Chapter 11

2D Plotting with PtPlot; classes and packages

It only seems fair for us to keep programs simple while you are in the process of learning how to write them. However, we would like to encourage you to include plotting as part of your programs, both because it's a good way to visualize your results, and because its fun. Fortunately, there is an excellent Java plotting package called *PtPlot* that is both free and easy to use, and in § **??** we will show you how to use it.

In order to use *PtPlot*, you will need to work with multiple classes. Yet understanding how to do this is rather advanced for beginners. Accordingly, you can either use the sample programs without understanding them (the "black box" or operational approach), or you can try to make some sense of the commands that you use, and in the process get familiar with some advanced Java notation and techniques. In this section we talk about multiple classes and packages.

We suggest that, in the least, you skim this section. You may well want to return to it after you have more experience with Java notation and objects. *However, if the material to follow appears too advanced for you, you can still jump to the* § **??** *on PtPlot and just follow the instructions there.*

11.1 Classes and Packages*

Up until this point, your programs have contained a single class contained in a file with a `.java` extension. That class contained a main method and possibly a number of other methods. Just as we encourage you to modify old programs rather than writing all programs from scratch, so we encourage you to include methods you have already written and debugged when you write your new programs. In addition, Java contains a large number of *libraries* consisting of collections of methods for various purposes (for example, to draw a pull–down menu).

Accordingly, it is most efficient if you do not try to write all programs completely from scratch, but rather think of the existing libraries (your personal ones as well of others) as *tool boxes* from which you can extract the individual tools needed for you to construct your program. This will save you time, both because you do not have to write all the methods, and because the methods have already been debugged (usually the most time consuming part of programming). In fact, the *object–oriented* approach of Java is specifically designed to make reuse of components easier and safer.

11.1.1 Using Packages

We have already referred to a collection of related methods as a library. For example, the math library contains the methods that calculate various math functions like sines and cosines. In Java terminology, each method would be in a class file, so we could also say that a library is a collection of related classes. In Java terminology, libraries are called *packages*.

In general, there are two types of packages: the standard Java packages that constitute the Java language, and user–defined packages that extend standard Java. Some of the standard Java packages that we employ in this book are:

Java Package	Classes for
`java.lang`	Basic elements of Java language.
`java.util`	Utilities; random number generators, date, time, *etc.*.
`java.awt`	Abstract Windowing Toolkit; creating graphical user interfaces.
`java.applet`	Creating applets and interact with browsers
`java.beans`	Creating reusable software components.
`java.io`	Data input and output

The PtPlot package is an example of a user–defined package.

Because these Java packages contain hundreds or thousands of classes (sometimes with methods of the same name), some organization is necessary to keep tract of what each method does. Java does this with a hierarchical directory structure in which there are parent packages containing sub-packages, with the sub-packages containing more sub-packages or various classes. To make the name of each class unique, one precedes it with the names of the package and sub-packages that contain this class.

For example, consider the command

System.out.println

that we have been using to print a line of output on the screen. Here `System` is the name of the class (classes begin with capital letters) containing many of Java's methods. When we give the combination **System.out**, we are referring to an object (or the class that creates it) that represents the **standard output stream** (what gets written on your screen). The final `println` that gets affixed to `System.out` is the name of a method in the class `System` that prints lines (in this case it adds a line to the object representing the output stream).

To repeat, by convention, the names of classes are capitalized, which helps set them apart from packages which have lowercase names. This is relevant here because the **System.out.println** command is in the `java.lang` package, and so the proper full name of the command is actually

java.lang.System.out.println,

which contains the package name as well. Because `java.lang` is the most basic package, the java compiler automatically looks there to find the methods we invoke, so we can (fortunately) leave off the `java.lang` prefix. (Presumably, `java` is the main package and `lang` is a sub-package, but it is easier to just say `java.lang` is the package.)

We must admit that we sometimes find Java's naming conventions confusing. Nevertheless, we will use them when importing packages and in using use methods from other classes, so some familiarity with it the conventions is helpful.

You include the classes from a package with the **import** command. The command can be given in one of two forms:

`import <packageName>.<specific classes>` import specific classes from packagenName

`import <packageName>.*`	import all classes from packagenName

The **import** command tells the Java compiler to look in the package `packageName` for methods that it might not find otherwise. However, for this to work the compiler must know *where* the package of classes and their methods is stored on your particular computer. In other words, the compiler needs to know the path to follow through the local disk memory to get to the directory or folder where the classes are stored. Accordingly, each computer system, be it Windows, Unix, or MacOS, has an environmental variable named `CLASSPATH` that contains the explicit path to where the classes are stored on that particular computer. As we show in the PtPlot case below, you will need to modify this variable before a package can be imported.

Even though what follows is more advanced programming than we do in this book, for completeness we indicate how you could create your own packages. This is done by placing several classes in the same `.java` file, and then including a **package** command at the beginning of the file:

```
package <mypackage_name>;
public class <myclass1_name>
 {
 <normal class structure, multiple methods OK>
 }
public class <myclass2_name>
 {
 <normal class structure, multiple methods OK>
 }
```

Note, your package may be a collection of methods *without* any main method, for example, mathematical subroutines that are called from all the programs that you write. However, there must be one main method someplace, if the program is to run since execution always begins in a main method. Likewise, the main method must be a public class (other classes can read only the public classes).

11.2 Graphing with PtPlot

One of the exceptional things about Java is that it contains commands at the programming level that permit you to draw graphs on your computer screen — regardless of your operating system. Yet these commands are rather low level, as we shall in see Chapter **??**, *Web Computing*, and you must attend to a number of details if you want to use truly elementary Java graphics to make your plots.

To avoid all that fuss, we recommend the use of PtPlot [PtPlot],

http://ptolemy.eecs.berkeley.edu/java/ptplot/

as a basic plotting package for 2-D graphs[1]. PtPlot is free, supported by the University of California, written in Java (and thus runs under Unix, Linux, Mac's and MS Windows), and is easy to use for 2-D data plotting or histograms[2]. It can be incorporated right into your programs or applets (applets are discussed in Chapter **??**, *Web Computing*), or used as a stand–alone application or applet. We include on the disk a Java library (class file) containing the needed methods to plot graphs; alternatively, you may download the most recent version over the Web (we describe how to do that in a separate section).

[1] This section prepared with the help of Connelly Barnes.

[2] For our research use, and for many of the graphs in this book, we use **AceGr/Xmgr** for 2-D plotting and **gnuplot** for 3-D graphics. Yet AceGr is available only for Unix systems at present, while gnuplot, which is available for Unix and MS Windows, cannot be called from Java programs. In Chap. **??**, *Electric Potential of Realistic Capacitor*, we present a short tutorial on the use of gnuplot.

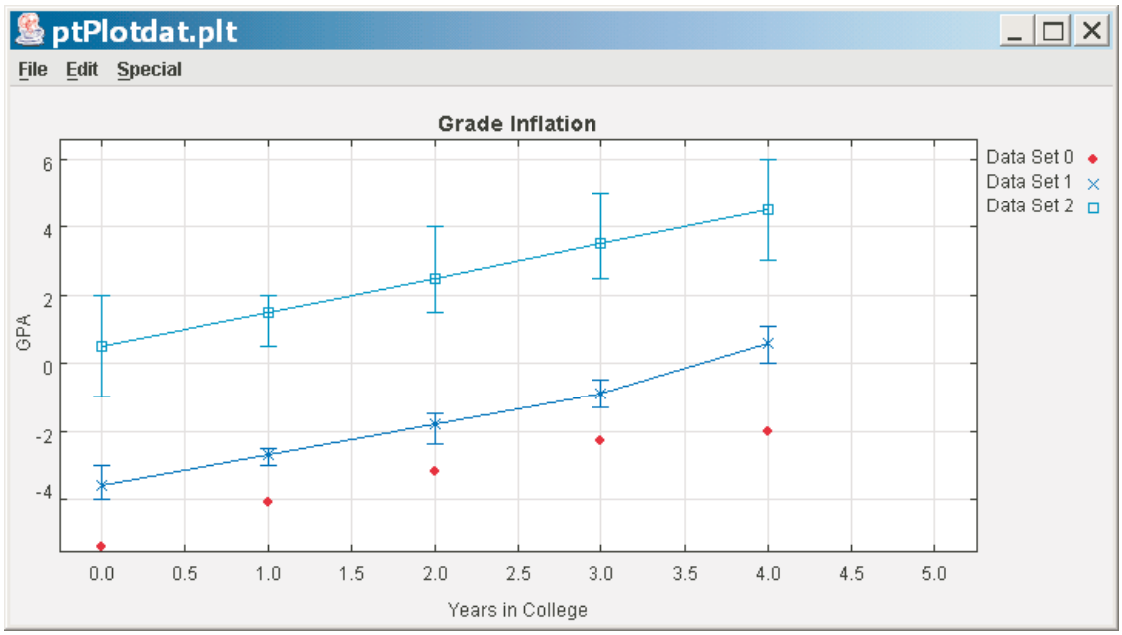

Figure 11.1: Sample output from elementary use of PtPlot in which three data sets are placed on one plot. Note how error bars are included for two. The data file read by PtPlot here is in `ptPlotdat.plt`.

11.2.1 EasyPtPlot.java

As an example of how this works, here is a program `EasyPlot.java` that plots the function $\cos(x)$ versus x:

```
1.  /* EasyPtPlot.java by D.McI, C.B, RHL, 8/03 OSU
2.   * Plot function using PtPlot package */
3.
4.  import ptolemy.plot.*;      // Import plotting package.
5.  public class EasyPtPlot
6.  {
7.     public static void main(String[] args)
8.     {
9.       // Create ptplot ``Plot'' object
10.      Plot myPlot = new Plot();
11.
12.      myPlot.setTitle("f(x) vs x");
13.      myPlot.setXLabel("x");
14.      myPlot.setYLabel("f(x)");
15.      // Add (x,cos(x)) data points to Plot object using addPoint
16.
17.      for (double x = -5.; x <= 5.; x += 0.02)
18.      {
19.        /* myPlot.addPoint( int dataSet, double x, double y, boolean connect)
20.        dataSet != 0 for multiple functions on same graph
21.        connect = ``true'' to connect points */
22.
23.        double y = Math.cos(x);
24.        myPlot.addPoint(0, x, y, true);
25.      }
26.
27.      // Create PlotApplication to display Plot object
28.      PlotApplication app = new PlotApplication(myPlot);
29.    }
30. }
```

Before you try using this file, you need to have PtPlot installed on your computer (see your system administrator or §**??** to do it yourself).

Because it is so much more fun to be able to see your results graphically, we recommend you start including graphical output in your programs, even as you are still learning Java. That being the case, at first you may just use these commands without fully understanding how they go about their business. As you get more familiar with Java, the innards of this "black box" will make more sense to you. However, there are several parts of **EasyPtPlot.java** that you will need to understand so that you can modify it for your purposes. So let's open the box a bit.

On line 4 we see the statement `import ptolemy.plot.*;`. This copies in or includes the plotting package programs (class files) with your program[3].

In a general sense, PtPlot deals with an *object* that represents your plot. We call this object `myPlot` and create it on line 10. There it is created as a variable type that PtPlot has defined as `Plot`[4]. We then add various features, step–by–step, to `myPlot` to make it just the plot we want. On line 12 we give it a title, and on lines 13 and 14 we label the x and y axes. Note, as is standard with objects in Java, we first give the name of the object and then modify it with "dot modifiers". (These modifiers are like the arguments to the *plot* command in Maple, only here the arguments get attached to the objects.)

At this point could also tell PtPlot what range of x and y values are to be plotted, but we don't! Instead we let PtPlot set the x and y ranges based on the extents of the data which it is given.

Line 17 begins a `for` loop that repeats itself for a number of increasing `x` values. (`for` loops are discussed in Chap.10, *Simple Projectile Motion.*) During each repetition, line 23 calculates a new value of $y = \cos(x)$ and then line 24 adds a point (x, y) to your plot object. By having `true` as the fourth argument in `myPlot.addPoint(0, x, y, true)`, we are telling PtPlot to connect the previous point to the new one. (This is explained in the comments on lines 19-21.)

You will need to place your own function, or data to be plotted, on line 23 where we call $\cos(x)$. If you have written a method or function whose output you wish to plot, it can be called on line 23.

So far, your plot object `myPlot` exists only in the computer's memory. For you to have that plot appear on the computer screen in front of you, you need line 28 to create a `PlotApplication` with your plot as its input. Then when the program is run, `PlotApplication` runs another program to display your plot on the screen.

11.2.2 Running the Sample Program

Our program `EasyPtPlot.java` is about as simple as you can get. However, you can make it fancier by including more options as commands, or by using the pull-down menus in the PtPlot window in which your plot is displayed. We discuss the options below, but before that, we recommend that you run the sample program now and see for yourself what we will be talking about.

- Compile and execute `EasyPtPlot.java` in the usual way:

`> javac EasyPtPlot.java`	Compile EasyPtplot.java into EasyPtplot.class.
`> java EasyPtPlot`	Run EasyPtplot.class.

[3]More specifically, you are copying a subdirectory `ptolemy`, which, in turn, contains the subdirectory `plot`. The `*` is a wild card character, and its use here means to copy all the files (class files) in that subdirectory.

[4]Objects in their own right are discussed in Chapters 12 and 13. Here we get some practical experience with them, which will help when we need to understand them.

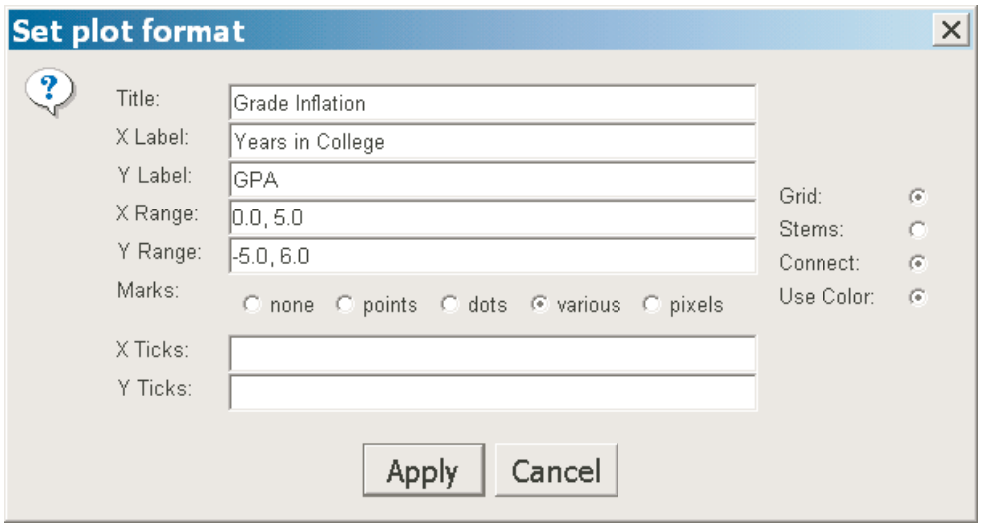

Figure 11.2: The em Format submenu available under the *Edit* menu when running a PtPlot `PlotApplication`. This submenu basic control of the plot's features.

- You should get a pretty window on your screen just like Fig. **??**. You may move it anywhere on your screen by grabbing the bar on top, and resize it by grabbing a corner and pulling. Try that!
- By grabbing a corner and pulling, make the window fill about one quarter of your screen.
- Notice the *Edit* pull–down menu. Select *Edit* and pull it down. (Alternatively, the *E* indicates that you can do this via keystrokes alone. In this case, `Alt + E` also pulls down the menu.)
- From the *Edit* pull–down menu select *Format*. You should get a window like that in Fig. **??** which lets you control most of the options in your graph.
- Experiment with the *Format* menu. In particular, change the graph so that only points are plotted, with the points being pixels and black, and so that your name is in the title.
- Select a central portion of your plot and **zoom in** on it by drawing a box (with mouse button depressed) starting from upper–left corner and then moving down before you release the mouse button. You **zoom out** by drawing a box from the lower right corner and moving up. You can also resize your graph by selecting *Special/Reset Axes* or by resetting the x and y ranges. And of course, you can always start over by closing the Java window and running the `java` command again.
- Notice that you can print your graphs under the *File* menu, as well as write them to file in postscript (.ps format). You may also export plots in various formats.

11.2.3 PtPlot Options

On the disk and Web we also give some examples containing more options. The program `ProjectilePlot.java`, used in Chap.10, places several graphs on same plot, as well as generating different plots. In addition, the program `TwoPlotExample.java` and its data file `data.plt`, taken from the PtPlot web site, shows how to place two plots side–by–side, and how to read in a data file containing error bars and various symbols for the points. Here we list of some of

the options available. More are to be found in the description of the commands (methods) on the PtPlot Web site [PtPlot].

Calling PtPlot from Your Program

Plot myPlot = new Plot();	Name and create plot object `myPlot`.
PlotApplication app = new PlotApplication(myPlot);	Display plot on screen (after stuff below).
myPlot.setTitle("f(x) vs x");	Add title to plot.
myPlot.setXLabel("x");	Label x axis.
myPlot.setYLabel("f(x)");	Label y axis.
myPlot.addPoint(0, x**,** y**, true);**	Add (x, y) to plot set 0, connect points.
myPlot.addPoint(1, x**,** y**, false);**	Add (x, y) to set 1, don't connect points.
myPlot.addLegend(0, "Set 0");	Label data set 0 in legend.
myPlot.addPointWithErrorBars(0, x**,** y**,** yLo**,** yHi**, true);**	Plot, (x, y-YLo), (x, y+yHi) with error bars.
myPlot.clear(0);	Remove all points from data set 0.
myPlot.clear(false);	Remove data from all sets.
myPlot.clear(true);	For all sets, remove points, set options to default.
myPlot.setSize(500, 400);	Set plot size in pixels (optional).
myPlot.setXRange(-10., 10.);	Set x range (default fit to data).
myPlot.setYRange(-8., 8.);	Set a y range (default fit to data).
myPlot.setXLog(true);	Use log scale for x axis.
myPlot.setYLog(true);	Use log scale for y axis.
myPlot.setGrid(false);	Turn off the grid.
myPlot.setColor(false);	Color in black and white.
myPlot.setButtons(true);	Display zoom-to-fit button on plot.
myPlot.fillPlot();	Adjust x, y ranges to fit data.
myPlot.setImpulses(true, 0);	Lines down from points to x axis, set 0.
myPlot.setMarksStyle("none", 0);	Draw points as "`none`" (or `points, dots, various, pixels`).
myPlot.setBars(true);	Display data sets as bar charts.
String s = myPlot.getTitle();	Extract title as string (also works for other properties).

Sometimes you may have your data in a file, possibly because it came from another application or because you wanted to get it all computed or measured before you started looking at it. If you write your data in a format that PtPlot can read, then you will be able to plot it and replot at your pleasure. Although this may sound like a lot of work, most reasonable formats will work. However, you can also include some formatting commands in the data file, and those are specific to PtPlot.

In the simplest form, a **PtPlot Data Format** is just a text file with a single x, y point per line. The x and y values may be separated by spaces, tabs, or commas. To plot up your data files, enter

```
> java ptolemy.plot.PlotApplication dataFile
```
Plot data in file `dataFile`.

This causes the standard PtPlot window to open and display your data. (If this does not work, then your `CLASSPATH` variable may not be defined properly, or PtPlot may not be installed. See *Installing PtPlot* below.)

Alternatively, you can read your data in from the PtPlot window. You can either use a window you already have open, or issue the Java run command without a data file name:

```
> java ptolemy.plot.PlotApplication
```
Open PtPlot window.

To look at your data from the PtPlot window, you can choose

$$\text{File} \rightarrow \text{Open} \rightarrow \text{yourDataFile} \tag{11.1}$$

Note that by default, PtPlot will look for files with `.plt` or `.xml` suffixes. However, you can enter any name you want, or pull down the `Filter` menu and select `*` to see all of your files. The same options holds for the *File* → *SaveAs* option. (In addition, you can *Export* your plot as an Encapsulated PostScript (.eps) file, a format useful for inserting in documents.)

In addition to having your data plotted, you should label your axes, add a title, and customize other parts of your plots to make them informative and clear. To do these things, you can include PtPlot commands with your data, or work with the pull–down menus under *Edit* and *Special* in the PtPlot window. The options are essentially the same as ones you would call from your program. Below is a useful lists. A complete and up–to–date list can be found on the PtPlot web site.

PtPlot Data Format

TitleText: f(x) vs x	Add title to plot.
XLabel: x	Label x axis.
YLabel: y	Label y axis.
XRange: 0, 12	Set x range (default: fit to data).
YRange: -3, 62	Set y range (default: fit to data).
Marks: none	(Default) No marks at points, lines connects points.
Marks: points	or: dots, various, pixels.
Lines: on/off	Do not connect points with lines; default: `on`.
Impulses: on/off	Lines down from points to x axis; default: `off`.
Bars: on/off	Bar graph (turn off lines) default: `off`.
Bars: width (, offset)	Bar graph; bars of `width` and (optional) offset.
DataSet: *string*	Specify data set to plot; *string* appears in legend.
x, y	Specify a data point; comma, space, tab separators.
move: *x, y*	Do not connect this point to previous.
x, y, yLowErrorBar, yHighErrorBar	Plot (x, y-YLo), (x, y+yHi) with error bars.

Note, if commands appear before `DataSet` directives, then the command will apply to all data sets. If commands appear after `DataSet` directives, then it will apply to that data set only.

On the disk and on the Web we give sample programs `EasyPlot.java` and sample data file `PtPlotdat.plt`. By convention, PtPlot looks for its data files with a `.plt` extender. This program and sample data file are simple to use. Output from `EasyPtPlot` is shown in Fig. **??**. We also give the program **TwoPlotExample.java** and its data file **data.plt**. These are from the PtPlot home page and demonstrate how to place several plots next to each other. It is more advanced. Output from `TwoPlotExample` is shown in Fig. **??**.

11.2.4 Sample PtPlot Data file `PtPlotdat.plt`

```
# This is a comment: Sample data for PtPlot

TitleText: Grade Inflation
XRange: 0,5
YRange: -5, 6
Grid: on
XLabel: Years in College
YLabel: GPA
Marks: various
NumSets:3
Color: on
```

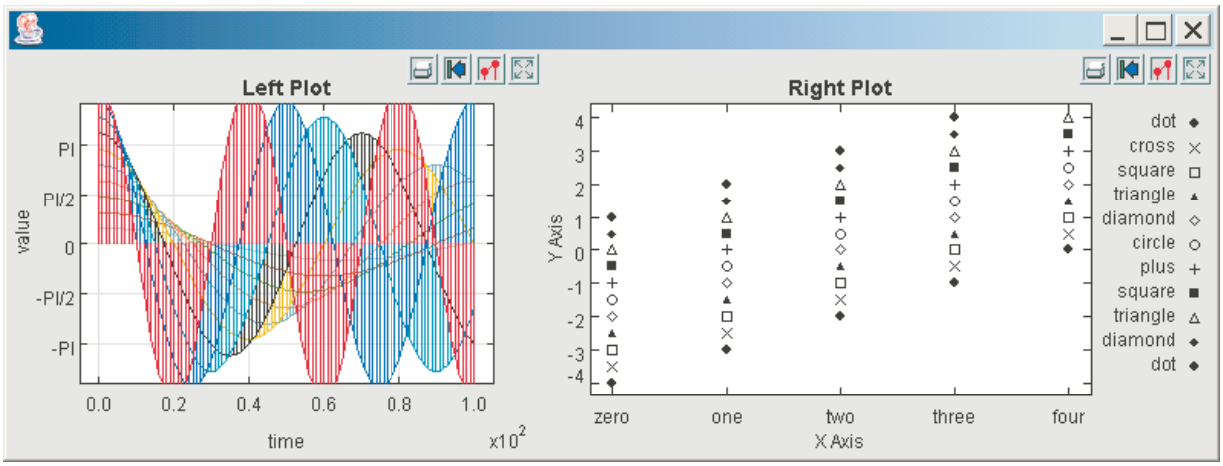

Figure 11.3: Sample output from advanced use of PtPlot in which two plots are placed side by side (in file `TwoPlotExample.java`).

```
DataSet: Data Set 0
Lines:off
0,-5.4
1,-4.1
2,-3.2
3,-2.3
4, -2

DataSet: Data Set 1
Lines:on
0,-3.6, -4,-3
1,-2.7, -3, -2.5
2,-1.8, -2.4,-1.5
3,-0.9, -1.3, -0.5
4, 0.6, 0,1.1

DataSet: Data Set 2
0,0.5, -1,2
1, 1.5, 0.5, 2
2, 2.5, 1.5, 4
3, 3.5, 2.5, 5
4, 4.5, 3, 6
```

11.3 Appendix: Installing PtPlot

The first step in setting up PtPlot is to download its latest version and then uncompress it[5]. The plotting package we call PtPlot is part of a larger Java project called *Ptolemy*. You can find documentation and download this free software from

```
http://ptolemy.eecs.berkeley.edu/java/ptplot/
```

After you have properly unzipped or untarred the Ptplot package, a directory such as `ptplot5.2` should be created. The number 5.2 here is the version number of the PtPlot package that we are now using; there may be a newer version when you do your download. For our examples, we have renamed the directory in which PtPlot resides as simply

[5] If you have never done this before, you may want to do it with a friend who has. In Windows, you can use *WinZip* to unzip files [`http://www.winzip.com/`]. In Unix, you can try double–clicking on an icon of the file, or decompress it from within a shell with `gunzip` and `tar -xvf`.

`ptplot`, with no version number attached. On Unix, we assume that your ptplot directory is ~/java_packages/ptplot, where the ~ indicates your home directory. On Windows, we assume that your ptplot directory is C:\ptplot[6]. Advanced users may prefer to keep the version number in the directory name, or use a different organizational system. However, if this is the first time that you have installed a Java package, we recommend that you use the same directory names as we.

Now that we have placed the ptplot package in its own directory, we need to tell Java where to find it. As a matter of convention, the Java compiler `javac` and interpreter `java` assume that the value of a variable named `CLASSPATH` contains the information on where packages such as ptplot are stored. This type of variable that controls the environment in which programs run is called an *environmental variable*. Since the programs in the packages have already been compiled into class files, the variable that directs Java to the classes is called `CLASSPATH`.

To get PtPlot to work under Java you need to modify the `CLASSPATH` variable to include the location where PtPlot is stored. Here are some directions on how to do that for various systems:

Windows 95 Open the `autoexec.bat` file (usually `C:\autoexec.bat`) in a text editor such as *Notepad*. At the end of the file, add the line

```
SET CLASSPATH=%CLASSPATH%;C:\ptplot
```

Save the `autoexec.bat` file and restart your computer.

Windows 98/ME Click *Start* and then *Run* under that. Key in the command name `msconfig`, and press *Enter*. The *System Configuration Utility* should appear. Select the tab named *Autoexec.bat* and look for a line that says `SET CLASSPATH`. If you cannot find that line, click *New*, and enter

```
SET CLASSPATH=C:\ptplot
```

If the `SET CLASSPATH` line already exists, select it, choose *Edit*, add a semicolon (;) to the end of the line, and then add

```
C:\ptplot
```

after the semicolon. The semicolon is a separator that tells Java that it should look in more than one location for class files. Make sure that the checkbox by the `CLASSPATH` line is checked, and click *OK*. Answer "Yes" when Windows asks you whether you want to restart the computer.

Windows NT/2000/XP Open the Control Panel (*Start, Settings, Control Panel* in NT/2000, or *Start, Control Panel* in XP). Open the *System* icon (you may need to switch to the *Classic View* in Windows XP). Under the *System Properties* window, select the *Advanced* tab, and choose *Environment Variables*. Two lists should be shown. One contains environment variables just for you, and one contains environment variables for all users on the system. In your personal environment variable list, look for `CLASSPATH`. If you cannot find the `CLASSPATH` variable, click *New*, enter `CLASSPATH` for the variable name, and `C:\ptplot` for the value. If the `CLASSPATH` variable already exists, select it, choose *Edit*, add a semicolon (;) to the end of the current value, and then add

```
C:\ptplot
```

[6] If you use the Windows automatic installer, that means you should install to `C:\ptplot`, rather than `C:\ptolemy\ptplot5.2`, if you want to follow our examples ad verbatim.

after the semicolon. The semicolon is a separator that tells Java that it should look in more than one location for class files. Click *OK* until you get back to the Control Panel, and then restart your machine.

Solaris We assume that you do not have system authority for your computer, and so will install ptplot in your home directory. We suggest that you make a subdirectory called `java_packages` in your home directory and will install PtPlot there. To make that subdirectory:

`> cd`	Change to my home directory.
`> mkdir java_packages`	Create subdirectory in present directory.
`> mkdir java_packages/ptplot`	Create subdirectory in `java_packages`.

This assumes that you will be installing the PtPlot package in `~/java_packages/ptplot`, where the `~` represents your home directory.

Your `CLASSPATH` variable that needs updating is contained in the initiation (init) file file `.cshrc` in your home directory. Because this file name begins with a ".", it is usually hidden from view. Beware, it is easy to mess up your `.cshrc` file and end up in a quagmire. Accordingly, we suggest that you first create a backup copy of your `.cshrc` file, in case anything goes wrong:

`> cp .cshrc .cshrc_bk`	Make a backup, just in case.

Next, open the `.cshrc` file in a text editor. Because this file may contain some very long lines that must be kept intact, if your text editor has an "automatic word wrap" feature, make sure it is turned off. Next look for a line that starts with `setenv CLASSPATH`. If you cannot find that line, add `setenv CLASSPATH ~/java_packages/ptplot` on its own line at the end of your `.cshrc` file. If the `setenv CLASSPATH` line already exists, add a colon (`:`) to the end of the existing line, and then add `~/java_packages/ptplot` after the colon. The colon is a separator, which tells Java that it should look in more than one location for class files. Save your `.cshrc` file, then close and reopen all C shell terminals that you have open (or log off and then back on).

Once you have the `CLASSPATH` variable set, you should make sure that it is working. To check, go to a command prompt in a shell and enter

`> echo %CLASSPATH%`	Windows check.
`> echo $CLASSPATH`	Unix check.

The complete value for the `CLASSPATH` variable should be printed to the screen, for example:

```
/home/rubin/java/classes:/home/rubin:/home/rubin/mpiJava:/usr/local/mpiJava/lib/classes:/home/rubin/java_packages:/home/rubin/java_packages/ptplot:.
```

If your changes do not show up, carefully follow the directions again or ask for help.

At this point, you are ready to try out PtPlot. Get the file `EasyPtPlot.java` containing a sample plot program from the disk or the Web and enter

`> javac EasyPtPlot.java`	Compile sample plot program.
`> java EasyPtPlot`	Run sample plot program.

If the PtPlot package was installed correctly, you should get a nice graph on your screen. If this does not work, ask for help.

Add stuff here