Chapter 3

Inheritance and Interfaces

©2002 by Harvey Gould, Jan Tobochnik, and Wolfgang Christian
16 February 2002

We discuss some further examples of inheritance and then introduce the idea of an interface.

3.1 Inheritance

In Section 2.4 we introduced the idea of *inheritance*, one of the cornerstones of object oriented programming. Inheritance allows one or more subclasses to perform methods and access data in the parent class or *superclass*. If class B and class C inherit from class A, both B and C automatically inherit A's methods and (public and protected) instance variables, but not A's constructors.

The general rule is that subclasses are more specialized than superclasses. For example, in introductory physics we often define a particle to be a structureless mass with specified coordinates. A particle falling under the action of gravity without drag and a particle falling under the action of gravity with drag are specialized examples of particles. We can define a *class hierarchy* using a `Particle` superclass and two subclasses, `ParticleFree` and `ParticleDrag`, to make the relationships explicit.

At this early stage of learning Java, our examples of inheritance will be relatively simple and the advantages of inheritance will not be much better than "copying and pasting." Many of our future uses of inheritance will not be obvious and will involve implicit inheritance from `Object` and the use of inheritance in the graphical input and output classes that we will develop (see Chapter 4).

Let's return to the particle example considered in Section 2.4.1 and consider the fall of a feather and rock. For simplicity, we will ignore the effect of drag resistance on the fall of the rock, but obviously we have to take into account the effect of drag resistance due to the earth's atmosphere on the fall of a feather. Both the rock and the feather are examples of particles (if we ignore their internal motion), but the motion of their center of mass can be quite different. For this reason we define the `Particle class` as follows:

```
package org.opensourcephysics.sip.ch3;    // note package name

public abstract class Particle {
   protected double mass = 1.0;
   protected double y,v,a; // position, velocity , acceleration
   // static constants are usually upper case; break convention here
   public static final double g = 9.8;

   public Particle(double _y, double _v) {
      y = _y;
      v = _v;
   }

   public abstract void move(double dt);

   public String toString() {
      return "y = " + y + "," + "v = " + v + "," + "a = " + a;
   }
}
```

Note that `Particle` is an*abstract* class, which means that it cannot be used directly. Also note that we have also declared the `move` method to be abstract. In the latter context, the *abstract* keyword is used whenever we wish to declare a method, but do not yet know how it will be implemented. In this case we know that all particles will move, but we do not yet know how they will move. Abstract methods, such as `move`, contain only a header with no method body. The advantage of including the `move` method in `Particle` is that all classes that inherit from `Particle` can differ in the way this method is implemented. By creating abstract classes, we require common methods that all subclasses must have.

You might wonder why we didn't define the move method as `public void move(double dt)`, in which case we would not have to make `Particle` abstract. The reason is that `Particle` does not express a particular implementation of a particle, and hence creating a `Particle` object makes no sense. A class that declares at least one abstract method must declare itself to be abstract in the class header and cannot be instantiated.

We defined the instance variables `mass`, `y`, `v`, and `a` as *protected*, because all classes in the package as well as all their subclasses might need to have access to these variables.

Exercise 3.1. Directory hierarchy

Given that there is a directory named `org` that contains a directory named `opensourcephysics`, what is the directory hierarchy implied by the package statement in the file Particle.java? The acronym `sip` stands for Simulations in Physics.

We now redefine the `ParticleFree` class first introduced on page 24 by inheriting from `Particle` and assuming that the acceleration of the particle is due only to gravity.

```
package org.opensourcephysics.sip.ch3;

public class ParticleFree extends Particle {

   public ParticleFree(double _y, double _v) {
```

```
    super(_y,_v);     // use constructor from super (Particle) class
    a = -g;           // note choice of reference system
  }

  public void move(double dt) {
    y = y + v*dt + 0.5*a*dt*dt;
    v = v + a*dt;
  }
}
```

Subclasses that define bodies for all abstract methods are said to be *concrete* manifestations of the abstract superclass.

The keyword `super` refers to a member of the superclass, which in this case is `Particle`. To ensure that the subclass is initialized correctly, we need to initialize the superclass first. By invoking the constructor from the superclass, we guarantee that we instantiate the member variables of the superclass. (If we do not explicitly make a call to the superclass constructor in the subclass constructor, Java would do so by default if the constructor has no arguments.)

To discuss the fall of the feather, we have to take into account the effects of drag resistance. Such effects are discussed in more detail in Chapter 5. For particles near the earth's surface, the direction of the retarding drag force due to air resistance is opposite to the motion of the particle. For a falling body the direction of the drag force F_d is upward, which we take to be positive, and the total force F on the particle can be written as

$$F = -mg + F_d, \tag{3.1}$$

where we have adopted a coordinate system where y increases upward, and thus $v < 0$ for a falling object. We know that the greater the speed of an object, the greater the magnitude of F_d. For simplicity, we will assume that $F_d/m = C|\mathbf{v}|$, that is, the drag force is proportional to the speed of the falling particle. For our choice of coordinate system and for a falling body, we have $|\mathbf{v}| = -v$, and the rate of change of the velocity becomes

$$\frac{dv}{dt} = -g - Cv. \tag{3.2}$$

Note that the falling object will eventually reach a terminal velocity v_t when $dv/dt = 0$. From (3.2) we see that $v_t = -g/C$ and we can express (3.2) in the convenient form (see (5.23a))

$$\frac{dv}{dt} = -g\big(1 - \frac{v}{v_t}\big), \tag{3.3}$$

with the initial condition $v(t = 0) = v_0$.

In Section 5.7 we will learn how to solve (3.2) on a computer using various numerical methods. However, (3.2) is sufficiently easy that we can guess its analytical solution. If you do not understand the following derivation, just look at the analytical solutions (3.7) and (3.8) which we will use to define the `move` method for a particle falling with drag resistance.

If the first term in (3.3) were absent, the derivative of the velocity would be proportional to the velocity, and the solution would be of the form $v(t) = Ae^{-\alpha t}$. So we guess that the general

form of the solution to (3.2) is

$$v(t) = Ae^{-\alpha t} + B. \tag{3.4}$$

It is easy to check that the general form (3.4) satisfies (3.3) and the initial condition if we choose A, B, and α appropriately. First we take the derivative of (3.4) and substitute it into (3.3):

$$\begin{aligned}\frac{dv(t)}{dt} &= -A\alpha e^{-\alpha t} = -g + \frac{gv(t)}{v_t} \\ &= -g + \frac{g}{v_t}(Ae^{-\alpha t} + B).\end{aligned} \tag{3.5}$$

Equation (3.5) can be satisfied for all values of the time t if

$$-A\alpha = \frac{g}{v_t}A \tag{3.6a}$$

$$0 = -g + \frac{g}{v_t}B. \tag{3.6b}$$

These two equations tell us that $\alpha = -g/v_t$ and $B = v_t$. The constant A is determined by the initial condition, $v(t=0) = v_0$. Hence, we find that the general solution for $v(t)$ is

$$v(t) = v_t + (v_0 - v_t)e^{gt/v_t} = v_t + (v_0 - v_t)e^{-gt/|v_t|}. \tag{3.7}$$

If you are not very familiar with differential equations and their solutions, don't worry. We will learn in Section 5.2 how to solve differential equations such as (3.3) numerically using simple methods.

Exercise 3.2. Analytical solution

Fill in the missing steps in the derivation of (3.7) and show that $v(t)$ has the expected behavior at $t = 0$ and at $t \to \infty$. Also show that the solution to the differential equation $v(t) = dy/dt$ is

$$y(t) = y_0 + v_t t - \frac{v_t(v_0 - v_t)}{g}(1 - e^{gt/v_t}). \tag{3.8}$$

We now use the analytical solution to define a `ParticleDrag` class (a particle with drag resistance):

```
package org.opensourcephysics.sip.ch3;

public class ParticleDrag extends Particle {
  double vt;          // terminal velocity
  final double g = 9.8;

  public ParticleDrag(double _y, double _v, double _vt) {
    super(_y,_v);     // use constructor from super (Particle) class
    vt = _vt;
  }

  public void move(double dt) {
```

```
        y = y + vt*dt - vt*((v - vt)/g)*(1.0 - Math.exp(g*dt/vt));
        v = vt + (v - vt)*Math.exp(g*dt/vt);
    }
}
```

Note that the `move` method computes the velocity and the position at the end of the time step given their values at the beginning of the time step. For that reason, $v_0 \to v$, $y_0 \to y$, and $t \to \Delta t$ in going from (3.7) and (3.8) to the corresponding statements in the `move` method.

It is not possible to instantiate an object of type `Particle`, because `Particle` is an abstract class. Instead, we write a target class that instantiates particular members (objects) of `ParticleFree` and `ParticleDrag`.

```
package org.opensourcephysics.sip.ch3;

public class ParticleMoverApp {
    double minimumHeight, dt;

    public ParticleMoverApp(double _minimumHeight, double _dt) {
        minimumHeight = _minimumHeight;
        dt = _dt;
    }

    // return total time for particle to reach minimum height
    public double calculateTotalTime(Particle p) {
        double time = 0;
        while (p.y >= minimumHeight) {
            p.move(dt);
            time += dt;
        }
        return time;
    }

    public static void main(String[] args) {
        ParticleMoverApp particleMover = new ParticleMoverApp(0,0.01);
        Particle rock = new ParticleFree(20,0);
        double totalTime = particleMover.calculateTotalTime(rock);
        System.out.println("Time for rock to hit ground =" + totalTime);
        Particle feather = new ParticleDrag(20,0,-10);
        totalTime = particleMover.calculateTotalTime(feather);
        System.out.println("Time for feather to hit ground =" + totalTime);
    }
}
```

Note that the correct `move` method is automatically applied to each particle based on the subclass to which it belongs. This ability to choose the appropriate method depending on its subclass is known as *polymorphism*. (The mechanism that makes this choice automatic is called late or dynamical binding, which means that the appropriate connection or binding of the method to the correct object is not done until run time.) The advantage of this feature of object oriented programming is that it makes it very easy to modify our programs. For example, if we wanted to

introduce a particle whose drag resistance varies quadratically with the speed, we could define a new subclass and modify the `move` method (see Section 5.7).

The relation of `ParticleFree` and `ParticleDrag` is shown in the inheritance diagram of Figure 3.1. Note that the upcast can occur in a statement such as

```
Particle p = new ParticleFree(20,0);
```

The result is that a `ParticleFree` object is created and the resulting reference is assigned to a `Particle`. This assignment would seem to be an error because we have assigned one type to another, but such an assignment is correct because a `ParticleFree` is a `Particle` by inheritance. Suppose we later write

```
p.move(0.01);
```

Because of polymorphism and dynamic binding, the correct move method is used.

Note also that we did not instantiate a particle as a particular member of the Particle class, but instead instantiated particular kinds of particles. Analogously, we might define an Animal class and then subclass a Dog, Cat, and Cow from Animal.

3.2 Interfaces

An *interface* is a declaration of the type of functionality that an object is able to provide. Although an interface has a superficial similarity to a class containing all abstract methods, an interface is fundamentally different. Abstract classes work in conjunction with inheritance to define relationships. For example, a free particle is a type of particle. Why are interfaces useful? Interfaces add functionality to a class without having to worry about class inheritance. In other words, functionality can be added anywhere within the class hierarchy. In particular, a class can only inherit from one superclass, but it can *implement* many different interfaces.

Any object can implement any interface so long as the correct methods are defined. Suppose, that we define a Java class for dogs with subclasses for German Shepherds, Scotch Terriers, and other breeds. But many animals can be trained to fetch, and we want to be able to let the world know that some other object has this ability. So we are led to define the `Fetch` interface that includes the following method:

```
public interface Fetch {
   public void getBall();
}
```

However, we might have a very intelligent dog that can speak, and we could also define the `CanSpeak` interface:

```
public interface CanSit {
   public void speak();
}
```

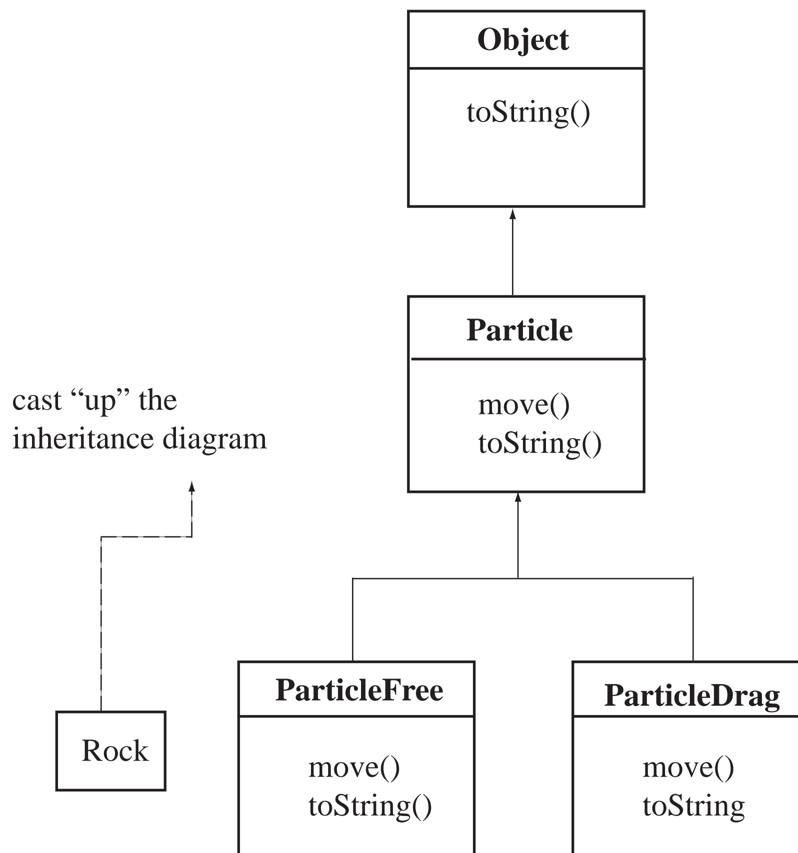

Figure 3.1: Inheritance diagram showing the relationships between `Particle`, `ParticleFree`, and `ParticleDrag`. The dotted line shows a particular instantiation of `ParticleFree`.

Any object, maybe even a `Horse`, can have one or both of these interfaces. All that is required is that the `Horse` know how to perform the required methods and that this ability be declared in the `Horse` definition using the `implements` keyword.

As a more relevant example, suppose that we wish to compute the numerical derivative of a function. Of course, we cannot compute numerical derivatives exactly on a digital computer because that would mean taking the limit of a difference Δx going to zero. So instead we will use what are called *finite differences*. If we look up derivatives in a calculus textbook, we would find the approximate relation

$$y' = f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}, \tag{3.9}$$

where y' is the approximate value of the derivative of $f(x)$ at x. A more accurate finite difference

relation for the first derivative is

$$y' = f'(x) \approx \frac{f(x+\Delta x) - f(x-\Delta x)}{2\,\Delta x}. \tag{3.10}$$

Equation (3.10) is more accurate because the errors due to the nonzero value of Δx cancel each other when the function is evaluated on a symmetrical interval about x.

Problem 3.3. Numerical derivatives

Do a Taylor series expansion of $f(x)$ about x to show that (3.10) is correct to second-order in Δx, while (3.9) is correct only to first-order in Δx.

We next use the relation (3.10) and write a program without using interfaces that computes the derivative of the sin function.

Listing 3.1: The derivative of the sine function.

```
package org.opensourcephysics.sip.ch3;

public class Derivative1App {

   public static void main(String[] args) {
      int numberOfPoints = 10;
      double xmin = 0;
      // compute derivative of sin(x) between x = 0 and x = 1
      double xmax = 0.5*Math.PI;
      double dx = (xmax - xmin)/numberOfPoints;
      double x = xmin;
      for (int i = 0; i < numberOfPoints; i++) {
         double derivative = (Math.sin(x+dx) - Math.sin(x-dx))/(2.0*dx);
         System.out.println("x = " + x + " derivative = " + derivative);
         x += dx;
      }
   }
}
```

Exercise 3.4. Numerical derivatives

a. Use class `Derivative1App` to compute the derivative of $\sin(x)$ at various values of x. Vary the value of Δx and compare your numerical results to the exact value of the derivative at the same values of x.

b. Modify the program so that it computes the numerical derivatives of the function $f(x) = (x^3 + \sin x)e^{-x^2}$ for x between 0 and 1.

c. Modify the program so that it computes the the derivative of two different functions.

As you probably found in Exercise 3.4b, it is tedious to change Lisitng 3.1 so that it computes the derivative of a different function. So in the following we will define a function interface. Defining an interface is similar to defining a class. An interface definition begins with the keyword `interface`, rather than the keyword `class`, and the body of the interface lists the methods that

must be present for the interface to be implemented. However, the bodies of the methods are undefined. As for a class, an interface must be saved in file with the same name as the interface.

We can think of a *function* as a rule, f, that gives a well-defined output, y, corresponding to a well-defined input, x.

$$y = f(x), \tag{3.11}$$

for some range of x. The variable x is called the independent variable and variable y is called the dependent variable. If we restrict the input and output of $f(x)$ to real numbers, it is easy to define an interface that implements this idea:

```
public interface Function {
   public double evaluate(double x);
}
```

That's all to it in this case. Note that we have defined the signature of the method `evaluate`, but have not written the body of the method that would implement the proposed functionality.

Next we add a method to the `MyMath` class that computes the numerical derivative of an object of type `Function` using the relation (3.10):

```
   public static double derivative(Function f, double x, double dx) {
      return (f.evaluate(x+dx) − f.evaluate(x−dx))/(2.0*dx);
   }
```

Then we define class `MyFunction` that implements the `Function` interface and defines the desired function $(x^3 - \sin x)e^{-x^2}$.

```
package org.opensourcephysics.sip.ch3;
import org.opensourcephysics.numerics.Function;

public class MyFunction implements Function {
   public double evaluate(double x) {
      double f = (Math.pow(x,3) + Math.sin(x))*Math.exp(−x*x);
      return f;
   }
}
```

Finally we write `Derivative2App` to take advantage of the interface `Function`, method `derivative`, and class `MyFunction`:

Listing 3.2: Testing the derivative method.

```
package org.opensourcephysics.sip.ch3;
import org.opensourcephysics.numerics.Function;
import org.opensourcephysics.sip.extras.MyMath;

public class Derivative2App {

   public static void main(String[] args) {
      int numberOfPoints = 10;
      double xmin = 0;
```

```
        double xmax = 0.5*Math.PI;
        double dx = (xmax - xmin)/numberOfPoints;
        double x = xmin;
        // note that an identifier can be defined by its interface
        Function function = new MyFunction();
        for (int i = 0; i < numberOfPoints; i++) {
            double derivative = MyMath.derivative(function,x,dx);
            System.out.println("x = " + x + " derivative = " + derivative);
            x += dx;
        }
    }
}
```

Exercise 3.5. Numerical derivatives

a. Use class `Derivative2App` to compute the derivative of different functions at various values of x. What do you have to do? Vary the value of Δx and compare your numerical results to the exact value of the derivative at the same values of x.

b. Define another class that implements `Function` and include this function in `Derivative2App`.

Exercise 3.6. Implementing the Function interface

Write a class that implements the `Function` interface and returns the value of the function $\sin x/x$ in the interval $-\pi \leq x \leq 1$.

We next look at an another example of how the Function interface is useful. Suppose we want to compute the area under a function between two limits, a and b (the integral of the function). At this point, we do not know how to perform this operation, but suppose somebody else does. This person can write a method that computes this area, without knowing the functions for which we will want to consider. For example, the static method `computeArea(Function f, double a, double b)` in the `MyMath` class takes a `Function` and the two limits as parameters and returns the area. Note that we can use this method without knowing how it is implemented.

```
public class IntegralApp {
    public static void main(String[] args) {
        Function f = new SinFunction();
        double a = 0;
        double b = 1;
        double area = MyMath.computeArea(f, a, b);
        System.out.println("area = " + area);
    }
}
```

3.3 Special Functions

Special functions are a mainstay of computational physics, and we can now define classes that encapsulate these functions. A toy example that defines a first-order linear equation $f(x) = mx+b$ can be written as follows:

```
public class LinearFunction implements Function {

    double m,b;

    public LinearFunction(double _m, double _b) {
        m = _m;
        b = _b;
    }

    public double evaluate(double x) {
        return m*x + b;
    }
}
```

A constructor is often used to initialize variables and we have done so here. Linear functions can now be created in the usual way by using the `new` operator.

```
LinearFunction f1 = new LinearFunction(2,3);
LinearFunction f2 = new LinearFunction(-1,2);
```

The linear function works (that is, we can evaluate `f1` or `f2` or both), but we can make it better from an object oriented point of view. The coefficients m and b are encapsulated, but not really hidden. Any class within the package in which the function is defined can modify these values by statements such as

```
f1.m = 4;
f1.b = -1;
```

We can prohibit this type of (unintended) mischief by making the `LinearFunction` class *immutable*. This change can be accomplished by making defining the class to be `final` so that it cannot be subclassed and by defining its member variables to be `private`:

```
public final class LinearFunction implements Function {

    private final double m,b;

    public LinearFunction(double _m, double _b) {
        m = _m;
        b = _b;
    }

    public double evaluate(double x) {
        return m*x + b;
    }
}
```

3.4 Effective Java

The main reason for using interfaces is that a class can implement many different ones. When should we use an interface or an abstract class? In general, we will start from an abstract class

only if we have a well defined organizational hierarchy. However, the flexibility of being able to implement multiple interfaces usually means that interfaces are preferable. Only if you are forced to have concrete method definitions or member variables should you use a class hierarchy.

Exercise 3.7. Implementing a moveable interface

a. Define a `Moveable` interface that contains the methods `move` and `getY`.

b. Change the `Particle` class so that it is not abstract and does not contain a `move` method. Change `ParticleFree` and `ParticleDrag` classes so that they implement the `Movable` interface. Test your new classes using the `ParticleMoverApp` program.

The major disadvantage to interfaces is that it is easier to change an abstract class than it is to change an interface. For example, if we add the non-abstract method `throw` to the `Particle` class, all existing subclasses will be able to perform this method. But if we add the `throw` method to the `Movable` interface, we must add this method to all classes that implement this interface. If other programmers use our Particle class, they will probably not notice if the new `throw` method exists. But if add `throw` to the `Movable` interface and don't tell anybody, any code that uses this interface will cease to work.

Inheritance is a powerful way to achieve code reuse, but it breaks encapsulation. A subclass all too often depends on the implementation details of its superclass if for no other reason than it invokes the superclass constructor. It is generally considered to be safe to use inheritance within a package where all classes are under the control of the same programmer. Carefully designed and documented class hierarchies, such as the user interface library distributed by Sun, can easily be subclassed.

3.5 Arrays

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 array variables are defined and created:

```
double[] x;                    // x defined to be an array of doubles
double x[];                    // same meaning
x = new double[32];            // x array created of 32 elements
double[] y = new double[32];   // y array defined and created in one step
int [] num = new int[100];     // array of 100 integers
double[][] sigma = new double[3][3]; // 3 x 3 array of doubles
Particle [] p = new Particle[2];     // array of 2 Particle objects
```

We will adopt the style `double[] x` instead of `double x[]`, although both forms are acceptable.

Just as defining a variable of type `Particle` does not create that object, creating an array of objects does not create the objects. We must create each object using the `new` operator. For the `Particle[]` array example, we still have to create the elements of the array `p`:

```
p[0] = new ParticleFree(20.0,0);
p[1] = new ParticleDrag(10.0,0,5.0);
```

Note that the array index starts at zero and is always one less than the number of elements.

Arrays are treated as objects insofar as they are created with the `new` operator and have a `length` property that can be accessed using dot notation. For example, we can create an array named `square` and assign each element to the square of its index using the following code:

```
int numberOfElements = 32;
double square = new double[numberOfElements]; // create array
for (int i = 0, n = square.length; i < n; i++)
   square[i] = i*i;
```

If computational speed is important, it is usually more efficient not to determine the size or length of an array dynamically using the `length` method.

Exercise 3.8. Freely falling body

Modify `ParticleFreeApp` so that you can consider any number of particles with different initial conditions. Treat the freely falling particles as an array.

3.6 Summary

As in Chapter 2, we have covered much more Java syntax and object oriented programming concepts. However, the good news is that the core syntax and concepts that we have discussed in Chapters 2 and 3 are sufficient to write programs that are roughly equivalent to what you might write using procedural languages like Fortran or C. Of course, there is more to learn about programming, but not much compared to what we have already covered.

In Chapter 4, we will introduce Java syntax associated with graphical input and output. Unless you are already familiar with Java, it will probably be difficult for you to understand all the syntax that we will introduce there in one reading. However, it doesn't really matter because we have written a package of classes that make graphical input and output much easier. We have already used the `Math` class written by programmers at Sun and didn't think much about it. One of the advantages of object oriented programming is that we only have to understand how a class works, not how it is written. So if you still feel a bit shaky about writing your own Java programs, relax. It will be downhill from now on.

References and Suggestions for Further Reading

Joshua Bloch, *Effective Java*, Addison Wesley (2001).

Stephen J. Chapman, *Java for Engineers and Scientists*, Prentice-Hall (2000).

Bruce Eckel, *Thinking in Java*, second edition, Prentice-Hall (2000).

David Flanagan, *Java in a Nutshell*, third edition, O'Reilly (2000).

Brian R. Overland, *Java in Plain English*, third edition, M&T Books (2001).

Walter Savitch, *Java: An Introduction to Computer Science and Programming*, second edition, Prentice-Hall (2001).