panel. The `Graphics` object that is passed as a parameter to the `paintComponent` method is used to draw the output. The following example shows how to draw a filled red square on a panel.

Listing 4.9: Red square on a JPanel.

```
package org.opensourcephysics.sip.ch4;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.JPanel;
import javax.swing.JFrame;

// draw filled red square on a JPanel
public class SquarePanelApp extends JPanel {

    public void paintComponent(Graphics g) {
        super.paintComponent(g); // paint background
        // set color to red, all subsequent operations on graphics object g use this color
        g.setColor(Color.red);
        int rpix = 30; // width of square in pixels
        int xpix = 50; // center x position of square
        int ypix = 50; // center y position of square
        // draw filled square on panel
        g.fillRect(xpix - rpix,ypix - rpix,2*rpix,2*rpix);
    }

    public static void main(String[] args) {
        SquarePanelApp squarePanel = new SquarePanelApp();
        JFrame frame = new JFrame();
        Container c = frame.getContentPane();
        c.add(squarePanel, BorderLayout.CENTER);
        frame.setSize(400,400);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

To draw a plot, we need to store the values that we want to plot. We will store these values in two arrays. Our first attempt at a dataset class is given in Listing 4.10.

Listing 4.10: Dataset class.

```

package org.opensourcephysics.sip.ch4;

public class Dataset {
    protected double[] xpoints; // array of x points
    protected double[] ypoints; // array of y points
    protected int index;        // current index of arrays

    public Dataset(int numberOfPoints) {
        xpoints = new double[numberOfPoints];
        ypoints = new double[numberOfPoints];
        index = 0;
    }

    public void append(double x, double y) {
        xpoints[index] = x;
        ypoints[index] = y;
        index++;
    }
}

```

One limitation of a `JPanel` is that it uses a pixel-based coordinate system with its origin at the top left of the panel; its  $y$  value increases as we move down the panel. These coordinates are inconvenient, so we need to create methods that transform screen (pixel) coordinates to a more natural coordinate system known as *world coordinates*. We construct a `DatasetPanel` to draw our dataset.

Listing 4.11: DatasetPanel.

```

package org.opensourcephysics.sip.ch4;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JPanel;

public class DatasetPanel extends JPanel { // draw one dataset
    // Dimension object has width and height
    Dimension size; // size of panel in pixels
    double xmin, ymin, xmax, ymax; // world coordinates
    Dataset dataset; // dataset to be drawn on panel
    int markerSize = 4; // size in pixels of marker in dataset

    public DatasetPanel (Dataset _dataset) {
        dataset = _dataset;
    }

    public void paintComponent (Graphics g) {
        super.paintComponent(g); // paint background
    }
}

```

```

size = getSize ();          // get size of panel, method inherited from Component
for (int i = 0; i < dataset.index; i++) {
    // convert world coordinates of dataset to pixels
    int xpix = xToPix(dataset.xpoints[i]);
    int ypix = yToPix(dataset.ypoints[i]);
    g. fillRect (xpix - markerSize/2, ypix - markerSize/2, markerSize, markerSize);
}
}

// set min and max values in world coordinates that will appear on panel
public void setXYMinMax (double _xmin, double _xmax, double _ymin, double _ymax) {
    xmin = _xmin;
    xmax = _xmax;
    ymin = _ymin;
    ymax = _ymax;
}

// convert x from world units to pixel units
public int xToPix (double x) {
    double ratio = (x - xmin)/(xmax - xmin);
    double pix = ratio*size.width;
    if (pix > Integer.MAX_VALUE) {
        return Integer.MAX_VALUE;
    }
    if (pix < Integer.MIN_VALUE) {
        return Integer.MIN_VALUE;
    }
    return (int)Math.floor(pix);
}

// convert y from world units to pixel units
public int yToPix (double y) {
    double ratio = (y - ymin)/(ymax - ymin);
    // pix is defined differently from above because y pixel origin is at top of panel
    double pix = size.height - ratio*size.height;
    if (pix > Integer.MAX_VALUE) {
        return Integer.MAX_VALUE;
    }
    if (pix < Integer.MIN_VALUE) {
        return Integer.MIN_VALUE;
    }
    return (int)Math.floor(pix);
}
}
}

```

We now test our `Dataset` and `DatasetPanel`, by drawing the function,  $(x^3 - \sin x)e^{-x^2}$ , that we defined on page 42.

Listing 4.12: Plot of test function.

```
package org.opensourcephysics.sip.ch4;
```

```

import java.awt.BorderLayout;
import javax.swing.JFrame;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.sip.ch3.MyFunction;
import org.opensourcephysics.numerics.Function;
import org.opensourcephysics.sip.extras.MyMath;

// plot function
public class DatasetPanelTestApp {
    public static void main (String[] args) {
        Function function = new MyFunction();
        double xmin = 0;
        double xmax = Math.PI;
        int numberOfPoints = 10;
        double dx = (xmax - xmin)/(numberOfPoints-1);
        Dataset dataset = new Dataset(numberOfPoints);
        double x = xmin;
        for (int i = 0; i < numberOfPoints; i++) {
            dataset.append(x, function.evaluate(x));
            x += dx;
        }
        JFrame frame = new JFrame("DatasetPanel Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        DatasetPanel panel = new DatasetPanel(dataset);
        panel.setXYMinMax(-1, Math.PI+1, -2, 2);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        frame.setSize (400, 400);
        frame.setVisible (true);
    }
}

```

The above design of the panel class works fine for the above case of drawing one scatter plot of a dataset. What modifications do we need to make to our panel if we want to draw two plots? Or what if we want to draw a line plot instead of a scatter plot? How about five particles in a box, or what if we want to represent particles as rods instead of circles? The objects that we would like to draw will change frequently from program to program. With our current design, we need to modify the panel class each time we change what we are drawing, which is time consuming and inefficient. We need to keep what and how we draw separate from the panel, whose purpose is to convert between world and pixel coordinates and provide a blank canvas onto which we can draw.

A solution to this problem involves creating the `Drawable` interface that allows any object to draw itself on our panel, which we call the `PixelPanel`. This panel maintains a list of all the objects that we want to be displayed on the panel. When the panel paints itself (using the `paintComponent` method), it iterates through its list of `Drawable` objects and tells each `Drawable` object to draw itself on the panel. The `Drawable` interface is shown below.

Listing 4.13: Drawable interface.

```

package org.opensourcephysics.sip.ch4;

public interface Drawable {

```

```

// panel is the PixelPanel on which we will draw
// call methods xToPix and yToPix on the panel to convert between world and pixel coordinates
// g is graphics object that we will use to perform our drawing onto the PixelPanel
public void draw(PixelPanel panel, Graphics g);
}

```

In Listing 4.14 we define a dataset that knows how to draw itself on a `PixelPanel`:

Listing 4.14: A Dataset that can draw itself.

```

package org.opensourcephysics.sip.ch4;
import java.awt.Graphics;

public class DrawableDataset extends Dataset implements Drawable {
    int markerSize = 4; // size in pixels of marker

    public DrawableDataset(int numberOfPoints) {
        super(numberOfPoints);
    }

    public void draw(PixelPanel drawingPanel, Graphics g) {
        for (int i = 0; i < index; i++) {
            int xp = drawingPanel.xToPix(xpoints[i]);
            int yp = drawingPanel.yToPix(ypoints[i]);
            g.fillRect(xp - markerSize/2, yp - markerSize/2, markerSize,
                markerSize);
        }
    }
}

```

Finally, we define the `PixelPanel` class, which is a simplified version of the `DrawingPanel` class in the OSP display package:

Listing 4.15: `PixelPanel` class.

```

package org.opensourcephysics.sip.ch4;
import java.awt.Dimension;
import java.awt.Graphics;
import javax.swing.JPanel;
import java.util.ArrayList;

public class PixelPanel extends JPanel {
    Dimension size; // same as previous example
    // list of objects that will draw themselves on panel
    ArrayList drawableList = new ArrayList();
    double xmin, xmax, ymin, ymax;

    public void paintComponent(Graphics g) {
        super.paintComponent(g); // paint background
        size = getSize(); // update size of panel for xToPix and yToPix methods
        for (int i = 0; i < drawableList.size(); i++) {
            /* iterate through list of drawable objects, invoking

```

```

        draw method for every drawable object */
        Drawable drawable = (Drawable) drawableList.get(i);
        drawable.draw(this, g);
    }
}

// method that drawable objects can use to tell panel that they want to be drawn on panel
public void addDrawable(Drawable drawable) {
    drawableList.add(drawable);
}

// following methods are the same as the previous example
public void setXYMinMax(double _xmin, double _xmax, double _ymin, double _ymax)
public int xToPix(double x)
public int yToPix(double y)
}

```

In the above, we used the `ArrayList` class which implements an array that can vary automatically in size.

In Listing 4.16 we give an example of how to use these classes to plot the sin function.

Listing 4.16: Example of plot of sine function.

```

package org.opensourcephysics.sip.ch4;
import javax.swing.JFrame;
import java.awt.BorderLayout;
import java.awt.Container;

public class PixelPanelTestApp {
    public static void main(String[] args) {
        PixelPanel panel = new PixelPanel();
        JFrame frame = new JFrame();
        Container c = frame.getContentPane();
        c.add(panel, BorderLayout.CENTER);
        frame.setSize(400,400);
        frame.setVisible(true);
        double xmax = Math.PI;
        double xmin = 0;
        double ymax = 1;
        double ymin = -1;
        panel.setXYMinMax(xmin,xmax,ymin,ymax);
        int numberOfPoints = 10;
        double dx = (xmax - xmin)/(numberOfPoints-1);
        DrawableDataset dataset = new DrawableDataset(numberOfPoints);
        panel.addDrawable(dataset);
        double x = xmin;
        while (x <= xmax) {
            double y = Math.sin(x);
            dataset.append(x,y);
            x += dx;
        }
    }
}

```

```

    }
    panel.repaint ();
  }
}

```

The `org.opensourcephysics.display` package contains the `PlottingPanel` class that we will use to draw plots. This class is similar to the `PixelPanel` class, with some added functionality, such as an  $x$  and  $y$  axis. We also will use the `Dataset` class in the display package, which is similar to the `DrawableDataset`. Listing 4.17 shows how to use these classes to plot the sin function.

Listing 4.17: Use of OSP library to plot a function.

```

package org.opensourcephysics.sip.ch4;
import org.opensourcephysics.display.Dataset;
import org.opensourcephysics.display.PlottingPanel;
import org.opensourcephysics.display.DrawingFrame;

// example that shows how to use OSP tools to make a plot
public class SimplePlotApp {
    public static void main(String[] args) {
        PlottingPanel panel = new PlottingPanel("x", "y", "sin x");
        DrawingFrame frame = new DrawingFrame(panel);
        double xmax = Math.PI;
        double xmin = 0;
        double ymax = 1;
        double ymin = -1;
        panel.setPreferredMinMax(xmin,xmax,ymin,ymax);
        int numberOfPoints = 10;
        double dx = (xmax - xmin)/(numberOfPoints-1);
        Dataset dataset = new Dataset();
        panel.addDrawable(dataset);
        double x = xmin;
        while (x <= xmax) {
            double y = Math.sin(x);
            dataset.append(x,y);
            x += dx;
        }
        panel.repaint ();
    }
}

```

## References and Suggestions for Further Reading

David M. Geary, *Graphic Java: Mastering the JFC*, Vol. 1, *AWT* and Vol. 2, *Swing*, Prentice Hall (1999).

Jonathan Knudsen, *Java 2D Graphics*, O'Reilly (1999).