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 was 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 8), 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 Your code is now out of date because it does not contain the bug fix.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
   // initial position and velocity
   private double y0 = 0, v0 = 0;

   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;
// beginning of class definition
public class FallingParticleApp {
   // beginning of method definition
   public static void main(String[] args) {
      // declaration and instantiation
      Particle ball = new FallingParticle(10, 0);
      ball.t = 0;
      ball.dt = 0.01;
      while(ball.y>0) {
```

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