to write. In addition, for each problem we would need to write and debug new code to implement the numerical algorithm. The complications become worse for better algorithms, most of which are algebraically more complex. Moreover, the numerical solution of simple first-order differential equations is a well-developed part of numerical analysis, and thus there is little reason to worry about the details of these algorithms now that we know how they work. In Section 3.5 we will introduce an interface for solving the differential equations associated with Newton's equations of motion. Before we do so we discuss a few features of arrays that we will need.

As we discussed on page 39, ordered lists of data are most easily stored in arrays. For example, if we have an array variable named x, then we can access its first element as `x[0]`, its second element as `x[1]`, etc. All elements must be of the same data type, but they can be just about anything: primitive data types such as doubles or integers, objects, or even other arrays. The following statements show how arrays of primitive data types are defined and instantiated:

```
// x defined to be an array of doubles
double[] x;
double x[];                              // same meaning as double [] x
// x array created with 32 elements
x = new double[32];
// y array defined and created in one statement
double[] y = new double[32];
int[] num = new int[100];                // array of 100 integers
double [] x,y                            // preferred notation
// same meaning as double[] x,y
double x[], y[]
// array of doubles specified by two indices
double[][] sigma = new double[3][3];
// reference to first row of sigma array
double[] row = sigma[0];
```

We will adopt the syntax `double[] x` instead of `double x[]`. The array index starts at zero, and the largest index is one less than the number of elements. Note that Java supports multiple array indices by creating arrays of arrays. Although `sigma[0][0]` refers to a single value of type `double` in the `sigma` object, we can refer to an entire row of values in the `sigma` object using the syntax `sigma[i]`.

As shown in Chapter 2, arrays can contain objects such as bouncing balls.

```
// array of two BouncingBall objects
BouncingBall [] ball = new BouncingBall[2];
ball[0] = new BouncingBall(0,10.0,0,5.0);      // creates first ball
ball[1] = new BouncingBall(0,-13.0,0,7.0);     // creates second ball
```

The first statement allocates an array of `BouncingBall` objects, each of which is initialized to null. We need to create each object in the array using the `new` operator.

The numerical solution of an ordinary differential equation (frequently called an ODE) begins by expressing the equation as several first-order differential equations. If the highest derivative in the ODE is order n (for example, $d^n x/dt^n$), then it can be shown that the ODE can be written equivalently as n first-order differential equations. For example, Newton's equation of motion is a second-order differential equation and can be written as two first-order differential equations for the position and velocity in each spatial dimension. For

example, in one dimension we can write

$$\frac{dy}{dt} = v(t) \tag{3.5a}$$

$$\frac{dv}{dt} = a(t) = F(t)/m. \tag{3.5b}$$

If we have more than one particle, there are additional first-order differential equations for each particle. It is convenient to have a standard way of handling all these cases.

Let us assume that each differential equation is of the form

$$\frac{dx_i}{dt} = r_i(x_0, x_i, x_2, \ldots, x_{n-1}, t), \tag{3.6}$$

where x_i is a dynamical variable such as a position or a velocity. The rate function r_i can depend on any of the dynamical variables including the time t. We will store the values of the dynamical variables in the `state` array and the values of the corresponding rates in the `rate` array. In the following we show some examples:

```
// one particle in one dimension:
state[0]  // stores x
state[1]  // stores v
state[2]  // stores t (time)
// one particle in two dimensions:
state[0]  // stores x
state[1]  // stores vx
state[2]  // stores y
state[3]  // stores vy
state[4]  //stores t
// two particles in one dimension:
state[0]  // stores x1
state[1]  // stores v1
state[2]  // stores x2
state[3]  // stores v2
state[4]  // stores t
```

Although the Euler algorithm does not assume any special ordering of the state variables, we adopt the convention that a velocity rate follows every position rate in the state array so that we can efficiently code the more sophisticated numerical algorithms that we discuss in Appendix 3A and in later chapters. To solve problems for which the rate contains an explicit time dependence, such as a driven harmonic oscillator (see Section 4.4), we store the time variable in the last element of the state array. Thus, for one particle in one dimension, the time is stored in `state[2]`. In this way we can treat all dynamical variables on an equal footing.

Because arrays can be arguments of methods, we need to understand how Java passes variables from the class that calls a method to the method being called. Consider the following method:

```
public void example(int r, int s[]) {
   r = 20;
   s[0] = 20;
}
```