What do you expect the output of the following statements to be?

```
int x = 10;
int[] y = {10};    // array of one element initialized to y[0] = 10
example(x, y);
System.out.println("x = " + x + " y[0] = " + y[0]);
```

The answer is that the output will be `x = 10, y[0] = 20`. Java parameters are "passed by value," which means that the values are copied. The method cannot modify the value of the x variable because the method received only a copy of its value. In contrast, when an object or an array is in a method's parameter list, Java passes a copy of the reference to the object or the array. The method can use the reference to read or modify the data in the array or object. For this reason the `step` method of the ODE solvers, discussed in Section 3.6, does not need to explicitly return an updated `state` array, but implicity changes the contents of the state array.

Exercise 3.5 Pass by value
As another example of how Java handles primitive variables differently from arrays and objects, consider the statements

```
int x = 10;
int y = x;
x = 20;
```

What is y? Next consider the statements

```
// declares an array of one element initialized to the value 10
int[] x = {10};
int[] y = x;
x[0] = 20;
```

What is `y[0]`? ■

We are now ready to discuss the classes and interfaces from the Open Source Physics library for solving ordinary differential equations.

3.5 ■ THE ODE INTERFACE

To introduce the ODE interface, we again consider the equations of motion for a falling particle. We use a state array ordered as $s = (y, v, t)$, so that the dynamical equations can be written as:

$$\dot{s_0} = s_1 \tag{3.7a}$$

$$\dot{s_1} = -g \tag{3.7b}$$

$$\dot{s_2} = 1. \tag{3.7c}$$

The `ODE` interface enables us to encapsulate (3.7) in a class. The interface contains two methods, `getState` and `getRate`, as shown in Listing 3.4.

Listing 3.4 The ODE interface.

```
package org.opensourcephysics.numerics;

public interface ODE {
   public double[] getState();
   public void getRate(double[] state, double[] rate);
}
```


The `getState` method returns the state array $(s_0, s_1, \ldots, s_n)$. The `getRate` method evaluates the derivatives using the given state array and stores the result in the rate array, $(\dot{s}_0, \dot{s}_1, \ldots, \dot{s}_n)$.

An example of a Java class that implements the `ODE` interface for a falling particle is shown in Listing 3.5.

Listing 3.5 Example of the implementation of the `ODE` interface for a falling particle.

```
package org.opensourcephysics.sip.ch03;
import org.opensourcephysics.numerics.*;

public class FallingParticleODE implements ODE {
   final static double g = 9.8;
   double[] state = new double[3];

   public FallingParticleODE(double y, double v) {
      state[0] = y;
      state[1] = v;
      state[2] = 0;              // initial time
   }

   public double[] getState() { // required to implement ODE interface
      return state;
   }

   public void getRate(double[] state, double[] rate) {
      rate[0] = state[1]; // rate of change of y is v
      rate[1] = -g;
      rate[2] = 1;        // rate of change of time is 1
   }
}
```

3.6 ■ THE ODESOLVER INTERFACE

There are many possible numerical algorithms for advancing a system of first-order ODEs from an initial state to a final state. The Open Source Physics library defines ODE solvers such as `Euler` and `EulerRichardson` as well as `RK4`, a fourth-order algorithm that is discussed in Appendix 3A. You can write additional classes for other algorithms if they are needed. Each of these classes implements the `ODESolver` interface, which is defined in Listing 3.6.

Listing 3.6 The ODE solver interface. Note the four methods that must be defined.

```
package org.opensourcephysics.numerics;

public interface ODESolver {
   public void initialize(double stepSize);
   public double step();
   public void setStepSize(double stepSize);
```