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 have only touched on the graphical capabilities of Java. See *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*, 3rd ed. (Prentice Hall, 1999).

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

Scott Oaks and Henry Wong, *Java Threads*, 3rd ed. (O'Reilly, 2004).

CHAPTER 3 Simulating Particle Motion

We discuss several numerical methods needed to simulate the motion of particles using Newton's laws and introduce *interfaces*, an important Java construct that makes it possible for unrelated objects to declare that they perform the same methods.

3.1 ■ MODIFIED EULER ALGORITHMS

To motivate the need for a general differential equation solver, we discuss why the simple Euler algorithm is insufficient for many problems. The Euler algorithm assumes that the velocity and acceleration do not change significantly during the time step Δt. Thus, to achieve an acceptable numerical solution, the time step Δt must be chosen to be sufficiently small. However, if we make Δt too small, we run into several problems. As we do more and more iterations, the round-off error due to the finite precision of any floating point number will accumulate, and eventually the numerical results will become inaccurate. Also, the greater the number of iterations, the greater the computer time required for the program to finish. In addition to these problems, the Euler algorithm is unstable for many systems, which means that the errors accumulate exponentially, and thus the numerical solution becomes inaccurate very quickly. For these reasons more accurate and stable numerical algorithms are necessary.

To illustrate why we need algorithms other than the simple Euler algorithm, we make a very simple change in the Euler algorithm and write

$$v(t + \Delta t) = v(t) + a(t)\Delta t \tag{3.1a}$$
$$y(t + \Delta t) = y(t) + v(t + \Delta t)\Delta t, \tag{3.1b}$$

where a is the acceleration. The only difference between this algorithm and the simple Euler algorithm,

$$v(t + \Delta t) = v(t) + a(t)\Delta t \tag{3.2a}$$
$$y(t + \Delta t) = y(t) + v(t)\Delta t, \tag{3.2b}$$

is that the computed velocity at the end of the interval, $v(t + \Delta t)$, is used to compute the new position, $y(t + \Delta t)$ in (3.1b). As we found in Problem 2.12 and will see in more detail in Problem 3.1, this modified Euler algorithm is significantly better for oscillating systems. We refer to this algorithm as the Euler–Cromer algorithm.