3.7 Wrap-Up

In this chapter, you learned how to create your own classes and methods, create objects of those classes and call methods of those objects to perform useful actions. You declared instance variables of a class to maintain data for each object of the class, and you declared your own methods to operate on that data. You learned how to call a method to tell it to perform its task and how to pass information to a method as arguments whose values are assigned to the method's parameters. 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. You saw how to create UML class diagrams that model the methods, attributes and constructors of classes. Finally, you learned about floating-point numbers (numbers with decimal points)—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 Chapter 8, we'll begin representing monetary amounts precisely with class `BigDecimal`.] You may have also begun the optional GUI and Graphics case study, learning how to write your first GUI applications. 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 order their tasks.

Summary

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

- Each class you create becomes a new type that can be used to declare variables and create objects.
- You can declare new classes as needed; this is one reason Java is known as an extensible language.

Section 3.2.1 `Account` *Class with an Instance Variable, a* set *Method and a* get *Method*

- Each class declaration that begins with the access modifier (p. 71) `public` must be stored in a file that has the same name as the class and ends with the `.java` filename extension.
- Every class declaration contains keyword `class` followed immediately by the class's name.
- Class, method and variable names are identifiers. By convention all use camel case names. Class names begin with an uppercase letter, and method and variable names begin with a lowercase letter.
- An object has attributes that are implemented as instance variables (p. 72) and carried with it throughout its lifetime.
- Instance variables exist before methods are called on an object, while the methods are executing and after the methods complete execution.
- A class normally contains one or more methods that manipulate the instance variables that belong to particular objects of the class.
- Instance variables are declared inside a class declaration but outside the bodies of the class's method declarations.
- Each object (instance) of the class has its own copy of each of the class's instance variables.
- Most instance-variable declarations are preceded with the keyword `private` (p. 72), which 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.

- Parameters are declared in a comma-separated parameter list (p. 73), which is located inside the parentheses that follow the method name in the method declaration. Multiple parameters are separated by commas. Each parameter must specify a type followed by a variable name.
- Variables declared in the body of a particular method are local variables and can be used only in that method. When a method terminates, the values of its local variables are lost. A method's parameters are local variables of the method.
- Every method's body is delimited by left and right braces (`{` and `}`).
- Each method's body contains one or more statements that perform the method's task(s).
- The method's return type specifies the type of data returned to a method's caller. Keyword `void` indicates that a method will perform a task but will not return any information.
- Empty parentheses following a method name indicate that the method does not require any parameters to perform its task.
- When a method that specifies a return type (p. 73) other than `void` is called and completes its task, the method must return a result to its calling method.
- The `return` statement (p. 74) passes a value from a called method back to its caller.
- Classes often provide `public` methods to allow the class's clients to *set* or *get* `private` instance variables. The names of these methods need not begin with *set* or *get*, but this naming convention is recommended.

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

- A class that creates an object of another class, then calls the object's methods, is a driver class.
- `Scanner` method `nextLine` (p. 75) reads characters until a newline character is encountered, then returns the characters as a `String`.
- `Scanner` method `next` (p. 75) reads characters until any white-space character is encountered, then returns the characters as a `String`.
- A class instance creation expression (p. 75) begins with keyword `new` and creates a new object.
- A constructor is similar to a method but is called implicitly by the `new` operator to initialize an object's instance variables at the time the object is created.
- To call a method of an object, follow the object name with a dot separator (p. 76), the method name and a set of parentheses containing the method's arguments.
- Local variables are not automatically initialized. Every instance variable has a default initial value—a value provided by Java when you do not specify the instance variable's initial value.
- The default value for an instance variable of type `String` is `null`.
- A method call supplies values—known as arguments—for each of the method's parameters. Each argument's value is assigned to the corresponding parameter in the method header.
- The number of arguments in a method call must match the number of parameters in the method declaration's parameter list.
- The argument types in the method call must be consistent with the types of the corresponding parameters in the method's declaration.

Section 3.2.3 Compiling and Executing an App with Multiple Classes

- The `javac` command can compile multiple classes at once. Simply list the source-code filenames after the command with each filename separated by a space from the next. If the directory containing the app includes only one app's files, you can compile all of its classes with the command `javac *.java`. The asterisk (`*`) in `*.java` indicates that all files in the current directory ending with the filename extension "`.java`" should be compiled.

Section 3.2.4 **`Account`** *UML Class Diagram with an Instance Variable and* set *and* get *Methods*

- In the UML, each class is modeled in a class diagram (p. 77) as a rectangle with three compartments. The top one contains the class's name centered horizontally in boldface. The middle one contains the class's attributes, which correspond to instance variables in Java. The bottom one contains the class's operations (p. 78), which correspond to methods and constructors in Java.
- 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 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).
- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses after the operation name.
- 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 do not specify return types for operations that do not return values.
- Declaring instance variables `private` is known as data hiding or information hiding.

Section 3.2.5 Additional Notes on Class **`AccountTest`**

- You must call most methods other than `main` explicitly to tell them to perform their tasks.
- A key part of enabling the JVM to locate and call method `main` to begin the app's execution is the `static` keyword, which indicates that `main` is a `static` method that can be called without first creating an object of the class in which the method is declared.
- Most 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. By default, such classes are considered to be in the same package—known as the default package. Classes in the same package are implicitly imported into the source-code files of other classes in that package. An `import` declaration is not required when one class in a package uses another in the same package.
- An `import` declaration is not required if you always refer to a class with its fully qualified class name, which includes its package name and class name.

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

- Declaring instance variables `private` is known as data hiding or information hiding.

Section 3.3 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. 81) 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 instance variables are initialized by default to the value `null`.

- A reference to an object (p. 81) is required to invoke an object's methods. A primitive-type variable does not refer to an object and therefore cannot be used to invoke a method.

Section 3.4 `Account` Class: Initializing Objects with Constructors

- Each class you declare can optionally provide a constructor with parameters that can be used to initialize an object of a class when the object is created.
- Java requires a constructor call for every object that's created.
- Constructors can specify parameters but not return types.
- If a class does not define constructors, the compiler provides a default constructor (p. 83) with no parameters, and the class's instance variables are initialized to their default values.
- If you declare a constructor for a class, the compiler will *not* create a *default constructor* for that class.
- 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. 84) before the constructor's name.

Section 3.5 `Account` Class with a Balance; Floating-Point Numbers and Type `double`

- A floating-point number (p. 84) is a number with a decimal point. Java provides two primitive types for storing floating-point numbers in memory—`float` and `double` (p. 84).
- 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. 84) are of type `double` by default.
- `Scanner` method `nextDouble` (p. 88) returns a `double` value.
- The format specifier `%f` (p. 88) is used to output values of type `float` or `double`. The format specifier `%.2f` specifies that two digits of precision (p. 88) should be output to the right of the decimal point in the floating-point number.
- The default value for an instance variable of type `double` is `0.0`, and the default value for an instance variable 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` filename 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) Java provides two primitive types for storing floating-point numbers in memory: ________ and ________.
g) Variables of type `double` represent ________ floating-point numbers.
h) `Scanner` method ________ returns a `double` value.
i) Keyword `public` is an access ________.

j) Return type ________ indicates that a method will not return a value.
k) `Scanner` method ________ reads characters until it encounters a newline character, then returns those characters as a `String`.
l) Class `String` is in package ________.
m) A(n) ________ is not required if you always refer to a class with its fully qualified class name.
n) A(n) ________ is a number with a decimal point, such as 7.33, 0.0975 or 1000.12345.
o) Variables of type `float` represent ________ -precision floating-point numbers.
p) The format specifier ________ is used to output values of type `float` or `double`.
q) 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) A primitive-type variable can be used to invoke a method.
e) Variables declared in the body of a particular method are known as instance variables and can be used in all methods of the class.
f) Every method's body is delimited by left and right braces (`{` and `}`).
g) Primitive-type local variables are initialized by default.
h) Reference-type instance variables are initialized by default to the value `null`.
i) Any class that contains `public static void main(String[] args)` can be used to execute an app.
j) The number of arguments in the method call must match the number of parameters in the method declaration's parameter list.
k) 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 an instance variable?

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) `float`, `double`. g) double-precision. h) `nextDouble`. i) modifier. j) `void`. k) `nextLine`. l) `java.lang`. m) `import` declaration. n) floating-point number. o) single. p) `%f`. q) 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) 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. e) False. Such variables are called local variables and can be used only in the method in which they're declared. f) True. g) False. Primitive-type instance variables are initialized by default. Each local variable must explicitly be assigned a value. h) True. i) True. j) True. k) 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. An instance variable is declared in a class, but not in the body of any of the class's methods. Also, instance variables are accessible to all methods of the class. (We'll see an exception to this in Chapter 8.)

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 app. Why is every app 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* **`Account`** *Class)* Modify class `Account` (Fig. 3.8) to provide a method called `withdraw` that withdraws money from an `Account`. Ensure that the withdrawal 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 `"Withdrawal amount exceeded account balance."` Modify class `AccountTest` (Fig. 3.9) to test method `withdraw`.

3.12 *(***`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 app named `InvoiceTest` that demonstrates class `Invoice`'s capabilities.

3.13 *(***`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 app 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.

3.14 *(***`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 app named `DateTest` that demonstrates class `Date`'s capabilities.

3.15 *(Removing Duplicated Code in Method* **`main`***)* In the `AccountTest` class of Fig. 3.9, method `main` contains six statements (lines 13–14, 15–16, 28–29, 30–31, 40–41 and 42–43) that each display an `Account` object's `name` and `balance`. Study these statements and you'll notice that they differ only in the `Account` object being manipulated—`account1` or `account2`. In this exercise, you'll define a new `displayAccount` method that contains *one* copy of that output statement. The method's parameter will be an `Account` object and the method will output the object's `name` and `balance`. You'll then replace the six duplicated statements in `main` with calls to `displayAccount`, passing as an argument the specific `Account` object to output.

Modify class `AccountTest` class of Fig. 3.9 to declare the following `displayAccount` method *after* the closing right brace of `main` and *before* the closing right brace of class `AccountTest`:

```
public static void displayAccount(Account accountToDisplay)
{
   // place the statement that displays
   // accountToDisplay's name and balance here
}
```

Replace the comment in the method's body with a statement that displays `accountToDisplay`'s `name` and `balance`.

Recall that `main` is a `static` method, so it can be called without first creating an object of the class in which `main` is declared. We also declared method `displayAccount` as a `static` method. When `main` needs to call another method in the same class without first creating an object of that class, the other method *also* must be declared `static`.

Once you've completed `displayAccount`'s declaration, modify `main` to replace the statements that display each `Account`'s `name` and `balance` with calls to `displayAccount`—each receiving as its argument the `account1` or `account2` object, as appropriate. Then, test the updated `AccountTest` class to ensure that it produces the same output as shown in Fig. 3.9.

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 app 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 app 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.