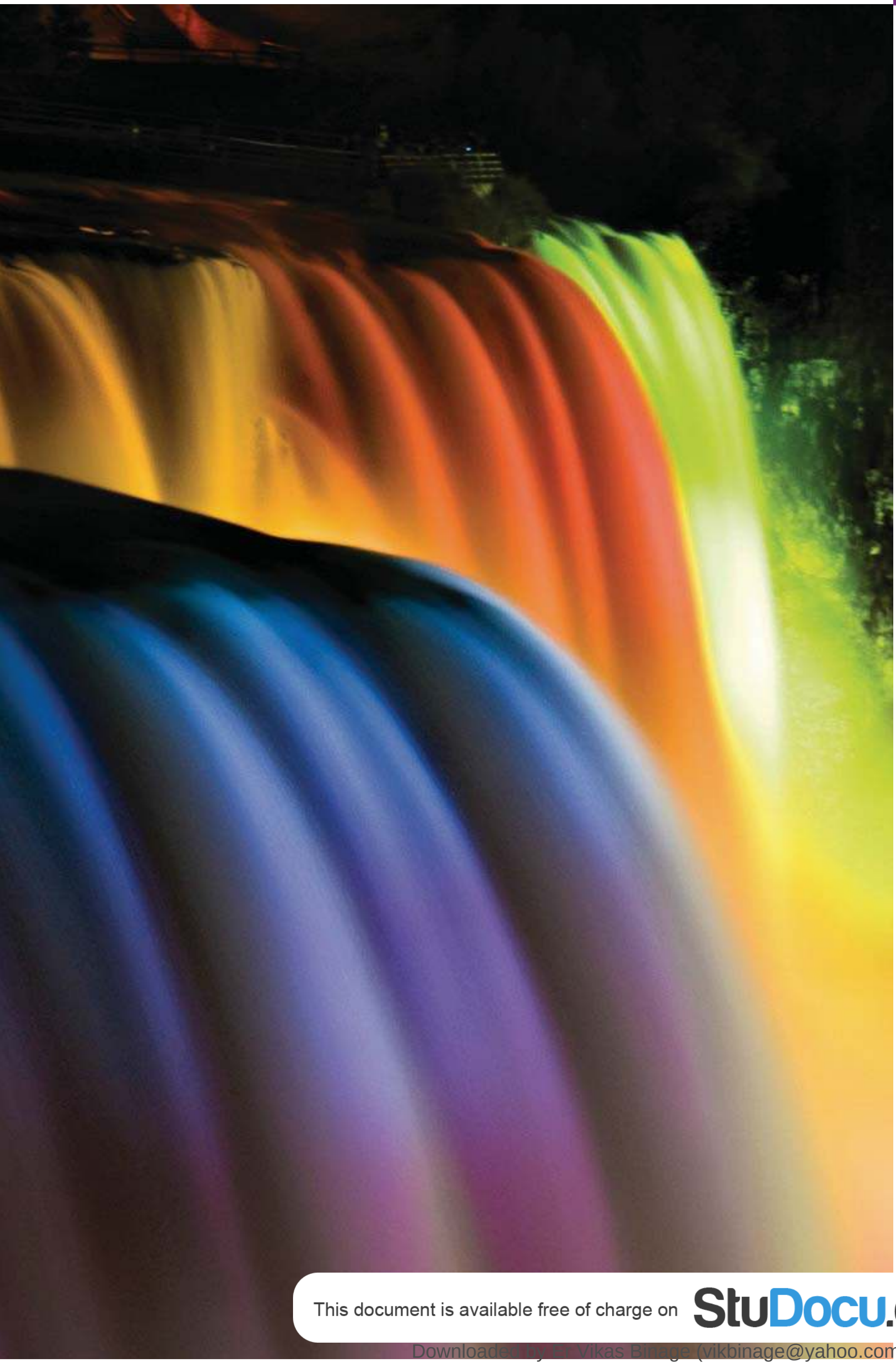
Chapter 03 - Introduction to Classes, Objects, Methods and Strings

Object Oriented Software Design and Java Programming (University of Birmingham)

Introduction to Classes, Objects, Methods and Strings

3



Nothing can have value without being an object of utility.
—Karl Marx

Your public servants serve you right.
—Adlai E. Stevenson

You'll see something new. Two things. And I call them Thing One and Thing Two.
—Dr. Theodor Seuss Geisel

Objectives

In this chapter you'll learn:

- How to declare a class and use it to create an object.
- How to implement a class's behaviors as methods.
- How to implement a class's attributes as instance variables and properties.
- How to call an object's methods to make them perform their tasks.
- What instance variables of a class and local variables of a method are.
- How to use a constructor to initialize an object's data.
- The differences between primitive and reference types.

Outline

3.1 Introduction

We introduced the basic terminology and concepts of object-oriented programming in Section 1.6. In this chapter, we present a simple framework for organizing object-oriented applications in Java. Typically, the applications you develop in this book will consist of two or more classes. If you become part of a development team in industry, you might work on applications that contain hundreds, or even thousands, of classes.

First, we motivate the notion of classes with a real-world example. Then we present five applications to demonstrate creating and using your own classes. The first four of these begin our case study on developing a grade book class that instructors can use to maintain student test scores. This case study is enhanced in Chapters 4, 5 and 7. The last example introduces floating-point numbers—that is, numbers containing decimal points—in a bank account class that maintains a customer's balance.

3.2 Declaring a Class with a Method and Instantiating an Object of a Class

In Sections 2.5 and 2.8, you created an object of the *existing* class `Scanner`, then used that object to read data from the keyboard. In this section, you'll create a *new* class, then use it to create an object. We begin by delcaring classes `GradeBook` (Fig. 3.1) and `GradeBookTest` (Fig. 3.2). Class `GradeBook` (declared in the file `GradeBook.java`) will be used to display a message on the screen (Fig. 3.2) welcoming the instructor to the grade book application. Class `GradeBookTest` (declared in the file `GradeBookTest.java`) is an application class in which the `main` method will create and use an object of class `GradeBook`. *Each class declaration that begins with keyword `public` must be stored in a file having the same name as the class and ending with the `.java` file-name extension.* Thus, classes `GradeBook` and `GradeBookTest` must be declared in *separate* files, because each class is declared `public`.

Class GradeBook

The `GradeBook` class declaration (Fig. 3.1) contains a `displayMessage` method (lines 7–10) that displays a message on the screen. We'll need to make an object of this class and call its method to execute line 9 and display the message.

The *class declaration* begins in line 4. The keyword `public` is an **access modifier**. For now, we'll simply declare every class `public`. Every class declaration contains keyword

```
 1  // Fig. 3.1: GradeBook.java
 2  // Class declaration with one method.
 3
 4  public class GradeBook
 5  {
 6     // display a welcome message to the GradeBook user
 7     public void displayMessage()
 8     {
 9        System.out.println( "Welcome to the Grade Book!" );
10     } // end method displayMessage
11  } // end class GradeBook
```

Fig. 3.1 | Class declaration with one method.

`class` followed immediately by the class's name. Every class's body is enclosed in a pair of left and right braces, as in lines 5 and 11 of class `GradeBook`.

In Chapter 2, each class we declared had one method named `main`. Class `GradeBook` also has one method—`displayMessage` (lines 7–10). Recall that `main` is a special method that's *always* called automatically by the Java Virtual Machine (JVM) when you execute an application. Most methods do not get called automatically. As you'll soon see, you must call method `displayMessage` explicitly to tell it to perform its task.

The method declaration begins with keyword `public` to indicate that the method is "available to the public"—it can be called from methods of other classes. Next is the method's **return type**, which specifies the type of data the method returns to its caller after performing its task. The return type `void` indicates that this method will perform a task but will *not* return (i.e., give back) any information to its **calling method**. You've used methods that return information—for example, in Chapter 2 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.

The name of the method, `displayMessage`, follows the return type. By convention, method names begin with a lowercase first letter and subsequent words in the name begin with a capital letter. The parentheses after the method name indicate that this is a method. Empty parentheses, as in line 7, indicate that this method does not require additional information to perform its task. Line 7 is commonly referred to as the **method header**. Every method's body is delimited by left and right braces, as in lines 8 and 10.

The body of a method contains one or more statements that perform the method's task. In this case, the method contains one statement (line 9) that displays the message `"Welcome to the Grade Book!"` followed by a newline (because of `println`) in the command window. After this statement executes, the method has completed its task.

Class `GradeBookTest`

Next, we'd like to use class `GradeBook` in an application. As you learned in Chapter 2, method `main` begins the execution of *every* application. A class that contains method `main` begins the execution of a Java application. Class `GradeBook` is *not* an application because it does *not* contain `main`. Therefore, if you try to execute `GradeBook` by typing `java GradeBook` in the command window, an error will occur. This was not a problem in Chapter 2, because every class you declared had a `main` method. To fix this problem, we must either declare a separate class that contains a `main` method or place a `main` method in class `Grade-`

`Book`. To help you prepare for the larger programs you'll encounter later in this book and in industry, we use a separate class (`GradeBookTest` in this example) containing method `main` to test each new class we create in this chapter. Some programmers refer to such a class as a *driver class*.

The `GradeBookTest` class declaration (Fig. 3.2) contains the `main` method that will control our application's execution. The `GradeBookTest` class declaration begins in line 4 and ends in line 15. The class, like many that begin an application's execution, contains *only* a `main` method.

```
 1  // Fig. 3.2: GradeBookTest.java
 2  // Creating a GradeBook object and calling its displayMessage method.
 3
 4  public class GradeBookTest
 5  {
 6     // main method begins program execution
 7     public static void main( String[] args )
 8     {
 9        // create a GradeBook object and assign it to myGradeBook
10        GradeBook myGradeBook = new GradeBook();
11
12        // call myGradeBook's displayMessage method
13        myGradeBook.displayMessage();
14     } // end main
15  } // end class GradeBookTest
```

```
Welcome to the Grade Book!
```

Fig. 3.2 | Creating a `GradeBook` object and calling its `displayMessage` method.

Lines 7–14 declare method `main`. A key part of enabling the JVM to locate and call method `main` to begin the application's execution is the `static` keyword (line 7), 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.* We discuss `static` methods in Chapter 6, Methods: A Deeper Look.

In this application, we'd like to call class `GradeBook`'s `displayMessage` method to display the welcome message in the command window. Typically, you cannot call a method that belongs to another class until you create an object of that class, as shown in line 10. We begin by declaring variable `myGradeBook`. The variable's type is `GradeBook`—the class we declared in Fig. 3.1. Each new *class* you create becomes a new *type* that can be used to declare variables and create objects. You can declare new class types as needed; this is one reason why Java is known as an **extensible language**.

Variable `myGradeBook` is initialized (line 10) with the result of the **class instance creation expression** `new GradeBook()`. Keyword **`new`** creates a new object of the class specified to the right of the keyword (i.e., `GradeBook`). The parentheses to the right of `GradeBook` are required. As you'll learn in Section 3.6, those parentheses in combination with a class name represent a call to a **constructor**, which is similar to a method but is used only at the time an object is *created* to *initialize* the object's data. You'll see that data can be placed in the parentheses to specify *initial values* for the object's data. For now, we simply leave the parentheses empty.

Just as we can use object `System.out` to call its methods `print`, `printf` and `println`, we can use object `myGradeBook` to call its method `displayMessage`. Line 13 calls the method `displayMessage` (lines 7–10 of Fig. 3.1) using `myGradeBook` followed by a **dot separator** (`.`), the method name `displayMessage` and an empty set of parentheses. This call causes the `displayMessage` method to perform its task. This method call differs from those in Chapter 2 that displayed information in a command window—each of those method calls provided arguments that specified the data to display. At the beginning of line 13, "`myGradeBook.`" indicates that `main` should use the `myGradeBook` object that was created in line 10. Line 7 of Fig. 3.1 indicates that method `displayMessage` has an *empty parameter list*—that is, `displayMessage` does *not* require additional information to perform its task. For this reason, the method call (line 13 of Fig. 3.2) specifies an empty set of parentheses after the method name to indicate that *no arguments* are being passed to method `displayMessage`. When method `displayMessage` completes its task, method `main` continues executing at line 14. This is the end of method `main`, so the program terminates.

Any class can contain a `main` method. The JVM invokes the `main` method *only* in the class used to execute the application. If an application has multiple classes that contain `main`, the one that's invoked is the one in the class named in the `java` command.

Compiling an Application with Multiple Classes

You must compile the classes in Fig. 3.1 and Fig. 3.2 before you can execute the application. First, change to the directory that contains the application's source-code files. Next, type the command

```
javac GradeBook.java GradeBookTest.java
```

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

```
javac *.java
```

The asterisk (*) in `*.java` indicates that *all* files in the current directory that end with the file-name extension "`.java`" should be compiled.

UML Class Diagram for Class `GradeBook`

Figure 3.3 presents a **UML class diagram** for class `GradeBook` of Fig. 3.1. In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top compartment contains the name of the class centered horizontally in boldface type. The middle compartment contains the class's attributes, which correspond to instance variables (discussed in Section 3.4) in Java. In Fig. 3.3, the middle compartment is empty, because this `GradeBook` class does *not* have any attributes. The bottom compartment contains the

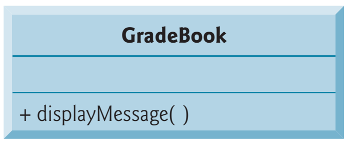


Fig. 3.3 | UML class diagram indicating that class `GradeBook` has a `public` `displayMessage` operation.

class's **operations**, which correspond to methods in Java. The UML models operations by listing the operation name preceded by an access modifier (in this case +) and followed by a set of parentheses. Class `GradeBook` has one method, `displayMessage`, so the bottom compartment of Fig. 3.3 lists one operation with this name. Method `displayMessage` does *not* require additional information to perform its tasks, so the parentheses following the method name in the class diagram are *empty*, just as they were in the method's declaration in line 7 of Fig. 3.1. The plus sign (+) in front of the operation name indicates that `displayMessage` is a public operation in the UML (i.e., a `public` method in Java). We'll often use UML class diagrams to summarize a class's attributes and operations.

3.3 Declaring a Method with a Parameter

In our car analogy from Section 1.6, we discussed the fact that pressing a car's gas pedal sends a *message* to the car to *perform a task*—to go faster. But *how fast* should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates. So the message to the car actually includes the *task to perform* and *additional information* that helps the car perform the task. This additional information is known as a **parameter**—the value of the parameter helps the car determine how fast to accelerate. Similarly, a method can require one or more parameters that represent additional information it needs to perform its task. Parameters are defined in a comma-separated **parameter list**, which is located inside the parentheses that follow the method name. Each parameter must specify a *type* and a variable name. The parameter list may contain any number of parameters, including none at all. Empty parentheses following the method name (as in Fig. 3.1, line 7) indicate that a method does *not* require any parameters.

Arguments to a Method

A method call supplies values—called *arguments*—for each of the method's parameters. For example, the method `System.out.println` requires an argument that specifies the data to output in a command window. Similarly, to make a deposit into a bank account, a `deposit` method specifies a parameter that represents the deposit amount. When the `deposit` method is called, an argument value representing the deposit amount is assigned to the method's parameter. The method then makes a deposit of that amount.

Class Declaration with a Method That Has One Parameter

We now declare class `GradeBook` (Fig. 3.4) with a `displayMessage` method that displays the course name as part of the welcome message. (See the sample execution in Fig. 3.5.) The new method requires a parameter that represents the course name to output.

Before discussing the new features of class `GradeBook`, let's see how the new class is used from the `main` method of class `GradeBookTest` (Fig. 3.5). Line 12 creates a `Scanner` named `input` for reading the course name from the user. Line 15 creates the `GradeBook` object `myGradeBook`. Line 18 prompts the user to enter a course name. Line 19 reads the name from the user and assigns it to the `nameOfCourse` variable, using `Scanner` method **`nextLine`** to perform the input. The user types the course name and presses *Enter* to submit the course name to the program. Pressing *Enter* inserts a newline character at the end of the characters typed by the user. Method `nextLine` reads characters typed by the user until it encounters the newline character, then returns a `String` containing the characters up to, but *not* including, the newline. The newline character is *discarded*.

```java
// Fig. 3.4: GradeBook.java
// Class declaration with one method that has a parameter.

public class GradeBook
{
   // display a welcome message to the GradeBook user
   public void displayMessage( String courseName )
   {
      System.out.printf( "Welcome to the grade book for\n%s!\n",
         courseName );
   } // end method displayMessage
} // end class GradeBook
```

Fig. 3.4 | Class declaration with one method that has a parameter.

```java
// Fig. 3.5: GradeBookTest.java
// Create GradeBook object and pass a String to
// its displayMessage method.
import java.util.Scanner; // program uses Scanner

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // create Scanner to obtain input from command window
      Scanner input = new Scanner( System.in );

      // create a GradeBook object and assign it to myGradeBook
      GradeBook myGradeBook = new GradeBook();

      // prompt for and input course name
      System.out.println( "Please enter the course name:" );
      String nameOfCourse = input.nextLine(); // read a line of text
      System.out.println(); // outputs a blank line

      // call myGradeBook's displayMessage method
      // and pass nameOfCourse as an argument
      myGradeBook.displayMessage( nameOfCourse );
   } // end main
} // end class GradeBookTest
```

```
Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.5 | Create a `GradeBook` object and pass a `String` to its `displayMessage` method.

Class `Scanner` also provides a similar method—**`next`**—that reads individual words. When the user presses *Enter* after typing input, 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 other statements that call the `Scanner`'s methods later in the program. Line 20 outputs a blank line.

Line 24 calls `myGradeBooks`'s `displayMessage` method. The variable `nameOfCourse` in parentheses is the *argument* that's passed to method `displayMessage` so that the method can perform its task. The value of variable `nameOfCourse` in `main` becomes the value of method `displayMessage`'s *parameter* `courseName` in line 7 of Fig. 3.4. When you execute this application, notice that method `displayMessage` outputs the name you type as part of the welcome message (Fig. 3.5).

More on Arguments and Parameters

In Fig. 3.4, `displayMessage`'s parameter list (line 7) declares one parameter indicating that the method requires a `String` to perform its task. When the method is called, the argument value in the call is assigned to the corresponding parameter (`courseName`) in the method header. Then, the method body uses the value of the `courseName` parameter. Lines 9–10 of Fig. 3.4 display parameter `courseName`'s value, using the `%s` format specifier in `printf`'s format string. The parameter variable's name (`courseName` in Fig. 3.4, line 7) can be the *same or different* from the argument variable's name (`nameOfCourse` in Fig. 3.5, line 24).

The number of arguments in a method call *must* match the number of parameters in the parameter list of the method's declaration. 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 always required to be *identical*.) In our example, the method call passes one argument of type `String` (`nameOfCourse` is declared as a `String` in line 19 of Fig. 3.5) and the method declaration specifies one parameter of type `String` (`courseName` is declared as a `String` in line 7 of Fig. 3.4). So in this example the type of the argument in the method call exactly matches the type of the parameter in the method header.

*Updated UML Class Diagram for Class **GradeBook***

The UML class diagram of Fig. 3.6 models class `GradeBook` of Fig. 3.4. Like Fig. 3.1, this `GradeBook` class contains `public` operation `displayMessage`. However, this version of `displayMessage` has a parameter. The UML models a parameter a bit differently from Java by listing the parameter name, followed by a colon and the parameter type in the parentheses following the operation name. The UML has its own data types similar to those of Java (but, as you'll see, not all the UML data types have the same names as the corresponding Java types). The UML type `String` does correspond to the Java type `String`. `GradeBook` method `displayMessage` (Fig. 3.4) has a `String` parameter named `courseName`, so Fig. 3.6 lists `courseName : String` between the parentheses following `displayMessage`.

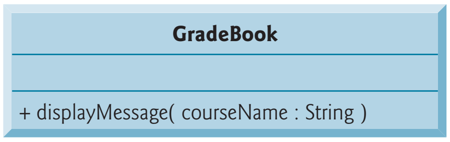


Fig. 3.6 | UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

Notes on `import` Declarations

Notice the `import` declaration in Fig. 3.5 (line 4). This indicates to the compiler that the program uses class `Scanner`. Why do we need to import class `Scanner`, but not classes `System`, `String` or `GradeBook`? 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 on disk, like classes `GradeBook` and `GradeBookTest`. 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 the same package. Thus, an `import` declaration is *not* required when one class in a package uses another in the same package—such as when class `GradeBookTest` uses class `GradeBook`.

The `import` declaration in line 4 is *not* required if we always refer to class `Scanner` 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 12 could be written as

```
    java.util.Scanner input = new java.util.Scanner( System.in );
```

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 in the source code. Most Java programmers prefer to use `import` declarations.

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

In Chapter 2, we declared all of an application's variables in the application's `main` method. Variables declared in the body of a particular method are known as **local variables** and can be used only in that method. When that method terminates, the values of its local variables are lost. Recall from Section 1.6 that an object has *attributes* that are carried with it as it's used in a program. Such attributes exist before a method is called on an object, while the method is executing and after the method completes execution.

A class normally consists of one or more methods that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class declaration. Such variables are called **fields** and are declared *inside* a class declaration but *outside* the bodies of the class's method declarations. When each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as an **instance variable**—each object (instance) of the class has a separate instance of the variable in memory. The example in this section demonstrates a `GradeBook` class that contains a `courseName` instance variable to represent a particular `GradeBook` object's course name.

`GradeBook` Class with an Instance Variable, a set Method and a get Method

In our next application (Figs. 3.7–3.8), class `GradeBook` (Fig. 3.7) maintains the course name as an instance variable so that it can be used or modified at any time during an application's execution. The class contains three methods—`setCourseName`, `getCourseName` and `displayMessage`. Method `setCourseName` stores a course name in a `GradeBook`. Method `getCourseName` obtains a `GradeBook`'s course name. Method `displayMessage`, which now specifies no parameters, still displays a welcome message that includes the course name; as you'll see, the method now obtains the course name by calling a method in the same class—`getCourseName`.

```
 1  // Fig. 3.7: GradeBook.java
 2  // GradeBook class that contains a courseName instance variable
 3  // and methods to set and get its value.
 4
 5  public class GradeBook
 6  {
 7     private String courseName; // course name for this GradeBook
 8
 9     // method to set the course name
10     public void setCourseName( String name )
11     {
12        courseName = name; // store the course name
13     } // end method setCourseName
14
15     // method to retrieve the course name
16     public String getCourseName()
17     {
18        return courseName;
19     } // end method getCourseName
20
21     // display a welcome message to the GradeBook user
22     public void displayMessage()
23     {
24        // calls getCourseName to get the name of
25        // the course this GradeBook represents
26        System.out.printf( "Welcome to the grade book for\n%s!\n",
27           getCourseName() );
28     } // end method displayMessage
29  } // end class GradeBook
```

Fig. 3.7 | `GradeBook` class that contains a `courseName` instance variable and methods to set and get its value.

A typical instructor teaches more than one course, each with its own course name. Line 7 declares `courseName` as a variable of type `String`. Because the variable is declared *in* the body of the class but *outside* the bodies of the class's methods (lines 10–13, 16–19 and 22–28), line 7 is a declaration for an *instance variable*. Every instance (i.e., object) of class `GradeBook` contains one copy of each instance variable. For example, if there are two `GradeBook` objects, each object has its own copy of `courseName`. A benefit of making `courseName` an instance variable is that all the methods of the class (in this case, `GradeBook`) can manipulate any instance variables that appear in the class (in this case, `courseName`).

Access Modifiers `public` *and* `private`

Most instance-variable declarations are preceded with the keyword `private` (as in line 7). Like `public`, keyword **`private`** is an *access modifier*. *Variables or methods declared with access modifier* `private` *are accessible only to methods of the class in which they're declared.* Thus, variable `courseName` can be used only in methods `setCourseName`, `getCourseName` and `displayMessage` of (every object of) class `GradeBook`.

Declaring instance variables with access modifier `private` is known as **data hiding** or information hiding. When a program creates (instantiates) an object of class `GradeBook`, variable `courseName` is *encapsulated* (hidden) in the object and can be accessed only by methods of the object's class. This prevents `courseName` from being modified accidentally by a class in another part of the program. In class `GradeBook`, methods `setCourseName` and `getCourseName` manipulate the instance variable `courseName`.

Software Engineering Observation 3.2

Precede each field and method declaration with an access modifier. Generally, instance variables should be declared `private` *and methods* `public`*. (It's appropriate to declare certain methods* `private`*, if they'll be accessed only by other methods of the class.)*

Good Programming Practice 3.1

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

Methods `setCourseName` *and* `getCourseName`

Method `setCourseName` (lines 10–13) does not return any data when it completes its task, so its return type is `void`. The method receives one parameter—`name`—which represents the course name that will be passed to the method as an argument. Line 12 assigns `name` to instance variable `courseName`.

Method `getCourseName` (lines 16–19) returns a particular `GradeBook` object's `courseName`. The method has an empty parameter list, so it does not require additional information to perform its task. The method specifies that it returns a `String`—this is the method's return type. When a method that specifies a return type other than `void` is called and completes its task, the method returns a *result* to its calling method. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you back a value that represents your balance. Similarly, when a statement calls method `getCourseName` on a `GradeBook` object, the statement expects to receive the `GradeBook`'s course name (in this case, a `String`, as specified in the method declaration's return type).

The **`return`** statement in line 18 passes the value of instance variable `courseName` back to the statement that calls method `getCourseName`. Consider, method `displayMessage`'s line 27, which calls method `getCourseName`. When the value is returned, the statement in lines 26–27 uses that value to output the course name. Similarly, if you have a method `square` that returns the square of its argument, you'd expect the statement

```
    int result = square( 2 );
```

to return 4 from method `square` and assign 4 to the variable `result`. If you have a method `maximum` that returns the largest of three integer arguments, you'd expect the statement

```
    int biggest = maximum( 27, 114, 51 );
```

to return `114` from method `maximum` and assign 114 to variable `biggest`.

The statements in lines 12 and 18 each use `courseName` *even though it was not declared in any of the methods*. We can use `courseName` in `GradeBook`'s methods because `courseName` is an instance variable of the class.

Method `displayMessage`

Method `displayMessage` (lines 22–28) does *not* return any data when it completes its task, so its return type is `void`. The method does *not* receive parameters, so the parameter list is empty. Lines 26–27 output a welcome message that includes the value of instance variable `courseName`, which is returned by the call to method `getCourseName` in line 27. Notice that one method of a class (`displayMessage` in this case) can call another method of the *same* class by using just the method name (`getCourseName` in this case).

`GradeBookTest` *Class That Demonstrates Class* `GradeBook`

Class `GradeBookTest` (Fig. 3.8) creates one object of class `GradeBook` and demonstrates its methods. Line 14 creates a `GradeBook` object and assigns it to local variable `myGradeBook` of type `GradeBook`. Lines 17–18 display the initial course name calling the object's `getCourseName` method. The first line of the output shows the name "`null`." *Unlike local variables, which are not automatically initialized, every field has a* ***default initial value****—a value provided by Java when you do not specify the field's initial value.* Thus, fields 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 a field of type `String` (like `courseName` in this example) is `null`, which we say more about in Section 3.5.

Line 21 prompts the user to enter a course name. Local `String` variable `theName` (declared in line 22) is initialized with the course name entered by the user, which is returned by the call to the `nextLine` method of the `Scanner` object `input`. Line 23 calls object `myGradeBook`'s `setCourseName` method and supplies `theName` as the method's argument. When the method is called, the argument's value is assigned to parameter `name` (line 10, Fig. 3.7) of method `setCourseName` (lines 10–13, Fig. 3.7). Then the parameter's value is assigned to instance variable `courseName` (line 12, Fig. 3.7). Line 24 (Fig. 3.8) skips a line in the output, then line 27 calls object `myGradeBook`'s `displayMessage` method to display the welcome message containing the course name.

```
 1  // Fig. 3.8: GradeBookTest.java
 2  // Creating and manipulating a GradeBook object.
 3  import java.util.Scanner; // program uses Scanner
 4
 5  public class GradeBookTest
 6  {
 7     // main method begins program execution
 8     public static void main( String[] args )
 9     {
10        // create Scanner to obtain input from command window
11        Scanner input = new Scanner( System.in );
12
13        // create a GradeBook object and assign it to myGradeBook
14        GradeBook myGradeBook = new GradeBook();
15
```

Fig. 3.8 | Creating and manipulating a `GradeBook` object. (Part 1 of 2.)

```
      // display initial value of courseName
      System.out.printf( "Initial course name is: %s\n\n",
         myGradeBook.getCourseName() );

      // prompt for and read course name
      System.out.println( "Please enter the course name:" );
      String theName = input.nextLine(); // read a line of text
      myGradeBook.setCourseName( theName ); // set the course name
      System.out.println(); // outputs a blank line

      // display welcome message after specifying course name
      myGradeBook.displayMessage();
   } // end main
} // end class GradeBookTest
```

```
Initial course name is: null

Please enter the course name:
CS101 Introduction to Java Programming

Welcome to the grade book for
CS101 Introduction to Java Programming!
```

Fig. 3.8 | Creating and manipulating a `GradeBook` object. (Part 2 of 2.)

set *and* get *Methods*

A class's `private` fields can be manipulated *only* by the class's methods. So a **client of an object**—that is, any class that calls the object's methods—calls the class's `public` methods to manipulate the `private` fields of an object of the class. This is why the statements in method `main` (Fig. 3.8) call the `setCourseName`, `getCourseName` and `displayMessage` methods on a `GradeBook` object. Classes often provide `public` methods to allow clients to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) `private` instance variables. The names of these methods need not begin with *set* or *get*, but this naming convention is recommended and is convention for special Java software components called JavaBeans, which can simplify programming in many Java integrated development environments (IDEs). The method that *sets* instance variable `courseName` in this example is called `setCourseName`, and the method that *gets* its value is called `getCourseName`.

GradeBook UML Class Diagram with an Instance Variable and set *and* get *Methods*

Figure 3.9 contains an updated UML class diagram for the version of class `GradeBook` in Fig. 3.7. This diagram models class `GradeBook`'s instance variable `courseName` as an attribute in the middle compartment of the class. The UML represents instance variables as attributes by listing the attribute name, followed by a colon and the attribute type. The UML type of attribute `courseName` is `String`. Instance variable `courseName` is `private` in Java, so the class diagram lists a minus sign (–) access modifier in front of the corresponding attribute's name. Class `GradeBook` contains three `public` methods, so the class diagram lists three operations in the third compartment. Recall that the plus sign (+) before each operation name indicates that the operation is `public`. Operation `setCourseName` has a `String` parameter called `name`. The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. Method `getCourseName` of class `GradeBook` (Fig. 3.7) has a `String` return type in Java, so the class diagram shows a `String` return type in the UML. Operations `setCourseName` and `displayMessage` *do not* return values (i.e., they return `void` in Java), so the UML class diagram *does not* specify a return type after the parentheses of these operations.

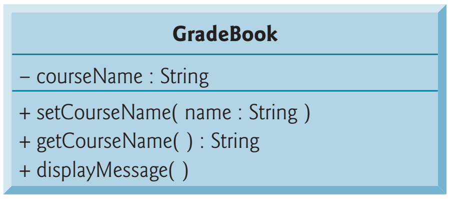


Fig. 3.9 | UML class diagram indicating that class `GradeBook` has a private `courseName` attribute of UML type `String` and three public operations—`setCourseName` (with a `name` parameter of UML type `String`), `getCourseName` (which returns UML type `String`) and `displayMessage`.

3.5 Primitive Types vs. Reference Types

Java's types are divided into primitive types and **reference types**. The primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`. All nonprimitive types are reference types, so classes, which specify the types of objects, are reference types.

A primitive-type variable can store exactly one *value of its declared type* at a time. For example, an `int` variable can store one whole number (such as 7) at a time. When another value is assigned to that variable, its initial value is replaced. Primitive-type instance variables are *initialized by default*—variables of types `byte`, `char`, `short`, `int`, `long`, `float` and `double` are initialized to 0, and variables of type `boolean` are initialized to `false`. You can specify your own initial value for a primitive-type variable by assigning the variable a value in its declaration, as in

```
    private int numberOfStudents = 10;
```

Recall that local variables are *not* initialized by default.

Error-Prevention Tip 3.1
An attempt to use an uninitialized local variable causes a compilation error.

Programs use variables of reference types (normally called **references**) to store the *locations* of objects in the computer's memory. Such a variable is said to **refer to an object** in the program. Objects that are referenced may each contain many instance variables. Line 14 of Fig. 3.8 creates an object of class `GradeBook`, and the variable `myGradeBook` contains a reference to that `GradeBook` object. *Reference-type instance variables are initialized by default to the value `null`*—a reserved word that represents a "reference to nothing." This is why the first call to `getCourseName` in line 18 of Fig. 3.8 returned `null`—the value of `courseName` had not been set, so the default initial value `null` was returned. The complete list of reserved words and keywords is listed in Appendix C.

When you use an object of another class, a reference to the object is required to **invoke** (i.e., call) its methods. In the application of Fig. 3.8, the statements in method `main` use the variable `myGradeBook` to send messages to the `GradeBook` object. These messages are calls to methods (like `setCourseName` and `getCourseName`) that enable the program to interact with the `GradeBook` object. For example, the statement in line 23 uses `myGradeBook` to send the `setCourseName` message to the `GradeBook` object. The message includes the argument that `setCourseName` requires to perform its task. The `GradeBook` object uses this information to set the `courseName` instance variable. Primitive-type variables do not refer to objects, so such variables cannot be used to invoke methods.

Software Engineering Observation 3.3

A variable's declared type (e.g., `int`*,* `double` *or* `GradeBook`*) indicates whether the variable is of a primitive or a reference type. If a variable is* not *of one of the eight primitive types, then it's of a reference type.*

3.6 Initializing Objects with Constructors

As mentioned in Section 3.4, when an object of class `GradeBook` (Fig. 3.7) is created, its instance variable `courseName` is initialized to `null` by default. What if you want to provide a course name when you create a `GradeBook` object? Each class you declare can provide a special method called a constructor that can be used to initialize an object of a class when the object is created. In fact, Java *requires* a constructor call for *every* object that's created. Keyword `new` requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object. The call is indicated by the parentheses after the class name. A constructor *must* have the *same name* as the class. For example, line 14 of Fig. 3.8 first uses `new` to create a `GradeBook` object. The empty parentheses after "`new GradeBook`" indicate a call to the class's constructor without arguments. By default, the compiler provides a **default constructor** with *no parameters* in any class that does *not* explicitly include a constructor. When a class has only the default constructor, its instance variables are initialized to their *default values*.

When you declare a class, you can provide your own constructor to specify custom initialization for objects of your class. For example, you might want to specify a course name for a `GradeBook` object when the object is created, as in

```
GradeBook myGradeBook =
   new GradeBook( "CS101 Introduction to Java Programming" );
```

In this case, the argument `"CS101 Introduction to Java Programming"` is passed to the `GradeBook` object's constructor and used to initialize the `courseName`. The preceding statement requires that the class provide a constructor with a `String` parameter. Figure 3.10 contains a modified `GradeBook` class with such a constructor.

```
// Fig. 3.10: GradeBook.java
// GradeBook class with a constructor to initialize the course name.

public class GradeBook
{
   private String courseName; // course name for this GradeBook

```

Fig. 3.10 | `GradeBook` class with a constructor to initialize the course name. (Part 1 of 2.)

```
 8      // constructor initializes courseName with String argument
 9      public GradeBook( String name ) // constructor name is class name
10      {
11         courseName = name; // initializes courseName
12      } // end constructor
13
14      // method to set the course name
15      public void setCourseName( String name )
16      {
17         courseName = name; // store the course name
18      } // end method setCourseName
19
20      // method to retrieve the course name
21      public String getCourseName()
22      {
23         return courseName;
24      } // end method getCourseName
25
26      // display a welcome message to the GradeBook user
27      public void displayMessage()
28      {
29         // this statement calls getCourseName to get the
30         // name of the course this GradeBook represents
31         System.out.printf( "Welcome to the grade book for\n%s!\n",
32            getCourseName() );
33      } // end method displayMessage
34   } // end class GradeBook
```

Fig. 3.10 | GradeBook class with a constructor to initialize the course name. (Part 2 of 2.)

Lines 9–12 declare GradeBook's constructor. Like a method, a constructor's parameter list specifies the data it requires to perform its task. When you create a new object (as we'll do in Fig. 3.11), this data is placed in the *parentheses that follow the class name*. Line 9 of Fig. 3.10 indicates that the constructor has a String parameter called name. The name passed to the constructor is assigned to instance variable courseName in line 11.

Figure 3.11 initializes GradeBook objects using the constructor. Lines 11–12 create and initialize the GradeBook object gradeBook1. The GradeBook constructor is called with the argument "CS101 Introduction to Java Programming" to initialize the course name. The class instance creation expression in lines 11–12 returns a reference to the new object, which is assigned to the variable gradeBook1. Lines 13–14 repeat this process, this time passing the argument "CS102 Data Structures in Java" to initialize the course name for gradeBook2. Lines 17–20 use each object's getCourseName method to obtain the course names and show that they were initialized when the objects were created. The output confirms that each GradeBook maintains its own copy of instance variable courseName.

An important difference between constructors and methods is that constructors cannot return values, so they cannot specify a return type (not even void). Normally, constructors are declared public. If a class does not include a constructor, the class's instance variables are initialized to their default values. *If you declare any constructors for a class, the Java compiler will not create a default constructor for that class.* Thus, we can no longer create a GradeBook object with new GradeBook() as we did in the earlier examples.

```java
// Fig. 3.11: GradeBookTest.java
// GradeBook constructor used to specify the course name at the
// time each GradeBook object is created.

public class GradeBookTest
{
   // main method begins program execution
   public static void main( String[] args )
   {
      // create GradeBook object
      GradeBook gradeBook1 = new GradeBook(
         "CS101 Introduction to Java Programming" );
      GradeBook gradeBook2 = new GradeBook(
         "CS102 Data Structures in Java" );

      // display initial value of courseName for each GradeBook
      System.out.printf( "gradeBook1 course name is: %s\n",
         gradeBook1.getCourseName() );
      System.out.printf( "gradeBook2 course name is: %s\n",
         gradeBook2.getCourseName() );
   } // end main
} // end class GradeBookTest
```

```
gradeBook1 course name is: CS101 Introduction to Java Programming
gradeBook2 course name is: CS102 Data Structures in Java
```

Fig. 3.11 | GradeBook constructor used to specify the course name at the time each GradeBook object is created.

Software Engineering Observation 3.4

Unless default initialization of your class's instance variables is acceptable, provide a constructor to ensure that they're properly initialized with meaningful values when each new object of your class is created.

*Adding the Constructor to Class **GradeBook**'s UML Class Diagram*

The UML class diagram of Fig. 3.12 models class `GradeBook` of Fig. 3.10, which has a constructor that has a `name` parameter of type `String`. Like operations, the UML models constructors in the third compartment of a class in a class diagram. To distinguish a

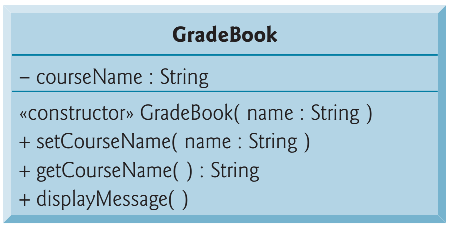

GradeBook
– courseName : String
«constructor» GradeBook(name : String)
+ setCourseName(name : String)
+ getCourseName( ) : String
+ displayMessage( )

Fig. 3.12 | UML class diagram indicating that class `GradeBook` has a constructor that has a `name` parameter of UML type `String`.

constructor from a class's operations, the UML requires that the word "constructor" be placed between **guillemets (« and »)** before the constructor's name. It's *customary* to list constructors *before* other operations in the third compartment.

Constructors with Multiple Parameters

Sometimes you'll want to initialize objects with multiple data items. In Exercise 3.11, we ask you to store the course name *and* the instructor's name in a `GradeBook` object. In this case, the `GradeBook`'s constructor would be modified to receive two `String`s, as in

```
    public GradeBook( String courseName, String instructorName )
```

and you'd call the `GradeBook` constructor as follows:

```
    GradeBook gradeBook = new GradeBook(
       "CS101 Introduction to Java Programming", "Sue Green" );
```

3.7 Floating-Point Numbers and Type `double`

We now depart temporarily from our `GradeBook` case study to declare an `Account` class that maintains the balance of a bank account. Most account balances are not whole numbers (such as 0, –22 and 1024). For this reason, class `Account` represents the account balance as a **floating-point number** (i.e., a number with a decimal point, such as 7.33, 0.0975 or 1000.12345). Java provides two primitive types for storing floating-point numbers in memory—`float` and `double`. They differ primarily in that `double` variables can store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point—also known as the number's **precision**) than `float` variables.

Floating-Point Number Precision and Memory Requirements

Variables of type **`float`** represent **single-precision floating-point numbers** and can represent up to *seven significant digits*. Variables of type **`double`** represent **double-precision floating-point numbers**. These require twice as much memory as `float` variables and provide *15 significant digits*—approximately double the precision of `float` variables. For the range of values required by most programs, variables of type `float` should suffice, but you can use `double` to "play it safe." In some applications, even `double` variables will be inadequate. Most programmers represent floating-point numbers with type `double`. In fact, Java treats all floating-point numbers you type in a program's source code (such as 7.33 and 0.0975) as `double` values by default. Such values in the source code are known as **floating-point literals**. See Appendix D, Primitive Types, for the ranges of values for `float`s and `double`s.

Although floating-point numbers are not always 100% precise, they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures. Owing to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more accurately. For this reason, we primarily use type `double` throughout the book. For precise floating-point numbers, Java provides class `BigDecimal` (package `java.math`).

Floating-point numbers also arise as a result of division. In conventional arithmetic, when we divide 10 by 3, the result is 3.3333333…, with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.

Account Class with an Instance Variable of Type double

Our next application (Figs. 3.13–3.14) contains a class named Account (Fig. 3.13) that maintains the balance of a bank account. A typical bank services many accounts, each with its own balance, so line 7 declares an instance variable named balance of type double. It's an instance variable because it's declared in the body of the class but outside the class's method declarations (lines 10–16, 19–22 and 25–28). Every instance (i.e., object) of class Account contains its own copy of balance.

```java
 1  // Fig. 3.13: Account.java
 2  // Account class with a constructor to validate and
 3  // initialize instance variable balance of type double.
 4
 5  public class Account
 6  {
 7     private double balance; // instance variable that stores the balance
 8
 9     // constructor
10     public Account( double initialBalance )
11     {
12        // validate that initialBalance is greater than 0.0;
13        // if it is not, balance is initialized to the default value 0.0
14        if ( initialBalance > 0.0 )
15           balance = initialBalance;
16     } // end Account constructor
17
18     // credit (add) an amount to the account
19     public void credit( double amount )
20     {
21        balance = balance + amount; // add amount to balance
22     } // end method credit
23
24     // return the account balance
25     public double getBalance()
26     {
27        return balance; // gives the value of balance to the calling method
28     } // end method getBalance
29  } // end class Account
```

Fig. 3.13 | Account class with a constructor to validate and initialize instance variable balance of type double.

The class has a constructor and two methods. It's common for someone opening an account to deposit money immediately, so the constructor (lines 10–16) receives a parameter initialBalance of type double that represents the *starting balance*. Lines 14–15 ensure that initialBalance is greater than 0.0. If so, initialBalance's value is assigned to instance variable balance. Otherwise, balance remains at 0.0—its default initial value.

Method credit (lines 19–22) does *not* return any data when it completes its task, so its return type is void. The method receives one parameter named amount—a double value that will be added to the balance. Line 21 adds `amount` to the current value of `balance`, then assigns the result to `balance` (thus replacing the prior balance amount).

Method `getBalance` (lines 25–28) allows clients of the class (i.e., other classes that use this class) to obtain the value of a particular `Account` object's `balance`. The method specifies return type `double` and an empty parameter list.

Once again, the statements in lines 15, 21 and 27 use instance variable `balance` even though it was *not* declared in any of the methods. We can use `balance` in these methods because it's an instance variable of the class.

`AccountTest` Class to Use Class `Account`

Class `AccountTest` (Fig. 3.14) creates two `Account` objects (lines 10–11) and initializes them with `50.00` and `-7.53`, respectively. Lines 14–17 output the balance in each `Account` by calling the `Account`'s `getBalance` method. When method `getBalance` is called for `account1` from line 15, the value of `account1`'s balance is returned from line 27 of Fig. 3.13 and displayed by the `System.out.printf` statement (Fig. 3.14, lines 14–15). Similarly, when method `getBalance` is called for `account2` from line 17, the value of the `account2`'s balance is returned from line 27 of Fig. 3.13 and displayed by the `System.out.printf` statement (Fig. 3.14, lines 16–17). The balance of `account2` is `0.00`, because the constructor ensured that the account could *not* begin with a negative balance. The value is output by `printf` with the format specifier `%.2f`. The format specifier **`%f`** is used to output values of type `float` or `double`. The `.2` between `%` and `f` represents the number of decimal places (2) that should be output to the right of the decimal point in the floating-point number—also known as the number's **precision**. Any floating-point value output with `%.2f` will be rounded to the hundredths position—for example, 123.457 would be rounded to 123.46, 27.333 would be rounded to 27.33 and 123.455 would be rounded to 123.46.

```
1   // Fig. 3.14: AccountTest.java
2   // Inputting and outputting floating-point numbers with Account objects.
3   import java.util.Scanner;
4
5   public class AccountTest
6   {
7      // main method begins execution of Java application
8      public static void main( String[] args )
9      {
10        Account account1 = new Account( 50.00 ); // create Account object
11        Account account2 = new Account( -7.53 ); // create Account object
12
13        // display initial balance of each object
14        System.out.printf( "account1 balance: $%.2f\n",
15           account1.getBalance() );
16        System.out.printf( "account2 balance: $%.2f\n\n",
17           account2.getBalance() );
18
19        // create Scanner to obtain input from command window
20        Scanner input = new Scanner( System.in );
21        double depositAmount; // deposit amount read from user
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 2.)

```
22
23        System.out.print( "Enter deposit amount for account1: " ); // prompt
24        depositAmount = input.nextDouble(); // obtain user input
25        System.out.printf( "\nadding %.2f to account1 balance\n\n",
26           depositAmount );
27        account1.credit( depositAmount ); // add to account1 balance
28
29        // display balances
30        System.out.printf( "account1 balance: $%.2f\n",
31           account1.getBalance() );
32        System.out.printf( "account2 balance: $%.2f\n\n",
33           account2.getBalance() );
34
35        System.out.print( "Enter deposit amount for account2: " ); // prompt
36        depositAmount = input.nextDouble(); // obtain user input
37        System.out.printf( "\nadding %.2f to account2 balance\n\n",
38           depositAmount );
39        account2.credit( depositAmount ); // add to account2 balance
40
41        // display balances
42        System.out.printf( "account1 balance: $%.2f\n",
43           account1.getBalance() );
44        System.out.printf( "account2 balance: $%.2f\n",
45           account2.getBalance() );
46     } // end main
47  } // end class AccountTest
```

```
account1 balance: $50.00
account2 balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

account1 balance: $75.53
account2 balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

account1 balance: $75.53
account2 balance: $123.45
```

Fig. 3.14 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 2.)

Line 21 declares local variable `depositAmount` to store each deposit amount entered by the user. Unlike the instance variable `balance` in class `Account`, local variable `depositAmount` in `main` is *not* initialized to 0.0 by default. However, this variable does not need to be initialized here, because its value will be determined by the user's input.

Line 23 prompts the user to enter a deposit amount for `account1`. Line 24 obtains the input from the user by calling `Scanner` object `input`'s **`nextDouble`** method, which returns a `double` value entered by the user. Lines 25–26 display the deposit amount. Line 27 calls object `account1`'s `credit` method and supplies `depositAmount` as the method's argument. When the method is called, the argument's value is assigned to parameter `amount` (line 19 of Fig. 3.13) of method `credit` (lines 19–22 of Fig. 3.13); then method `credit` adds that value to the `balance` (line 21 of Fig. 3.13). Lines 30–33 (Fig. 3.14) output the balances of both `Account`s again to show that only `account1`'s balance changed.

Line 35 prompts the user to enter a deposit amount for `account2`. Line 36 obtains the input from the user by calling `Scanner` object `input`'s `nextDouble` method. Lines 37–38 display the deposit amount. Line 39 calls object `account2`'s `credit` method and supplies `depositAmount` as the method's argument; then method `credit` adds that value to the balance. Finally, lines 42–45 output the balances of both `Account`s again to show that only `account2`'s balance changed.

UML Class Diagram for Class `Account`

The UML class diagram in Fig. 3.15 models class `Account` of Fig. 3.13. The diagram models the `private` attribute `balance` with UML type `Double` to correspond to the class's instance variable `balance` of Java type `double`. The diagram models class `Account`'s constructor with a parameter `initialBalance` of UML type `Double` in the third compartment of the class. The class's two `public` methods are modeled as operations in the third compartment as well. The diagram models operation `credit` with an `amount` parameter of UML type `Double` (because the corresponding method has an `amount` parameter of Java type `double`), and operation `getBalance` with a return type of `Double` (because the corresponding Java method returns a `double` value).

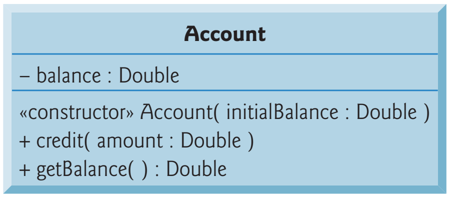

Account
– balance : Double
«constructor» Account(initialBalance : Double)
+ credit(amount : Double)
+ getBalance( ) : Double

Fig. 3.15 | UML class diagram indicating that class `Account` has a `private` `balance` attribute of UML type `Double`, a constructor (with a parameter of UML type `Double`) and two `public` operations—`credit` (with an `amount` parameter of UML type `Double`) and `getBalance` (returns UML type `Double`).

3.8 (Optional) GUI and Graphics Case Study: Using Dialog Boxes

This optional case study is designed for those who want to begin learning Java's powerful capabilities for creating graphical user interfaces (GUIs) and graphics early in the book, before the main discussions of these topics in Chapter 14, GUI Components: Part 1, Chapter 15, Graphics and Java 2D, and Chapter 25, GUI Components: Part 2.

The GUI and Graphics Case Study appears in 10 brief sections (see Fig. 3.16). Each section introduces new concepts and provides examples with screen captures that show sample interactions and results. In the first few sections, you'll create your first graphical applications. In subsequent sections, you'll use object-oriented programming concepts to create an application that draws a variety of shapes. When we formally introduce GUIs in Chapter 14, we use the mouse to choose exactly which shapes to draw and where to draw them. In Chapter 15, we add capabilities of the Java 2D graphics API to draw the shapes with different line thicknesses and fills. We hope you find this case study informative and entertaining.

Location	Title—Exercise(s)
Section 3.8	Using Dialog Boxes—Basic input and output with dialog boxes
Section 4.14	Creating Simple Drawings—Displaying and drawing lines on the screen
Section 5.10	Drawing Rectangles and Ovals—Using shapes to represent data
Section 6.13	Colors and Filled Shapes—Drawing a bull's-eye and random graphics
Section 7.15	Drawing Arcs—Drawing spirals with arcs
Section 8.16	Using Objects with Graphics—Storing shapes as objects
Section 9.8	Displaying Text and Images Using Labels—Providing status information
Section 10.8	Drawing with Polymorphism—Identifying the similarities between shapes
Exercise 14.17	Expanding the Interface—Using GUI components and event handling
Exercise 15.31	Adding Java 2D—Using the Java 2D API to enhance drawings

Fig. 3.16 | Summary of the GUI and Graphics Case Study in each chapter.

Displaying Text in a Dialog Box

The programs presented thus far display output in the command window. Many applications use windows or **dialog boxes** (also called **dialogs**) to display output. Web browsers such as Firefox, Internet Explorer, Chrome and Safari display web pages in their own windows. E-mail programs allow you to type and read messages in a window. Typically, dialog boxes are windows in which programs display important messages to users. Class **`JOptionPane`** provides prebuilt dialog boxes that enable programs to display windows containing messages—such windows are called **message dialogs**. Figure 3.17 displays the `String` `"Welcome\nto\nJava"` in a message dialog.

```java
// Fig. 3.17: Dialog1.java
// Using JOptionPane to display multiple lines in a dialog box.
import javax.swing.JOptionPane; // import class JOptionPane

public class Dialog1
{
   public static void main( String[] args )
   {
      // display a dialog with a message
      JOptionPane.showMessageDialog( null, "Welcome\nto\nJava" );
   } // end main
} // end class Dialog1
```

Fig. 3.17 | Using `JOptionPane` to display multiple lines in a dialog box. (Part 1 of 2.)

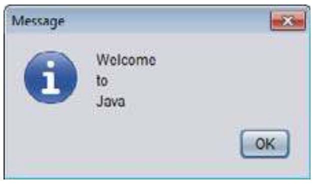


Fig. 3.17 | Using `JOptionPane` to display multiple lines in a dialog box. (Part 2 of 2.)

Line 3 indicates that the program uses class `JOptionPane` from package **`javax.swing`**. This package contains many classes that help you create **graphical user interfaces (GUIs)**. **GUI components** facilitate data entry by a program's user and presentation of outputs to the user. Line 10 calls `JOptionPane` method **`showMessageDialog`** to display a dialog box containing a message. The method requires two arguments. The first helps the Java application determine where to position the dialog box. A dialog is typically displayed from a GUI application with its own window. The first argument refers to that window (known as the parent window) and causes the dialog to appear centered over the application's window. If the first argument is `null`, the dialog box is displayed at the center of your screen. The second argument is the `String` to display in the dialog box.

Introducing `static` Methods

`JOptionPane` method `showMessageDialog` is a so-called **`static` method**. Such methods often define frequently used tasks. For example, many programs display dialog boxes, and the code to do this is the same each time. Rather than requiring you to "reinvent the wheel" and create code to display a dialog, the designers of class `JOptionPane` declared a `static` method that performs this task for you. A `static` method is called by using its class name followed by a dot (`.`) and the method name, as in

```
    ClassName.methodName( arguments )
```

Notice that you do *not* create an object of class `JOptionPane` to use its `static` method `showMessageDialog`.We discuss `static` methods in more detail in Chapter 6.

Entering Text in a Dialog

Figure 3.18 uses another predefined `JOptionPane` dialog called an **input dialog** that allows the user to enter data into a program. The program asks for the user's name and responds with a message dialog containing a greeting and the name that the user entered.

Lines 10–11 use `JOptionPane` method **`showInputDialog`** to display an input dialog containing a prompt and a field (known as a **text field**) in which the user can enter text. Method `showInputDialog`'s argument is the prompt that indicates what the user should enter. The user types characters in the text field, then clicks the **OK** button or presses the *Enter* key to return the `String` to the program. Method `showInputDialog` (line 11) returns a `String` containing the characters typed by the user. We store the `String` in variable `name` (line 10). [*Note:* If you press the dialog's **Cancel** button or press the *Esc* key, the method returns `null` and the program displays the word "null" as the name.]

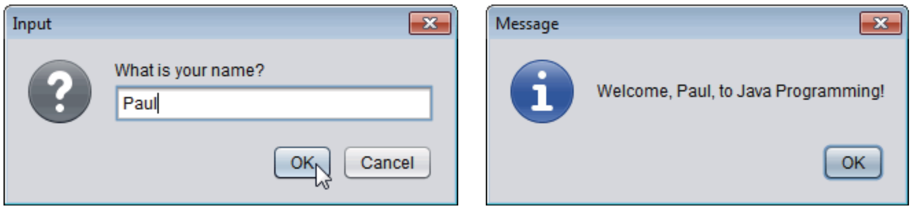
Lines 14–15 use `static String` method **`format`** to return a `String` containing a greeting with the user's name. Method `format` works like method `System.out.printf`, except that `format` returns the formatted `String` rather than displaying it in a command window. Line 18 displays the greeting in a message dialog, just as we did in Fig. 3.17.

```java
// Fig. 3.18: NameDialog.java
// Basic input with a dialog box.
import javax.swing.JOptionPane;

public class NameDialog
{
   public static void main( String[] args )
   {
      // prompt user to enter name
      String name =
         JOptionPane.showInputDialog( "What is your name?" );

      // create the message
      String message =
         String.format( "Welcome, %s, to Java Programming!", name );

      // display the message to welcome the user by name
      JOptionPane.showMessageDialog( null, message );
   } // end main
} // end class NameDialog
```



Welcome, Paul, to Java Programming!

Fig. 3.18 | Obtaining user input from a dialog.

GUI and Graphics Case Study Exercise

3.1 Modify the addition program in Fig. 2.7 to use dialog-based input and output with the methods of class `JOptionPane`. Since method `showInputDialog` returns a `String`, you must convert the `String` the user enters to an `int` for use in calculations. The `Integer` class's `static` method `parseInt` takes a `String` argument representing an integer (e.g., the result of `JOptionPane.showInputDialog`) and returns the value as an `int`. Method `parseInt` is a `static` method of class `Integer` (from package `java.lang`). If the `String` does not contain a valid integer, the program will terminate with an error.

3.9 Wrap-Up

In this chapter, you learned how to declare instance variables of a class to maintain data for each object of the class, and how to declare methods that operate on that data. You learned how to call a method to tell it to perform its task and how to pass information to methods as arguments. You learned the difference between a local variable of a method and an instance variable of a class and that only instance variables are initialized automatically. You also learned how to use a class's constructor to specify the initial values for an object's instance variables. Throughout the chapter, you saw how the UML can be used to create class diagrams that model the constructors, methods and attributes of classes. Finally, you learned about floating-point numbers—how to store them with variables of primitive type `double`, how to input them with a `Scanner` object and how to format them with `printf` and format specifier `%f` for display purposes. In the next chapter we begin our introduction to control statements, which specify the order in which a program's actions are performed. You'll use these in your methods to specify how they should perform their tasks.

Summary

Section 3.2 Declaring a Class with a Method and Instantiating an Object of a Class

- Each class declaration that begins with the access modifier (p. 72) `public` must be stored in a file that has exactly the same name as the class and ends with the `.java` file-name extension.
- Every class declaration contains keyword `class` followed immediately by the class's name.
- A method declaration that begins with keyword `public` indicates that the method can be called by other classes declared outside the class declaration.
- Keyword `void` indicates that a method will perform a task but will not return any information.
- By convention, method names begin with a lowercase first letter and all subsequent words in the name begin with a capital first letter.
- Empty parentheses following a method name indicate that the method does not require any parameters to perform its task.
- Every method's body is delimited by left and right braces (`{` and `}`).
- The method's body contains statements that perform the method's task. After the statements execute, the method has completed its task.
- When you attempt to execute a class, Java looks for the class's `main` method to begin execution.
- Typically, you cannot call a method of another class until you create an object of that class.
- A class instance creation expression (p. 74) begins with keyword `new` and creates a new object.
- To call a method of an object, follow the variable name with a dot separator (`.`; p. 75), the method name and a set of parentheses containing the method's arguments.
- In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top compartment contains the class's name centered horizontally in boldface. The middle one contains the class's attributes, which correspond to fields (p. 79) in Java. The bottom one contains the class's operations (p. 76), which correspond to methods and constructors in Java.
- The UML models operations by listing the operation name followed by a set of parentheses. A plus sign (+) in front of the operation name indicates that the operation is a `public` one in the UML (i.e., a `public` method in Java).

Section 3.3 Declaring a Method with a Parameter

- Methods often require parameters (p. 76) to perform their tasks. Such additional information is provided to methods via arguments in method calls.
- `Scanner` method `nextLine` (p. 76) reads characters until a newline character is encountered, then returns the characters as a `String`.
- `Scanner` method `next` (p. 77) reads characters until any white-space character is encountered, then returns the characters as a `String`.
- A method that requires data to perform its task must specify this in its declaration by placing additional information in the method's parameter list (p. 76).
- Each parameter must specify both a type and a variable name.

- At the time a method is called, its arguments are assigned to its parameters. Then the method body uses the parameter variables to access the argument values.
- A method specifies multiple parameters in a comma-separated list.
- The number of arguments in the 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.
- Class `String` is in package `java.lang`, which is imported implicitly into all source-code files.
- By default, classes compiled into the same directory are in the same package. Classes in the same package are implicitly imported into the source-code files of other classes in the same package.
- `import` declarations are not required if you always use fully qualified class names (p. 79).
- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses following the operation name.
- The UML has its own data types similar to those of Java. Not all the UML data types have the same names as the corresponding Java types.
- The UML type `String` corresponds to the Java type `String`.

Section 3.4 Instance Variables, set *Methods and* get *Methods*

- Variables declared in a method's body are local variables and can be used only in that method.
- A class normally consists of one or more methods that manipulate the attributes (data) that belong to a particular object of the class. Such variables are called fields and are declared inside a class declaration but outside the bodies of the class's method declarations.
- When each object of a class maintains its own copy of an attribute, the corresponding field is known as an instance variable.
- Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.
- Declaring instance variables with access modifier `private` (p. 80) is known as data hiding.
- A benefit of fields is that all the methods of the class can use the fields. Another distinction between a field and a local variable is that a field has a default initial value (p. 82) provided by Java when you do not specify the field's initial value, but a local variable does not.
- The default value for a field of type `String` (or any other reference type) is `null`.
- When a method that specifies a return type (p. 73) is called and completes its task, the method returns a result to its calling method (p. 73).
- Classes often provide `public` methods to allow the class's clients to *set* or *get* `private` instance variables (p. 83). The names of these methods need not begin with *set* or *get*, but this naming convention is recommended and is required for special Java software components called JavaBeans.
- The UML represents instance variables as an attribute name, followed by a colon and the type.
- Private attributes are preceded by a minus sign (–) in the UML.
- The UML indicates an operation's return type by placing a colon and the return type after the parentheses following the operation name.
- UML class diagrams (p. 75) do not specify return types for operations that do not return values.

Section 3.5 Primitive Types vs. Reference Types

- Types in Java are divided into two categories—primitive types and reference types. The primitive types are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`. All other types are reference types, so classes, which specify the types of objects, are reference types.

- A primitive-type variable can store exactly one value of its declared type at a time.
- Primitive-type instance variables are initialized by default. Variables of types `byte`, `char`, `short`, `int`, `long`, `float` and `double` are initialized to `0`. Variables of type `boolean` are initialized to `false`.
- Reference-type variables (called references; p. 84) store the location of an object in the computer's memory. Such variables refer to objects in the program. The object that's referenced may contain many instance variables and methods.
- Reference-type fields are initialized by default to the value `null`.
- A reference to an object (p. 84) is required to invoke an object's instance methods. A primitive-type variable does not refer to an object and therefore cannot be used to invoke a method.

Section 3.6 Initializing Objects with Constructors

- Keyword `new` requests memory from the system to store an object, then calls the corresponding class's constructor (p. 74) to initialize the object.
- A constructor can be used to initialize an object of a class when the object is created.
- Constructors can specify parameters but cannot specify return types.
- If a class does not define constructors, the compiler provides a default constructor (p. 85) with no parameters, and the class's instance variables are initialized to their default values.
- The UML models constructors in the third compartment of a class diagram. To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and »; p. 88) before the constructor's name.

Section 3.7 Floating-Point Numbers and Type `double`

- A floating-point number (p. 88) is a number with a decimal point. Java provides two primitive types for storing floating-point numbers (p. 88) in memory—`float` and `double`. The primary difference between these types is that `double` variables can store numbers with larger magnitude and finer detail (known as the number's precision; p. 88) than `float` variables.
- Variables of type `float` represent single-precision floating-point numbers and have seven significant digits. Variables of type `double` represent double-precision floating-point numbers. These require twice as much memory as `float` variables and provide 15 significant digits—approximately double the precision of `float` variables.
- Floating-point literals (p. 88) are of type `double` by default.
- `Scanner` method `nextDouble` (p. 91) returns a `double` value.
- The format specifier `%f` (p. 90) is used to output values of type `float` or `double`. The format specifier `%.2f` specifies that two digits of precision (p. 90) should be output to the right of the decimal point in the floating-point number.
- The default value for a field of type `double` is `0.0`, and the default value for a field of type `int` is `0`.

Self-Review Exercises

3.1 Fill in the blanks in each of the following:

a) Each class declaration that begins with keyword ________ must be stored in a file that has exactly the same name as the class and ends with the `.java` file-name extension.
b) Keyword ________ in a class declaration is followed immediately by the class's name.
c) Keyword ________ requests memory from the system to store an object, then calls the corresponding class's constructor to initialize the object.
d) Each parameter must specify both a(n) ________ and a(n) ________.
e) By default, classes that are compiled in the same directory are considered to be in the same package, known as the ________.

f) When each object of a class maintains its own copy of an attribute, the field that represents the attribute is also known as a(n) ________.
g) Java provides two primitive types for storing floating-point numbers in memory: ________ and ________.
h) Variables of type `double` represent ________ floating-point numbers.
i) `Scanner` method ________ returns a `double` value.
j) Keyword `public` is an access ________.
k) Return type ________ indicates that a method will not return a value.
l) `Scanner` method ________ reads characters until it encounters a newline character, then returns those characters as a `String`.
m) Class `String` is in package ________.
n) A(n) ________ is not required if you always refer to a class with its fully qualified class name.
o) A(n) ________ is a number with a decimal point, such as 7.33, 0.0975 or 1000.12345.
p) Variables of type `float` represent ________ floating-point numbers.
q) The format specifier ________ is used to output values of type `float` or `double`.
r) Types in Java are divided into two categories—________ types and ________ types.

3.2 State whether each of the following is *true* or *false*. If *false*, explain why.
a) By convention, method names begin with an uppercase first letter, and all subsequent words in the name begin with a capital first letter.
b) An `import` declaration is not required when one class in a package uses another in the same package.
c) Empty parentheses following a method name in a method declaration indicate that the method does not require any parameters to perform its task.
d) Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared.
e) A primitive-type variable can be used to invoke a method.
f) Variables declared in the body of a particular method are known as instance variables and can be used in all methods of the class.
g) Every method's body is delimited by left and right braces (`{` and `}`).
h) Primitive-type local variables are initialized by default.
i) Reference-type instance variables are initialized by default to the value `null`.
j) Any class that contains `public static void main( String[] args )` can be used to execute an application.
k) The number of arguments in the method call must match the number of parameters in the method declaration's parameter list.
l) Floating-point values that appear in source code are known as floating-point literals and are type `float` by default.

3.3 What is the difference between a local variable and a field?

3.4 Explain the purpose of a method parameter. What is the difference between a parameter and an argument?

Answers to Self-Review Exercises

3.1 a) `public`. b) `class`. c) `new`. d) type, name. e) default package. f) instance variable. g) `float`, `double`. h) double-precision. i) `nextDouble`. j) modifier. k) `void`. l) `nextLine`. m) `java.lang`. n) `import` declaration. o) floating-point number. p) single-precision. q) `%f`. r) primitive, reference.

3.2 a) False. By convention, method names begin with a lowercase first letter and all subsequent words in the name begin with a capital first letter. b) True. c) True. d) True. e) False. A primitive-type variable cannot be used to invoke a method—a reference to an object is required to invoke the object's methods. f) False. Such variables are called local variables and can be used only in the method in which they're declared. g) True. h) False. Primitive-type instance variables are initialized by default. Each local variable must explicitly be assigned a value. i) True. j) True. k) True. l) False. Such literals are of type `double` by default.

3.3 A local variable is declared in the body of a method and can be used only from the point at which it's declared through the end of the method declaration. A field is declared in a class, but not in the body of any of the class's methods. Also, fields are accessible to all methods of the class. (We'll see an exception to this in Chapter 8, Classes and Objects: A Deeper Look.)

3.4 A parameter represents additional information that a method requires to perform its task. Each parameter required by a method is specified in the method's declaration. An argument is the actual value for a method parameter. When a method is called, the argument values are passed to the corresponding parameters of the method so that it can perform its task.

Exercises

3.5 *(Keyword **`new`**)* What's the purpose of keyword `new`? Explain what happens when you use it.

3.6 *(Default Constructors)* What is a default constructor? How are an object's instance variables initialized if a class has only a default constructor?

3.7 *(Instance Variables)* Explain the purpose of an instance variable.

3.8 *(Using Classes Without Importing Them)* Most classes need to be imported before they can be used in an application. Why is every application allowed to use classes `System` and `String` without first importing them?

3.9 *(Using a Class Without Importing It)* Explain how a program could use class `Scanner` without importing it.

3.10 *(set and get Methods)* Explain why a class might provide a *set* method and a *get* method for an instance variable.

3.11 *(Modified **`GradeBook`** Class)* Modify class `GradeBook` (Fig. 3.10) as follows:

a) Include a `String` instance variable that represents the name of the course's instructor.
b) Provide a *set* method to change the instructor's name and a *get* method to retrieve it.
c) Modify the constructor to specify two parameters—one for the course name and one for the instructor's name.
d) Modify method `displayMessage` to output the welcome message and course name, followed by `"This course is presented by: "` and the instructor's name.

Use your modified class in a test application that demonstrates the class's new capabilities.

3.12 *(Modified **`Account`** Class)* Modify class `Account` (Fig. 3.13) to provide a method called `debit` that withdraws money from an `Account`. Ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the method should print a message indicating `"Debit amount exceeded account balance."` Modify class `AccountTest` (Fig. 3.14) to test method `debit`.

3.13 *(**`Invoice`** Class)* Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include four pieces of information as instance variables—a part number (type `String`), a part description (type `String`), a quantity of the item being purchased (type `int`) and a price per item (`double`). Your class should have a constructor that initializes the four instance variables. Provide a *set* and a *get* method for each instance variable. In addition, provide a method named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as a `double` value. If the quantity is not positive, it should be set to `0`. If the price per item is not positive, it should be set to `0.0`. Write a test application named `InvoiceTest` that demonstrates class `Invoice`'s capabilities.

3.14 *(`Employee` Class)* Create a class called `Employee` that includes three instance variables—a first name (type `String`), a last name (type `String`) and a monthly salary (`double`). Provide a constructor that initializes the three instance variables. Provide a *set* and a *get* method for each instance variable. If the monthly salary is not positive, do not set its value. Write a test application named `EmployeeTest` that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's *yearly* salary. Then give each `Employee` a 10% raise and display each `Employee`'s yearly salary again.

3.15 *(`Date` Class)* Create a class called `Date` that includes three instance variables—a month (type `int`), a day (type `int`) and a year (type `int`). Provide a constructor that initializes the three instance variables and assumes that the values provided are correct. Provide a *set* and a *get* method for each instance variable. Provide a method `displayDate` that displays the month, day and year separated by forward slashes (`/`). Write a test application named `DateTest` that demonstrates class `Date`'s capabilities.

Making a Difference

3.16 *(Target-Heart-Rate Calculator)* While exercising, you can use a heart-rate monitor to see that your heart rate stays within a safe range suggested by your trainers and doctors. According to the American Heart Association (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), the formula for calculating your *maximum heart rate* in beats per minute is 220 minus your age in years. Your *target heart rate* is a range that's 50–85% of your maximum heart rate. [*Note:* These formulas are estimates provided by the AHA. Maximum and target heart rates may vary based on the health, fitness and gender of the individual. Always consult a physician or qualified health care professional before beginning or modifying an exercise program.] Create a class called `HeartRates`. The class attributes should include the person's first name, last name and date of birth (consisting of separate attributes for the month, day and year of birth). Your class should have a constructor that receives this data as parameters. For each attribute provide *set* and *get* methods. The class also should include a method that calculates and returns the person's age (in years), a method that calculates and returns the person's maximum heart rate and a method that calculates and returns the person's target heart rate. Write a Java application that prompts for the person's information, instantiates an object of class `HeartRates` and prints the information from that object—including the person's first name, last name and date of birth—then calculates and prints the person's age in (years), maximum heart rate and target-heart-rate range.

3.17 *(Computerization of Health Records)* A health care issue that has been in the news lately is the computerization of health records. This possibility is being approached cautiously because of sensitive privacy and security concerns, among others. [We address such concerns in later exercises.] Computerizing health records could make it easier for patients to share their health profiles and histories among their various health care professionals. This could improve the quality of health care, help avoid drug conflicts and erroneous drug prescriptions, reduce costs and, in emergencies, could save lives. In this exercise, you'll design a "starter" `HealthProfile` class for a person. The class attributes should include the person's first name, last name, gender, date of birth (consisting of separate attributes for the month, day and year of birth), height (in inches) and weight (in pounds). Your class should have a constructor that receives this data. For each attribute, provide *set* and *get* methods. The class also should include methods that calculate and return the user's age in years, maximum heart rate and target-heart-rate range (see Exercise 3.16), and body mass index (BMI; see Exercise 2.33). Write a Java application that prompts for the person's information, instantiates an object of class `HealthProfile` for that person and prints the information from that object—including the person's first name, last name, gender, date of birth, height and weight—then calculates and prints the person's age in years, BMI, maximum heart rate and target-heart-rate range. It should also display the BMI values chart from Exercise 2.33.