a Java applet runs inside a browser and does not require a `main` method; instead, it has methods such as `init` and `start`.) The `main` method is the application's starting point. The argument of the `main` method will always be the same, and understanding its syntax is not necessary here.

Because the code for this book contains hundreds of classes, we will adopt our own convention that classes that define main methods have names that end with `App`. We sometimes refer to an application that we are about to run as the *target* class.

Familiarize yourself with your Java development environment by doing Exercise 2.3.

Exercise 2.3 Our first application

(a) Enter the listing of `FirstFallingBallApp` into a source file named FirstFallingBallApp.java. (Java programs can be written using any text editor that supports standard ASCII characters.) Be sure to pay attention to capitalization because Java is case sensitive. In what directory should you place the source file?

(b) Compile and run `FirstFallingBallApp`. Do the results look reasonable to you? In what directory did the compiler place the byte code? ■

Digital computers represent numbers in base 2, that is, sequences of ones and zeros. Each one or zero is called a *bit*. For example, the number 13 is equivalent to 1101 or $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$. It would be difficult to write a program if we had to write numbers in base 2. Computer languages allow us to reference memory locations using identifiers or variable names. A valid variable name is a series of characters consisting of letters, digits, underscores, and dollar signs ($) that does not begin with a digit nor contain any spaces. Because Java distinguishes between upper and lowercase characters, `T` and `t` are different variable names. The Java convention is that variable names begin with a lowercase letter, except in special cases, and each succeeding word in a variable name begins with an uppercase letter.

In a purely object-oriented language, all variables would be objects that would be introduced by their class definitions. However, there are certain variable types that are so common that they have a special status and are especially easy to create and access. These types are called *primitive data types* and represent integer, floating point, boolean, and character variables. An example that illustrates that classes are effectively new programmer-defined types is given in Appendix 2A.

An integer variable, a floating point variable, and a boolean variable are created and *initialized* by the following statements:

```
int n = 10;
double y0 = 10.0;
boolean inert = true;
char c = 'A';     // used for single characters
```

There are four types of integers, `byte`, `short`, `int`, and `long`, and two types of floating point numbers; the differences are the range of numbers that these types can store. We will almost always use type `int` because it does not require as much memory as type `long`. There are two types of floating point numbers, but we will always use type `double`, the type with greater precision, to minimize roundoff error and to avoid having to provide multiple versions of various algorithms. A variable must be *declared* before it can be used, and it can be initialized at the same time that its type is declared as is done in Listing 2.1.

Integer arithmetic is exact, in contrast to floating point arithmetic which is limited by the maximum number of decimal places that can be stored. Important uses of integers are as counters in loops and as indices of arrays. An example of the latter is on page 39, where we discuss the motion of many balls.

A subtle and common error is to use integers in division when a floating point number is needed. For example, suppose we flip a coin 100 times and find 53 heads. What is the percentage of heads? In the following we show an unintended side effect of integer division and several ways of obtaining a floating point number from an integer.

```
int heads = 53;
int tosses = 100;
double percentage = heads/tosses;         // percentage will equal 0
percentage = (double)heads/tosses;        // percentage will equal 0.53
percentage = (1.0*heads)/tosses;          // percentage will equal 0.53
```

These statements indicate that if at least one number is a `double`, the result of the division will be a `double`. The expression `(double)heads` is called a *cast* and converts `heads` to a `double`. Because a number with a decimal point is treated as a double, we can also do this conversion by first multiplying heads by 1.0 as is done in the last statement.

Note that we have used the *assignment operator*, which is the equal (=) sign. This operator assigns the value to the memory location that is associated with a variable, such as `y0` and `t`. The following statements illustrate an important difference between the equal sign in mathematics and the assignment operator in most programming languages.

```
int x = 10;
x = x + 1;
```

The equal sign replaces a value in memory and is not a statement of equality. The left and right sides of an assignment operator are usually not equal.

A statement is analogous to a complete sentence, and an expression is similar to a phrase. The simplest expressions are identifiers or variables. More interesting expressions can be created by combining variables using *operators*, such as the following example of the plus (+) operator:

```
x + 3.0
```

Lines twelve through eighteen of Listing 2.1 declare and initialize variables. If a variable is declared but not initialized, for example,

```
double dt;
```

then the default value of the variable is 0 for numbers and `false` for boolean variables. It is a good idea to initialize all variables explicitly and not rely on their default values.

A very useful control structure is the `for` loop in line 20 of Listing 2.1. Loops are blocks of statements that are executed repeatedly until some condition is satisfied. They typically require the initialization of a counter variable, a test to determine if the counter variable has reached its terminal value, and a rule for changing the counter variable. These three parts of the `for` loop are contained within parentheses and are separated by semicolons. It is common in Java to iterate from 0 to 99, as is done in Listing 2.1, rather than from 1 to 100. Note the use of the ++ operator in the loop construct rather than the equivalent statement `n = n + 1`. It is important to indent all the statements within a block so that they can be easily identified. Java ignores these spaces, but they are important visual cues to the structure of the program.