Chapter 1: Getting Started

Goals:

- To learn the basics of using a Macintosh
- To use CodeWarrior to compile and run prewritten Java programs
- To learn some basics of Java programming—logical and math operations, drawing on applets
- To answer questions about the geometry of particle motion in enclosed spaces

In this first chapter we will lead you step-by-step through the things we want you to do. Future chapters will be much less explicit in describing the precise way you should do things.

A. Preliminaries. First, find a Macintosh in the MacLab. If you are unfamiliar with the Macintosh, please talk to your TA or to the MacLab staff, and they will show you how to get started with basic operations such as creating and opening folders and files.

Next, if you are not familiar with a web browser, you should check out Netscape. To find it, double click on "Ryerson Maclab" and "Netscape x.x" (where x.x is the current version number). In addition to checking out some useful web sites (the U of C home page is at http://www.uchicago.edu), you might want to go to some of the sites running Java applets that are in the Gamelan directory at http://www.developer.com/directories/pages/dir.java.html.

Later we will run a Java applet using the Web browser. However, first we will run simple Java applets using CodeWarrior.

B. Getting Started. To start on chapter 1, open (i.e. double-click on) the icon called "MacLab Resources", then sequentially open the "Courses," "Winter 1999," and "Phys 251/CS 279/Math 292" folders.

* * *

This folder contains four subfolders, called "Class Notes," "Drop Box," "Programs," and "Solutions."

The "Class Notes" folder contains these Class Notes. You will be putting the programs you write for your projects in the "Drop Box," and "Solutions" will contain the solutions to the exercises, problems, quizzes, and required projects. Right now, we want you to open (i.e. double-click on) the "Programs" folder.

To start Chapter 1, copy the folder called "Chapter_1" to the desktop by dragging the "Chapter_1" folder icon to the desktop. (Eventually you may want to copy this folder onto a zip disk.) Now double click on the folder "Explore" in the (desktop) "Chapter_1" folder, and finally on the file "Explore.mcp." The computer will start running CodeWarrior, then after a bit you will be in the programming environment, working on a project represented by a window with label Explore.mcp. The window should contain, under “Sources,” a list of files in the project: “Explore.html” and “Explore.java.” If it doesn't, click on the triangle on the left next to “Sources” to see this list of files. “Explore.java” is the name of a Java program which is provided for you. Open this file by double-clicking on the word “Explore.java” in the project window.

This applet performs a set of very simple tasks. It takes two numbers and multiplies them together, and then displays the two numbers and their product on the applet. The program knows how to write onto the applet because it actually defines an "extension" of the class "Applet." The people who wrote Java have written code that tells members of the Applet class how to display text, among other things. After some comments (first 3 lines, each preceded by //), the next two lines tell Java that the program will be using some of this code. The following line of the program makes sure that Java knows that Explore is in the Applet class. To get the program into operation, open the **Project** menu and select the command **Run**.[1] Codewarrior compiles the program (i.e. the Java compiler translates the program into "bytecodes," a computer language specially designed so the program can be run over the internet) and the "class libraries" (the code written by others that tells the computer how to

[1]If you don’t see a Run command, it is because the debugger is ‘enabled.’ Select **Disable Debugger** from the **Project** menu, after which the Run command should appear on the **Project** menu.

* * *

display things on the applet viewer, among other things) are loaded. Then the application Metrowerks Java opens automatically and executes the instructions one at a time (it "interprets" the bytecode instructions), opens the applet viewer, and finally displays the applet. Some other windows will also open, which sometimes contain useful information (but not now). You can close them after running the program. Our program is shown below:

```
// Anything to the right of double slashes like these is a comment and is not read by the
// computer

// Explore.java   Our first program, which multiplies numbers
import java.applet.Applet;  // import Applet class
import java.awt.Graphics;  // import Graphics class (enables writing to applet viewer)
public class Explore extends Applet  //  defines the name of our Applet
{
// in the next lines, this program defines some variables
    int i;              // defines an integer variable
    int j, k;           // defines two more
    double x, y;        // defines two real number variables (not used)

// every applet automatically calls the methods init and paint, which we will modify
// so that they perform the operations that we want.  In Java, init and paint do nothing
// unless we specify otherwise.
    public void init()
    {
        i = 7;              // gives i the value of 7
        j = 5;              // gives j the value of 5
        k= i * j;           // computes i times j, puts the result in k
    }                   // end of init

    public void paint ( Graphics g )  // display the result on the Applet
    {
        int xpos = 25;          // x coordinate of output string on applet, in pixels
        int ypos = 25;          // y coordinate of output string, from top of applet

        g.drawString( Integer.toString(i), xpos, ypos);     // convert i to string and
                                                            // put onto Applet
```

* * *

```
        ypos = 40;
        g.drawString( Integer.toString(j), xpos, ypos);     // put value of j onto Applet
        ypos = 55;
        g.drawString( Integer.toString(k), xpos, ypos);     // put value of k onto Applet

    }           // end of paint
}               // end of definition of the applet Explore
```

Program 'Explore.' Our First Program. It takes two numbers and multiplies them together and then displays the factors and the product on the Applet.

Run the program 'Explore' to see that it operates correctly.

To those unfamiliar with computer programming, this may seem like quite a long program, since all it does is determine that 5×7=35. Look over it briefly. You'll see that over half of the text is in comments (text preceded by //), which really aren't part of the program at all (the compiler ignores them). Also, notice that the majority of the commands in the program are devoted to printing the output on the applet (everything from the command `public void paint` on). Another significant portion is devoted to organizing the computer's memory in preparation for doing the computation (the definition commands). The actual computation boils down to the single command `k= i * j`.

In reality, our program did not run on its own; we had to do an additional step so that our Java applet would run. Java applets are designed to be run using web browsers like Netscape and Internet Explorer, which can only recognize Java if the program is embedded in a document written in HTML (HyperText Markup Language). We have arranged this by putting an HTML file called "Explore.html" into the project "Explore.mcp." The file "Explore.html" contains the lines:

```
<title>Explore</title>
<hr>
<applet archive="AppletClasses.jar" code="Explore.class" width=200 height=200>
</applet>
<hr>
```

* * *

You can view this file by double-clicking on its name in the “Explore.mcp” project window. This HTML file tells the web browser to look in the archive "AppletClasses.jar" for the applet “Explore” (CodeWarrior sets up this archive automatically, as you'll see shortly) and fixes the size of the applet display at 200 x 200 pixels. Java has a hard and fast rule that the Java code for the applet “Explore” must be placed in the file “Explore.java.” (The html file need not be called "Explore.html.") The Java compiler (Codewarrior) puts the bytecodes it produces into the file “Explore.class” (which is compressed and put into the archive "AppletClasses.jar"), and the web browser runs the program by interpreting those bytecodes. We used the “appletviewer” that is built into CodeWarrior to run our applet, but you can also use a regular web browser such as Netscape. You can view the applet by opening Netscape (if you need to free up some memory, then "Quit" CodeWarrior either by choosing **Quit** from the **File** menu or else by ⌘-Q, simultaneously hitting ⌘ and q) and selecting **Open File** from the **File** menu and then using the dialog box to select “Explore.html.”

Now let’s alter our original program and see what happens. If you are not running CodeWarrior, then start it up, either by clicking the CodeWarrior icon, choosing it on the Apple menu, or by double clicking on our old project “Explore.mcp.” To make a change, open the file called “Explore.java” and make a change in the text. A good choice would be to change the value of ‘i’ in the line `i = 7`, perhaps to `i = 12`. Be sure to save and close the window after making the change. Then open the **Project** menu and select the command **Run**, as before. The program should execute and indicate that 12×5=60.

Having learned how to edit programs, it is now time to systematically explore some of the basic features of Java.

Variables are associated with locations in memory. When you declare a variable you tell the computer how much space it needs to reserve to store the variable and also tell the computer what type of data will be represented in that space. There are many different types of variables in Java. In our program, i, j, and k have been defined to be integers, while x and y have been defined to be real numbers with 'double' precision. In the following exercises you will explore what Java does with integer variables.

* * *

To complete the following exercises, you will need to make more modifications to "Explore.java." We recommend saving the altered Java and html files with different names (don't forget to change the name of the Applet!).[2] CodeWarrior will automatically put the new file in the project and remove the file it superseded from the project. The old file hasn't been erased, though, and you can **Add** (in the **Project** menu) it back to the project if you wish. If you change the name of the applet you will need to alter the html file so that all three occurrences of "Explore" are replaced with the new name. Alternatively, you can choose not to change the name of the Applet, but old versions of the program will be lost as you introduce new alterations.

Exercise 1.1. Elementary integer operations.

1.1.1 The basic integer operations are + - * / and %. Change "Explore.java" to use the other integer operations besides multiplication. Figure out what each operation means. All operations should be quite obvious except perhaps for % (which is called "modulus"). Make sure that you know what happens with negative numbers.

1.1.2 What happens if you have several such operations in a row, as 1 + 3*4; or 2*3 + 5 / 3? You can organize expressions like these by putting in parentheses, as for example in:

```
k = (4*3) + (4 / 2);
```

We usually put in parentheses to avoid having to remember the rules related to what happens without them.

1.1.3 What happens if you do an operation in which the same variable appears on both sides of the = sign? For example, see what k ends up being if, after the statement "`k = i*j;`" you add the statement:

```
k = k + 3;
```

1.1.4 There is also a function for taking the absolute value of a number: if i is a variable, `Math.abs(i)` is its absolute value, so that

```
Math.abs(-1) = 1
Math.abs(1) = 1
Math.abs(0) = 0.
```

[2] Alternatively, you can save the old version in a file with a different name (say, Explore.old.java).

* * *

Try this function out in the program “Explore.java.”

Exercise 1.2. Errors. There is nothing awful about making errors. Let us make a few. In each exercise start from the original version of “Explore.java.”

1.2.1 Insert a line in your program to set the variable ‘m’ equal to zero. What happens? The compiler will write a message in the **Errors & Warnings** window that will help you figure out how to fix this error (it even has a red arrow pointing to the culprit line of the program).

1.2.2 Now insert a line to set i equal to 1.1. What happens?

1.2.3 Set i equal to zero and define j by

```
j=1 / i;
```

What happens? If the computer locks up, you will need to force exit from ‘‘Metrowerks Java’’ by simultaneously hitting the three keys ⌘-option-esc. You may find information about what went wrong in the **Java Output** window.

1.2.4 One more error: Set j equal to 100000 and then insert a line

```
j = 600000*j;
```

Now run the program. Is your value for j what you expected? What happened?

Java also supports two other data types similar to integers: long and short. Variables of these types are declared and used in the same way as variables of type int. However, java assumes that any non-decimal number (such as 5) is of type int unless another type is explicitly specified in the following manner:

```
long y;
short z;
y = (long) 5l;
z = (short) 5;
```

The number 5 is cast as a long integer in the third line and as a short integer in the fourth line. Notice that long integers need to end in ‘l’, but short integers do not have to end in ‘s.’ To convert these data types to strings, use Long.toString(y) and Short.toString(z).

* * *

Problem 1.1. What is the largest possible value of an integer variable? The smallest, i.e. the most negative? What are the largest and smallest possible values of a "long" integer? Determine your answers to 5 significant figures.

In addition to integers, Java has the capability to manipulate real (decimal) numbers as well. Note that in the applet “Explore” there are two real number variables declared (but not used) in this applet called double variables. Edit the program “Explore” to use the variables x and y. To draw the output on the screen, you will need to use the command Double.toString(x).

Exercise 1.3. Elementary real operations. Modify the applet “Explore” to answer the following questions.

1.3.1 Are all of the following valid expressions for a real variable: -3.4, -7.01e11, .011, 1, 1e5? What does the ‘e’ notation mean?

1.3.2 Can you store the result of an integer computation in a real variable? How?

1.3.3 What happens if you have several such operations in a row, such as 1.0 + 3.0*4.0 or 2*3.0 + 5.0?

C. Loops. In doing a computation, you often want the computer to do the same operation (or set of operations) over and over again. The programming construction which allows you to execute the same commands repetitively is called a loop. In Java you can define a loop in several ways. One way is with a for statement.

Look at the program entitled “Fibonacci.java” below. This applet constructs the series 1,1,2,3,5,8, ... Note that each number in this series is the sum of the two preceding numbers. The algorithm that we will use to construct this series is as follows. Let ‘i’ be the first number in the series, ‘j’ the second, and ‘k’ the third. We compute ‘k’ by summing ‘i’ and ‘j.’ To create the fourth term in the series we let ‘i’ be the second number in the series, ‘j’ the third, and ‘k’ (created by summing ‘i’ and ‘j’ as before) the fourth. We step through this process over and over again. Let ‘n’ be our "step counter." At each step in this process we do two things. First we shift ‘i’ and ‘j’ to be the ‘n’ and ‘n+1’ terms in the series. Second, we compute ‘k’ by

summing ‘i’ and ‘j’. The for{} construction sets up a loop which will allow us to run through our process a specified number of times.

```
//  Fibonacci.java
//  An applet that writes out a Fibonacci series
import java.awt.*;
import java.applet.Applet;

public class Fibonacci extends Applet {    // defines the name Fibonacci for our Applet
    int i, j, k;
    int n, nsteps, ypos;

    public void init ()     // initialize our variables
    {
        nsteps = 8;     // we start with two numbers in the series
                        // nsteps gives us the number of "new" Fibonacci numbers
                        // which will be created
                        // The series length will then be nsteps + 2
        i = 1;          // i is initially the first term in the series
        j = 1;          // j is initially the second term
        ypos = 25;// vertical coordinate of first line of text (in pixels, from top)
    }           // end of init

    public void paint (Graphics g)    // use paint method to display on applet
    {
        for (n = 1; n <= nsteps; n=n+1)
        {                       // the next four lines will be executed nsteps times
            k = i + j;          // Sum the last two terms in the series to get the next
            g.drawString(Integer.toString(n) + ",  " +
                    Integer.toString(k), 25, ypos);
                            // Write the step number and k onto the applet
                            //  In this statement the '+' concatenates strings, so that
                            // n is written and then k
                            //
                            // The next two lines shift i and j
                            // to be the last two terms in the series
            i = j;          // the new value of i is that of j
            j = k;          // the new value of j is that of k
```

* * *

```
            ypos = ypos + 15;      // put next pair lower down on the applet
        }                          // end of the for loop
    }                              // end of paint
}                                  // end of applet
```

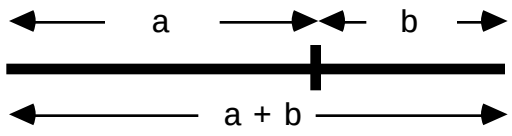
Program Fibonacci.java. This program computes a Fibonacci series of length n. Notice that the brackets serve to block off a portion of the program which can be treated as a whole. In this case, the brackets tell the computer to treat the four lines in the for loop as if they were a single statement. Brackets are also required at the beginning and end of a program. Also note that while the '+' sign denotes addition when applied to numbers, when used with strings (as in the line with "g.drawString"), it means "string concatenation" (writing the first string and then the second).

Create a Codewarrior project for “Fibonacci.” First, choose **New Project** under the **File** menu. A box will open asking you to pick the Project Stationery. First, click on the arrow next to Java , and then choose the "Java Applet" stationery. The stationery dialog box also gives you the option of automatically creating a new folder for the project, which we recommend that you do. Another dialog box will ask you to enter a name for the project and choose its location. We suggest that you name the new project “Fibonacci.mcp” (mcp is the file suffix for CodeWarrior project files) and select the folder “Chapter_1” (it is the folder that already contains Explore). If you selected the "create folder" option, CodeWarrior will create a folder “Fibonacci” in the folder “Chapter_1.”

When the project window appears, click the arrow next to "Sources" to reveal the default files “TrivialApplet.html” and “TrivialApplet.java.” These are the files we need to change to create and view our applet. First, double-click on “TrivialApplet.html” and edit it by replacing all occurrences of ‘TrivialApplet’ with ‘Fibonacci.’ Then save the file (you are free to change the name of the html file when you save it, but it's not necessary). Then open “TrivialApplet.java” and use the editor to replace the existing text with the text of the program “Fibonacci.java” given above. You can cut and paste the text of “Fibonacci.java” from the SimpleText file "Fibonacci.java text" that we have placed in the folder "Text Files," which is inside the

"Chapter_1" folder. To cut and paste, double-click on the "Fibonacci.java text" icon (which causes SimpleText to open it), select all the text in the file (either by holding down the mouse as you drag it down through the file's contents, or by using ⌘-A), **Copy** the text onto the clipboard (using ⌘-C), and then **Paste** it into the program (using ⌘-V). It is also possible to Select All, Copy, and Paste using the menus at the top of the screen. Finally use the **Save As**... command in the **File** menu to save the file as "Fibonacci.java." This time, the name "Fibonacci.java" is not optional; there is a strict rule that a Java file must have the same name as the public class it contains (determined by the "public class Fibonacci extends Applet" statement in the program). Now, you can compile and run the program by choosing **Run** from the **Project** menu.

The Fibonacci series has many interesting properties. Let us define f_n as the n^{th} term in the Fibonacci series and let $g_n = f_n/f_{n-1}$. As n becomes large, g_n approaches a number φ. It turns out that φ is a well known number called "the golden section" or "golden mean". The Ancient Greeks were familiar with φ in the following context. Consider the line segment shown below.



If the ratio a/b is the same as the ratio (a+b)/a, then $a/b = \varphi$. The Greeks thought that rectangles with sides a and b in this ratio were the most aesthetically pleasing. The sides of the Parthenon in Athens and the proportions of many ancient Greek vases and sculptures are based on the golden mean.

Problem 1.2. Show analytically that the golden mean is $(1+\sqrt{5})/2$. Then modify the Fibonacci program to compute and write out the ratio of each term to its predecessor. (Don't forget that you want this ratio to be a float or double variable.) Show analytically that g_n approaches the golden mean as n becomes large.

* * *

Here is another small note on for loops. In using such a loop, the 'counting' variable does not have to start at one. For instance, suppose you wanted to write to the screen a sequence of integers from 2 to 8, along with their squares. Assuming the variables i and ypos have been declared, the following lines of Java will do precisely that:

```
for ( i = 2; i <= 8; i = i+1) {
    g.drawString(Integer.toString(i) + ",  " +
            Integer.toString(i*i), 25, ypos);
                    // Write the step number and its square to the applet
    ypos = ypos + 15;      // put next pair lower down on the applet
}
```

D. Drawing on the Screen. Graphics can be displayed on an applet. The applet has an invisible coordinate system built into it with (x=0, y=0) being the upper left corner and the other corners having the coordinates determined by the width and height in the "applet code" statement in the html file that calls the applet.

The basic method one needs to start out with graphics is the Graphics method drawLine, which has the following specifications:

```
public abstract void drawLine(          // Graphics class
    int x1,    // x coordinate of first point, in pixels
    int y1,    // y coordinate of first point
    int x2,    // x coordinate of second point
    int y2 )   // y coordinate of second point

    Draws a line, using the current color, between the point
(x1, y1) and the point (x2, y2).
```

Ignore the keywords public, abstract, and void for now. We will discuss their meaning in the next chapter.

A program which uses drawLine is "Drawing," shown below. Once again, create a new project by selecting **New Project** from the **File** menu, modify the files "TrivialApplet.html" and "TrivialApplet.java," and change the name of "TrivialApplet.java" to "Drawing.java." Don't forget to change the Java Applet Settings .

* * *

The text of “Drawing.java” is in the Text Files folder. But we would like you to type the program in yourself, so you can see how CodeWarrior's editor works. Notice how the editor color-codes things to help you remember to close braces and put in the semicolons. Be sure to be aware of capital and small letters because Java is case-sensitive; "a" and "A" are not the same. After you have typed in the program, **Save** it as “Drawing.java,” and then **Run** it.

```
//  Drawing.java       draws a line on the applet
import java.awt.*;
import java.applet.Applet;
public class Drawing extends Applet {
    int iold, jold, inew, jnew,k;
    public void paint ( Graphics g )
    {
    // draw a line from (5, 10) to (60, 150)
        g.drawLine ( 5, 10, 60, 150 );
    }    // end of paint
}        // end of applet
```

Run the program and note the effect of adding more lines, say:

```
g.drawLine (100, 100, 150, 150);
g.drawLine (150, 150, 150, 100);
```

Java does not give us a direct way to plot a single point. Instead, we draw a line starting at the position where we want to plot the point and ending at the same point. Thus if we want to add a command to our paint method to plot a dot at the position (100,100), we would do so with the line:

```
g.drawLine(100, 100, 100, 100);
```

Or, if one wants a larger shape, one could plot a small filled square three pixels on a side centered at position (175,175) using:

```
g.fillRect(174,174,2,2);
```

The specifications for fillRect are:

* * *

```
public abstract void fillRect(        // Graphics class
   int x,       // x coordinate of first point, in pixels
   int y,       // y coordinate of first point
   int width,   // width of rectangle
   int height ) // height of rectangle
       Draws a filled rectangle from (x, y) to
      (x+width-1, y+height-1) in the current color.
```

One can also draw other shapes using methods such as drawArc, drawOval, etc., which are described in Chapter 13 of Beginning Java.

Exercise 1.4. We can now simulate the effect of a 'ball' bouncing off the top and the bottom of a rectangular region. More precisely, we can show the path of such a ball as it moves from top to bottom, bottom to top, and every time advances in the x direction by the same amount. Let this amount be k and let us modify the paint method in “Drawing.java.” First start out by issuing instructions which set initial values for i (the x-coordinate of our ball), j (its y-coordinate), and k (the step forward).

```
iold = 0;
jold= 0;
k = 25;
```

Next, do the bounces.

```
jnew = 200-jold;
inew= iold+k;
g.drawLine(iold,jold, inew,jnew);
```

The first line moves the y-coordinate up and back from top to bottom or from bottom to top. The next advances the x-coordinate. The last step draws the line. Insert a loop in the program to advance the coordinates and draw lines ‘nlines’ times. Call your counting variable ‘count.’ Make sure that each new line starts from where the last one ended. The result of the program is that a little picture is drawn out.

Now we have used computer graphics to display the answer to a (tiny) mechanics problem.

* * *

One final point about graphics: You could want to print out on a printer something displayed on an applet. The easiest way to do this is to take a "snapshot" of the whole screen, by simultaneously pressing ⌘-Shift-3. This will produce a file called "Picture 1" (or "Picture 2," "Picture 3," etc.) on the startup disk (if you don't know which disk is your startup disk, use **Find File** to find "Picture 1"). You can open the file using SimpleText, which happens automatically when you double-click on the icon of "Picture 1," and then select the **Print** command on the **File** menu in the SimpleText application. (Directions for saving Java applet output on Macintosh, Windows and Unix systems are at http://gdbwww.gdb.org/gdb/mapPrint.html.) You can use MacPaint, Photoshop, or Canvas to alter and print the file, if you wish.

E. More About Loops. A for(){} loop is not the only way to construct a loop. The program "FirstOrbit" below uses a second technique, the while(){} structure. Using this structure is very simple. Everything that is between the brackets will be executed, until a condition that you specify is no longer satisfied. We will just do the simplest thing: terminate the loop when we get to the edge of the screen, i.e. when the variable iold is greater than 200. (For a more complete discussion of loops see Chapter 3 of Beginning Java.

```
//  FirstOrbit.java
//  an orbit calculation for particles bouncing off the top
//  and bottom of a rectangle.
import java.awt.*;
import java.applet.Applet;
public class FirstOrbit extends Applet {
                              // declare variables
    int iold, jold, inew, jnew;     // x and y coordinates of particle at beginning
                                    //  and end of a flight between bounces
    int di;                         // advance of i in each bounce

    public void init ( )
    {
    //  set initial values
        iold = 0;
        jold = 0;
```

* * *

```
        di = 22;
    }

    public void paint ( Graphics g )
    {
        //  set up a loop to run through the same steps many times
        while (iold <= 200) {      // terminate loop when x coordinate exceeds 200
            inew = iold + di;       // advance i
            jnew = 200 - jold;          // go from top to bottom
            g.drawLine ( iold, jold, inew, jnew );     // draw line between points
                                // start next line at endpoint of the line just drawn
            iold = inew;
            jold = jnew;
        }    // end of while
    }    // end of paint
}        // end of applet
```

Program FirstOrbit. An orbit calculation for a ball bouncing off two walls. There are many bounces but the program ends when it should. Notice how while is used.

Exercise 1.5. Add "FirstOrbit.java" and a suitable html file that calls it to the project "Explore.mcp." Use ⌘-R to make the program run.

A note on nested loops. There is nothing wrong with having a loop inside a loop. However, be careful that the 'counting' variables are different from each other! As an exercise, can you determine what the output from the following lines of code would be?

* * *

```
i = 0;
ypos = 25;
while (i < 6) {
    j = 1;
    while (j < 3) {
        g.drawString(Integer.toString( i *j ), 25, ypos);
        j = j + 1;
        ypos = ypos + 25;
    }
    i = i + 2;
}
```

The answer is that this bit of program writes out the numbers: 0, 0, 2, 4, 4, 8. Please go through the loops by hand to make sure you understand how nested loops work.

Java has a shorthand notation that often comes in handy when you are incrementing variables. The expression "`j = j+2`" is equivalent to "`j += 2`." There is an even shorter notation "`j++`" and "`j--`" that increments or decrements the variable by one. So our bit of code could also be written:

```
i = 0;
ypos = 25;
while (i < 6) {
    j = 1;
    while (j < 3) {
        g.drawString(Integer.toString( i *j ), 25, ypos);
        j++;
        ypos += 25;
    }
    i += 2;
}
```

Of course, `for(){}` loops and `while(){}` loops can be mingled and nested. For example, in the example with nested `whiles` above, we could have used a `for` loop for the j loop:

* * *

```
i = 0;
while (i < 6) {
    for (j = 1; j <= 2; j++) {
        g.drawString(Integer.toString( i *j ), 25, ypos)
        ypos += 25;
    }
    i += 2;
}
```

Make sure you understand how this works.

Exercise 1.6. Now modify FirstOrbit.java in the following way. Replace the line that says "`while( i <= 200)`" with "`while (i != 201)`". Can you predict what will happen? Try it and see. Don't panic at what happens; instead, read the next paragraph.

Now, there is a bit of a problem. The program keeps on going, and going. It never stops. What to do? There are several things you can try. The first is to press ⌘-. (command-period). If that doesn't work, try ⌘-control-/ (command-control-slash). If that fails, then press ⌘-option-esc. Although ⌘-. and ⌘-control-/ may not always work, ⌘-option-esc will definitely stop the program. (⌘-option-esc will emergency-exit you from any Macintosh application, though sometimes this exit causes the machine to crash. So only use it in emergencies.) So now we have a program which runs perfectly well, but forever. Clearly, this is an undesirable situation, and one that you should take care to prevent in future programs.

Incidentally, you may have noticed the **Enable Debug** command (just above **Run**) on CodeWarrior's **Project** menu. You can run the program inside the debugger by choosing **Enable Debug** and then **Debug** (which replaces **Run** when the debugger is enabled) from the **Project** menu. The debugger can be very useful for tracking down and fixing errors because it enables you to step through the program to see exactly what it is doing. However, it also makes the program run much, much slower. So **Enable Debug** when you are debugging a program, and **Disable Debug** when you run a long program where speed is essential.

F. Logical Operations. In “FirstOrbit.java,” we tested to see whether to stay in the `while` loop with the condition `while (i <= 200)`,

which seems very natural. We want now to discuss the full range of logical statements which can be applied to numbers, such as "i > 200". The following symbols:

```
==  >  <
```

define a set of logical operations which can be applied to integers (or reals) and have as their outcome the values true or false .

Exercise 1.7.

1.7.1 To see logical (or "Boolean") operations at work reopen the file containing the program “Explore.java.” To do this select Open under the File Menu and then click on “Explore.java.” Assign some values to ‘i’ and ‘j.’ Inside the paint method, type the following (and then increment ypos):

```
g.drawString( new Boolean(5==3).toString(), 25, ypos);
g.drawString( new Boolean(2>1).toString(), 25, ypos);
g.drawString( new Boolean(i>j).toString(), 25, ypos);
```

Run the program to see whether the computer thinks these statements are true or false.

1.7.2 There are more logical operators. They are:

```
>=  <=  !=
```

What do they mean?

Notice the difference between the symbols = and == . Can you express the difference in words?

Logical operations have many more uses than simply stopping repeat loops. They can also be used to conditionally execute a command or sequence of commands, using the `if(){}` construction. If one has a piece of computer program which looks like

```
if (condition)
{
    statement1;
}
statement2;
```

then the computer first checks to see whether the thing called condition is true or false. If condition is true the computer performs statement1 and then statement2. If condition is false the computer

skips statement1 and goes on to statement2. This kind of approach allows the computer to make choices and do good things (like not getting stuck in infinite loops). As always in Java, statement1 (and statement2) can be one of two things: it can be a single command (like drawLine(i1, j1, i2, j2);), or it can be a set of commands sandwiched between a set of brackets. This is a general rule: any place you can have a single command, you can instead have several commands delineated by brackets.

It is important to note that there is no semicolon immediately after if(condition). If you put one in by mistake, then Java will think that the conditional applies only to the stuff before the semicolon, so any statements in brackets following the semicolon will always be executed. This is a common way to generate accidentally an infinite loop.

G. Supplied Functions and Constants. Java comes supplied with many useful functions and supplied constants. Among them is Math.PI. To see what Math.PI is (it's not hard to guess) include the statement

```
g.drawString(Double.toString(Math.PI), xpos, ypos);
```

in the paint method of “Explore.java.” Then incorporate the following lines into the paint method of the program to see the effect of such operations as:

```
x = Math.sin (Math.PI/2);
y = Math.cos ( Math.PI * x);
g.drawString("x="+x+",  y="+y, xpos, ypos);
```

Java also provides us with the Math methods Math.log(x) (natural log), Math.exp(x) (which returns e^x), and Math.pow(x,y), which computes x^y. Another function, especially useful for plotting real variables, is Math.round(). (Note: When operating on a float, Math.round returns an int, while operating on a double returns a long).

H. Writing. Here is a little more detail about displaying things on applets. In all our applets so far we have done this inside the paint method, which is always called with a Graphics object that we have been calling g. This Graphics object in turn has access to several methods that write onto the applet. For example, to put a string on the applet one uses drawString; to draw a line one uses drawLine. There is no separate command for writing integer or real numbers, so we

first convert them to strings and then display them using drawString. You cannot use drawString or drawLine unless you have access to the Graphics object.

Reopen the project “Explore.mcp.” You will be using “Explore.java” to practice writing.

Exercise 1.8. Writing. Inside the paint method type in the following statements.

```
g.drawString("The answer is " + (3+5), 25, 25);
g.drawString("The answer is " + 3 + 5,  25, 40);
```

What is the output? Notice that + can mean both 'string concatenation' and 'addition', depending on whether it is acting on strings or on numbers. Also notice that numbers can get converted automatically to strings (this is called automatic casting). Finally, notice that you can tell the computer to perform operations in a particular order by using parentheses.

Now try:

```
g.drawString(3+5, 25, 55);
```

This leads to an error because drawString sees the integer 8 when it is expecting a string and it complains. Two ways to fix this are:

```
g.drawString(Integer.toString(3)+5, 25, 70);
g.drawString(3+5 +" ", 25, 85);
```

The second one works because " " is a string, and when Java combines a number with a string, it automatically casts the result as a string.

Problem 1.3. Create an applet that prints out a string and underlines it.

I. The Graphical User Interface. Often we want a program to be able to communicate with the person running it, not only to display the result of calculations but also to follow user instructions. Java is designed so that it is straightforward to write a program that is easy for the user to control using the keyboard and mouse. This functionality is called the "graphical user interface" (GUI).

* * *

The key to the GUI is that Java has a built-in ability to recognize "events," such as pressing a key and moving the mouse. In addition, Java has built-in methods that enable an applet to "handle" these events. One key event-handling method is called "actionPerformed."

To see how event-handling works, let's go back to Explore and make it a bit fancier. For example, particularly if you didn't know how to multiply very well, you might want to change the values of i and j and see how that affected the result. You could do this by changing the assignment statements in "Explore.java" and recompiling each time, but it is much more convenient to run the program just once and change the numbers you want to multiply as the applet runs. So we have modified the program so that the user types integers onto text fields on the applet and then presses a button to display the sum on the applet. The people at Sun have written instructions that enable applets to make Labels, TextFields and Buttons. We can declare our instances of these objects and add them to our applet. We instruct the button to listen for an "action" (which will be when the mouse is clicked on it) and notify the applet every time the action occurs. (Sun's instructions are for this are in the java.awt and java.awt.event packages that are imported in the beginning of the program.) Because this new program "Exploremore" prints out the new product every time the user generates an event by hitting the button, we no longer calculate the product in the method init (which would only calculate the product once), but rather in the method actionPerformed, which is called every time the button is pressed.

```
// Exploremore.java
// A second program which multiplies numbers
import java.applet.Applet;  // import Applet class
import java.awt.*;          //  import the java.awt package (enables use of graphical
                            //  user interface components)
import java.awt.event.*;    //  import package which enables events to be recognized

public class Exploremore extends Applet    //  defines the name of our Applet
                implements ActionListener  //  specifies that applet will be listening
                                           //  for user-initiated actions (mouse clicks)
{
// in the next lines, this program defines some variables
```

* * *

```
    int num1;                   // defines an integer variable
    int num2, product;          // defines two more
    Label prompt1, prompt2;     // prompt user for two inputs
    Label label1, label2;       // labels for input text fields
    TextField input1, input2;   // store input values here
    Label resultLabel;          // label for result text field
    TextField result;           // result text field
    Button productButton;       // button to get product on result field

// set up the graphical user interface components and initialize variables
    public void init()
    {
        prompt1 = new Label( "Enter integers in text fields" );
        prompt2 = new Label( "Press button for product     " );
        label1 = new Label ("First integer" );
        input1 = new TextField( 10 );
        label2 = new Label ("Second integer");
        input2 = new TextField( 10 );
        resultLabel = new Label("Product:");
        result = new TextField ( 10 );
        result.setEditable( false );        // prevents user from changing result field
        productButton = new Button( "Click for product" );
        productButton.addActionListener( this );     // tells button to listen for mouse
                                                     // clicks
        add (prompt1 );
        add (prompt2 );                 // put prompts on applet
        add( label1 );
        add( input1 );                  // put first input on applet
        add( label2 );
        add( input2 );                  // put second input on applet
        add( resultLabel );
        add( result );                  // put result on applet
        add( productButton );           // put button on applet

        product = 0;                    // set product to 0
    }                                   // end of init

    //  when button is pressed (action is performed),
```

* * *

```
    // get the integers from the input text fields and then display product
    public void actionPerformed( ActionEvent e )
    {
        num1 = Integer.parseInt (input1.getText() );      // get first number
        num2 = Integer.parseInt (input2.getText() );      // get second number
        product = num1 * num2;                            // calculate product
        result.setText( Integer.toString( product ) );    // display product in
                                                          // result text field
    }                   // end of actionPerformed method
}                       // end of the applet
```

Program Exploremore. A modification of Explore. Each time the button is pushed, it takes two numbers input by the user, multiplies them together, and then displays the multiplicands and the product in the "result" TextField.

Create the Java project for “Exploremore.” The text of the program “Exploremore.java” is in the file "Exploremore.java text" that is in the folder "Text Files" inside the "Chapter_1" folder. Again, the name “Exploremore.java” is not optional; it is fixed by the "public class Exploremore extends Applet" statement in the program. Also remember that you need to make a suitable html file that calls “Exploremore.” Compile and run the program.

Now we will add a GUI to our orbit program so that the horizontal increment di can be changed "on the fly." We'd like to be able to draw the orbits for different values of di, so we would like to call paint every time the button is pressed. In Java you cannot call paint directly. Instead you must call repaint(). The method repaint() then calls the method update, which erases the applet and then calls paint.

Although we want to erase the old orbit, we don't want to erase the whole applet every time we change di (certainly we want to keep the button!). So we have overridden update so that it only calls paint and no longer erases the applet. We erase the old orbit using the Graphics method clearRect.

The new orbit program is in the project “SecondOrbit.mcp” in the folder “SecondOrbit.” The file “SecondOrbit.html” has an applet code

* * *

statement with large height so that there is enough room for a text field, a button, and a rectangle in which the orbits are drawn. Here is the program:

```
// SecondOrbit.java
// an orbit calculation for particles bouncing off the top and bottom of a rectangle
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*
public class SecondOrbit extends Applet     // defines name of applet
             implements ActionListener      //  specifies that applet will be listening for
                                            //  user-initiated actions (mouse clicks)

    final int HEIGHT = 200;         // The height of our rectangle
    final int WIDTH = 200;          // The width of our rectangle
    final int YOFFSET = 75;         // y starts at 75 to leave room for text and button
    final int XOFFSET = 50;         // x starts at 50 so rectangle is centered on applet
                                    //  the values of final variables are set when they
                                    // are declared, and can't be changed afterwards

    int iold, jold;                 // old x and y coordinates of our particle
    int inew, jnew;                 // new x and y coordinates of our particle
    int di;                         // advance in i in each bounce
    Label prompt;                   // prompt user to input di
    TextField input1;               // store input value here
    Button startButton;             // button user presses to start plotting the new orbit

// set up the graphical user interface components and initialize variables
    public void init()
    {
        prompt = new Label( "please enter advance in x-coordinate per step" );
        input1 = new TextField( 10 );
        startButton = new Button( "Click to start" );
        startButton.addActionListener( this );
        add (prompt );          // put prompt on applet
        add( input1 );          // put input TextField on applet
        add( startButton );     // put start button on applet
```

* * *

```
    di = 0;              // set horizontal increment to 0 initially
}                        // end of init

    public void actionPerformed( ActionEvent e )
{
    di = Integer.parseInt (input1.getText() );   // get di from TextField
    if (di <= 0)         // check if di is an allowed value
    {
        showStatus(
            "Please choose a step-size di greater than 0 and press start button");
                                // shows string at bottom of applet
    }
    else {       //  if di is OK, then call the paint method
        showStatus("di="+di);    // shows string at bottom of applet
        repaint();
    }
}                               //  end of actionPerformed method

// override Component class update
// do not clear background, only call paint
public void update( Graphics g )
{
    paint( g );
}

public void paint( Graphics g )
{
    g.clearRect(XOFFSET, YOFFSET, WIDTH, HEIGHT);  // clears rectangle interior
    g.drawRect(XOFFSET, YOFFSET, WIDTH, HEIGHT);   // draws rectangle

    if(di > 0)       // don't draw unless di is set to a value > 0
    {
        // set initial values for x and y
        iold = XOFFSET;
        jold = YOFFSET;

                //  set up a loop to run through the same steps many times
        while (iold + di <= XOFFSET+WIDTH)    // main loop
```

* * *

```
            {
               inew = iold + di;                    // advance x coordinate
               jnew = HEIGHT - jold + 2*YOFFSET;
                                          // go from top to bottom or bottom to top
               g.drawLine(iold,jold,inew,jnew);
               iold = inew;
               jold = jnew;
            }                          // end of main while loop
         }                             // end of if
      }                                // end of paint method
}                                 // end of applet
```

Program SecondOrbit. This version of the orbit program has a TextField and a Button so that the operator can control the program. Notice how the declaration of constants in the program works via the declaration of final variables. If we are ever to modify this program it is better to change the values of HEIGHT and WIDTH rather than figure out what '200' means. Notice also the use of an if{} to check that the user has chosen a value of di that is greater than 0.

Problem 1.4. Motion in a square region. Write a program to depict the motion of a particle bouncing inside a square region. Use the applet to answer the following questions: Under what circumstances will the orbit close, i.e. repeat itself? Is this circumstance likely or unlikely?

J. Projects on the Fundamentals of Statistical Mechanics. The reader might not think that our problems with the bouncing ball have very much physics in them. However, even these very simple problems can be used to illustrate some very interesting physics. The subject called statistical mechanics treats the outcomes of mechanical problems in terms of the probabilities for the occurrence of various events. One major result of statistical mechanics is that if the bouncing ball is in an enclosure with sufficiently complex shape then something very simple happens: In the long run the ball is equally likely to have any direction of motion and to be in any part of the box.

We describe below three projects which are intended to introduce the student to this idea. These projects are not too hard for any

* * *

student with computer experience so that by the end of the quarter most students in the course will find them reasonably simple. However, we suggest that any student who is relatively inexperienced in computer technique spend his or her time this first week learning the basics. Everyone would benefit from reading over the projects, however.

Menu Project. A region with four walls. So far, our mechanics example has had very little interesting physics in it. Now we ask you to turn to some more interesting examples. Problem 1.4 asked you to write a program to depict a situation in which there is a ball bouncing in a square region bounded by four walls. Now for a harder job: do the same thing for an arbitrary quadrilateral.

Physical Question 1. Under what circumstances will the orbit close, i.e. repeat itself?

Physical Question 2. Statistical Mechanics says that in sufficiently complicated situations particles will reach a state of statistical equilibrium in which, all other things being equal, they will spend a time within a given region of the container which is proportional to the area of that region. Since the time spent is proportional to the probability for finding the particle in the region, we have the simple rule equal probabilities for equal areas. Our statement on the conditions required for this rule is vague. However you can be much less vague. Show that this rule is not true in most square or rectangular boxes. For most quadrilateral boxes this rule is true. Construct a numerical demonstration of the plausibility of the equal areas rule in some quite unsymmetrical quadrilateral.

Menu Project . Dynamics in the Stadium. A 'stadium' is a couple of semicircles stuck on the ends of a rectangle.

* * *

First consider a situation in which there is a ball bouncing within a circular region. Under what circumstances will the orbit close, i.e. repeat itself? Is this circumstance likely or unlikely? Then consider the motion of a point particle which bounces around a stadium. Under what conditions does this orbit close?

Someone suggests that in a stadium, for 'most' starting conditions the probability that the particle will be moving with an angle q to the x-axis is independent of q. Pick a particular stadium and choose some starting condition 'at random' and see whether you can offer evidence for or against this hypothesis.

In doing this you will have to more sharply define the hypothesis mentioned above. Specifically, you will have to choose between two alternatives: Does the particle have equal numbers of trajectories at each angle or does it spend an equal time traveling at each angle? Note: the Java Math method Math.random() generates a double value from 0 up to (but not including) 1.

Menu Project . Dynamics in the Triangle. Take a triangle with sides which have lengths L_1, L_2, and L_3. There is a ball bouncing in the triangle. Can you find the law which says, for most starting conditions of the ball, what is the relative probability that the ball will hit each of the sides?

Hint 1: The law is simple.

Hint 2: Start with a case in which one length is very much smaller than the others.

Appendix A: Java Documentation

In our applets we have been taking advantage of the work of lots of Java developers by using software that is in the Java Application Programming Interface, or Java API. (We are using the version 1.1 of the API.) You may have wondered how one learns what code has been written, what it does, and how to use it. Of course, one place to look

is in your textbook. In addition, Sun has descriptions of everything in the Java API (together with lots of other Java information) at the web site http://java.sun.com.

Appendix B: CodeWarrior

It is perfectly possible to write and run Java programs without using CodeWarrior. Everything you need to write and run Java programs (the "Java Development Kit," plus a tutorial and other documentation) is at the Sun web site. We have elected to use the CodeWarrior integrated development environment in this course because of its features that make finding bugs easier, particularly the color-coded editor and the debugger.

There is documentation on CodeWarrior in the folder sequence "MacLab Resources/References."

* * *