Chapter 4

Objects and Classes

In this chapter, we

- Introduce you to object-oriented programming;
- Show you how you can create objects that belong to classes in the standard Java library; and
- Show you how to write your own classes.

If you do not have a background in object-oriented programming, you will want to read this chapter carefully. Thinking about object-oriented programming requires a different way of thinking than for procedural languages. The transition is not always easy, but you do need some familiarity with object concepts to go further with Java.

For experienced C++ programmers, this chapter, like the previous chapter, presents familiar information; however, there are enough differences between the two languages that you should read the later sections of this chapter carefully. You'll find the C++ notes helpful for making the transition.

Introduction to Object-Oriented Programming

Object-oriented programming (or OOP for short) is the dominant programming paradigm these days, having replaced the "structured," procedural programming techniques that were developed in the 1970s. Java is totally object oriented, and you have to be familiar with OOP to become productive with Java.

An object-oriented program is made of objects. Each object has a specific functionality that is exposed to its users, and a hidden implementation. Many objects in your programs will be taken "off-the-shelf" from a library; others are custom designed. Whether you build an object or buy it might depend on your budget or on time. But, basically, as long as objects satisfy your specifications, you don't care how the functionality was implemented. In OOP, you don't care how an object is implemented as long as it does what you want.

Traditional structured programming consists of designing a set of procedures (or *algorithms*) to solve a problem. After the procedures were determined, the traditional next step was to find appropriate ways to store the data. This is why the designer of the Pascal language, Niklaus Wirth, called his famous book on programming *Algorithms + Data Structures = Programs* (Prentice Hall, 1975). Notice that in Wirth's title, algorithms come first, and data structures come second. This mimics the way programmers worked at that time. First, they decided the procedures for manipulating the data; then, they decided what structure to impose on the data to make the manipulations easier. OOP reverses the order and puts data first, then looks at the algorithms that operate on the data.

For small problems, the breakdown into procedures works very well. But objects are more appropriate for larger problems. Consider a simple web browser. It might require 2,000 procedures for its implementation, all of which manipulate a set of global data. In the object-oriented style, there might be 100 classes with an average of 20 methods per class (see Figure 4–1). The latter structure is much easier for a programmer to grasp. It is also much easier to find bugs. Suppose the data of a particular object is in an incorrect state. It is far easier to search for the culprit among the 20 methods that had access to that data item than among 2,000 procedures.

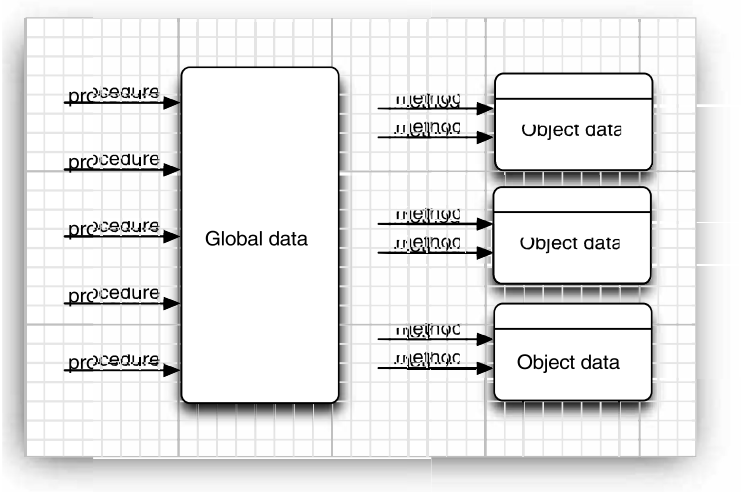

Figure 4–1 Procedural vs. OO programming

Classes

A *class* is the template or blueprint from which objects are made. Thinking about classes as cookie cutters. Objects are the cookies themselves. When you *construct* an object from a class, you are said to have created an *instance* of the class.

As you have seen, all code that you write in Java is inside a class. The standard Java library supplies several thousand classes for such diverse purposes as user interface design, dates and calendars, and network programming. Nonetheless, you still have to create your own classes in Java to describe the objects of the problem domains of your applications.

Encapsulation (sometimes called information hiding) is a key concept in working with objects. Formally, encapsulation is nothing more than combining data and behavior in one package and hiding the implementation details from the user of the object. The data in an object are called its *instance fields*, and the procedures that operate on the data are called its *methods*. A specific object that is an instance of a class will have specific values for its instance fields. The set of those values is the current *state* of the object. Whenever you invoke a method on an object, its state may change.

The key to making encapsulation work is to have methods *never* directly access instance fields in a class other than their own. Programs should interact with object data *only* through the object's methods. Encapsulation is the way to give the object its "black box" behavior, which is the key to reuse and reliability. This means a class may totally change how it stores its data, but as long as it continues to use the same methods to manipulate the data, no other object will know or care.

When you do start writing your own classes in Java, another tenet of OOP makes this easier: classes can be built by *extending* other classes. Java, in fact, comes with a "cosmic

superclass" called `Object`. All other classes extend this class. You will see more about the `Object` class in the next chapter.

When you extend an existing class, the new class has all the properties and methods of the class that you extend. You supply new methods and data fields that apply to your new class only. The concept of extending a class to obtain another class is called *inheritance*. See the next chapter for details on inheritance.

Objects

To work with OOP, you should be able to identify three key characteristics of objects:

- The object's *behavior*—What can you do with this object, or what methods can you apply to it?
- The object's *state*—How does the object react when you apply those methods?
- The object's *identity*—How is the object distinguished from others that may have the same behavior and state?

All objects that are instances of the same class share a family resemblance by supporting the same *behavior*. The behavior of an object is defined by the methods that you can call.

Next, each object stores information about what it currently looks like. This is the object's *state*. An object's state may change over time, but not spontaneously. A change in the state of an object must be a consequence of method calls. (If the object state changed without a method call on that object, someone broke encapsulation.)

However, the state of an object does not completely describe it, because each object has a distinct *identity*. For example, in an order-processing system, two orders are distinct even if they request identical items. Notice that the individual objects that are instances of a class *always* differ in their identity and *usually* differ in their state.

These key characteristics can influence each other. For example, the state of an object can influence its behavior. (If an order is "shipped" or "paid," it may reject a method call that asks it to add or remove items. Conversely, if an order is "empty," that is, no items have yet been ordered, it should not allow itself to be shipped.)

Identifying Classes

In a traditional procedural program, you start the process at the top, with the `main` function. When designing an object-oriented system, there is no "top," and newcomers to OOP often wonder where to begin. The answer is, you first find classes and then you add methods to each class.

A simple rule of thumb in identifying classes is to look for nouns in the problem analysis. Methods, on the other hand, correspond to verbs.

For example, in an order-processing system, some of these nouns are

- Item
- Order
- Shipping address
- Payment
- Account

These nouns may lead to the classes `Item`, `Order`, and so on.

Next, look for verbs. Items are *added* to orders. Orders are *shipped* or *canceled*. Payments are *applied* to orders. With each verb, such as "add," "ship," "cancel," and "apply," you identify the one object that has the major responsibility for carrying it out. For example, when a new item is added to an order, the order object should be the one in charge because it knows how it stores and sorts items. That is, `add` should be a method of the `Order` class that takes an `Item` object as a parameter.

Of course, the "noun and verb" rule is only a rule of thumb, and only experience can help you decide which nouns and verbs are the important ones when building your classes.

Relationships between Classes

The most common relationships between classes are

- *Dependence* ("uses–a")
- *Aggregation* ("has–a")
- *Inheritance* ("is–a")

The *dependence*, or "uses–a" relationship, is the most obvious and also the most general. For example, the `Order` class uses the `Account` class because `Order` objects need to access `Account` objects to check for credit status. But the `Item` class does not depend on the `Account` class, because `Item` objects never need to worry about customer accounts. Thus, a class depends on another class if its methods use or manipulate objects of that class.

Try to minimize the number of classes that depend on each other. The point is, if a class `A` is unaware of the existence of a class `B`, it is also unconcerned about any changes to `B`! (And this means that changes to `B` do not introduce bugs into `A`.) In software engineering terminology, you want to minimize the *coupling* between classes.

The *aggregation*, or "has–a" relationship, is easy to understand because it is concrete; for example, an `Order` object contains `Item` objects. Containment means that objects of class `A` contain objects of class `B`.

NOTE: Some methodologists view the concept of aggregation with disdain and prefer to use a more general "association" relationship. From the point of view of modeling, that is understandable. But for programmers, the "has–a" relationship makes a lot of sense. We like to use aggregation for a second reason—the standard notation for associations is less clear. See Table 4–1.

The *inheritance*, or "is–a" relationship, expresses a relationship between a more special and a more general class. For example, a `RushOrder` class inherits from an `Order` class. The specialized `RushOrder` class has special methods for priority handling and a different method for computing shipping charges, but its other methods, such as adding items and billing, are inherited from the `Order` class. In general, if class `A` extends class `B`, class `A` inherits methods from class `B` but has more capabilities. (We describe inheritance more fully in the next chapter, in which we discuss this important notion at some length.)

Many programmers use the UML (Unified Modeling Language) notation to draw *class diagrams* that describe the relationships between classes. You can see an example of such a diagram in Figure 4–2. You draw classes as rectangles, and relationships as arrows with various adornments. Table 4–1 shows the most common UML arrow styles.

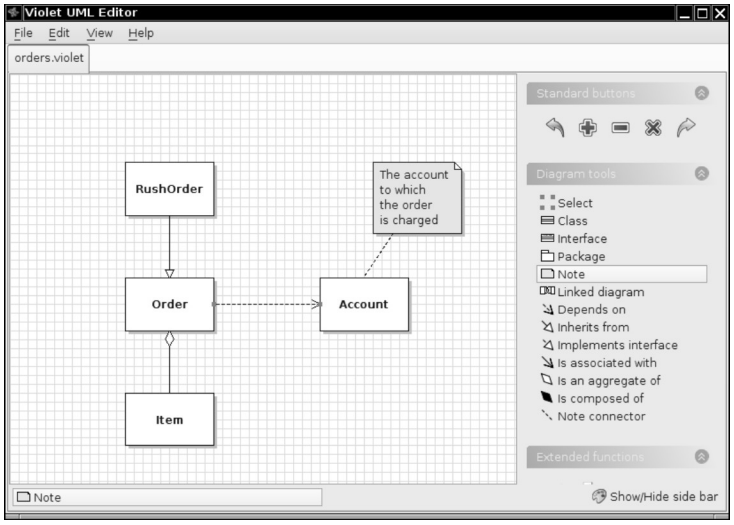

Figure 4–2 A class diagram

NOTE: A number of tools are available for drawing UML diagrams. Several vendors offer high-powered (and high-priced) tools that aim to be the focal point of your development process. Among them are Rational Rose (`http://www.ibm.com/software/awdtools/developer/rose`) and Together (`http://www.borland.com/us/products/together`). Another choice is the open source program ArgoUML (`http://argouml.tigris.org`). A commercially supported version is available from GentleWare (`http://gentleware.com`). If you just want to draw a simple diagrams with a minimum of fuss, try out Violet (`http://violet.sourceforge.net`).

Table 4–1 UML Notation for Class Relationships

Relationship	UML Connector
Inheritance	
Interface inheritance	
Dependency	
Aggregation	
Association	
Directed association	

Using Predefined Classes

Because you can't do anything in Java without classes, you have already seen several classes at work. However, not all of these show off the typical features of object orientation. Take, for example, the `Math` class. You have seen that you can use methods of the `Math` class, such as `Math.random`, without needing to know how they are implemented—all you need to know is the name and parameters (if any). That is the point of encapsulation and will certainly be true of all classes. But the `Math` class *only* encapsulates functionality; it neither needs nor hides data. Because there is no data, you do not need to worry about making objects and initializing their instance fields—there aren't any!

In the next section, we look at a more typical class, the `Date` class. You will see how to construct objects and call methods of this class.

Objects and Object Variables

To work with objects, you first construct them and specify their initial state. Then you apply methods to the objects.

In the Java programming language, you use *constructors* to construct new instances. A constructor is a special method whose purpose is to construct and initialize objects. Let us look at an example. The standard Java library contains a `Date` class. Its objects describe points in time, such as "December 31, 1999, 23:59:59 GMT".

NOTE: You may be wondering: Why use classes to represent dates rather than (as in some languages) a built-in type? For example, Visual Basic has a built-in date type and programmers can specify dates in the format `#6/1/1995#`. On the surface, this sounds convenient—programmers can simply use the built-in date type rather than worrying about classes. But actually, how suitable is the Visual Basic design? In some locales, dates are specified as month/day/year, in others as day/month/year. Are the language designers really equipped to foresee these kinds of issues? If they do a poor job, the language becomes an unpleasant muddle, but unhappy programmers are powerless to do anything about it. With classes, the design task is offloaded to a library designer. If the class is not perfect, other programmers can easily write their own classes to enhance or replace the system classes. (To prove the point: The Java date library is a bit muddled, and a major redesign is underway; see `http://jcp.org/en/jsr/detail?id=310`.)

Constructors always have the same name as the class name. Thus, the constructor for the `Date` class is called `Date`. To construct a `Date` object, you combine the constructor with the `new` operator, as follows:

```
new Date()
```

This expression constructs a new object. The object is initialized to the current date and time.

If you like, you can pass the object to a method:

```
System.out.println(new Date());
```

Alternatively, you can apply a method to the object that you just constructed. One of the methods of the `Date` class is the `toString` method. That method yields a string representation of the date. Here is how you would apply the `toString` method to a newly constructed `Date` object:

```
String s = new Date().toString();
```

In these two examples, the constructed object is used only once. Usually, you will want to hang on to the objects that you construct so that you can keep using them. Simply store the object in a variable:

```
Date birthday = new Date();
```

Figure 4–3 shows the object variable `birthday` that refers to the newly constructed object.

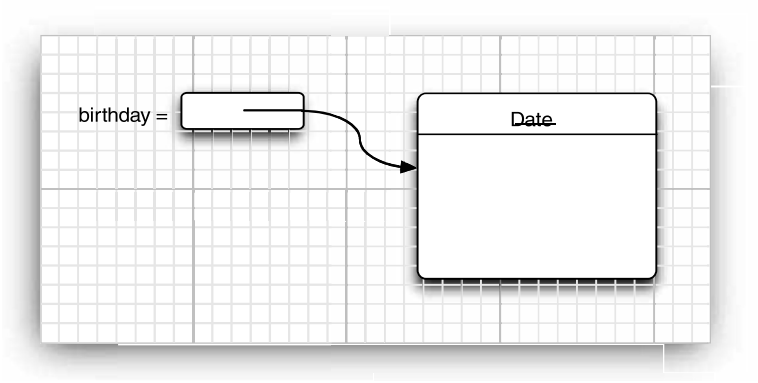

Figure 4–3 Creating a new object

There is an important difference between objects and object variables. For example, the statement

```
Date deadline; // deadline doesn't refer to any object
```

defines an object variable, `deadline`, that can refer to objects of type `Date`. It is important to realize that the variable `deadline` *is not an object* and, in fact, does not yet even refer to an object. You cannot use any `Date` methods on this variable at this time. The statement

```
s = deadline.toString(); // not yet
```

would cause a compile-time error.

You must first initialize the `deadline` variable. You have two choices. Of course, you can initialize the variable with a newly constructed object:

```
deadline = new Date();
```

Or you can set the variable to refer to an existing object:

```
deadline = birthday;
```

Now both variables refer to the *same* object (see Figure 4–4).

It is important to realize that an object variable doesn't actually contain an object. It only *refers* to an object.

In Java, the value of any object variable is a reference to an object that is stored elsewhere. The return value of the `new` operator is also a reference. A statement such as

```
Date deadline = new Date();
```

has two parts. The expression `new Date()` makes an object of type `Date`, and its value is a reference to that newly created object. That reference is then stored in the `deadline` variable.

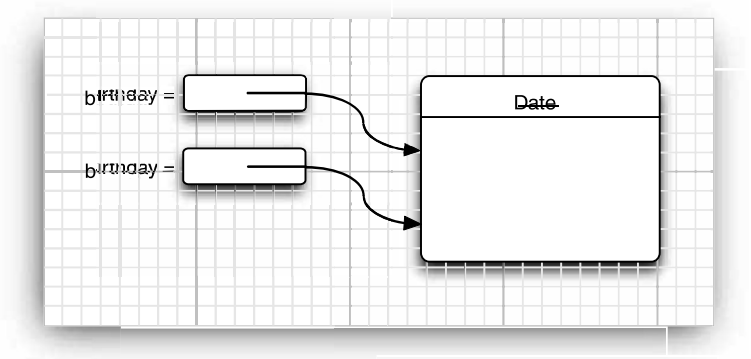

Figure 4–4 Object variables that refer to the same object

You can explicitly set an object variable to `null` to indicate that it currently refers to no object.

```
deadline = null;
. . .
if (deadline != null)
   System.out.println(deadline);
```

If you apply a method to a variable that holds `null`, then a runtime error occurs.

```
birthday = null;
String s = birthday.toString(); // runtime error!
```

Variables are not automatically initialized to `null`. You must initialize them, either by calling `new` or by setting them to `null`.

C++ NOTE: Many people mistakenly believe that Java object variables behave like C++ references. But in C++ there are no null references, and references cannot be assigned. You should think of Java object variables as analogous to *object pointers* in C++. For example,

```
Date birthday; // Java
```

is really the same as

```
Date* birthday; // C++
```

Once you make this association, everything falls into place. Of course, a `Date*` pointer isn't initialized until you initialize it with a call to `new`. The syntax is almost the same in C++ and Java.

```
Date* birthday = new Date(); // C++
```

If you copy one variable to another, then both variables refer to the same date—they are pointers to the same object. The equivalent of the Java `null` reference is the C++ `NULL` pointer.

All Java objects live on the heap. When an object contains another object variable, that variable still contains just a pointer to yet another heap object.

In C++, pointers make you nervous because they are so error prone. It is easy to create bad pointers or to mess up memory management. In Java, these problems simply go away. If you use an uninitialized pointer, the runtime system will reliably generate a runtime error instead of producing random results. You don't worry about memory management, because the garbage collector takes care of it.

C++ makes quite an effort, with its support for copy constructors and assignment operators, to allow the implementation of objects that copy themselves automatically. For example, a copy of a linked list is a new linked list with the same contents but with an independent set of links. This makes it possible to design classes with the same copy behavior as the built-in types. In Java, you must use the `clone` method to get a complete copy of an object.

The `GregorianCalendar` *Class of the Java Library*

In the preceding examples, we used the `Date` class that is a part of the standard Java library. An instance of the `Date` class has a state, namely *a particular point in time*.

Although you don't need to know this when you use the `Date` class, the time is represented by the number of milliseconds (positive or negative) from a fixed point, the so-called *epoch*, which is 00:00:00 UTC, January 1, 1970. UTC is the Coordinated Universal Time, the scientific time standard that is, for practical purposes, the same as the more familiar GMT or Greenwich Mean Time.

But as it turns out, the `Date` class is not very useful for manipulating dates. The designers of the Java library take the point of view that a date description such as "December 31, 1999, 23:59:59" is an arbitrary convention, governed by a *calendar*. This particular description follows the Gregorian calendar, which is the calendar used in most places of the world. The same point in time would be described quite differently in the Chinese or Hebrew lunar calendars, not to mention the calendar used by your customers from Mars.

NOTE: Throughout human history, civilizations grappled with the design of calendars that attached names to dates and brought order to the solar and lunar cycles. For a fascinating explanation of calendars around the world, from the French Revolutionary calendar to the Mayan long count, see *Calendrical Calculations, Second Edition* by Nachum Dershowitz and Edward M. Reingold (Cambridge University Press, 2001).

The library designers decided to separate the concerns of keeping time and attaching names to points in time. Therefore, the standard Java library contains two separate classes: the `Date` class, which represents a point in time, and the `GregorianCalendar` class, which expresses dates in the familiar calendar notation. In fact, the `GregorianCalendar` class extends a more generic `Calendar` class that describes the properties of calendars in general. In theory, you can extend the `Calendar` class and implement the Chinese lunar calendar or a Martian calendar. However, the standard library does not contain any calendar implementations besides the Gregorian calendar.

Separating time measurement from calendars is good object-oriented design. In general, it is a good idea to use separate classes to express different concepts.

The `Date` class has only a small number of methods that allow you to compare two points in time. For example, the `before` and `after` methods tell you if one point in time comes before or after another:

```
if (today.before(birthday))
   System.out.println("Still time to shop for a gift.");
```

NOTE: Actually, the `Date` class has methods such as `getDay`, `getMonth`, and `getYear`, but these methods are *deprecated*. A method is deprecated when a library designer realizes that the method should have never been introduced in the first place.

These methods were a part of the `Date` class before the library designers realized that it makes more sense to supply separate calendar classes. When the calendar classes were introduced, the `Date` methods were tagged as deprecated. You can still use them in your programs, but you will get unsightly compiler warnings if you do. It is a good idea to stay away from using deprecated methods because they may be removed in a future version of the library.

The `GregorianCalendar` class has many more methods than the `Date` class. In particular, it has several useful constructors. The expression

```
new GregorianCalendar()
```

constructs a new object that represents the date and time at which the object was constructed.

You can construct a calendar object for midnight on a specific date by supplying year, month, and day:

```
new GregorianCalendar(1999, 11, 31)
```

Somewhat curiously, the months are counted from 0. Therefore, 11 is December. For greater clarity, there are constants like `Calendar.DECEMBER`:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

You can also set the time:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

Of course, you will usually want to store the constructed object in an object variable:

```
GregorianCalendar deadline = new GregorianCalendar(. . .);
```

The `GregorianCalendar` has encapsulated instance fields to maintain the date to which it is set. Without looking at the source code, it is impossible to know the representation that the class uses internally. But, of course, the point of encapsulation is that this doesn't matter. What matters are the methods that a class exposes.

Mutator and Accessor Methods

At this point, you are probably asking yourself: How do I get at the current day or month or year for the date encapsulated in a specific `GregorianCalendar` object? And how do I change the values if I am unhappy with them? You can find out how to carry out these tasks by looking at the on-line documentation or the API notes at the end of this section. We go over the most important methods in this section.

The job of a calendar is to compute attributes, such as the date, weekday, month, or year, of a certain point in time. To query one of these settings, you use the `get` method of the `GregorianCalendar` class. To select the item that you want to get, you pass a constant defined in the `Calendar` class, such as `Calendar.MONTH` or `Calendar.DAY_OF_WEEK`:

```
GregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

The API notes list all the constants that you can use.

You change the state with a call to the `set` method:

```
deadline.set(Calendar.YEAR, 2001);
deadline.set(Calendar.MONTH, Calendar.APRIL);
deadline.set(Calendar.DAY_OF_MONTH, 15);
```

There is also a convenience method to set the year, month, and day with a single call:

```
deadline.set(2001, Calendar.APRIL, 15);
```

Finally, you can add a number of days, weeks, months, and so on, to a given calendar object:

```
deadline.add(Calendar.MONTH, 3); // move deadline by 3 months
```

If you add a negative number, then the calendar is moved backwards.

There is a conceptual difference between the `get` method on the one hand and the `set` and `add` methods on the other hand. The `get` method only looks up the state of the object and reports on it. The `set` and `add` methods modify the state of the object. Methods that change instance fields are called *mutator methods*, and those that only access instance fields without modifying them are called *accessor methods*.

C++ NOTE: In C++, the `const` suffix denotes accessor methods. A method that is not declared as `const` is assumed to be a mutator. However, in the Java programming language, no special syntax distinguishes between accessors and mutators.

A common convention is to prefix accessor methods with the prefix `get` and mutator methods with the prefix `set`. For example, the `GregorianCalendar` class has methods `getTime` and `setTime` that get and set the point in time that a calendar object represents:

```
Date time = calendar.getTime();
calendar.setTime(time);
```

These methods are particularly useful for converting between the `GregorianCalendar` and `Date` classes. Here is an example. Suppose you know the year, month, and day and you want to make a `Date` object with those settings. Because the `Date` class knows nothing about calendars, first construct a `GregorianCalendar` object and then call the `getTime` method to obtain a date:

```
GregorianCalendar calendar = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

Conversely, if you want to find the year, month, or day of a `Date` object, you construct a `GregorianCalendar` object, set the time, and then call the `get` method:

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

We finish this section with a program that puts the `GregorianCalendar` class to work. The program displays a calendar for the current month, like this:

```
Sun Mon Tue Wed Thu Fri Sat
                        1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19* 20  21  22
23  24  25  26  27  28  29
30  31
```

The current day is marked with an asterisk (*). As you can see, the program needs to know how to compute the length of a month and the weekday of a given day.

Let us go through the key steps of the program. First, we construct a calendar object that is initialized with the current date.

```
GregorianCalendar d = new GregorianCalendar();
```

We capture the current day and month by calling the `get` method twice.

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

Then we set `d` to the first of the month and get the weekday of that date.

```
d.set(Calendar.DAY_OF_MONTH, 1);
int weekday = d.get(Calendar.DAY_OF_WEEK);
```

The variable `weekday` is set to `Calendar.SUNDAY` if the first day of the month is a Sunday, to `Calendar.MONDAY` if it is a Monday, and so on. (These values are actually the integers 1, 2, . . . , 7, but it is best not to write code that depends on that knowledge.)

Note that the first line of the calendar is indented, so that the first day of the month falls on the appropriate weekday. This is a bit tricky since there are varying conventions about the starting day of the week. In the United States, the week starts with Sunday and ends with Saturday, whereas in Europe, the week starts with Monday and ends with Sunday.

The Java virtual machine is aware of the *locale* of the current user. The locale describes local formatting conventions, including the start of the week and the names of the weekdays.

TIP: If you want to see the program output in a different locale, add a line such as the following as the first line of the `main` method:

```
Locale.setDefault(Locale.ITALY);
```

The `getFirstDayOfWeek` method gets the starting weekday in the current locale. To determine the required indentation, we subtract 1 from the day of the calendar object until we reach the first day of the week.

```
int firstDayOfWeek = d.getFirstDayOfWeek();
int indent = 0;
while (weekday != firstDayOfWeek)
{
   indent++;
   d.add(Calendar.DAY_OF_MONTH, -1);
   weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

Next, we print the header with the weekday names. These are available from the class DateFormatSymbols.

```
String [] weekdayNames = new DateFormatSymbols().getShortWeekdays();
```

The getShortWeekdays method returns a string with short weekday names in the user's language (such as "Sun", "Mon", and so on in English). The array is indexed by weekday values. Here is the loop to print the header:

```
do
{
   System.out.printf("%4s", weekdayNames[weekday]);
   d.add(Calendar.DAY_OF_MONTH, 1);
   weekday = d.get(Calendar.DAY_OF_WEEK);
}
while (weekday != firstDayOfWeek);
System.out.println();
```

Now, we are ready to print the body of the calendar. We indent the first line and set the date object back to the start of the month. We enter a loop in which d traverses the days of the month.

In each iteration, we print the date value. If d is today, the date is marked with an *. If we reach the beginning of each new week, we print a new line. Then, we advance d to the next day:

```
d.add(Calendar.DAY_OF_MONTH, 1);
```

When do we stop? We don't know whether the month has 31, 30, 29, or 28 days. Instead, we keep iterating while d is still in the current month.

```
do
{
   . . .
}
while (d.get(Calendar.MONTH) == month);
```

Once d has moved into the next month, the program terminates.

Listing 4–1 shows the complete program.

As you can see, the GregorianCalendar class makes it possible to write a calendar program that takes care of complexities such as weekdays and the varying month lengths. You don't need to know *how* the GregorianCalendar class computes months and weekdays. You just use the *interface* of the class—the get, set, and add methods.

The point of this example program is to show you how you can use the interface of a class to carry out fairly sophisticated tasks without having to know the implementation details.

Listing 4–1 CalendarTest.java

```
1. import java.text.DateFormatSymbols;
2. import java.util.*;
3.
4. /**
5.  * @version 1.4 2007-04-07
6.  * @author Cay Horstmann
7.  */
```

Listing 4–1 CalendarTest.java (continued)

```
8.
9.  public class CalendarTest
10. {
11.    public static void main(String[] args)
12.    {
13.       // construct d as current date
14.       GregorianCalendar d = new GregorianCalendar();
15.
16.       int today = d.get(Calendar.DAY_OF_MONTH);
17.       int month = d.get(Calendar.MONTH);
18.
19.       // set d to start date of the month
20.       d.set(Calendar.DAY_OF_MONTH, 1);
21.
22.       int weekday = d.get(Calendar.DAY_OF_WEEK);
23.
24.       // get first day of week (Sunday in the U.S.)
25.       int firstDayOfWeek = d.getFirstDayOfWeek();
26.
27.       // determine the required indentation for the first line
28.       int indent = 0;
29.       while (weekday != firstDayOfWeek)
30.       {
31.          indent++;
32.          d.add(Calendar.DAY_OF_MONTH, -1);
33.          weekday = d.get(Calendar.DAY_OF_WEEK);
34.       }
35.
36.       // print weekday names
37.       String[] weekdayNames = new DateFormatSymbols().getShortWeekdays();
38.       do
39.       {
40.          System.out.printf("%4s", weekdayNames[weekday]);
41.          d.add(Calendar.DAY_OF_MONTH, 1);
42.          weekday = d.get(Calendar.DAY_OF_WEEK);
43.       }
44.       while (weekday != firstDayOfWeek);
45.       System.out.println();
46.
47.       for (int i = 1; i <= indent; i++)
48.          System.out.print("    ");
49.
50.       d.set(Calendar.DAY_OF_MONTH, 1);
51.       do
52.       {
53.          // print day
54.          int day = d.get(Calendar.DAY_OF_MONTH);
55.          System.out.printf("%3d", day);
```

Listing 4–1 CalendarTest.java (continued)

```
56.
57.         // mark current day with *
58.         if (day == today) System.out.print("*");
59.         else System.out.print(" ");
60.
61.         // advance d to the next day
62.         d.add(Calendar.DAY_OF_MONTH, 1);
63.         weekday = d.get(Calendar.DAY_OF_WEEK);
64.
65.         // start a new line at the start of the week
66.         if (weekday == firstDayOfWeek) System.out.println();
67.       }
68.       while (d.get(Calendar.MONTH) == month);
69.       // the loop exits when d is day 1 of the next month
70.
71.       // print final end of line if necessary
72.       if (weekday != firstDayOfWeek) System.out.println();
73.    }
74. }
```

API `java.util.GregorianCalendar 1.1`

- `GregorianCalendar()`
  constructs a calendar object that represents the current time in the default time zone with the default locale.
- `GregorianCalendar(int year, int month, int day)`
- `GregorianCalendar(int year, int month, int day, int hour, int minutes, int seconds)`
  constructs a Gregorian calendar with the given date and time.

  Parameters:

`year`	the year of the date
`month`	the month of the date. This value is 0-based; for example, 0 for January
`day`	the day of the month
`hour`	the hour (between 0 and 23)
`minutes`	the minutes (between 0 and 59)
`seconds`	the seconds (between 0 and 59)

- `int get(int field)`
  gets the value of a particular field.

  Parameters:

`field`	one of `Calendar.ERA`, `Calendar.YEAR`, `Calendar.MONTH`, `Calendar.WEEK_OF_YEAR`, `Calendar.WEEK_OF_MONTH`, `Calendar.DAY_OF_MONTH`, `Calendar.DAY_OF_YEAR`, `Calendar.DAY_OF_WEEK`, `Calendar.DAY_OF_WEEK_IN_MONTH`, `Calendar.AM_PM`, `Calendar.HOUR`, `Calendar.HOUR_OF_DAY`, `Calendar.MINUTE`, `Calendar.SECOND`, `Calendar.MILLISECOND`, `Calendar.ZONE_OFFSET`, `Calendar.DST_OFFSET`

- `void set(int field, int value)`
  sets the value of a particular field.

  Parameters:

`field`	one of the constants accepted by `get`
`value`	the new value

- `void set(int year, int month, int day)`
- `void set(int year, int month, int day, int hour, int minutes, int seconds)`
  sets the fields to new values.

  Parameters:

`year`	the year of the date
`month`	the month of the date. This value is 0-based; for example, 0 for January
`day`	the day of the month
`hour`	the hour (between 0 and 23)
`minutes`	the minutes (between 0 and 59)
`seconds`	the seconds (between 0 and 59)

- `void add(int field, int amount)`
  is a date arithmetic method. Adds the specified amount of time to the given time field. For example, to add 7 days to the current calendar date, call `c.add(Calendar.DAY_OF_MONTH, 7)`.

  Parameters:

`field`	the field to modify (using one of the constants documented in the `get` method)
`amount`	the amount by which the field should be changed (can be negative)

- `int getFirstDayOfWeek()`
  gets the first day of the week in the locale of the current user, for example, `Calendar.SUNDAY` in the United States.
- `void setTime(Date time)`
  sets this calendar to the given point in time.

  Parameters:

`time`	a point in time

- `Date getTime()`
  gets the point in time that is represented by the current value of this calendar object.

API `java.text.DateFormatSymbols 1.1`

- `String[] getShortWeekdays()`
- `String[] getShortMonths()`
- `String[] getWeekdays()`
- `String[] getMonths()`
  gets the names of the weekdays or months in the current locale. Uses `Calendar` weekday and month constants as array index values.

Defining Your Own Classes

In Chapter 3, you started writing simple classes. However, all those classes had just a single `main` method. Now the time has come to show you how to write the kind of "workhorse classes" that are needed for more sophisticated applications. These classes typically do not have a `main` method. Instead, they have their own instance fields and methods. To build a complete program, you combine several classes, one of which has a `main` method.

An `Employee` *Class*

The simplest form for a class definition in Java is

```
class ClassName
{
   constructor1
   constructor2
   . . .
   method1
   method2
   . . .
   field1
   field2
   . . .
}
```

NOTE: We adopt the style that the methods for the class come first and the fields come at the end. Perhaps this, in a small way, encourages the notion of looking at the interface first and paying less attention to the implementation.

Consider the following, very simplified, version of an `Employee` class that might be used by a business in writing a payroll system.

```
class Employee
{
   // constructor
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }

   // a method
   public String getName()
   {
      return name;
   }

   // more methods
   . . .
```

```
   // instance fields

   private String name;
   private double salary;
   private Date hireDay;
}
```

We break down the implementation of this class in some detail in the sections that follow. First, though, Listing 4–2 shows a program that shows the `Employee` class in action.

In the program, we construct an `Employee` array and fill it with three employee objects:

```
Employee[] staff = new Employee[3];

staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

Next, we use the `raiseSalary` method of the `Employee` class to raise each employee's salary by 5%:

```
for (Employee e : staff)
   e.raiseSalary(5);
```

Finally, we print out information about each employee, by calling the `getName`, `getSalary`, and `getHireDay` methods:

```
for (Employee e : staff)
   System.out.println("name=" + e.getName()
      + ",salary=" + e.getSalary()
      + ",hireDay=" + e.getHireDay());
```

Note that the example program consists of *two* classes: the `Employee` class and a class `EmployeeTest` with the `public` access specifier. The `main` method with the instructions that we just described is contained in the `EmployeeTest` class.

The name of the source file is `EmployeeTest.java` because the name of the file must match the name of the `public` class. You can have only one public class in a source file, but you can have any number of nonpublic classes.

Next, when you compile this source code, the compiler creates two class files in the directory: `EmployeeTest.class` and `Employee.class`.

You start the program by giving the bytecode interpreter the name of the class that contains the `main` method of your program:

```
java EmployeeTest
```

The bytecode interpreter starts running the code in the `main` method in the `EmployeeTest` class. This code in turn constructs three new `Employee` objects and shows you their state.

Listing 4–2 EmployeeTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program tests the Employee class.
5.  * @version 1.11 2004-02-19
6.  * @author Cay Horstmann
7.  */
8. public class EmployeeTest
9. {
10.    public static void main(String[] args)
11.    {
12.       // fill the staff array with three Employee objects
13.       Employee[] staff = new Employee[3];
14.
15.       staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
16.       staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17.       staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
18.
19.       // raise everyone's salary by 5%
20.       for (Employee e : staff)
21.          e.raiseSalary(5);
22.
23.       // print out information about all Employee objects
24.       for (Employee e : staff)
25.          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
26.                + e.getHireDay());
27.    }
28. }
29.
30. class Employee
31. {
32.    public Employee(String n, double s, int year, int month, int day)
33.    {
34.       name = n;
35.       salary = s;
36.       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.       // GregorianCalendar uses 0 for January
38.       hireDay = calendar.getTime();
39.    }
40.
41.    public String getName()
42.    {
43.       return name;
44.    }
45.
46.    public double getSalary()
47.    {
48.       return salary;
49.    }
```

Listing 4–2 EmployeeTest.java (continued)

```
50.
51.    public Date getHireDay()
52.    {
53.       return hireDay;
54.    }
55.
56.    public void raiseSalary(double byPercent)
57.    {
58.       double raise = salary * byPercent / 100;
59.       salary += raise;
60.    }
61.
62.    private String name;
63.    private double salary;
64.    private Date hireDay;
65. }
```

Use of Multiple Source Files

The program in Listing 4–2 has two classes in a single source file. Many programmers prefer to put each class into its own source file. For example, you can place the `Employee` class into a file `Employee.java` and the `EmployeeTest` class into `EmployeeTest.java`.

If you like this arrangement, then you have two choices for compiling the program. You can invoke the Java compiler with a wildcard:

```
javac Employee*.java
```

Then, all source files matching the wildcard will be compiled into class files. Or, you can simply type

```
javac EmployeeTest.java
```

You may find it surprising that the second choice works even though the `Employee.java` file is never explicitly compiled. However, when the Java compiler sees the `Employee` class being used inside `EmployeeTest.java`, it will look for a file named `Employee.class`. If it does not find that file, it automatically searches for `Employee.java` and then compiles it. Even more is true: if the time stamp of the version of `Employee.java` that it finds is newer than that of the existing `Employee.class` file, the Java compiler will *automatically* recompile the file.

NOTE: If you are familiar with the "make" facility of UNIX (or one of its Windows cousins such as "nmake"), then you can think of the Java compiler as having the "make" functionality already built in.

Dissecting the `Employee` Class

In the sections that follow, we want to dissect the `Employee` class. Let's start with the methods in this class. As you can see by examining the source code, this class has one constructor and four methods:

```
public Employee(String n, double s, int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)
```

All methods of this class are tagged as `public`. The keyword `public` means that any method in any class can call the method. (The four possible access levels are covered in this and the next chapter.)

Next, notice that three instance fields will hold the data we will manipulate inside an instance of the `Employee` class.

```
private String name;
private double salary;
private Date hireDay;
```

The `private` keyword makes sure that the *only* methods that can access these instance fields are the methods of the `Employee` class itself. No outside method can read or write to these fields.

NOTE: You could use the `public` keyword with your instance fields, but it would be a very bad idea. Having `public` data fields would allow any part of the program to read and modify the instance fields. That completely ruins encapsulation. Any method of any class can modify public fields—and, in our experience, some code usually will take advantage of that access privilege when you least expect it. We strongly recommend that you always make your instance fields `private`.

Finally, notice that two of the instance fields are themselves objects: the `name` and `hireDay` fields are references to `String` and `Date` objects. This is quite usual: classes will often contain instance fields of class type.

First Steps with Constructors

Let's look at the constructor listed in our `Employee` class.

```
public Employee(String n, double s, int year, int month, int day)
{
   name = n;
   salary = s;
   GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
   hireDay = calendar.getTime();
}
```

As you can see, the name of the constructor is the same as the name of the class. This constructor runs when you construct objects of the `Employee` class—giving the instance fields the initial state you want them to have.

For example, when you create an instance of the `Employee` class with code like this:

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

you have set the instance fields as follows:

```
name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;
```

There is an important difference between constructors and other methods. A constructor can only be called in conjunction with the `new` operator. You can't apply a constructor to an existing object to reset the instance fields. For example,

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ERROR
```

is a compile-time error.

We have more to say about constructors later in this chapter. For now, keep the following in mind:

- A constructor has the same name as the class.
- A class can have more than one constructor.
- A constructor can take zero, one, or more parameters.
- A constructor has no return value.
- A constructor is always called with the `new` operator.

C++ NOTE: Constructors work the same way in Java as they do in C++. But keep in mind that all Java objects are constructed on the heap and that a constructor must be combined with `new`. It is a common C++ programmer error to forget the `new` operator:

```
Employee number007("James Bond", 100000, 1950, 1, 1);
   // C++, not Java
```

That works in C++ but does not work in Java.

CAUTION: Be careful not to introduce local variables with the same names as the instance fields. For example, the following constructor will not set the salary:

```
public Employee(String n, double s, . . .)
{
   String name = n; // ERROR
   double salary = s; // ERROR
   . . .
}
```

The constructor declares *local* variables `name` and `salary`. These variables are only accessible inside the constructor. They *shadow* the instance fields with the same name. Some programmers—such as the authors of this book—write this kind of code when they type faster than they think, because their fingers are used to adding the data type. This is a nasty error that can be hard to track down. You just have to be careful in all of your methods that you don't use variable names that equal the names of instance fields.

Implicit and Explicit Parameters

Methods operate on objects and access their instance fields. For example, the method

```
public void raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
}
```

sets a new value for the `salary` instance field in the object on which this method is invoked. Consider the call

```
number007.raiseSalary(5);
```

The effect is to increase the value of the `number007.salary` field by 5%. More specifically, the call executes the following instructions:

```
double raise = number007.salary * 5 / 100;
number007.salary += raise;
```

The `raiseSalary` method has two parameters. The first parameter, called the *implicit* parameter, is the object of type `Employee` that appears before the method name. The second parameter, the number inside the parentheses after the method name, is an *explicit* parameter.

As you can see, the explicit parameters are explicitly listed in the method declaration, for example, `double byPercent`. The implicit parameter does not appear in the method declaration.

In every method, the keyword `this` refers to the implicit parameter. If you like, you can write the `raiseSalary` method as follows:

```
public void raiseSalary(double byPercent)
{
   double raise = this.salary * byPercent / 100;
   this.salary += raise;
}
```

Some programmers prefer that style because it clearly distinguishes between instance fields and local variables.

C++ NOTE: In C++, you generally define methods outside the class:

```
void Employee::raiseSalary(double byPercent) // C++, not Java
{
   . . .
}
```

If you define a method inside a class, then it is automatically an inline method.

```
class Employee
{
   . . .
   int getName() { return name; } // inline in C++
}
```

In the Java programming language, all methods are defined inside the class itself. This does not make them inline. Finding opportunities for inline replacement is the job of the Java virtual machine. The just-in-time compiler watches for calls to methods that are short, commonly called, and not overridden, and optimizes them away.

Benefits of Encapsulation

Finally, let's look more closely at the rather simple `getName`, `getSalary`, and `getHireDay` methods.

```
public String getName()
{
   return name;
}

public double getSalary()
{
   return salary;
}

public Date getHireDay()
{
   return hireDay;
}
```

These are obvious examples of accessor methods. Because they simply return the values of instance fields, they are sometimes called *field accessors*.

Wouldn't it be easier to simply make the `name`, `salary`, and `hireDay` fields public, instead of having separate accessor methods?

The point is that the `name` field is a read-only field. Once you set it in the constructor, there is no method to change it. Thus, we have a guarantee that the `name` field will never be corrupted.

The `salary` field is not read-only, but it can only be changed by the `raiseSalary` method. In particular, should the value ever be wrong, only that method needs to be debugged. Had the `salary` field been public, the culprit for messing up the value could have been anywhere.

Sometimes, it happens that you want to get and set the value of an instance field. Then you need to supply *three* items:

- A private data field;
- A public field accessor method; and
- A public field mutator method.

This is a lot more tedious than supplying a single public data field, but there are considerable benefits.

First, you can change the internal implementation without affecting any code other than the methods of the class.

For example, if the storage of the name is changed to

```
String firstName;
String lastName;
```

then the `getName` method can be changed to return

```
firstName + " " + lastName
```

This change is completely invisible to the remainder of the program.

Of course, the accessor and mutator methods may need to do a lot of work and convert between the old and the new data representation. But that leads us to our second benefit: Mutator methods can perform error-checking, whereas code that simply assigns to a field may not go to the trouble. For example, a `setSalary` method might check that the salary is never less than 0.

CAUTION: Be careful not to write accessor methods that return references to mutable objects. We violated that rule in our `Employee` class in which the `getHireDay` method returns an object of class `Date`:

```
class Employee
{
   . . .
   public Date getHireDay()
   {
      return hireDay;
   }
   . . .
   private Date hireDay;
}
```

This breaks the encapsulation! Consider the following rogue code:

```
Employee harry = . . .;
Date d = harry.getHireDay();
double tenYearsInMilliSeconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliSeconds);
// let's give Harry ten years added seniority
```

The reason is subtle. Both `d` and `harry.hireDay` refer to the same object (see Figure 4–5). Applying mutator methods to `d` automatically changes the private state of the employee object!

If you need to return a reference to a mutable object, you should *clone* it first. A clone is an exact copy of an object that is stored in a new location. We discuss cloning in detail in Chapter 6. Here is the corrected code:

```
class Employee
{
   . . .
   public Date getHireDay()
   {
      return (Date) hireDay.clone();
   }
   . . .
}
```

As a rule of thumb, always use `clone` whenever you need to return a copy of a mutable data field.

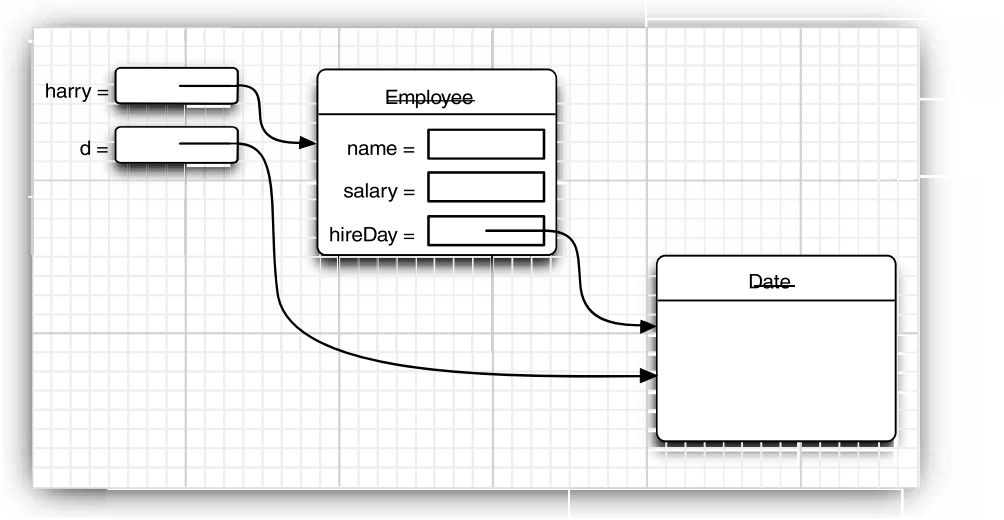

Figure 4–5 Returning a reference to a mutable data field

Class-Based Access Privileges

You know that a method can access the private data of the object on which it is invoked. What many people find surprising is that a method can access the private data of *all objects of its class*. For example, consider a method `equals` that compares two employees.

```
class Employee
{
   . . .
   boolean equals(Employee other)
   {
      return name.equals(other.name);
   }
}
```

A typical call is

```
if (harry.equals(boss)) . . .
```

This method accesses the private fields of `harry`, which is not surprising. It also accesses the private fields of `boss`. This is legal because `boss` is an object of type `Employee`, and a method of the `Employee` class is permitted to access the private fields of *any* object of type `Employee`.

C++ NOTE: C++ has the same rule. A method can access the private features of any object of its class, not just of the implicit parameter.

Private Methods

When implementing a class, we make all data fields private because public data are dangerous. But what about the methods? While most methods are public, private methods are used in certain circumstances. Sometimes, you may wish to break up the code for a computation into separate helper methods. Typically, these helper methods should not become part of the public interface—they may be too close to the current implementation or require a special protocol or calling order. Such methods are best implemented as `private`.

To implement a private method in Java, simply change the `public` keyword to `private`.

By making a method private, you are under no obligation to keep it available if you change to another implementation. The method may well be *harder* to implement or *unnecessary* if the data representation changes: this is irrelevant. The point is that as long as the method is private, the designers of the class can be assured that it is never used outside the other class operations and can simply drop it. If a method is public, you cannot simply drop it because other code might rely on it.

Final Instance Fields

You can define an instance field as `final`. Such a field must be initialized when the object is constructed. That is, it must be guaranteed that the field value has been set after the end of every constructor. Afterwards, the field may not be modified again. For example, the `name` field of the `Employee` class may be declared as `final` because it never changes after the object is constructed—there is no `setName` method.

```
class Employee
{
   . . .
   private final String name;
}
```

The `final` modifier is particularly useful for fields whose type is primitive or an *immutable class*. (A class is immutable if none of its methods ever mutate its objects. For example, the `String` class is immutable.) For mutable classes, the `final` modifier is likely to confuse the reader. For example,

```
private final Date hiredate;
```

merely means that the object reference stored in the `hiredate` variable doesn't get changed after the object is constructed. That does not mean that the `hiredate` object is constant. Any method is free to invoke the `setTime` mutator on the object to which `hiredate` refers.

Static Fields and Methods

In all sample programs that you have seen, the `main` method is tagged with the `static` modifier. We are now ready to discuss the meaning of this modifier.

Static Fields

If you define a field as `static`, then there is only one such field per class. In contrast, each object has its own copy of all instance fields. For example, let's suppose we want to assign a unique identification number to each employee. We add an instance field `id` and a static field `nextId` to the `Employee` class:

```
class Employee
{
```

```
   . . .
   private int id;
   private static int nextId = 1;
}
```

Every employee object now has its own `id` field, but there is only one `nextId` field that is shared among all instances of the class. Let's put it another way. If there are 1,000 objects of the `Employee` class, then there are 1,000 instance fields `id`, one for each object. But there is a single static field `nextId`. Even if there are no employee objects, the static field `nextId` is present. It belongs to the class, not to any individual object.

NOTE: In most object-oriented programming languages, static fields are called *class fields*. The term "static" is a meaningless holdover from C++.

Let's implement a simple method:

```
public void setId()
{
   id = nextId;
   nextId++;
}
```

Suppose you set the employee identification number for `harry`:

```
harry.setId();
```

Then, the `id` field of `harry` is set to the current value of the static field `nextId`, and the value of the static field is incremented:

```
harry.id = Employee.nextId;
Employee.nextId++;
```

Static Constants

Static variables are quite rare. However, static constants are more common. For example, the `Math` class defines a static constant:

```
public class Math
{
   . . .
   public static final double PI = 3.14159265358979323846;
   . . .
}
```

You can access this constant in your programs as `Math.PI`.

If the keyword `static` had been omitted, then `PI` would have been an instance field of the `Math` class. That is, you would need an object of the `Math` class to access `PI`, and every `Math` object would have its own copy of `PI`.

Another static constant that you have used many times is `System.out`. It is declared in the `System` class as follows:

```
public class System
{
   . . .
   public static final PrintStream out = . . .;
   . . .
}
```

As we mentioned several times, it is never a good idea to have public fields, because everyone can modify them. However, public constants (that is, `final` fields) are fine. Because `out` has been declared as `final`, you cannot reassign another print stream to it:

```
System.out = new PrintStream(. . .); // ERROR--out is final
```

NOTE: If you look at the `System` class, you will notice a method `setOut` that lets you set `System.out` to a different stream. You may wonder how that method can change the value of a `final` variable. However, the `setOut` method is a *native* method, not implemented in the Java programming language. Native methods can bypass the access control mechanisms of the Java language. This is a very unusual workaround that you should not emulate in your own programs.

Static Methods

Static methods are methods that do not operate on objects. For example, the `pow` method of the `Math` class is a static method. The expression

```
Math.pow(x, a)
```

computes the power x^a. It does not use any `Math` object to carry out its task. In other words, it has no implicit parameter.

You can think of static methods as methods that don't have a `this` parameter. (In a non-static method, the `this` parameter refers to the implicit parameter of the method—see the section "Implicit and Explicit Parameters" on page 127.)

Because static methods don't operate on objects, you cannot access instance fields from a static method. But static methods can access the static fields in their class. Here is an example of such a static method:

```
public static int getNextId()
{
   return nextId; // returns static field
}
```

To call this method, you supply the name of the class:

```
int n = Employee.getNextId();
```

Could you have omitted the keyword `static` for this method? Yes, but then you would need to have an object reference of type `Employee` to invoke the method.

NOTE: It is legal to use an object to call a static method. For example, if `harry` is an `Employee` object, then you can call `harry.getNextId()` instead of `Employee.getnextId()`. However, we find that notation confusing. The `getNextId` method doesn't look at `harry` at all to compute the result. We recommend that you use class names, not objects, to invoke static methods.

You use static methods in two situations:

- When a method doesn't need to access the object state because all needed parameters are supplied as explicit parameters (example: `Math.pow`)
- When a method only needs to access static fields of the class (example: `Employee.getNextId`)

C++ NOTE: Static fields and methods have the same functionality in Java and C++. However, the syntax is slightly different. In C++, you use the :: operator to access a static field or method outside its scope, such as `Math::PI`.

The term "static" has a curious history. At first, the keyword `static` was introduced in C to denote local variables that don't go away when a block is exited. In that context, the term "static" makes sense: the variable stays around and is still there when the block is entered again. Then `static` got a second meaning in C, to denote global variables and functions that cannot be accessed from other files. The keyword `static` was simply reused, to avoid introducing a new keyword. Finally, C++ reused the keyword for a third, unrelated, interpretation—to denote variables and functions that belong to a class but not to any particular object of the class. That is the same meaning that the keyword has in Java.

Factory Methods

Here is another common use for static methods. The `NumberFormat` class uses *factory methods* that yield formatter objects for various styles.

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x)); // prints 10%
```

Why doesn't the `NumberFormat` class use a constructor instead? There are two reasons:

- You can't give names to constructors. The constructor name is always the same as the class name. But we want two different names to get the currency instance and the percent instance.
- When you use a constructor, you can't vary the type of the constructed object. But the factory methods actually return objects of the class `DecimalFormat`, a subclass that inherits from `NumberFormat`. (See Chapter 5 for more on inheritance.)

The `main` Method

Note that you can call static methods without having any objects. For example, you never construct any objects of the `Math` class to call `Math.pow`.

For the same reason, the `main` method is a static method.

```
public class Application
{
   public static void main(String[] args)
   {
      // construct objects here
      . . .
   }
}
```

The `main` method does not operate on any objects. In fact, when a program starts, there aren't any objects yet. The static `main` method executes, and constructs the objects that the program needs.

TIP: Every class can have a `main` method. That is a handy trick for unit testing of classes. For example, you can add a `main` method to the `Employee` class:

```
class Employee
{
   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
      hireDay = calendar.getTime();
   }
   . . .
   public static void main(String[] args) // unit test
   {
      Employee e = new Employee("Romeo", 50000, 2003, 3, 31);
      e.raiseSalary(10);
      System.out.println(e.getName() + " " + e.getSalary());
   }
   . . .
}
```

If you want to test the `Employee` class in isolation, you simply execute

```
java Employee
```

If the employee class is a part of a larger application, then you start the application with

```
java Application
```

and the `main` method of the `Employee` class is never executed.

The program in Listing 4–3 contains a simple version of the `Employee` class with a static field `nextId` and a static method `getNextId`. We fill an array with three `Employee` objects and then print the employee information. Finally, we print the next available identification number, to demonstrate the static method.

Note that the `Employee` class also has a static `main` method for unit testing. Try running both

```
java Employee
```

and

```
java StaticTest
```

to execute both `main` methods.

Listing 4–3 StaticTest.java

```
1. /**
2.  * This program demonstrates static methods.
3.  * @version 1.01 2004-02-19
4.  * @author Cay Horstmann
5.  */
6. public class StaticTest
7. {
8.    public static void main(String[] args)
9.    {
```

Listing 4–3 StaticTest.java (continued)

```
10.       // fill the staff array with three Employee objects
11.       Employee[] staff = new Employee[3];
12.
13.       staff[0] = new Employee("Tom", 40000);
14.       staff[1] = new Employee("Dick", 60000);
15.       staff[2] = new Employee("Harry", 65000);
16.
17.       // print out information about all Employee objects
18.       for (Employee e : staff)
19.       {
20.          e.setId();
21.          System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
22.                + e.getSalary());
23.       }
24.
25.       int n = Employee.getNextId(); // calls static method
26.       System.out.println("Next available id=" + n);
27.    }
28. }
29.
30. class Employee
31. {
32.    public Employee(String n, double s)
33.    {
34.       name = n;
35.       salary = s;
36.       id = 0;
37.    }
38.
39.    public String getName()
40.    {
41.       return name;
42.    }
43.
44.    public double getSalary()
45.    {
46.       return salary;
47.    }
48.
49.    public int getId()
50.    {
51.       return id;
52.    }
53.
54.    public void setId()
55.    {
56.       id = nextId; // set id to next available id
57.       nextId++;
58.    }
```

Listing 4–3 StaticTest.java (continued)

```
59.
60.    public static int getNextId()
61.    {
62.       return nextId; // returns static field
63.    }
64.
65.    public static void main(String[] args) // unit test
66.    {
67.       Employee e = new Employee("Harry", 50000);
68.       System.out.println(e.getName() + " " + e.getSalary());
69.    }
70.
71.    private String name;
72.    private double salary;
73.    private int id;
74.    private static int nextId = 1;
75. }
```

Method Parameters

Let us review the computer science terms that describe how parameters can be passed to a method (or a function) in a programming language. The term *call by value* means that the method gets just the value that the caller provides. In contrast, *call by reference* means that the method gets the *location* of the variable that the caller provides. Thus, a method can *modify* the value stored in a variable that is passed by reference but not in one that is passed by value. These "call by . . . " terms are standard computer science terminology that describe the behavior of method parameters in various programming languages, not just Java. (In fact, there is also a *call by name* that is mainly of historical interest, being employed in the Algol programming language, one of the oldest high-level languages.)

The Java programming language *always* uses call by value. That means that the method gets a copy of all parameter values. In particular, the method cannot modify the contents of any parameter variables that are passed to it.

For example, consider the following call:

```
double percent = 10;
harry.raiseSalary(percent);
```

No matter how the method is implemented, we know that after the method call, the value of `percent` is still 10.

Let us look a little more closely at this situation. Suppose a method tried to triple the value of a method parameter:

```
public static void tripleValue(double x) // doesn't work
{
   x = 3 * x;
}
```

Let's call this method:

```
double percent = 10;
tripleValue(percent);
```

However, this does not work. After the method call, the value of `percent` is still 10. Here is what happens:

1. `x` is initialized with a copy of the value of `percent` (that is, 10).
2. `x` is tripled—it is now 30. But `percent` is still 10 (see Figure 4–6).
3. The method ends, and the parameter variable `x` is no longer in use.

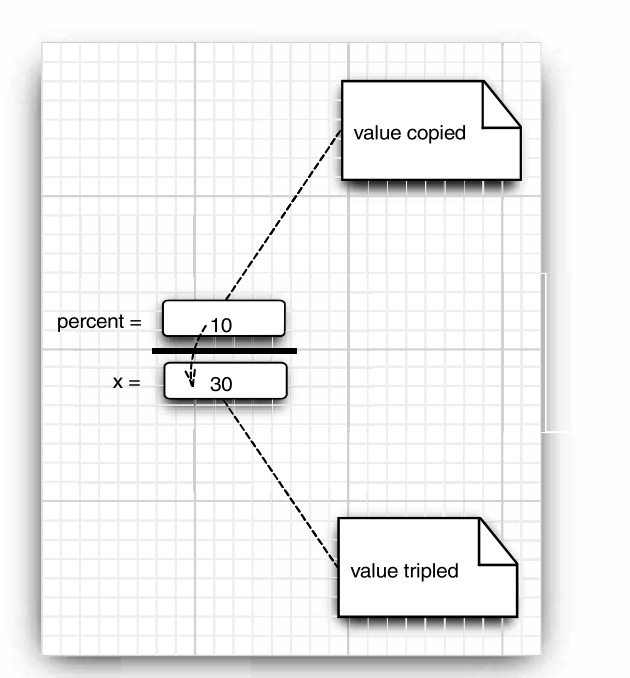

Figure 4–6 Modifying a numeric parameter has no lasting effect

There are, however, two kinds of method parameters:

- Primitive types (numbers, `boolean` values)
- Object references

You have seen that it is impossible for a method to change a primitive type parameter. The situation is different for object parameters. You can easily implement a method that triples the salary of an employee:

```
public static void tripleSalary(Employee x) // works
{
   x.raiseSalary(200);
}
```

When you call

```
harry = new Employee(. . .);
tripleSalary(harry);
```

then the following happens:

1. x is initialized with a copy of the value of harry, that is, an object reference.
2. The raiseSalary method is applied to that object reference. The Employee object to which both x and harry refer gets its salary raised by 200 percent.
3. The method ends, and the parameter variable x is no longer in use. Of course, the object variable harry continues to refer to the object whose salary was tripled (see Figure 4–7).

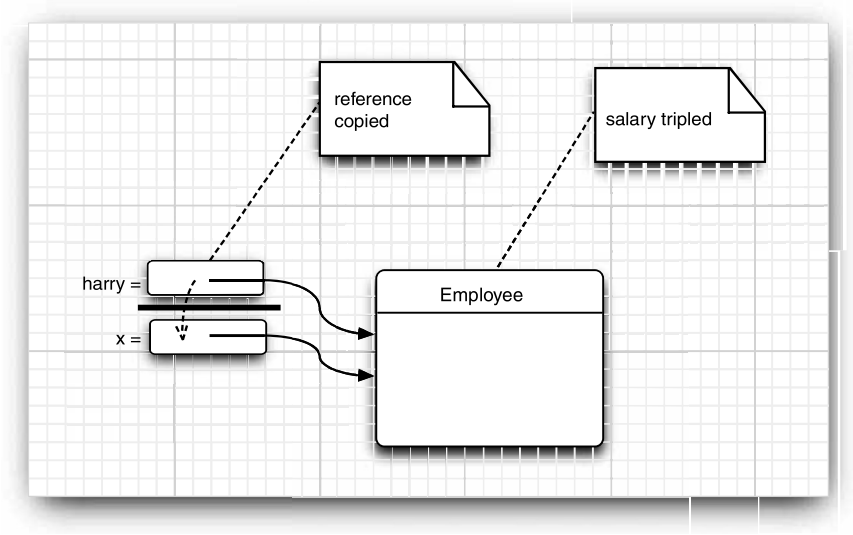

Figure 4–7 Modifying an object parameter has a lasting effect

As you have seen, it is easily possible—and in fact very common—to implement methods that change the state of an object parameter. The reason is simple. The method gets a copy of the object reference, and both the original and the copy refer to the same object.

Many programming languages (in particular, C++ and Pascal) have two methods for parameter passing: call by value and call by reference. Some programmers (and unfortunately even some book authors) claim that the Java programming language uses call by reference for objects. However, that is false. Because this is such a common misunderstanding, it is worth examining a counterexample in detail.

Let's try to write a method that swaps two employee objects:

```
public static void swap(Employee x, Employee y) // doesn't work
{
   Employee temp = x;
   x = y;
```

```
   y = temp;
}
```

If the Java programming language used call by reference for objects, this method would work:

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// does a now refer to Bob, b to Alice?
```

However, the method does not actually change the object references that are stored in the variables `a` and `b`. The `x` and `y` parameters of the `swap` method are initialized with *copies* of these references. The method then proceeds to swap these copies.

```
// x refers to Alice, y to Bob
Employee temp = x;
x = y;
y = temp;
// now x refers to Bob, y to Alice
```

But ultimately, this is a wasted effort. When the method ends, the parameter variables `x` and `y` are abandoned. The original variables `a` and `b` still refer to the same objects as they did before the method call (see Figure 4–8).

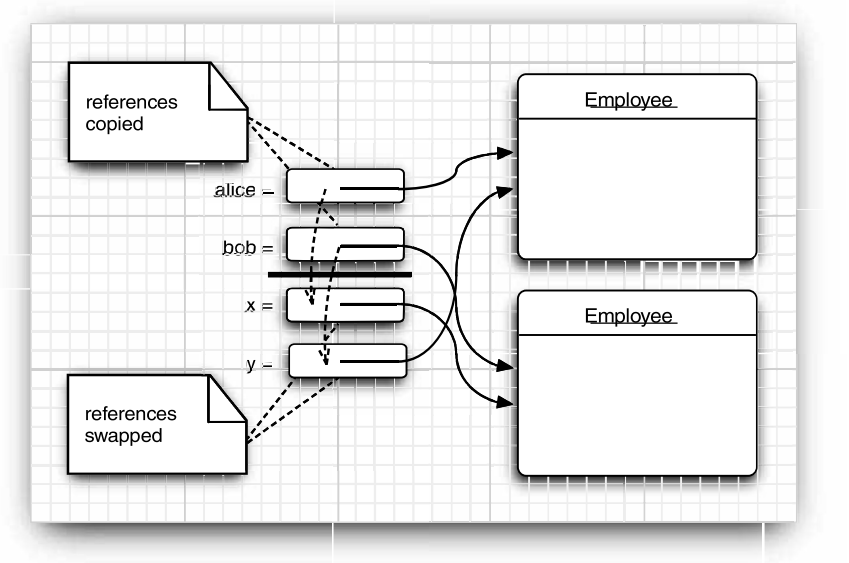

Figure 4–8 Swapping object parameters has no lasting effect

This discussion demonstrates that the Java programming language does not use call by reference for objects. Instead, *object references are passed by value.*

Here is a summary of what you can and cannot do with method parameters in the Java programming language:

- A method cannot modify a parameter of primitive type (that is, numbers or `boolean` values).
- A method can change the *state* of an object parameter.
- A method cannot make an object parameter refer to a new object.

The program in Listing 4–4 demonstrates these facts. The program first tries to triple the value of a number parameter and does not succeed:

```
Testing tripleValue:
Before: percent=10.0
End of method: x=30.0
After: percent=10.0
```

It then successfully triples the salary of an employee:

```
Testing tripleSalary:
Before: salary=50000.0
End of method: salary=150000.0
After: salary=150000.0
```

After the method, the state of the object to which `harry` refers has changed. This is possible because the method modified the state through a copy of the object reference.

Finally, the program demonstrates the failure of the `swap` method:

```
Testing swap:
Before: a=Alice
Before: b=Bob
End of method: x=Bob
End of method: y=Alice
After: a=Alice
After: b=Bob
```

As you can see, the parameter variables `x` and `y` are swapped, but the variables `a` and `b` are not affected.

C++ NOTE: C++ has both call by value and call by reference. You tag reference parameters with `&`. For example, you can easily implement methods `void tripleValue(double& x)` or `void swap(Employee& x, Employee& y)` that modify their reference parameters.

Listing 4–4 ParamTest.java

```
1. /**
2.  * This program demonstrates parameter passing in Java.
3.  * @version 1.00 2000-01-27
4.  * @author Cay Horstmann
5.  */
6. public class ParamTest
7. {
8.    public static void main(String[] args)
9.    {
10.      /*
11.       * Test 1: Methods can't modify numeric parameters
12.       */
```

Listing 4–4 ParamTest.java (continued)

```
13.       System.out.println("Testing tripleValue:");
14.       double percent = 10;
15.       System.out.println("Before: percent=" + percent);
16.       tripleValue(percent);
17.       System.out.println("After: percent=" + percent);
18.
19.       /*
20.        * Test 2: Methods can change the state of object parameters
21.        */
22.       System.out.println("\nTesting tripleSalary:");
23.       Employee harry = new Employee("Harry", 50000);
24.       System.out.println("Before: salary=" + harry.getSalary());
25.       tripleSalary(harry);
26.       System.out.println("After: salary=" + harry.getSalary());
27.
28.       /*
29.        * Test 3: Methods can't attach new objects to object parameters
30.        */
31.       System.out.println("\nTesting swap:");
32.       Employee a = new Employee("Alice", 70000);
33.       Employee b = new Employee("Bob", 60000);
34.       System.out.println("Before: a=" + a.getName());
35.       System.out.println("Before: b=" + b.getName());
36.       swap(a, b);
37.       System.out.println("After: a=" + a.getName());
38.       System.out.println("After: b=" + b.getName());
39.    }
40.
41.    public static void tripleValue(double x) // doesn't work
42.    {
43.       x = 3 * x;
44.       System.out.println("End of method: x=" + x);
45.    }
46.
47.    public static void tripleSalary(Employee x) // works
48.    {
49.       x.raiseSalary(200);
50.       System.out.println("End of method: salary=" + x.getSalary());
51.    }
52.
53.    public static void swap(Employee x, Employee y)
54.    {
55.       Employee temp = x;
56.       x = y;
57.       y = temp;
58.       System.out.println("End of method: x=" + x.getName());
59.       System.out.println("End of method: y=" + y.getName());
60.    }
61. }
```

Listing 4–4 ParamTest.java (continued)

```
62.
63. class Employee // simplified Employee class
64. {
65.    public Employee(String n, double s)
66.    {
67.       name = n;
68.       salary = s;
69.    }
70.
71.    public String getName()
72.    {
73.       return name;
74.    }
75.
76.    public double getSalary()
77.    {
78.       return salary;
79.    }
80.
81.    public void raiseSalary(double byPercent)
82.    {
83.       double raise = salary * byPercent / 100;
84.       salary += raise;
85.    }
86.
87.    private String name;
88.    private double salary;
89. }
```

Object Construction

You have seen how to write simple constructors that define the initial state of your objects. However, because object construction is so important, Java offers quite a variety of mechanisms for writing constructors. We go over these mechanisms in the sections that follow.

Overloading

Recall that the `GregorianCalendar` class had more than one constructor. We could use

```
GregorianCalendar today = new GregorianCalendar();
```

or

```
GregorianCalendar deadline = new GregorianCalendar(2099, Calendar.DECEMBER, 31);
```

This capability is called *overloading*. Overloading occurs if several methods have the same name (in this case, the `GregorianCalendar` constructor method) but different parameters. The compiler must sort out which method to call. It picks the correct method by matching the parameter types in the headers of the various methods with the types of the values used in the specific method call. A compile-time error occurs if the compiler cannot match the parameters or if more than one match is possible. (This process is called *overloading resolution*.)

NOTE: Java allows you to overload any method—not just constructor methods. Thus, to completely describe a method, you need to specify the name of the method together with its parameter types. This is called the *signature* of the method. For example, the `String` class has four public methods called `indexOf`. They have signatures

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

The return type is not part of the method signature. That is, you cannot have two methods with the same names and parameter types but different return types.

Default Field Initialization

If you don't set a field explicitly in a constructor, it is automatically set to a default value: numbers to `0`, `boolean` values to `false`, and object references to `null`. But it is considered poor programming practice to rely on this. Certainly, it makes it harder for someone to understand your code if fields are being initialized invisibly.

NOTE: This is an important difference between fields and local variables. You must always explicitly initialize local variables in a method. But if you don't initialize a field in a class, it is automatically initialized to a default (`0`, `false`, or `null`).

For example, consider the `Employee` class. Suppose you don't specify how to initialize some of the fields in a constructor. By default, the `salary` field would be initialized with `0` and the `name` and `hireDay` fields would be initialized with `null`.

However, that would not be a good idea. If anyone called the `getName` or `getHireDay` method, then they would get a `null` reference that they probably don't expect:

```
Date h = harry.getHireDay();
calendar.setTime(h); // throws exception if h is null
```

Default Constructors

A *default constructor* is a constructor with no parameters. For example, here is a default constructor for the `Employee` class:

```
public Employee()
{
   name = "";
   salary = 0;
   hireDay = new Date();
}
```

If you write a class with no constructors whatsoever, then a default constructor is provided for you. This default constructor sets *all* the instance fields to their default values. So, all numeric data contained in the instance fields would be `0`, all `boolean` values would be `false`, and all object variables would be set to `null`.

If a class supplies at least one constructor but does not supply a default constructor, it is illegal to construct objects without construction parameters. For example, our original `Employee` class in Listing 4–2 provided a single constructor:

```
Employee(String name, double salary, int y, int m, int d)
```

With that class, it was not legal to construct default employees. That is, the call

```
e = new Employee();
```

would have been an error.

CAUTION: Please keep in mind that you get a free default constructor *only* when your class has no other constructors. If you write your class with even a single constructor of your own and you want the users of your class to have the ability to create an instance by a call to

```
new ClassName()
```

then you must provide a default constructor (with no parameters). Of course, if you are happy with the default values for all fields, you can simply supply

```
public ClassName()
{
}
```

Explicit Field Initialization

Because you can overload the constructor methods in a class, you can obviously build in many ways to set the initial state of the instance fields of your classes. It is always a good idea to make sure that, regardless of the constructor call, every instance field is set to something meaningful.

You can simply assign a value to any field in the class definition. For example:

```
class Employee
{
   . . .
   private String name = "";
}
```

This assignment is carried out before the constructor executes. This syntax is particularly useful if all constructors of a class need to set a particular instance field to the same value.

The initialization value doesn't have to be a constant value. Here is an example in which a field is initialized with a method call. Consider an `Employee` class where each employee has an `id` field. You can initialize it as follows:

```
class Employee
{
   . . .
   static int assignId()
   {
      int r = nextId;
      nextId++;
      return r;
   }
   . . .
   private int id = assignId();
}
```

C++ NOTE: In C++, you cannot directly initialize instance fields of a class. All fields must be set in a constructor. However, C++ has a special initializer list syntax, such as

```
Employee::Employee(String n, double s, int y, int m, int d) // C++
:  name(n),
   salary(s),
   hireDay(y, m, d)
{
}
```

C++ uses this special syntax to call field constructors. In Java, there is no need for it because objects have no subobjects, only pointers to other objects.

Parameter Names

When you write very trivial constructors (and you'll write a lot of them), then it can be somewhat frustrating to come up with parameter names.

We have generally opted for single-letter parameter names:

```
public Employee(String n, double s)
{
   name = n;
   salary = s;
}
```

However, the drawback is that you need to read the code to tell what then and s parameters mean.

Some programmers prefix each parameter with an "a":

```
public Employee(String aName, double aSalary)
{
   name = aName;
   salary = aSalary;
}
```

That is quite neat. Any reader can immediately figure out the meaning of the parameters.

Another commonly used trick relies on the fact that parameter variables *shadow* instance fields with the same name. For example, if you call a parameter salary, then salary refers to the parameter, not the instance field. But you can still access the instance field as this.salary. Recall that this denotes the implicit parameter, that is, the object that is being constructed. Here is an example:

```
public Employee(String name, double salary)
{
   this.name = name;
   this.salary = salary;
}
```

C++ NOTE: In C++, it is common to prefix instance fields with an underscore or a fixed letter. (The letters m and x are common choices.) For example, the salary field might be called _salary, mSalary, or xSalary. Java programmers don't usually do that.

Calling Another Constructor

The keyword `this` refers to the implicit parameter of a method. However, the keyword has a second meaning.

If *the first statement of a constructor* has the form `this(. . .)`, then the constructor calls another constructor of the same class. Here is a typical example:

```
public Employee(double s)
{
   // calls Employee(String, double)
   this("Employee #" + nextId, s);
   nextId++;
}
```

When you call `new Employee(60000)`, then the `Employee(double)` constructor calls the `Employee(String, double)` constructor.

Using the `this` keyword in this manner is useful—you only need to write common construction code once.

C++ NOTE: The `this` reference in Java is identical to the `this` pointer in C++. However, in C++ it is not possible for one constructor to call another. If you want to factor out common initialization code in C++, you must write a separate method.

Initialization Blocks

You have already seen two ways to initialize a data field:

- By setting a value in a constructor
- By assigning a value in the declaration

There is actually a *third* mechanism in Java; it's called an *initialization block*. Class declarations can contain arbitrary blocks of code. These blocks are executed whenever an object of that class is constructed. For example:

```
class Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public Employee()
   {
      name = "";
      salary = 0;
   }
   . . .
   private static int nextId;

   private int id;
   private String name;
   private double salary;
   . . .
```

```
   // object initialization block
   {
      id = nextId;
      nextId++;
   }
}
```

In this example, the `id` field is initialized in the object initialization block, no matter which constructor is used to construct an object. The initialization block runs first, and then the body of the constructor is executed.

This mechanism is never necessary and is not common. It usually is more straightforward to place the initialization code inside a constructor.

NOTE: It is legal to set fields in initialization blocks even though they are only defined later in the class. Some versions of Sun's Java compiler handled this case incorrectly (bug # 4459133). This bug has been fixed in Java SE 1.4.1. However, to avoid circular definitions, it is not legal to read from fields that are only initialized later. The exact rules are spelled out in section 8.3.2.3 of the Java Language Specification (`http://java.sun.com/docs/books/jls`). Because the rules are complex enough to baffle the compiler implementors, we suggest that you place initialization blocks after the field definitions.

With so many ways of initializing data fields, it can be quite confusing to give all possible pathways for the construction process. Here is what happens in detail when a constructor is called:

1. All data fields are initialized to their default value (`0`, `false`, or `null`).
2. All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.
3. If the first line of the constructor calls a second constructor, then the body of the second constructor is executed.
4. The body of the constructor is executed.

Naturally, it is always a good idea to organize your initialization code so that another programmer could easily understand it without having to be a language lawyer. For example, it would be quite strange and somewhat error prone to have a class whose constructors depend on the order in which the data fields are declared.

You initialize a static field either by supplying an initial value or by using a static initialization block. You have already seen the first mechanism:

```
static int nextId = 1;
```

If the static fields of your class require complex initialization code, use a static initialization block.

Place the code inside a block and tag it with the keyword `static`. Here is an example. We want the employee ID numbers to start at a random integer less than 10,000.

```
// static initialization block
static
{
   Random generator = new Random();
   nextId = generator.nextInt(10000);
}
```

Static initialization occurs when the class is first loaded. Like instance fields, static fields are `0`, `false`, or `null` unless you explicitly set them to another value. All static field initializers and static initialization blocks are executed in the order in which they occur in the class declaration.

NOTE: Here is a Java trivia fact to amaze your fellow Java coders: You can write a "Hello, World" program in Java without ever writing a `main` method.

```
public class Hello
{
   static
   {
      System.out.println("Hello, World");
   }
}
```

When you invoke the class with `java Hello`, the class is loaded, the static initialization block prints "Hello, World," and only then do you get an ugly error message that `main` is not defined. You can avoid that blemish by calling `System.exit(0)` at the end of the static initialization block.

The program in Listing 4–5 shows many of the features that we discussed in this section:

- Overloaded constructors
- A call to another constructor with `this(...)`
- A default constructor
- An object initialization block
- A static initialization block
- An instance field initialization

Listing 4–5 ConstructorTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates object construction.
5.  * @version 1.01 2004-02-19
6.  * @author Cay Horstmann
7.  */
8. public class ConstructorTest
9. {
10.    public static void main(String[] args)
11.    {
12.       // fill the staff array with three Employee objects
13.       Employee[] staff = new Employee[3];
14.
15.       staff[0] = new Employee("Harry", 40000);
16.       staff[1] = new Employee(60000);
17.       staff[2] = new Employee();
18.
```

Listing 4–5 ConstructorTest.java (continued)

```
      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
               + e.getSalary());
   }
}

class Employee
{
   // three overloaded constructors
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public Employee(double s)
   {
      // calls the Employee(String, double) constructor
      this("Employee #" + nextId, s);
   }

   // the default constructor
   public Employee()
   {
      // name initialized to ""--see below
      // salary not explicitly set--initialized to 0
      // id initialized in initialization block
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public int getId()
   {
      return id;
   }

   private static int nextId;

   private int id;
   private String name = ""; // instance field initialization
   private double salary;
```

Listing 4–5 ConstructorTest.java (continued)

```
69.
70.    // static initialization block
71.    static
72.    {
73.       Random generator = new Random();
74.       // set nextId to a random number between 0 and 9999
75.       nextId = generator.nextInt(10000);
76.    }
77.
78.    // object initialization block
79.    {
80.       id = nextId;
81.       nextId++;
82.    }
83. }
```

API `java.util.Random` 1.0

- `Random()`
  constructs a new random number generator.
- `int nextInt(int n)` 1.2
  returns a random number between 0 and `n` – 1.

Object Destruction and the `finalize` *Method*

Some object-oriented programming languages, notably C++, have explicit destructor methods for any cleanup code that may be needed when an object is no longer used. The most common activity in a destructor is reclaiming the memory set aside for objects. Because Java does automatic garbage collection, manual memory reclamation is not needed and so Java does not support destructors.

Of course, some objects utilize a resource other than memory, such as a file or a handle to another object that uses system resources. In this case, it is important that the resource be reclaimed and recycled when it is no longer needed.

You can add a `finalize` method to any class. The `finalize` method will be called before the garbage collector sweeps away the object. In practice, *do not rely on the* `finalize` *method* for recycling any resources that are in short supply—you simply cannot know when this method will be called.

NOTE: The method call `System.runFinalizersOnExit(true)` guarantees that finalizer methods are called before Java shuts down. However, this method is inherently unsafe and has been deprecated. An alternative is to add "shutdown hooks" with the method `Runtime.addShutdownHook`—see the API documentation for details.

If a resource needs to be closed as soon as you have finished using it, you need to manage it manually. Supply a method such as `dispose` or `close` that *you* call to clean up what

needs cleaning. Just as importantly, if a class you use has such a method, you need to call it when you are done with the object.

Packages

Java allows you to group classes in a collection called a *package*. Packages are convenient for organizing your work and for separating your work from code libraries provided by others.

The standard Java library is distributed over a number of packages, including `java.lang`, `java.util`, `java.net`, and so on. The standard Java packages are examples of hierarchical packages. Just as you have nested subdirectories on your hard disk, you can organize packages by using levels of nesting. All standard Java packages are inside the `java` and `javax` package hierarchies.

The main reason for using packages is to guarantee the uniqueness of class names. Suppose two programmers come up with the bright idea of supplying an `Employee` class. As long as both of them place their class into different packages, there is no conflict. In fact, to absolutely guarantee a unique package name, Sun recommends that you use your company's Internet domain name (which is known to be unique) written in reverse. You then use subpackages for different projects. For example, `horstmann.com` is a domain that one of the authors registered. Written in reverse order, it turns into the package `com.horstmann`. That package can then be further subdivided into subpackages such as `com.horstmann.corejava`.

From the point of view of the compiler, there is absolutely no relationship between nested packages. For example, the packages `java.util` and `java.util.jar` have nothing to do with each other. Each is its own independent collection of classes.

Class Importation

A class can use all classes from its own package and all *public* classes from other packages.

You can access the public classes in another package in two ways. The first is simply to add the full package name in front of *every* class name. For example:

```
java.util.Date today = new java.util.Date();
```

That is obviously tedious. The simpler, and more common, approach is to use the `import` statement. The point of the `import` statement is simply to give you a shorthand to refer to the classes in the package. Once you use `import`, you no longer have to give the classes their full names.

You can import a specific class or the whole package. You place `import` statements at the top of your source files (but below any `package` statements). For example, you can import all classes in the `java.util` package with the statement

```
import java.util.*;
```

Then you can use

```
Date today = new Date();
```

without a package prefix. You can also import a specific class inside a package:

```
import java.util.Date;
```

The `java.util.*` syntax is less tedious. It has no negative effect on code size. However, if you import classes explicitly, the reader of your code knows exactly which classes you use.

TIP: In Eclipse, you can select the menu option Source -> Organize Imports. Package statements such as `import java.util.*;` are automatically expanded into a list of specific imports such as

```
import java.util.ArrayList;
import java.util.Date;
```

This is an extremely convenient feature.

However, note that you can only use the `*` notation to import a single package. You cannot use `import java.*` or `import java.*.*` to import all packages with the `java` prefix.

Most of the time, you just import the packages that you need, without worrying too much about them. The only time that you need to pay attention to packages is when you have a name conflict. For example, both the `java.util` and `java.sql` packages have a `Date` class. Suppose you write a program that imports both packages.

```
import java.util.*;
import java.sql.*;
```

If you now use the `Date` class, then you get a compile-time error:

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

The compiler cannot figure out which `Date` class you want. You can solve this problem by adding a specific `import` statement:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

What if you really need both `Date` classes? Then you need to use the full package name with every class name.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date(...);
```

Locating classes in packages is an activity of the *compiler*. The bytecodes in class files always use full package names to refer to other classes.

C++ NOTE: C++ programmers usually confuse `import` with `#include`. The two have nothing in common. In C++, you must use `#include` to include the declarations of external features because the C++ compiler does not look inside any files except the one that it is compiling and explicitly included header files. The Java compiler will happily look inside other files provided you tell it where to look.

In Java, you can entirely avoid the `import` mechanism by explicitly naming all classes, such as `java.util.Date`. In C++, you cannot avoid the `#include` directives.

The only benefit of the `import` statement is convenience. You can refer to a class by a name shorter than the full package name. For example, after an `import java.util.*` (or `import java.util.Date`) statement, you can refer to the `java.util.Date` class simply as `Date`.

The analogous construction to the package mechanism in C++ is the namespace feature. Think of the `package` and `import` statements in Java as the analogs of the `namespace` and `using` directives in C++.

Static Imports

Starting with Java SE 5.0, the `import` statement has been enhanced to permit the importing of static methods and fields, not just classes.

For example, if you add the directive

```
import static java.lang.System.*;
```

to the top of your source file, then you can use static methods and fields of the `System` class without the class name prefix:

```
out.println("Goodbye, World!"); // i.e., System.out
exit(0); // i.e., System.exit
```

You can also import a specific method or field:

```
import static java.lang.System.out;
```

In practice, it seems doubtful that many programmers will want to abbreviate `System.out` or `System.exit`. The resulting code seems less clear. But there are two practical uses for static imports.

- Mathematical functions: If you use a static import for the `Math` class, you can use mathematical functions in a more natural way. For example,

  ```
  sqrt(pow(x, 2) + pow(y, 2))
  ```

  seems much clearer than

  ```
  Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
  ```

- Cumbersome constants: If you use lots of constants with tedious names, you will welcome static import. For example,

  ```
  if (d.get(DAY_OF_WEEK) == MONDAY)
  ```

  is easier on the eye than

  ```
  if (d.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
  ```

Addition of a Class into a Package

To place classes inside a package, you must put the name of the package at the top of your source file, *before* the code that defines the classes in the package. For example, the file `Employee.java` in Listing 4–7 starts out like this:

```
package com.horstmann.corejava;

public class Employee
{
   . . .
}
```

If you don't put a `package` statement in the source file, then the classes in that source file belong to the *default package*. The default package has no package name. Up to now, all our example classes were located in the default package.

You place source files into a subdirectory that matches the full package name. For example, all source files in the package `com.horstmann.corejava` package should be in a subdirectory `com/horstmann/corejava` (`com\horstmann\corejava` on Windows). The compiler places the class files into the same directory structure.

The program in Listings 4–6 and 4–7 is distributed over two packages: the `PackageTest` class belongs to the default package and the `Employee` class belongs to the `com.horstmann.corejava` package. Therefore, the `Employee.java` file must be contained in a subdirectory `com/horstmann/corejava`. In other words, the directory structure is as follows:

```
. (base directory)
   ├─ PackageTest.java
   ├─ PackageTest.class
   └─ com/
      └─ horstmann/
         └─ corejava/
            ├─ Employee.java
            └─ Employee.class
```

To compile this program, simply change to the base directory and run the command

```
javac PackageTest.java
```

The compiler automatically finds the file `com/horstmann/corejava/Employee.java` and compiles it.

Let's look at a more realistic example, in which we don't use the default package but have classes distributed over several packages (`com.horstmann.corejava` and `com.mycompany`).

```
. (base directory)
   └─ com/
      ├─ horstmann/
      │  └─ corejava/
      │     ├─ Employee.java
      │     └─ Employee.class
      └─ mycompany/
         ├─ PayrollApp.java
         └─ PayrollApp.class
```

In this situation, you still must compile and run classes from the *base* directory, that is, the directory containing the `com` directory:

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

Note again that the compiler operates on *files* (with file separators and an extension `.java`), whereas the Java interpreter loads a *class* (with dot separators).

CAUTION: The compiler does *not* check the directory structure when it compiles source files. For example, suppose you have a source file that starts with the directive

```
package com.mycompany;
```

You can compile the file even if it is not contained in a subdirectory `com/mycompany`. The source file will compile without errors *if it doesn't depend on other packages*. However, the resulting program will not run. The *virtual machine* won't find the resulting classes when you try to run the program.

Listing 4–6 PackageTest.java

```
1. import com.horstmann.corejava.*;
2. // the Employee class is defined in that package
3.
4. import static java.lang.System.*;
5.
6. /**
7.  * This program demonstrates the use of packages.
8.  * @author cay
9.  * @version 1.11 2004-02-19
10.  * @author Cay Horstmann
11.  */
12. public class PackageTest
13. {
14.    public static void main(String[] args)
15.    {
16.       // because of the import statement, we don't have to use com.horstmann.corejava.Employee here
17.       Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
18.
19.       harry.raiseSalary(5);
20.
21.       // because of the static import statement, we don't have to use System.out here
22.       out.println("name=" + harry.getName() + ",salary=" + harry.getSalary());
23.    }
24. }
```

Listing 4–7 Employee.java

```
1. package com.horstmann.corejava;
2.
3. // the classes in this file are part of this package
4.
5. import java.util.*;
6.
7. // import statements come after the package statement
8.
9. /**
10.  * @version 1.10 1999-12-18
11.  * @author Cay Horstmann
12.  */
13. public class Employee
14. {
15.    public Employee(String n, double s, int year, int month, int day)
16.    {
17.       name = n;
18.       salary = s;
19.       GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
```

Listing 4–7 Employee.java (continued)

```
20.       // GregorianCalendar uses 0 for January
21.       hireDay = calendar.getTime();
22.    }
23.
24.    public String getName()
25.    {
26.       return name;
27.    }
28.
29.    public double getSalary()
30.    {
31.       return salary;
32.    }
33.
34.    public Date getHireDay()
35.    {
36.       return hireDay;
37.    }
38.
39.    public void raiseSalary(double byPercent)
40.    {
41.       double raise = salary * byPercent / 100;
42.       salary += raise;
43.    }
44.
45.    private String name;
46.    private double salary;
47.    private Date hireDay;
48. }
```

Package Scope

You have already encountered the access modifiers `public` and `private`. Features tagged as `public` can be used by any class. Private features can be used only by the class that defines them. If you don't specify either `public` or `private`, the feature (that is, the class, method, or variable) can be accessed by all methods in the same *package*.

Consider the program in Listing 4–2 on page 124. The `Employee` class was not defined as a public class. Therefore, only other classes in the same package—the default package in this case—such as `EmployeeTest` can access it. For classes, this is a reasonable default. However, for variables, this default was an unfortunate choice. Variables must explicitly be marked `private` or they will default to being package visible. This, of course, breaks encapsulation. The problem is that it is awfully easy to forget to type the `private` keyword. Here is an example from the `Window` class in the `java.awt` package, which is part of the source code supplied with the JDK:

```
public class Window extends Container
{
   String warningString;
   . . .
}
```

Note that the `warningString` variable is not `private`! That means the methods of all classes in the `java.awt` package can access this variable and set it to whatever they like (such as `"Trust me!"`). Actually, the only methods that access this variable are in the `Window` class, so it would have been entirely appropriate to make the variable private. We suspect that the programmer typed the code in a hurry and simply forgot the `private` modifier. (We won't mention the programmer's name to protect the guilty—you can look into the source file yourself.)

NOTE: Amazingly enough, this problem has never been fixed, even though we have pointed it out in eight editions of this book—apparently the library implementors don't read *Core Java*. Not only that—new fields have been added to the class over time, and about half of them aren't private either.

Is this really a problem? It depends. By default, packages are not closed entities. That is, anyone can add more classes to a package. Of course, hostile or clueless programmers can then add code that modifies variables with package visibility. For example, in early versions of the Java programming language, it was an easy matter to smuggle another class into the `java.awt` package. Simply start out the class with

```
package java.awt;
```

Then, place the resulting class file inside a subdirectory `java/awt` somewhere on the class path, and you have gained access to the internals of the `java.awt` package. Through this subterfuge, it was possible to set the warning string (see Figure 4–9).

Figure 4–9 Changing the warning string in an applet window

Starting with version 1.2, the JDK implementors rigged the class loader to explicitly disallow loading of user-defined classes whose package name starts with `"java."`! Of course, your own classes won't benefit from that protection. Instead, you can use another mechanism, *package sealing*, to address the issue of promiscuous package access. If you seal a package, no further classes can be added to it. You will see in Chapter 10 how you can produce a JAR file that contains sealed packages.

The Class Path

As you have seen, classes are stored in subdirectories of the file system. The path to the class must match the package name.

Class files can also be stored in a JAR (Java archive) file. A JAR file contains multiple class files and subdirectories in a compressed format, saving space and improving performance. When you use a third-party library in your programs, you will usually be given one or more JAR files to include. The JDK also supplies a number of JAR files, such as the file `jre/lib/rt.jar` that contains thousands of library classes. You will see in Chapter 10 how to create your own JAR files.

TIP: JAR files use the ZIP format to organize files and subdirectories. You can use any ZIP utility to peek inside `rt.jar` and other JAR files.

To share classes among programs, you need to do the following:

1. Place your class files inside a directory, for example, `/home/user/classdir`. Note that this directory is the *base* directory for the package tree. If you add the class `com.horstmann.corejava.Employee`, then the `Employee.class` file must be located in the subdirectory `/home/user/classdir/com/horstmann/corejava`.
2. Place any JAR files inside a directory, for example, `/home/user/archives`.
3. Set the *class path*. The class path is the collection of all locations that can contain class files.

On UNIX, the elements on the class path are separated by colons:

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

On Windows, they are separated by semicolons:

```
c:\classdir;.;c:\archives\archive.jar
```

In both cases, the period denotes the current directory.

This class path contains

- The base directory `/home/user/classdir` or `c:\classdir`;
- The current directory (`.`); and
- The JAR file `/home/user/archives/archive.jar` or `c:\archives\archive.jar`.

Starting with Java SE 6, you can specify a wildcard for a JAR file directory, like this:

```
/home/user/classdir:.:/home/user/archives/'*'
```

or

```
c:\classdir;.;c:\archives\*
```

In UNIX, the `*` must be escaped to prevent shell expansion.

All JAR files (but not `.class` files) in the `archives` directory are included in this class path.

The runtime library files (`rt.jar` and the other JAR files in the `jre/lib` and `jre/lib/ext` directories) are always searched for classes; you don't include them explicitly in the class path.

CAUTION: The `javac` compiler always looks for files in the current directory, but the `java` virtual machine launcher only looks into the current directory if the "." directory is on the class path. If you have no class path set, this is not a problem—the default class path consists of the "." directory. But if you have set the class path and forgot to include the "." directory, your programs will compile without error, but they won't run.

The class path lists all directories and archive files that are *starting points* for locating classes. Let's consider our sample class path:

```
/home/user/classdir:.:/home/user/archives/archive.jar
```

Suppose the virtual machine searches for the class file of the `com.horstmann.corejava.Employee` class. It first looks in the system class files that are stored in archives in the `jre/lib` and `jre/lib/ext` directories. It won't find the class file there, so it turns to the class path. It then looks for the following files:

- `/home/user/classdir/com/horstmann/corejava/Employee.class`
- `com/horstmann/corejava/Employee.class` starting from the current directory
- `com/horstmann/corejava/Employee.class` inside `/home/user/archives/archive.jar`

The compiler has a harder time locating files than does the virtual machine. If you refer to a class without specifying its package, the compiler first needs to find out the package that contains the class. It consults all `import` directives as possible sources for the class. For example, suppose the source file contains directives

```
import java.util.*;
import com.horstmann.corejava.*;
```

and the source code refers to a class `Employee`. The compiler then tries to find `java.lang.Employee` (because the `java.lang` package is always imported by default), `java.util.Employee`, `com.horstmann.corejava.Employee`, and `Employee` in the current package. It searches for *each* of these classes in all of the locations of the class path. It is a compile-time error if more than one class is found. (Because classes must be unique, the order of the `import` statements doesn't matter.)

The compiler goes one step further. It looks at the *source files* to see if the source is newer than the class file. If so, the source file is recompiled automatically. Recall that you can import only public classes from other packages. A source file can only contain one public class, and the names of the file and the public class must match. Therefore, the compiler can easily locate source files for public classes. However, you can import nonpublic classes from the current package. These classes may be defined in source files with different names. If you import a class from the current package, the compiler searches *all* source files of the current package to see which one defines the class.

Setting the Class Path

It is best to specify the class path with the `-classpath` (or `-cp`) option:

```
java -classpath /home/user/classdir:.:/home/user/archives/archive.jar MyProg.java
```

or

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

The entire command must be typed onto a single line. It is a good idea to place such a long command line into a shell script or a batch file.

Using the `-classpath` option is the preferred approach for setting the class path. An alternate approach is the `CLASSPATH` environment variable. The details depend on your shell. With the Bourne Again shell (bash), use the command

```
export CLASSPATH=/home/user/classdir:.:/home/user/archives/archive.jar
```

With the C shell, use the command

```
setenv CLASSPATH /home/user/classdir:.:/home/user/archives/archive.jar
```

With the Windows shell, use

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

The class path is set until the shell exits.

CAUTION: Some people recommend to set the `CLASSPATH` environment variable permanently. This is generally a bad idea. People forget the global setting, and then they are surprised when their classes are not loaded properly. A particularly reprehensible example is Apple's QuickTime installer in Windows. It globally sets `CLASSPATH` to point to a JAR file that it needs, but it does not include the current directory in the classpath. As a result, countless Java programmers have been driven to distraction when their programs compiled but failed to run.

CAUTION: Some people recommend to bypass the class path altogether, by dropping all JAR files into the `jre/lib/ext` directory. That is truly bad advice, for two reasons. Archives that manually load other classes do not work correctly when they are placed in the extension directory. (See Volume II, Chapter 9 for more information on class loaders.) Moreover, programmers have a tendency to forget about the files they placed there months ago. Then, they scratch their heads when the class loader seems to ignore their carefully crafted class path, when it is actually loading long-forgotten classes from the extension directory.

Documentation Comments

The JDK contains a very useful tool, called `javadoc`, that generates HTML documentation from your source files. In fact, the on-line API documentation that we described in Chapter 3 is simply the result of running `javadoc` on the source code of the standard Java library.

If you add comments that start with the special delimiter `/**` to your source code, you too can easily produce professional-looking documentation. This is a very nice scheme because it lets you keep your code and documentation in one place. If you put your documentation into a separate file, then you probably know that the code and comments tend to diverge over time. But because the documentation comments are in the same file as the source code, it is an easy matter to update both and run `javadoc` again.

Comment Insertion

The `javadoc` utility extracts information for the following items:

- Packages
- Public classes and interfaces
- Public and protected methods
- Public and protected fields

Protected features are introduced in Chapter 5, interfaces in Chapter 6.

You can (and should) supply a comment for each of these features. Each comment is placed immediately *above* the feature it describes. A comment starts with a `/**` and ends with a `*/`.

Each `/** . . . */` documentation comment contains *free-form text* followed by *tags*. A tag starts with an `@`, such as `@author` or `@param`.

The *first sentence* of the free-form text should be a *summary statement*. The `javadoc` utility automatically generates summary pages that extract these sentences.

In the free-form text, you can use HTML modifiers such as `<em>...</em>` for emphasis, `<code>...</code>` for a monospaced "typewriter" font, `<strong>...</strong>` for strong emphasis, and even `<img ...>` to include an image. You should, however, stay away from headings `<h1>` or rules `<hr>` because they can interfere with the formatting of the document.

NOTE: If your comments contain links to other files such as images (for example, diagrams or images of user interface components), place those files into a subdirectory of the directory containing the source file named `doc-files`. The `javadoc` utility will copy the `doc-files` directories and their contents from the source directory to the documentation directory. You need to use the `doc-files` directory in your link, such as `<img src="doc-files/uml.png" alt="UML diagram"/>`.

Class Comments

The class comment must be placed *after* any `import` statements, directly before the `class` definition.

Here is an example of a class comment:

```
/**
 * A <code>Card</code> object represents a playing card, such
 * as "Queen of Hearts". A card has a suit (Diamond, Heart,
 * Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
 * 12 = Queen, 13 = King)
 */
public class Card
{
   . . .
}
```

NOTE: There is no need to add an `*` in front of every line. For example, the following comment is equally valid:

```
/**
   A <code>Card</code> object represents a playing card, such
   as "Queen of Hearts". A card has a suit (Diamond, Heart,
   Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
   12 = Queen, 13 = King).
*/
```

However, most IDEs supply the asterisks automatically and rearrange them when the line breaks change.

Method Comments

Each method comment must immediately precede the method that it describes. In addition to the general-purpose tags, you can use the following tags:

- `@param` *variable description*

  This tag adds an entry to the "parameters" section of the current method. The description can span multiple lines and can use HTML tags. All `@param` tags for one method must be kept together.
- `@return` *description*

  This tag adds a "returns" section to the current method. The description can span multiple lines and can use HTML tags.
- `@throws` *class description*

  This tag adds a note that this method may throw an exception. Exceptions are the topic of Chapter 11.

Here is an example of a method comment:

```
/**
 * Raises the salary of an employee.
 *  @param byPercent the percentage by which to raise the salary (e.g. 10 = 10%)
 *  @return the amount of the raise
 */
public double raiseSalary(double byPercent)
{
   double raise = salary * byPercent / 100;
   salary += raise;
   return raise;
}
```

Field Comments

You only need to document public fields—generally that means static constants. For example:

```
/**
 *  The "Hearts" card suit
 */
public static final int HEARTS = 1;
```

General Comments

The following tags can be used in class documentation comments:

- `@author` *name*

  This tag makes an "author" entry. You can have multiple `@author` tags, one for each author.
- `@version` *text*

  This tag makes a "version" entry. The `text` can be any description of the current version.

The following tags can be used in all documentation comments:

- `@since` *text*

  This tag makes a "since" entry. The `text` can be any description of the version that introduced this feature. For example, `@since version 1.7.1`

- `@deprecated` *text*

  This tag adds a comment that the class, method, or variable should no longer be used. The `text` should suggest a replacement. For example:

  ```
  @deprecated Use <code>setVisible(true)</code> instead
  ```

You can use hyperlinks to other relevant parts of the `javadoc` documentation, or to external documents, with the `@see` and `@link` tags.

- `@see` *reference*

  This tag adds a hyperlink in the "see also" section. It can be used with both classes and methods. Here, *reference* can be one of the following:

  ```
  package.class#feature   label
  <a href="...">label</a>
  "text"
  ```

  The first case is the most useful. You supply the name of a class, method, or variable, and `javadoc` inserts a hyperlink to the documentation. For example,

  ```
  @see com.horstmann.corejava.Employee#raiseSalary(double)
  ```

  makes a link to the `raiseSalary(double)` method in the `com.horstmann.corejava.Employee` class. You can omit the name of the package or both the package and class name. Then, the feature will be located in the current package or class.

  Note that you must use a `#`, not a period, to separate the class from the method or variable name. The Java compiler itself is highly skilled in guessing the various meanings of the period character, as separator between packages, subpackages, classes, inner classes, and methods and variables. But the `javadoc` utility isn't quite as clever, and you have to help it along.

  If the `@see` tag is followed by a `<` character, then you need to specify a hyperlink. You can link to any URL you like. For example:

  ```
  @see <a href="www.horstmann.com/corejava.html">The Core Java home page</a>
  ```

  In each of these cases, you can specify an optional *label* that will appear as the link anchor. If you omit the label, then the user will see the target code name or URL as the anchor.

  If the `@see` tag is followed by a `"` character, then the text is displayed in the "see also" section. For example:

  ```
  @see "Core Java 2 volume 2"
  ```

  You can add multiple `@see` tags for one feature, but you must keep them all together.

- If you like, you can place hyperlinks to other classes or methods anywhere in any of your documentation comments. You insert a special tag of the form

  `{@link` *package.class#feature label*`}`

  anywhere in a comment. The feature description follows the same rules as for the `@see` tag.

Package and Overview Comments

You place class, method, and variable comments directly into the Java source files, delimited by `/** . . . */` documentation comments. However, to generate package comments, you need to add a separate file in each package directory. You have two choices:

1. Supply an HTML file named `package.html`. All text between the tags `<body>...</body>` is extracted.
2. Supply a Java file named `package-info.java`. The file must contain an initial Javadoc comment, delimited with `/**` and `*/`, followed by a `package` statement. It should contain no further code or comments.

You can also supply an overview comment for all source files. Place it in a file called `overview.html`, located in the parent directory that contains all the source files. All text between the tags `<body>...</body>` is extracted. This comment is displayed when the user selects "Overview" from the navigation bar.

Comment Extraction

Here, *docDirectory* is the name of the directory where you want the HTML files to go. Follow these steps:

1. Change to the directory that contains the source files you want to document. If you have nested packages to document, such as `com.horstmann.corejava`, you must be working in the directory that contains the subdirectory `com`. (This is the directory that contains the `overview.html` file if you supplied one.)
2. Run the command

   `javadoc -d` *docDirectory nameOfPackage*

   for a single package. Or run

   `javadoc -d` *docDirectory* $nameOfPackage_1$ $nameOfPackage_2$...

   to document multiple packages. If your files are in the default package, then instead run

   `javadoc -d` *docDirectory* `*.java`

If you omit the `-d` *docDirectory* option, then the HTML files are extracted to the current directory. That can get messy, and we don't recommend it.

The `javadoc` program can be fine-tuned by numerous command-line options. For example, you can use the `-author` and `-version` options to include the `@author` and `@version` tags in the documentation. (By default, they are omitted.) Another useful option is `-link`, to include hyperlinks to standard classes. For example, if you use the command

```
javadoc -link http://java.sun.com/javase/6/docs/api *.java
```

all standard library classes are automatically linked to the documentation on the Sun web site.

If you use the `-linksource` option, each source file is converted to HTML (without color coding, but with line numbers), and each class and method name turns into a hyperlink to the source.

For additional options, we refer you to the on-line documentation of the `javadoc` utility at `http://java.sun.com/javase/javadoc`.

NOTE: If you require further customization, for example, to produce documentation in a format other than HTML, you can supply your own *doclet* to generate the output in any form you desire. Clearly, this is a specialized need, and we refer you to the on-line documentation for details on doclets at `http://java.sun.com/j2se/javadoc`.

TIP: A useful doclet is DocCheck at `http://java.sun.com/j2se/javadoc/doccheck/`. It scans a set of source files for missing documentation comments.

Class Design Hints

Without trying to be comprehensive or tedious, we want to end this chapter with some hints that may make your classes more acceptable in well-mannered OOP circles.

1. *Always keep data private.*

   This is first and foremost: doing anything else violates encapsulation. You may need to write an accessor or mutator method occasionally, but you are still better off keeping the instance fields private. Bitter experience has shown that how the data are represented may change, but how they are used will change much less frequently. When data are kept private, changes in their representation do not affect the user of the class, and bugs are easier to detect.

2. *Always initialize data.*

   Java won't initialize local variables for you, but it will initialize instance fields of objects. Don't rely on the defaults, but initialize the variables explicitly, either by supplying a default or by setting defaults in all constructors.

3. *Don't use too many basic types in a class.*

   The idea is to replace multiple *related* uses of basic types with other classes. This keeps your classes easier to understand and to change. For example, replace the following instance fields in a `Customer` class

   ```
   private String street;
   private String city;
   private String state;
   private int zip;
   ```

   with a new class called `Address`. This way, you can easily cope with changes to addresses, such as the need to deal with international addresses.

4. *Not all fields need individual field accessors and mutators.*

   You may need to get and set an employee's salary. You certainly won't need to change the hiring date once the object is constructed. And, quite often, objects have instance fields that you don't want others to get or set, for example, an array of state abbreviations in an `Address` class.

5. *Use a standard form for class definitions.*

   We always list the contents of classes in the following order:

   public features
   package scope features
   private features

   Within each section, we list:

   instance methods
   static methods
   instance fields
   static fields

After all, the users of your class are more interested in the public interface than in the details of the private implementation. And they are more interested in methods than in data.

However, there is no universal agreement on what is the best style. The Sun coding style guide for the Java programming language recommends listing fields first and then methods. Whatever style you use, the most important thing is to be consistent.

6. *Break up classes that have too many responsibilities.*

This hint is, of course, vague: "too many" is obviously in the eye of the beholder. However, if there is an obvious way to make one complicated class into two classes that are conceptually simpler, seize the opportunity. (On the other hand, don't go overboard; 10 classes, each with only one method, is usually overkill.)

Here is an example of a bad design.

```
public class CardDeck // bad design
{
   public CardDeck() { . . . }
   public void shuffle() { . . . }
   public int getTopValue() { . . . }
   public int getTopSuit() { . . . }
   public void draw() { . . . }

   private int[] value;
   private int[] suit;
}
```

This class really implements two separate concepts: a *deck of cards*, with its `shuffle` and `draw` methods, and a *card*, with the methods to inspect the value and suit of a card. It makes sense to introduce a `Card` class that represents an individual card. Now you have two classes, each with its own responsibilities:

```
public class CardDeck
{
   public CardDeck() { . . . }
   public void shuffle() { . . . }
   public Card getTop() { . . . }
   public void draw() { . . . }

   private Card[] cards;
}

public class Card
{
   public Card(int aValue, int aSuit) { . . . }
   public int getValue() { . . . }
   public int getSuit() { . . . }

   private int value;
   private int suit;
}
```

7. *Make the names of your classes and methods reflect their responsibilities.*

   Just as variables should have meaningful names that reflect what they represent, so should classes. (The standard library certainly contains some dubious examples, such as the `Date` class that describes time.)

   A good convention is that a class name should be a noun (`Order`) or a noun preceded by an adjective (`RushOrder`) or a gerund (an "-ing" word, like `BillingAddress`). As for methods, follow the standard convention that accessor methods begin with a lowercase `get` (`getSalary`), and that mutator methods use a lowercase `set` (`setSalary`).

In this chapter, we covered the fundamentals of objects and classes that make Java an "object-based" language. In order to be truly object oriented, a programming language must also support inheritance and polymorphism. The Java support for these features is the topic of the next chapter.