3.2 Instance Variables, *set* Methods and *get* Methods

In this section, you'll create two classes—`Account` (Fig. 3.1) and `AccountTest` (Fig. 3.2). Class `AccountTest` is an *application class* in which the `main` method will create and use an `Account` object to demonstrate class `Account`'s capabilities.

3.2.1 Account Class with an Instance Variable, and *set* and *get* Methods

Different accounts typically have different names. For this reason, class `Account` (Fig. 3.1) contains a `name` *instance variable*. A class's instance variables maintain data for each object (that is, each instance) of the class. Later in the chapter we'll add an instance variable named `balance` so we can keep track of how much money is in the account. Class `Account` contains two methods—method `setName` stores a name in an `Account` object and method `getName` obtains a name from an `Account` object.

```
// Fig. 3.1: Account.java
```

```
 2   // Account class that contains a name instance v
 3   // and methods to set and get its value.
 4
 5   public class Account {
 6      private String name; // instance variable
 7
 8      // method to set the name in the object
 9      public void setName(String name) {
10         this.name = name; // store the name
11      }
12
13      // method to retrieve the name from the objec
14      public String getName() {
15         return name; // return value of name to ca
16      }
17   }
```

Fig. 3.1

`Account` class that contains a `name` instance variable and methods to *set* and *get* its value.

Class Declaration

The *class declaration* begins in line 5:

```
public class Account {
```

The keyword `public` (which Chapter 8 explains in detail) is an **access modifier**. For now, we'll simply declare every class `public`. Each `public` class declaration must be stored in a file having the *same* name as the class and ending with the `.java` filename extension; otherwise, a compilation error will occur. Thus, `public` classes `Account` and `AccountTest` (Fig. 3.2) *must* be declared in the *separate* files `Account.java` and `AccountTest.java`, respectively.

Every class declaration contains the keyword `class` followed immediately by the class's name—in this case, `Account`. Every class's body is enclosed in a pair of left and right braces as in lines 5 and 17 of Fig. 3.1.

Identifiers and Camel-Case Naming

Recall from Chapter 2 that class names, method names and variable names are all *identifiers* and by convention all use the *camel-case* naming scheme. Also by convention, class names begin with an initial *uppercase* letter, and method names and variable names begin with an initial *lowercase* letter.

Instance Variable name

Recall from Section 1.5 that an object has attributes, implemented as instance variables and carried with it throughout its lifetime. Instance variables exist before methods are called on an object, while the methods are executing and after the methods complete execution. Each object (instance) of the class has its *own* copy of the class’s instance variables. A class normally contains one or more methods that manipulate the instance variables belonging to particular objects of the class.

Instance variables are declared *inside* a class declaration but *outside* the bodies of the class’s methods. Line 6

```
private String name; // instance variable
```

declares instance variable `name` of type `String` *outside* the bodies of methods `setName` (lines 9–11) and `getName` (lines 14–16). `String` variables can hold character string values such as `"Jane Green"`. If there are many `Account` objects, each has its own `name`. Because `name` is an instance variable, it can be manipulated by each of the class’s methods.

Good Programming Practice 3.1

We prefer to list a class’s instance variables first in the class’s body, so that you see the names and types of the variables before they’re used in the class’s methods. You can list the class’s instance variables anywhere in the class outside its method declarations, but scattering the instance variables can *lead to hard-to-read code.*

Access Modifiers `public` and `private`

Most instance-variable declarations are preceded with the keyword `private` (as in line 6). Like `public`, `private` is an *access modifier*. Variables or methods declared with `private` are accessible only to methods of the class in which they're declared. So, the variable `name` can be used only in each `Account` object's methods (`setName` and `getName` in this case). You'll soon see that this presents powerful software engineering opportunities.

`setName` Method of Class `Account`

Let's walk through the code of `setName`'s method declaration (lines 9–11):

```
public void setName(String name) {
   this.name = name; // store the name
}
```

We refer to the first line of each method declaration (line 9 in this case) as the *method header*. The method's **return type** (which appears before the method name) specifies the type of data the method returns to its *caller* after performing its task. As you’ll soon see, the statement in line 19 of `main` (Fig. 3.2) *calls* method `setName`, so `main` is `setName`’s *caller* in this example. The return type `void` (line 9 in Fig. 3.1) indicates that `setName` will perform a task but will *not* return (i.e., give back) any information to its caller. In Chapter 2, you used methods that return information—for example, you used `Scanner` method `nextInt` to input an integer typed by the user at the keyboard. When `nextInt` reads a value from the user, it *returns* that value for use in the program. As you’ll see shortly, `Account` method `getName` returns a value.

Method `setName` receives *parameter* `name` of type `String` —which represents the name that will be passed to the method as an *argument*. You’ll see how parameters and arguments work together when we discuss the method call in line 19 of Fig. 3.2.

Parameters are declared in a **parameter list**, which is located inside the parentheses that follow the method name in the method header. When there are multiple parameters, each is separated from the next by a comma. Each parameter *must* specify a type (in this case, `String`) followed by a variable name (in this case, `name`).

Parameters Are Local Variables

In Chapter 2, we declared all of an app’s variables in the `main` method. Variables declared in a particular method’s body (such as `main`) are **local variables** which can be used *only* in that method. Each method can access its own local variables, not those of other methods. When a method terminates, the values of its local variables are *lost*. A method’s parameters also are local variables of the method.

`setName` Method Body

Every *method body* is delimited by a pair of *braces* (as in lines 9 and 11 of Fig. 3.1) containing one or more statements that perform the method’s task(s). In this case, the method body contains a single statement (line 10) that assigns the value of the `name` *parameter* (a `String`) to the class’s `name` *instance variable*, thus storing the account name in the object.

If a method contains a local variable with the *same* name as an instance variable (as in lines 9 and 6, respectively), that method’s body will refer to the local variable rather than the instance variable. In this case, the local variable is said to *shadow* the instance variable in the method’s body. The method’s body can use the keyword `this` to refer to the shadowed instance variable explicitly, as shown on the left side of the assignment in line 10. After line 10 executes, the method has completed its task, so it returns to its *caller*.

Good Programming Practice 3.2

We could have avoided the need for keyword `this` *here by choosing a different name for the parameter in line 9, but using the* `this` *keyword as shown in line 10 is a widely accepted practice to minimize the proliferation of identifier names.*

`getName` Method of Class `Account`

Method `getName` (lines 14–16)

```
public String getName() {
   return name; // return value of name to caller
}
```

returns a particular `Account` object's `name` to the caller. The method has an *empty* parameter list, so it does *not* require additional information to perform its task. The method returns a `String`. When a method that specifies a return type *other* than `void` is called and completes its task, it *must* return a result to its caller. A statement that calls method `getName` on an `Account` object (such as the ones in lines 14 and 24 of Fig. 3.2) expects to receive the `Account`'s name—a

`String`, as specified in the method declaration's *return type*.

The `return` statement in line 15 of Fig. 3.1 passes the `String` value of instance variable `name` back to the caller. For example, when the value is returned to the statement in lines 23–24 of Fig. 3.2, the statement uses that value to output the name.

3.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account`

Next, we'd like to use class `Account` in an app and *call* each of its methods. A class that contains a `main` method begins the execution of a Java app. Class `Account` *cannot* execute by itself because it does *not* contain a `main` method—if you type `java Account` in the command window, you'll get an error indicating "`Main method not found in class Account`." To fix this problem, you must either declare a *separate* class that contains a `main` method or place a `main` method in class `Account`.

Driver Class `AccountTest`

A person drives a car by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without having to know how the car's internal mechanisms work. Similarly, a method (such as `main`) "drives" an `Account` object by calling its methods—without having to know how the class's internal mechanisms work. In this sense, the class containing method `main` is referred to as a **driver class**.

To help you prepare for the larger programs you'll encounter later in this book and in industry, we define class `AccountTest` and its `main` method in the file `AccountTest.java` (Fig. 3.2). Once `main` begins executing, it may call other methods in this and other classes; those may, in turn, call other methods, and so on. Class `AccountTest`'s `main` method creates one `Account` object and calls its `getName` and `setName` methods.

```
// Fig. 3.2: AccountTest.java
// Creating and manipulating an Account object.
import java.util.Scanner;

public class AccountTest {
   public static void main(String[] args) {
      // create a Scanner object to obtain input
      Scanner input = new Scanner(System.in);

      // create an Account object and assign it
      Account myAccount = new Account();

      // display initial value of name (null)
      System.out.printf("Initial name is: %s%n%n

      // prompt for and read name
      System.out.println("Please enter the name:
      String theName = input.nextLine(); // read
      myAccount.setName(theName); // put theName
      System.out.println(); // outputs a blank l
21
22         // display the name stored in object myAcc
23         System.out.printf("Name in object myAccoun
24            myAccount.getName());
25      }
26   }
```

```
Initial name is: null
Please enter the name:
Jane Green
Name in object myAccount is:
Jane Green
```

Fig. 3.2

Creating and manipulating an `Account` object.

`Scanner` Object for Receiving Input from the User

Line 8 creates a `Scanner` object named `input` for inputting a name from the user. Line 17 prompts the user to enter a name. Line 18 uses the `Scanner` object's `nextLine` method to read the name from the user and assign it to the *local* variable `theName`. You type the name and press *Enter* to submit it to the program. Pressing *Enter* inserts a newline character after the characters you typed. Method `nextLine` reads characters (*including white-space characters*, such as the blank in `"Jane Green"`) until it encounters the newline, then returns a `String` containing the characters up to, but *not* including, the newline, which is *discarded*.

Class `Scanner` provides various other input methods, as you'll see throughout the book. A method similar to `nextLine`—named `next`—reads the *next word*. When you press *Enter* after typing some text, method `next` reads characters until it encounters a *white-space character* (such as a space, tab or newline), then returns a `String` containing the characters up to, but *not* including, the white-space character, which is *discarded*. All information after the first white-space character is *not lost*—it can be read by subsequent statements that call the `Scanner`'s methods later in the program.

Instantiating an Object—Keyword new and Constructors

Line 11 creates an `Account` object and assigns it to variable `myAccount` of type `Account`. The variable is initialized with the result of `new Account()`—a **class instance creation expression**. Keyword `new` creates a new object of the specified class—in this case, `Account`. The parentheses are *required*. As you'll learn in Section 3.3, those parentheses in combination with a class name represent a call to a **constructor**, which is *similar* to a method but is called implicitly by the `new` operator to *initialize* an object's instance variables when the object is *created*. In Section 3.3, you'll see how to place an *argument* in the parentheses to specify an *initial value* for an `Account` object's `name` instance variable —you'll enhance class `Account` to enable this. For now, we simply leave the parentheses *empty*. Line 8 contains a class instance creation expression for a `Scanner` object—the expression initializes the `Scanner` with `System.in`, which tells the `Scanner` where to read the input from (i.e., the keyboard).

Calling Class `Account`'s `getName` Method

Line 14 displays the *initial* name, which is obtained by calling the object's `getName` method. Just as we can use object `System.out` to call its methods `print`, `printf` and `println`, we can use object `myAccount` to call its methods `getName` and `setName`. Line 14 calls `getName` using the `myAccount` object created in line 11, followed by a **dot separator** (**.**), then the method name `getName` and an *empty* set of parentheses because no arguments are being passed. When `getName` is called:

1. The app transfers program execution from the call (line 14 in `main`) to method `getName`'s declaration (lines 14–16 of Fig. 3.1). Because `getName` was called via the `myAccount` object, `getName` "knows"

which object's instance variable to manipulate.

2. Next, method `getName` performs its task—that is, it *returns* the `name` (line 15 of Fig. 3.1). When the `return` statement executes, program execution continues where `getName` was called (line 14 in Fig. 3.2).
3. `System.out.printf` displays the `String` returned by method `getName`, then the program continues executing at line 17 in `main`.

Error-Prevention Tip 3.1

Never use as a format-control a string that was input from the user. When method `System.out.printf` *evaluates the format-control string in its first argument, the method performs tasks based on the conversion specifier(s) in that string. If the format-control string were obtained from the user, a malicious user could supply conversion specifiers that would be executed by* `System.out.printf`*, possibly causing a security breach.*

`null`—the Default Initial Value for String Variables

The first line of the output shows the name "`null`." Unlike local variables, which are *not* automatically initialized, *every instance variable has a* **default initial value**—a value provided by Java when you do *not* specify the instance variable's initial value. Thus, *instance variables* are *not* required to be explicitly initialized before they're used in a program—unless they must be initialized to values *other than* their default values. The default value for an instance variable of type `String` (like `name` in this example) is `null`, which we discuss further in Section 3.5 when we consider *reference types*.

Calling Class `Account`'s `setName` Method

Line 19 calls `myAccounts`'s `setName` method. A method call can supply *arguments* whose *values* are assigned to the corresponding method parameters. In this case, the value of `main`'s local variable `theName` in parentheses is the *argument* that's passed to `setName` so that the method can perform its task. When `setName` is called:

1. The app transfers program execution from line 19 in `main` to `setName` method's declaration (lines 9–11 of Fig. 3.1), and the *argument value* in the call's parentheses (`theName`) is assigned to the corresponding *parameter* (`name`) in the method header (line 9 of Fig. 3.1). Because `setName` was called via the `myAccount` object, `setName` "knows" which object's instance variable to manipulate.
2. Next, method `setName` performs its task—that is, it assigns the `name` parameter's value to instance variable `name` (line 10 of Fig. 3.1).
3. When program execution reaches `setName`'s closing right brace, it returns to where `setName` was called (line 19 of Fig. 3.2), then continues at line 20 of Fig. 3.2.

The number of *arguments* in a method call *must match* the number of *parameters* in the method declaration's parameter list. Also, the argument types in the method call must be *consistent* with the types of the corresponding parameters in the method's declaration. (As you'll learn in Chapter 6, an argument's type and its corresponding parameter's type are *not* required to be *identical*.) In our example, the method call passes one argument of type `String` (`theName`)—and the method declaration specifies one parameter of type `String` (`name`, declared in line 9 of Fig. 3.1). So in this example the type of the argument in the method call *exactly* matches the type of the parameter in the method header.

Displaying the Name That Was Entered by the User

Line 20 of Fig. 3.2 outputs a blank line. When the second call to method `getName` (line 24) executes, the name entered by the user in line 18 is displayed. After the statement at lines 23–24 completes execution, the end of method `main` is reached, so the program terminates.

Note that lines 23–24 represent only *one* statement. Java allows long statements to be split over multiple lines. We indent line 24 to indicate that it's a *continuation* of line 23.

Common Programming Error 3.1

Splitting a statement in the middle of an identifier or a string is a syntax error.

3.2.3 Compiling and Executing an App with Multiple Classes

You must compile the classes in Figs. 3.1 and 3.2 before you can *execute* the app. This is the first time you've created an app with *multiple* classes. Class `AccountTest` has a `main` method; class `Account` does not. To compile this app, first change to the directory that contains the app's source-code files. Next, type the command

```
javac Account.java AccountTest.java
```

to compile *both* classes at once. If the directory containing the app includes *only* this app's files, you can compile both classes with the command

```
javac *.java
```

The asterisk (`*`) in `*.java` is a wildcard that indicates *all* files in the *current* directory ending with the filename extension "`.java`" should be compiled. If both classes compile correctly—that is, no compilation errors are displayed

—you can then run the app with the command

```
java AccountTest
```

3.2.4 `Account` UML Class Diagram

We'll often use UML class diagrams to help you visualize a class's *attributes* and *operations*. In industry, UML diagrams help systems designers specify a system in a concise, graphical, programming-language-independent manner, before programmers implement the system in a specific programming language. Figure 3.3 presents a **UML class diagram** for class `Account` of Fig. 3.1.

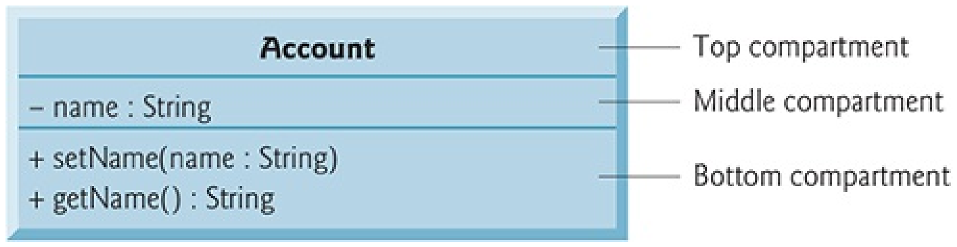

Fig. 3.3

UML class diagram for class `Account` of Fig. 3.1.

Description

Top Compartment

In the UML, each class is modeled in a class diagram as a rectangle with three compartments. In this diagram the *top* compartment contains the *class name* `Account` centered horizontally in boldface type.

Middle Compartment

The *middle* compartment contains the *class's attribute* `name`, which corresponds to the instance variable of the same name in Java. Instance variable `name` is `private` in Java, so the UML class diagram lists a *minus sign (−) access modifier* before the attribute name. Following the attribute name are a *colon* and the *attribute type*, in this case `String`.

Bottom Compartment

The *bottom* compartment contains the class's **operations**, `setName` and `getName`, which correspond to the methods of the same names in Java. The UML models operations by listing the operation name preceded by an *access modifier*, in this case `+ getName`. This plus sign (`+`) indicates that `getName` is a *public* operation in the UML (because it's a `public` method in Java). Operation `getName` does *not* have any parameters, so the parentheses following the operation name in the class diagram are *empty*, just as they are in the method's declaration in line 14 of Fig. 3.1. Operation `setName`, also a public operation, has a `String` parameter called `name`.

Return Types

The UML indicates the *return type* of an operation by placing a colon and the return type *after* the parentheses following the operation name. `Account` method `getName` (Fig. 3.1) has a `String` return type. Method `setName` *does not* return a value (because it returns `void` in Java), so the UML class diagram *does not* specify a return type after the parentheses of this operation.

Parameters

The UML models a parameter a bit differently from Java (recall that the UML is programming-language independent) by listing the parameter name, followed by a colon and the parameter type in the parentheses after the operation name. The UML has its own data types similar to those of Java, but for simplicity, we'll use the Java data types. `Account` method `setName` (Fig. 3.1) has a `String` parameter named `name`, so Fig. 3.3 lists `name : String` between the parentheses following the method name.

3.2.5 Additional Notes on

Class `AccountTest`

`static` Method `main`

In Chapter 2, each class we declared had one `main` method. Recall that `main` is *always* called automatically by the Java Virtual Machine (JVM) when you execute an app. You must call most other methods *explicitly* to tell them to perform their tasks. In Chapter 6, you'll learn that method `toString` is commonly invoked *implicitly*.

Lines 6–25 of Fig. 3.2 declare method `main`. A key part of enabling the JVM to locate and call method `main` to begin the app's execution is the `static` keyword (line 6), which indicates that `main` is a `static` method. A `static` method is special, because you can call it *without first creating an object of the class in which the method is declared*—in this case class `AccountTest`. We discuss `static` methods in detail in Chapter 6.

Notes on `import` Declarations

Notice the `import` declaration in Fig. 3.2 (line 3), which indicates to the compiler that the program uses class `Scanner`. As you learned in Chapter 2, classes `System` and

`String` are in package `java.lang`, which is *implicitly* imported into *every* Java program, so all programs can use that package's classes without explicitly importing them. Most other classes you'll use in Java programs must be imported *explicitly*.

There's a special relationship between classes that are compiled in the *same* directory, like classes `Account` and `AccountTest`. By default, such classes are considered to be in the *same* package—known as the **default package**. Classes in the same package are *implicitly imported* into the source-code files of other classes in that package. Thus, an `import` declaration is *not* required when one class in a package uses another in the same package—such as when class `AccountTest` uses class `Account`.

The `import` declaration in line 3 is *not* required if we refer to class `Scanner` throughout this file as `java.util.Scanner`, which includes the *full package name and class name*. This is known as the class's **fully qualified class name**. For example, line 8 of Fig. 3.2 also could be written as

```
java.util.Scanner input = new java.util.Scanner(Syste
```

Software Engineering

Observation 3.1

The Java compiler does not require `import` *declarations in a Java source-code file if the fully qualified class name is specified every time a class name is used. Most Java programmers prefer the more concise programming style enabled by* `import` *declarations.*

3.2.6 Software Engineering with private Instance Variables and public *set* and *get* Methods

As you'll see, through the use of *set* and *get* methods, you can *validate* attempted modifications to `private` data and control how that data is presented to the caller—these are compelling software engineering benefits. We'll discuss this in more detail in Section 3.4.

If the instance variable were `public`, any **client** of the class —that is, any other class that calls the class's methods—could see the data and do whatever it wanted with it, including setting it to an *invalid* value.

You might think that even though a client of the class cannot directly access a `private` instance variable, the client can do whatever it wants with the variable through `public` *set* and *get* methods. You would think that you could peek at the `private` data any time with the `public` *get* method and that you could modify the `private` data at will through the `public` *set* method. But *set* methods can be programmed to *validate* their arguments and reject any attempts to *set* the data to bad values, such as a negative body temperature, a day in March out of the range 1 through 31, a product code not in the company's product catalog, etc. And a *get* method can present the data in a different form. For example, a `Grade` class might store a grade as an `int` between 0 and 100, but a `getGrade` method might return a letter grade as a `String`, such as `"A"` for grades between 90 and 100, `"B"` for grades between 80 and 89, etc. Tightly controlling the access to and presentation of `private` data can greatly reduce errors, while increasing the robustness and security of your programs.

Declaring instance variables with access modifier `private` is known as *information hiding*. When a program creates (instantiates) an object of class `Account`, variable `name` is *encapsulated* (hidden) in the object and can be accessed only by methods of the object's class.

Software Engineering Observation 3.2

Precede each instance variable and method declaration with an access modifier. Generally, instance variables should be declared `private` *and methods* `public`. *Later in the book,* *we’ll discuss why you might want to declare a method* `private`.

Conceptual View of an `Account` Object with Encapsulated Data

You can think of an `Account` object as shown in Fig. 3.4. The `private` instance variable `name` is *hidden inside* the object (represented by the inner circle containing `name`) and *protected by an outer layer* of `public` methods (represented by the outer circle containing `getName` and `setName`). Any client code that needs to interact with the `Account` object can do so *only* by calling the `public` methods of the protective outer layer.

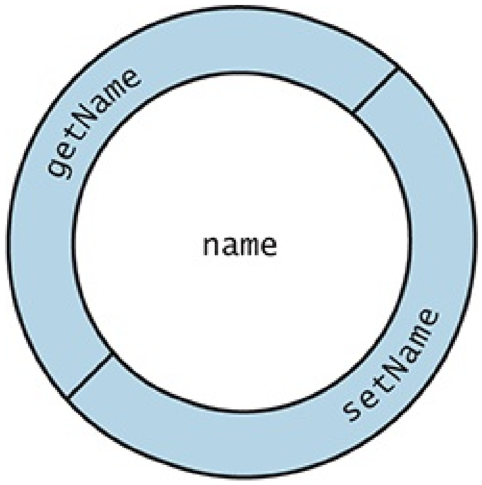

Fig. 3.4

Conceptual view of an `Account` object with its encapsulated `private` instance variable `name` and protective layer of `public` methods.