Table 2.1 Common operators. The result for each row assumes that the statements from previous rows have been executed with `double x = 7, y = 3` declared initially. The *mod* or *modulus* operator `%` computes the remainder after the division by an integer has been performed.

Operator	Operand	Description	Example	Result
`++, -`	number	increment, decrement	`x++;`	8.0 stored in x
`+, -`	numbers	addition, subtraction	`3.5 + x`	11.5
`!`	boolean	logical complement	`!(x == y)`	true
`=`	any	assignment	`y = 3;`	`3.0` stored in y
`*, /, %`	numbers	multiplication, division, modulus	`7/2`	3.0
`==`	any	test for equality	`x == y`	false
`+=`	numbers	`x += 3;` equivalent to `x = x + 3;`	`x += 3;`	14.5 stored in x
`-=`	numbers	`x -= 2;` equivalent to `x = x - 2;`	`x -= 2.3;`	12.2 stored in x
`*=`	numbers	`x *= 4;` equivalent to `x = 4*x;`	`x *= 4;`	48.8 stored in x
`/=`	numbers	`x /= 2;` equivalent to `x = x/2;`	`x /= 2;`	24.4 stored in x
`%=`	numbers	`x %= 5;` equivalent to `x = x % 5;`	`x %= 5;`	4.4 stored in x

Listing 2.2 `FallingBall` class.

```
package org.opensourcephysics.sip.ch02;
public class FallingBall {
   double y, v, t;                  // instance variables
   double dt;                       // default package protection
   final static double g = 9.8;

   public FallingBall() { // constructor
      System.out.println("A new FallingBall object is created.");
   }

   public void step() {
      y = y+v*dt; // Euler algorithm for numerical solution
      v = v-g*dt;
      t = t+dt;
   }

   public double analyticPosition(double y0, double v0) {
      return y0+v0*t-0.5*g*t*t;
   }

   public double analyticVelocity(double v0) {
      return v0-g*t;
   }
}
```

As we will see, a class is a blueprint for creating objects, not an object itself. Except for the constant `g`, all the variable declarations in Listing 2.2 are *instance* variables. Each time an object is created or *instantiated* from the class, a separate block of memory is set aside for the instance variables. Thus, two objects created from the same class will, in general, have different values of the instance variables. We can insure that the value of a variable is the same for all objects created from the class by adding the word `static` to

the declaration. Such a variable is called a *class* variable and is appropriate for the constant `g`. In addition, you might not want the quantity referred to by an identifier to change. For example, `g` is a constant of nature. We can prevent a change by adding the keyword `final` to the declaration. Thus the statement

```
final static double g = 9.8;
```

means that a single copy of the constant `g` will be created and shared among all the objects instantiated from the class. Without the `final` qualifier, we could change the value of a class variable in every instantiated object by changing it in any one object. Static variables and methods are accessed from another class using the class name without first creating an instance (see page 25).

Another Java convention is that the names of constants should be in upper case. But in physics the meaning of g, the gravitational field, and G, the gravitational constant, have completely different meanings. So we will disregard this convention if doing so makes our programs more readable.

We have used certain words such as `double`, `false`, `main`, `static`, and `final`. These reserved words cannot be used as variable names and are examples of *keywords*.

In addition to the four instance variables `y`, `v`, `t`, and `dt`, and one class variable `g`, the `FallingBall` class has four methods. The first method is `FallingBall` and is a special method known as the *constructor*. A constructor must have the same name as the class and does not have an explicit return type. We will see that constructors allocate memory and initialize instance variables when an object is created.

The second method is `step`, a name that we will frequently use to advance a system's coordinates by one time step. The qualifier `void` means that this method does not return a value.

The next two methods, `analyticPosition` and `analyticVelocity`, each return a double value and have arguments enclosed by parentheses, the parameter list. The list of parameters and their types must be given explicitly and be separated by commas. The parameters can be primitive data types or class types. When the method is invoked, the argument types must match that given in the definition or be convertible into the type given in the definition, but need not have the same names. (Convertible means that the given variable can be unambiguously converted into another data type. For example, an integer can always be converted into a double.) For example, we can write

```
double y0 = 10;      // declaration and assignment
int v0 = 0;          // note v0 is an integer
// v0 becomes a double before method is called
double y = analyticPosition(y0,v0);
double v = analyticVelocity(v0);
```

but the following statements are incorrect:

```
// can't convert String to double automatically
double y = analyticPosition(y0,"0");
// method expects only one argument
double v = analyticVelocity(v0,0);
```

If a method does not receive any parameters, the parentheses are still required as in method `step()`.