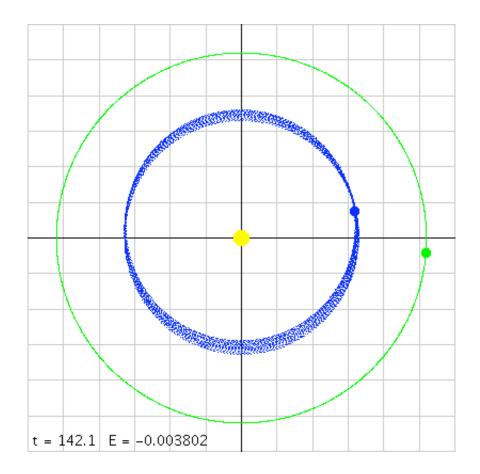
Physics Simulations in Java A Lab Manual



Daniel V. Schroeder Physics Department Weber State University

Version 2.4, August 2011

Copyright and Permissions

This lab manual is copyright ©2005–2011 by Daniel V. Schroeder.

Permission is hereby granted to reproduce this manual for noncommercial purposes. All reproductions must include this copyright notice.

You may download the latest version of this manual from the author's web site, http://physics.weber.edu/schroeder/javacourse.

The T_EX source code for this manual is also available, so other instructors may modify it to suit their needs. All modifications must be for noncommercial purposes and must carry the same copyright (adding the names of the additional authors) and permissions as the original. To obtain the source code please visit the web site or contact the author.

Contents

Preface		1
Project 1:	Hello, World!	1
Project 2:	Range of a Projectile	3
Project 3:	Adding Sine Waves	5
Project 4:	Simulating Projectile Motion	9
Project 5:	Pendulum	1
Project 6:	Orbits	5
Project 7:	Molecular Dynamics	7
Project 8:	Random Processes	3
Project 9:	The Ising Model $\ldots \ldots \ldots$	1
Java Langu	age Words	1
Java Classe	es Used in This Manual	2
Glossary		5

Preface

Introductory physics courses are full of simplifications: projectiles fly without air resistance, pendulums swing only at small angles, orbits are always circular, and no more than two particles move at any time. These kinds of simplifications are necessary and appropriate when you're first trying to understand the basic laws of nature. But the real world is far more complex, and far more interesting. Because the ultimate goal of physics is to understand the real world, students deserve a course that applies the laws of physics to more complex situations.

Fortunately, modern electronic computers make it possible to perform extremely lengthy calculations in a negligible amount of time. These days, therefore, computers offer the best avenue toward applying the basic laws of nature to complex and realistic physical systems. A computer program that models the behavior of a physical system is called a **computer simulation**. Creating and using computer simulations is an integral part of modern science and engineering.

This lab manual is intended for a hands-on introductory course in computer simulations of physical systems, using the Java programming language. The goals of the course are as follows:

- Learn enough of the Java language to write programs that do numerical calculations with graphical output;
- Learn some special techniques for doing mathematical calculations (such as solving differential equations) on a computer;
- Gain a better understanding of Newton's laws and other physical principles;
- Study a variety of physical systems that are too complex for simple pencil-andpaper calculations, and see what sorts of phenomena emerge in such systems.

Prerequisites

Before taking this course you should have completed a semester of introductory physics, covering Newton's laws of motion, conservation principles, and a bit of thermodynamics. You should also have taken at least one semester of calculus. Prior experience in programming a computer is *not* required, but you should be fairly comfortable using word processing and spreadsheet software.

Required Materials

Naturally, you'll need a computer. You should have access to a computer lab with suitably equipped computers. If you would like to use your own computer instead (or in addition), you'll need to install and configure the needed software—the Java language and a programmer's text editor—as described in Project 1. Any reasonably new computer running Windows, Macintosh OS X, or Linux should work fine.

You'll also need a convenient way of backing up your programs and other files. I recommend a USB flash memory device; even the smallest available sizes are more than adequate.

Finally, you'll need a few low-tech materials such as scratch paper, pencils, and a 3-ring binder to hold this manual.

How To Use This Manual

This manual is divided into nine chapters, corresponding to nine separate projects. In each project you will write a computer program or (more often) a small number of closely related computer programs. Rather than giving you complete programs to run, this manual provides only code fragments and general instructions on how to write your programs. This way, once you have completed each program, it will be yours.

As you create your computer programs, you will inevitably have questions and encounter difficulties. While you should try to think things through for yourself whenever possible, don't spend too much time being stuck and getting frustrated. Ask your instructor or your classmates for help. This is not a test.

Exercises and questions are sprinkled among the instructions in this manual, with space for you to write your answers. Please work each exercise and answer each question immediately, before you read on.

The general premise of this manual is that you'll learn more by *trying* something than by reading a comprehensive *explanation* of it. Computer languages are like ordinary languages in this respect: We normally learn new words by hearing, reading, and using them in context, not by studying a dictionary. But whenever you want to see a term clearly defined, be sure to consult the glossary at the end of this manual. There's some other reference material there as well.

Computer programming is fun because it's so open-ended. You'll constantly think of things to try that go beyond the explicit instructions in this manual. By all means, try anything you want! If you're not sure how to add a certain feature to one of your simulations, or if you're not sure whether it's practical to do so within a limited amount of time, be sure to ask your instructor.

When you finish a project, remove its pages from this manual and staple them together with any printed output from your programs. This stapled packet, together

6 Preface

with the source code of your computer programs, will be your "lab report." Your instructor will tell you how to electronically submit the programs themselves.

What This Manual Is Not

This manual is not a comprehensive introduction to the Java programming language. Many features of the language are not needed for the types of simulations we'll be doing, so I've ignored them. Several other features (for example, exception handling and anonymous inner classes) are used once or twice but never fully explained.

The current distribution of Java includes libraries containing about 4000 standard classes for performing various common tasks. The projects in this manual use only about 30 of them.

At some point you might want to distribute your "finished" Java software as applets, double-clickable applications, or packages. This manual won't tell you how to do that.

I've tried to design the examples in this manual to illustrate good programming practices that are appropriate to the relatively small scale of the projects. This manual is not a treatise on the principles of professional software design.

This manual is not a textbook on numerical analysis, nor is it a recipe book of numerical algorithms. We'll try out just a few algorithms, make some crude comparisons, and leave it at that.

The projects in this manual touch on some fascinating fields of physics, including nonlinear dynamics, celestial mechanics, and phase transformations. But this manual is not a textbook on any of these subjects.

Perhaps most importantly, this manual is not intended to be of any use whatsoever to someone who merely reads it without actually working through all the projects and exercises.

Why Java?

Choosing a computer language always involves trade-offs. Fortunately, there are more choices today than ever before.

An obvious choice for this course would be one of the traditional computer languages like **Fortran**, **C**, or **C++**. These languages are widely used for scientific computation due to their flexibility and speed. The languages are defined by standards committees rather than by commercial vendors, and free versions are available. However, they have grown quite complex over the years, as features have been added while maintaining compatibility with older versions. Another disadvantage is that none of these languages include built-in support for graphics, and add-on graphics libraries tend to be awkward and platform-dependent.

The **Basic** programming language was specifically designed to be easy to learn, and current versions of Basic have kept this feature. Because Basic is widely used by students and hobbyists, all modern versions include built-in, easy-to-use graphics support. Some versions (True Basic and Real Basic) are cross-platform, but the most widely used version, Microsoft's Visual Basic, runs only on the Windows operating system. The fragmentation of Basic into multiple versions, each with its own ideosyncracies, is a major disadvantage. Programs written in Basic also tend to run rather slowly. Most versions of Basic are sold commercially, though the prices are generally reasonable.

A more modern easy-to-learn language is **Python**, which is available free for virtually any platform. As an interpreted language Python is not particularly fast; nor does it include built-in graphics support. However, a great variety of add-on modules are available, including modules for scientific computing and graphics.

For mathematical calculations, the most convenient choice is often a specialized mathematical programming environment such as **Mathematica**, **Maple**, or **Matlab**. These packages contain sophisticated, speedy, built-in routines for a great variety of mathematical tasks, but their high overhead can make them rather slow and awkward when you need to program a custom step-by-step algorithm. Because they are commercial products aimed at relatively narrow markets, these packages tend to be expensive. (However, there is a free product called **Octave** that is very similar to Matlab.)

Java is a relatively new computer language, introduced by Sun Microsystems (now Oracle) in 1995 and distributed free of charge. Though the Java language superficially resembles C and C++, it is simpler and easier to learn because it does not try to incorporate every feature of these languages. A product of the internet age, Java was designed from the beginning with networks in mind. It is available for most current-model computers and operating systems, and most code written in Java will run equally well on all platforms. Java comes with a large standard library of routines for performing common tasks, including platform-independent graphics (an essential feature for this course) and graphical user interfaces. Java has a sophisticated set of security features, which have the side effect of making errors ("bugs") easier to locate and far less likely to crash your computer.

Naturally, Java has a few disadvantages. Its platform independence and security features result in a performance penalty, with calculation-intensive programs typically taking 50% longer to run than in Fortran or C or C++. Consequently, Java will never be the language of choice for performance-limited computational research. In many cases, however, Java's longer running time is more than offset by the shorter time spent *writing* a Java program. (After all, what matters is *your* time, not the computer's.) Java is still much faster than Basic or Python, though it is somewhat harder for beginners to learn.

Given the choice of the Java language, many further decisions remain. In deciding exactly *how* to use Java, I've tried to keep in mind the goals and prerequisites of this course. Those who are familiar with Java will be interested to know that this manual uses only the AWT package for graphical user interface elements, not the newer, more sophisticated Swing package. This is because AWT has everything

8 Preface

we need, it's slightly simpler to use, and even programmers who will move on to Swing need to learn most of AWT first. In addition to about 30 of the built-in Java classes, the projects in this manual make use of two custom classes that I've written: Plot (for plotting results on graphs) and DoubleScroller (for adjusting variables at run-time). These classes free the student from certain mundane programming chores, yet are small and simple enough that beginning students can learn to use them quickly and even read and understand their source code. I've resisted the temptation to use more elaborate custom libraries for graphics, controls, and mathematical algorithms, because any time spent learning to use such libraries would leave less time to learn Java, and to learn physics.

Acknowledgments

Practically none of the ideas in this manual are original. My greatest debt is to Harvey Gould and Jan Tobochnik for writing their brilliant and influential textbook, *An Introduction to Computer Simulation Methods.* From the overall course plan to dozens of specific exercises and questions, I've shamelessly borrowed ideas from this book. I've also borrowed quite a bit from *Computational Physics* by Nicholas J. Giordano.

For helping me learn the Java language I'm grateful to Michael Peskin, Wolfgang Christian, Brian Rague, and the authors of countless books and internet resources.

It is a pleasure to thank Weber State University for granting me a sabbatical leave during which this manual was written, and for giving me the opportunity to teach a course from this manual afterwards.

References

Although this manual is fairly self-contained, you may wish to consult the following references for more information on the Java language, numerical computation, and physics simulations.

- The Java Tutorial. Oracle, 1995-2011. Available online at
 - http://download.oracle.com/javase/tutorial. Also available as a set of printed books. This excellent resource covers most of the language basics, starting with "Hello, world!" instructions for various platforms.
- Introduction to Programming Using Java (sixth edition). David J. Eck. 2011. Available online at http://math.hws.edu/javanotes/. Another terrific online resource.
- Learning Java (third edition). Patrick Niemeyer and Jonathan Knudsen. O'Reilly, Sebastopol, CA, 2005. A good systematic introduction to Java.
- **Core Java 2** (eighth edition, 2 volumes). Cay Horstmann and Gary Cornell. Prentice Hall, Upper Saddle River, NJ, 2007. A thorough textbook on the Java language.

- Java Examples in a Nutshell (third edition). David Flanagan. O'Reilly, Sebastopol, CA, 2004. A useful collection of complete, working Java programs demonstrating most of Java's core features, with well-written explanations.
- Java Cookbook (second edition). Ian F. Darwin. O'Reilly, Sebastopol, CA, 2004. A handy reference providing code fragments for all sorts of tasks.
- Java API Specification. Oracle, 1993-2011. Available online at http://download.oracle.com/javase/7/docs/api/. Complete documentation on all of the built-in Java classes (about 4000 in the latest version).
- JavaTech: An Introduction to Scientific and Technical Computing with Java. Clark S. Lindsey, Johnny S. Tolliver, and Thomas Lindblad. Cambridge University Press, New York, 2005. A book that appears to be a good reference for scientists using Java.
- Numerical Methods for Physics (second edition). Alejandro L. Garcia. Prentice Hall, Upper Saddle River, NJ, 2000. An excellent undergraduate-level textbook on numerical algorithms, with applications to various physics problems. The example codes are given in Matlab and C++.
- Numerical Recipes (third edition). William H. Press *et al.* Cambridge University Press, New York, 2007. By far the most widely used reference on numerical algorithms, aimed at professional researchers and graduate students. Well written but quite advanced. Code implementations are in C++, although earlier editions are also available in C and Fortran versions.
- "Numerical integration of Newton's equations including velocity-dependent forces." Ian R. Gatland, American Journal of Physics **62**, 259–265 (1993). This excellent article emphasizes the advantages of the Euler-Richardson algorithm and explains how to implement adaptive step-size control for this algorithm. Includes references to earlier AJP articles that advocate other simple algorithms.
- An Introduction to Computer Simulation Methods (third edition). Harvey Gould, Jan Tobochnik, and Wolfgang Christian. Addision-Wesley, San Francisco, 2006. An innovative textbook that covers far-ranging physics applications at a level accessible to undergraduates. This book inspired the manual you're now reading, and provides a wealth of ideas for further projects. Earlier editions of this book used the True Basic language for the code implementations. The third edition uses the Java language and a large library of custom Java classes written by the authors.
- **Computational Physics** (second edition). Nicholas J. Giordano and Hisao Nakanishi. Prentice Hall, Upper Saddle River, NJ, 2006. This book is remarkably similar in outline and level to Gould and Tobochnik, and also used the True Basic language in its first edition. The new second edition gives algorithms in pseudo-code, with an accompanying web site that provides implementations in True Basic and Fortran. This book is also a bit more focused than Gould and Tobochnik, with fewer topics but more discussion of the results of the simula-

10 Preface

tions. I borrowed quite a bit from it in preparing this manual, and I recommend it as another source of ideas for further projects.

Project 1: Hello, World!

Writing and Running a Java Program

The first goal in learning any new programming language is always the same: Write and run a program to print (or display) the words "Hello, world!".

Why bother with such a boring program? Because the steps required to write and run even the most trivial program can be quite intricate. These steps can include some or all of the following:

- finding a computer with suitable hardware and a suitable operating system;
- installing the programming language on your computer, that is, installing the compiler (or interpreter) and any other needed software;
- correctly typing the program into a text editor or similar software;
- saving the program with an appropriate name, in an appropriate place;
- compiling the program, that is, translating it into the computer's native "machine language";
- telling the computer to run the program;
- arranging for the program's output ("Hello, world!") to end up in a place where you can see it;
- troubleshooting the many sorts of problems that can crop up while carrying out any of these steps.

The number of things that can go wrong during these steps is remarkably large. Furthermore, different people prefer different computers running different operating systems and different text editors and other programming tools, set up in different configuations, so it's nearly impossible for writers of manuals to give specific instructions and to anticipate all the types of problems that can arise.

But I have to try. So in the space below, I'll give a set of basic instructions for compiling and running a "Hello, world!" program in the Java language. I'll assume that you're using a computer running either Mac OS X or Windows. (If you're using Linux, you're enough of a computer geek to figure out how to adapt these instructions.) I'll also assume that you'll use a text-editing program to write your Java programs, and that you'll use a command-line interface to compile and run them. (If you know what an IDE is and prefer to use one, you're again on your own.) Finally, I'll assume that the Java compiler and run-time environment are already

12 Project 1 Hello, World!

installed on your computer. (All Mac OS X systems have these installed by default, but if you're using your own Windows computer, you'll probably need to download Java from Oracle's web site; see the appendix to this chapter for instructions.)

The first step is to launch your text-editing program. If your computer has a programmer's text editor installed (such as TextWrangler or JEdit), please use it. Otherwise, you can get by for now with TextEdit on a Macintosh (found under Applications/Utilities), or with Notepad on Windows (under Start-Programs-Accessories). If you use TextEdit, go immediately to the Format menu and select "Make Plain Text." All of these text editors function in basically the same way as a word processing program (such as Microsoft Word), but without all the unnecessary features to control fonts and other formatting. You *could* even use a word processor to type your programs, saving them in plain text format, but I don't recommend it.

After launching your text editor, open a new document and type in the following lines:

```
/* A program to print "Hello, world!" */
class Hello {
   public static void main(String[] arg) {
      System.out.println("Hello, world!");
   }
}
```

Here are a few things to keep in mind as you type:

- Be careful with upper and lower case. Everything in Java is case-sensitive, so, for instance, string is not equivalent to String.
- Well, not quite everything. The first line of this program is a **comment**, meant to be read only by humans, not by the computer. So it can contain any text at all, or can be omitted entirely. The comment is delimited by the /* and */ character combinations.
- Java gives you a great deal of freedom in deciding how much space to put between words and symbols, and in deciding where to start a new line. Generally, you should use new lines and indentation to make the structure of your program as clear as possible to human readers. For now, please follow the examples in this manual. You may use the tab key for indentation.

After typing this short program and checking it for accuracy, save it with the name "Hello.java" (again, case matters) in an appropriate folder. You'll want to keep all your work for this course in a single folder (I'll call it "java"), located on your hard drive in an appropriate place. On a Macintosh, the best place for your java folder is inside your home folder. On Windows, the best place is on the C drive. Either way, it's most convenient to put the java folder immediately at this level, not nested within other folders. If your computer doesn't already have a java folder, please create one at this time.

Now you need to tell the computer to compile and run your program. To do so you will use a **command-line** application, which lets you type commands to the computer like everyone did back before graphical user interfaces. On Mac OS X, the command-line application is called Terminal, and is found under Applications/Utilities. On newer versions of Windows, the command-line application is called Command Prompt, found under Start-Programs-Accessories.

When you launch your command-line application you should see a window displaying a brief message followed by a cryptic sequence of characters that ends with > or % or \$. This character sequence is called a **prompt**; it indicates that the computer is ready for you to type a command.

The commands that you'll use will vary somewhat, depending on your computer's operating system. Mac OS X is actually a version of Unix, so its commands are usually referred to as Unix commands. Windows is the successor to a command-based operating system called DOS, so its commands are usually called DOS commands. Start by typing either the Unix command 1s (for *list*) or the DOS command **dir** (for *directory*), to list the contents of what's called the "current" directory. All command-line environments use the concept of a current directory, which is where the computer will look for things by default. On Mac OS X this is initially your home folder, while on Windows it is initially the C drive. Either way, one of the items listed should be your java folder.

Since your files live in the java folder, the next step is to make this folder your current directory. The command to do this is the same in Unix or DOS:

cd java

Here cd stands for *change directory*. Notice that the prompt changes to reflect your new current directory. Type ls or dir again to list the contents of this folder, which should include your Hello.java program.

Before you can run your program, you need to translate it from the Java language into something closer to the computer's native language. This step is called **compiling** the program, and is accomplished with the javac command (c for *compile*). So type "javac Hello.java" to compile your program. If all goes well, the command window should simply display another prompt on the next line after a few seconds. If you get an error message, it probably means that you mistyped something in the program. Mistakes in the code of a computer program are called **bugs**, and fixing them is called **debugging**. Go back and debug your program if necessary, then try to compile it again. You can use the up-arrow key in the command application to automatically retype the most recent command.

After your program compiles without errors, type 1s or dir again. You'll now see a new file called Hello.class. This is the compiled version of your program. It is in a format that is virtually unreadable by humans, so don't bother trying to examine its contents. With most languages, the compiled code is in the native machine language of a particular type of computer. This means that you cannot run a compiled program on a different type of computer, and the compiler itself must be

14 Project 1 Hello, World!

different for each type of computer. Java is unusual in that the compiled code is in a machine-independent format called **bytecode**. Thus, your compiled Hello.class file can be run on a machine with any operating system (e.g., Macintosh, Windows, Linux) and any processor chip (e.g., Intel, PowerPC). The drawback of this arrangement is that additional software, called the Java Virtual Machine (JVM) or the **bytecode interpreter**, is required to run your program. (See Figure 1.1 for an overview of the process of compiling and running a Java program.)

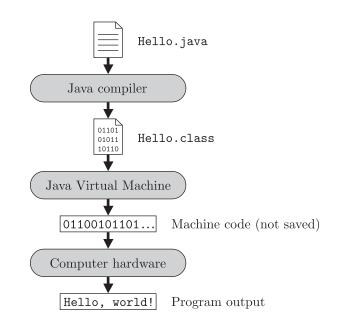


Figure 1.1. Overview of the process of compiling and running a Java program. First the source file is compiled to produce a class file in bytecode format. Then the Java Virtual Machine (or bytecode interpreter) translates the bytecode into machine code and passes this code to the computer's processor for execution.

The final step, then, is to invoke the Java Virtual Machine to run your compiled program. You do this by typing the command "java Hello", which tells the JVM to run the code contained in the file Hello.class. (Notice that the ".class" is understood, so you don't type it explicitly. I'm not sure why the compiler doesn't similarly assume that the source code file name ends in ".java".)

If all goes well, your program should type the words "Hello, world!" in the command window. The program then finishes running and the command window displays another prompt. Congratulations! You have successfully written, compiled, and run your first Java program!

If all hasn't gone well, and you can't think of any obvious fixes, please consult your instructor. Again, the number of things that can go wrong during this whole process is quite large, so it wouldn't be practical to provide a complete troubleshooting guide in this manual. For future reference, here's a summary of the commands needed to compile and run a Java program:

cd java javac Hello.java java Hello

(Use 1s or dir commands as desired to check the contents of the current folder, and use the up-arrow key to automatically retype the previous command.)

Exercise: Try the Unix/DOS command "cd ..." (note the space between the cd and the two dots), and describe what it does.

Exercise: Another useful Unix command is "rm *.class"; the DOS version is "del *.class". Try the appropriate version of this command (after cd'ing back to your java directory) and describe what it does. What do you think "rm" (or "del") stands for? (Warning: Be very careful using this command, since it cannot be undone. *Never* type something like "rm *".)

The "Hello, World!" Program

Now that you know how to create, compile, and run a "Hello, world!" program, let's look at the code of the program itself. Here it is again, without the initial comment line:

```
class Hello {
   public static void main(String[] arg) {
      System.out.println("Hello, world!");
   }
}
```

Java programs are made up of major units called **classes** (see Figure 1.2). The code that you typed defines a single new class called Hello, and also mentions two of Java's built-in classes: String and System. Technically, the Hello program includes not just the code you typed, but the code of these built-in classes as well. But I'll often use the word "program" more loosely to mean just the classes that you write, not including the built-in classes that they use.

Classes, in turn, are made up of code blocks called **methods**, each of which carries out a particular task. The Hello class defines only a single method, called **main**, but it also uses a built-in method called **println**. (Don't worry just yet about which class **println** belongs to.)

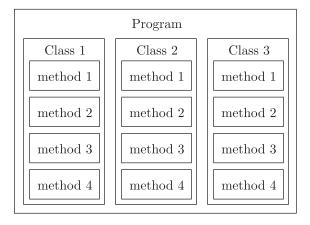


Figure 1.2. The code of a Java program is grouped into classes, each of which is made up of code blocks called methods.

Curly braces, { and }, are used in Java programs to enclose all the code that belongs to a certain class or method or smaller unit. In the file Hello.java, the outer pair of curly braces encloses the body of the Hello class, while the inner pair encloses the body of the main method. In each case, the stuff that appears immediately before the opening brace is called the **declaration** of the class or method. In these declarations we're telling Java that the name Hello represents a class and that the name main represents a method (with certain properties called public, static, and void).

Every Java program must contain at least one method called main, where execution of the program will begin. More precisely, the command java Hello tells the Java Virtual Machine to begin execution at the main method of the class Hello.

Besides the declarations of the Hello class and the main method, the only nontrivial line in our program is the one that contains println. We say that this line calls the println method, passing the characters "Hello, world!" to it. Java then executes whatever code is in the println method; apparently this code somehow causes these characters to be displayed in the command window. Fortunately, we needn't worry about how it does this.

Notice that the line containing println ends with a semicolon. This is our way of telling Java that the statement is finished. Java requires such a symbol because the end of a line has no significance whatsoever; it's perfectly legal for a statement to span two or more lines, or for multiple statements to be combined on a single line. The only statements that don't end in semicolons are those that are followed by a block of code in curly braces, like class and method declarations.

Methods are like machines that can input and output data. The inputs are called **parameters** of the method, and are placed in parentheses immediately after the method name. So the parameter that our program gives to the **println** method is the sequence of characters that we want printed, i.e., "Hello, world!". (The quotation marks are not part of the sequence; they are merely a signal to the

compiler to treat what is enclosed as a single object, called a **string**.) You'll learn about the parameter of the **main** method in the following project. You'll also learn there about methods that "return" a value; such methods are also called **functions**. Although both the **main** and **println** methods produce the side-effect of displaying the phrase "Hello, world!" in the command window, neither returns any data to the method that called it. This is the meaning of the word **void** in the declaration of the **main** method.

Exercise: Try changing println to simply print, and describe what happens.

Besides classes and methods, we also need to give names to pieces of data. Any named piece of data in Java is called a **variable**. In this course most of our variables will represent numerical data. The Hello class doesn't use any numerical data, but it still contains two variable names: **arg** and **out**. Neither of these is worth explaining in detail at this stage.

So "names" can refer to classes, methods, and variables. And what, exactly, constitutes a "name"? In Java we can use almost any name we like for the things we create, so long as it starts with a letter and contains only letters, numerals, and the underscore character (_). The only exceptions are the 50 or so **reserved words** that are part of the Java language itself, such as **class**, **public**, **static**, and **void**. Here are a few examples of valid Java names:

x x1 x_1 positionOfBall position_of_ball

Remember that Java is case-sensitive, so the names Hello and hello are just as distinct (to Java) as Hello and Goodbye.

It is conventional (not required but highly recommended) in Java to begin all class names with capital letters, and to begin all method and variable names with lower-case letters. This convention allows humans to tell at a glance what is a class and what isn't. I sometimes violate this convention for single-letter variable names such as T for temperature, since physics has its own notational conventions.

Exercise: List all the words that appear in the Hello program. For each, indicate whether it is the name of a class, method, or variable, or a part of the Java language, or part of a string of text.

18 Project 1 Hello, World!

Exercise: If you're using a programmer's text editor, it probably uses different colors to distinguish comments, strings, names, and words that are part of the Java language. Describe your editor's color scheme. (If you don't like your editor's color scheme, you can probably change it via a Preferences or Settings menu command. If you're still using TextEdit or Notepad, now would be a good time to switch to a programmer's text editor; see the end of this chapter for some specific suggestions.)

Dynamic Java

Before we leave the Hello program, I'd like you to make a modification to it. This modification won't affect the program's behavior, but it will better illustrate some of Java's most important features. It will also help you develop better programming habits.

So please go back to your source code and modify it to read as follows:

```
/* A dynamic program to print "Hello, world!" */
class Hello {
    Hello(String name) { // constructor method
        System.out.println("Hello, " + name + "!");
    }
    public static void main(String[] arg) {
        new Hello("world"); // create a new Hello object
    }
}
```

Compile and run this program, verifying that it produces the same output as before.

Here's an explanation of the new features of this version of Hello:

- This version illustrates a second type of comment, which starts with // and continues to the end of the line.
- The Hello class now contains *two* methods. The new method, also called Hello, is of a special type called a **constructor**. A constructor method must have the same name as its parent class (thus requiring an exception to the usual capitalization convention), and its declaration never includes the modifier void (or any other modifier to indicate the data type returned).
- The body of the main method now contains only "new Hello("world");", a statement that calls the constructor method of the Hello class (which then prints the desired message). Constructor methods are always called with the word new in this way; this process is called *creating a new instance of the* Hello *class*, or **instantiating** an object of type Hello. An **object** is a particular instance of a class, created while the program is running.

• Rather than always having the Hello method print "Hello, world!", we've made this method a bit more general. The method has a parameter called name (a variable), of type String. When we call the Hello method from main, the variable name is set equal to the value we provide, in this case "world". In the println statement, we use the + symbol to combine ("concatenate") this string with two others to form the longer string that will be printed.

Exercise: To make your code more readable, insert a blank line before and after each method.

Exercise: Instead of concatenating everything into a single string, you could use three separate print and println statements to print "Hello", name, and "!", respectively. Show exactly how you would do this in the space below. If you're even a little bit unsure, try it. (Note that *within* a given method, Java executes your code statements sequentially, from top to bottom.)

Exercise: The variable name "name" is completely arbitrary. Think of another good name for this variable, and change it in your code. Write your new variable name here:

As we'll see in later examples, the use of constructors and **new** gives the Java language a great deal of power and flexibility. For example, a program itself can determine, while it is running, what objects it needs to create. To describe this feature we say that Java is a **dynamic** language.

The opposite of dynamic is static, a modifier used in Java to denote variables and objects that cannot be created dynamically during run-time. A static method can create objects using new, but cannot explicitly mention any nonstatic variables. Static methods have their uses, and Java *requires* that the main method be static. Sometimes in a simple program you can get away with putting all your program statements in the main method, as we did in the first version of Hello. But this is a bad programming habit, because static methods are so restrictive. The better practice is to put as little code as possible into main—often just a single instantiation statement to transfer control to a nonstatic method.

Exercise: A method can have any number of parameters, or none at all. In the latter case, the parentheses are still required even though nothing goes between them. Modify the Hello program to eliminate the parameter of the constructor method, putting the entire string to be printed explicitly into the parameter of **println** as before. After checking that the program still works, please change it back.

20 Project 1 Hello, World!

Exercise: Add a line to the main method to create a second Hello object. Instead of passing the string "world" to the constructor, give it the name of someone else you would like to greet. Then add a third such line, and more if you wish. Compile and run the modified program, and describe the output.

Exercise: Interchange the Hello and main methods in your code, so main is above Hello instead of below it. Does this affect the program's behavior? (Please put them back where they were before turning in your source code.)

Exercise: As a review, describe the use or meaning of each of the following symbols in the Hello program.

/* */ // ; " " { } ()

Question: Describe one bug that you've had to fix during this project, and describe the error message (if any) that alerted you to the bug. If you're a perfect typist and haven't had to fix any bugs, introduce one on purpose (perhaps by leaving out a symbol) and describe what happens when you try to compile the program.

Question: What words have you looked up so far in the glossary to this manual? What words should be in the glossary that aren't?

Appendix: Text Editors

I strongly recommend that you create your Java programs using a programmer's text editor, rather than a generic text editing program like TextEdit or Notepad. A programmer's text editor will provide many small but useful features such as line numbering, automatic indentation, highlighting matching delimiters, and syntax-directed text coloring.

Here is a list of some good programmer's text editors and where to get them.

Macintosh OS X

- **TextWrangler** from Bare Bones Software is my first recommendation. This is the latest incarnation of what used to be called BBEdit Lite, a stripped-down version of their commercial product, BBEdit. It is a free download from www.barebones.com.
- **irEdit** from Idea Resources is another nice editor without too many frills. You can download it from idearesources.com, but after the 30-day trial period it costs \$25. Site licenses are available.
- **JEdit** (www.jedit.org) is a versatile, cross-platform, open-source editor written in Java. The Macintosh version works fine, but isn't as "Mac-like" as native Mac applications.
- Xcode is a full-blown IDE (integrated development environment) from Apple that includes a good programmer's text editor plus a bewildering array of commands for organizing, compiling, and debugging large software projects. If you aren't easily distracted by these other features, Xcode may be for you. Xcode is provided free with every copy of OS X as part of the "developer tools." It is not installed on new machines by default, but you can easily install it from the OS X installation disk.
- **Emacs** is a Unix text editor that has been ported to Mac OS X. Unix geeks love it, but it isn't easy to learn. A number of distributions are available, for various versions of OS X.

Windows

- **JEdit** (www.jedit.org) is a versatile, cross-platform, open-source editor written in Java. It seems to be fairly popular with Windows users.
- **Notepad2** by Florian Balmer (www.flos-freeware.ch) is nice and simple—and free.
- Notepad++ (notepad-plus-plus.org) is a free, open-source editor with more features than Notepad2.
- **TextPad** by Helios Software (www.textpad.com) is a very nice editor that is distributed as shareware (free to download and evaluate) for about \$30.
- EditPlus by ES-Computing (www.editplus.com) is another full-featured editor that is distributed as shareware (\$35), with a 30-day free trial period.

22 Project 1 Hello, World!

Linux

• If you're a Unix/Linux geek then you probably already use either Vi or Emacs, or can at least figure out where to get them.

Appendix: Installing Java

The Java compiler and virtual machine are already installed by default on all Macintosh OS X computers. If your computer runs Windows or Linux, chances are you'll have to download and install Java yourself. The download is free from Oracle's web site, java.oracle.com.

Oracle's web site is confusing because it offers so many different Java-related products. You want the "standard edition" (SE), as opposed to the enterprise/server (EE) or micro (ME) edition. Also, you want the Java "development kit" (JDK), which includes the compiler, as opposed to the runtime environment (JRE), which would only let you run programs that are already compiled. Don't worry about version numbers: Java's version numbers are extremely confusing, and even the oldest version on the web site will be much newer than what is required for the projects in this manual.

You'll want a fast internet connection for the download, which is about 80 megabytes for the most recent versions of Java. You'll also need about 200 MB of disk space. You should perform the installation from an account with administrator privileges.

The installation process is fairly self-explanatory; follow the instructions on the web site. (You may need to hunt around a bit for the instructions. As of this writing, the instructions for Windows are at http://download.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html.) Pay attention to the location where the Java programs and files are installed.

After installing Java, you'll still need to tell your command-line environment where to find it. On either Windows or Unix/Linux, this information is stored in a variable called PATH (or Path). Oracle's installation instructions explain in detail how to set the Path variable for various versions of Windows. For Windows XP/2000, the procedure is to right-click on My Computer and select Properties, then select the Advanced tab and then Environment Variables. Look for "Path" under System variables, then click Edit. The Path variable is a long string of text consisting of several directory paths separated by semicolons. What you want to do is add to this list the directory where java.exe and javac.exe have been installed. By default, that directory will be something like

C:\Program Files\Java\jdk1.6.0_02\bin

(though probably with a different version number). Unless this directory is already part of the Path variable, add a semicolon followed by this directory path to the end of what's already in Path. Then click OK, launch or relaunch the Command Prompt application, and try compiling and running your Hello program to test that your computer now knows where to find Java.

```
Name _____
```

Project 2: Range of a Projectile

You're now ready to learn how to use Java to manipulate numbers. Rather than discussing this subject in the abstract, let's again jump right in with an actual program:

```
/* Program to calculate the range of a projectile,
   neglecting air resistance. */
class RangeCalc {
   RangeCalc(double speed, double angleInDegrees) {
        double g, angleInRads, range;
                                       // declare three variables
        g = 9.8;
                                        // SI units
        angleInRads = angleInDegrees * Math.PI / 180;
        range = 2 * speed * speed *
           Math.sin(angleInRads) * Math.cos(angleInRads) / g;
        System.out.println("Range = " + range + " meters");
   }
   public static void main(String[] arg) {
       new RangeCalc(20,45); // launch at 20 m/s, 45 degrees
   }
}
```

Exercise: Type this program into your text editor and save it as RangeCalc.java. Compile and run the program as before, fixing any bugs if necessary. Write the program's output below, then try some different values of the launch speed and angle, and again write down the output.

Notice that this program has the same general structure as the dynamic version of Hello.java: a single class containing a constructor method and a main method, with main simply calling the constructor. The program's output is again displayed using println. The program also includes comments of both types, and a few blank lines for readability.

24 Project 2 Range of a Projectile

Storing and Manipulating Numbers

The major new element in this program is its use of five numerical variables, all of type double. Two of these (speed and angleInDegrees) are parameters of the constructor method, while the other three (g, angleInRads, and range) are declared and used within the body of the constructor. Notice that all five variables have names chosen to convey their meanings—a practice that makes no difference to Java, but greatly assists humans who may be trying to read and understand the program. Also notice that all five variable names start with lower-case letters, obeying the convention that only class names start with capitals.

Each of these five variables represents a storage location that can hold a real number. Java actually provides two primitive data types for real numbers, depending on the range of values and precision that are required. In this course we'll always use type **double**, which stands for double precision. The allowed range of values for this type extends to about $\pm 10^{308}$, and the precision is about 15 decimal digits—more than you're ever likely to need.

Before you can use a variable in Java, you must **declare** it, in order to tell Java what type of data the variable will store. To do this you simply put the type name (e.g., **double**) in front of the variable name, as in the first statement in the constructor method. This statement also shows how you can declare multiple variables of the same type using a comma-separated list. Each variable must be declared exactly once.

To assign a value to a variable (that is, to put some data into its storage location), you use a statement with an equals sign, putting the variable name on the left and the desired value on the right. So, for example, the statement "g = 9.8;" stores the numerical value 9.8 in the variable called g. Although the RangeCalc program never changes the value of g again, Java permits you to change the value of a variable as many times as you like.

For brevity, Java also allows you to combine a variable's declaration with its first assignment statement. Thus,

double g; // declaration
g = 9.8; // initial assignment

can be replaced with simply

double g = 9.8; // combined declaration and initial assignment

Most people, most of the time, prefer the latter form—but the choice is really a matter of taste.

Exercise: Delete the declaration line from the constructor method of RangeCalc, and instead add the word double to declare each of the three variables in its assignment statement.

The value assigned to g (namely, 9.8) is what we call a **literal** number. By contrast, the values assigned to angleInRads and range are computed from formulas, also called **expressions**. Java lets you put an expression just about anywhere you're allowed to put literal value. Notice that multiplication and division are denoted by the symbols * and /. (You can guess the symbols for addition and subtraction.) These expressions also make use of a built-in Java class called Math, which provides the static variable Math.PI and the static methods Math.sin and Math.cos. These are our first examples of methods that **return** a value, in this case of type double; calls to such methods can appear in expressions anywhere a variable of that type is allowed. In general, whenever you want to access a static variable or method that belongs to a different class, you use a dot (that is, a period or decimal point) to combine the class name with the variable or method name. Other useful methods in the Math class are summarized on the Quick Reference Sheet provided with this manual. One of these is Math.pow(x,y) for raising x to the power y, which is especially useful since Java provides no exponentiation symbol (though for small integer powers it's better to just multiply).

The particular expressions used to compute **angleInRads** and **range** are no great mystery. In conventional notation, these expressions are

$$\theta_{\rm rads} = \theta_{\rm degrees} \cdot \frac{\pi}{180} \qquad \text{and} \qquad \text{range} = \frac{2|\vec{v}_0|^2 \sin \theta \cos \theta}{g}.$$
(2.1)

The latter formula is found in most introductory physics textbooks, or you may wish to derive it from the familiar equations for projectile motion with constant acceleration. The formula assumes that the launch point and landing point are at the same height.

Exercise: Use the range formula and a pocket calculator to calculate the range of a projectile launched at an initial speed of 20 m/s and an initial angle of 45°. Write the result here, and compare to the result printed by Java.

The code that calculates the range also illustrates how it's perfectly legal to split an expression into two lines. Java lets you put extra spaces, tabs, and new lines anywhere you want, except in the middle of a word or a quoted string or a symbol combination such as //. Spaces around symbols such as = and + are optional, but are usually a good idea to make the code easier to read.

Exercise: In the space below, write a line of Java code to calculate the position (\mathbf{y}) at time **t** of a projectile thrown vertically from the origin with an initial velocity of vy0. Use the standard formula from introductory physics, $y = v_{y0}t - \frac{1}{2}gt^2$. (In Java, as in standard math notation, multiplication and division are normally performed before addition and subtraction. You can use parentheses when you need to override this rule, or just to make the order of operations more clear to human readers.)

26 Project 2 Range of a Projectile

Input and Output Refinements

The next few exercises ask you to modify and extend the **RangeCalc** program to produce more refined output, and to accept input when it is run.

Exercise: Modify the constructor method to print not just the range, but also the speed and angle that were supplied. The output should look something like this:

```
Speed = 20.0 m/s; Angle = 60.0 degrees; Range = 35.35 meters
```

You can do this either with a single long println statement, or (more elegantly) with a couple of print statements followed by a println.

In the example output that I just quoted, I conveniently rounded the range to two decimal places in order to fit the line on the page. You're probably getting tired of seeing so many insignificant figures yourself. Fortunately, Java comes with a built-in class called **DecimalFormat** that can format a number for output in almost any way you like.

To use DecimalFormat, you first have to put the line

import java.text.DecimalFormat;

above the class declaration near the top of your program. Java's 4000 or so builtin classes are grouped into **packages**; this line of code tells Java to look in the java.text package for the DecimalFormat class. (The System, String, and Math classes are in the java.lang package, which is automatically available without using import.) Next, you have to create a DecimalFormat object and initialize it to produce the type of output you want. To create a format that rounds a number to two decimal places, put the following line of code into your constructor method, somewhere before you start printing the output:

```
DecimalFormat twoDecPlaces = new DecimalFormat("0.00");
```

The string "0.00" is a pattern or template that describes the format we want. Notice that in addition to creating the DecimalFormat object with the new operator, I've also assigned it to a variable named twoDecPlaces (declared as type Decimal-Format, of course). The name doesn't have to be twoDecPlaces, but you do need a name; otherwise, you would have no way of using it later.

The DecimalFormat class includes lots of methods, but we need only one of them: format, which converts any double variable into a String that is formatted according to our pattern. To call the format method using our particular format, you could say something like:

```
aString = twoDecPlaces.format(aDouble);
```

Notice that instead of putting the class name before the dot (as in Math.sin), you need to use the name of the particular instance of the class. This is because format is not a static method—it needs to access information that belongs only to our particular DecimalFormat, not all DecimalFormats. In the RangeCalc program, though, there's no need to store the formatted string in a variable. Instead you can just wrap twoDecPlaces.format() around range inside the println statement.

Exercise: Modify your RangeCalc program as just described, to round all printed output to two decimal places. Check that you get the desired output.

Next, suppose you'd like to choose the projectile's initial speed and angle when you run the program, rather than before you compile it. In modern GUI (graphicaluser-interface) software, you would normally do this by interacting with some type of graphical control, such as a scrollbar. You'll learn how to set up scrollbars in Project 4; for now, I'll explain a simpler method.

Whenever you launch a Java program using the java command, you can optionally provide the program with additional information by typing that information after the program's name, before hitting return. Let's set up the RangeCalc program to accept values for the speed and angle in this way, so you could type

java RangeCalc 20 45

to set the speed to 20 m/s and the angle to 45°. The values 20 and 45 are called **command-line arguments**, and are automatically passed as parameters to the program's main method. If the parameter of main is called arg (the name is actually arbitrary), then you can access the first command-line argument as arg[0], the second argument as arg[1], and so on. (The arg object is actually a list of values, called an **array**. You'll learn all about arrays in a future project. For now, just note that square brackets are used to access the entries in the list, and that the first entry is numbered 0, not 1.)

Unfortunately, you can't get the first command-line argument into the variable speed by saying simply

double speed = arg[0]; // doesn't work

This doesn't work because arg[0] is stored as a string of text characters (which we hope are numerals), not as a number. You just learned how to use Decimal-Format to convert a number to a string of digits; to do the reverse, you can use the Double.parseDouble method:

double speed = Double.parseDouble(arg[0]);

The Double class (note the capital D) provides a variety of methods for dealing with variables of type double. It's in the java.lang package, so you don't need to import it.

Exercise: Add this line to your main method, and a similar line to grab the angle from the next command-line argument. Then pass the speed and angle variables as parameters to the constructor method, instead of using literal constants. Compile your program and test it for several values of the speed and angle. For a given speed, what angle yields the maximum range?

Although entering numbers as command-line arguments in this way seems easy enough, the process is fraught with peril—as you may have already learned. 28 Project 2 Range of a Projectile

Exercise: What happens now if you include nonnumeric characters (such as letters) in your command-line arguments, instead of just digits?

Exercise: What happens now if you omit one or both command-line arguments when you run the program:

As you can see, your program doesn't handle unexpected input very gracefully. A more robust approach would be to provide a default value of each variable whenever the command line doesn't contain a valid value. Here is a chunk of code that accomplishes this for the first argument:

```
double speed = 10; // default value to use if...
try {speed = Double.parseDouble(arg[0]);}
    catch (ArrayIndexOutOfBoundsException e) {} // ...no arg
    catch (NumberFormatException e) {} // ...or invalid arg
```

The try-catch construction is Java's handy way of trapping and handling what are called exceptions. But the empty sets of curly braces after each catch say to simply do nothing in the case of either error. This is fine since we've already supplied a default value of speed. Optionally, you could put println statements inside the empty curly braces to notify the user that the input was invalid so the default value was used.

Exercise: Modify your program to handle these error conditions in this way, for both the speed and angle arguments. Check that it works as expected, whether or not you supply valid input on the command line.

Conditions and Loops

Sometimes you want a program to execute some code only under a certain condition. Here's a rather trivial example of how to do this in Java:

```
if (range >= 40) {
    System.out.println("Nice throw!");
} else {
    System.out.println("Try again.");
}
```

Exercise: Add this code to the RangeCalc method in your program, right after the lines that print the numerical data. Test your program to be sure that the appropriate phrase is printed in either case.

The general form of the *if-else* construction should be fairly clear. The condition to be tested is always placed in parentheses after the word *if*. If the condition holds, the code in the next set of curly braces is executed. Any group of statements enclosed in curly braces like this is called a **block**; the curly braces are actually optional when the block contains only one statement. If the condition does not hold, the block of code following **else** is executed instead. The **else** clause can be omitted, in which case the whole statement does nothing if the condition doesn't hold.

The symbol combination >= is called a **comparison operator**, and its meaning should be apparent by now. Java's six comparison operators are

> >= < <= == !=

with the last two meaning *equals* and *does not equal*, respectively. You can also modify and combine comparisons using && for *and*, || for *or*, and ! for *not*.

Question: Why do you think Java uses a double == sign for the "equals" comparison operator, instead of a single = sign?

One of the most powerful (and necessary) features of computer languages is the ability to execute a block of code repeatedly. Such a construction is called a **loop**. Suppose, for instance, that you want your program to calculate the range for all angles between 0° and 90° , in 5° increments. You can do this using a while loop, as follows:

```
double angle = 0;
while (angle <= 90) {
    new RangeCalc(speed,angle);
    angle += 5;
}
```

The block of code within the curly braces is then executed over and over again, as long as the condition (angle <= 90) holds. The statement "angle += 5" is a shortcut for "angle = angle + 5"; the shortcut operators -=, *=, and /= work similarly.

Exercise: Put this block of code into your program's main method, replacing the code to extract the angle from the second command-line argument. (Instead of simply deleting the unneeded code, just put the comment symbols /* and */ around it. This is called **commenting-out** the code.) Your program should now print out a whole list of range values. If you haven't noticed a pattern in the values already, you should notice one now. What is the pattern?

30 Project 2 Range of a Projectile

Exercise: Most programmers consider it bad practice to put too much code into the main method. Therefore, modify your program to put a while loop in the constructor method instead of in main. The program's output should be the same as before. (Hint: The constructor method should now have only one parameter instead of two.)

An alternative to the while loop, useful when the loop is controlled by a single variable that gets changed only at the end of the block, is the for loop. Here's a for loop that's equivalent to the while loop shown above:

```
for (double angle = 0; angle <= 90; angle += 5) {
    // code to be executed for each angle goes here
}</pre>
```

Within the parentheses following for you have to put three statements, separated by semicolons: an initialization statement for the variable that controls the loop; a condition to be tested before each time through the loop (such that the loop terminates when the condition no longer holds); and a statement to be executed at the end of each loop iteration, normally incrementing the value of the loop variable. Although the for construction is less flexible than while, and takes a little more getting used to, most programmers prefer it due to its brevity.

Exercise: Comment-out the while loop in your program and replace it with a for loop. Check that you get the same output as before.

Your RangeCalc program is now complete. Before turning it in, please be sure that the formatting of the code is consistent and readable—and that it includes plenty of comments.

Exercise: Describe three different uses of parentheses in Java. (Pick three that are as different as possible.)

Graphics!

Getting tired of seeing your program's output only as text? Much of the usefulness of computers is in their ability to help us visualize what's going on.

Java comes with a sophisticated collection of built-in classes for displaying graphics on your computer screen. You'll learn to use some of these classes directly in future projects. What we really need for now, though, is an easy-to-use class that plots points on a graph. I've written such a class, called Plot; you should have a copy of its source code, Plot.java, in your java directory. (If you don't have a copy, ask your instructor how to get it or download it from the author's web site.) Open this source file in your text editor now, and briefly scan its contents. A few lines from the bottom, within a method called printThePlot, you should see the following line:

```
g.drawString(plotTitle + ", printed by XYZ on " + today,0,-10);
```

Edit this line to replace XYZ with your name or initials. Then save the file and compile it. You won't need to modify this file again.

The Plot class is not particularly fancy, but is relatively easy to use and understand. To create a new Plot object you call the constructor method, which has seven parameters. Here is an example:

```
Plot myPlot = new Plot("Plot of y vs. x",0,10,1,-1.5,1.5,0.5);
```

The first parameter is a string to put at the top of the plot window as a title. The next two parameters are the minimum and maximum values for the plot's horizontal coordinate, and the fourth parameter is the interval for this coordinate at which to draw grid lines. The last three parameters are the minimum, maximum, and grid interval for the vertical axis. All of the numerical parameters are of type double.

After creating a Plot object, you add points to it with the addPoint method. This method takes two parameters of type double, for the point's horizontal and vertical coordinates, respectively. For example,

```
myPlot.addPoint(4.2,0.42);
```

The Plot class also provides methods for changing the size, shape, and color of the plotted points, but you won't need any of these features for this project.

Exercise: Create a new Java program called RangePlot, which calculates and plots the range of a projectile (using the same formulas as in RangeCalc) as a function of angle, for a given launch speed of 20 m/s. Use the same program outline as in RangeCalc, with most of the code, including the loop, in the constructor method. Use either a while loop or a for loop to plot points for all angles from 0° to 90°, in 1° increments. Spend some time fine-tuning the program until both the code and the plot look reasonably nice. Be sure to include a few comments in your code. Use the plot's Print button to print it, and label the printed page with any important information that isn't already on it, such as the scales and units of the axes and the value used for the launch speed.

Hey, wait a minute! Why does your RangePlot program keep running, long after the code that you wrote has finished executing? Notice that the plot window doesn't disappear, and you don't get a new prompt in the command window, until you click on the window's close button, or use the Quit menu command (Mac OS only), or type control-c in the command window. For that matter, where is the code that responds to mouse events such as clicking buttons and moving the plot window across the screen?

The answer is that the Java graphics classes used by the Plot class are doing an awful lot behind the scenes. Most importantly, these classes set up a separate **thread** of execution, a sort of separate mini-process that runs in parallel with the

32 Project 2 Range of a Projectile

thread that executes your code. (On a computer with more than one processor chip, multiple threads can literally run at the same time. On a computer with just one processor, the operating system normally switches back and forth between threads so fast that to a human, they might as well be running simultaneously.) So even when the thread that executes your code has nothing left to do, the background graphics thread is still running, responding to certain types of mouse events. In Project 5 you'll learn how to set up your own threads.

Exercise: If you haven't already, make the initial speed in your RangePlot program a parameter of the constructor method, so you can use whatever value you like in your **new** statement. Then add a second **new** statement with a different value of the initial speed. Did the second data set appear in the same plot window or in a different one? Explain why.

There's actually an easy way to change the behavior you just discovered, as described in the next exercise.

Exercise: Move the line of code that declares and initializes the Plot object to the space *above* the constructor method, so it's inside the class **RangePlot** but not inside any method. Then add the adjective **static** at the beginning of this line of code. Compile and run your program, and describe the results.

In the preceding exercise you changed your Plot from a local variable to a class variable. A local variable is declared inside a method; it gets created whenever the method is called, and it cannot be accessed from other methods. Local variables are the easiest to understand, so you should use them whenever possible. But if you need to share a variable between two or more methods (or preserve its data over multiple calls to the same method), you can declare it at the class level, outside of any method; it is then an **instance** variable that gets created when you create an instance of the class (using **new**), and that can be accessed from any of the class's nonstatic methods. (Since RangePlot's only nonstatic method is its constructor method, there's no point in giving it any instance variables.) However, if you declare a variable at the class level with the word static, then it becomes a class variable—which gets created once and for all when the code for that class is loaded into the computer's memory. (Also, a class variable is accessible from any of the class's methods, even static ones.) The differences between local, instance, and class variables are extremely important in Java, and we'll come back to this topic (called **variable scope**) repeatedly.

As I said, the Plot class isn't particularly fancy. It doesn't let you label the axes or interactively resize the plot or zoom in or out. But it's easy to use and it

gets the job done. You'll be using it repeatedly during this course. Should you ever need a fancier plotting class, others are available.

Your **RangePlot** program is now finished. Please clean up its code if necessary, to make the formatting consistent and add sufficient comments.

Exercise: As a review, list all the Java language words that you've used so far in your programs. (Include only language words, not names of classes, methods, or variables.)

Exercise: Now list all the built-in Java class names that you've used so far.

Postscript: You're now ready to understand the construction

System.out.println()

which I never bothered to explain in Project 1. System is a built-in class that provides several important utility functions. System.out is a (static) class variable belonging to the System class, whose type is another built-in class called PrintStream. A PrintStream is a type of object that allows information to be converted to text and "printed" to some output device (or file). The System.out object is set up by default to send the output to your command window (though the destination can be changed if you ever need to). The PrintStream class provides print and println methods for printing strings and other objects. Thus, System.out.println() calls the println method of the out object of the System class.

You may also be wondering why it's legal to say something like

34 Project 2 Range of a Projectile

```
System.out.println("Range = " + range + " meters");
```

when range is a double, not a String. The answer is that Java is smart enough to convert *any* data type to a string, if it appears in a context where a string is expected. So range gets converted to a string automatically (using a default format), and this string is then concatenated with the two others to form the string that gets passed to println. While this automatic conversion is certainly convenient, it can also be confusing. You may be reassured to learn that Java has very few such *ad hoc* features. (If you like features like this, you'll love the Perl programming language.)

Project 3: Adding Sine Waves

The previous project introduced the essential low-level building blocks of a computer program: variables, expressions, conditions, and loops. In Java, these building blocks are grouped at a higher level into methods and classes to make a complete program.

The main purpose of this project is to give you more practice at assembling a program out of its basic components. You'll also learn a couple of new variations on those components, and you'll make the computer work quite a bit harder, doing calculations that are much too lengthy to perform by hand. The results will illustrate a deep mathematical principle that comes up over and over again in physics.

One Sine Wave

Exercise: Write a Java program called Wave.java which plots a graph of the function $y = \sin x$, where x ranges from 0 to 4π . (Note that x is in radians. Don't bother to start with degrees and convert.) Use our standard program outline, with all the interesting stuff in the constructor method and a main method that merely calls the constructor. Use a for loop to determine the set of x values to plot. Spend a bit of time adjusting the scales and grid intervals on the plot so it looks nice.

Exercise: The Plot class provides several additional methods that give you more control over the plot's appearance. Referring to your Quick Reference sheet for examples, try out each of the following, one at a time: setPointSize, setPointShape, setColor, and setConnected. These settings affect *subsequent* plotting; they have no effect on points that have already been added to the plot. Print one version of your graph showing a set of non-default options, and attach the printout to your report. As always, label the printout by hand to indicate exactly what is being plotted (including the range of values for each axis).

Two Sine Waves

Exercise: Now modify your Wave program to plot instead the following function, formed by adding two sine functions of different wavelengths:

$$y = \sin x + \frac{1}{3}\sin(3x). \tag{3.1}$$

36 Project 3 Adding Sine Waves

Make a printout of the plot, and show with some additional sketching how its shape comes about by adding the two terms. (Sketch at least a portion of the separate functions $\sin x$ and $\frac{1}{3}\sin(3x)$, and point out where they add and where they cancel.)

For all but the simplest functions, it's a good idea in Java to put the function definition into its own method. In fact, any method that returns a value (i.e., is not void) is also called a **function**. To define a function for our sum of two sine functions, you can use the following code:

```
double wave1(double x) {
    double y = Math.sin(x) + Math.sin(3*x)/3;
    return y;
}
```

Notice that in place of "void", a function definition begins with the type of value that it returns (in this case, double). The return statement tells Java what value to return. You can now use the function wave1(x) in any expression where a double value is expected—for instance, in a call to Plot.addPoint.

Exercise: Modify your Wave program to move the function definition into its own method, as just described. Put this function definition in between the constructor and main methods. Check that the plotted graph is unchanged.

Fourier Series

If you add up sine functions of shorter and shorter wavelengths, using just the right coefficients on each, you can produce periodic functions with various regular shapes. One example is the series

$$y = \sin x + \frac{1}{3}\sin(3x) + \frac{1}{5}\sin(5x) + \dots = \sum_{\text{odd } k} \frac{\sin(kx)}{k}.$$
 (3.2)

This is an example of a **Fourier series**. Adding up more than a few terms in the series by hand is quite a chore, but a computer will add up hundreds of terms without complaining!

Although you could use a variable of type double for the integer k that appears in formula 3.2, it's better to tell Java that k is an exact integer rather than a real number. Java provides several different integer data types, with different maximum values (determined by the amount of storage space used). For most purposes you can get by with the int data type, which can store integers in the range of about $\pm 2 \times 10^9$.

Exercise: Modify the definition of your wave1 function to add up an arbitrary number of terms in the series 3.2. Use a for loop contolled by a variable k of type int, being sure to sum only over odd values of k. Cut off the sum at a maximum value kMax. Try several values of kMax ranging from 5 up to a few hundred. For large values of kMax, you'll also want to make sure that your code plots the function at x values that are spaced close enough together.

Exercise: Modify your program to make kmax a command-line argument, so you won't have to recompile the code every time you want to change kMax. To convert the typed digits to an integer, use the built-in Integer.parseInt method (rather than Double.parseDouble). You'll need to pass kMax as a parameter (of type int) from your main method to your constructor method, then pass it as a (second) parameter from the constructor to the wave1 function. After all this is working, explore some more values of kMax and print out a plot for kMax = 100.

Question: As you've seen by now, this Fourier series approaches the form of a "square wave" in the limit $kMax \rightarrow \infty$. What is the approximate value of y, in this limit, on the positive portion of the square wave? (Please obtain an answer that's accurate to at least three significant figures, and explain how you know the answer is this accurate.)

Other Shapes

Exercise: Add a second function definition to your Wave program called wave2, which is the same as wave1 except that it sums over *all* integer k values, even as well as odd. Add code to plot this function in a separate window (don't delete the code that plots wave1). Run your program for several values of kMax and describe the results.

Here is another Fourier series that produces a nice symmetrical shape:

$$y = \sin x - \frac{1}{9}\sin(3x) + \frac{1}{25}\sin(5x) + \dots = \sum_{\text{odd } k} (-1)^{(k-1)/2} \cdot \frac{\sin(kx)}{k^2}.$$
 (3.3)

Exercise: Add a wave3 function to your program to calculate series 3.3. Also add code to plot this function, in the third window. Again, describe the results.

38 Project 3 Adding Sine Waves

Question: You've probably noticed that the Fourier series for wave3 converges to its final shape much more rapidly than the series for wave1 and wave2—so you can obtain a good graph of the approximate final shape using a much smaller value of kMax. Why is this? That is, what feature of wave3 makes it converge more rapidly?

Exercise: Add a fourth wave function, and a fourth plot, to your Wave program. This time, make up your own formula for the coefficients of the sine functions in the Fourier series. Spend some time trying various formulas until you find one that you like. Then clean up the code of your Wave program (if necessary), making sure it has plenty of comments.

There is a mathematical theorem that states that any periodic function, with period 2π , can be broken down into an infinite sum of sine and cosine functions of the form

$$(a_1 \sin x + a_2 \sin(2x) + \cdots) + (b_1 \cos x + b_2 \cos(2x) + \cdots), \qquad (3.4)$$

for some set of coefficients a_k and b_k . In this project you've used only sine terms (all the b_k 's have been zero), but cosine terms are also needed if the periodic function isn't odd under the operation $x \to -x$. For most periodic functions there is no simple pattern for the coefficients a_k and b_k , but there is a formula for computing these coefficients (one at a time) from the periodic function that one wishes to construct. Determining the coefficients to construct a given function is called **Fourier analysis**.

Fourier series and Fourier analysis have endless applications in physics. The functions of interest could be audio signals (sound waves), electrical signals, electromagnetic fields, quantum mechanical wavefunctions, or any other functions that represent periodic phenomena. Breaking down a complex signal into its sinusoidal components gives us a simple and powerful tool for understanding these phenomena. In the case of audio signals, the first sine wave in the series is called the fundamental, while the higher-frequency waves are called overtones. The relative strengths of the overtones determine the tone quality of a musical note.

Exercise: As a review, list all the Java language words used in your Wave program. (Include only language words, not the names of classes, methods, or variables.)

Exercise: List all the built-in Java class names used in your Wave program.

Project 4: Simulating Projectile Motion

You now know enough Java to write your first simulation program.

The idea of a **simulation** is to program the laws of physics into the computer, and then let the computer calculate what happens as a function of time, step by step into the future. (The range calculation in Project 2 wasn't really a simulation because we merely programmed in a formula for the final answer. In more complicated situations no such formula exists, so we need to go back to the basic laws of physics.)

In this project you'll simulate the motion of a projectile, first in one dimension and then in two dimensions. When the motion is purely vertical, the state of the projectile is defined by its position, y, and its velocity, v_y . These quantities are related by

$$v_y = \frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = \frac{y_{\text{final}} - y_{\text{initial}}}{\Delta t}.$$
(4.1)

In a computer simulation, we already know the current value of y and want to predict the future value. So let's solve this equation for y_{final} :

$$y_{\text{final}} \approx y_{\text{initial}} + v_y \,\Delta t.$$
 (4.2)

Similarly, we can predict the future value of v_y if we know the current value as well as the acceleration:

$$v_{y,\text{final}} \approx v_{y,\text{initial}} + a_y \Delta t.$$
 (4.3)

These equations are valid for any moving object. For a projectile moving near earth's surface without air resistance, $a_y = -g$ (taking the +y direction to be upward). In general, a_y is given by Newton's second law,

$$a_y = \frac{\sum F_y}{m},\tag{4.4}$$

where m is the object's mass and the various forces can depend on y, v_y , or both.

In a computer simulation of one-dimensional motion, the idea is to start with the state of the particle at t = 0, then use equations 4.2 through 4.4 to calculate y and v_y at $t = \Delta t$, then repeat the calculation for the next time interval, and the next, and so on. Fortunately, computers don't mind doing repetitive calculations.

But there's one remaining issue to address before we try to program these equations into a computer. Equation 4.2 is ambiguous regarding *which* value of v_y appears on the right-hand side. Should we use the initial value, or the final value,

40 Project 4 Simulating Projectile Motion

or some intermediate value? In the limit $\Delta t \to 0$ it wouldn't matter, but for any nonzero value of Δt , some choices give more accurate results than others. The easiest choice is to use the *initial* value of v_y , since we already know this value without any further computation. Similarly, the simplest choice in equation 4.3 is to use the initial value of a_y on the right-hand side.

With these choices, we can use the following Java code to simulate projectile motion in one dimension without air resistance:

```
while (y > 0) {
    ay = -g;
    y += vy * dt; // use old vy to calculate new y
    vy += ay * dt; // use old ay to calculate new vy
    t += dt;
}
```

This simple procedure is called the **Euler algorithm**, after the mathematician Leonard Euler (pronounced "oiler"). As we'll see, it is only one of many algorithms that give correct results in the limit $\Delta t \rightarrow 0$.

Exercise: Write a Java program called **Projectile1** to simulate the motion of a projectile moving in one dimension, using the Euler algorithm as written in the code fragment above. Follow the same program outline as in the previous two projects, putting most of the code in the constructor method (and as little code as possible in main). Use a time step (dt) of 0.1 second. Start the projectile at time zero with a height of 10 meters and a velocity of zero. Notice that I've written the loop to terminate when the particle is no longer above y = 0. After the loop terminates, have the program print out the time and the particle's velocity. Test your program and check that its results are approximately what you would expect. (Show your calculation of what to expect in the space below.)

Exercise: Text output is boring, so modify your **Projectile1** program to produce a graph of y vs. t for the projectile's motion. Describe how you modified the code in the space below. When the program is working correctly, print your graph and label it appropriately.

Exercise: The time and velocity printed by your program do not apply at the instant when the projectile reaches y = 0, because that instant occurs somewhere in the middle of the final time interval. Add some code after the end of the loop to estimate the time when the projectile actually reaches y = 0. (Hint: Use the final values of y and vy to make your estimate. The improved time estimate still won't be exact, but it will be much more accurate than what your program has been printing so far.) Have the program print out the improved value of t instead, as well as an improved estimate of the velocity at this time. Write your new code and your new results in the space below. Please have your instructor *check* your answer to this exercise before you go on to the next.

Exercise: The inaccuracy caused by the nonzero size of dt is called **truncation error**. You can reduce the size of the truncation error by making dt smaller. Try it! How small must dt be to give results that are accurate to four significant figures? Justify your answer. (Warning: If you make dt *too* small, your program will take a long time to run. If necessary, you can interrupt it by typing control-c in the command application.)

Air Resistance

There's not much point in writing a computer simulation when you can calculate the exact answer so easily. So let's make the problem more difficult by adding some air resistance. At normal speeds, the force of air resistance is approximately proportional to the *square* of the projectile's velocity. This is because a faster projectile not only collides with *more* air molecules per unit time, but also imparts more momentum to each molecule it hits. So we can write the magnitude of the air force as

$$|\vec{F}_{\rm air}| = c|\vec{v}|^2,\tag{4.5}$$

for some constant c that will depend on the size and shape of the object and the density of the air. The *direction* of the air force is always directly opposite to the direction of \vec{v} (at least for a symmetrical, nonspinning projectile).

42 Project 4 Simulating Projectile Motion

Exercise: Assuming that the motion is purely in the y direction, write down a formula for the y component of the air force, in terms of v_y . Your formula should have the correct sign for both possible signs of v_y . (Hint: Use an absolute value function.)

Exercise: Now modify your **Projectile1** program to include air resistance. Define a new constant called **drag**, equal to the coefficient *c* in equation 4.5 divided by the projectile's mass. Then add a term for air resistance to the line that calculates **ay**. Java's absolute value function is called **Math.abs()**. Run the program for the following values of **drag**: 0 (to check that you get the same results as before), 0.01, 0.1, and 1.0. Use the same initial conditions as before, with a time step small enough to give about four significant figures. Write down the results for the time of flight and final speed below.

Exercise: What are the SI units of the drag constant in your program?

Question: How can you tell that your program is accurate to about four significant figures, when you no longer have an "exact" result to compare to?

Exercise: Modify your **Projectile1** program to draw a second plot showing the velocity as a function of time. Run it again with **drag** equal to 1.0, and print the velocity graph. Discuss the results briefly.

Exercise: When the projectile is no longer accelerating, the forces acting on it must be in balance. Use this fact to calculate your projectile's terminal speed by hand, and compare to the result of your computer simulation.

A Better Algorithm

Today's computers are fast enough that so far, you shouldn't have had to wait too long for answers accurate to four significant figures. Still, the Euler algorithm is sufficiently inaccurate that you've needed to use pretty small values of dt, making the calculation rather lengthy. Fortunately, it isn't hard to improve the Euler algorithm.

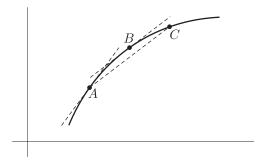


Figure 4.1. The derivative of a function at the middle of an interval (point B) is a much better approximation to the average slope (AC) than the derivative at the beginning of the interval (point A).

Remember that the Euler algorithm uses the values of v_y and a_y at the beginning of the time interval to estimate the changes in the position and velocity, respectively. A much better approximation would be to instead use the values of v_y and a_y at the *middle* of the time interval (see Figure 4.1). Unfortunately, these values are not yet known. But even a rough estimate of these values should be better than none at all. Here is an improved algorithm that uses such a rough estimate:

- 1. Use the values of v_y and a_y at the beginning of the interval to estimate the position and velocity at the middle of the time interval.
- 2. Use the estimated position and velocity at the middle of the interval to calculate an estimated acceleration at the middle of the interval.

- 44 Project 4 Simulating Projectile Motion
- 3. Use the estimated v_y and a_y at the middle of the interval to calculate the changes in y and v_y over the whole interval.

This procedure is called the **Euler-Richardson algorithm**, also known as the **second-order Runge-Kutta algorithm**.

Here is an implementation of the Euler-Richardson algorithm in Java for a projectile moving in one dimension, without air resistance:

The acceleration calculations in this example aren't very interesting, because a_y doesn't depend on y or v_y . Still, the basic idea is to estimate y, v_y , and a_y in the middle of the interval and then use these values to update y and v_y . Although each step of the Euler-Richardson algorithm requires roughly twice as much calculation as the original Euler algorithm, it is usually many times more accurate and therefore allows us to use a much larger time interval.

Exercise: Write down the correct modifications to the lines that calculate **ay** and **aymid**, for a projectile falling *with* air resistance. (Be careful to use the correct velocity value when calculating **aymid**!)

Exercise: One of the lines in the Euler-Richardson implementation above is not needed, even when there's air resistance. Which line is it, and why do you think I included it if it isn't needed?

Exercise: Modify your **Projectile1** program to use the Euler-Richardson algorithm. To declare all the new variables, you can either specify the type when each variable is first used, or declare them all at once in a line such as

double ymid, vymid, aymid;

For a drag constant of 0.1 and the same initial conditions as before $(y = 10 \text{ m}, v_y = 0)$, how small must you now make dt to get answers accurate to four significant

figures? (You should find that dt can now be *significantly* larger than before. If this isn't what you find, there's probably an error in your implementation of the Euler-Richardson algorithm.)

Your Projectile1 program is now finished. Please make sure that it contains plenty of comments and is well-enough formatted to be easily legible to human readers. Also, if you haven't already, have the program round all printed numbers to three decimal places.

Two-Dimensional Projectile Motion

Simulating projectile motion is only slightly more difficult in two dimensions than in one. To do so you'll need an x variable for every y variable, and about twice as many lines of code to initialize these variables and update them within the simulation loop. To calculate the x and y components of the drag force, it's helpful to draw a picture (see figure 4.2).

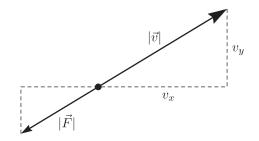


Figure 4.2. Assuming that the drag force is always opposite to the velocity vector, the similar triangles in this diagram can be used to express the force components in terms of the velocity components.

Exercise: Finish labeling this picture, and use it to derive formulas for the x and y components of the drag force, written in terms of v_x and v_y . The magnitude of the drag force is again given by equation 4.5. (Hint: This is not an easy exercise if you've never done this sort of thing before. Do not simply guess the answers! You should find that the correct formula for F_x involves both v_x and v_y . Have your instructor check your answers before you go on.)

46 Project 4 Simulating Projectile Motion

Exercise: In the space below, write the code to implement the Euler-Richardson algorithm for the motion of a projectile in two dimensions, with air resistance. As mentioned above, for every line that calculates a y component, you'll need a corresponding line for the x component. Think carefully about the correct order of these lines, remembering that to calculate the acceleration at the middle of the interval, you need to know both v_x and v_y at the middle.

Exercise: Write a program called **Projectile2** to simulate projectile motion in two dimensions. Use the Euler-Richardson algorithm, with a time step of 0.01 s. Rather than putting the simulation code into the constructor method, put it into a void method called **launch**, with three parameters of type **double**: the drag constant, the launch speed, and the launch angle in degrees. To simplify the program, take the launch position to be x = y = 0 in all cases. The constructor method should then simply call the **launch** method several times, with different parameters. At the end of the **launch** method, have the program calculate the time when the projectile crosses y = 0 and the value of x at this time, i.e., the range of the projectile. The program should print out the launch speed, angle, time of flight, and range.

Exercise: Add a Plot of y vs. x to your Projectile2 program, using the same scale for both coordinates. Think carefully about where in the code this Plot should be declared and initialized. Recall that a **local** variable is declared inside a method and is created whenever that method is called, while an **instance** variable is declared outside of any method and is created when you use **new** to create a new instance of its class. Which choice lets you plot multiple trajectories on the same graph? Explain fully.

Exercise: Check that your **Projectile2** program gives the expected results when there is no air resistance, and briefly summarize this check.

A Graphical User Interface

Are you tired of having to recompile your programs every time you want to change one of the constants or initial conditions? In the previous two projects you avoided this inconvenience by using command-line arguments to specify certain values. But this technique still required you to rerun the program for every new set of arguments. If you want to plot several runs on the same graph, you could instead have the program repeatedly prompt you to type a set of constants, starting the simulation over each time. But nowadays almost all computer software instead uses **graphical user interface**, or **GUI**, elements to accept user input from a mouse or other pointing device.

Creating a graphical user interface for a computer program can be a lot of work. You need to plan out exactly how broad a set of options to offer the user who runs your program, then design a set of graphical **controls** (buttons, scrollbars, menus, etc.) to display those options, and finally write the code to draw the controls and accept input from them. Fortunately, Java has an extensive set of built-in classes to handle GUI's.

Here is a short program that demonstrates some of Java's basic GUI classes:

```
import java.awt.*;
class HelloButton {
    HelloButton() {
        Frame myFrame = new Frame("See the button!");
        Panel myPanel = new Panel();
        Button myButton = new Button("Press me!");
        myFrame.add(myPanel);
        myFrame.add(myButton);
        myFrame.pack();
        myFrame.pack();
        myFrame.setVisible(true);
    }
    public static void main(String[] arg) {
            new HelloButton();
        }
}
```

48 Project 4 Simulating Projectile Motion

Try typing in this program now, save it as HelloButton.java, and compile and run it. Don't be disappointed when clicking on the button produces no effect. Clicking the window's close button has no effect either, so to quit the program you'll have to use the Quit menu command (on a Macintosh) or type control-c in the command window.

Java's basic GUI classes live in a package called java.awt, where AWT stands for abstract windowing toolkit. The import instruction in HelloButton tells the compiler where to find these classes. This particular program uses just three AWT classes: Frame, Panel, and Button. A Frame is just a window, complete with a title bar that displays whatever string you supply to the constructor method. A Panel is an invisible rectangular "container" for holding GUI components, which you can place within a Frame or within another Panel. A Button is a standard push-button that normally produces some effect when you click it. Notice how the Frame and Panel classes provide add methods that allow you to insert things into them. (You could add the Button directly to the Frame, skipping the intermediate Panel, but in more complicated examples a Panel is usually needed to produce the desired arrangement of GUI elements.) The pack method sizes the Frame to be just large enough to hold everything in it, while the setVisible method causes the Frame and its contents to be displayed on your screen.

Now let's make the button do something when it's pressed. Just below the line that creates the Button object, add the following code:

```
myButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hello, world!");
    }
});
```

(Be careful with all the parentheses and curly braces.) This code uses one of Java's trickiest features, an **anonymous inner class**, which I won't try to explain here. Suffice it to say that the actionPerformed method will automatically be called when the button is pressed. The ActionListener and ActionEvent classes belong to a standard Java package called java.awt.event, so you also need to add the line "import java.awt.event.*;" at the top of the program.

Exercise: Add this code to your HelloButton program and check that it prints "Hello, world!" whenever the button is pressed.

Exercise: Would you like your HelloButton program to quit when you click the close button at the top of the window? If so, you can use another anonymous inner class to respond to window-closing events:

```
myFrame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
```

Insert this code immediately after the line that creates the Frame, and check that it works. From now on it will be up to you to decide whether to include this feature in your programs. It's not really necessary when you launch programs from the command line, or when you also use a Plot window which already has an active close button. On a Macintosh it's even less necessary, because of the Quit menu command that's built into the Mac implementation of the AWT classes.

Exercise: Now go back to your **Projectile2** program. Add code to the constructor method to create a control window called **controlFrame** with a "Launch" button, so that your **launch** method is called whenever the button is clicked. For now, use any convenient constants for this method's three parameters. Compile and test your program.

The whole idea of the GUI is to let you *adjust* the three parameters of the launch method before pressing the button. In a GUI program you can set a numerical value either by typing digits into a text field, or by adjusting an analog control such as a slider or scrollbar. The AWT package provides both TextField and Scrollbar classes, which you could use directly. To simplify your programs, however, I've written a class called DoubleScroller that includes both a scrollbar for adjusting a number of type double, and a numerical readout of the current value. For example, you can create a DoubleScroller for adjusting the launch angle with the following code:

The four numerical parameters of the DoubleScroller constructor are the minimum value, maximum value, increment size, and initial value of the number being adjusted. The DoubleScroller should be an instance variable, because you'll need to access it from two different methods.

To place a DoubleScroller in your control window, simply add it like any other component: controlFrame.add(angleScroller). The DoubleScroller object is actually a Panel that contains the scrollbar and text label, so it can be added directly to a Frame without using another intermediate Panel. To retrieve the current value at which the scrollbar is set, use the getValue method, which takes no parameters and returns the double value: angleScroller.getValue().

Exercise: Make sure you have the source code of the DoubleScroller class in your java directory. Browse through this source code (briefly), then compile it. Add the lines just described to your Projectile2 program, passing the scrollbar's current value as the angle parameter to the launch method. Another nice touch is to add the statement

```
controlFrame.setLayout(new GridLayout(0,1));
```

just after creating the Frame; this line causes the various Panels in the Frame to be stacked vertically on top of each other. Compile and test your program.

50 Project 4 Simulating Projectile Motion

Exercise: Add two more **DoubleScroller** objects to your **Projectile2** program, for adjusting the drag coefficient and the launch speed. Let the drag coefficient range from 0 to 2.0 in increments of 0.005 (in SI units); let the launch speed range from 0 to 50 m/s in increments of 1 m/s. Compile and test your program.

Exercise: Temporarily modify your **Projectile2** program so that the **Button** is added directly to the **Frame**, with no intermediate **Panel**. Describe the result below, then put the **Panel** back.

Exercise: Run your **Projectile2** program, experimenting with various values of the drag coefficient, launch speed, and launch angle. Check that it produces the expected results when there is no air resistance. Print out a couple of plots showing the trajectories for various settings (be sure to label the plots). For a drag coefficient of 0.1 and a launch speed of 25 m/s, what launch angle gives the maximum range? Record your data below and explain why you would expect the optimum angle to be less when there is air resistance.

Exercise: The maximum speed of a batted baseball is about 110 mph, or about 50 m/s. At this speed, the ball's drag coefficient (as defined in your program) is approximately 0.005 m^{-1} . Using your program with these inputs, estimate the maximum range of a batted baseball, and the angle at which the maximum range is attained. Write down and justify your results below. Is your answer reasonable, compared to the typical distance of an outfield fence from home plate (about 350–400 feet)? For the same initial speed and angle, how far would the baseball go without air resistance?

Exercise: As a review, list all the built-in Java classes that you've used (directly) so far in your programs. (Either write small or use a separate page.)

Project 5: Pendulum

Name ____

In this project you will explore the behavior of a pendulum. There is no better example of a system that seems simple at first but turns out to hold intricate layers of complexity.

Figure 5.1 shows the basic setup: a fixed pivot, a massless rod of length L, and a point-mass m at the end, swinging in the plane of the page. It's easiest to analyze the motion in terms of torque and angular acceleration; recall that the angular version of Newton's second law is

$$\sum \tau = I\alpha. \tag{5.1}$$

Here I is the rotational inertia of the object, simply mL^2 for our pendulum. The angular acceleration α is defined analogously to the ordinary acceleration:

$$\alpha = \frac{d\omega}{dt} = \frac{d^2\theta}{dt^2},\tag{5.2}$$

where ω is the angular velocity and θ is assumed to be in radians.

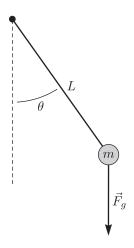


Figure 5.1. A simple pendulum of mass m and length L, acted upon by a gravitational force \vec{F}_{g} .

From the diagram we see that the torque due to gravity is

$$\tau_g = -L|\vec{F}_g|\sin\theta = -Lmg\sin\theta, \tag{5.3}$$

52 Project 5 Pendulum

where the minus sign indicates that the torque is negative (clockwise) when θ is positive, and vice-versa. If there is no friction or other torque acting, then Newton's law (equation 5.1) says simply

$$-Lmg\sin\theta = mL^2\alpha, \quad \text{or} \quad \alpha = -\frac{g}{L}\sin\theta.$$
 (5.4)

Although I drew the diagram for a small positive value of θ , this equation is valid for all angles—even angles greater than 90°, if the rod is rigid.

With this equation in hand, it's now straightforward to program a computer to simulate the motion of a pendulum. Although we could choose any desired value for the pendulum length, it's simplest to choose L so that g/L equals 1. In SI units this would imply a 9.8-meter pendulum. But there's no law requiring that we use SI units. No matter how long or short the pendulum is, there's always some choice of units for which g/L = 1. In fact, we can take L and g to equal 1 separately, by choosing L as our unit of distance and by choosing some appropriate unit of time.

Exercise: Suppose that we take our unit of distance to be 1/2 meter; let's call this unit the *ham*. Define a corresponding unit of time, called the *tic*, such that g = 1.0 ham/tic². How many seconds are in a tic?

Exercise: Write a Java program called **Pendulum1** to simulate the motion of a pendulum swinging under the influence of gravity and no other torques. Use the Euler algorithm, with the variables **theta** (in radians), **omega**, and **alpha** playing the roles of x, v_x , and a_x in linear motion. The output of the program should be a plot of θ vs. t. Use units where g = L = 1 (so you don't need these variables at all), and a time step of 0.01 in these units. Choose initial conditions such that the amplitude of the swing will be fairly small. What is the approximate period of the pendulum? Does this agree with what you learned in introductory physics?

Accuracy Over Long Time Periods

Exercise: If you haven't already, modify your Pendulum1 program so it runs for about 100 units of time. Print your graph of θ vs. t, and explain what's wrong with it in the space below.

Exercise: Work out a formula for the total energy (kinetic plus gravitational) of the pendulum, in terms of θ , ω , and constants. (Use a picture and some trigonometry to relate θ to the height of the pendulum bob. Do not set g = L = 1 in this exercise.)

Exercise: Add some lines to your program to compute and print the total energy of the pendulum at the beginning and end of the simulation. For the purpose of this computation you may take L, g, and m all to equal 1 (in appropriate units). Write the results below, and explain why something must be wrong.

Exercise: Run the simulation again with a smaller value of dt. How do the results change? Is this what you would expect?

The preceding exercises illustrate a common problem with numerical calculations: small truncation errors can add up over time to produce large inaccuracies.

54 Project 5 Pendulum

When a quantity such as the total energy is supposed to be exactly conserved, you should always monitor this quantity for any significant drift that might come from compounded truncation errors. While using a smaller time step can provide a brute-force solution to this problem, it's usually better to switch to a more accurate algorithm (if possible).

Exercise: Modify your Pendulum1 program to use the Euler-Richardson algorithm instead of the Euler algorithm. For a time step of 0.01 and a total running time of 100 time units, how much does the total energy drift?

Large-Angle Motion

Now that you've minimized your program's numerical inaccuracies, let's examine how the motion varies when the amplitude is changed.

Exercise: To avoid recompiling the program over and over, set it up to accept the amplitude in degrees as a command-line argument. Then try a range of amplitudes and observe the results. Print a graph of θ vs. t for an amplitude of 179°. Why is the period so much longer for large angles than for small angles?

Although you can read the approximate period of the pendulum from your graph, it's much more accurate to have the program calculate and print the period. Here's one way to do this: Introduce a variable called lastTheta, and set it equal to theta just before each time theta is updated. After updating theta, check whether theta and lastTheta have opposite signs (hint: check whether their product is less than or equal to zero). If so, the pendulum has just crossed $\theta = 0$. In this case, compute the time when the crossing occurred (like when you computed the time the projectile landed in the previous project) and remember this time in another variable. The time between one crossing and the next equals half the period of the pendulum.

Exercise: Implement the algorithm just described for determining the period of the pendulum. Run your program to find the period at a variety of angles ranging from 1° to 179° . Use a spreadsheet program to plot a graph of period as a function of angle, and turn in a printout of your data and graph.

Friction and Driving Forces

Now let's add some further complications: Friction to slow the pendulum down, and a periodic external torque that continually adds energy to the system.

A simple way to add friction is to subtract a term proportional to ω from the right-hand side of equation 5.4 for the angular acceleration:

$$\alpha = -\frac{g}{L}\sin\theta - c\omega. \tag{5.5}$$

A linear resistive force (or torque) of this form is called **damping**, and the coefficient c is called the damping constant.

Exercise: Add a damping term to each of the lines that compute α in your Pendulum1 program. Run the program for various values of the damping constant, ranging up to about 2.0. Describe the results briefly.

Damping removes energy from the pendulum, so the motion dies out. But we can keep the motion going by continually applying an external torque to the pendulum. A simple yet interesting way to do this is to add a term to equation 5.5 that is sinusoidal in time:

$$\alpha = -\frac{g}{L}\sin\theta - c\omega + A\sin(ft).$$
(5.6)

This additional term is called a **driving** term. Physically, it would represent a smooth, back-and-forth twisting force that is unaffected by the pendulum's position and speed. The constants A and f represent the amplitude and angular frequency of the driving torque.

Because the Euler-Richardson algorithm requires two calculations of α (one at the beginning of the time interval and one at the middle), and because the formula for α is getting rather complex, it's a good idea to move this calculation into a separate method (or function). Let's call the function **torque**, which is the same as angular acceleration in our system of units. This function should accept three parameters— θ , ω , and t—and return the value of the torque. It's best in this case to declare and initialize the constants **damp**, **driveAmp**, and **driveFreq** as instance variables, so they're available to all nonstatic methods. Then the **torque** function can be defined as follows:

56 Project 5 Pendulum

Exercise: Add this function definition to your program, and use it appropriately each time your program computes the angular acceleration. (Be sure to pass the correct time value to your **torque** function for the mid-point calculation!) Test your program with the damping constant equal to 0.5, the drive amplitude equal to 0.5, and the drive frequency equal to 2/3. (In your code, write 2.0/3.0 instead of 2/3; otherwise Java will do integer arithmetic and throw away the remainder.) You should find that after an initial "transient" behavior, the motion becomes periodic. Print a graph showing this behavior. What is the period of the motion, and what is the significance of this value?

Your Pendulum1 program is now complete, so you may wish to clean up its code and add some comments if necessary.

Chaos!

Exercise: Now increase the drive amplitude to 1.2, and run the program again (for about 100 time units). Print a graph of the motion, and describe the behavior briefly.

When the drive amplitude is sufficiently large, the pendulum can swing over the top of the pivot and the motion becomes much more complex. In some cases, such as the one you just saw, the motion never settles down to become periodic. Because the motion seems so unpredictable, the word "chaos" often comes to mind.

Chaos is actually a technical term in dynamics, used to describe behavior that is unpredictable in practice because even a tiny change in the initial condition results in an exponentially increasing change in the motion. A good way to test for chaotic behavior is to run a simulation twice, with two slightly different initial conditions, and monitor the difference in the motion as a function of time. **Exercise:** Create a new Java program called Pendulum2 to simulate the motion of two damped, driven pendulums simultaneously. (That is, use two sets of variables for the two separate pendulums, and simulate both motions using a single loop in your code.) Feel free to copy code from your Pendulum1 program, but you can omit the code for parsing the command-line argument and the code that generates text output. Call the variables theta1 and theta2, omega1 and omega2, and so on. Plot θ vs. t for both systems on the same graph. It's a nice touch to use different colors for the two systems; you can do so by adding

import java.awt.Color;

at the top of the program, then saying, for instance,

```
myPlot.setColor(Color.blue);
```

to change from one color to another. Start one system at $\theta = 0$ and the other at $\theta = 0.001$ (radians), with $\omega = 0$ initially for both. Use the same constants as in the two previous exercises, and describe the results. Print the more interesting of the two graphs.

Exercise: To better see how the motions of the two pendulums relate to each other, modify your program to produce a second graph, plotting $\ln |\theta_2 - \theta_1|$ vs. t. The Java function for the natural logarithm is Math.log. Run your program using the same constants as before, including both values (0.5 and 1.2) of the drive amplitude. Print the log-difference plot for both cases, and discuss it in some detail. Why do both plots have several downward-pointing spikes?

The roughly exponential behavior of $\theta_2 - \theta_1$ as a function of time is a feature of many dynamical systems. We can write this behavior as

$$\theta_2 - \theta_1 \approx (\text{constant}) \times e^{\lambda t},$$
 (5.7)

58 Project 5 Pendulum

where λ is called a **Lyapunov exponent**. When λ is positive the system is chaotic: even the smallest difference in initial conditions will grow over time until the two systems have radically different behavior. Since we can never know the initial conditions of a real physical system with infinite accuracy, it is effectively impossible to predict the behavior of a chaotic system over long time periods. Although our damped and driven pendulum is a somewhat contrived system, chaotic behavior is also quite common in the real world.

Exercise: Estimate the Lyapunov exponent of the pendulum system for each of the values of the contants used in the previous exercise. (Hint: Use a ruler to draw a reasonable straight line to fit your logarithmic graphs of $\theta_2 - \theta_1$, ignoring the downward-pointing spikes.)

Drawing

To further explore the chaotic behavior of the damped and driven pendulum, you'll need a simulation program that can run for longer periods of time, and that you can interact with as it is running. It will also be helpful to add some graphics to the program, to draw a picture of the pendulum itself. So let's start over with a new program called Pendulum3. Please type in the following code, compile it, and run it to see what it does:

```
/* A pendulum simulation with graphics */
import java.awt.*;
class Pendulum3 extends Canvas {
    Pendulum3() {
        setSize(400,400); // the Canvas is 400 pixels square
        Frame pictureFrame = new Frame("Driven, Damped Pendulum");
        Panel canvasPanel = new Panel();
        canvasPanel.add(this); // add the Canvas to the Panel
        pictureFrame.add(canvasPanel);
        pictureFrame.pack();
        pictureFrame.setVisible(true);
    }
```

```
// Java calls this method when the Canvas needs to be drawn:
public void paint(Graphics g) {
    g.drawString("Hello, world!",160,200);
}
public static void main(String[] arg) {
    new Pendulum3();
}
```

This program uses one of the most important features of Java (and other objectoriented languages): It extends an existing class (Canvas, another AWT class) to create a new class (Pendulum3). The new class is then called a subclass, while the class that it extends is called its superclass. When we do this, our subclass inherits all the variables and methods of its superclass. For example, the Canvas class (which provides a blank space where we can draw) has a setSize method, which our class can automatically use as if it were its own. Our class, however, can add its own new methods that do not belong to Canvas (such as main). We can also override any of the existing methods of Canvas by writing our own versions of them. The paint method is an example of this. Canvas already provides a paint method, which merely draws the Canvas's rectangle in the background color. In our own paint method we can draw anything we like.

}

All classes in Java are part of a **class hierarchy** of superclasses and subclasses. Figure 5.2 shows part of the class hierarchy, including several of the AWT classes. The **add** method actually belongs to the **Container** class, from which **Panel** and **Frame** inherit it. This method requires a **Component** as a parameter, but Java allows you to substitute any subclass for its superclass in variable assignments. Ultimately, all classes descend from Java's most basic class, called **Object**. When you declare a class without an **extends** clause, that class automatically becomes a direct subclass of **Object**.

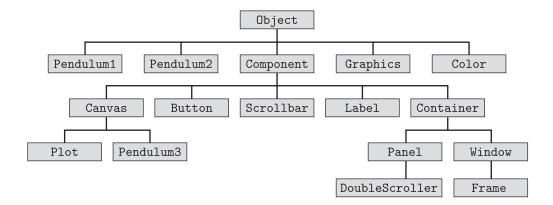


Figure 5.2. Part of Java's class hierarchy, showing some of the AWT classes and some of our own classes.

60 Project 5 Pendulum

Notice that to add our (extended) Canvas to the Panel, we use the Java language word this, which always refers to the object in which it occurs.

The Java AWT classes will call our paint method whenever our Canvas needs to be drawn. In order for them to do this, paint needs to be declared as public; otherwise it could be called only from classes within its own package. The parameter passed to paint is an object of type Graphics, another AWT class. This object contains information about where our Canvas is located on the screen, and provides a variety of methods for drawing rectangles, ovals, lines, and text; see your Quick Reference Sheet for examples. These methods use a coordinate system in pixels, with the origin at the top-left corner of the Canvas and the vertical coordinate increasing downward (opposite to the usual convention in mathematics).

Incidentally, screen coordinates must be integers. In fact, they are always of type int, which can hold any positive or negative integer whose absolute value is less than about two billion (2^{31}) . For those times when you need to work with larger integers, Java also provides a type called long whose range extends to 2^{63} . To round a double value to the nearest integer you can use the Math.round function, which returns a long. To convert a long to type int, you need to precede it by "(int)"; this is called a **cast**. Thus, a statement to round a **double** and store the result in an int might look like this:

```
myInt = (int) Math.round(myDouble);
```

Exercise: Remove the drawString statement from your program's paint method, and replace it with code to draw a pendulum. Use a small circle in the center of the **Canvas** to represent the pivot, a larger circle to represent the pendulum bob, and a line (about 150 pixels long) to represent the pendulum's rod. Introduce an instance variable called theta for the current angle of the pendulum, and set theta equal to 30° for now. Your paint method should work for any value of theta, using trig functions to calculate the rectangular coordinates from theta and the pendulum length. Refer to your Quick Reference Sheet for the needed graphics methods. Use a bit of color to add some interest.

Animation

To bring your picture of a pendulum to life, you need to put the pendulum simulation code into your program. Rather than putting the code into the constructor method, please put it into a new method called **run**, as follows:

```
public void run() {
    while (true) {
        for (int i = 0; i < 0.1/dt; i++) {
            // Euler-Richardson algorithm goes here
        }
        repaint();
    }
}</pre>
```

I'll explain the reason for using a separate run method in a few paragraphs. The condition "while (true)" simply tells Java to keep executing the loop forever (or rather, until you interrupt the program). The repaint() method (another method that belongs to the Canvas class) tells Java to call your paint method at the next opportunity. But we don't want to redraw the pendulum after every iteration of the Euler-Richardson algorithm—that would be too slow. The for loop solves this problem by using an integer variable i to count iterations before each repaint. I suggest setting dt = 0.002, which should make the truncation error quite small; in this case the pendulum will be redrawn only once for every 50 time steps. The statement "i++" in the header of the for loop is a shortcut for "i += 1"; you can similarly say "i--" to subtract 1.

Exercise: Add the **run** method, as just described, to your **Pendulum3** program, and call it at the end of the constructor method. Calculate the torque in the same way as in your previous pendulum programs, using a separate function. Run the program and describe the results.

On most computer platforms, the preceding exercise gives unsatisfactory behavior: The pendulum is redrawn erratically at best. This is because the **paint** method is called by a separate thread (a thread created by Java's AWT classes), but your main program thread never pauses to relinquish control to other threads. You can solve this problem by inserting the following code immediately after calling repaint:

try {Thread.sleep(10);} catch (InterruptedException e) {}

The Thread.sleep method (a static method of the Thread class) tells your thread to pause for the specified number of milliseconds, thus relinquishing control to the graphics thread that calls your paint method.

Exercise: Insert this code as described. You should now find that the animation is reasonably smooth. However, some fine tuning may help further. Try some other values of the sleep duration and of the time between repaints (0.1 time units). What values seem to produce the most pleasing results on your computer?

Although your program should now work fine, I'd like you to make a further refinement in the interest of developing good programming habits. Your **run** method contains an **infinite loop**, which never terminates unless interrupted. This means that if you were to put any further code into your constructor method, after calling **run**, that code would never get executed. The same goes for any further code that you might put into **main**, after calling the constructor. However, some day you may want to write code that starts a simulation and then goes on to do something else while the simulation is running. More generally, it's considered impolite in Java to write a constructor method that never returns control to the method that calls it.

62 Project 5 Pendulum

Exercise: Add a println statement just after the constructor call in your main method. Run the program and verify that this statement never gets executed.

So the preferred method of doing lengthy calculations in Java is to start your own separate thread of execution, and run the calculation from that thread. To do this, instead of calling **run** directly from your constructor method, put in the following two lines of code:

Thread myThread = new Thread(this);
myThread.start();

Also, add the words "implements Runnable" to your class declaration, right after extends Canvas. Runnable is an example of an interface, a special type of Java entity that pretends to be a class but actually isn't, because it declares one or more methods but provides no code to implement them. When you declare that your Pendulum3 class implements an interface, you are promising to provide code for these methods. The Runnable interface, in particular, requires that any class implementing it provide a public method called run. Thanks to your uncanny foresight, your class already has such a method!

(You've actually implemented an interface already, when you wrote the code to respond to Button pushes in the previous project. There you used an "anomyous inner class" to implement the ActionListener interface.)

When you call the constructor of the Thread class, you must provide it with the name of an object that implements the Runnable interface—in this case, your Pendulum3 object itself. Calling the start method of the Thread object then starts a separate thread running, which immediately calls your run method as desired. The procedure is a bit tricky, but remarkably easy.

Exercise: As just described, modify your **Pendulum3** program to run the simulation from a separate thread. Check that the **println** statement in your **main** method now *does* get executed. (You may then delete the **println** statement.)

Variable Scope

Scope	Declared	Created	Accessible to
Local	Inside a method	When method is called	Only its own method
Instance	At class level, not in any method	When object is created using new	Any nonstatic method
Class	At class level, with static	When class's code is loaded into memory	Any method

This is a good place to pause and review the concept of **variable scope** in Java programs. Recall the three levels of variable scope:

Notice that as you go down this list, there are fewer restrictions on how the variable can be used. For this reason, beginning programmers often prefer class variables to instance variables, and prefer instance variables to local variables. But as you write longer programs, you'll discover that it's easier to understand your code and to avoid bugs if you develop the opposite tendency: Use local variables most of the time, and promote them to instance or class variables only with good reason. The big advantage of local variables is that you can completely forget about them when you're thinking about some other method. You can even use the same local variable name in two different methods with no interference. The advantage of instance variables over class variables is that they can take on different values for different instances of the same class.

Question: Why doesn't your Pendulum3 program need any class variables?

Exercise: Go through the code of your **Pendulum3** program and make sure that all the variables are local except those that need to be instance variables. Which ones *do* need to be instance variables?

From now on, whenever you introduce a new variable in one of your programs, think carefully about its scope. If in doubt, make it a local variable at first and then later, if necessary, promote it to an instance or class variable.

One technicality that will come up in a moment is that the methods in an anonymous inner class count as separate methods, for the purpose of variable scope, from the method where the inner class is declared. So if an anonymous inner class needs access to a variable, you'll have to make it an instance (or occasionally a class) variable.

Finishing Touches

Now that you have a working simulation with animation, you're ready to dress it up with GUI controls and plots. Let's add a panel for controls at the bottom of the program's window:

```
Panel controlPanel = new Panel();
pictureFrame.add(controlPanel,BorderLayout.SOUTH);
```

(The second argument of the add method says that controlPanel should go at the bottom of the window.) Then create a DoubleScroller object to adjust the amplitude of the driving force, with the range varying from 0 to 2.0 in increments of 0.01. Be sure to add code to your run method to periodically check the current value of this control and set the drive amplitude variable accordingly. You could optionally create additional scrollbars to adjust the drive frequency and the damping constant, but for simplicity let's leave these parameters at 2/3 and 0.5, respectively.

64 Project 5 Pendulum

Exercise: As just described, add a scrollbar to adjust the amplitude of the driving force. Check that it works as desired.

Another useful feature to add is a button to start and stop the simulation. To implement such a button you need to create a new variable, which your **run** method can test to determine whether the simulation should be currently running. While you could use any type of variable for this, Java provides a special data type, called **boolean**, that's especially suited to this task. A **boolean** variable can have only two possible values, called **true** and **false**. So put the line

boolean running = false;

among the instance variable declarations at the top of your program, and put the condition if (running) { } around everything inside your simulation loop except the try-catch construction that calls the sleep method. (The comparison operators that you're used to using with "if" yield boolean results; if you ever need to, you can store the result of a comparison in a boolean variable, for example, myBoolean = (a < b).) Finally, add code in your constructor method to create a "Start" button and, when it is pressed, change the value of running to !running. It's also a nice touch to change the button to read "Stop" when the animation is running; you can do this with the button's setLabel method, whose parameter is the string to be displayed on the button.

Exercise: Your Quick Reference Sheet has a fairly trivial five-line example of the **if-else** construction. Write a single line of code that accomplishes the same thing, without using **if-else**.

Exercise: As just described, add a Start/Stop button to your controlPanel, and check to make sure it works.

Congratulations! You've now written your first Java program that uses lowlevel graphics functions, threads, and animation. If you'd ever like to see another working example of these features, there's one on the author's web site that you can download: AnimationDemo.java. The demo program doesn't do any physics, but it does illustrate these program features in a fairly simple context.

Exercise: Add code to your program to produce a plot of ω (vertically) vs. θ (horizontally). This is called a **phase space** plot. To keep the plotted points confined to a limited area, add code to your simulation loop to shift θ by 2π whenever necessary to keep θ between $-\pi$ and π . To show more detail on the plot, set the point size to one pixel. Run your simulation for several different values of the drive amplitude, and print the corresponding phase space plots. (Be sure to clear the plot after each time you change the drive amplitude.) Label each printout appropriately, and describe the motion in a sentence or two. On a separate sheet of paper, or on the back of one of the printouts, list the values of the drive amplitude for which the motion is chaotic.

Project 6: Orbits

In this project you will simulate the motion of planets orbiting the sun.

The only important force in this situation is gravity. According to Newton, the gravitational force between two objects is always attractive, with magnitude

$$|\vec{F}_g| = \frac{Gm_1m_2}{r_{12}^2},\tag{6.1}$$

where m_1 and m_2 are the masses of the two objects and r_{12} is the distance between them.

We'll start with a single planet orbiting the sun. Because the sun is so much more massive than any planet, it's a reasonable approximation to neglect the sun's motion. Then we can put the sun at the origin of our coordinate system, and the formula for its force on the planet will simplify.

Exercise: Use equation 6.1 and an appropriate diagram to find the x and y components of the sun's gravitational force on a planet, in terms of the planet's coordinates x and y. Be sure that you have the correct formulas before you try to write any code!

Units

Rather than using SI units, it's usually a good idea to choose units that are natural for the problem being solved. In the case of planets orbiting the sun, a natural unit of distance is the **astronomical unit** (AU), defined as the average distance between the earth and the sun (about 150 million kilometers). A natural unit of time is the year $(3 \times 10^7 \text{ seconds})$, and a natural unit of mass is the sun's mass $(2 \times 10^{30} \text{ kg})$.

66 Project 6 Orbits

Exercise: What is earth's orbital speed in these natural units (assuming its orbit to be circular)? (In this exercise and the next, do not start with SI units and then convert; there's a much easier method.)

Exercise: Find the value of G in these natural units, by applying Newton's laws to earth's orbit.

A One-Planet Simulation

Exercise: Write a simulation program called **Orbit1**, for a single planet orbiting the sun in the xy plane. Follow the same outline as in your previous simulations, putting the simulation code into the constructor method. Use the Euler algorithm for now, with a time step of 0.01 (in years). Start the planet at x = 1.5, y = 0, and an initial velocity that should give a circular orbit. The program's output should be a plot of the orbit. Here and throughout this project, be sure that your plot uses the same scale for x and y. Describe the result of your simulation for a running time of four years. Try the simulation again with a time step of 0.001, and discuss the implications of your results.

Exercise: Add a totalE function to your program, which computes and returns the planet's total energy per unit mass. Recall that the gravitational potential energy of a two-body system is $-Gm_1m_2/r_{12}$. To compute the total energy, your function will need to know x, y, v_x , and v_y ; you could make these parameters of the function, but it will be easier in the long run to simply make them instance variables. Then add code both before and after your simulation loop to call the totalE function and print out the result. Write down the results for a couple of different values of dt, and comment briefly.

Exercise: By now you should be anxious to replace the Euler algorithm in your simulation with something more accurate. Try the Euler-Richardson algorithm next, and write down the results (initial and final energy) for a couple of values of dt. Comment briefly.

The Verlet Algorithm

The Euler-Richardson algorithm is *so* much better than the Euler algorithm that you may be tempted to settle for it. However, in problems such as this where the force depends only on the positions of the particles (not on their velocities), there is another algorithm that usually does significantly better still, and is no harder to program.

You may have already noticed, while programming the Euler-Richardson algorithm, that there's no actual need to calculate the velocity at the interval's midpoint when this velocity won't be needed to calculate the force. In this case, you could instead combine this velocity calculation with the calculation of the updated position. So, for the x components, the equations

$$v_{x,\text{mid}} = v_{x,\text{initial}} + \frac{1}{2}a_{x,\text{initial}}dt$$
 and $x_{\text{final}} = x_{\text{initial}} + v_{x,\text{mid}}dt$ (6.2)

can simply be combined into the single formula

$$x_{\text{final}} = x_{\text{initial}} + v_{x,\text{initial}} dt + \frac{1}{2} a_{x,\text{initial}} (dt)^2.$$
(6.3)

68 Project 6 Orbits

You should recognize this formula from introductory physics: It is the *exact* formula for the motion of an object whose acceleration is *constant*. We'll continue to use it (for small dt) even when the acceleration isn't constant, as we've already been doing, in effect, with the Euler-Richardson algorithm.

Our improvement on the Euler-Richardson algorithm will be in the calculation of the updated velocity. Once we've updated the position using equation 6.3, we can use this updated position to calculate the acceleration at the *end* of the time interval (so long as the force depends only on the position, not on the velocity). We can then estimate the *average* acceleration as the average of the initial and final accelerations, and use this average to update the velocity:

$$v_{x,\text{final}} = v_{x,\text{initial}} + a_{x,\text{average}} dt \approx v_{x,\text{initial}} + \left(\frac{a_{x,\text{initial}} + a_{x,\text{final}}}{2}\right) dt.$$
(6.4)

This estimate of the average acceleration is more symmetrical, and hence more accurate in most situations, than the Euler-Richardson method of calculating a_x from an estimated value of x_{mid} . The combination of equations 6.3 and 6.4 is known as the **Verlet algorithm**, or alternatively as the **second Taylor approximation** (STA).

The slick way to implement the Verlet algorithm is to break equation 6.4 into two parts, separated by the calculation of the new acceleration, as follows:

- 0. Before the loop begins, calculate the initial acceleration.
- 1. Update the position using equation 6.3.
- 2. Update the velocity half-way, adding $\frac{1}{2}a_x dt$.
- 3. Update the acceleration, calculating it from the new position.
- 4. Finish updating the velocity, again adding $\frac{1}{2}a_x dt$.
- 5. Repeat steps 1 through 4 in each loop iteration.

Notice that this procedure requires only *one* evaluation of the acceleration for each time interval. In the next project, where execution speed will be an issue, this will be another significant advantage over the Euler-Richardson algorithm.

Exercise: Modify your **Orbit1** program to use the Verlet algorithm. Since the acceleration needs to be calculated once outside the loop and once inside it, move this code into a separate method and call it from both places. Test your program for the same initial conditions, running time, and values of **dt** that you used in the previous exercise, and compare the accuracy to which the two algorithms conserve energy.

Kepler's Laws

In the early 1600's, Johannes Kepler was the first to discover simple mathematical laws to describe the observed motions of the planets. Nowadays we number Kepler's important discoveries 1, 2, and 3; I like to throw in a 0th law that may have gone without saying at one time, but isn't at all obvious from a modern perspective:

- 0. Planetary orbits are closed paths; each planet returns to the same point after one full orbit.
- 1. The shape of each orbit is an ellipse, with the sun at one focus of the ellipse.
- 2. The planets move faster when they are closer to the sun, in such a way that a line drawn from the sun to any planet sweeps out equal areas in equal times.
- 3. The outer planets move slower than the inner ones, in such a way that the cube of the length of the ellipse's semimajor axis is proportional to the square of the period of the orbit.

Later, in 1687, Isaac Newton published a book showing how all of Kepler's laws (and much more) can be *deduced* from his more fundamental laws of motion and gravity. Even today, however, a rigorous deduction of Kepler's laws from Newton's laws requires some sophisticated and lengthy calculations. On the other hand, you now have a computer program that simulates planetary motion according to Newton's laws. Let's check, then, whether the resulting motion obeys Kepler's laws.

Exercise: (Kepler's 0th Law.) If you haven't already, try out some other initial conditions for your simulated planet. A good way to start is to make the initial values of x, y, and v_x the same as for earth, and vary the value of v_y from slightly lower than earth's speed to slightly higher. Describe the results in the space below. Does your simulated planet obey Kepler's zeroth law? How small a value of dt must you use to obtain consistent results?

Exercise: (Kepler's 1st Law.) An ellipse is defined as the set of all points for which the sum of the distances to the two foci is a constant. According to Kepler's first law, one focus of the ellipse should be at the sun, which is at the origin of your coordinate system. For the initial conditions suggested in the previous exercise, the other focus should be symmetrically located exactly 1 AU from the left end of the

70 Project 6 Orbits

ellipse. (The second focus may lie either left or right of the first.) Adjust your initial conditions to obtain a reasonably elongated orbit, and adjust the scale of your plot so the orbit fills most of the window (but keep the same scale for x and y). Print this plot and label the two foci. Then pick at least three dissimilar points along the orbit and for each, use a ruler to measure the sum of the distances to the two foci. Show your calculations on the printout, and discuss whether Kepler's first law seems to hold for your orbit.

Exercise: (Kepler's 2nd Law.) Temporarily modify your **Orbit1** program to plot points on the graph only at longer, fixed time intervals, so that only two or three dozen plotted points appear around the orbit. Use the same values of **dt** and the initial conditions as in the previous exercise. Print the plot, and at three dissimilar intervals along the orbit, use a ruler to determine the approximate area swept out by an imaginary line from the sun to the planet. Show your area calculations on the printout, and discuss whether Kepler's second law seems to hold for your orbit.

Exercise: (Kepler's 3rd Law.) The semimajor axis of an ellipse is defined as half of its widest width. As long as you use an initial v_x of zero, this width should be along the x axis of your plot. You can then test Kepler's third law as follows. Add code to your program to determine when the planet crosses the x axis, and for each crossing, to print out the time and the value of x. From these data, you can calculate both the semimajor axis and the period of the orbit. Run the simulation for three dissimilar orbits, recording the data below. Then for each orbit, calculate the cube of the semimajor axis and the square of the period. Discuss whether your results are consistent with Kepler's third law.

Elongated Orbits and Variable Time Steps

By now you may have noticed that the closer your simulated planet gets to the sun, the smaller you need to make dt to reduce truncation errors to an acceptable level. For highly elongated orbits this is awkward, because the small value of dt is needed only near one end of the orbit, while a much larger value of dt would suffice elsewhere.

Fortunately, there's no law that says we have to use the same value of dt throughout the simulation. Instead, we can use what is called **adaptive step size control** to continually adjust dt as appropriate. Nearly all of the "professional quality" algorithms for numerically solving differential equations employ some sort of adaptive step size control. Most of these algorithms are quite complex and sophisticated—not worth our time in a first course on computer simulations. In your **Orbit1** program, however, there is a very simple way to add adaptive step size control.

Think about it: We want dt to be small when the planet is close to the sun (where its acceleration is large and rapidly changing) but large when the planet is farther away (where its acceleration is small and slowly changing). A natural way to accomplish this would be to make dt proportional to r, or to some positive power of r. Equivalently, we could make dt proportional to some negative power of the acceleration, $|\vec{a}|$, which is proportional to $1/r^2$:

$$dt \propto |\vec{a}|^{-n} \propto r^{2n}.$$
(6.5)

Let's work with $|\vec{a}|$, since this will make it easier to generalize our approach to multi-planet simulations in the next section. The optimum power of $|\vec{a}|$ is not easy to guess, but I've done a bit of pencil-and-paper analysis that seems to indicate that n = 1 is a good choice. In practice, this choice seems to work just fine.

Exercise: Modify your Orbit1 program to set dt equal to a constant (call it tolerance) times $|\vec{a}|^{-1}$. This should be done at the beginning of each loop iteration. To select a good value of tolerance, either do a quick calculation or just try some values and see what works. Explain how you chose your value of tolerance in the space below.

Exercise: Use your **Orbit1** program to model the orbit of Halley's comet, which is 0.58 AU from the sun at its closest approach and has a period of 76 years. How far from the sun is the orbit's most distant point?

72 Project 6 Orbits

A Two-Planet Simulation

Don't let this get you down, but all the results of your **Orbit1** program were already known to Newton more than 300 years ago. The real advantage of a computer simulation over traditional analytic methods comes at the next stage, when we want to model a system that is more complex than a single particle subject to an inversesquare force. For instance, you could easily explore what happens when the force law is modified, either in some hypothetical universe or in the real solar system due to the sun's nonspherical shape or the effects of general relativity. An even more difficult problem, though, is to predict the motion of three or more mutually gravitating objects. With the exception of a few unrealistically symmetrical configurations, there are no analytic formulas for the motions of such systems.

Your next task will be to write a new computer program (Orbit2) to model the behavior of *two* planets orbiting the sun, including the effects of their mutual gravitational attraction. Because these effects are often small, you'll need to run the simulation for a relatively long time—long enough to observe dozens or hundreds of orbits. This requirement dictates a different approach to the overall design of the program. For one thing, you'll want to run the simulation from a separate thread, as in the Pendulum3 program. Secondly, you'll want to show each planet's *current* position in a more visible way on the plot, so you can follow the motion even when the trail of dots becomes too thick to see the individual points.

Let's focus on this second issue. What we need is a Plot class with an additional feature, namely the ability to show the current positions of a couple of planets as large circles, even while their past positions are plotted as tiny dots. We could create a totally new class that includes most of the code that's in Plot, plus some custom code to suit our needs. Instead, we'll take a more object-oriented approach and *extend* the Plot class, overriding some of its methods to add our custom features. Then there will be no need to duplicate code that's already in Plot, or to modify the Plot class itself (which would have undesirable side effects for other programs that use Plot).

Now would be a good time to look at the Plot.java source file and get some idea of how the Plot class works. Notice that Plot is a subclass of Canvas, with a paint method that puts the graphics onto the screen. This method actually contains only a single line of code, which copies the entire image from a separate, off-screen image that has already been drawn. This approach allows the image to be copied to the screen very quickly, even when it contains pixel-level detail; drawing the whole image from scratch in paint would be too slow when the Plot contains more than a few hundred points.

In your Orbit2 program, you'll want to override this paint method with your own version that does the same thing, then goes on to draw the current positions of the two planets as larger circles. Here, then, is an outline to use for Orbit2:

```
class Orbit2 extends Plot implements Runnable {
  static double rMax = 2;
                               // max range of plot
  Orbit2() {
      super("Orbits", -rMax, rMax, 1, -rMax, rMax, 1);
      // initialize variables and start simulation thread
  }
 public void run() {
      while (true) {
         doStep();
         try {Thread.sleep(10);} catch (InterruptedException e) {}
      }
  }
  synchronized void doStep() {
      // execute one step of the simulation
  }
 public void paint(Graphics g) {
      super.paint(g);
      // draw planets at current positions
  }
 public static void main(String[] arg) {
      new Orbit2();
  }
```

Project 6 Orbits

 $\mathbf{73}$

You can probably guess that "super" calls the constructor method of the superclass (in this case Plot) of the present class, while "super.paint" calls the paint method of the superclass. The other new word in this code is synchronized, which prohibits any so-labeled method(s) within a given class from being called simultaneously by two different threads. Don't worry for now about why this is necessary.

}

Exercise: Use the outline above to write an orbit simulation program that runs indefinitely as just described. Although the goal is to simulate the motion of two planets orbiting the sun, start with just a single planet (but call its variables x1, vx1, and so on, in anticipation of adding a second planet). In the doStep method, use addPoint to plot the planet's position as a single-pixel square and then execute one step of the Verlet algorithm with adaptive step size control as described in the previous section. In the paint method, add code to draw the planet's current position as a small circle; you may wish to borrow some code from Plot for converting the planet's location to pixel coordinates. Test your program for a couple of different initial conditions, and make sure it is working before you go on.

74 Project 6 Orbits

Exercise: Add a Start/Stop button to your program, as you did in Pendulum3. You can add the button to the controlPanel object (see the source code of Plot) in the usual way; you'll need to call plotFrame.pack to make it visible. Test the button to make sure it works.

Exercise: It's a nice touch to add a yellow circle at the origin of your plot to represent the sun. You can do this by overriding the clearThePlot method, first calling super.clearThePlot and then using addPoint to draw the yellow circle. Do this, and test to make sure it works.

Exercise: Now add a second planet to your simulation, orbiting like the first under the influence of the sun's gravitational pull. Don't worry yet about including the gravitational force *between* the two planets. For simplicity, please assume that the second planet also orbits in the xy plane. In the line that calculates dt, use the larger of the two planets' accelerations. (Use the Math.max function.) Plot the two planets' orbits in different colors. Again, test to make sure everything is working.

Exercise: Finally, modify your simulation code to include the gravitational force between the two planets. Up until now the masses of the planets have been irrelevant, but at this point you'll have to introduce variables to represent the masses (in units of the sun's mass). To calculate the force, it's helpful to first calculate the x and y separations between the two planets, storing these in temporary variables. Be very careful to check that you're using the correct formula for the force. To test your program, start the planets in orbits that would be circular if they did not interact, with radii of 1 and 1.5 (about right for earth and Mars). First set both masses equal to 10^{-6} , and verify that the orbits remain essentially circular. Then set both masses equal to 0.02 (about 20 times the mass of Jupiter), and describe what happens.

Exercise: As a further test, add a totalE function to your program that calculates and returns the total energy of the two planets, including the gravitational energy of their mutual interaction. To monitor the energy, have your paint method call the totalE function and display the result, to six decimal places, in the bottom-left corner of the plot. Do some tests and describe the results.

Your Orbit2 program is now finished! It can be used to explore a great variety of scenarios; feel free to experiment. The following exercise should give you a good start.

Exercise: Set the masses of your two planets to 0.001 and 10^{-8} , to simulate Jupiter and an asteroid. Start both in orbits that would be circular if they did not interact with each other, with Jupiter at 5.2 AU from the sun and the asteroid at 3 AU. Describe the resulting orbits. Then gradually increase the size of the asteroid's orbit, and describe the results. For what sizes of the asteroid's orbit does the influence of Jupiter appear to be the greatest? Can you explain the results? Print out at least two representative plots, and label them appropriately. Observations of the actual asteroid belt show that there are gaps in it, where very few asteroids are found. At what distances from the sun would you expect to find gaps?

76 Project 6 Orbits

[This page intentionally left blank.]

Project 7: Molecular Dynamics

If a computer can model three mutually interacting objects, why not model more than three? As you'll soon see, there is little additional difficulty in coding a simulation of arbitrarily many interacting particles. Furthermore, today's personal computers are fast enough to simulate the motion of hundreds of particles "while you wait," and to animate this motion at reasonable frame rates as the calculations are being performed.

The main difficulty in designing a many-particle simulation is not in the simulation itself but rather in deciding what we want to learn from it. Predicting the individual trajectories of a hundred particles is usually impractical because these trajectories are chaotic. Even if we could make such predictions, the sheer amount of data would leave us bewildered. Instead, we'll want to focus on higher-level patterns and statistical data.

In this project we'll also shift our attention from the very large to the very small—from planets to molecules. The main goal will be to learn how the familiar properties of matter arise from motions at the molecular scale.

Molecular Forces

Under ordinary circumstances, molecules are electrically neutral. This implies that the forces between them are negligible when they're far apart. However, when two molecules approach each other, the distortion of their electron clouds usually produces a weak attractive force. When they get too close, the force becomes strongly repulsive (see Figure 7.1). For all but the simplest molecules, these forces also depend on the molecules' shapes and orientations. In this project we'll ignore such complications and just simulate the behavior of spherically symmetric molecules such as noble gas atoms.



Figure 7.1. When two molecules approach each other, there is a weak attractive force between them. When they get too close, the force becomes strongly repulsive.

Even for the simplest molecules, there is no simple, exact formula for the intermolecular force. Fortunately, we don't really need an exact formula; any approximate formula with the right general behavior will give us useful results. The

78 Project 7 Molecular Dynamics

formula that is most commonly used to model the force between noble gas atoms is the **Lennard-Jones potential**,

$$U(r) = 4\epsilon \left[\left(\frac{r_0}{r}\right)^{12} - \left(\frac{r_0}{r}\right)^6 \right].$$
(7.1)

This is a formula for the potential energy, U(r), in terms of the distance r between the centers of the two interacting molecules. The constants r_0 and ϵ represent the approximate molecular diameter and the overall strength of the interaction. The numerical values of these constants will depend on the specific type of atom; the table below gives some values obtained by fitting the Lennard-Jones model to experimental data taken at low densities.

	r_0 (Å)	$\epsilon~(\mathrm{eV})$
helium	2.65	0.00057
neon	2.76	0.00315
argon	3.44	0.0105

A graph of the Lennard-Jones potential is shown in Figure 7.2. When $r \gg r_0$, the energy is negative and falls off in proportion to $1/r^6$. This is the correct behavior of the so-called van der Waals force, and can be derived from quantum mechanics and electromagnetic theory. When $r < r_0$ the energy becomes positive, rising very rapidly to give a strong repulsive force as r decreases. John Lennard-Jones chose an r^{-12} term to model this repulsive behavior because it was mathematically convenient in the days before computers. We'll stick with his choice because it's as easy to use as any, and no simple improvement to it would give significantly more accurate results.

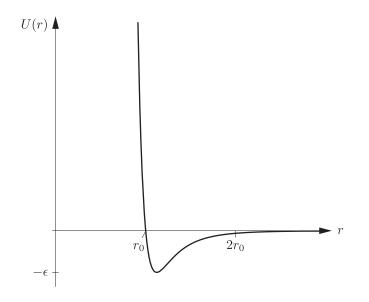


Figure 7.2. The Lennard-Jones potential energy function (equation 7.1).

Exercise: Find the value of r (in terms of r_0) at which the Lennard-Jones function reaches its minimum, and show that its value at that point is $-\epsilon$.

Exercise: Consider two molecules, located at (x_1, y_1) and (x_2, y_2) , interacting via the Lennard-Jones force. Find formulas for the x and y components of the force acting on each molecule, in terms of their coordinates and their separation distance r. (Hint: First differentiate the Lennard-Jones function with respect to r, to obtain the magnitude of the force. Draw a picture, and be careful with minus signs.)

Units

Once again we can make our lives easier by choosing units that are natural to the system being modeled. For a collection of identical molecules interacting via the Lennard-Jones force, a natural system of units would set r_0 , ϵ , and the molecular mass (m) all equal to 1. These three constants then define our units of distance, energy, and mass. Units for other mechanical quantities such as time and velocity are implicitly defined in terms of these.

Exercise: What combination of the constants r_0 , ϵ , and m has units of time?

Exercise: If we are modeling argon atoms using natural units, what is the duration of one natural unit of time, expressed in seconds? (Use the values of r_0 and ϵ from the table above. Note that the ϵ values are given in electron-volts (eV), where 1 eV = 1.60×10^{-19} J.)

Exercise: Suppose that an argon atom has a speed of 1, expressed in natural units. What is its speed in meters per second?

Another quantity that we'll eventually want to determine for our collection of molecules is temperature. To define a natural unit of temperature we can set Boltzmann's constant k_B (which is essentially a conversion factor between molecular energy and temperature) equal to 1. In conventional units,

$$k_B = 1.38 \times 10^{-23} \text{ J/K} = 8.62 \times 10^{-5} \text{ eV/K}.$$
 (7.2)

I'll explain later how to actually determine the temperature of a system from the speeds of the particles.

Exercise: Suppose that a collection of argon atoms has a temperature of 1 in natural units. What is its temperature in kelvin? (For comparison, the boiling point of argon at atmospheric pressure is 87 K.)

Exercise: Repeat the previous exercise for helium, and discuss the result briefly.

Arrays

To simulate the motion of just *two* noble gas atoms, you could simply modify your **Orbit2** program to use the Lennard-Jones force law instead of Newton's law of gravity. Recall that in that program you used the variables x1, y1, vx1, and so on for the first planet, and x2, y2, vx2, and so on for the second planet. As you can imagine, this approach becomes awkward if you add more planets (or molecules). Fortunately, Java (like nearly all programming languages) provides a convenient method for dealing with a list of similar variables. Such a list is called an **array**.

A Java array can have up to 2^{31} elements, assuming that your computer has enough memory. These elements are numbered from 0 up to some maximum value. To access a particular element of an array, you simply put that element's number (which must be an int) in square brackets. For example, if x is an array of 100 numerical values, you can access its elements by typing x[0], x[1], and so on up to x[99]. The number in brackets is called an **index**. Because the index of an array's first element is zero, the index of the last element is always one less than the size of the array. (This convention is common to Java, C, and C++. It drives Fortran programmers nuts, because Fortran array indices start at 1 rather than 0.)

This notation would be no improvement at all if the quantity in brackets always had to be a literal integer constant. But it doesn't: You're allowed to put any integer variable, or any integer-valued expression, inside the brackets. So, for instance, if you want to set every element of an array of 100 doubles equal to 42, you could write something like this:

```
for (int i=0; i<100; i++) x[i] = 42.0;</pre>
```

Programs that use arrays tend to have lots of for loops. Once you get into the habit of starting the loop variable at 0 and using a strict < (rather than <=) in the loop condition, you'll hardly ever make the mistake of trying to access the nonexistent array element just beyond the end of the list (x[100] in this case).

Before you can use an array you have to declare it, which you do by putting an empty pair of brackets immediately after the variable type:

double[] x;

(Any primitive or class type is allowed, so if you ever need to, you could create a whole array of Plots, for instance.) Notice that the declaration does not specify

82 Project 7 Molecular Dynamics

the size of the array. You do this when you actually allocate memory for the array, using **new**:

x = new double[N];

In this example I've used an integer variable, N, for the size of the array. This integer is the number of array elements, so the last element will be x[N-1]. Naturally, you'll need to initialize N before creating the array.

When you create an array of numerical variables using **new**, the values of all the elements are automatically initialized to zero.

The Program Outline

You're finally ready to start writing the code of a molecular dynamics simulation. I should warn you that this program will end up being the longest one described in this manual: over 200 lines when it's finished. But I'll make the major design decisions for you, and guide you through the coding and testing process in stages.

First decision: You'll simulate a system of noble gas atoms that live in a twodimensional "box," where their motions are confined to a fixed square region in the xy plane. (A three-dimensional simulation would obviously be more realistic, and would be only a little more difficult to code, but would be much harder to visualize and to display on a flat screen.)

Second decision: You'll use the same general program design as in Pendulum3: extending a Canvas to draw a picture of the system, and implementing Runnable to calculate the motions and animate the display. Eventually you'll add some GUI controls to a Panel below the Canvas.

With these decisions made, you can go ahead and start writing the program. Please call it MD. java.

Exercise: Create a new Java source file called MD.java, and type in the outline of your program as described above. Include declarations for a constructor method, a run method, a paint method, and a main method.

Third decision: Please use the following instance variable names and values:

int N = 50;	// number of molecules
<pre>int canvasWidth = 400</pre>	<pre>// canvas width (and height) in pixels</pre>
<pre>int pixelsPerUnit = 20</pre>	<pre>// number of pixels in one distance unit</pre>
double boxWidth = canvas	Width * 1.0 / pixelsPerUnit;

The variable pixelsPerUnit determines the scale of your screen representation of the system; you'll draw the molecules as circles with this diameter, in pixels. The variable boxWidth is the width (and height) of the box in natural units, which needn't be an integer; notice that I've inserted "* 1.0" in the computation to force Java to do real arithmetic, rather than integer arithmetic in which the remainder would be discarded.

It's worthwhile to pause and think about the two different coordinate systems that your program will use. Most of the program will work in our natural system of units in which each molecule is one unit wide. But the program's paint method, which actually draws the molecules on the screen, will need to convert these natural coordinates into pixel coordinates, using pixelsPerUnit as a conversion factor and rounding to the nearest integer.

Exercise: Add these variable declaration/initializations to your program. Then add code to the constructor method to set up and display a Frame containing a Panel containing the Canvas. To make the Canvas visible, set its background color to something other than the default for your operating system, for example,

```
setBackground(Color.white);
```

Also have the constructor create a new thread that will execute your **run** method. For now, have **run** simply print a message to confirm that it's being called. If you haven't already, add code to **main** to call the constructor method. Compile your program and check that everything works as expected.

Exercise: Add declarations and **new** statements for arrays to store the positions, velocities, and accelerations of the molecules. At the beginning of the constructor method (before initializing the graphics), add some temporary code to set the positions of a couple of the molecules to interesting nonzero values. Then add code to your **paint** method to draw all the molecules as circles, centered on the current position values, with diameter **pixelsPerUnit**. Test your code, changing the temporary position values to be sure that the **paint** method is working properly.

Exercise: Now replace your temporary initialization code with a loop that places all the molecules in nice rows, with the centers of neighboring molecules separated by a distance of 1.5. (Call this constant neighborSpace.) This initialization loop should work for any reasonable values of N and boxWidth. Be sure that all the molecules are entirely inside the box! Test your program for a variety of values of N and pixelsPerUnit. What do you think the program should do if the molecules don't all fit in the box?

Exercise: If you haven't already, introduce a bug in your program so that it tries to access the nonexistent array element x[N]. What happens when you run the program?

Exercise: Fill in the code of your **run** method, making it similar to the **run** method of your **Orbit2** program. A separate method, called **doStep**, will execute one step of the integration algorithm. The **run** method should call **doStep** about 10 times, call **repaint**, and then sleep long enough for the **paint** method to redraw all the

84 Project 7 Molecular Dynamics

molecules. Try 30 milliseconds for the sleep duration. (The optimum sleep duration and number of steps per animation frame will depend on the number of molecules, the speed of your computer, and your personal preferences, so you'll want to come back and fine-tune these constants later.) Write a temporary doStep method that simply moves molecule number 0 by a very small amount, so you should see it move gradually across the screen as the program runs. Compile, test, and debug your program once again.

Coding the Simulation

With your program's basic outline and graphics code in place, you're now ready to write the code for the simulation itself. Please use the Verlet algorithm, with a fixed time step of 0.02. (This is another constant that you may need to change later. In theory it would be better to use a variable time step, but the coding needed to optimize the time step isn't worth the trouble at this stage.)

In your doStep method, be sure to implement the Verlet algorithm in the same way as in the previous project: Update the positions, and update the velocities half-way, using the old acceleration values. Then call a separate method (for clarity, since it will be rather long) called computeAccelerations, which will update the accelerations. Finally, finish updating the velocities using the new accelerations.

The computeAccelerations method must have two parts: one to calculate the accelerations from collisions with the walls of the box, and one to calculate the accelerations from collisions between molecules. The collisions with the walls are potentially trickier, especially if we want infinitely stiff walls that produce instantaneous changes in velocity. Instead, it's easier to make the walls "soft," so the force they apply increases gradually as the edge of a molecule moves farther into the wall. A linear "spring" force works well; here is a code fragment that implements such a force for the vertical walls:

```
for (i=0; i<N; i++) {
    if (x[i] < 0.5) {
        ax[i] = wallStiffness * (0.5 - x[i]);
    } else
        if (x[i] > (boxWidth - 0.5)) {
            ax[i] = wallStiffness * (boxWidth - 0.5 - x[i]);
        } else
            ax[i] = 0.0;
}
```

Here wallStiffness is the "spring constant"; a value of 50 in natural units works pretty well. Notice that since m = 1, the force on a molecule is the same as its acceleration.

Question: Why does the first if statement in this code test whether x[i] is less than 0.5, rather than testing whether x[i] is less than 0?

Exercise: Write the code of doStep, and the code to handle collisions with the walls of the box, as described above. To test your code, give molecule 0 an initial v_x value of 1. You should see it bounce back and forth between the two vertical walls. Be sure to insert similar code for collisions with the horizontal walls.

Exercise: Now add the code to handle molecular interactions. You'll need a double loop over all pairs of molecules:

```
for (i=0; i<N; i++) {
   for (j=0; j<i; j++) {
    // compute forces between molecules i and j
    // and add on to the accelerations of both
   }
}</pre>
```

To compute the force components, use the formulas you worked out at the beginning of this project. Be careful with minus signs. Then compile your code, run it, and enjoy the show!

Optimizing Performance

Ninety-five percent of the time, you shouldn't worry about writing code that will run as fast as possible. It's much more important to make your code simple and easy for a *human* to read and understand. *Your time is more valuable than the computer's!*

However, your doStep and computeAccelerations methods fall in the other five percent. These methods contain loops that execute the same code thousands upon thousands of times. Any effort that you spend speeding up the code inside these loops will be rewarded in proportion.

Exercise: (In this exercise you will estimate how many times the code within these loops is being executed. Please assume that the various constants are set to the values that I've recommended above, even if you've changed them in your program.) A typical molecule in this simulation might have a speed of 1 in natural units. How many units of time will it take this molecule to cross from one side of the box to the other (assuming no collisions along the way)? How many calls to doStep are required for such a one-way trip? During this same time, how many times will the code within each of the loops in doStep be executed? How many times will the code within the double loop in computeAccelerations be executed? How many

86 Project 7 Molecular Dynamics

times will your paint method call fillOval? (Please show your calculations and answers below.)

Here, then, are some tips for optimizing the performance of your MD simulation:

- Don't even think about optimizing performance until after you're sure that your program is working correctly.
- Don't worry about optimizing code that isn't executed at least thousands of times per second.
- The doStep method makes repeated use of the combinations dt/2 and $dt^2/2$. So compute these quantities once and for all in your program's initialization section, storing them in instance variables.
- When you compute the Lennard-Jones force, avoid using the Math.pow and Math.sqrt functions. Since only even powers of r appear in the force, you can work with r^2 (or better, $1/r^2$) instead of r. Cube $1/r^2$ to get $1/r^6$, then square that to get $1/r^{12}$. Use intermediate variables where necessary to avoid repeating calculations.
- To speed up the force calculations even more, don't bother to compute the force between two molecules when they're farther than (say) 3 units apart. (Be sure to test whether they're too far apart in a way that avoids calculating a square root.)
- A brute-force way to speed up the simulation is to increase the value of dt. Naturally, this will also increase the truncation error. Don't try this until later, when you'll have a way of checking whether energy is approximately conserved.
- Although you probably can't do much to speed up the graphics, be aware of the fact that graphics may be a major performance bottleneck. Try adjusting the number of calculation steps per animation frame, and the sleep duration, to see if these constants affect the performance. Remember that the human eye can't follow more than about 30 animation frames per second, and there's no point in redrawing molecules so frequently that they hardly move between one redraw and the next.

Exercise: Spend some time optimizing the performance of your program, and describe the effects in the space below. Which changes seem to make the most difference? After optimizing, how large can you make N and still get reasonably smooth animation?

GUI Controls

Your program still lacks two important features: It doesn't let you control the simulation in any way while it is running, and it doesn't give you any quantitative data to describe what's going on. You could add all sorts of features of both types. The instructions given here describe only the essentials.

First, you need to create a Panel to hold the GUI controls; this Panel can also hold a separate Canvas for text output. So go back and add code to your constructor method to create a new Panel, add it to the bottom (BorderLayout.SOUTH) of your Frame, and set its layout to GridLayout(0,1), so its contents will be stacked vertically.

You can create a separate Canvas for text output by using an anonymous inner class:

```
dataCanvas = new Canvas() {
    public void paint(Graphics g) {
        g.drawString("Hello, world!", 5, 15);
    }
};
```

Add this Canvas to your new Panel; the GridLayout will automatically size it to fill the width of the Frame and give it the same height as the next thing that's added.

That next thing should be yet another Panel, which will hold several Buttons. First, put in a Button to start and stop the simulation. Then put in a pair of Buttons to increase and decrease the system's energy. A good way to do this is to multiply or divide the velocities of all the molecules by 1.1.

Exercise: Add all the GUI features just described to your simulation. Test the buttons to make sure they work.

88 Project 7 Molecular Dynamics

Question: Describe what happens when you continually remove energy from the system. How do the molecules arrange themselves? (Sketch a typical arrangement.) What happens when an isolated molecule hits a large group of molecules?

Question: Describe what happens when you continually add energy to the system.

Exercise: Play with your program some more, perhaps trying different numbers of molecules, changing their sizes in relation to the box, and using the buttons to add and remove energy. Describe at least one other interesting phenomenon that the simulated molecules exhibit.

Data Output

What kind of data should you collect from this simulation? Energy is always a good place to start.

To compute the kinetic energy you might as well use a new method; please call it computeData. In it, use yet another for loop to add up the kinetic energies of all the molecules and store the sum in an instance variable for later output. Call this method from your run method every time you call repaint (that is, about once for every ten calls to doStep). To compute the potential energy, it's easiest to add a few lines of code to your **computeAccelerations** method. Be sure to include both the Lennard-Jones intermolecular potential energy and the "spring" potential energy associated with interactions with the walls $(\frac{1}{2}kx^2)$, where k is the spring constant and x is the amount of compression). To be absolutely precise, you should add a small constant to the Lennard-Jones energy to compensate for the fact that you're setting the force to zero beyond a certain cutoff distance.

Now modify the paint method of dataCanvas to display the kinetic energy, potential energy, and total energy, to four decimal places each. You'll also need to call dataCanvas.repaint from your computeData method. Because the paint method is actually called from a separate thread, you might find that it sometimes catches the energy values while your calculation thread is still in the middle of computing them. You can solve this problem by using local variables for the computations and copying the results to the instance variables only when the computations are complete.

Exercise: Add the code just described to your program, and check that the energy values are reasonable. About how much does the total energy fluctuate? (Because the energy can be positive or negative, please describe the fluctuation as an absolute amount, not as a percentage.) What about the kinetic and potential energies? What happens if you increase the value of dt?

Another variable of interest is the temperature of the system. For a collection of N classical particles, the equipartition theorem tells us that each energy term that is quadratic in a coordinate or velocity (that is, each "degree of freedom") has an *average* energy of $\frac{1}{2}k_BT$, where k_B is Boltzmann's constant and T is the temperature. In two dimensions each molecule has only two translational degrees of freedom (v_x and v_y), so the average kinetic energy per molecule should be $2 \cdot \frac{1}{2}k_BT$. Thus, in natural units where $k_B = 1$, the temperature is precisely equal to the average kinetic energy per molecule.

In a macroscopic system with something like 10^{23} particles, the average kinetic energy per molecule wouldn't fluctuate measurably over time. In your much smaller system, the kinetic energy fluctuates quite a bit. To get a good value of the temperature, therefore, you need to average not only over all the molecules but also over a fairly long time period. This requires just a little more computation.

90 Project 7 Molecular Dynamics

Exercise: Add code to your program to compute and display the average temperature of the system. You'll need a variable to accumulate the sum of the temperatures computed at many different times, and another variable to count how many values have been added into this sum so far. Increment these variables in your compute-Data method, then divide to get the average when you display the temperature in the paint method of dataCanvas. You'll need to reset both of these variables whenever energy is added to or removed from the system. It's also a good idea to add a button that manually resets the variables. Test your code and check that the results are reasonable.

In a similar way, you can also compute and display the average pressure of the system. In two dimensions, pressure is defined as force per unit *length*.

Exercise: Add code to your computeAccelerations method to compute the total outward force exerted on the walls of the box, and hence the instantaneous pressure of the system. Use this result to compute and display the average pressure over time, just as you did for the temperature. (If there isn't enough space in dataCanvas, you may omit the kinetic and potential energies from the display.) To check that your results are reasonable, recall that in the limit of low density (where the distance between molecules is large compared to their sizes), the pressure is given by the ideal gas law: $P = Nk_BT/V$. (In two dimensions, V is actually an area.) Set N and pixelsPerUnit so the density of your system is reasonably low, add energy until the molecules are behaving like a gas, and compare the pressure of your system to the prediction of the ideal gas law.

Exercise: Add one more Button to your program, to print the temperature, pressure, and total energy values in your command window when it is pressed. This will allow you to record data for later analysis. Check that it works.

Congratulations! Your molecular dynamics simulation program is finished. Now would be a good time to clean up the code and add more comments if necessary. Then get ready to use your program for a systematic numerical "experiment." You'll be studying a system of a fixed number of molecules, in a fixed volume, over a wide range of temperatures. The number of molecules should be at least 100, but feel free to use more if your computer is fast enough. The volume should be large enough to give the molecules plenty of space: at least 10 units of volume per molecule. Before you start to take data, fine-tune your program and check that everything is working correctly.

Exercise: Setting N to at least 100 and the volume to at least 10 units per molecule, use your simulation to determine the energy and pressure as functions of temperature over a wide range of temperatures, from about 0.0001 to 1.0, with enough

intermediate points to produce smooth graphs. Before recording each data point, be sure to let the system equilibrate long enough to give reasonably stable values of the temperature and pressure. Also please make some notes to describe the system's appearance at various temperatures. Copy your program's output into a spreadsheet and use the spreadsheet to plot graphs of E vs. T and P vs. T. Print the data and graphs and attach them to your lab report. Discuss these graphs in as much detail as you can, correlating them to your verbal notes. How does the pressure compare to the prediction of the ideal gas law? How does the system's heat capacity behave in the various temperature regions, and how does this behavior relate to what you learned about heat capacities in introductory physics?

92 Project 7 Molecular Dynamics

[This page intentionally left blank.]

Project 8: Random Processes

Many physical systems, such as the molecules modeled in the previous project, are so chaotic that the motions might as well be random. But in that case, why should we even bother to calculate the actual trajectories? For many purposes we can get away with substituting fictitious "random" behavior for the actual behavior of a system. Randomness is also an intrinsic property of quantum mechanical measurements, for instance, of the time when a nucleus decays.

In this project you will write several short computer programs that use computergenerated "random" numbers to model simple physical systems. Programs that use random numbers to choose among various possible outcomes are called **Monte Carlo simulations**, after the famous European gambling resort.

Computer-Generated "Random" Numbers

Java comes with a built-in function called Math.random(), which returns a "random" number (of type double) between 0 and 1 each time it is called. Before using these numbers in physics simulations, it's a good idea to get some feel for what they're like.

Exercise: Write a short Java program called RandomTest that prints out the values of 20 successive numbers returned by Math.random(). Feed the numbers directly to println, without any rounding. Do the numbers appear random to you? How can you tell?

Question: If you run your **RandomTest** program a second time, do you get the same sequence of numbers, or a different sequence?

Often we'll want random numbers that span a range other than 0 to 1; a common need is to choose a random integer within a certain range. You may be interested to know that although Math.random can (very rarely) return a value of exactly

94 Project 8 Random Processes

zero, it will never return a value of exactly 1. This means that if you multiply a Math.random result by N and then round down, you'll get a random integer between 0 and N - 1, inclusive.

Exercise: Add code to your RandomTest program (without deleting the code that's already there) to obtain and print the values of 50 random numbers (digits) in the range 0 to 9 inclusive. Count the number of times each digit occurs, and write the results of your counts below.

Another common need is to generate random points in some region of space, in two or more dimensions.

Exercise: Add code to your RandomTest program (without deleting the code that's already there) to generate 1000 random real-number (x, y) pairs and display them in a Plot. The plot's gridlines should divide it into ten equal-size units in each direction. Print the plot and include it with your lab report.

Question: How many points, on average, do you expect to appear in each of the 100 grid squares of your plot? What are the approximate maximum and minimum numbers of points that actually appear within the squares? (Please circle the corresponding squares on your printout.)

Now that you've gotten a feel for what Math.random does, here's the bad news: The numbers that it returns are not actually random. In fact, each of these numbers is computed from the previous one in a completely deterministic way. (The computer's clock is used to get the sequence started, so you get a different sequence each time you run the program. If you ever want to get the same sequence every time, you can do so using the java.util.Random class.) But the function used to generate each number from the last is chaotic, so that even a tiny change in the input produces a substantial change in the output. The function also has the property that over the long term, it generates numbers that are distributed evenly throughout the interval from 0 to 1. But the function is not guaranteed to return numbers that are sufficiently random for all purposes. In particular, despite the appearance of your plot, this function is not guaranteed to generate evenly distributed points in higher-dimensional spaces. In recognition of these imperfections, computer-generated "random" numbers are often called **pseudo-random numbers**.

In this course we'll ignore this potential pitfall and assume that the numbers returned by Math.random are sufficiently random for our purposes. For researchquality work, however, you should always use a better pseudo-random number generator than Math.random. For an excellent discussion of this problem and several examples of much better generators (coded in C++), I recommend the book Numerical Recipes by Press, et al.

Expansion of a Gas

Consider the situation shown in figure 8.1: A box is divided into two sections of equal size, separated by a partition. On one side of the partition is a gas of n molecules; on the other side is a vacuum. We then puncture the partition and allow the molecules to pass from one side to the other. What happens to the number of molecules on each side as a function of time?

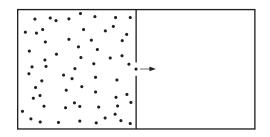


Figure 8.1. All the molecules start on the left side of the partition. What happens when we puncture the partition and let the molecules travel back and forth?

To answer this question we *could* write a full-blown molecular dynamics simulation. Or we could just recognize that for our purposes the motions will be essentially random, and say that each of the n molecules has an equal probability of switching sides during any short time interval.

Exercise: Write a program called TwoBoxes to simulate the behavior of this system. Use the variables n for the total number of molecules and nLeft for the number that are currently on the left side of the box. In each step of "time," choose a molecule at random and move it to the other side of the box (by increasing or decreasing nLeft by 1). To determine which side your randomly chosen molecule is on, you can simply assume that the first nLeft molecules are on the left and the rest are on the right. The output of your program should be a plot of nLeft vs. "time." Turn in a printout of this plot for n = 1000.

Question: Why did I put the word "time" in quotes? Can you use this simulation to determine the amount of time (in seconds) before about half the molecules will be on each side of the box?

Notice from your simulation that even after roughly half the molecules are on each side of the box, the number on the left side fluctuates significantly. It's interesting to study these fluctuations in a little more detail.

96 Project 8 Random Processes

Exercise: To study fluctuations around equilibrium, modify your simulation to start with exactly half of the molecules on each side. Run the simulation for n equal to 10, 100, and 1000, and in each case, estimate (from the graph) the typical amount by which nLeft fluctuates away from n/2. Write down these estimated numbers and discuss them briefly.

Exercise: Now modify your simulation to create a histogram plot, with nLeft on the horizontal axis and the number of times that that value of nLeft occurs on the vertical axis. To store the histogram you'll need an array, whose index runs over all possible values of nLeft. After each step of the simulation, add 1 to the appropriate element of this array. To display the histogram, keep track of which array entry is the largest, and use this value to determine the vertical scale of a new Plot. Run your simulation again for n equal to 10, 100, and 1000, and briefly describe the appearance of the histogram plots.

If you've studied a little probability theory, you may have already realized that over the long term, the probabilities of the various possible **nLeft** values should be given by the **binomial distribution**:

Probability =
$$\frac{1}{2^n} \frac{n!}{(n_{\text{left}}!)(n-n_{\text{left}})!} = \frac{1}{2^n} \frac{n!}{(n_{\text{left}}!)(n_{\text{right}}!)}.$$
 (8.1)

For a sufficiently long run, therefore, the values in your histogram should be approximately equal to this formula times the number of steps in the simulation.

Exercise: Write a static Java function that takes an integer parameter and returns the factorial of that integer as a **double**. Test your function by printing (to your screen) its results for a few different inputs, and write the results below.

Question: Why is it better to have the function return a double than an int or a long?

Question: What is the largest integer whose factorial fits into a double? What happens if you try to compute the factorial of a larger integer?

The previous exercise may have been your first experience with numerical **overflow**—attempting to compute a number that is too large to fit into the variable type being used. In Java, overflow of an integer variable type (including long) can give nonsensical results. For variables of type double the result is easy to understand, but not very useful. Because the largest allowed double is far larger than the number of atoms in the observable universe, it is unusual for scientific computations of observable quantities to overflow as long as double variables are used. The main exceptions are computations of combinatoric functions such as factorials, and exponential functions when they are used in similar ways.

To prevent overflow in the following exercise, you can simply limit the value of n to 100. (With some fancy programming you could actually compute the binomial distribution for somewhat larger values of n, but don't bother.)

Exercise: Write a static Java function to compute and return the value of the binomial distribution as a function of **n** and **nLeft**. (Hint: This function should call the factorial function that you just wrote.) Test your binomial function in a couple of simple cases, and write the results below.

Exercise: Now use your binomial distribution function to plot the "theoretical" (long-term average) distribution of nLeft values on your histogram plot. Use a different color and/or shape to distinguish the theoretical distribution from the actual distribution. Run the simulation for n = 10 and 100, with enough "time" steps to produce reasonably good agreement between the actual and theoretical distributions. Print the plot for n = 100 and label it appropriately.

98 Project 8 Random Processes

Random Walks

Next let us consider the erratic motion of a single microscopic particle. The particle could be a gas molecule, a dust grain suspended in a fluid, or a conduction electron in a copper wire. In all these cases the particle collides frequently with neighboring particles and therefore moves back and forth in a way that appears mostly or entirely random. Such motion is referred to as a **random walk**.

The simplest example of a random walk is in one dimension, with steps all the same size, each step equally likely to be one way or the other. This example is mathematically similar to the previous simulation of a gas in two boxes.

Exercise: Write a new Java program called RandomWalk.java to model a random walk in the x direction. During each time step the particle should move one unit of distance, with a 50-50 chance of moving in the positive or negative direction. Start the particle at x = 0. Plot a graph of the particle's position as a function of time for some fixed number of steps, then repeat the simulation several (20 or more) times over, plotting all the results on the same graph. Describe the results in the space below, and turn in a printed graph from a typical run of the program. As you increase the number of steps in the walk, what happens to the typical net distance traveled?

Exercise: To quantify your answer to the previous question, modify your program to calculate the root-mean-square (rms) net distance traveled by all of your random walkers. This is the square root of the average of the squares of the final positions. Run your program for several different values of the number of steps in the walk, and record the results below. Can you guess an approximate formula for the rms displacement as a function of the number of steps?

Question: Why not simply compute the *average* net displacement of the random walkers, rather than the rms displacement?

Nuclear Decay

A classic example of random behavior is nuclear decay. Each radioactive isotope has a certain intrinsic probability of decaying per unit time. As far as we can tell, the time when any particular nucleus decays is truly random.

Exercise: Write a new Java program called **Decay.java** to model the decay of a collection of n radioactive nuclei of the same isotope. Let the probability of each nucleus decaying per unit time be 0.001. During each time step, for each remaining nucleus, generate a random number to determine whether that nucleus decays. Run the simulation long enough for at least 90% of the nuclei to decay. The output of the program should be a graph of n vs. time. Run your program with n initially equal to 100, and keep a printout of the graph. Repeat for n = 10,000.

Exercise: Use one of your graphs to determine the approximate half-life of this isotope. Mark this value on the graph and write the result below.

Question: When n = 10,000, how many nuclei do you expect to decay (on average) during the very first time interval of this simulation? Explain.

Exercise: Write a new simulation program (call it DecayDist) to answer the previous question in more detail. The program should simulate only the very first time interval of the decay of 10,000 nuclei, where each nucleus has a probability to decay of 0.001. However, the program should repeat this one-time-interval "experiment" a hundred or more times, and plot a histogram of the results, so you can see in detail how the actual number of decays tends to fluctuate around the average expected number.

After a sufficiently large number of trials, the histogram plotted by your Decay-Dist program should take on a well-defined shape. And as you might guess, there is a fairly simple formula for this shape. If the average expected number of decays is λ , then the probability of actually getting k decays is given by the **Poisson** distribution:

Probability =
$$\frac{1}{k!} \lambda^k e^{-\lambda}$$
. (8.2)

100 Project 8 Random Processes

Exercise: Using pencil and paper (in the space below), sum the Poisson distribution over all possible k values, and interpret the result.

Exercise: Add a static function to your **DecayDist** program to compute the Poisson distribution, and use it to plot the "theoretical" shape of your histogram along with the actual histogram. Once everything works, print a graph of the results and turn it in with your lab report.

Project 9: The Ising Model

Systems at Fixed Temperature

As you saw in Project 7, physical systems tend to behave more randomly at higher temperatures. This fact is summarized in an amazingly simple formula, which says that at temperature T, the probability of finding a system in any particular state with energy E is proportional to the so-called **Boltzmann factor**:

Probability
$$\propto e^{-E/k_B T}$$
. (9.1)

The constant of proportionality depends on T but is the same for all states, so a higher-energy state always has a lower probability than a lower-energy state. Notice, however, that the probabilities vary much more dramatically at low T than at high T. In the limit $T \rightarrow 0$, even the tiniest increase in E causes a dramatic decrease in the probability, so the system is virtually certain to be in whatever state has the very lowest energy (the **ground state**). At sufficiently high T, on the other hand, there will be a huge number of states with probabilities that are not much less than that of the ground state, so the system appears to behave much more randomly as it makes transitions among all these states.

(This discussion assumes that our system is free to exchange energy with its surroundings—otherwise its energy would be fixed, and all possible states with that energy would have the same probability. In fact, the very concept of holding a system at a fixed temperature requires the assumption that the system can exchange energy with a "reservoir" that is so large that its temperature is unaffected by these energy exchanges.)

The Boltzmann factor is the basis of an extremely useful algorithm for Monte Carlo simulation of thermodynamic systems. Suppose that a simulated system is currently in a state with energy E_1 . We first use pseudo-random numbers to choose some other state (state 2) that differs from the current state in some random way (for example, by moving a randomly chosen molecule by a random displacement). Suppose that state 2 has energy E_2 . If $E_2 < E_1$, we go ahead and put the system into state 2. On the other hand, if $E_2 > E_1$, we use another pseudo-random number to decide whether to put the system into state 2, with probability

$$\frac{e^{-E_2/k_BT}}{e^{-E_1/k_BT}} = e^{-(E_2 - E_1)/k_BT}.$$
(9.2)

Either way, we then repeat the process many times over, long enough for the system to sample a representative collection of states that are typical for our chosen value of T. This algorithm is called the **Metropolis algorithm**, after the first author of the 1953 article in which it was originally described.

102 Project 9 The Ising Model

Exercise: Consider a very simple system with only two possible states, 1 and 2. Suppose that $E_2 > E_1$, and that the ratio of Boltzmann factors in equation 9.2 is equal to 1/2. Describe in detail how a simulation using the Metropolis algorithm might progress, and write down a series expression for the average number of time steps that it takes before the system makes a transition from state 1 to state 2. To sum the series, use the formula $\sum_{n=1}^{\infty} n r^{n-1} = (1-r)^{-2}$. Thus, explain why over the long term, the system will behave as predicted by the Boltzmann factor (equation 9.1).

The preceding exercise can easily be generalized to systems with arbitrary transition probabilities and arbitrarily many states; again the Metropolis algorithm yields the correct probabilities, over the long term. However, most realistic systems have so *many* possible states that to sample all of them, even with the fastest imaginable computers, would require far longer than the age of the known universe. Running the Metropolis algorithm for relatively shorter time periods will sample only a tiny fraction of the available states, and we have no guarantee that these will be truly representative of those that would occur in the real world. Still, the Metropolis algorithm often works remarkably well.

Ferromagnetism

We could apply the Metropolis algorithm to a system of interacting molecules, such as that studied in Project 7. For simplicity and variety, however, we'll apply it to a completely different system: A collection of magnetic particles that are fixed in space but free to orient themselves in different directions. Most types of atoms are magnetic, but only a few materials, such as iron, have the additional property that the atoms have a strong tendency at room temperature to align themselves parallel to their neighbors. This tendency can cause the magnetic effects to combine rather than cancel, resulting in a **ferromagnet**: a material with large-scale magnetic behavior.

In the simplest possible model of a ferromagnet, each magnetic particle has only two possible orientations, usually called "up" and "down". (This model therefore mimics some aspects of the behavior of spin-1/2 particles.) To give the particles a tendency to orient parallel to their neighbors, we assign an energy of -1 unit for every neighboring pair with the same orientation, and an energy of +1 unit for every neighboring pair with opposite orientations. This model is called the **Ising model** after Ernst Ising, who studied it (without computers) as a graduate student in the 1920's.

The magnetic particles in an Ising model can be arranged in a variety of ways. The simplest option is to line them up in a one-dimensional row, so each atom has only two neighbors. This arrangement turns out to be rather trivial, however, so we'll study the next-simplest arrangement: a two-dimensional square lattice, in which each particle (except those on the edges) has four nearest neighbors. Figure 9.1 shows a small example of such a system.

¥	¥	1	¥
	1	↑	1
¥	1	1	1
¥	¥	¥	1

Figure 9.1. One possible state of an Ising model on a 4×4 square lattice.

Exercise: For the particular state shown in Figure 9.1, count the number of neighboring particles that are aligned parallel and the number that are aligned antiparallel. How many units of energy does this system have?

Exercise: Consider a somewhat larger Ising model, with 100 magnetic particles. How many possible states does this system have? If a computer simulation can sample a billion states per second, about how long would it take to sample all of the possible states of this system?

For the Ising model it is especially simple to implement the Metropolis algorithm described above. We can start with the system in any state at all, and during each iteration of the algorithm, simply choose a lattice site at random and consider whether to flip the particle at that site to the opposite orientation. The algorithm should then steer the system toward states that are typical for the assumed value of the temperature. 104 Project 9 The Ising Model

Simulation Design

Now let's think about how to implement a computer simulation of the Ising model.

To store the current state of the square lattice of particles you can use a *two*dimensional array of integers. A two-dimensional array is basically just an array of arrays, and the syntax for creating and accessing such an array is a natural extension of the one-dimensional case:

int[][] s;	<pre>// declare the array of lattice sites</pre>
<pre>s = new int[size][size];</pre>	<pre>// allocate memory for the array</pre>
s[i][j] = 1;	<pre>// set the value of an array element</pre>

Here I'm assuming that size is an integer variable holding the width of the square lattice; the array indices will range from 0 to size-1. The symbol s could stand for *site* or *spin*. Please use the convention that an s value of 1 corresponds to the "up" orientation, while an s value of -1 corresponds to the "down" orientation. Then you can compute the energy associated with any neighboring pair as simply minus the product of their s values.

To display the current state of the lattice you should use an extended Canvas, divided into squares to represent the lattice sites. Fill the squares with one color to represent up and another color to represent down. You'll want to update the display continually as the simulation runs, so you'll need to use a separate thread for the computation. In many ways, the structure of this simulation will be similar to that of your molecular dynamics simulation (Project 7).

Exercise: Create a new Java source file called Ising.java, and type in the basic outline of a program to simulate a two-dimensional Ising model (on a square lattice) using an extended Canvas and a separate thread to run the simulation. Use a lattice size of 20×20 for now, but plan to increase the size later. Include code to initialize the array of lattice sites so they all point up, as well as a paint method that will display the lattice correctly whatever its state. Don't worry yet about putting any physics into the simulation—just focus on getting the graphics to work. Test your code to make sure it does work, and describe below how you went about testing it.

Exercise: Before calling your paint method, Java normally fills the entire Canvas with its background color (which you can set using setBackground). In this case, however, your colored squares probably cover the entire Canvas, so the background is superfluous. To eliminate this inefficiency (which can also cause screen flicker on some systems), please add the following method to your program:

```
public void update(Graphics g) {
    paint(g);
}
```

Whenever your Canvas needs to be painted, Java actually calls update. The default update method draws the background before calling paint, but this version overrides the default to skip drawing the background.

Implementing the Metropolis Algorithm

Your program's **run** method should execute a few hundred iterations of the Metropolis algorithm, then call **repaint()** to update the screen display. Each iteration of the algorithm, in turn, consists of the following steps:

- 1. Choose a lattice site at random.
- 2. Compute the hypothetical change in the system's energy if the particle at this site were to flip its orientation.
- 3. If the hypothetical change in energy is negative or zero, go ahead and flip that particle's orientation. Otherwise, generate a random number between 0 and 1 and compare this to the Boltzmann ratio (equation 9.2) to decide whether to flip the particle.

To implement step 3, you can use Java's or operator (||):

```
if ((eChange <= 0) || (Math.random() < Math.exp(-eChange/T)) {
    s[i][j] *= -1;
}</pre>
```

When the condition before the || operator is true, Java doesn't bother to test the condition after the operator—so the calls to Math.random and Math.exp will occur only when necessary. Notice also that in this expression I've set Boltzmann's constant equal to 1, so T is measured in the same units as the energy.

Exercise: Fill in the code of your program's **run** method to implement the Metropolis algorithm as just described. Use a separate function (call it deltaE(i,j)) to compute and return the hypothetical energy change (step 2), given the indices of the chosen lattice site. Finish coding the rest of the algorithm, but don't fill in the code for the deltaE function yet.

Exercise: The main complication in the deltaE function will be dealing with the cases where the chosen site is on an edge of the lattice. Pretend, for now, that you can assume the site *isn't* on an edge. In the space below, write down appropriate code for the deltaE function. Be sure to check a few explicit cases to be sure that your code would work as intended.

106 Project 9 The Ising Model

Exercise: When the chosen lattice site *is* on an edge, you'll need to substitute a zero into your formula for the missing neighbor's **s** value. There's no particularly elegant way to do this—it'll require a separate **if-else** statement for each of the four edges. With this in mind, fill in the code of your **deltaE** function, so that it works correctly whether or not the chosen lattice site is on an edge. Test this function, and describe in the space below how you tested it.

Exercise: You should now have a working simulation of the two-dimensional Ising model. Run it for several different values of the temperature, keeping the lattice size at 20×20 for now. Describe the results (in some detail) in the space below.

Controls and Data Output

By now you've probably decided that your simulation needs a scrollbar to control the temperature; you're absolutely right. But as long as you're adding this scrollbar, it's a good idea to design a layout that leaves room for some other useful things: a small Canvas for some text output, and a row of buttons to control the simulation. You can use a layout similar to the one you used in your molecular dynamics simulation, with an extra row for the scrollbar.

Exercise: Add a Panel at the bottom of your Frame to hold the items just described: a small Canvas for some text output, a Panel to hold a row of buttons, and a DoubleScroller to control the temperature. Set up the DoubleScroller so you can adjust the temperature from 0.01 to 10.00 (in natural units), in increments of 0.01. Also add a button to start and stop the simulation. Test these controls to be sure they work.

What sorts of data should your simulation collect? There are two primary variables of interest: the total energy of the system and its total magnetization (which in natural units is simply the sum of all the \mathbf{s} values). Let's call these variables E and M.

Exercise: Add code to your simulation to compute the current values of E and M, and also to compute the averages of these variables over the course of the simulation. Have the program display the current values and the averages (to a suitable number of decimal places) in the small **Canvas** that you recently created. Also add a Reset button to reset the averages, and a Record button to print the current and average values, along with the temperature, in your command window. Label the methods that compute and display the data as **synchronized**, where necessary, to make sure the program won't try to display quantities that are in the middle of being calculated. Test all of these features to be sure they work. In particular, check that the E and M values are computed correctly in some appropriate special cases. Describe your tests in the space below.

Larger Lattices and Graphics Performance

Exercise: Now that your program is essentially complete, try gradually increasing the lattice size. At some point you'll find that the animation slows down and loses its smoothness, because it takes too long for the **paint** method to draw so many rectangles. At what lattice size does this effect become noticeable?

To improve your program's graphics performance for large lattices, you can use a common technique called **double buffering**. The idea is to store a copy of the lattice image in an off-screen location, and update this copy from your computation thread whenever a lattice site changes its color. Then all your **paint** method has to do is copy the entire image to the screen—which is much faster than redrawing the whole image from scratch.

The Java AWT package includes an Image class for storing off-screen images. To create such an image, you'd think you would call this class's constructor method, but for some reason you instead have to say something like

```
Image offScreenImage = createImage(width,height);
```

The createImage method actually belongs to the Component class, so you're basically associating the image with some other AWT component. I have no idea why this is necessary, but it's certainly easy enough to do, especially when the class that creates the image is itself a descendant of Component. The parameters of createImage are the width and height of the image in pixels, which should match the dimensions of your on-screen Canvas. 108 Project 9 The Ising Model

Once you've created an off-screen image, you can get its **Graphics** object with the instruction

Graphics offScreenGraphics = offScreenImage.getGraphics();

Now you can draw anything you want to this graphics object, for example,

offScreenGraphics.fillRect(x1,y1,dx,dy);

To copy the off-screen image to the screen, your **paint** method merely has to execute a single line of code:

```
g.drawImage(offScreenImage,0,0,width,height,this);
```

Here the width and height parameters are again the pixel dimensions of the image. Don't worry about the purpose of the final parameter, "this". The only other subtlety is that you don't want paint to be called while your computation thread is in the middle of drawing to the off-screen image, so you should put the fillRect statement into its own method, and label both this method and your paint method as synchronized.

If you'd like to see an example of how to implement double buffering, take a look at the source code of the Plot class. You'll see that the addPoint method actually plots the point on an off-screen image, then calls repaint to trigger a call to paint in the graphics thread. The paint method then simply calls drawImage to copy the off-screen image to the screen.

Exercise: Implement double buffering in your Ising program, as just described. Then increase the lattice size to 100×100 (or more). You'll probably also want to increase the number of Monte Carlo steps between calls to **repaint** to a few thousand. Spend some time fine-tuning the simulation's performance; the faster your computer, the larger you can make the lattice without having to wait too long for it to equilibrate to different temperature settings. Once you're pleased with the simulation's performance, run it at several different temperatures and observe the system's behavior. In the space below, describe any further observations and details that you may not have noticed when the lattice was smaller.

The Ferromagnetic Phase Transition

By now you should have noticed that above a certain temperature, your twodimensional Ising lattice does not acquire (or retain) a predominant magnetization on large scales. That temperature is called the **critical temperature**. The transition from a magnetized state at low temperatures to an unmagnetized state at high temperatures is a type of phase transformation.

Question: In the natural units used by your program, what is the approximate value of the critical temperature for the two-dimensional Ising model?

Exercise: In a strong ferromagnet such as iron, the typical interaction energies between neighboring particles are on the order of 0.01 eV (plus for parallel alignments, minus for antiparallel). For this value of the interaction energy, what would be the critical temperature of a two-dimensional Ising system, in kelvin? (Please take this exercise with a grain of salt—real ferromagnets are quite a bit more complicated than the Ising model. For comparison, however, the actual critical temperature of iron is 1043 K.)

Exercise: Use your simulation to record the average energy and magnetization of the Ising system at a variety of temperatures, focusing especially on temperatures near the critical temperature. Use as large a lattice as you can, without having to wait too long for the system to equilibrate at each temperature. Copy your data into a spreadsheet and plot the energy and magnetization as a function of temperature. Also compute the heat capacity $(\Delta E/\Delta T)$ at each temperature and plot it. Print out your plots and write enough comments on them to demonstrate your understanding of the behavior of the Ising model.

110 Project 9 The Ising Model

[This page intentionally left blank.]

Java Language Words

This table lists all of Java's 52 "reserved words," which are part of the language and therefore may not be used as variable, method, or class names. I've tried to group them into categories, separating those that are used in this manual from those that aren't.

Used in this manual	Not used in this manual
Operators	
new	instanceof
Special values	
true false this super	null
Flow control statements	
if else while for return	switch case default do break continue
Primitive variable types	
int long double boolean	byte short float char
Class declaration modifiers	
class extends implements	interface
Other declaration modifiers	
static public void synchronized	final transient volatile private protected abstract native strictfp
Error handling	
try catch	finally throw throws assert
Package management	
import	package
Obsolete	
	goto const

Java Classes Used in This Manual

This table summarizes the built-in Java classes and interfaces used by the projects in this manual. A few of them are used only indirectly, but are still good to know about. For systematic documentation of all the built-in Java classes, see the Java API Specification at http://java.sun.com/reference/api/index.html.

package java.lang fundamental language features	
Object superclass of all other classes	
Systemmiscellaneous utility functionsouta PrintStream that writes to the standard output windowexit(0)quit the program in good standing	
String a sequence of text characters	
Math commonly used math constants and functions (See the Quick Reference Sheet for details.)	
Double utilities for manipulating double values parseDouble(String) converts the String to a double	
Integer utilities for manipulating int values parseInt(String) converts the String to an int	
<pre>Thread a separate mini-process that runs in parallel with others Thread(Runnable) creates a Thread to run the object specified start() starts the thread sleep(long) sleeps for the specified number of milliseconds</pre>	
Runnableinterface for objects passed to Threadrun()the method that is called by Thread.start	
ArrayIndexOutOfBoundsException self-explanatory	
NumberFormatException thrown by Double.parseDouble	
InterruptedException thrown by Thread.sleep	
package java.io input and output	
PrintStreamsends text to some locationprint(String)prints whatever you give itprintln(String)same as print but adds a carriage return	

text manipulation package java.text DecimalFormat handles formatting of real numbers DecimalFormat(String) constructor, with specified pattern format(double) formats the double according to the pattern package java.awt abstract windowing toolkit superclass of Container, Canvas, Button, Label, Scrollbar Component sets the width and height in pixels setSize(int,int) setBackground(Color) sets the background color setVisible(boolean) shows or hides the Component paint(Graphics) override this method to draw stuff draws background then calls paint update(Graphics) tells the AWT thread to call update repaint() createImage(int,int) returns an Image with specified width, height Container superclass of Frame, Panel adds the Component to the Container add(Component) add(Component, int) adds it at specified location, shifting others specifies location in BorderLayout add(Component,Object) specify how to arrange the contents setLayout(LayoutManager) a framed window, with BorderLayout by default Frame Frame(String) creates a frame with the specified title sizes the Frame so everything just fits pack() addWindowListener(WindowListener) respond to window events an invisible Container, with FlowLayout by default Panel Panel() creates a new Panel Canvas a rectangular space on the screen (meant to be extended) Canvas() creates a new Canvas a push-button that causes some immediate action Button Button(String) creates a new Button setLabel(String) changes the text displayed on the Button addActionListener(ActionListener) makes the Button active a static text label Label a scrollbar with arrows and a slider Scrollbar provides methods for drawing lines, shapes, and text Graphics (See the Quick Reference Sheet for the most commonly used methods.) Image an off-screen image for double-buffering getGraphics() returns the Graphics object of this Image Color stores the RGB values for any possible screen color Color(int, int, int) creates a new color, with R,G,B from 0 to 255 LayoutManager interface implemented by the layouts listed below

114 Java Classes Used in This Manual

FlowLayout arranges contents like words on a page

- GridLayout arranges contents in rows and columns, all the same size GridLayout(int,int) constructor with number of rows, columns
- BorderLayout puts one Component in center, others around edges NORTH, SOUTH, EAST, WEST, CENTER locations to use in add

package java.awt.event classes for handling user-input events

- ActionListener interface for responding to Button presses ActionPerformed(ActionEvent) called when Button is pressed
- ActionEvent contains detailed information about the event
- WindowListener interface for handling window events
- WindowAdapter minimal implementation of WindowListener windowClosing(WindowEvent) called when window is closed
- WindowEvent contains detailed information about the window event

- **algorithm.** A step-by-step procedure for carrying out a task, specified in sufficient detail for a computer to carry out the procedure.
- anonymous inner class. A nameless "mini-class" defined within curly braces as part of a **new** operation, which either extends or implements the class that is being instantiated. A common use of anonymous inner classes is to implement event-listener interfaces for responding to GUI input events.
- **API.** Application Program Interface, a term used to describe any library of classes and/or functions for performing common tasks—especially for interacting with low-level operating system software. The standard Java class libraries are an example.
- Apple. A computer hardware and software manufacturer based in Cupertino, CA.
- **applet.** A Java program that is normally downloaded as part of a web page and run under the control of a web browser. The initial popularity of Java was mostly due to the ability to create applets.
- **application.** A newer name for a computer program, meant to emphasize that the program can be applied to do something useful.
- **array.** A list of values, all of the same type. In Java, array elements are numbered starting with 0, and are referenced using square brackets, for example, x[0].
- **AWT.** The Java Abstract Windowing Toolkit (java.awt), a package of classes for graphics and graphical user interfaces. Some of the most important AWT classes are Frame, Panel, Button, Canvas, Graphics, and Color. Newer versions of Java also include a more sophisticated GUI package called Swing.
- **Basic.** A programming language originally developed for instructional use at Dartmouth College in the 1960's, now available in several commercial versions and widely used by students and hobbyists. The name is an acronym for Beginner's All-purpose Symbolic Instruction Code.
- **bit.** The smallest possible unit of information stored or transmitted in a computer, having two possible values that are often interpreted as 0 and 1. The word is a shortening of "binary digit."
- block. A chunk of code enclosed in a set of curly braces.
- boolean. A Java primitive variable type that can have just two possible values, true and false. The if, while, and for constructions all require boolean values for the test being performed. (Often these values are computed by comparison operators such as == and <, without being stored in a variable.)</pre>
- **bug.** A mistake in a computer program; also sometimes called a *feature*. (The name comes from an incident in the early days of computers when an insect caused a hardware failure inside a large computer.)

- **byte.** A small chunk of computer memory (or disk storage) equal to eight bits, capable of storing $2^8 = 256$ different numbers or characters. It has long been standard practice to store each character of a text file in one byte of memory, although the use of two bytes for this purpose is becoming more common as more attempts are being made to accommodate international character sets.
- **bytecode.** A computer language that is similar to machine language but not specific to any particular machine. Java .class files are in bytecode format, which makes them portable between different types of computers but requires that a bytecode interpreter (the Java Virtual Machine) be used to execute them.
- C. A programming language developed at Bell Laboratories in the early 1970's, originally intended for low-level system programming but also widely used for application programming. Much of the basic notation of C is now used by other languages including Java. C was originally the successor to a long-forgotten language called B.
- C++. A widely used extension of the C programming language that includes classes, objects, and inheritance. The name is intended to imply that C++ is one better than C.
- call. To invoke, or transfer control to, a method.
- **cast.** A data conversion from one type to another, specified in Java (like in C and C++) by preceding a variable or expression with the name of the desired type in parentheses. When a floating-point number is cast to an integer, it is rounded toward zero. Casting of numerical data types can, of course, result in out-of-range values, which are normally mapped to the maximum or minimum allowed value for the resulting type. Java also allows casting between superclasses and subclasses.
- class. A discrete, relatively self-contained, chunk of Java code, stored in a single file. Each class has its own variables to hold data, and its own methods to manipulate that data. A Java program normally consists of many different classes, interacting with each other. Each class (with a few exceptions such as System) also defines a variable type that can be instantiated using the new operator to create objects of that type.
- class variable. A static variable that is declared outside of a class's methods, and is therefore accessible to all those methods, and common to all instances of the class. Compare local variable and instance variable.
- **code.** An instruction or set of instructions for a computer, in either source or compiled format.
- **command.** An instruction that you give to a computer, normally meant to be carried out immediately. Newer operating systems tend to use mouse-driven commands displayed in menus, while older operating systems relied exclusively on typed commands.
- **Command Prompt.** On newer versions of Windows, the application that provides you with an old-fashioned DOS-like command environment.
- **comment.** Any text in the source code of a computer program that is meant to be read only by humans, not by the compiler. In Java, C, and C++, any text between /* and */ is treated as a comment. Java and C++ also allow short

comments that start with // and continue to the end of the current line.

- **comparison operator.** An operator that compares two quantities to each other, yielding a boolean result. Examples are >, >=, <, <=, ==, and !=.
- **compile.** To translate a computer program (or portion thereof) from source code into machine language (or bytecode), storing the translated code in a file for later use.
- **compiler.** A computer program that compiles other computer programs. The Java compiler is invoked by the javac command.
- **concatenate.** To join two character strings together into a single string. Accomplished in Java with the + operator.
- **constructor.** A special type of method that is called when a new object is created. A constructor method must have the same name as the class it belongs to.
- **control.** A GUI element such as a button, scrollbar, or menu, that allows the user to control what a program does while it is running.
- debug. To find and remove the bugs from a program.
- declaration. Code that defines a name to represent a variable or method or class, and provides other necessary information such as the variable type and method parameters. In Java, certain other properties such as **static** and **public** can also be specified in declarations.
- **directory.** An older name for a folder, that is, a logical subunit of a disk or other storage device that can hold files.
- **disk.** A disc-shaped magnetic or optical storage device, used for long-term storage of computer files. (Flash memory devices are used in the same way, but are made of silicon chips rather than magnetic material.)
- **DOS.** Short for Disk Operating System, the original operating system developed by Microsoft for personal computers in the 1980's. Unlike its successor Windows, DOS used only a command-prompt user interface.
- double. Java's higher-precision format for storing floating-point numbers, which can handle a range of approximately $\pm 10^{308}$ and a precision of approximately 15 decimal digits.
- encapsulation. A design property of most object-oriented programming, whereby the variables belonging to a class are never accessed directly by other classes; rather, the class provides methods for reading and setting variables as appropriate. This is considered a good practice because it allows the class to be modified with fewer side effects on other classes that interact with it.
- **Euler algorithm.** A simple and relatively inaccurate algorithm for solving differential equations, in which the deriviative at the *beginning* of each time interval is used to extrapolate to the end of the interval. See Project 4.
- **Euler-Richardson algorithm.** An improvement on the Euler algorithm in which the derivative at the *middle* of the time interval is first estimated, and then used to extrapolate to the end of the interval. Also known as the second-order Runge-Kutta algorithm. See Project 4.
- **exception.** An error condition that normally must be dealt with in some way, such as attempting to reference a nonexistent array element or attempting to convert a string containing nonnumeric characters into a number.

execute. To carry out the instructions in a computer program.

- **expression.** A formula to be evaluated, often placed on the right-hand side of an assignment statement.
- **extend.** To change or add to the functionality of a previously existing class. See Project 5. A Java class may extend only one other class, called its superclass.
- false. A Java language word meaning just what it says; one of the possible values of a boolean variable or expression.
- field. Another name for an instance variable.
- file. A discrete collection of data, stored on a computer's hard disk or other storage device. A file can contain a computer program (in any format), or other data to be manipulated by computer programs. In the latter case, the file is often called a document.
- final. A modifier used in Java variable declaration-initialization statements to indicate that the quantity is actually a constant, not allowed to change while the program is running.
- floating-point number. A numerical value stored in a format that is analogous to scientific notation, with an exponent and a mantissa. (The mantissa of 2.998×10^8 is 2.998.) Java provides two different floating-point formats: float (for single precision, stored in 32 bits) and double (for double precision, stored in 64 bits).
- **folder.** A newer name for a directory, that is, a logical subunit of a disk or other storage device that can hold files.
- for. The Java word used to define a loop with an initialization statement, a condition to test, and an increment statement. See Project 2.
- **Fortran.** The first high-level, general-purpose programming language, originally developed in the 1950's and still widely used for scientific programming. The name is short for FORmula TRANslation.
- function. Another name for a method, often used when the method returns a variable or object (i.e., when the method is not void).
- graphical user interface (GUI). A way of interacting with a computer based on graphical elements (windows, menus, buttons, etc.) displayed on the screen, together with a mouse or other pointing device for user input. Before GUI's became widespread in the 1980's and 90's, most computers used a command-line interface similar to the Terminal and Command Prompt applications.
- **IDE.** Integrated Development Environment; a computer application that combines a programmer's text editor with additional features for organizing, compiling, and debugging large software projects. Examples include Eclipse, JBuilder, and Xcode. Most full-time programmers use an IDE rather than compiling and running programs from a command-line environment. For small projects, however, the overhead of using an IDE can be cumbersome.
- if-else. The Java words used to execute code only when a certain condition is true (or false). See Project 2.
- **implement.** To provide the necessary code for the methods that are merely declared in an interface.
- import. The Java statement that makes other packages available, when placed

at the top of a source code file. (The java.lang package is always available, without using import.)

- inheritance. A feature of object-oriented languages by which every class automatically includes all the variables and methods of the class(es) it extends.
- instance variable. A nonstatic variable that is declared outside a class's methods, and is therefore accessible to all of the class's nonstatic methods. Compare local variable and class variable.
- instantiate. To "create" (i.e., allocate memory for) an object and call the object's constructor method, normally accomplished in Java with the **new** operator.
- int. A Java primitive variable type for storing exact integers in the approximate range $\pm 2 \times 10^9$. An int occupies 32 bits of storage.
- interface. A special type of class-like entity that declares one or more methods but does not provide any code to implement these methods; that code is instead provided by any class that implements the interface. For example, a class can implement the Runnable interface by providing a run method. Although a Java class may extend only one other class, it may implement arbitrarily many interfaces.
- interpreter. A computer program that translates other computer programs into machine language and causes the translated instructions to be executed "on the fly," rather than storing the translated instructions in a file for later use (as a compiler would). Running a program through an interpreter is usually slower than compiling it first and then running the compiled code directly.
- Java. 1. A programming language, introduced by Sun Microsystems (now Oracle) in 1995, whose syntax resembles that of C and C++ but whose deeper features differ significantly from these languages to allow better security, platformindependent code, and easier development of large software projects. 2. The Java Virtual Machine, a software system that interprets and runs bytecode, which has normally (but not necessarily) been generated by compiling programs written in the Java language. Often invoked by the java command. 3. A standard collection of software libraries, now consisting of nearly 4000 classes, that is distributed and installed along with the Java Virtual Machine.
- Javascript. A computer language that is often confused with Java, due to the unfortunate decision to give it a similar name. Javascript is a language used to give web pages various dynamic features such as graphics that change as the user moves the mouse cursor. Javascript code is normally embedded within the html code in a web page, to be interpreted and executed by the web browser. Java applets can also add dynamic features to web pages, but are more self-contained and far more powerful than Javascript enhancements.
- launch. To begin running a program.
- **library.** A collection of computer software for performing common tasks, which programmers can incorporate into their own higher-level programs. Current versions of Java include a standard library of approximately 3000 classes (grouped into packages), and additional libraries are available for more specialized uses.
- Linux. An operating system for personal computers that mimics Unix, developed by a dispersed community of programmers and distributed under an open-

source license.

- **Literal.** In the code of a computer program, an explicitly provided value (as opposed to a variable name or expression), such as 42 or 3.14 or "Hello, world!".
- **local variable.** A variable that is declared within a method, and is consequently "visible" only within this method. If a variable is needed only within one method, it is safer to make it a local variable. Compare instance variable and class variable.
- long. A Java primitive variable type for storing exact integers in the approximate range $\pm 9 \times 10^{18}$. A long occupies 64 bits of storage.
- **loop.** A block of code that is executed repeatedly, as long as some condition remains true. Java's while and for constructions define loops.
- machine language. The language (essentially of 1's and 0's) that a computer's microprocessor chip can execute directly. Humans rarely write programs directly in machine language.
- **Macintosh.** A brand of personal computer made by Apple, first introduced in 1984. Also, the operating system (abbreviated Mac OS) that runs on these computers, which was the first widely used operating system to employ a graphical user interface.
- main. The name of the method in a Java (or C or C++) program where execution automatically begins. More precisely, the command java MyClass causes execution to begin at the main method of class MyClass (even if the program includes other classes with their own main methods).
- **memory.** The portion of a computer that is used for temporary storage of programs and data, consisting of silicon chips. While a program is running, its machine code and the values of all its variables are normally stored in memory.
- **method.** A chunk of Java code that carries out a specific task. Also sometimes called a function, subroutine, or procedure. A method can optionally have one or more input parameters (variables of any type), and can optionally return a value of any type.
- Metropolis algorithm. A widely useful Monte Carlo algorithm for simulating systems at fixed temperature. See Project 9.
- Microsoft. A monopolistic software company based in Redmond, WA. Products include the Windows operating system and the application programs Word, Excel, and PowerPoint.
- Monte Carlo simulation. A computer simulation in which pseudo-random numbers are used to choose among different possible outcomes.
- new. The Java language word used to create a new object.
- **object.** An instance of a class, created (in Java) using the **new** operator. When a program uses multiple objects of the same class, each instance has its own set of instance variables.
- **object-oriented programming.** The practice of using classes and objects to break large programming projects into smaller, more manageable pieces that can be written, tested, and debugged more or less independently. Often abbreviated OOP.
- operating system. The low-level software that manages basic tasks on a com-

puter, such as launching application programs and keeping track of files stored on disks. Examples include Unix, Linux, MacOS (including OS X), DOS, and Windows.

- **operator.** A symbol, symbol combination, or language word that specifies some computational operation. Examples in Java include the assignment and short-cut operators (= += -= *= /= ++ --), arithmetic operators (+ * /), comparison operators (== != < > <= >=), boolean operators (&& || !), and instantiation operator (new).
- **OS X.** Version 10 of the Macintosh operating system, which (unlike previous Mac OS versions) is actually a version of Unix. Pronounced "oh ess ten."
- **overflow.** The condition that occurs when the result of a numerical computation is too large to fit into the variable type being used. In Java, overflow of an integer variable type produces erratic results. Overflow of a real data type gives "infinity."
- **overloading.** Using the same name or symbol to mean more than one thing. In Java the only overloaded operator is +, which is used both to add numbers and to concatenate strings. Java also allows overloading of method names: Two different methods can have the same name, so long as they have different parameter types or different numbers of parameters. An example is the multiple add methods of the AWT Container class—one that takes a single parameter and another that takes two parameters.
- **package.** A collection of related classes, usually distributed together. Examples include java.text, java.awt, and org.opensourcephysics. Classes not declared to be part of a package automatically become part of the "nameless" package. (To declare a class to be part of some other package, you use the **package** statement.)
- **parameter.** A variable whose value must be provided to a method when that method is called. In Java, parameters are enclosed in parentheses after the name of the method, separated by commas from other parameters.
- primitive type. A variable type that is fundamental to the language itself rather than defined as a class. Java provides eight primitive variable types: byte, short, int, long, float, double, char, and boolean. All other types are really classes, with their own instance variables that are either primitive types or other class types.
- **program.** A self-contained collection of computer instructions for accomplishing some task. The task could be as simple as printing "Hello, world!" or as complex as simulating Earth's climate.
- **prompt.** One or more characters displayed by the computer to indicate that it is ready to accept a typed command (for example, in the Terminal application on Mac OS X, or the Command Prompt application on newer versions of Windows).
- **pseudo-random number.** A number that is returned from a function such as Java's Math.random function. Although each such number is computed from the previous one using a deterministic function, the function is sufficiently chaotic that we can treat the numbers as "random" for many (though not all) purposes.

See Project 8.

- public. A modifier used in class, method, and variable declarations to indicate that the thing being declared should be "visible" outside its own package.
- return. To cease execution of the current method and go back to executing further statements in the method that called it. Also, for methods that are not void, to supply the value that the method is intended to compute. The Java return instruction accomplishes both tasks, although a void method also returns automatically after its last statement is executed.
- **run.** To carry out the instructions in a computer program (if you're a computer), or to instruct a computer to do so (if you're a human).
- **simulation.** A computer program that uses the laws of physics (or some approximation thereof) to simulate the behavior of a physical system as a function of time.
- **source code.** A computer program (or portion thereof) that is still in the format in which it was typed by a human, stored in one or more text files. The javac command compiles Java source code into bytecode.
- **statement.** A small, logically complete unit of code, analogous to a complete English sentence, that instructs the computer to do something. Although a typical statement occupies one line of the source code file, Java (like C and C++) lets you split a statement onto more than one line, or combine multiple statements on a single line. The end of a statement in these languages is indicated by a semicolon.
- static. A modifier that can be used on Java variables and methods. A static variable (also called a class variable) is shared among all instances of a class, rather than being duplicated so that each instance has its own value of the variable. A static method can access only the class's static variables, not instance variables.
- string. A sequence of text characters, for example, "Hello, world!". In Java, String is a special class that is automatically instantiated whenever the code contains a sequence of characters enclosed by double-quotes. Java prohibits you from modifying a String once it has been created, but provides a separate string class (called StringBuffer) whose instances can be modified.
- subclass. A class that extends some other class.
- **Sun Microsystems.** A computer hardware and software company based in Santa Clara, CA, founded in 1982 and acquired by Oracle Corporation in 2010. Perhaps best remembered for developing Java.
- super. This Java language word is used to call methods belonging to a class's
 superclass, even when the present class has overridden those methods. By
 itself, super() calls the superclass's constructor method; to call method xxx,
 use "super.xxx()".
- superclass. The class that the present class extends, that is, the class immediately above it in Java's class hierarchy. For a class declared with no explicit extends clause, the superclass is Object.
- synchronized. A modifier used in method declarations to ensure that the method and any others that are also declared as synchronized are never called simul-

taneously by more than one thread. So before any thread can begin to execute a synchronized method, any synchronized method (in the same object) that is being executed by another thread must first terminate.

- **syntax.** The basic rules for composing instructions in a particular computer language, somewhat analogous to spelling, vocabulary, and grammar in ordinary languages.
- **Terminal.** A Macintosh OS X application that accepts and executes Unix commands.
- **text editor.** A program for creating and modifying plain text files such as computer programs and html files. Such files consist only of characters found on computer keyboards, with no graphics, font information, or other formatting. Compare word processor.
- this. A Java language word that stands for the present object.
- thread. A separate "mini-process" that runs in parallel with other threads that belong to the same program.
- true. A Java language word meaning just what it says; one of the possible values of a boolean variable or expression.
- truncation error. Inaccuracy in a numerical computation that is caused by approximating derivatives as ratios of differences. Truncation errors can be lessened by using smaller increments (such as dt) or by using more accurate algorithms.
- try-catch. The Java words used to handle potential error conditions, also called exceptions.
- type. The sort of data that a variable can hold. Java has several "primitive" data types such as int, double, and boolean. In addition, virtually every Java class is its own data type, where the data consists of the class's collection of instance variables.
- **Unix.** An operating system originally developed at Bell Laboratories in New Jersey, now found in many versions (including Linux and OS X) and widely used by scientists and computer programmers.
- user. A person who runs and interacts with a computer program.
- variable. A discrete chunk of data used by a computer program, with its own name. In Java, variable names are made of letters, numerals, and the underscore character. Each variable must be of a definite type—either a primitive type such as int or double, or a class type such as String or Frame. Access to a variable is controlled by whether it is a local variable, instance variable, or class variable.
- **Verlet algorithm.** An algorithm for solving Newton's second law that is useful when the forces depend only on the positions of the particles, not on their velocities. Also called the second Taylor approximation (STA). See Project 6.
- void. A modifier used in method declarations to indicate that the method does not return a value.
- while. The Java word used to define a loop that executes repeatedly as long as a certain condition holds true. See Project 2.
- Windows. An operating system developed and licensed by Microsoft Corp., now installed on the vast majority of personal computers worldwide. The name

signifies that Windows incorporates a graphical user interface, unlike its predecessor, DOS.

word processor. A program that facilitates the creation of text documents that are formatted to look nice when printed; for example, Microsoft Word.