Chapter 7 Introduction to Classes and Objects

Java™ How to Program, 10/e

Late Objects Version

OBJECTIVES

In this chapter you'll:

- Declare a class and use it to create an object.
- Implement a class's behaviors as methods.
- Implement a class's attributes as instance variables.
- Call an object's methods to make them perform their tasks.

OBJECTIVES

- Learn how local variables of a method differ from instance variables of a class.
- Learn what primitive types and reference types are.
- Use a constructor to initialize an object's data.
- Learn why classes are a natural way to model real-world things and abstract entities.

OUTLINE

7.1 Introduction

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

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

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

7.2.3 Compiling and Executing an App with Multiple Classes

7.2.4 `Account` UML Class Diagram

7.2.5 Additional Notes on Class `AccountTest`

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

OUTLINE

7.4 `Account` Class: Initializing Objects with Constructors

7.4.1 Declaring an `Account` Constructor for Custom Object Initialization

7.4.2 Class `AccountTest`: Initializing `Account` Objects When They're Created

OUTLINE

7.5 `Account` Class with a Balance

7.5.1 `Account` Class with a `balance` Instance Variable of Type `double`

7.5.2 `AccountTest` Class to Use Class `Account`

7.7 Case Study: Class `GradeBook` Using an Array to Store Grades

7.8 Case Study: Class `GradeBook` Using a Two-Dimensional Array

7.2 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.

```
1  // Fig. 7.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account {
6     private String name; // instance variable
7
8     // method to set the name in the object
9     public void setName(String name) {
10       this.name = name; // store the name
11    }
12
13    // method to retrieve the name from the object
14    public String getName() {
15       return name; // return value of name to caller
16    }
17 }
```

Fig. 7.1 | Account class that contains a name instance variable and methods to *set* and *get* its value.

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Class Declaration

- Each class declaration that begins with the access modifier `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.

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Identifiers and Camel Case Naming

- 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.

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Instance Variable `name`

- An object has attributes that are implemented as instance variables and carried with it throughout its lifetime.
- Instance variables exist before methods are called on an object, while the methods are executing and after the methods complete execution.
- 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.

Good Programming Practice 7.1

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

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

***Access Modifiers* `public` *and* `private`**

- Most instance-variable declarations are preceded with the keyword `private`, 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.

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

Instance Method `setName` of Class `Account`

- In the preceding chapters, you've declared only `static` methods in each class.
- A class's `non-static` methods are known as instance methods.
- Method `setName`'s declaration indicates that `setName` receives parameter name of type `String`—which represents the name that will be passed to the method as an argument.

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

- Recall that variables declared in the body of a particular method are local variables and can be used only in that method and that a method's parameters also are local variables of the method.

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

- Method `setName`'s body contains a single statement that assigns the value of the name *parameter* (a `String`) to the class's `name` *instance variable*, thus storing the account name in the object.
- If a method contains a local variable with the *same* name as an instance variable, that method's body will refer to the local variable rather than the instance variable.
- In this case, the local variable is said to *shadow* the instance variable in the method's body.
- The method's body can use the keyword `this` to refer to the shadowed instance variable explicitly.

Good Programming Practice 7.2

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

7.2.1 `Account` Class with an Instance Variable, a *set* Method and a *get* Method (Cont.)

`getName` Method of Class `Account`

- Method `getName` returns a particular `Account` object's `name` to the caller.
- The method has an empty parameter list, so it does *not* require additional information to perform its task.

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

Driver Class *`AccountTest`*

- A class that creates an object of another class, then calls the object's methods, is a driver class.

```
1  // Fig. 7.2: AccountTest.java
2  // Creating and manipulating an Account object.
3  import java.util.Scanner;
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        // create a Scanner object to obtain input from the command window
8        Scanner input = new Scanner(System.in);
9
10       // create an Account object and assign it to myAccount
11       Account myAccount = new Account();
12
13       // display initial value of name (null)
14       System.out.printf("Initial name is: %s%n%n", myAccount.getName());
15
```

Fig. 7.2 | Creating and manipulating an `Account` object. (Part 1 of 3.)

```
16          // prompt for and read name
17          System.out.println("Please enter the name:");
18          String theName = input.nextLine(); // read a line of text
19          myAccount.setName(theName); // put theName in myAccount
20          System.out.println(); // outputs a blank line
21
22          // display the name stored in object myAccount
23          System.out.printf("Name in object myAccount is:%n%s%n",
24             myAccount.getName());
25       }
26    }
```

Fig. 7.2 | Creating and manipulating an `Account` object. (Part 2 of 3.)

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Fig. 7.2 | Creating and manipulating an `Account` object. (Part 3 of 3.)

7.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

`Scanner` ***Object for Receiving Input from the User***

- `Scanner` method `nextLine` reads characters until a newline character is encountered, then returns the characters as a `String`.
- `Scanner` method `next` reads characters until any white-space character is encountered, then returns the characters as a `String`.

7.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

Instantiating an Object—Keyword `new` and Constructors

- A class instance creation expression 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*.

7.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

Calling Class `Account`'s `getName` Method

- To call a method of an object, follow the object name with a dot separator, the method name and a set of parentheses containing the method's arguments.

Error-Prevention Tip 7.1

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

7.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

`null`—*the Default Initial Value for `String` Variables*

- 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`.

7.2.2 `AccountTest` Class That Creates and Uses an Object of Class `Account` (Cont.)

Calling Class `Account`'s `setName` Method

- 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.

7.2.3 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.

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

- We'll often use UML class diagrams to summarize a class's *attributes* and *operations*.
- UML diagrams help systems designers specify a system in a concise, graphical, programming-language-independent manner, before programmers implement the system in a specific programming language.
- Figure 7.3 presents a UML class diagram for class `Account` of Fig. 7.1.

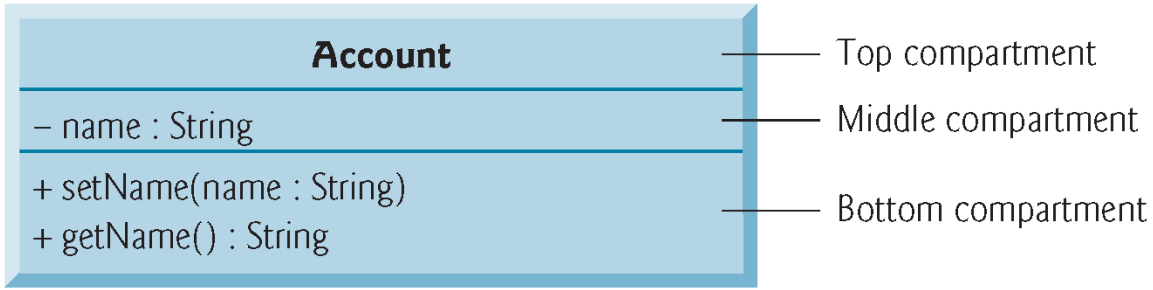

Fig. 7.3 | UML class diagram for class `Account` of Fig. 7.1.

7.2.4 `Account` UML Class Diagram with an Instance Variable and *set* and *get* Methods (Cont.)

Top Compartment

- In the UML, each class is modeled in a class diagram as a rectangle with three compartments.
- The top one contains the class's name centered horizontally in boldface.

7.2.4 `Account` UML Class Diagram with an Instance Variable and *set* and *get* Methods (Cont.)

Middle Compartment

- The middle compartment contains the class's attributes, which correspond to instance variables in Java.

7.2.4 `Account` UML Class Diagram with an Instance Variable and *set* and *get* Methods (Cont.)

Bottom Compartment

- The bottom compartment contains the class's operations, 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).

7.2.4 `Account` UML Class Diagram with an Instance Variable and *set* and *get* Methods (Cont.)

Return Types

- 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.

7.2.4 `Account` UML Class Diagram with an Instance Variable and *set* and *get* Methods (Cont.)

Parameters

- 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

7.2.5 Additional Notes on This Example

Notes on `static` Methods

- A `static` method can call other `static` methods of the same class directly (i.e., using the method name by itself) and can manipulate `static` variables in the same class directly.
- To access the class's instance variables and instance methods, a `static` method must use a reference to an object of the class.
- Instance methods can access all fields (`static` variables and instance variables) and methods of the class.

7.2.5 Additional Notes on This Example

- Many objects of a class, each with its *own* copies of the instance variables, may exist at the same time.
- Java does *not* allow a `static` method to directly access instance variables and instance methods of the same class.

7.2.5 Additional Notes on Class `AccountTest` (Cont.)

Notes on `import` Declarations

- 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.

7.2.5 Additional Notes on Class `AccountTest` (Cont.)

- 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.

Software Engineering Observation 7.1

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

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

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

Software Engineering Observation 7.2

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

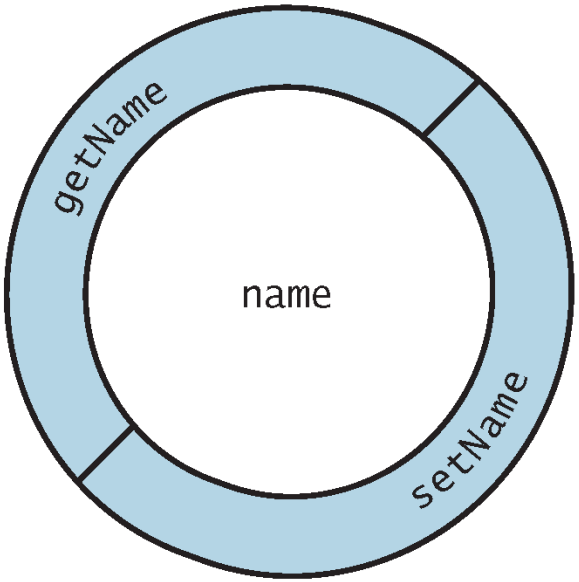

Fig. 7.4 | Conceptual view of an `Account` object with its encapsulated `private` instance variable `name` and protective layer of `public` methods.

7.3 Default and Explicit Initialization for Instance Variables

- Recall that local variables are *not* initialized by default.
- Primitive-type instance variables *are* initialized by default—instance 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 instance variable by assigning the variable a value in its declaration, as in

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

7.3 Default and Explicit Initialization for Instance Variables

- Reference-type instance variables (such as those of type `String`), if not explicitly initialized, are initialized by default to the value `null`—which represents a "reference to nothing."

7.4 `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.

7.4.1 Declaring an `Account` Constructor for Custom Object Initialization

- Figure 7.5 contains a modified `Account` class with such a constructor.

```
1  // Fig. 7.5: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account {
5     private String name; // instance variable
6
7     // constructor initializes name with parameter name
8     public Account(String name) { // constructor name is class name
9        this.name = name;
10    }
11
12    // method to set the name
13    public void setName(String name) {
14       this.name = name;
15    }
16
17    // method to retrieve the name
18    public String getName() {
19       return name;
20    }
21 }
```

Fig. 7.5 | `Account` class with a constructor that initializes the `name`.

Error-Prevention Tip 7.2

Even though it's possible to do so, do not call methods from constructors. We'll explain this in Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces.

7.4.2 Class `AccountTest`: Initializing Account Objects When They're Created

- The `AccountTest` program (Fig. 7.6) initializes two `Account` objects using the constructor.

```java
// Fig. 7.6: AccountTest.java
// Using the Account constructor to initialize the name instance
// variable at the time each Account object is created.

public class AccountTest {
   public static void main(String[] args) {
      // create two Account objects
      Account account1 = new Account("Jane Green");
      Account account2 = new Account("John Blue");

      // display initial value of name for each Account
      System.out.printf("account1 name is: %s%n", account1.getName());
      System.out.printf("account2 name is: %s%n", account2.getName());
   }
}
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 7.6 | Using the `Account` constructor to initialize the `name` instance variable at the time each `Account` object is created.

7.4.2 Class `AccountTest`: Initializing Account Objects When They're Created (Cont.)

Constructors Cannot Return Values

- Constructors can specify parameters but not return types.

Default Constructor

- If a class does not define constructors, the compiler provides a default constructor with no parameters, and the class's instance variables are initialized to their default values.

There's No Default Constructor in a Class That Declares a Constructor

- If you declare a constructor for a class, the compiler will *not* create a *default constructor* for that class.

Software Engineering Observation 7.3

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

7.4.2 Class `AccountTest`: Initializing Account Objects When They're Created (Cont.)

Adding the Contructor to Class `Account`'s UML Class Diagram

- 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 ») before the constructor's name.

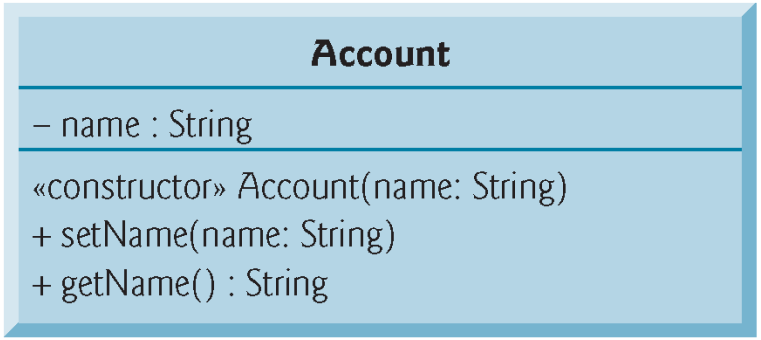

Fig. 7.7 | UML class diagram for `Account` class of Fig. 7.5.

7.5 `Account` Class with a Balance; Floating-Point Numbers

- We now declare an `Account` class that maintains the *balance* of a bank account in addition to the name.
- Most account balances are not integers.
- So, class `Account` represents the account balance as a `double`.

7.5.1 `Account` Class with a `balance` Instance Variable of Type `double`

- Our next app contains a version of class `Account` (Fig. 7.8) that maintains as instance variables the `name` and the `balance` of a bank account.

```
1  // Fig. 7.8: Account.java
2  // Account class with a double instance variable balance and a constructor
3  // and deposit method that perform validation.
4
5  public class Account {
6     private String name; // instance variable
7     private double balance; // instance variable
8
9     // Account constructor that receives two parameters
10    public Account(String name, double balance) {
11       this.name = name; // assign name to instance variable name
12
13       // validate that the balance is greater than 0.0; if it's not,
14       // instance variable balance keeps its default initial value of 0.0
15       if (balance > 0.0) { // if the balance is valid
16          this.balance = balance; // assign it to instance variable balance
17       }
18    }
19
```

Fig. 7.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 1 of 3.)

```
20      // method that deposits (adds) only a valid amount to the balance
21      public void deposit(double depositAmount) {
22         if (depositAmount > 0.0) { // if the depositAmount is valid
23            balance += depositAmount; // add it to the balance
24         }
25      }
26
27      // method returns the account balance
28      public double getBalance() {
29         return balance;
30      }
31
```

Fig. 7.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 2 of 3.)

```
32      // method that sets the name
33      public void setName(String name) {
34         this.name = name;
35      }
36
37      // method that returns the name
38      public String getName() {
39         return name;
40      }
41   }
```

Fig. 7.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 3 of 3.)

7.5.2 `AccountTest` Class to Use Class `Account`

- Class `AccountTest` (Fig. 7.9) creates two `Account` objects and initializes them with a *valid* balance of `50.00` and an *invalid* balance of `-7.53`, respectively—for the purpose of our examples, we assume that balances must be greater than or equal to zero.

```
1  // Fig. 7.9: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.util.Scanner;
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        Account account1 = new Account("Jane Green", 50.00);
8        Account account2 = new Account("John Blue", -7.53);
9
10       // display initial balance of each object
11       System.out.printf("%s balance: $%.2f%n",
12          account1.getName(), account1.getBalance());
13       System.out.printf("%s balance: $%.2f%n%n",
14          account2.getName(), account2.getBalance());
15
```

Fig. 7.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 4.)

```
16          // create a Scanner to obtain input from the command window
17          Scanner input = new Scanner(System.in);
18
19          System.out.print("Enter deposit amount for account1: "); // prompt
20          double depositAmount = input.nextDouble(); // obtain user input
21          System.out.printf("%nadding %.2f to account1 balance%n%n",
22             depositAmount);
23          account1.deposit(depositAmount); // add to account1's balance
24
25          // display balances
26          System.out.printf("%s balance: $%.2f%n",
27             account1.getName(), account1.getBalance());
28          System.out.printf("%s balance: $%.2f%n%n",
29             account2.getName(), account2.getBalance());
30
```

Fig. 7.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 4.)

```
31          System.out.print("Enter deposit amount for account2: "); // prompt
32          depositAmount = input.nextDouble(); // obtain user input
33          System.out.printf("%nadding %.2f to account2 balance%n%n",
34             depositAmount);
35          account2.deposit(depositAmount); // add to account2 balance
36
37          // display balances
38          System.out.printf("%s balance: $%.2f%n",
39             account1.getName(), account1.getBalance());
40          System.out.printf("%s balance: $%.2f%n%n",
41             account2.getName(), account2.getBalance());
42       }
43    }
```

Fig. 7.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 3 of 4.)

```
Jane Green balance: $50.00
John Blue balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

Jane Green balance: $75.53
John Blue balance: $0.00

Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

Jane Green balance: $75.53
John Blue balance: $123.45
```

Fig. 7.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 4 of 4.)

7.5.2 `AccountTest` Class to Use Class `Account` (Cont.)

Formatting Floating-Point Numbers for Display

- The format specifier `%f` is used to output values of type `float` or `double`.
- The format specifier `%.2f` specifies that two digits of precision should be output to the right of the decimal point in the floating-point number.

Error-Prevention Tip 7.3

The Java compiler issues a compilation error if you attempt to use the value of an uninitialized local variable. This helps you avoid dangerous execution-time logic errors. It's always better to get the errors out of your programs at compilation time rather than execution time.

7.5.2 `AccountTest` Class to Use Class `Account` (Cont.)

- 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`.

Software Engineering Observation 7.4

Replacing duplicated code with calls to a method that contains one copy of that code can reduce the size of your program and improve its maintainability.

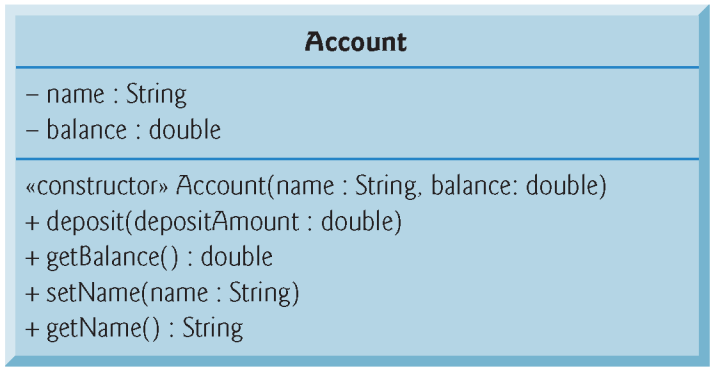

Fig. 7.10 | UML class diagram for `Account` class of Fig. 7.8.

7.6 Case Study: Card Shuffling and Dealing Simulation

- Examples in Chapter 6 demonstrated arrays containing only elements of primitive types.
- Elements of an array can be either primitive types or reference types.
- Next example uses an array of reference-type elements—objects representing playing cards—to develop a class that simulates card shuffling and dealing.

7.6 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Class `Card` (Fig. 7.11) contains two `String` instance variables—`face` and `suit`—that are used to store references to the face and suit names for a specific `Card`.
- Method `toString` creates a `String` consisting of the `face` of the `card`, `" of "` and the `suit` of the card.
  - Can invoke explicitly to obtain a string representation of a `Card`.
  - Called implicitly when the object is used where a `String` is expected.

```
1  // Fig. 7.11: Card.java
2  // Card class represents a playing card.
3
4  public class Card {
5     private final String face; // face of card ("Ace", "Deuce", ...)
6     private final String suit; // suit of card ("Hearts", "Diamonds", ...)
7
8     // two-argument constructor initializes card's face and suit
9     public Card(String cardFace, String cardSuit) {
10       this.face = cardFace; // initialize face of card
11       this.suit = cardSuit; // initialize suit of card
12    }
13
14    // return String representation of Card
15    public String toString() {
16       return face + " of " + suit;
17    }
18 }
```

Fig. 7.11 | Card class represents a playing card.

7.6 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Class `DeckOfCards` (Fig. 7.12) declares as an instance variable a `Card` array named `deck`.
- `Deck`'s elements are `null` by default
  - Constructor fills the `deck` array with `Card` objects.
- Method `shuffle` shuffles the `Card`s in the deck.
  - Loops through all 52 `Card`s (array indices 0 to 51).
  - Each `Card` swapped with a randomly chosen other card in the deck.
- Method `dealCard` deals one `Card` in the array.
  - `currentCard` indicates the index of the next `Card` to be dealt
  - Returns `null` if there are no more cards to deal

```
1  // Fig. 7.12: DeckOfCards.java
2  // DeckOfCards class represents a deck of playing cards.
3  import java.security.SecureRandom;
4
5  public class DeckOfCards {
6     // random number generator
7     private static final SecureRandom randomNumbers = new SecureRandom();
8     private static final int NUMBER_OF_CARDS = 52; // constant # of Cards
9
10    private Card[] deck = new Card[NUMBER_OF_CARDS]; // Card references
11    private int currentCard = 0; // index of next Card to be dealt (0-51)
12
```

Fig. 7.12 | DeckOfCards class represents a deck of playing cards. (Part 1 of 4.)

```
13      // constructor fills deck of Cards
14      public DeckOfCards() {
15         String[] faces = {"Ace", "Deuce", "Three", "Four", "Five", "Six",
16            "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
17         String[] suits = {"Hearts", "Diamonds", "Clubs", "Spades"};
18
19         // populate deck with Card objects
20         for (int count = 0; count < deck.length; count++) {
21            deck[count] =
22               new Card(faces[count % 13], suits[count / 13]);
23         }
24      }
25
```

Fig. 7.12 | `DeckOfCards` class represents a deck of playing cards. (Part 2 of 4.)

```
26      // shuffle deck of Cards with one-pass algorithm
27      public void shuffle() {
28         // next call to method dealCard should start at deck[0] again
29         currentCard = 0;
30
31         // for each Card, pick another random Card (0-51) and swap them
32         for (int first = 0; first < deck.length; first++) {
33            // select a random number between 0 and 51
34            int second = randomNumbers.nextInt(NUMBER_OF_CARDS);
35
36            // swap current Card with randomly selected Card
37            Card temp = deck[first];
38            deck[first] = deck[second];
39            deck[second] = temp;
40         }
41      }
42
```

Fig. 7.12 | DeckOfCards class represents a deck of playing cards. (Part 3 of 4.)

```
43      // deal one Card
44      public Card dealCard() {
45         // determine whether Cards remain to be dealt
46         if (currentCard < deck.length) {
47            return deck[currentCard++]; // return current Card in array
48         }
49         else {
50            return null; // return null to indicate that all Cards were dealt
51         }
52      }
53   }
```

Fig. 7.12 | `DeckOfCards` class represents a deck of playing cards. (Part 4 of 4.)

7.6 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- Figure 7.13 demonstrates class `DeckOfCards`.
- When a `Card` is output as a `String`, the `Card`'s `toString` method is implicitly invoked.

```
1  // Fig. 7.13: DeckOfCardsTest.java
2  // Card shuffling and dealing.
3
4  public class DeckOfCardsTest {
5     // execute application
6     public static void main(String[] args) {
7        DeckOfCards myDeckOfCards = new DeckOfCards();
8        myDeckOfCards.shuffle(); // place Cards in random order
9
10       // print all 52 Cards in the order in which they are dealt
11       for (int i = 1; i <= 52; i++) {
12          // deal and display a Card
13          System.out.printf("%-19s", myDeckOfCards.dealCard());
14
15          if (i % 4 == 0) { // output a newline after every fourth card
16             System.out.println();
17          }
18       }
19    }
20 }
```

Fig. 7.13 | Card shuffling and dealing. (Part 1 of 2.)

```
Six of Spades      Eight of Spades    Six of Clubs       Nine of Hearts
Queen of Hearts    Seven of Clubs     Nine of Spades     King of Hearts
Three of Diamonds  Deuce of Clubs     Ace of Hearts      Ten of Spades
Four of Spades     Ace of Clubs       Seven of Diamonds  Four of Hearts
Three of Clubs     Deuce of Hearts    Five of Spades     Jack of Diamonds
King of Clubs      Ten of Hearts      Three of Hearts    Six of Diamonds
Queen of Clubs     Eight of Diamonds  Deuce of Diamonds  Ten of Diamonds
Three of Spades    King of Diamonds   Nine of Clubs      Six of Hearts
Ace of Spades      Four of Diamonds   Seven of Hearts    Eight of Clubs
Deuce of Spades    Eight of Hearts    Five of Hearts     Queen of Spades
Jack of Hearts     Seven of Spades    Four of Clubs      Nine of Diamonds
Ace of Diamonds    Queen of Diamonds  Five of Clubs      King of Spades
Five of Diamonds   Ten of Clubs       Jack of Spades     Jack of Clubs
```

Fig. 7.13 | Card shuffling and dealing. (Part 2 of 2.)

7.6 Case Study: Card Shuffling and Dealing Simulation (Cont.)

Preventing `NullPointerExceptions`

- In Fig. 7.12, we created a deck array of 52 `Card` references—each element of every reference-type array created with new is default initialized to `null`.
- Reference-type variables which are fields of a class are also initialized to `null` by default.
- A `NullPointerException` occurs when you try to call a method on a null reference.
- In industrial-strength code, ensuring that references are not null before you use them to call methods prevents `NullPointerExceptions`.

7.7 Case Study: Class `GradeBook` Using an Array to Store Grades

- We now present the first part of our case study on developing a `GradeBook` class that instructors can use to maintain students' grades on an exam and display a grade report that includes the grades, class average, lowest grade, highest grade and a grade distribution bar chart.
- The version of class `GradeBook` presented in this section stores the grades for one exam in a one-dimensional array.
- In Section 7.8, we present a version of class `GradeBook` that uses a two-dimensional array to store students' grades for *several* exams.

```
1  // Fig. 7.14: GradeBook.java
2  // GradeBook class using an array to store test grades.
3
4  public class GradeBook {
5     private String courseName; // name of course this GradeBook represents
6     private int[] grades; // array of student grades
7
8     // constructor
9     public GradeBook(String courseName, int[] grades) {
10       this.courseName = courseName;
11       this.grades = grades;
12    }
13
14    // method to set the course name
15    public void setCourseName(String courseName) {
16       this.courseName = courseName;
17    }
18
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 1 of 8.)

```
19      // method to retrieve the course name
20      public String getCourseName() {
21         return courseName;
22      }
23
24      // perform various operations on the data
25      public void processGrades() {
26         // output grades array
27         outputGrades();
28
29         // call method getAverage to calculate the average grade
30         System.out.printf("%nClass average is %.2f%n", getAverage());
31
32         // call methods getMinimum and getMaximum
33         System.out.printf("Lowest grade is %d%nHighest grade is %d%n%n",
34            getMinimum(), getMaximum());
35
36         // call outputBarChart to print grade distribution chart
37         outputBarChart();
38      }
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 2 of 8.)

```
39
40      // find minimum grade
41      public int getMinimum() {
42         int lowGrade = grades[0]; // assume grades[0] is smallest
43
44         // loop through grades array
45         for (int grade : grades) {
46            // if grade lower than lowGrade, assign it to lowGrade
47            if (grade < lowGrade) {
48               lowGrade = grade; // new lowest grade
49            }
50         }
51
52         return lowGrade;
53      }
54
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 3 of 8.)

```
55      // find maximum grade
56      public int getMaximum() {
57         int highGrade = grades[0]; // assume grades[0] is largest
58
59         // loop through grades array
60         for (int grade : grades) {
61            // if grade greater than highGrade, assign it to highGrade
62            if (grade > highGrade) {
63               highGrade = grade; // new highest grade
64            }
65         }
66
67         return highGrade;
68      }
69
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 4 of 8.)

```
70      // determine average grade for test
71      public double getAverage() {
72         int total = 0;
73
74         // sum grades for one student
75         for (int grade : grades) {
76            total += grade;
77         }
78
79         // return average of grades
80         return (double) total / grades.length;
81      }
82
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 5 of 8.)

```
83      // output bar chart displaying grade distribution
84      public void outputBarChart() {
85         System.out.println("Grade distribution:");
86
87         // stores frequency of grades in each range of 10 grades
88         int[] frequency = new int[11];
89
90         // for each grade, increment the appropriate frequency
91         for (int grade : grades) {
92            ++frequency[grade / 10];
93         }
94
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 6 of 8.)

```
95          // for each grade frequency, print bar in chart
96          for (int count = 0; count < frequency.length; count++) {
97             // output bar label ("00-09: ", ..., "90-99: ", "100: ")
98             if (count == 10) {
99                System.out.printf("%5d: ", 100);
100            }
101            else {
102               System.out.printf("%02d-%02d: ", count * 10, count * 10 + 9);
103            }
104
105            // print bar of asterisks
106            for (int stars = 0; stars < frequency[count]; stars++) {
107               System.out.print("*");
108            }
109
110            System.out.println();
111         }
112      }
113
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 7 of 8.)

```
114     // output the contents of the grades array
115     public void outputGrades() {
116        System.out.printf("The grades are:%n%n");
117
118        // output each student's grade
119        for (int student = 0; student < grades.length; student++) {
120           System.out.printf("Student %2d: %3d%n",
121              student + 1, grades[student]);
122        }
123     }
124  }
```

Fig. 7.14 | GradeBook class using an array to store test grades. (Part 8 of 8.)

7.7 Case Study: Class `GradeBook` Using an Array to Store Grades (Cont.)

- The application of Fig. 7.15 creates an object of class `GradeBook` (Fig. 7.14) using the `int` array `grades-Array`.
- Lines 9–10 pass a course name and `gradesArray` to the `GradeBook` constructor.

Software Engineering Observation 7.5

A test harness (or test application) is responsible for creating an object of the class being tested and providing it with data. This data could come from any of several sources. Test data can be placed directly into an array with an array initializer, it can come from the user at the keyboard, from a file (as you'll see in Chapter 15), from a database (as you'll see in Chapter 24) or from a network (as you'll see in online Chapter 28). After passing this data to the class's constructor to instantiate the object, the test harness should call upon the object to test its methods and manipulate its data. Gathering data in the test harness like this allows the class to be more reusable, able to manipulate data from several sources.

```
1  // Fig. 7.15: GradeBookTest.java
2  // GradeBookTest creates a GradeBook object using an array of grades,
3  // then invokes method processGrades to analyze them.
4  public class GradeBookTest {
5     public static void main(String[] args) {
6        // array of student grades
7        int[] gradesArray = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
8
9        GradeBook myGradeBook = new GradeBook(
10          "CS101 Introduction to Java Programming", gradesArray);
11       System.out.printf("Welcome to the grade book for%n%s%n%n",
12          myGradeBook.getCourseName());
13       myGradeBook.processGrades();
14    }
15 }
```

Fig. 7.15 | GradeBookTest creates a GradeBook object using an array of grades, then invokes method processGrades to analyze them. (Part 1 of 3.)

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

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87
```

Fig. 7.15 | `GradeBookTest` creates a `GradeBook` object using an array of grades, then invokes method `processGrades` to analyze them. (Part 2 of 3.)

```
Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.15 | GradeBookTest creates a GradeBook object using an array of grades, then invokes method processGrades to analyze them. (Part 3 of 3.)

7.7 Case Study: Class `GradeBook` Using an Array to Store Grades (Cont.)

- In Chapter 17, Lambdas and Streams, the example of Fig. 17.5 uses stream methods `min`, `max`, `count` and `average` to process the elements of an `int` array elegantly and concisely without having to write repetition statements.
- In the Concurrency chapter, we present an example that uses `stream` method `summaryStatistics` to perform all of these operations in one method call.

7.8 Case Study: Class `GradeBook` Using a Two-Dimensional Array

- In most semesters, students take several exams.
- Figure 7.16 contains a version of class `GradeBook` that uses a two-dimensional array `grades` to store the grades of several students on multiple exams.
  - Each row represents a student's grades for the entire course.
  - Each column represents the grades of all the students who took a particular exam.
- Class `GradeBookTest` (Fig. 7.17) passes the array as an argument to the `GradeBook` constructor.
- In this example, we use a ten-by-three array containing ten students' grades on three exams.

```
1  // Fig. 7.16: GradeBook.java
2  // GradeBook class using a two-dimensional array to store grades.
3
4  public class GradeBook {
5     private String courseName; // name of course this grade book represents
6     private int[][] grades; // two-dimensional array of student grades
7
8     // two-argument constructor initializes courseName and grades array
9     public GradeBook(String courseName, int[][] grades) {
10       this.courseName = courseName;
11       this.grades = grades;
12    }
13
14    // method to set the course name
15    public void setCourseName(String courseName) {
16       this.courseName = courseName;
17    }
18
19    // method to retrieve the course name
20    public String getCourseName() {
21       return courseName;
22    }
23
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 1 of 9.)

```
24      // perform various operations on the data
25      public void processGrades() {
26         // output grades array
27         outputGrades();
28
29         // call methods getMinimum and getMaximum
30         System.out.printf("%n%s %d%n%s %d%n%n",
31            "Lowest grade in the grade book is", getMinimum(),
32            "Highest grade in the grade book is", getMaximum());
33
34         // output grade distribution chart of all grades on all tests
35         outputBarChart();
36      }
37
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 2 of 9.)

```
38      // find minimum grade
39      public int getMinimum() {
40         // assume first element of grades array is smallest
41         int lowGrade = grades[0][0];
42
43         // loop through rows of grades array
44         for (int[] studentGrades : grades) {
45            // loop through columns of current row
46            for (int grade : studentGrades) {
47               // if grade less than lowGrade, assign it to lowGrade
48               if (grade < lowGrade) {
49                  lowGrade = grade;
50               }
51            }
52         }
53
54         return lowGrade;
55      }
56
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 3 of 9.)

```
57      // find maximum grade
58      public int getMaximum() {
59         // assume first element of grades array is largest
60         int highGrade = grades[0][0];
61
62         // loop through rows of grades array
63         for (int[] studentGrades : grades) {
64            // loop through columns of current row
65            for (int grade : studentGrades) {
66               // if grade greater than highGrade, assign it to highGrade
67               if (grade > highGrade) {
68                  highGrade = grade;
69               }
70            }
71         }
72
73         return highGrade;
74      }
75
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 4 of 9.)

```
76      // determine average grade for particular set of grades
77      public double getAverage(int[] setOfGrades) {
78         int total = 0;
79
80         // sum grades for one student
81         for (int grade : setOfGrades) {
82            total += grade;
83         }
84
85         // return average of grades
86         return (double) total / setOfGrades.length;
87      }
88
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 5 of 9.)

```
89      // output bar chart displaying overall grade distribution
90      public void outputBarChart() {
91         System.out.println("Overall grade distribution:");
92
93         // stores frequency of grades in each range of 10 grades
94         int[] frequency = new int[11];
95
96         // for each grade in GradeBook, increment the appropriate frequency
97         for (int[] studentGrades : grades) {
98            for (int grade : studentGrades) {
99               ++frequency[grade / 10];
100           }
101        }
102
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 6 of 9.)

```
103         // for each grade frequency, print bar in chart
104         for (int count = 0; count < frequency.length; count++) {
105            // output bar label ("00-09: ", ..., "90-99: ", "100: ")
106            if (count == 10) {
107               System.out.printf("%5d: ", 100);
108            }
109            else {
110               System.out.printf("%02d-%02d: ",
111                  count * 10, count * 10 + 9);
112            }
113
114            // print bar of asterisks
115            for (int stars = 0; stars < frequency[count]; stars++) {
116               System.out.print("*");
117            }
118
119            System.out.println();
120         }
121      }
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 7 of 9.)

```
122
123     // output the contents of the grades array
124     public void outputGrades() {
125        System.out.printf("The grades are:%n%n");
126        System.out.print("            "); // align column heads
127
128        // create a column heading for each of the tests
129        for (int test = 0; test < grades[0].length; test++) {
130           System.out.printf("Test %d  ", test + 1);
131        }
132
133        System.out.println("Average"); // student average column heading
134
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 8 of 9.)

```
135         // create rows/columns of text representing array grades
136         for (int student = 0; student < grades.length; student++) {
137            System.out.printf("Student %2d", student + 1);
138
139            for (int test : grades[student]) { // output student's grades
140               System.out.printf("%8d", test);
141            }
142
143            // call method getAverage to calculate student's average grade;
144            // pass row of grades as the argument to getAverage
145            double average = getAverage(grades[student]);
146            System.out.printf("%9.2f%n", average);
147         }
148      }
149   }
```

Fig. 7.16 | GradeBook class using a two-dimensional array to store grades. (Part 9 of 9.)

```
1  // Fig. 7.17: GradeBookTest.java
2  // GradeBookTest creates GradeBook object using a two-dimensional array
3  // of grades, then invokes method processGrades to analyze them.
4  public class GradeBookTest {
5     // main method begins program execution
6     public static void main(String[] args) {
7        // two-dimensional array of student grades
8        int[][] gradesArray = {{87, 96, 70},
9                               {68, 87, 90},
10                              {94, 100, 90},
11                              {100, 81, 82},
12                              {83, 65, 85},
13                              {78, 87, 65},
14                              {85, 75, 83},
15                              {91, 94, 100},
16                              {76, 72, 84},
17                              {87, 93, 73}};
18
```

Fig. 7.17 | GradeBookTest creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 1 of 4.)

```
19          GradeBook myGradeBook = new GradeBook(
20             "CS101 Introduction to Java Programming", gradesArray);
21          System.out.printf("Welcome to the grade book for%n%s%n%n",
22             myGradeBook.getCourseName());
23          myGradeBook.processGrades();
24       }
25    }
```

Fig. 7.17 | GradeBookTest creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 2 of 4.)

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

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33
```

Fig. 7.17 | `GradeBookTest` creates `GradeBook` object using a two-dimensional array of grades, then invokes method `processGrades` to analyze them. (Part 3 of 4.)

```
Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: **********
90-99: *******
  100: ***
```

Fig. 7.17 | GradeBookTest creates GradeBook object using a two-dimensional array of grades, then invokes method processGrades to analyze them. (Part 4 of 4.)