maintenance problems when that code needs to be updated—if *six* copies of the same code all have the same error or update to be made, you must make that change *six* times, *without making errors*. Exercise 7.14 asks you to modify Fig. 7.9 to include a `displayAccount` method that takes as a parameter an Account object and outputs the object's `name` and `balance`. You'll then replace `main`'s duplicated statements with six calls to `displayAccount`, thus reducing the size of your program and improving its maintainability by having only *one* copy of the code that displays an `Account`'s `name` and `balance`.

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.

UML Class Diagram for Class **`Account`**

The UML class diagram in Fig. 7.10 concisely models class `Account` of Fig. 7.8. The diagram models in its *second* compartment the `private` attributes `name` of type `String` and `balance` of type `double`. Class `Account`'s *constructor* is modeled in the *third* compartment with parameters `name` of type `String` and `initialBalance` of type `double`. The class's four `public` methods also are modeled in the *third* compartment—operation `deposit` with a `depositAmount` parameter of type `double`, operation `getBalance` with a return type of `double`, operation `setName` with a `name` parameter of type `String` and operation `getName` with a return type of `String`.

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

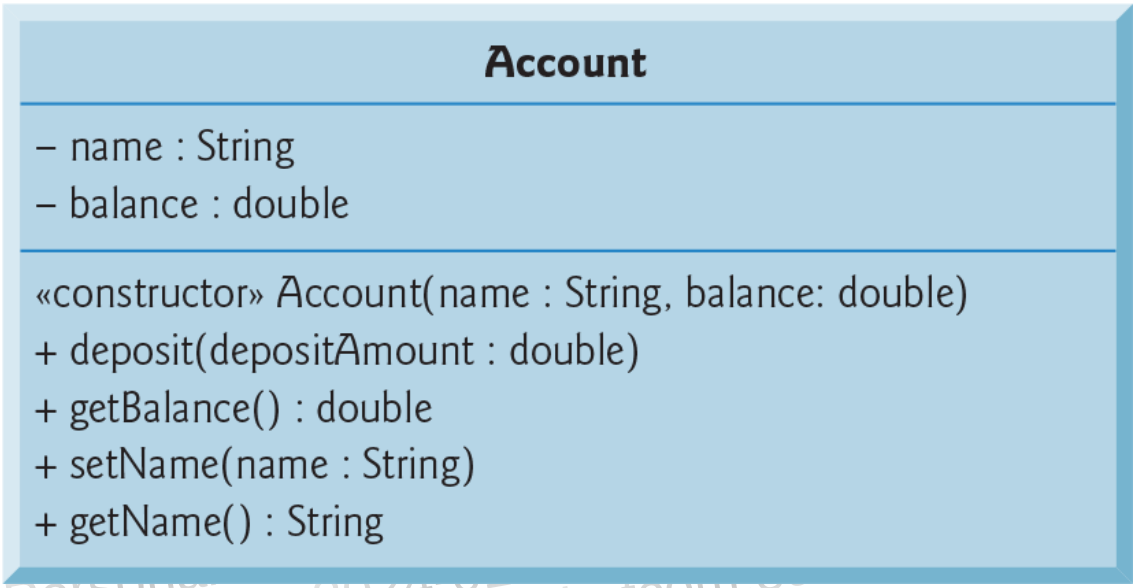

Figure 7.10 Full Alternative Text

7.6 Case Study: Card Shuffling and Dealing Simulation

The examples in Chapter 6 demonstrate arrays containing only elements of primitive types. The elements of an array can be either primitive types or reference types. This section uses random-number generation and an array of reference-type elements—namely objects representing playing cards—to develop a class that simulates card shuffling and dealing. This class can then be used to implement applications that play specific card games. The exercises at the end of the chapter use the classes developed here to build a simple poker application.

We first develop class `Card` (Fig. 7.11), which represents a playing card that has a face (e.g., `"Ace"`, `"Deuce"`, `"Three"`, ..., `"Jack"`, `"Queen"`, `"King"`) and a suit (e.g., `"Hearts"`, `"Diamonds"`, `"Clubs"`, `"Spades"`). Next, we develop the `DeckOfCards` class (Fig. 7.12), which creates a deck of 52 playing cards in which each element is a `Card` object. We then build a test application (Fig. 7.13) that demonstrates class `DeckOfCards`'s card shuffling and dealing capabilities.

Fig. 7.11 **`Card` class represents a playing card.**

```
// Fig. 7.11: Card.java
// Card class represents a playing card.

public class Card {
   private final String face; // face of card ("Ace",
"Deuce", ...)
   private final String suit; // suit of card ("Hearts",
"Diamonds", ...)

   // two-argument constructor initializes card's face
and suit
   public Card(String cardFace, String cardSuit) {
      this.face = cardFace; // initialize face of card
      this.suit = cardSuit; // initialize suit of card
   }

   // return String representation of Card
   public String toString() {
   return face + " of " + suit;
```

```
17      }
18   }
```

Fig. 7.12 **DeckOfCards class represents a deck of playing cards.**


```
 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
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
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
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. 7.13 **Card shuffling and dealing.**

```
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   }
```

```
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
```

Class **`Card`**

Class `Card` (Fig. 7.11) contains two `String` instance variables—`face` and `suit`—that are used to store references to the face name and suit name for a specific `Card`. The constructor for the class (lines 9–12) receives two `String`s that it uses to initialize `face` and `suit`. Method `toString` (lines 15–17) creates a `String` consisting of the `face` of the `card`, the `String " of "` and the `suit` of the card.[2] `Card`'s `toString` method can be invoked *explicitly* to obtain a string representation of a `Card` object (e.g., `"Ace of Spades"`). The `toString` method of an object is called *implicitly* when the object is used where a `String` is expected (e.g., when `printf` outputs the object as a `String` using the `%s` format specifier or when the object is concatenated to a `String` using the `+`

operator). For this behavior to occur, `toString` must be declared with the header shown in Fig. 7.11.

2. You'll learn in Chapter 9 that when we provide a custom toString method for a class, we are actually "overriding" a version of that method supplied by class Object from which all Java classes "inherit." As of Chapter 9, every method we explicitly override will be preceded by the "annotation" @Override, which prevents a common programming error. [Return to reference]

Class `DeckOfCards`

Class `DeckOfCards` (Fig. 7.12) creates and manages an array of Card references. The named constant `NUMBER_OF_CARDS` (line 8) specifies the number of `Cards` in a deck (52). Line 10 declares and initializes an instance variable named `deck` that refers to a new array of `Cards` that has `NUMBER_OF_CARDS` (52) elements—the `deck` array's elements are `null` by `default`. Recall from Chapter 7 that `null` represents a "reference to nothing," so no `Card` objects exist yet. An array of a *reference* type is declared like any other array. Class `DeckOfCards` also declares int instance variable `currentCard` (line 11), representing the sequence number (0–51) of the next `Card` to be dealt from the `deck` array.

`DeckOfCards` Constructor

The constructor uses a loop (lines 20–23) to fill instance variable `deck` with `Card` objects. The loop iterates from `0` and while `count` is less than `deck.length`, causing `count` to take on each integer value from 0 through 51 (the array's indices). Each `Card` is initialized with two `Strings`—one from the `faces` array (which contains the `Strings "Ace"` through `"King"`) and one from the `suits` array (which contains the `Strings "Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`). The calculation `count % 13` always results in a value from 0 to 12 (the 13 indices of the `faces` array in lines 15–16), and the calculation `count / 13` always results in a value from 0 to 3 (the four indices of the `suits` array in line 17). When the loop completes, `deck` contains the `Cards` with faces `"Ace"` through `"King"` in order for each suit (13 `"Hearts"`, then 13 `"Diamonds"`, then 13 `"Clubs"`, then 13 `"Spades"`). We use arrays of `Strings` to represent the faces and suits in this example. In Exercise 7.20, we ask you to modify this example to use arrays of `enum` constants to represent the faces and suits.

`DeckOfCards` Method `shuffle`

Method `shuffle` (lines 27–41) shuffles the `Cards` in the deck. The method loops through all 52 `Cards` (array indices 0 to 51). For each `Card`, line 34 selects a random index between 0 and 51 to select another `Card`. Next, lines 37–39 swap the current `Card` and the randomly selected `Card` in the array. The extra variable `temp` (line 37) temporarily stores one of the two `Card` objects being swapped. After the `for loop` terminates, the `Card` objects are randomly ordered. A total of only 52 swaps are made in a single pass of the entire array, and the array of Card objects is shuffled!

The swap in lines 37–39 cannot be performed with only the two statements

```
deck[first] = deck[second];
deck[second] = deck[first];
```

If `deck[first]` is the `"Ace"` of `"Spades"` and `deck[second]` is the `"Queen"` of `"Hearts"`, after the first assignment, both array elements contain the `"Queen"` of `"Hearts"` and the `"Ace"` of `"Spades"` is lost—so, the extra variable `temp` is needed.

[*Note:* It's recommended that you use a so-called *unbiased* shuffling algorithm for real card games. Such an algorithm ensures that all possible shuffled card sequences are equally likely to occur. Exercise 7.21 asks you to research the popular unbiased Fisher-Yates shuffling algorithm and use it to reimplement the `DeckOfCards` method `shuffle`.]

`DeckOfCards` Method `dealCard`

Method `dealCard` (lines 44–52) deals one `Card` in the array. Recall that `currentCard` indicates the index of the next Card to be dealt (i.e., the `Card` at the *top* of the deck). Thus, line 46 compares `currentCard` to the length of the `deck` array. If the `deck` is not empty (i.e., `currentCard` is less than 52), line 47 returns the "top" `Card` and postincrements `currentCard` to prepare for the next call to `dealCard`—otherwise, line 50 returns `null`.

Shuffling and Dealing Card

Figure 7.13 demonstrates class `DeckOfCards`. Line 7 creates a `DeckOfCards` object named `myDeckOfCards`. The `DeckOfCards` constructor creates the deck with the 52

`Card` objects in order by suit and face. Line 8 invokes `myDeckOfCards`'s `shuffle` method to rearrange the `Card` objects. Lines 11–18 deal all 52 `Cards` and print them in four columns of 13 `Cards` each. Line 13 deals one `Card` object by invoking `myDeckOfCards`'s `dealCard` method, then displays the `Card` left justified in a field of 19 characters. When a `Card` is output