Chapter 2

Tools for Doing Simulations

19 May 2005

We introduce some of the core syntax of Java in the context of simulating the motion of falling particles near the Earth's surface. A simple algorithm for solving first-order differential equations numerically also is discussed.

2.1 Introduction

If you were to take a laboratory-based course in physics, you would soon be introduced to the oscilloscope. You would learn the function of many of the knobs, how to read the display, and how to connect various devices so that you could measure various quantities. If you did not know already, you would learn about voltage, current, impedance, and AC and DC signals. Your goal would be to learn how to use the oscilloscope. In contrast, you would learn only a little about the inner workings of the oscilloscope.

The same approach can be easily adopted with an object oriented language such as Java. If you are new to programming, you will learn how to make Java do what you want, but you will not learn everything about Java. In this chapter, we will present some of the essential syntax of Java and introduce the Open Source Physics library, which will facilitate writing programs with a graphical user interface and visual output such as plots and animations.

One of the ways that science progresses is by making models. If the model is sufficiently detailed, we can determine its behavior, and then compare the behavior with experiment. This comparison might lead to verification of the model, changes in the model, and further simulations and experiments. In the context of computer simulation, we usually begin with a set of initial conditions, determine the dynamical behavior of the model numerically, and generate data in the form of tables of numbers, plots, and animations. We begin with a simple example to see how this process works.

Imagine a particle such as a ball near the surface of the Earth subject to a single force, the force of gravity. We assume that air friction is negligible and the gravitational force is given by

$$F_g = -mg, \tag{2.1}$$

where m is the mass of the ball and $g = 9.8\,\mathrm{N/kg}$ is the gravitational field (force per unit mass) near the Earth's surface. To make our example as simple as possible, we first assume that there is only vertical motion. We use Newton's second law to find the motion of the ball,

$$m\frac{d^2y}{dt^2} = F, \tag{2.2}$$

where y is the vertical coordinate defined so that up is positive, t is the time, F is the total force on the ball, and m is the inertial mass (which is the same as the gravitational mass in (2.1)). If we set $F = F_g$, (2.1) and (2.2) lead to

$$\frac{d^2y}{dt^2} = -g. \tag{2.3}$$

Equation (2.3) is a statement of a model for the motion of the ball. In this case the model is in the form of a second-order differential equation.

You are probably familiar with the model summarized in (2.3) and know the analytical solution:

$$y(t) = y(0) + v(0)t - \frac{1}{2}gt^2 \tag{2.4a}$$

$$v(t) = v(0) - gt. \tag{2.4b}$$

Nevertheless, we will determine the motion of a freely falling particle numerically in order to introduce the tools that we will need in a familiar context.

We begin by expressing (2.3) as two first-order differential equations:

$$\frac{dy}{dt} = v \tag{2.5a}$$

$$\frac{dv}{dt} = -g, \tag{2.5b}$$

where v is the vertical velocity of the ball. We next *approximate* the derivatives by small (finite) differences:

$$\frac{y(t+\Delta t) - y(t)}{\Delta t} = v(t) \tag{2.6a}$$

$$\frac{v(t+\Delta t) - v(t)}{\Delta t} = -g. \tag{2.6b}$$

Note that in the limit $\Delta t \to 0$, (2.6) reduces to (2.5). We can rewrite (2.6) as

$$y(t+\Delta t) = y(t) + v(t)\Delta t \tag{2.7a}$$

$$v(t+\Delta t) = v(t) - g\Delta t, \tag{2.7b}$$

The finite difference approximation we used to obtain (2.7) is an example of the *Euler* algorithm. Equation (2.7) is an example of a *finite difference* equation, and Δt is the time step.

Now we are ready to follow $y(t)$ and $v(t)$ in time. We begin with an initial value for y and v and then *iterate* (2.7). If Δt is sufficiently small, we will obtain a numerical answer that is close to the solution of the original differential equations in (2.6). In this case we know the answer, and we can test our numerical results directly.

Exercise 2.1. A simple example

Consider the first-order differential equation

$$\frac{dy}{dx} = f(x), \tag{2.8}$$

where $f(x)$ is a function of x. The approximate solution as given by the Euler algorithm is

$$y_{n+1} = y_n + f(x_n)\Delta x. \tag{2.9}$$

Note that the rate of change of y has been approximated by its value at the *beginning* of the interval, $f(x_n)$

a. Suppose that $f(x) = 2x$ and $y(x = 0) = 0$. The analytical solution is $y(x) = x^2$, which we can confirm by taking the derivative of $y(x)$. Convert (2.8) into a finite difference equation using the Euler algorithm. For simplicity, choose $\Delta x = 0.1$. It would be a good idea to first use a calculator or pencil and paper to determine y_n for the first several time steps.

b. Sketch the difference between the exact solution and the approximate solution given by the Euler algorithm. What condition would the rate of change, $f(x)$, have to satisfy for the Euler algorithm to give the exact answer?

Problem 2.2. Invent your own numerical algorithm

As we have mentioned, the Euler algorithm evaluates the rate of change of y by its value at the beginning of the interval, $f(x_n)$. The choice of where to approximate the rate of change of y during the interval from x to $x + \Delta x$ is arbitrary, although we will learn that some choices are better than others. All that is required is that the finite difference equation must reduce to the original differential equation in the limit $\Delta x \to 0$. Think of several other algorithms that are consistent with this condition.

2.2 Simulating free fall

The source code for the *class* `FirstFallingBallApp` shown in Listing 2.1 is defined in a file named FirstFallingBallApp.java. The code consists of a sequence of *statements* that create *variables* and define *methods*. Each statement ends with a semicolon. Each source code file is compiled into *byte code* that can then be executed. The compiler places the byte code in a file with the same name as the Java source code file with the extension `class`. For example, the compiler converts FirstFallingBallApp.java into byte code and produces the FirstFallingBallApp.class file. One of the features of Java is that this byte code can be used by any computer that can run Java programs.

A Java application is a class that contains a *main* method. The following application is an implementation of the Euler algorithm given in (2.7). The program also compares the numerical and analytic results. We will next describe the syntax used in each line of the program.

Listing 2.1: First version of a simulation of a falling particle.

```
1 // example of a single line comment statement (ignored by compiler)
2 package org.opensourcephysics.sip.ch02;      // location of file
```

```
3  public class FirstFallingBallApp {          // beginning of class definition
4     public static void main(String[] args) { // beginning of method definition
5        // braces { } used to group statements.
6        // indent statements within a block so that they can be easily identified
7        // following statements form the body of main method
8        double y0 = 10;    // example of declaration and assignment statement
9        double v0 = 0;     // initial velocity
10       double t = 0;      // time
11       double dt = 0.01; // time step
12       double y = y0;
13       double v = v0;
14       double g = 9.8;    // gravitational field
15       for(int n = 0;n<100;n++) { // beginning of loop, n++ equivalent to n = n + 1
16          // repeat following three statements 100 times
17          y = y+v*dt; // indent statements in loop for clarity
18          v = v-g*dt; // use Euler algorithm
19          t = t+dt;
20       }              // end of for loop
21       System.out.println("Results");
22       System.out.println("final time = "+t);
23       // display numerical result
24       System.out.println("y = "+y+" v = "+v);
25       // display analytic result
26       double yAnalytic = y0+v0*t-0.5*g*t*t;
27       double vAnalytic = v0-g*t;
28       System.out.println("analytic y = "+yAnalytic+" v = "+vAnalytic);
29    } // end of method definition
30 } // end of class definition
```

The first line in Listing 2.1 is an example of a single line comment statement. Comment statements are ignored by the computer, but can be very important for the user. Multiple line comments begin with `/*` and end with `*/`. Javadoc comments begin with `/**`, but have been removed from the code listings in the book to save space. Download the source code from the Open Source Physics website to view the complete code with documentation.

The second line in Listing 2.1 declares a *package* name, which corresponds to the location (directory) of the source and byte code files. According to the package declaration, the file FirstFallingBallApp.java is in the directory org/opensourcephysics/sip/ch02. The package statement must be the first non-comment statement in the source file. For organizational convenience, it is a good idea to put related files in the same package. When executing a Java program, the Java Virtual Machine (the run-time environment) will search a specific set of directories (called the classpath) for the relevant class files. The documentation for your local development environment will describe how to specify the classpath.

The third line in Listing 2.1 declares the class name, `FirstFallingBallApp`. The Java convention is to begin a class name with an uppercase letter. If a name consists of more than one word, the words are joined together and each succeeding word begins with an uppercase letter (another Java convention). The keyword `public` means that this class can be used by any other Java class.

Braces are used to delimit a block of code. The left brace, {, after the name of the class begins the body of the class definition and the corresponding right brace, }, at the end of the code listing on line 31 ends the class definition.

The fourth line in Listing 2.1 begins the definition of the `main` method. A method describes a sequence of actions that use the associated data and can be *called* (invoked) within the class or by other classes. The `main` method has a special status in Java. To run a class as a stand-alone program (an application), the class must define the `main` method. (In contrast, a Java applet runs inside a browser and does not require a `main` method; instead, it has methods such as `init` and `start`.) The `main` method is the application's starting point. The argument of the `main` method will always be the same, and understanding its syntax is not necessary here.

Because the code for this book contains hundreds of classes, we will adopt our own convention that classes that define main methods have names that end with `App`. We sometimes refer to an application that we are about to run as the *target* class.

Familiarize yourself with your Java development environment by doing Exercise 2.3.

Exercise 2.3. Our first application

a. Enter the listing of `FirstFallingBallApp` into a source file named FirstFallingBallApp.java. (Java programs can be written using any text editor that supports standard ASCII characters.) Be sure to pay attention to capitalization because Java is case sensitive. In what directory should you place the source file?

b. Compile and run `FirstFallingBallApp`. Do the results look reasonable to you? In what directory did the compiler place the byte code?

Digital computers represent numbers in base 2, that is, sequences of ones and zeros. Each one or zero is called a *bit*. For example, the number 13 is equivalent to 1101 or $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$. It would be difficult to write a program if we had to write numbers in base 2. Computer languages allow us to reference memory locations using identifiers or variable names. A valid variable name is a series of characters consisting of letters, digits, underscores, and dollar signs ($) that does not begin with a digit nor contain any spaces. Because Java distinguishes between upper and lowercase characters, `T` and `t` are different variable names. The Java convention is that variable names begin with a lowercase letter, except in special cases, and each succeeding word in a variable name begins with an uppercase letter.

In a purely object oriented language, all variables would be objects that would be introduced by their class definitions. However, there are certain variable types that are so common that they have a special status and are especially easy to create and access. These types are called *primitive data types* and represent integer, floating point, boolean, and character variables. An example that illustrates that classes are effectively new programmer-defined types is given in Appendix 2A.

An integer variable, a floating point variable, and a boolean variable are created and *initialized* by the following statements:

```
int n = 10;
double y0 = 10.0;
boolean inert = true;
char c = 'A';     // used for single characters
```

There are four types of integers, `byte`, `short`, `int`, and `long`, and two types of floating point numbers; the differences are the range of numbers that these types can store. We will almost always use type `int` because it does not require as much memory as type `long`. There are two types of floating point numbers, but we will always use type `double`, the type with greater precision, to minimize roundoff error and to avoid having to provide multiple versions of various algorithms. A variable must be *declared* before it can be used, and it can be initialized at the same time that its type is declared as is done in Listing 2.1.

Integer arithmetic is exact, in contrast to floating point arithmetic which is limited by the maximum number of decimal places that can be stored. Important uses of integers are as counters in loops and as indices of arrays. An example of the latter is on page 39, where we discuss the motion of many balls.

A subtle and common error is to use integers in division when a floating point number is needed. For example, suppose we flip a coin 100 times and find 53 heads. What is the percentage of heads? In the following we show an unintended side effect of integer division and several ways of obtaining a floating point number from an integer.

```
int heads = 53;
int tosses = 100;
double percentage = heads/tosses;     // percentage will equal 0 (common error)
percentage = (double)heads/tosses;    // percentage will equal 0.53
percentage = (1.0*heads)/tosses;      // percentage will equal 0.53
```

These statements indicate that if at least one number is a `double`, the result of the division will be a `double`. The expression `(double)tails` is called a *cast* and converts `heads` to a `double`. Because a number with a decimal point is treated as a double, we also can do this conversion by first multiplying heads by 1.0 as is done in the last statement.

Note that we have used the *assignment operator*, which is the equals (`=`) sign. This operator assigns the value to the memory location that is associated with a variable, such as `y0` and `t`. The following statements illustrate an important difference between the equals sign in mathematics and the assignment operator in most programming languages.

```
int x = 10;
x = x + 1;
```

The equals sign replaces a value in memory and is not a statement of equality. The left and right sides of an assignment operator are usually not equal.

A statement is analogous to a complete sentence, and an expression is similar to a phrase. The simplest expressions are identifiers or variables. More interesting expressions can be created by combining variables using *operators*, such as the following example of the plus (`+`) operator:

```
x + 3.0
```

Lines seven through thirteen of Listing 2.1 declare and initialize variables. If a variable is declared but not initialized, for example,

```
double dt;
```

then the default value of the variable is 0 for numbers and `false` for boolean variables. It is a good idea to initialize all variables explicitly and not rely on their default values.

A very useful control structure is the `for` loop in line 15 of Listing 2.1. Loops are blocks of statements that are executed repeatedly until some condition is satisfied. They typically require the initialization of a counter variable, a test to determine if the counter variable has reached its terminal value, and a rule for changing the counter variable. These three parts of the `for` loop are contained within parentheses and are separated by semicolons. It is common in Java to iterate from 0 to 99 as is done in Listing 2.1, rather than from 1 to 100. Note the use of the `++` operator in the loop construct rather than the equivalent statement `n = n + 1`. It is important to indent all the statements within a block so that they can be easily identified. Java ignores these spaces, but they are important visual cues to the structure of the program.

After the program finishes the loop, the results are displayed using the `System.out.println` method. We will explain the meaning of this syntax later. The parameter passed to this method, which appears between the parentheses, is a `String`. A `String` is a sequence of characters and can be created by enclosing text in quotation marks as shown in the first `println` statement in Listing 2.1. We displayed our numerical results by using the `+` operator. When applied to a `String` and a number, the number is converted to the appropriate `String` and the two Strings are concatenated (joined). This use is shown in the next three `println` statements in lines 21 through 29 of Listing 2.1. Note the different outputs produced by the following two statements:

```
System.out.println(("x = " + 2) + 3);     // displays x = 23
System.out.println("x = " + (2 + 3));     // displays x = 5
```

The parentheses in the second line force the compiler to treat the enclosed `+` operator as the addition operator, but both `+` operators in the first line are treated as concatenation operators.

Exercise 2.4. Exploring FirstFallingBallApp

a. Run `FirstFallingBallApp` for various values of the time step Δt. Do the numerical results become closer to the analytic results as Δt is made smaller?

b. Use an acceptable value for Δt and run the program for various values for the number of iterations. What criteria do you have for acceptable? At approximately what time does the ball hit the ground at $y = 0$?

c. What happens if you replace the `System.out.println` method by the `System.out.print` method?

d. What happens if you try to access the value of the counter variable `n` outside the for loop? The *scope* of `n` extends from its declaration to the end of the loop block; `n` is said to have *block scope*. If a loop variable is not needed outside the loop, it should be declared in the initialization expression so that its scope is limited.

You might have found that doing Exercise 2.4 was a bit tedious and frustrating. To do Exercise 2.4(a) it would be desirable to change the number of iterations at the same time that the value of Δt is changed so that we could compare the results for y and v at the same time. And it is difficult to do Exercise 2.4(b) because we don't know in advance how many iterations are needed to reach the ground. For starters, we can improve `FirstFallingBallApp` using a `while` statement instead of the `for` loop.

```
while (y > 0) {
   // statements go here
}
```

In this example the boolean test for the `while` statement is done at the beginning of a loop.

It is also possible to do the test at the end:

```
do {
   // statements go here
}
while (y > 0);
```

Exercise 2.5. Using while statements

Modify `FirstFallingBallApp` so that the `while` statement is used and the program ends when the ball hits the ground at $y = 0$. Then repeat Exercise 2.4(b).

Exercise 2.6. Summing a series

a. Write a program to sum the following series for a given value of N:

$$S = \sum_{m=1}^{N} \frac{1}{m^2}. \tag{2.10}$$

The following statements may be useful:

```
double sum = 0;          // sum is equivalent to S in (2.9)
for (int m = 1; m <= N; m++) {
   sum = sum + 1.0/(m*m); // put this statement in for loop
}
```

Note that in this case it is more convenient to start the loop from $m = 1$ instead of $m = 0$. Also note that we have not followed the Java convention, because we have used the variable name `N` instead of `n` so that the Java statements look more like the mathematical equations.

b. First run your program with $N = 10$. Then run for larger values of N. Does the series converge as $N \to \infty$? What value of N is needed to obtain S to within two decimal places?

c. Modify your program so that it uses a while loop so that the summation continues until the added term to the sum is less than some value ϵ. Run your program for $\epsilon = 10^{-2}$, 10^{-3}, and 10^{-6}.

d. Instead of using the `=` operator in the statement

```
sum = sum + 1.0/(m*m);
```

use the equivalent operator:

```
sum += 1.0/(m*m);
```

Check that you obtain the same results.

Java provides several shortcut assignment operators that allow you to combine an arithmetic and an assignment operation. Table 2.1 shows the operators that we will use most often.

operator	operand	description	sample expression	result
`++, --`	number	increment, decrement	`x++;`	8.0 stored in `x`
`+, -`	numbers	addition, subtraction	`3.5 + x`	11.5
`!`	boolean	logical complement	`!(x == y)`	true
`=`	any	assignment	`y = 3;`	`3.0` stored in `y`
`*, /, %`	numbers	multiplication, division, modulus	`7/2`	3.0
`==`	any	test for equality	`x == y`	false
`+=`	numbers	`x += 3;` equivalent to `x = x + 3;`	`x += 3;`	14.5 stored in `x`
`-=`	numbers	`x -= 2;` equivalent to `x = x - 2;`	`x -= 2.3;`	12.2 stored in `x`
`*=`	numbers	`x *= 4;` equivalent to `x = 4*x;`	`x *= 4;`	48.8 stored in `x`
`/=`	numbers	`x /= 2;` equivalent to `x = x/2;`	`x /= 2;`	24.4 stored in `x`
`%=`	numbers	`x %= 5;` equivalent to `x = x % 5;`	`x %= 5;`	4.4 stored in `x`

Table 2.1: Common operators. The result for each row assumes that the statements from previous rows have been executed with `double x = 7, y = 3` declared initially. The *mod* or *modulus* operator, `%`, computes the remainder after the division by an integer has been performed.

2.3 Getting started with object oriented programming

The first step in making our program more object oriented is to separate the implementation of the model from the implementation of other programming tasks such as producing output. In general, we will do so by creating two classes. The class that defines the model is shown in Listing 2.2. The `FallingBall` class first declares several (instance) variables and one constant that can be used by any method in the class. To aid reusability, we need to be very careful about the accessibility of these class variables to other classes. For example, if we write `private double dt`, then the value of `dt` would only be available to the methods in `FallingBall`. If we wrote `public double dt`, then `dt` would be available to any class in any package that tried to access it. For our purposes we will use the default package protection, which means that the instance variables can be accessed by classes in the same package.

Listing 2.2: `FallingBall` class.

```
package org.opensourcephysics.sip.ch02;
public class FallingBall {
   double y, v, t;                  // instance variables
   double dt;                       // default package protection
   final static double g = 9.8; // constant (note non-use of Java convention)

   public FallingBall() {           // constructor
      System.out.println("A new FallingBall object is created.");
   }

   public void step() {
      y = y+v*dt; // Euler algorithm for numerical solution
      v = v-g*dt;
      t = t+dt;
   }
```

```
   public double analyticPosition(double y0, double v0) {
      return y0+v0*t-0.5*g*t*t;
   }

   public double analyticVelocity(double v0) {
      return v0-g*t;
   }
}
```

As we will see, a class is a blueprint for creating objects, not an object itself. Except for the constant `g`, all the variable declarations in Listing 2.2 are *instance* variables. Each time an object is created or *instantiated* from the class, a separate block of memory is set aside for the instance variables. Thus, two objects created from the same class will, in general, have different values of the instance variables. We can insure that the value of a variable is the same for all objects created from the class by adding the word `static` to the declaration. Such a variable is called a *class* variable, and is appropriate for the constant `g`. In addition, you might not want the quantity referred to by an identifier to change. For example, `g` is a constant of nature. We can prevent a change by adding the keyword `final` to the declaration. Thus the statement

```
final static double g = 9.8;
```

means that a single copy of the constant `g` will be created and shared among all the objects instantiated from the class. Without the `final` qualifier, we could change the value of a class variable in every instantiated object, by changing it in any one object. Static variables and methods are accessed from another class using the class name without first creating an instance (see page 25).

Another Java convention is that the names of constants should be in upper case. But in physics, the meaning of g, the gravitational field, and G, the gravitational constant, have completely different meanings. So we will disregard this convention if doing so makes our programs more readable.

We have used certain words such as `double`, `false`, `main`, `static`, and `final`. These reserved words cannot be used as variable names and are examples of *keywords*.

In addition to the four instance variables, `y`, `v`, `t`, and `dt`, and one class variable, `g`, the `FallingBall` class has four methods. The first method is `FallingBall`, and is a special method known as the *constructor*. A constructor must have the same name as the class and does not have an explicit return type. We will see that constructors allocate memory and initialize instance variables when an object is created.

The second method is `step`, a name that we will frequently use to advance a system's coordinates by one time step. The qualifier `void` means that this method does not return a value.

The next two methods, `analyticPosition` and `analyticVelocity`, each return a double value and have arguments enclosed by parentheses, the parameter list. The list of parameters and their types must be given explicitly and be separated by commas. The parameters can be primitive data types or class types. When the method is invoked, the argument types must match that given in the definition or be convertible into the type given in the definition, but need not have the same names. (Convertible means that the given variable can be unambiguously converted into another data type. For example, an integer can always be converted into a double.) For example, we can write

```
double y0 = 10;      // declaration and assignment
int v0 = 0;          // note v0 is an integer
double y = analyticPosition(y0,v0);  // v0 becomes a double before method is called
double v = analyticVelocity(v0);
```

but the following statements are incorrect:

```
double y = analyticPosition(y0,"0");      // can't convert String to double automatically
double v = analyticVelocity(v0,0);        // method expects only one argument
```

If a method does not receive any parameters, the parentheses are still required as in method `step()`.

The `FallingBall` class in Listing 2.2 cannot be used in isolation because it does not contain a `main` method. Thus, we create a target class, which we place in a separate file in the same package. This class will communicate with `FallingBall` and include the output statements. This class is shown in Listing 2.3.

Listing 2.3: `FallingBallApp` class.

```
// package statement appears before beginning of class definition
package org.opensourcephysics.sip.ch02;
public class FallingBallApp {                  // beginning of class definition
   public static void main(String[] args) { // beginning of method definition
      FallingBall ball = new FallingBall(); // declaration and instantiation
      double y0 = 10;                        // example of declaration and assignment stateme
      double v0 = 0;
      ball.t = 0; // note use of dot operator to access instance variable
      ball.dt = 0.01;
      ball.y = y0;
      ball.v = v0;
      while(ball.y>0) {
         ball.step();
      }
      System.out.println("Results");
      System.out.println("final time = "+ball.t);
      // displays numerical results
      System.out.println("y = "+ball.y+" v = "+ball.v);
      // displays analytic results
      System.out.println("analytic y = "+ball.analyticPosition(y0, v0));
      System.out.println("analytic v = "+ball.analyticVelocity(v0));
      System.out.println("acceleration = "+FallingBall.g);
   } // end of method definition
} // end of class definition
```

Note how `FallingBall` is declared and *instantiated* by creating an object called `ball`, and how the instance variables and the methods are accessed. The statement

```
FallingBall ball = new FallingBall();  // declaration and instantiation
```

is equivalent to two statements:

```
FallingBall ball;           // declaration
ball = new FallingBall();   // instantiation
```

The declaration statement tells the compiler that the variable `ball` is of type `FallingBall`. It is analogous to the statement `int x` for an integer variable. The `new` operator allocates memory for this object, initializes all the instance variables, and invokes the constructor. We can create two identical balls using the following statements:

```
FallingBall ball1 = new FallingBall();
FallingBall ball2 = new FallingBall();
```

The variables and methods of an object are accessed by using the *dot operator*. For example, the variable `t` of object `ball` is accessed by the expression `ball.t`, and the method `step` is called as `ball.step()`. Because the methods, `analyticPosition` and `analyticVelocity`, return values of type `double`, they can appear in any expression in which a double valued constant or variable can appear. In the present context the values returned by these two methods will be displayed by the `println` statement. Note that the static variable `g` in class `FallingBallApp` is accessed through the class name.

Exercise 2.7. Use of two classes

a. Enter the listing of `FallingBall` into a file named FallingBall.java and `FallingBallApp` into a file named FallingBallApp.java and put them in the same directory. Run your program and make sure your results are the same as those found in Exercise 2.5.

b. Modify `FallingBallApp` by adding a second instance variable `ball2` of the same type as `ball`. Add the necessary code to initialize `ball2`, iterate `ball2`, and display the results for both objects. Write your program so that the only difference between the two balls is the value of Δt. How much smaller does Δt have to be to reduce the error in the numerical results by a factor of two for the same final time? What about a factor of four? How does the error depend on Δt?

c. Add the statement `FallingBall.g = 2.0` to your program from part (b) and use the same value of `dt` for `ball` and `ball2`. What happens when you try to compile the program?

d. Delete the `final` qualifier for `g` in `FallingBall` and recompile and run your program. Is there any difference between the results for the two balls? Is there a difference between the results compared to what you found for $g = 9.8$?

e. Remove the qualifier `static`. Now `g` must be accessed using the object name, `ball` or `ball2` instead of `FallingBall`. Recompile your program again, and run your program. How do the results for the two balls compare now?

f. Explain in your own words the meaning of the qualifiers, `static` and `final`.

It is possible for a class to have more than one constructor. For example, we could have a second constructor defined by

```
public FallingBall(double dt) {
// "this.dt" refers to an instance variable that has the same name as the argument
   this.dt = dt;
}
```

Note the possible confusion of the variable name, `dt`, in the argument of the `FallingBall` constructor and the variable defined near the beginning of the `FallingBall` class. A variable that is passed to a method as an argument (parameter) or that is defined (created) within a method is known as a *local variable*. A variable that is defined outside of a method is known as an *instance variable*. Instance variables are more powerful than local variables because they can be referenced (used) anywhere within an object and because their values are not lost when the execution of the method is finished. When a variable name conflict occurs, it is necessary to use the keyword `this` to access the instance variable. Otherwise, the program would access the variable in the argument (the local variable) with the same name.

Exercise 2.8. Multiple constructors

a. Add a second constructor with the argument `double dt` to `FallingBall`, but make no other changes. Run your program. Nothing changed because you didn't use this new constructor.

b. Now modify `FallingBallApp` to use the new constructor:

```
FallingBall ball = new FallingBall(0.01); // declaration and instantiation
```

What statement in `FallingBallApp` can now be removed? Run your program and make sure it works. How can you tell that the new constructor was used?

c. Show that the number of parameters and their type in the argument list determines which constructor is used in `FallingBall`. For example, show that the statements,

```
double tau = 0.01;
FallingBall ball = new FallingBall(tau); // declaration and instantiation
```

are equivalent to the syntax used in part (b).

It is easy to create additional models for other kinds of motion. Cut and paste the code in the `FallingBall` into a new file named SHO.java and change the code to solve the following two first-order differential equations for a ball attached to a spring:

$$\frac{dx}{dt} = v \tag{2.11a}$$

$$\frac{dv}{dt} = -\frac{k}{m}x, \tag{2.11b}$$

where x is the displacement from equilibrium and k is the spring constant. Note that the new class shown in Listing 2.4 has a structure similar to that of the class shown in Listing 2.2.

Listing 2.4: `SHO` class.

```
package org.opensourcephysics.sip.ch02;
public class SHO {
```

```
   double x, v, t;                  // the dynamical variables
   double dt;
   double k = 1.0;                  // spring constant
   double omega0 = Math.sqrt(k); // assume unit mass

   public SHO() {                   // constructor
      System.out.println("A new harmonic oscillator object is created.");
   }

   public void step() {
      // modified Euler algorithm
      v = v-k*x*dt;
      x = x+v*dt; // note that updated v is used
      t = t+dt;
   }

   public double analyticPosition(double y0, double v0) {
      return y0*Math.cos(omega0*t)+v0/omega0*Math.sin(omega0*t);
   }

   public double analyticVelocity(double y0, double v0) {
      return -y0*omega0*Math.sin(omega0*t)+v0*Math.cos(omega0*t);
   }
}
```

Exercise 2.9. Simple harmonic oscillator

a. Explain how the implementation of the Euler algorithm in the `step` method of class `SHO` differs from what we did previously.

b. The general form of the analytical solution of (2.11) can be expressed as

$$y(t) = A\cos\omega_0 t + B\sin\omega_0 t, \tag{2.12}$$

where $\omega_0^2 = k/m$. What is the form of $v(t)$? Show that (2.12) satisfies (2.11) with $A = y(t=0)$ and $B = v(t=0)/\omega_0$. These analytical solutions are used in class `SHO`.

c. Write a target class called `SHOApp` that creates an `SHO` object and solves (2.11). Start the ball with displacements of $x = 1$, $x = 2$, and $x = 4$. Is the time it takes for the ball to reach $x = 0$ always the same?

The methods that we have written so far have been non-static methods (except for `main`). As we have seen, these methods cannot be used without first creating or instantiating an object. In contrast, *static* methods can be used directly without first creating an object. A class that is included in the core Java distribution and that we will use often is the `Math` class, which provides many common mathematical methods, including trigonometric, logarithmic, exponential, and rounding operations, and predefined constants. Some examples of the use of the `Math` class include:

```
double theta = Math.PI/4;        // constant pi defined in Math class
double u = Math.sin(theta);      // sine of theta
double v = Math.log(0.1);        // natural logarithm of 0.1
double w = Math.pow(10,0.4);     // 10 to the 0.4 power
double x = Math.atan(3.0);       // inverse tangent
```

Note the use of the dot notation in these statements and the Java convention that constants such as the value of π are written in uppercase letters, that is, `Math.PI`. Exercise 2.10 asks you to read the `Math` class documentation to learn about the methods in the `Math` class. To use these methods we need only to know what mathematical functions they compute; we do not need to know about the details of how the methods are implemented.

Exercise 2.10. The Math class

The documentation for Java is a part of most development environments. It also can be downloaded from `<java.sun.com/docs/>`. Look for API docs and a link to the latest standard edition.

a. Read the documentation of the `Math` class and describe the difference between the two versions of the arctangent method.

b. Write a program to verify the output of several of the methods in the `Math` class.

2.4 Inheritance

The falling ball and the simple harmonic oscillator have important features in common. Both are models of physical systems that represent a physical object as if all its mass were concentrated at a single point. Writing two separate classes by cutting and pasting is straightforward and reasonable because the programs are small and easy to understand. But this approach fails when the code becomes more complex. For example, suppose that you wish to simulate a model of a liquid consisting of particles that interact with one another according to some specified force law. Because such simulations are now standard (see Chapter 9), efficient code for such simulations is available. In principle, it would be desirable to use an already written program, assuming that you understood the nature of such simulations. However, in practice, using someone else's program can require much effort if the code is not organized properly. Fortunately, this situation is changing as more programmers learn object oriented techniques, and write their programs so that they can be used by others without needing to know the details of the implementation.

For example, suppose that you decided to modify an already existing program by changing to a different force law. You change the code and save it under a new name. Later you discover that you need a different numerical algorithm to advance the particles' positions and velocities. You again change the code and save the file under yet another name. At the same time the original author discovers a bug in the initialization method and changes her code. Your code is now out of date because it does not contain the bug fix. Although strict documentation and programming standards can minimize these types of difficulties, a better approach is to use object oriented features such as *inheritance*. Inheritance avoids duplication of code and makes it easier to debug a number of classes without needing to change each class separately.

We now write a new class that *encapsulates* the common features of the falling ball and the simple harmonic oscillator. We name this new class `Particle`. The falling ball and harmonic oscillator that we will define later implement their distinguishing features.

Listing 2.5: `Particle` class.

```
package org.opensourcephysics.sip.ch02;
abstract public class Particle {
   double y, v, t;      // instance variables
   double dt;           // time step

   public Particle() { // constructor
      System.out.println("A new Particle is created.");
   }

   abstract protected void step();

   abstract protected double analyticPosition();

   abstract protected double analyticVelocity();
}
```

The `abstract` keyword allows us to define the `Particle` class without knowing how the `step`, `analyticPosition`, and `analyticVelocity` methods will be implemented. Abstract classes are useful in part because they serve as templates for other classes. The abstract class contains some but not all of what a user will need. By making the class abstract, we must express the abstract idea of "particle" explicitly and customize the abstract class to our needs.

By using inheritance we now *extend* the `Particle` class (the superclass) to another class (the subclass). The `FallingParticle` class shown in Listing 2.6 implements the three abstract methods. Note the use of the keyword `extends`. We also have used a constructor with the initial position and velocity as arguments.

Listing 2.6: `FallingParticle` class.

```
package org.opensourcephysics.sip.ch02;
public class FallingParticle extends Particle {
   final static double g = 9.8;                  // constant
   private double
      y0 = 0, v0 = 0;                            // initial position and velocity

   public FallingParticle(double y, double v) { // constructor
      System.out.println("A new FallingParticle object is created.");
      this.y = y; // instance value set equal to passed value
      this.v = v; // instance value set equal to passed value
      y0 = y;     // no need to use "this" because there is only one y0
      v0 = v;
   }

   public void step() {
      y = y+v*dt; // Euler algorithm
      v = v-g*dt;
```

```
      t = t+dt;
   }

   public double analyticPosition() {
      return y0+v0*t-(g*t*t)/2.0;
   }

   public double analyticVelocity() {
      return v0-g*t;
   }
}
```

`FallingParticle` is a subclass of its superclass `Particle`. Because the methods and data of the superclass are available to the subclass (except those that are explicitly labeled `private`), `FallingParticle` inherits the variables `y`, `v`, `t`, and `dt`.[1]

We now write a target class to make use of our new abstraction. Note that we create a new `FallingParticle`, but assign it to a variable of type `Particle`.

Listing 2.7: `FallingParticleApp` class.

```
package org.opensourcephysics.sip.ch02;
public class FallingParticleApp {           // beginning of class definition
   public static void main(String[] args) { // beginning of method definition
      Particle ball = new FallingParticle(10, 0); // declaration and instantiation
      ball.t = 0;
      ball.dt = 0.01;
      while(ball.y>0) {
         ball.step();
      }
      System.out.println("Results");
      System.out.println("final time = "+ball.t);
      System.out.println("y = "+ball.y+" v = "+ball.v);            // numerical result
      System.out.println("y analytic = "+ball.analyticPosition()); // analytic result
   }                                                               // end of method definitio
} // end of class definition
```

Problem 2.11. Inheritance

a. Run the `FallingParticleApp` class. How can you tell that the constructor of the superclass was called?

b. Rewrite the `SHO` class so that it is a subclass of `Particle`. Remove all unnecessary variables and implement the abstract methods.

c. Write the target class `SHOParticleApp` to use the new `SHOParticle` class. Use the `analyticPosition` and `analyticVelocity` methods to compare the accuracy of the numerical and analytic answers in both the falling particle and harmonic oscillator models.

[1] In this case `Particle` and `FallingParticle` must be in the same package. If `FallingParticle` were in a different package, it would be able to access these variables only if they were declared `protected` or `public`.

d. Try to instantiate a `Particle` directly by calling the `Particle` constructor. Explain what happens when you compile this program.

If you examine the console output in Problem 2.11a, you should find that whenever an object from the subclass is instantiated, the constructor of the super class is executed as well as the constructor of the subclass. You also will find that an abstract class cannot be instantiated directly; it must be extended first.

Exercise 2.12. Extending classes

a. Extend the `FallingParticle` and `SHOParticle` classes and give them names such as `FallingParticleEC` and `SHOParticleEC`, respectively. These subclasses should redefine the `step` method so that it first calculates the new velocity and then calculates the new position using the new velocity, that is,

```
public void step() {
   v = v - g*dt;          // falling ball
   y = y + v*dt;f
   t = t + dt;
}
```

```
public void step() {
   v = v - k*x*dt;        // harmonic oscillator
   x = x + v*dt;
   t = t + dt;
}
```

Methods can be redefined (overloaded) in the subclass by writing a new method in the subclass definition with the same name and parameter list as the super class definition.

b. Confirm that your new `step` method is executed instead of the one in the superclass.

c. The algorithm that is implemented in the redefined step method is known as the *Euler-Cromer* algorithm. Compare the accuracy of this algorithm to the original Euler algorithm for both the falling particle and the harmonic oscillator. We will explore the Euler-Cromer algorithm in more detail in Problem 4.1.

The falling particle and harmonic oscillator programs are simple, but they demonstrate important object-oriented concepts. However, we typically will not build our models using inheritance because our focus is on the physics and not on producing a software library, and also because readers will not use our programs in the same order. We will find that our main use of inheritance will be to extend abstract classes in the Open Source Physics library to implement calculations and simulations by customizing a small number of methods.

So far our target classes have only included one method, `main`. We could have used more than one method, but for the short demonstration and test programs we have written so far, such a practice is unnecessary. When you send a short email to a friend, you are not likely to break up your message into paragraphs. But when you write a paper longer than about a half a page, it makes sense to use more than one paragraph. The same sensitivity to the need for structure should be used in programming. Most of the programs in the following chapters will consist of two classes, each of which will have several instance variables and methods.

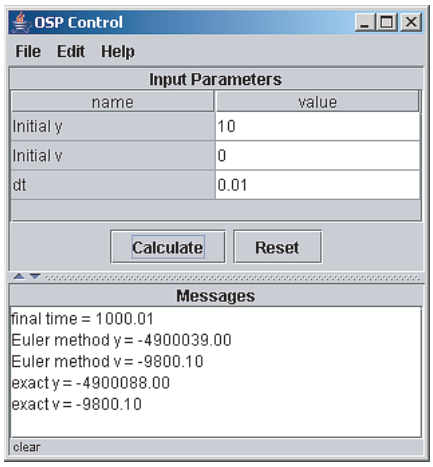

Figure 2.1: An Open Source Physics control that is used to input parameter values and display results.

2.5 The Open Source Physics library

For each exercise in this chapter, you have had to change the program, compile it, and then run it. It would be much more convenient to input initial conditions and values for the parameters without having to recompile. However, a discussion of how to make input fields and buttons using Java would distract us from our goal of learning how to simulate physical systems. Moreover, the code we would use for input (and output) would be almost the same in every program. For this reason input and output should be in separate classes so that we can easily use them in all our programs. Our emphasis will be to describe how to use the Open Source Physics library as a tool for writing graphical interfaces, plotting graphs, and doing visualizations. If you are interested, you can read the source code of the many Open Source Physics classes and can modify or subclass them to meet your special needs.

We first introduce the Open Source Physics library in several simple contexts. Download the Open Source Physics library from `<www.opensourcephysics.org>` and include the library in your development environment. The following program illustrates how to make a simple plot.

Listing 2.8: An example of a simple plot.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.frames.PlotFrame;

public class PlotFrameApp {
   public static void main(String[] args) {
      PlotFrame frame = new PlotFrame("x", "sin(x)/x", "Plot example");
      for(int i = -100;i<=100;i++) {
         double x = i*0.2;
         frame.append(0, x, Math.sin(x)/x);
      }
      frame.setVisible(true);
```

```
        frame.setDefaultCloseOperation(javax.swing.JFrame.EXIT_ON_CLOSE);
    }
}
```

The `import` statement tells the Java compiler where to find the Open Source Physics classes that are needed. A *frame* often is referred to as a window and can include a title and a menu bar as well as objects such as buttons, graphics, and text information. The Open Source Physics frames package defines several frames that contain data visualization and analysis tools. We will use the `PlotFrame` class to plot x-y data. The constructor for `PlotFrame` has three arguments corresponding to the name of the horizontal axis, the name of the vertical axis, and the title of the plot. To add data to the plot, we use the `append` method. The first argument of `append` is an integer that labels a particular set of data points, the second argument is the horizontal (x) value of the data point, and the third argument is the vertical (y) value. The `setVisible(true)` method makes a frame appear on the screen or brings it to the front. The last statement makes the program exit when the frame is closed. What happens when this statement is not included?

The example from the Open Source Physics library in Listing 2.9 illustrates how to control a *calculation* with two buttons, determine the value of an input parameter, and display the result in the text message area.

Listing 2.9: An example of a `Calculation`.

```
package org.opensourcephysics.sip.ch02;
// gets needed classes, asterisk * means get all classes in controls subdirectory

import org.opensourcephysics.controls.*;

public class CalculationApp extends AbstractCalculation {
    public void calculate() { // Does a calculation
        control.println("Calculation button pressed.");
        double x = control.getDouble("x value"); // String must match argument of setValue
        control.println("x*x = "+(x*x));
        control.println("random = "+Math.random());
    }

    public void reset() {
        control.setValue("x value", 10.0);     // describes parameter and sets its value
    }

    public static void main(String[] args) { // Create a calculation control structure using
        CalculationControl.createApp(new CalculationApp());
    }
}
```

`AbstractCalculation` is an *abstract* class, which as we have seen means that it cannot be instantiated directly, and must be extended in order to implement the `calculate` method, that is, you must write (implement) the `calculate` method. You also can write an optional `reset` method, which is called whenever the Reset button is clicked. Finally, we need to create a graphical user interface that will invoke methods when the Calculate and Reset buttons are clicked. This user interface is an object of type `CalculationControl`:

```
CalculationControl.createApp(new CalculationApp());
```

The method `createApp` is a static method that instantiates an object of type `CalculationControl` and returns this object. We could have written

```
CalculationControl control = CalculationControl.createApp(new CalculationApp());
```

which shows explicitly the returned object which we gave the name `control`. However, because we do not use the object `control` explicitly in the `main` method, we do not need to actually declare an object name for it.

Exercise 2.13. CalculationApp

Compile and run `CalculationApp`. Describe what the graphical user interface looks like and how it works by clicking the buttons (see Figure 2.1).

The `reset` method is called automatically when a program is first created and whenever the Reset button is clicked. The purpose of this method is to clear old data and recreate the initial state with the default values of the parameters and instance variables. The default values of the parameters are displayed in the control window so that they can be changed by the user. An example of how to show values in a control is the following:

```
public void reset () {
   control.setValue("x value",10.0); // describes parameter and sets the value
}
```

The string appearing in the `setValue` method must be identical to the one appearing in the `getDouble` method. If you write your own `reset` method, it will override the `reset` method that is already defined in the `AbstractCalculation` superclass.

After the `reset` method stores the parameters in the control, the user can edit the parameters and we can later read these parameters using the `calculate` method:

```
public void calculate () {
   double x = control.getDouble("x value"); // String must match argument of setValue
   control.println("x*x = " + (x*x));
}
```

Exercise 2.14. Changing parameters

a. Run `CalculateApp` to see how the control window can be used to change a program's parameters. What happens if the string in the `getDouble` method does not match the string in the `setValue` method?

b. Incorporate the plot statements in Listing 2.8 into a class that extends the `AbstractCalculation` class and plots the function $\sin kx$ for various values of the input parameter k.

When you run the modified `CalculationApp` in Exercise 2.14, you should see a window with two buttons and an input parameter and its default value. Also, there should be a text area below the buttons where messages can appear. When the calculate button is clicked, the `calculate` method is executed. The `control.getDouble` method reads in values from the control window.

These values can be changed by the user. Then the calculation is performed and the result displayed in the message area using the `control.println` method, similar to the way we used `System.out.println` earlier. If the Reset button is clicked, the message area is cleared and the `reset` method is called.

We now will use a `CalculationControl` to change the input parameters for a falling particle. The modified `FallingParticleApp` is shown in Listing 2.10.

Listing 2.10: `FallingParticleCalcApp` class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.*;

public class FallingParticleCalcApp extends AbstractCalculation { // beginning of class defi
   public void calculate() {
      // gets initial conditions
      double y0 = control.getDouble("Initial y");
      double v0 = control.getDouble("Initial v");
      // sets initial conditions
      Particle ball = new FallingParticle(y0, v0);
      // reads parameters and sets dt
      ball.dt = control.getDouble("dt");
      while(ball.y>0) {
         ball.step();
      }
      control.println("final time = "+ball.t);
      control.println("y = "+ball.y+" v = "+ball.v);            // displays numerical results
      control.println("analytic y = "+ball.analyticPosition()); // displays analytic position
      control.println("analytic v = "+ball.analyticVelocity()); // displays analytic velocity
   }

   public void reset() {
      control.setValue("Initial y", 10);    // sets default input values
      control.setValue("Initial v", 0);
      control.setValue("dt", 0.01);
   }

   public static void main(String[] args) { // creates a calculation control structure using
      CalculationControl.createApp(new FallingParticleCalcApp());
   }
} // end of class definition
```

Exercise 2.15. Input of parameters and initial conditions

a. Run `FallingParticleCalcApp` and make sure you understand how the control works. Try inputting different values of the parameters and the initial conditions.

b. Vary Δt and find the value of t when $y = 0$ to two decimal places.

Exercise 2.16. Displaying floating point numbers

Double precision numbers store 16 significant digits and every digit is included when the number is converted to a string. We can reduce the number of digits that are displayed using the `DecimalFormat` class in the `java.text` package. A formatter is created using a pattern, such as `#0.00` or `#0.00E0`, and this format is applied to a number to produce a string.

```
DecimalFormat decimal2 = new DecimalFormat("#0.00");
double x = 1.0/3.0;
System.out.println("x = "+decimal2.format(x));  // displays 3.33
```

a. Use the `DecimalFormat` class to modify the output from `FallingParticleCalcApp` so that it matches the output shown in Figure 2.1.

b. Modify the output so that results are shown using scientific notation with three decimal places.

c. The Open Source Physics `ControlUtils` class in the controls package contains a static method `f3` that formats a floating point number using three decimal places. Use this method to format the output from `FallingParticleCalcApp`.

You probably have found that it is difficult to write a program so that it ends exactly when the falling ball is at $y = 0$. We could write the program so that Δt keeps changing near $y = 0$ so that the last value computed is at $y = 0$. Another limitation of our programs that we have written so far is that we have shown the results only at the end of the calculation. We could put `println` statements inside the while loop, but it would be better to plot the results and have a table of the data. An example is shown in Listing 2.11.

Listing 2.11: `FallingParticlePlotApp` class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class FallingParticlePlotApp extends AbstractCalculation {
   PlotFrame plotFrame = new PlotFrame("t", "y", "Falling Ball");

   public void calculate() {
      plotFrame.setAutoclear(false); // data not cleared at beginning of each calculation
      // gets initial conditions
      double y0 = control.getDouble("Initial y");
      double v0 = control.getDouble("Initial v");
      // sets initial conditions
      Particle ball = new FallingParticle(y0, v0);
      // gets parameters
      ball.dt = control.getDouble("dt");
      double t = ball.t; // gets value of time from ball object
      while(ball.y>0) {
         ball.step();
         plotFrame.append(0, ball.t, ball.y);
         plotFrame.append(1, ball.t, ball.analyticPosition());
```

```
      }
   }

   public void reset() {
      control.setValue("Initial y", 10);
      control.setValue("Initial v", 0);
      control.setValue("dt", 0.01);
   }

   public static void main(String[] args) { // sets up calculation control structure using t
      CalculationControl.createApp(new FallingParticlePlotApp());
   }
}
```

The two data sets, indexed by 0 and 1, correspond to the numerical data and the analytical results, respectively. The default action in the Open Source Physics library is to clear the data and redraw data frames when the Calculate button is clicked. This automatic clearing of data can be disabled using the `setAutoclear` method. We have disabled it here to allow the user to compare the results of multiple calculations. Data is automatically cleared when the Reset button is clicked.

Exercise 2.17. Data output

a. Run `FallingParticlePlotApp`. Under the Views menu choose DataTable to see a table of data corresponding to the plot. You can copy this data and use it in another program for further analysis.

b. Your plotted results probably look like one set of data because the numerical and analytical results are similiar. Let `dt` = 0.1 and click the Calculate button. Does the discrepancy between the numerical and analytical results become larger with increasing time? Why?

c. Run the program for two different values of `dt`. How do the plot and the table of data differ when two runs are done, first separated without clicking Reset, and then done by clicking Reset between calculations? Make sure you look at the entire table to see the difference. When is the data cleared? What happens if you eliminate the `plotFrame.setAutoclear(false)` statement? When is the data cleared now?

d. Modify your program so that the velocity is shown in a separate window from the position.

2.6 Animation and Simulation

The `AbstractCalculation` class provides a structure for doing a single computation for a fixed amount of time. However, frequently we do not know how long we want to run a program, and it would be desirable if the user could intervene at any time. In addition, we would like to be able to visualize the results of a simulation and do an animation. To do so involves a programming construct called a *thread*. Threads enable a program to execute statements independently of each other as if they were run on separate processors (which would be the case on a multiprocessor computer). We will use one thread to update the model and display the results. The other thread,

the event thread, will monitor the keyboard and mouse so that we can stop the computation whenever we desire.

The `AbstractSimulation` class provides a structure for doing simulations by performing a series of computations (steps) that can be started and stopped by the user using a graphical user interface. You will need to know nothing about threads because their use is "hidden" in the `AbstractSimulation` class. However, it is good to know that the Open Source Physics library is written so that the graphical user interface does not let us change a program's input parameters while the simulation is running. Most of the programs in the text will be done by extending the `AbstractSimulation` class and implementing the `doStep` method as shown in Listing 2.12. Just as the `AbstractCalculation` class uses the graphical user interface of type `CalculationControl`, the `AbstractSimulation` class uses one of type `SimulationControl`. This graphical user interface has three buttons whose labels change depending on the user's actions. As was the case with `CalculationControl`, the buttons in `SimulationControl` invoke specific methods.

Listing 2.12: A simple example of the extension of the `AbstractSimulation` class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.AbstractSimulation;
import org.opensourcephysics.controls.SimulationControl;

public class SimulationApp extends AbstractSimulation {
   int counter = 0;

   public void doStep() { // does a simulation step
      control.println("Counter = "+(counter--));
   }

   public void initialize() {
      counter = control.getInt("counter");
   }

   public void reset() { // invoked when reset button is pressed
      control.setAdjustableValue("counter", 100); // allows dt to be changed after initializa
   }

   public static void main(String[] args) {
      // creates a simulation structure using this class
      SimulationControl.createApp(new SimulationApp());
   }
}
```

Exercise 2.18. AbstractSimulation class

Run `SimulationApp` and see how it works by clicking the buttons. Explain the role of the various buttons. How many times per second is the `doStep` method invoked when the simulation is running?

The buttons in the `SimulationControl` that were used in `SimulationApp` in Listing 2.12 invoke methods in the `AbstractSimulation` class. These methods start and stop threads and

perform other housekeeping chores. When the user clicks the Initialize button, the simulation's `Initialize` method is executed. When the Reset button is clicked, the `reset` method is executed. If you don't write your own versions of these two methods, their default versions will be used. After the initialize button is clicked, it becomes the Start button. After the start button is clicked, it is replaced by a Stop button, and the `doStep` method is invoked continually until the Stop button is clicked. The default is that the frames are redrawn every time `doStep` is executed. Clicking the Step button will cause the `doStep` method to be executed once. The New button changes the Start button to an Initialize button, which forces the user to initialize a new simulation before restarting. Later we will learn how to add other buttons that give the user even more control over the simulation.

A typical simulation needs to (1) specify the initial state of the system in the `initialize` method, (2) tell the computer what to execute while the thread is running in the `doStep` method, and (3) specify what state the system should return to in the `reset` method.

We could modify the falling particle model to use `AbstractSimulation`, but such a modification would not be very interesting because there is only one particle and all motion takes place in one dimension. Instead, we will define a new class that models a ball moving in two dimensions, and we will allow the ball to bounce off the ground and off of the walls.

Listing 2.13: `BouncingBall` class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.display.Circle;

public class BouncingBall extends Circle { // Circle is a class that can draw itself
   final static double g = 9.8;                                    // constant
   final static double WALL = 10;
   private double x, y, vx, vy;                                    // initial position and v

   public BouncingBall(double x, double vx, double y, double vy) { // constructor
      this.x = x;   // sets instance value equal to passed value
      this.vx = vx; // sets instance value equal to passed value
      this.y = y;
      this.vy = vy;
      setXY(x, y); // sets the position using setXY in Circle superclass
   }

   public void step(double dt) {
      x = x+vx*dt; // Euler algorithm for numerical solution
      y = y+vy*dt;
      vy = vy-g*dt;
      if(x>WALL) {
         vx = -Math.abs(vx); // bounce off right wall
      } else if(x<-WALL) {
         vx = Math.abs(vx);  // bounce off left wall
      }
      if(y<0) {
         vy = Math.abs(vy); // bounce off floor
      }
      setXY(x, y);
```

```
    }
}
```

To model the bounce of the ball off a wall, we have added statements such as

```
if (y < 0) vy = Math.abs(vy);
```

This statement insures that the ball will move up if $y < 0$, and is a crude implementation of an elastic collision. (The `Math.abs` method returns the absolute value of its argument.)

Note our first use of the `if` statement. The general form of an `if` statement is as follows:

```
if (boolean_expression) {
   // code executed if boolean expression is true
} else {
   // code executed if boolean expression is false
}
```

We can test multiple conditions by chaining `if` statements.

```
if (boolean_expression) {
   // code goes here
} else if (boolean_expression) {
   // code goes here
} else {
   // code goes here
}
```

If the first boolean expression is true, then only the statements within the first brace will be executed. If the first boolean expression is false, then the second boolean expression in the `else if` expression will be tested, and so forth. If there is an `else` expression, then the statements after it will be executed if all the other boolean expressions are false. If there is only one statement to execute, the braces are optional.

The `BouncingBall` class is similar to the `FallingBall` class except that it extends `Circle`. We inherit from the `Circle` class because this class includes a simple method that allows the object to draw itself in an Open Source Physics frame, called `DisplayFrame`, which we will use in `BouncingBallApp`. In the latter we instantiate `BouncingBall` and `DisplayFrame` objects so that the circle will be drawn at its x-y location when the frame is displayed or while a simulation is running.

To make the animation more interesting, we will animate the motion of many non-interacting balls with random initial velocities. `BouncingBallApp` creates an arbitrary number of non-interacting bouncing balls by creating an array of BouncingBall objects.

Listing 2.14: `BouncingBallApp` class.

```
package org.opensourcephysics.sip.ch02;
import org.opensourcephysics.controls.*;
import org.opensourcephysics.frames.*;

public class BouncingBallApp extends AbstractSimulation {
   // declares and instantiates a window to draw balls
```

```
   DisplayFrame frame = new DisplayFrame("x", "y", "Bouncing Balls");
   BouncingBall[] ball; // declares an array of BouncingBall objects
   double time, dt;

   public void initialize() {
      // sets boundaries of window in world coordinates
      frame.setPreferredMinMax(-10.0, 10.0, 0, 10);
      time = 0;
      frame.clearDrawables(); // removes old particles
      int n = control.getInt("number of balls");
      int v = control.getInt("speed");
      ball = new BouncingBall[n]; // instantiates array of n BouncingBall objects
      for(int i = 0;i<n;i++) {
         double theta = Math.PI*Math.random(); // random angle
         // instantiates the ith BouncingBall object
         ball[i] = new BouncingBall(0, v*Math.cos(theta), 0, v*Math.sin(theta));
         frame.addDrawable(ball[i]);           // // adds ball to frame so that it will be d
      }
      // decimalFormat instantiated in superclass and used to format numbers conveniently
      frame.setMessage("t = "+decimalFormat.format(time)); // appears in lower right hand co
   }

   public void doStep() {                                  // invoked every 1/10 second by ti
      for(int i = 0;i<ball.length;i++) {
         ball[i].step(dt);
      }
      time += dt;
      frame.setMessage("t="+decimalFormat.format(time));
   }

   public void startRunning() { // invoked when start or step button is pressed
      dt = control.getDouble("dt");
   }                    // gets time step

   public void reset() { // invoked when reset button is pressed
      control.setAdjustableValue("dt", 0.1); // allows dt to be changed after initializaton
      control.setValue("number of balls", 40);
      control.setValue("speed", 10);
   }

   public static void main(String[] args) {  // sets up animation control structure using th
      SimulationControl.createApp(new BouncingBallApp());
   }
}
```

Because we will advance the dynamical variables of each ball using a loop, we store them in an *array*. An array such as `ball` is a data structure that holds many objects (or primitive data) of the same type. The elements of an array are accessed using an index in square brackets. The index begins at 0 and ends at the length of the array minus 1. Arrays are created with the `new` operator and have several properties such as `length`. We will discuss arrays in more detail in Section 4.4.

In Listing 2.13 we represent each ball as an object of type `BouncingBall` in an array. This use of objects is appealing, but for better performance, it usually is better to store the positions and the velocities of the balls in an array of doubles. In Chapter 9 we will simulate a system of N mutually interacting particles. Because computational speed will be very important in this case, we will not allocate separate objects for each particle, and instead will treat the system of N particles as one object.

The `initialize` method in `BouncingBallApp` reads the number of particles and creates an array of the appropriate length. Creating an array sets primitive variables to zero and object values to null. For this reason we next loop to create the balls and add each ball to the frame. We place each ball initially at (0,0) with a random velocity. To produce random angles for the initial velocity, the `Math.random()` method is used. This method returns a random double between 0 and 1, not including the exact value 1. We define the random angle to be between 0 and π so that the initial vertical component of the velocity is positive. Clicking the Initialize button removes old objects from the drawing.

Most programming languages, including Java, use pixels to define the location on a window, with the origin at the upper left-hand corner and the vertical coordinate increasing in the downward direction. This choice of coordinates is usually not convenient in physics, and it often is more convenient to choose coordinates such that the vertical coordinate increases upward. The `Circle.setXY` method uses *world* or physical coordinates to set the position of the circle, and its implementation converts these coordinates to pixels so that the Java graphics methods can be used. In `initialize` we set the boundaries for the *world* coordinates using the `setPreferredMinMax` method whose arguments are the minimum x coordinate, maximum x coordinate, minimum y coordinate, and maximum y coordinate, respectively.

The `doStep` method implements a straightforward loop to advance the dynamical state of each ball in the array. It then advances the time and displays the time in the frame. Frames are automatically redrawn each time the `doStep` method is executed.

Finally, we note that there are two types of input parameters. Some parameters, such as the number of particles, determine properties of the model that should not be changed after the model has been created. We refer to these parameters as *fixed*, because their values should be determined when the model is initialized. Other parameters, such as the time step Δt, can be changed between computations, but should not be changed during a computation. For example, if the time step is changed while a differential equation is being solved, one variable might be advanced using the old value of the time step while another variable is advanced using the new value. This type of synchronization error can be avoided by reading the parameters before the `doStep` method is executed. If you wish to allow a parameter to be changed between computations, you can use the optional `startRunning` method. This method is invoked once when the Step button is clicked and once when the Run button is clicked. In other words, this method is called before the thread starts and insures that the simulation has the opportunity to read the most recent values.

In `BouncingBallApp` the time step `dt` is set using the `setAdjustableValue` method rather than the `setValue` method. Parameters that are set using `setAdjustableValue` are editable in the `SimulationControl` after the program has been initialized, whereas those that are set using `setValue` are only editable before the program has been initialized.

Exercise 2.19. Follow the bouncing balls

a. Run `BouncingBallApp` and try the different buttons and note how they affect the input parameters.

b. Add the statement, `enableStepsPerDisplay(true)` to the `reset` method, and run your program again. You should see a new input in the control window that lets you change the number of simulation steps that are computed between redrawing the frame. Vary this input and note what happens.

c. What is wrong with the physics of the simulation?

d. Add a method to the `BouncingBall` class to calculate and return the total energy. Sum the energy of the balls in the program's `doStep` method and display this value in the message box. Does the simple model make sense?

e. Look at the source code for the `setXY` method. If you are using an Integrated Development Environment (IDE), finding the method and looking at the source code is easy. What would you need to do to change the radius of the circle that is drawn?

Many of the visualization components in the Open Source Physics library are written using classes provided by others including programmers at Sun Microsystems. The goal of this library is to make it easier for you to begin writing your own programs. You are encouraged to look under the hood as you gain experience. The Open Source Physics controls and visualizations will almost always inherit from the `JFrame` class. Drawing is almost always done on a `DrawingPanel` which inherits from the `JPanel` class. Both these superclasses are defined in the `javax.swing` package.

Exercise 2.20. Peeking into Open Source Physics

a. Look at the source code for `PlotFrame` (in the frames package) and follow its inheritance until you reach the `JFrame` class. How may subclasses are there between `JFrame` and `PlotFrame`. Follow the inheritance from `SimulationControl` (in the controls package) to `JFrame`. Describe in general terms what features are added in each subclass.

b. Read through the different methods in `PlotFrame`. Don't worry about how the methods are implemented, but try to understand what they do. Which methods have not yet appeared in a program listing? When might you use them?

c. Look at the source code for `PlottingPanel` (in the display package), which is used in many of the frames. Follow its inheritance until you reach the `JPanel` class. Do you see why we have not described the `PlottingPanel` class in detail? Look through the various methods, and describe in your own words what several of them do and how they might be used.

d. Find the closest common ancestor (superclass) for `JFrame` and `JPanel` in the core Java library. Note that all objects have `Object` as a common ancestor.

2.7 Model-View-Controller

Developing large software programs is best viewed as a design process. One criterion for good design is the reuse of data structures and behaviors that can facilitate reuse. Separating the

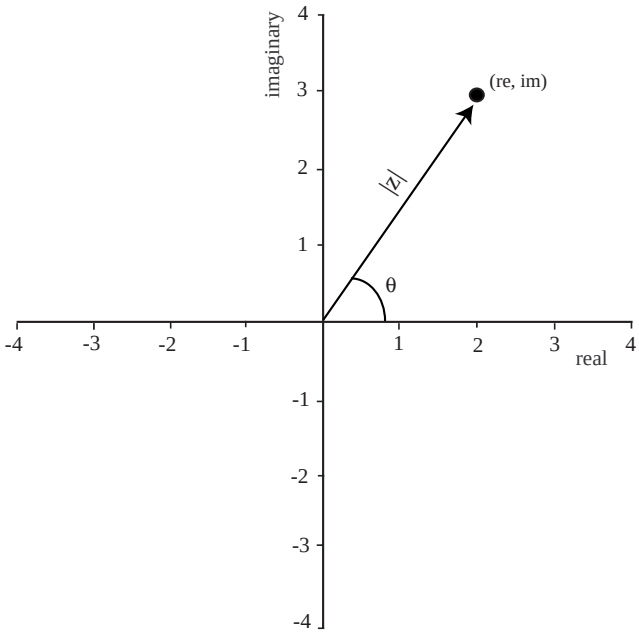

Figure 2.2: A complex number, z, can be defined by its real and imaginary parts, *real* and *imag*, respectively, or by its magnitude, $|z|$, and phase angle, θ.

physics (the model) from the user interface (the controller) and the data visualization (the view) facilitate good design. In Open Source Physics the control object is responsible for handling user initiated events such as button clicks and passing them to other objects. The plots that we have constructed present visual representations of the data and are examples of a *view*. By using this design strategy, it is possible to have multiple views of the same data. For example, we can show a plot and a table view of the same data. The physics is expressed in terms of a *model* which contains the data and provides the methods by which the data can change.

At this point we have described a large fraction of the Java syntax and Open Source Physics tools that we will need in the rest of this book. One important topic that we still need to discuss is the use of *interfaces*. There also is much more in the Open Source Physics library that we can use. For example, there are classes to draw and manipulate lattices as well as classes to iterate differential equations more accurately than the Euler method used in this chapter.

At this stage, we hope that you have gained a feel for how Java works, and can focus on the physics in the rest of the text. Additional aspects of Java will be taught by example as they are needed.

Appendix 2A: Complex numbers

Complex numbers are used in physics to represent quantities such as alternating currents and quantum mechanical wave functions which have an amplitude and phase (see Figure 2.2). Java does not provide a complex number as a primitive data type, so we will write a class that implements some common complex arithmetic operations. This class is an explicit example of the fact that classes are effectively new programmer-defined types.

If our new class is called `Complex`, we could test it by using code such as the following:

```
package org.opensourcephysics.sip.ch02;
public class ComplexApp {
   public static void main(String[] args) {
      Complex a = new Complex(3.0, 2.0);  // complex number 3 + i2
      Complex b = new Complex(1.0, -4.0); // complex number 1 - i4
      System.out.println(a); // print a using a.toString()
      System.out.println(b); // print b using b.toString()
      Complex sum = b.add(a); // add a to b
      System.out.println(sum); // print sum
      Complex product = b.multiply(a); // multiply b by a
      System.out.println(product); // print product
      a.conjugate();               // complex conjugate of a
      System.out.println(a);
   }
}
```

Because the methods of class `Complex` are not static, we must first instantiate a `Complex` object with a statement such as

```
Complex a = new Complex(3.0, 2.0);
```

The variable `a` is an object of class `Complex`. As before, we can think of `new` as creating the instance variables and memory of the object. Compare the form of this statement to the declaration,

```
double x = 3.0;
```

A variable of class type `Complex` is literally more complex than a primitive variable because its definition also involves associated methods and instance variables.

Note that we have first written a class that uses the `Complex` class before we have actually written the latter. Although programming is an iterative process, it is usually a good idea to think about how the objects of a class are to be used first. Exercise 2.21 encourages you to do so.

Exercise 2.21. Complex number test

What will be the output when `ComplexApp` is run? Make reasonable assumptions about how the methods of the `Complex` class will perform using your knowledge of Java and complex numbers.

We need to define methods that add, multiply, and take the conjugate of complex numbers and define a method that prints their value. We next list the code for the `Complex` class.

Listing 2.15: Listing of the `Complex` class.

```
package org.opensourcephysics.sip.ch02;
public class Complex {
   private double real = 0;
   private double imag = 0; // real, imag are instance variables

   public Complex() {
      this(0, 0); // invokes second constructor with 0 + i0
   }

   public Complex(double real, double imag) {
      this.real = real;
      this.imag = imag;
   }

   public void conjugate() {
      imag = -imag;
   }

   public Complex add(Complex c) {
      // result also is complex so need to introduce another variable of type Complex
      Complex sum = new Complex();
      sum.real = real+c.real;
      sum.imag = imag+c.imag;
      return sum;
   }

   public Complex multiply(Complex c) {
      Complex product = new Complex();
      product.real = (real*c.real)-(imag*c.imag);
      product.imag = (real*c.imag)+(imag*c.real);
      return product;
   }

   public String toString() {
      // note example of method overriding
      if(imag>=0) {
         return real+" + i"+Math.abs(imag);
      } else {
         return real+" - i"+Math.abs(imag);
      }
   }
}
```

The `Complex` class defines two constructors that are distinguished by their parameter list. The constructor with two arguments allows us to initialize the values of the instance variables. Notice how the class *encapsulates* (hides) both the data and the methods that characterize a complex number. That is, we can use the `Complex` class without any knowledge of how its methods are implemented or how its data is stored.

The general features of this class definition are as before. The variables `real` and `imag` are the instance variables of class `Complex`. In contrast, the variable `sum` in method `add` is a *local* variable because it can be accessed only within the method in which it is defined.

The most important new feature of the `Complex` class is that the `add` and `multiply` methods return new `Complex` objects. One reason we need to return a variable of type `Complex` is that a method returns (at most) a *single* value. For this reason we cannot return both `sum.real` and `sum.imag`. More importantly, we want the sum of two complex numbers to also be of type `Complex` so that we can add a third complex number to the result. Note also that we have defined `add` and `multiply` so that they do not change the values of the instance variables of the numbers to be added, but create a new complex number that stores the sum.

Exercise 2.22. Complex numbers

Another way to represent complex numbers is by their magnitude and phase, $|z|e^{\imath\theta}$. If $z = a + \imath b$, then

$$|z| = \sqrt{a^2 + b^2}, \tag{2.13a}$$

and

$$\theta = \arctan \frac{b}{a}. \tag{2.13b}$$

a. Write methods to get the magnitude and phase of a complex number, `getMagnitude` and `getPhase`, respectively. Add test code to invoke these methods. Be sure to check the phase in all four quadrants.

b. Create a new class named `ComplexPolar` that stores a complex number as a magnitude and phase. Define methods for this class so that it behaves the same as the `Complex` class. Test this class using the code for `ComplexApp`.

This example of the `Complex` class illustrates the nature of objects, their limitations, and the tradeoffs that enter into design choices. Because accessing an object requires more computer time than accessing primitive variables, it is faster to represent a complex number by two doubles, corresponding to its real and imaginary parts. Thus N complex data points could be represented by an array of $2N$ doubles, with the first N values corresponding to the real values. Considerations of computational speed are important only if complex data types are used extensively.

References and Suggestions for Further Reading

By using the Open Source Physics library we have hidden most of the Java code needed to use threads, and we have only touched on the graphical capabilities of Java. See the *Open Source Physics: A User's Guide with Examples* for a description of additional details on how threads and the other Open Source Physics tools are implemented and used.

There are many good books on Java graphics and Java threads. We list a few of our favorites in the following.

David M. Geary, *Graphic Java: Vol. 2, Swing*, third edition, Prentice Hall (1999).

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

Scott Oaks and Henry Wong, *Java Threads*, third edition, O'Reilly (2004).