Object-Oriented Programming

Topics in This Chapter

Chapter 2

In object-oriented programming, work is carried out by collaborating objects whose behavior is defined by the classes to which they belong. Java was one of the first mainstream programming languages to fully embrace object-oriented programming. As you have already seen, in Java every method is declared in a class and, except for a few primitive types, every value is an object. In this chapter, you will learn how to implement your own classes and methods.

The key points of this chapter are:

1. Mutator methods change the state of an object; accessor methods don't.
2. In Java, variables don't hold objects; they hold references to objects.
3. Instance variables and method implementations are declared inside the class declaration.
4. An instance method is invoked on an object, which is accessible through the `this` reference.
5. A constructor has the same name as the class. A class can have multiple (overloaded) constructors.
6. Static variables don't belong to any objects. Static methods are not invoked on objects.
7. Classes are organized into packages. Use import declarations so that you don't have to use the package name in your programs.

8. Classes can be nested in other classes.
9. An inner class is a nonstatic nested class. Its instances have a reference to the object of the enclosing class that constructed it.
10. The `javadoc` utility processes source files, producing HTML files with declarations and programmer-supplied comments.

2.1 Working with Objects

In ancient times, before objects were invented, you wrote programs by calling *functions*. When you call a function, it returns a result that you use without worrying how it was computed. Functions have an important benefit: they allow work to be shared. You can call a function that someone else wrote without having to know how it does its task.

Objects add another dimension. Each object can have its own *state*. The state affects the results that you get from calling a method. For example, if `in` is a `Scanner` object and you call `in.next()`, the object remembers what was read before and gives you the next input token.

When you use objects that someone else implemented and invoke methods on them, you do not need to know what goes on under the hood. This principle, called *encapsulation*, is a key concept of object-oriented programming.

At some point, you may want to make your work available for other programmers by providing them with objects they can use. In Java, you provide a *class*—a mechanism for creating and using objects with the same behavior.

Consider a common task: manipulation of calendar dates. Calendars are somewhat messy, with varying month lengths and leap years, not to mention leap seconds. It makes sense to have experts who figure out those messy details and who provide implementations that other programmers can use. In this situation, objects arise naturally. A date is an object whose methods can provide information such as "on what weekday does this date fall" and "what date is tomorrow."

In Java, experts who understand date computations provided classes for dates and other date-related concepts such as weekdays. If you want to do computations with dates, use one of those classes to create date objects and invoke methods on them, such as a method that yields the weekday or the next date.

Few of us want to ponder the details of date arithmetic, but you are probably an expert in some other area. To enable other programmers to leverage your knowledge, you can provide them with classes. And even if you are not enabling other programmers, you will find it useful in your own work to use classes so that your programs are structured in a coherent way.

Before learning how to declare your own classes, let us run through a nontrivial example of using objects.

The Unix program `cal` prints a calendar for a given month and year, in a format similar to the following:

```
 Mon Tue Wed Thu Fri Sat Sun
                           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
```

How can you implement such a program? With the standard Java library, you use the `LocalDate` class to express a date at some unspecified location. We need an object of that class representing the first of the month. Here is how you get one:

```
LocalDate date = LocalDate.of(year, month, 1);
```

To advance the date, you call `date.plusDays(1)`. The result is a newly constructed `LocalDate` object that is one day further. In our application, we simply reassign the result to the `date` variable:

```
date = date.plusDays(1);
```

You apply methods to obtain information about a date, such as the month on which it falls. We need that information so that we can keep printing while we are still in the same month.

```
while (date.getMonthValue() == month) {
    System.out.printf("%4d", date.getDayOfMonth());
    date = date.plusDays(1);
    ...
}
```

Another method yields the weekday on which a date falls.

```
DayOfWeek weekday = date.getDayOfWeek();
```

You get back an object of another class `DayOfWeek`. In order to compute the indentation of the first day of the month in the calendar, we need know the numerical value of the weekday. There is a method for that:

```
int value = weekday.getValue();
for (int i = 1; i < value; i++)
    System.out.print("    ");
```

The getValue method follows the international convention where the weekend comes at the end of the week, returning 1 for Monday, 2 for Tuesday, and so on. Sunday has value 7.

NOTE: You can *chain* method calls, like this:

```
int value = date.getDayOfWeek().getValue();
```

The first method call is applied to the date object, and it returns a DayOfWeek object. The getValue method is then invoked on the returned object.

You will find the complete program in the book's companion code. It was easy to solve the problem of printing a calendar because the designers of the LocalDate class provided us with a useful set of methods. In this chapter, you will learn how to implement methods for your own classes.

2.1.1 Accessor and Mutator Methods

Consider again the method call date.plusDays(1). There are two ways in which the designers of the LocalDate class could have implemented the plusDays method. They could make it change the state of the date object and return no result. Or they could leave date unchanged and return a newly constructed LocalDate object. As you can see, they chose to do the latter.

We say that a method is a *mutator* if it changes the object on which it was invoked. It is an *accessor* if it leaves the object unchanged. The plusDays method of the LocalDate class is an accessor.

In fact, *all* methods of the LocalDate class are accessors. This situation is increasingly common because mutation can be risky, particularly if two computations mutate an object simultaneously. Nowadays, most computers have multiple processing units, and safe concurrent access is a serious issue. One way to address this issue is to make objects *immutable* by providing only accessor methods.

Still, there are many situations where mutation is desirable. The add method of the ArrayList class is an example of a mutator. After calling add, the array list object is changed.

```
ArrayList<String> friends = new ArrayList<>();
    // friends is empty
friends.add("Peter");
    // friends has size 1
```

2.1.2 Object References

In some programming languages (such as C++), a variable can actually hold the object—that is, the bits that make up the object's state. In Java, that is not the case. A variable can only hold a *reference* to an object. The actual object is elsewhere, and the reference is some implementation-dependent way of locating the object (see Figure 2-1).

NOTE: References behave like pointers in C and C++, except that they are perfectly safe. In C and C++, you can modify pointers and use them to overwrite arbitrary memory locations. With a Java reference, you can only access a specific object.

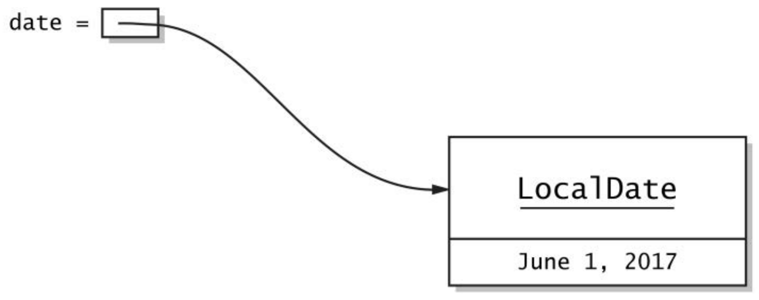

Figure 2-1 An object reference

When you assign a variable holding an object reference to another, you have two references to the same object.

```
ArrayList<String> people = friends;
    // Now people and friends refer to the same object
```

If you mutate the shared object, the mutation is observable through both references. Consider the call

```
people.add("Paul");
```

Now the array list `people` has size 2, and so does `friends` (see Figure 2-2). (Of course, it isn't technically true that `people` or `friends` "have" size 2. After all, `people` and `friends` are not objects. They are references to an object, namely an array list with size 2.)

Most of the time, this sharing of objects is efficient and convenient, but you have to be aware that it is possible to mutate a shared object through any of its references.

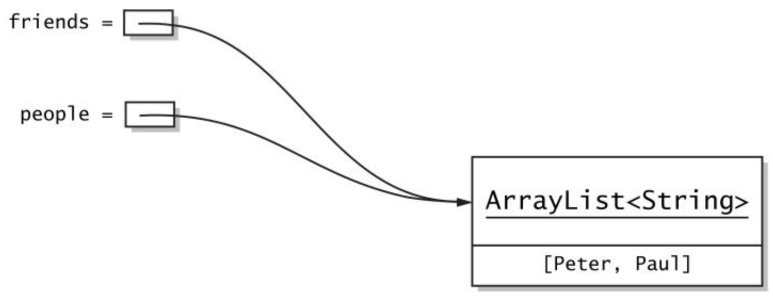

Figure 2-2 Two references to the same object

However, if a class has no mutator methods (such as `String` or `LocalDate`), you don't have to worry. Since nobody can change such an object, you can freely give out references to it.

It is possible for an object variable to refer to no object at all, by setting it to the special value `null`.

```
LocalDate date = null; // Now date doesn't refer to any object
```

This can be useful if you don't yet have an object for `date` to refer to, or if you want to indicate a special situation, such as an unknown date.

CAUTION: Null values can be dangerous when they are not expected. Invoking a method on `null` causes a `NullPointerException` (which should really have been called a `NullReferenceException`). For that reason, it is not recommended to use `null` for optional values. Use the `Optional` type instead (see Chapter 8).

Finally, have another look at the assignments

```
date = LocalDate.of(year, month, 1);
date = date.plusDays(1);
```

After the first assignment, `date` refers to the first day of the month. The call to `plusDays` yields a new `LocalDate` object, and after the second assignment, the `date` variable refers to the new object. What happens to the first one?

There is no reference to the first object, so it is no longer needed. Eventually, the *garbage collector* will recycle the memory and make it available for reuse. In Java, this process is completely automatic, and programmers never need to worry about deallocating memory.

2.2 Implementing Classes

Now let us turn to implementing our own classes. To show the various language rules, I use the classic example of an `Employee` class. An employee has a name and a salary. In this example, the name can't change, but ever so often an employee can get a well-deserved raise.

2.2.1 Instance Variables

From the description of employee objects, you can see that the state of such an object is described by two values: name and salary. In Java, you use *instance variables* to describe the state of an object. They are declared in a class like this:

```
public class Employee {
    private String name;
    private double salary;
    ...
}
```

That means that every instance of the `Employee` class has these two variables.

In Java, instance variables are usually declared as `private`. That means that only methods of the same class can access them. There are a couple of reasons why this protection is desirable: You control which parts of your program can modify the variables, and you can decide at any point to change the internal representation. For example, you might store the employees in a database and only leave the primary key in the object. As long as you reimplement the methods so they work the same as before, the users of your class won't care.

2.2.2 Method Headers

Now let's turn to implementing the methods of the `Employee` class. When you declare a method, you provide its name, the types and names of its parameters, and the return type, like this:

```
public void raiseSalary(double byPercent)
```

This method receives a parameter of type `double` and doesn't return any value, as indicated by the return type `void`.

The `getName` method has a different signature:

```
public String getName()
```

The method has no parameters and returns a `String`.

NOTE: Most methods are declared as `public`, which means anyone can call such a method. Sometimes, a helper method is declared as `private`, which restricts it to being used only in other methods of the same class. You should do that for methods that are not relevant to class users, particularly if they depend on implementation details. You can safely change or remove private methods if the implementation changes.

2.2.3 Method Bodies

Following the method header, you provide the body:

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

Use the `return` keyword if the method yields a value:

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

Place the method declarations inside the class declaration:

```
public class Employee {
    private String name;
    private double salary;

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

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

2.2.4 Instance Method Invocations

Consider this example of a method call:

```
fred.raiseSalary(5);
```

In this call, the argument `5` is used to initialize the parameter variable `byPercent`, equivalent to the assignment

```
double byPercent = 5;
```

Then the following actions occur:

```
double raise = fred.salary * byPercent / 100;
fred.salary += raise;
```

Note that the salary instance variable is applied to the instance on which the method is invoked.

Unlike the methods that you have seen at the end of the preceding chapter, a method such as raiseSalary operates on an instance of a class. Therefore, such a method is called an *instance method*. In Java, all methods that are not declared as static are instance methods.

As you can see, two values are passed to the raiseSalary method: a reference to the object on which the method is invoked, and the argument of the call. Technically, both of these are parameters of the method, but in Java, as in other object-oriented languages, the first one takes on a special role. It is sometimes called the *receiver* of the method call.

2.2.5 The this Reference

When a method is called on an object, this is set to that object. If you like, you can use the this reference in the implementation:

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

Some programmers prefer that style because it clearly distinguishes between local and instance variables—it is now obvious that raise is a local variable and salary is an instance variable.

It is very common to use the this reference when you don't want to come up with different names for parameter variables. For example,

```
public void setSalary(double salary) {
    this.salary = salary;
}
```

When an instance variable and a local variable have the same name, the unqualified name (such as salary) denotes the local variable, and this.salary is the instance variable.

NOTE: In some programming languages, instance variables are decorated in some way, for example _name and _salary. This is legal in Java but is not commonly done.

NOTE: If you like, you can even declare `this` as a parameter of a method (but not a constructor):

```
public void setSalary(Employee this, double salary) {
    this.salary = salary;
}
```

However, this syntax is very rarely used. It exists so that you can annotate the receiver of the method—see Chapter 11.

2.2.6 Call by Value

When you pass an object to a method, the method obtains a copy of the object reference. Through this reference, it can access or mutate the parameter object. For example,

```
public class EvilManager {
    private Random generator;
    ...
    public void giveRandomRaise(Employee e) {
        double percentage = 10 * generator.nextGaussian();
        e.raiseSalary(percentage);
    }
}
```

Consider the call

```
boss.giveRandomRaise(fred);
```

The reference `fred` is copied into the parameter variable `e` (see Figure 2-3). The method mutates the object that is shared by the two references.

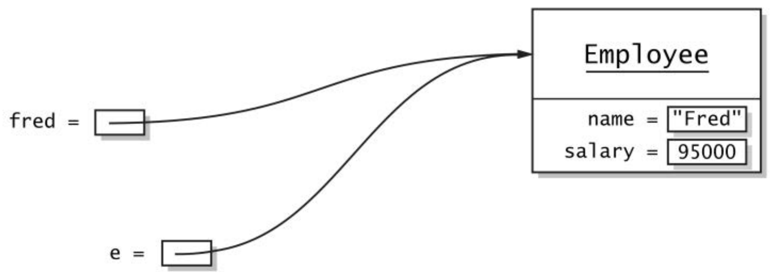

Figure 2-3 A parameter variable holding a copy of an object reference

In Java, you can never write a method that updates primitive type parameters. A method that tries to increase a `double` value won't work:

```
public void increaseRandomly(double x) { // Won't work
    double amount = x * generator.nextDouble();
    x += amount;
}
```

If you call

```
boss.increaseRandomly(sales);
```

then `sales` is copied into `x`. Then `x` is increased, but that doesn't change `sales`. The parameter variable then goes out of scope, and the increase leaves no useful effect.

For the same reason, it is not possible to write a method that changes an object reference to something different. For example, this method does not work as presumably intended:

```
public class EvilManager {
    ...
    public void replaceWithZombie(Employee e) {
        e = new Employee("", 0);
    }
}
```

In the call

```
boss.replaceWithZombie(fred);
```

the reference `fred` is copied into the variable `e` which is then set to a different reference. When the method exits, `e` goes out of scope. At no point was `fred` changed.

NOTE: Some people say that Java uses "call by reference" for objects. As you can see from the second example, that is not true. In a language that supports call by reference, a method can replace the contents of variables passed to it. In Java, all parameters—object references as well as primitive type values—are passed by value.

2.3 Object Construction

One step remains to complete the `Employee` class: We need to provide a constructor, as detailed in the following sections.

2.3.1 Implementing Constructors

Declaring a constructor is similar to declaring a method. However, the name of the constructor is the same as the class name, and there is *no return type*.

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

NOTE: This constructor is public. It can also be useful to have private constructors. For example, the `LocalDate` class has no public constructors. Instead, users of the class obtain objects from “factory methods” such as `now` and `of`. These methods call a private constructor.

CAUTION: If you accidentally specify a return type, such as

```
public void Employee(String name, double salary)
```

then you declare a method named `Employee`, not a constructor!

A constructor executes when you use the `new` operator. For example, the expression

```
new Employee("James Bond", 500000)
```

allocates an object of the `Employee` class and invokes the constructor body, which sets the instance variables to the arguments supplied in the constructor.

The `new` operator returns a reference to the constructed object. You will normally want to save that reference in a variable:

```
Employee james = new Employee("James Bond", 500000);
```

or pass it to a method:

```
ArrayList<Employee> staff = new ArrayList<>();
staff.add(new Employee("James Bond", 500000));
```

2.3.2 Overloading

You can supply more than one version of a constructor. For example, if you want to make it easy to model nameless worker bees, supply a second constructor that only accepts a salary.

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

Now the `Employee` class has two constructors. Which one is called depends on the arguments.

```
Employee james = new Employee("James Bond", 500000);
    // calls Employee(String, double) constructor
Employee anonymous = new Employee(40000);
    // calls Employee(double) constructor
```

In this case, we say that the constructor is *overloaded*.

NOTE: A method is overloaded if there are multiple versions with the same name but different parameters. For example, there are overloaded versions of the `println` method with parameters `int`, `double`, `String`, and so on. Since you have no choice how to name a constructor, it is common to overload constructors.

2.3.3 Calling One Constructor from Another

When there are multiple constructors, they usually have some work in common, and it is best not to duplicate that code. It is often possible to put that common initialization into one constructor.

You can call one constructor from another, but only as the *first statement* of the constructor body. Somewhat surprisingly, you don't use the name of the constructor for the call but the keyword `this`:

```
public Employee(double salary) {
    this("", salary); // Calls Employee(String, double)
    // Other statements can follow
}
```

NOTE: Here, `this` is *not* a reference to the object that is being constructed. Instead, it is a special syntax that is only used for invoking another constructor of the same class.

2.3.4 Default Initialization

If you don't set an instance variable explicitly in a constructor, it is automatically set to a default value: numbers to `0`, `boolean` values to `false`, and object references to `null`.

For example, you could supply a constructor for unpaid interns.

```
public Employee(String name) {
    // salary automatically set to zero
    this.name = name;
}
```

NOTE: In this regard, instance variables are very different from local variables. Recall that you must always explicitly initialize local variables.

For numbers, the initialization with zero is often convenient. But for object references, it is a common source of errors. Suppose we didn't set the `name` variable to the empty string in the `Employee(double)` constructor:

```
public Employee(double salary) {
    // name automatically set to null
    this.salary = salary;
}
```

If anyone called the `getName` method, they would get a null reference that they probably don't expect. A condition such as

```
if (e.getName().equals("James Bond"))
```

would then cause a null pointer exception.

2.3.5 Instance Variable Initialization

You can specify an initial value for any instance variables, like this:

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

This initialization occurs after the object has been allocated and before a constructor runs. Therefore, the initial value is present in all constructors. Of course, some of them may choose to overwrite it.

In addition to initializing an instance variable when you declare it, you can include arbitrary *initialization blocks* in the class declaration.

```
public class Employee() {
    private String name = "";
    private int id;
    private double salary;

    { // An initialization block
        Random generator = new Random();
        id = 1 + generator.nextInt(1_000_000);
    }

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

NOTE: This is not a commonly used feature. Most programmers place lengthy initialization code into a helper method and invoke that method from the constructors.

Instance variable initializations and initialization blocks are executed in the order in which they appear in the class declaration, and before the body of the constructor.

2.3.6 Final Instance Variables

You can declare an instance variable as final. Such a variable must be initialized by the end of every constructor. Afterwards, the variable may not be modified again. For example, the `name` variable of the `Employee` class may be declared as `final` because it never changes after the object is constructed—there is no `setName` method.

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

NOTE: When used with a reference to a mutable object, the `final` modifier merely states that the reference will never change. It is perfectly legal to mutate the object.

```
public class Person {
    private final ArrayList<Person> friends = new ArrayList<>();
        // OK to add elements to this array list
    ...
}
```

Methods may mutate the array list to which `friends` refers, but they can never replace it with another. In particular, it can never become `null`.

2.3.7 The Constructor with No Arguments

Many classes contain a constructor with no arguments that creates an object whose state is set to an appropriate default. For example, here is a constructor with no arguments for the `Employee` class:

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

Just like an indigent defendant is provided with a public defender, a class with no constructors is automatically given a constructor with no arguments that does nothing at all. All instance variables stay at their default values (zero, `false`, or `null`) unless they have been explicitly initialized.

Thus, every class has at least one constructor.

NOTE: If a class already has a constructor, it does *not* automatically get another constructor with no arguments. If you supply a constructor and also want a no-argument constructor, you have to write it yourself.

NOTE: In the preceding sections, you saw what happens when an object is constructed. In some programming languages, notably C++, it is common to specify what happens when an object is destroyed. Java does have a mechanism for "finalizing" an object when it is reclaimed by the garbage collector. But this happens at unpredictable times, so you should not use it. However, as you will see in Chapter 5, there is a mechanism for closing resources such as files.

2.4 Static Variables and Methods

In all sample programs that you have seen, the main method is tagged with the `static` modifier. In the following sections, you will learn what this modifier means.

2.4.1 Static Variables

If you declare a variable in a class as `static`, then there is only one such variable per class. In contrast, each object has its own copy of an instance variable. For example, suppose we want to give each employee a distinct ID number. Then we can share the last ID that was given out.

```
public class Employee {
    private static int lastId = 0;
    private int id;
    ...
    public Employee() {
        lastId++;
        id = lastId;
    }
}
```

Every `Employee` object has its own instance variable `id`, but there is only one `lastId` variable that belongs to the class, not to any particular instance of the class.

When a new `Employee` object is constructed, the shared `lastId` variable is incremented and the `id` instance variable is set to that value. Thus, every employee gets a distinct `id` value.

CAUTION: This code will not work if `Employee` objects can be constructed concurrently in multiple threads. Chapter 10 shows how to remedy that problem.

NOTE: You may wonder why a variable that belongs to the class, and not to individual instances, is named "static." The term is a meaningless holdover from C++ which borrowed the keyword from an unrelated use in the C language instead of coming up with something more appropriate. A more descriptive term is "class variable."

2.4.2 Static Constants

Mutable static variables are rare, but static constants (that is, `static final` variables) are quite common. For example, the `Math` class declares a static constant:

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

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

Without the `static` keyword, `PI` would have been an instance variable of the `Math` class. That is, you would need an object of the class to access `PI`, and every `Math` object would have its own copy of `PI`.

Here is an example of a static `final` variable that is an object, not a number. It is both wasteful and insecure to construct a new random number generator each time you want a random number. You are better off sharing a single generator among all instances of a class.

```
public class Employee {
    private static final Random generator = new Random();
    private int id;
    ...
    public Employee() {
        id = 1 + generator.nextInt(1_000_000);
    }
}
```

Another example of a static constant is `System.out`. It is declared in the `System` class like this:

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

CAUTION: Even though `out` is declared as `final` in the `System` class, there is a method `setOut` that sets `System.out` to a different stream. This method is a "native" method, not implemented in Java, which can bypass the access control mechanisms of the Java language. This is a very unusual situation from the early days of Java, and not something you are likely to encounter elsewhere.

2.4.3 Static Initialization Blocks

In the preceding sections, static variables were initialized as they were declared. Sometimes, you need to do additional initialization work. You can put it into a *static initialization block*.

```
public class CreditCardForm {
    private static final ArrayList<Integer> expirationYear = new ArrayList<>();
    static {
        // Add the next twenty years to the array list
        int year = LocalDate.now().getYear();
        for (int i = year; i <= year + 20; i++) {
            expirationYear.add(i);
        }
    }
    ...
}
```

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

2.4.4 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.

As you have already seen in Chapter 1, a static method is declared with the `static` modifier:

```
public class Math {
    public static double pow(double base, double exponent) {
        ...
    }
}
```

Why not make `pow` into an instance method? It can't be an instance method of `double` since, in Java, primitive types are not classes. One could make it an instance method of the `Math` class, but then you would need to construct a `Math` object in order to call it.

Another common reason for static methods is to provide added functionality to classes that you don't own. For example, wouldn't it be nice to have a method that yields a random integer in a given range? You can't add a method to the `Random` class in the standard library. But you can provide a static method:

```
public class RandomNumbers {
    public static int nextInt(Random generator, int low, int high) {
        return low + generator.nextInt(high - low + 1);
    }
}
```

Call this method as

```
int dieToss = RandomNumbers.nextInt(gen, 1, 6);
```

NOTE: It is legal to invoke a static method on an object. For example, instead of calling `LocalDate.now()` to get today's date, you can call `date.now()` on an object `date` of the `LocalDate` class. But that does not make a lot of sense. The `now` method doesn't look at the `date` object to compute the result. Most Java programmers would consider this poor style.

Since static methods don't operate on objects, you cannot access instance variables from a static method. However, static methods can access the static variables in their class. For example, in the `RandomNumbers.nextInt` method, we can make the random number generator into a static variable:

```
public class RandomNumbers {
    private static Random generator = new Random();
    public static int nextInt(int low, int high) {
        return low + generator.nextInt(high - low + 1);
            // OK to access the static generator variable
    }
}
```

2.4.5 Factory Methods

A common use for static methods is a *factory method*, a static method that returns new instances of the class. For example, 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 not use a constructor instead? The only way to distinguish two constructors is by their parameter types. You cannot have two constructors with no arguments.

Moreover, a constructor `new NumberFormat(...)` yields a `NumberFormat`. A factory method can return an object of a subclass. In fact, these factory methods return instances of the `DecimalFormat` class. (See Chapter 4 for more information about subclasses.)

A factory method can also return a shared object, instead of unnecessarily constructing new ones. For example, the call `Collections.emptyList()` returns a shared immutable empty list.

2.5 Packages

In Java, you place related classes into a package. Packages are convenient for organizing your work and for separating it from code libraries provided by others. As you have seen, the standard Java library is distributed over a number of packages, including `java.lang`, `java.util`, `java.math`, and so on.

One reason for using packages is to guarantee the uniqueness of class names. Suppose two programmers come up with the bright idea of supplying an `Element` class. (In fact, at least five developers had that bright idea in the Java

API alone.) As long as all of them place their classes into different packages, there is no conflict.

In the following sections, you will learn how to work with packages.

2.5.1 Package Declarations

A package name is a dot-separated list of identifiers such as `java.util.regex`.

To guarantee unique package names, it is a good idea to use an Internet domain name (which is known to be unique) written in reverse. For example, I own the domain name `horstmann.com`. For my projects, I use package names such as `com.horstmann.corejava`. A major exception to this rule is the standard Java library whose package names start with `java` or `javax`.

NOTE: In Java, packages do not nest. For example, the packages `java.util` and `java.util.regex` have nothing to do with each other. Each is its own independent collection of classes.

To place a class in a package, you add a `package` statement as the first statement of the source file:

```
package com.horstmann.corejava;

public class Employee {
    ...
}
```

Now the `Employee` class is in the `com.horstmann.corejava` package, and its *fully qualified name* is `com.horstmann.corejava.Employee`.

There is also a *default package* with no name that you can use for simple programs. To add a class to the default package, don't provide a `package` statement. However, the use of the default package is not recommended.

When class files are read from a file system, the path name needs to match the package name. For example, the file `Employee.class` must be in a subdirectory `com/horstmann/corejava`.

If you arrange the source files in the same way and compile from the directory that contains the initial package names, then the class files are automatically put in the correct place. Suppose the `EmployeeDemo` class makes use of `Employee` objects, and you compile it as

```
javac com/horstmann/corejava/EmployeeDemo.java
```

The compiler generates class files `com/horstmann/corejava/EmployeeDemo.class` and `com/horstmann/corejava/Employee.class`. You run the program by specifying the fully qualified class name:

```
java com.horstmann.corejava.EmployeeDemo
```

CAUTION: If a source file is not in a subdirectory that matches its package name, the `javac` compiler will not complain and will generate a class file, but you will need to put the class file in the right place. This can be quite confusing—see Exercise 12.

TIP: It is a good idea to run `javac` with the `-d` option. Then the class files are generated in a separate directory, without cluttering up the source tree, and they have the correct subdirectory structure.

2.5.2 The `jar` Command

Instead of storing class files in the file system, you can place them into one or more archive files called JAR files. You can make such an archive with the `jar` utility that is a part of the JDK. Its command-line options are similar to those of the Unix `tar` program.

```
jar --create --verbose --file library.jar com/mycompany/*.class
```

or, with short options,

```
jar -c -v -f library.jar com/mycompany/*.class
```

or, with `tar`-style options,

```
jar cvf library.jar com/mycompany/*.class
```

JAR files are commonly used to package libraries.

TIP: You can use JAR files to package a program, not just a library. Generate the JAR file with

```
jar -c -f program.jar -e com.mycompany.MainClass com/mycompany/*.class
```

Then run the program as

```
java -jar program.jar
```

CAUTION: The options of commands in the Java development kit have traditionally used single dashes followed by multi-letter option names, such as `java -jar`. The exception was the `jar` command, which followed the classic option format of the `tar` command without dashes. Java 9 is moving towards the more common option format where multi-letter option names are preceded by double dashes, such as `--create`, with single-letter shortcuts for common options, such as `-c`.

This has created a muddle that will hopefully get cleaned up over time. Right now, `java -jar` works as always, but `java --jar` doesn't. You can combine some single-letter options but not others. For example, `jar -cvf` *filename* works, but `jar -cv -f` *filename* doesn't. Long argument options can follow a space or `=`, and short argument options can follow with or without a space. However, this is not fully implemented: `jar -c --file=`*filename* works, but `jar -c -f`*filename* doesn't.

2.5.3 The Class Path

When you use library JAR files in a project, you need to tell the compiler and the virtual machine where these files are by specifying the *class path*. A class path can contain

- Directories containing class files (in subdirectories that match their package names)
- JAR files
- Directories containing JAR files

The `javac` and `java` programs have an option `-cp` (with a verbose version `--class-path` or, for backwards compatibility, `-classpath`). For example,

```
java -cp .:../libs/lib1.jar:../libs/lib2.jar com.mycompany.MainClass
```

This class path has three elements: the current directory (`.`) and two JAR files in the directory `../libs`.

NOTE: In Windows, use semicolons instead of colons to separate the path elements:

```
java -cp .;..\libs\lib1.jar;..\libs\lib2.jar com.mycompany.MainClass
```

If you have many JAR files, put them all in a directory and use a wildcard to include them all:

```
java -cp .:../libs/\* com.mycompany.MainClass
```

NOTE: In Unix, the `*` must be escaped to prevent shell expansion.

CAUTION: The `javac` compiler always looks for files in the current directory, but the `java` program 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 won't run.

CAUTION: The wildcard option for the class path is convenient, but it only works reliably if the JAR files are well structured. It is possible (but not a good idea) to have two versions of the same class in different JAR files. In such a situation, the first encountered class wins. The wildcard syntax does not guarantee the ordering in which the JAR files are processed, and you should not use it if you require a particular ordering of the JAR files. (Such "JAR file hell" is a problem that the Java platform module system aims to prevent—see Chapter 15.)

Using the `-cp` option is the preferred approach for setting the class path. An alternate approach is the `CLASSPATH` environment variable. The details depend on your shell. If you use `bash`, use a command such as

```
export CLASSPATH=.:/home/username/project/libs/\*
```

In Windows, it is

```
SET CLASSPATH=.;C:\Users\username\project\libs\*
```

CAUTION: You can set the `CLASSPATH` environment variable globally (for example, in `.bashrc` or the Windows control panel). However, many programmers have regretted this when they forgot the global setting and were surprised that their classes were not found.

NOTE: As you will see in Chapter 15, you can group packages together into *modules*. Modules provide strong encapsulation, hiding all packages except those that you make visible. You will see in Chapter 15 how to use the *module path* to specify the locations of the modules that your programs use.

2.5.4 Package Access

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 declares 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.

Package access is useful for utility classes and methods that are needed by the methods of a package but are not of interest to the users of the package. Another common use case is for testing. You can place test classes in the same package, and then they can access internals of the classes being tested.

NOTE: A source file can contain multiple classes, but at most one of them can be declared `public`. If a source file has a public class, its name must match the class name.

For variables, it is unfortunate that package access is the default. It is a common mistake to forget the `private` modifier and accidentally make an instance variable accessible to the entire package. Here is an example from the `Window` class in the `java.awt` package:

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

Since the `warningString` variable is not private, the methods of all classes in the `java.awt` package can access it. Actually, no method other than those of the `Window` class itself does that, so it seems likely that the programmer simply forgot the `private` modifier.

This can be a security issue because packages are open ended. Any class can add itself to a package by providing the appropriate `package` statement.

If you are concerned about this openness of packages, you are not alone. A remedy is to place your package into a module—see Chapter 15. When a package is in a module, it is not possible to add classes to the package. All packages in the Java library are grouped into modules, so you cannot access the `Window.warningString` variable simply by crafting a class in the `java.awt` package.

2.5.5 Importing Classes

The `import` statement lets you use classes without the fully qualified name. For example, when you use

```
import java.util.Random;
```

then you can write `Random` instead of `java.util.Random` in your code.

NOTE: Import declarations are a convenience, not a necessity. You could drop all import declarations and use fully qualified class names everywhere.

```
java.util.Random generator = new java.util.Random();
```

Place `import` statements above the first class declaration in the source file, but below the `package` statement.

You can import all classes from a package with a wildcard:

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

The wildcard can only import classes, not packages. You cannot use `import java.*;` to obtain all packages whose name starts with `java`.

When you import multiple packages, it is possible to have a name conflict. For example, the packages `java.util` and `java.sql` both contain a `Date` class. Suppose you import both packages:

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

If your program doesn't use the `Date` class, this is not a problem. But if you refer to `Date`, without the package name, the compiler complains.

In that case, you can import the specific class that you want:

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

If you really need both classes, you must use the fully qualified name for at least one of them.

NOTE: The `import` statement is a convenience for programmers. Inside class files, all class names are fully qualified.

NOTE: The `import` statement is very different from the `#include` directive in C and C++. That directive includes header files for compilation. Imports do not cause files to be recompiled. They just shorten names, like the C++ `using` statement.

2.5.6 Static Imports

A form of the `import` statement permits the importing of static methods and variables. For example, if you add the directive

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

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

```
r = sqrt(pow(x, 2) + pow(y, 2)); // i.e., Math.sqrt, Math.pow
```

You can also import a specific static method or variable:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

NOTE: As you will see in Chapters 3 and 8, it is common to use static import declarations with `java.util.Comparator` and `java.util.stream.Collectors`, which provide a large number of static methods.

2.6 Nested Classes

In the preceding section, you have seen how to organize classes into packages. Alternatively, you can place a class inside another class. Such a class is called a *nested class*. This can be useful to restrict visibility, or to avoid cluttering up a package with generic names such as `Element`, `Node`, or `Item`. Java has two kinds of nested classes, with somewhat different behavior. Let us examine both in the following sections.

2.6.1 Static Nested Classes

Consider an `Invoice` class that bills for items, each of which has a description, quantity, and unit price. We can make `Item` into a nested class:

```
public class Invoice {
    private static class Item { // Item is nested inside Invoice
        String description;
        int quantity;
        double unitPrice;

        double price() { return quantity * unitPrice; }
    }

    private ArrayList<Item> items = new ArrayList<>();
    ...
}
```

It won't be clear until the next section why this inner class is declared `static`. For now, just accept it.

There is nothing special about the `Item` class, except for access control. The class is private in `Invoice`, so only `Invoice` methods can access it. For that reason, I did not bother making the instance variables of the inner class private.

Here is an example of a method that constructs an object of the inner class:

```
public class Invoice {
    ...
    public void addItem(String description, int quantity, double unitPrice) {
        Item newItem = new Item();
        newItem.description = description;
        newItem.quantity = quantity;
        newItem.unitPrice = unitPrice;
        items.add(newItem);
    }
}
```

A class can make a nested class public. In that case, one would want to use the usual encapsulation mechanism.

```
public class Invoice {
    public static class Item { // A public nested class
        private String description;
        private int quantity;
        private double unitPrice;

        public Item(String description, int quantity, double unitPrice) {
            this.description = description;
            this.quantity = quantity;
            this.unitPrice = unitPrice;
        }
        public double price() { return quantity * unitPrice; }
        ...
    }

    private ArrayList<Item> items = new ArrayList<>();

    public void add(Item item) { items.add(item); }
    ...
}
```

Now anyone can construct `Item` objects by using the qualified name `Invoice.Item`:

```
Invoice.Item newItem = new Invoice.Item("Blackwell Toaster", 2, 19.95);
myInvoice.add(newItem);
```

There is essentially no difference between this `Invoice.Item` class and a class `InvoiceItem` declared outside any other class. Nesting the class just makes it obvious that the `Item` class represents items in an invoice.

2.6.2 Inner Classes

In the preceding section, you saw a nested class that was declared as `static`. In this section, you will see what happens if you drop the `static` modifier. Such classes are called *inner classes*.

Consider a social network in which each member has friends that are also members.

```
public class Network {
    public class Member { // Member is an inner class of Network
        private String name;
        private ArrayList<Member> friends;

        public Member(String name) {
            this.name = name;
            friends = new ArrayList<>();
        }
        ...
    }

    private ArrayList<Member> members = new ArrayList<>();
    ...
}
```

With the `static` modifier dropped, there is an essential difference. A `Member` object knows to which network it belongs. Let's see how this works.

First, here is a method to add a member to the network:

```
public class Network {
    ...
    public Member enroll(String name) {
        Member newMember = new Member(name);
        members.add(newMember);
        return newMember;
    }
}
```

So far, nothing much seems to be happening. We can add a member and get a reference to it.

```
Network myFace = new Network();
Network.Member fred = myFace.enroll("Fred");
```

Now let's assume Fred feels this isn't the hottest social network anymore, so he wants to deactivate his membership.

```
fred.deactivate();
```

Here is the implementation of the `deactivate` method:

```
public class Network {
    public class Member {
        ...
        public void deactivate() {
            members.remove(this);
        }
    }

    private ArrayList<Member> members;
    ...
}
```

As you can see, a method of an inner class can access instance variables of its outer class. In this case, they are the instance variables of the outer class object that created it, the unpopular `myFace` network.

This is what makes an inner class different from a static nested class. Each inner class object has a reference to an object of the enclosing class. For example, the method

```
members.remove(this);
```

actually means

```
outer.members.remove(this);
```

where I use *outer* to denote the hidden reference to the enclosing class.

A static nested class does not have such a reference (just like a static method does not have the `this` reference). Use a static nested class when the instances of the nested class don't need to know to which instance of the enclosing class they belong. Use an inner class only if this information is important.

An inner class can also invoke methods of the outer class through its outer class instance. For example, suppose the outer class had a method to unenroll a member. Then the `deactivate` method can call it:

```
public class Network {
    public class Member {
        ...
        public void deactivate() {
            unenroll(this);
        }
    }

    private ArrayList<Member> members;

    public Member enroll(String name) { ... }
    public void unenroll(Member m) { ... }
    ...
}
```

In this case,

```
unenroll(this);
```

actually means

```
outer.unenroll(this);
```

2.6.3 Special Syntax Rules for Inner Classes

In the preceding section, I explained the outer class reference of an inner class object by calling it *outer*. The actual syntax for the outer reference is a bit more complex. The expression

```
OuterClass.this
```

denotes the outer class reference. For example, you can write the `deactivate` method of the `Member` inner class as

```
public void deactivate() {
    Network.this.members.remove(this);
}
```

In this case, the `Network.this` syntax was not necessary. Simply referring to `members` implicitly uses the outer class reference. But sometimes, you need the outer class reference explicitly. Here is a method to check whether a `Member` object belongs to a particular network:

```
public class Network {
    public class Member {
        ...
        public boolean belongsTo(Network n) {
            return Network.this == n;
        }
    }
}
```

When you construct an inner class object, it remembers the enclosing class object that constructed it. In the preceding section, a new member was created by this method:

```
public class Network {
    ...
    Member enroll(String name) {
        Member newMember = new Member(name);
        ...
    }
}
```

That is a shortcut for

```
Member newMember = this.new Member(name);
```

You can invoke an inner class constructor on any instance of an outer class:

```
Network.Member wilma = myFace.new Member("Wilma");
```

NOTE: Inner classes cannot declare static members other than compile-time constants. There would be an ambiguity about the meaning of "static." Does it mean there is only one instance in the virtual machine? Or only one instance per outer object? The language designers decided not to tackle this issue.

NOTE: By historical accident, inner classes were added to the Java language at a time when the virtual machine specification was considered complete, so they are translated into regular classes with a hidden instance variable referring to the enclosing instance. Exercise 14 invites you to explore this translation.

NOTE: *Local classes* are another variant of inner classes that we will discuss in Chapter 3.

2.7 Documentation Comments

The JDK contains a very useful tool, called `javadoc`, that generates HTML documentation from your source files. In fact, the online API documentation that we described in Chapter 1 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 approach because it lets you keep your code and documentation in one place. In the bad old days, programmers often put their documentation into a separate file, and it was just a question of time for the code and the comments to diverge. When documentation comments are in the same file as the source code, it is an easy matter to update both and run `javadoc` again.

2.7.1 Comment Insertion

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

- Public classes and interfaces
- Public and protected constructors and methods

- Public and protected variables
- Packages and modules

Interfaces are introduced in Chapter 3 and protected features in Chapter 4.

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 `/**` and ends with `*/`.

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 boldface, and even `<img ...>` to include an image. You should, however, stay away from headings `<h`*n*`>` or rules `<hr>` because they can interfere with the formatting of the documentation.

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 specify the `doc-files` directory in your link, for example `<img src="doc-files/uml.png" alt="UML diagram"/>`.

2.7.2 Class Comments

The class comment must be placed directly before the class declaration. You may want to document the author and version of a class with the `@author` and `@version` tags. There can be multiple authors.

Here is an example of a class comment:

```
/**
 * An <code>Invoice</code> object represents an invoice with
 * line items for each part of the order.
 * @author Fred Flintstone
 * @author Barney Rubble
 * @version 1.1
 */
```

```
public class Invoice {
    ...
}
```

NOTE: There is no need to put a `*` in front of every line. However, most IDEs supply the asterisks automatically, and some even rearrange them when the line breaks change.

2.7.3 Method Comments

Place each method comment immediately before its method. Document the following features:

- Each parameter, with a comment `@param` *variable description*
- The return value, if not `void`: `@return` *description*
- Any thrown exceptions (see Chapter 5): `@throws` *ExceptionClass description*

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 means 10%)
 * @return the amount of the raise
*/
public double raiseSalary(double byPercent) {
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

2.7.4 Variable Comments

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

```
/**
 * The number of days per year on Earth (excepting leap years)
 */
public static final int DAYS_PER_YEAR = 365;
```

2.7.5 General Comments

In all documentation comments, you can use the `@since` tag to describe the version in which this feature became available:

```
@since version 1.7.1
```

The `@deprecated` 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
```

NOTE: There is also a `@Deprecated` annotation that compilers use to issue warnings when deprecated items are used—see Chapter 11. The annotation does not have a mechanism for suggesting a replacement, so you should supply both the annotation and the Javadoc comment for deprecated items.

2.7.6 Links

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

The tag `@see` *reference* 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 its 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 a separator between packages, subpackages, classes, inner classes, and their methods and variables. But the `javadoc` utility isn't quite as clever, so you have to help it along.

If the `@see` tag is followed by a `<` character, you're specifying a hyperlink. You can link to any URL you like. For example: `@see <a href="http://en.wikipedia.org/wiki/Leap_year">Leap years</a>`.

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

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

```
@see "Core Java for the Impatient"
```

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. Insert a 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.

2.7.7 Package, Module, and Overview Comments

The class, method, and variable comments are placed directly into the Java source files, delimited by `/** ... */`. However, to generate package comments, you need to add a separate file in each package directory.

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.

To document a module, place your comments into `module-info.java`. You can include the `@moduleGraph` directive to include a module dependency graph. (See Chapter 15 about modules and the `module-info.java` file.)

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.

2.7.8 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 package1 package2 ...
   ```

If you omit the `-d` *docDirectory* option, the HTML files are extracted to the current directory. That can get messy, so I 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 run the command

```
javadoc -link http://docs.oracle.com/javase/9/docs/api *.java
```

all standard library classes are automatically linked to the documentation on the Oracle website.

If you use the `-linksource` option, each source file is converted to HTML, and each class and method name turns into a hyperlink to the source.

Exercises

1. Change the calendar printing program so it starts the week on a Sunday. Also make it print a newline at the end (but only one).
2. Consider the `nextInt` method of the `Scanner` class. Is it an accessor or mutator? Why? What about the `nextInt` method of the `Random` class?
3. Can you ever have a mutator method return something other than `void`? Can you ever have an accessor method return `void`? Give examples when possible.
4. Why can't you implement a Java method that swaps the contents of two `int` variables? Instead, write a method that swaps the contents of two `IntHolder` objects. (Look up this rather obscure class in the API documentation.) Can you swap the contents of two `Integer` objects?
5. Implement an immutable class `Point` that describes a point in the plane. Provide a constructor to set it to a specific point, a no-arg constructor to set it to the origin, and methods `getX`, `getY`, `translate`, and `scale`. The `translate` method moves the point by a given amount in *x*- and *y*-direction. The `scale` method scales both coordinates by a given factor. Implement these methods so that they return new points with the results. For example,

   ```
   Point p = new Point(3, 4).translate(1, 3).scale(0.5);
   ```

   should set `p` to a point with coordinates (2, 3.5).
6. Repeat the preceding exercise, but now make `translate` and `scale` into mutators.

7. Add `javadoc` comments to both versions of the `Point` class from the preceding exercises.
8. In the preceding exercises, providing the constructors and getter methods of the `Point` class was rather repetitive. Most IDEs have shortcuts for writing the boilerplate code. What does your IDE offer?
9. Implement a class `Car` that models a car traveling along the *x*-axis, consuming gas as it moves. Provide methods to drive by a given number of miles, to add a given number of gallons to the gas tank, and to get the current distance from the origin and fuel level. Specify the fuel efficiency (in miles/gallons) in the constructor. Should this be an immutable class? Why or why not?
10. In the `RandomNumbers` class, provide two static methods `randomElement` that get a random element from an array or array list of integers. (Return zero if the array or array list is empty.) Why couldn't you make these methods into instance methods of `int[]` or `ArrayList<Integer>`?
11. Rewrite the `Cal` class to use static imports for the `System` and `LocalDate` classes.
12. Make a file `HelloWorld.java` that declares a class `HelloWorld` in a package `ch01.sec01`. Put it into some directory, but *not* in a `ch01/sec01` subdirectory. From that directory, run `javac HelloWorld.java`. Do you get a class file? Where? Then run `java HelloWorld`. What happens? Why? (Hint: Run `javap HelloWorld` and study the warning message.) Finally, try `javac -d . HelloWorld.java`. Why is that better?
13. Download the JAR file for OpenCSV from `http://opencsv.sourceforge.net`. Write a class with a `main` method that reads a CSV file of your choice and prints some of the content. There is sample code on the OpenCSV website. You haven't yet learned to deal with exceptions. Just use the following header for the `main` method:

    ```
    public static void main(String[] args) throws Exception
    ```

    The point of this exercise is not to do anything useful with CSV files, but to practice using a library that is delivered as a JAR file.
14. Compile the `Network` class. Note that the inner class file is named `Network$Member.class`. Use the `javap` program to spy on the generated code. The command

    ```
    javap -private Classname
    ```

displays the methods and instance variables. Where do you see the reference to the enclosing class? (In Linux/Mac OS, you need to put a \ before the `$` symbol when running `javap`.)

15. Fully implement the `Invoice` class in Section 2.6.1, "Static Nested Classes" (page 85). Provide a method that prints the invoice and a demo program that constructs and prints a sample invoice.
16. Implement a class `Queue`, an unbounded queue of strings. Provide methods `add`, adding at the tail, and `remove`, removing at the head of the queue. Store elements as a linked list of nodes. Make `Node` a nested class. Should it be static or not?
17. Provide an *iterator*—an object that yields the elements of the queue in turn—for the queue of the preceding class. Make `Iterator` a nested class with methods `next` and `hasNext`. Provide a method `iterator()` of the `Queue` class that yields a `Queue.Iterator`. Should `Iterator` be static or not?