7.6 Case Study: Card Shuffling and Dealing Simulation

The examples in the chapter thus far have used arrays containing elements of primitive types. Recall from Section 7.2 that 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. 7.9), 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. 7.10), which creates a deck of 52 playing cards in which each element is a `Card` object. We then build a test application (Fig. 7.11) that demonstrates class `DeckOfCards`’s card shuffling and dealing capabilities.

Class `Card`

Class `Card` (Fig. 7.9) 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.[1] `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. 7.9.

1. 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.

```
1   // Fig. 7.9: Card.java
2   // Card class represents a playing card.
3
4   public class Card {
5      private final String face; // face of card ("
6      private final String suit; // suit of card ("
7
8      // two-argument constructor initializes card'
9      public Card(String cardFace, String cardSuit)
```

```
        this.face = cardFace; // initialize face o
        this.suit = cardSuit; // initialize suit o
    }

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

Fig. 7.9

`Card` class represents a playing card.

Class `DeckOfCards`

Class `DeckOfCards` (Fig. 7.10) creates and manages an array of `Card` references. The named constant `NUMBER_OF_CARDS` (line 8) specifies the number of `Card`s in a deck (52). Line 10 declares and initializes an instance variable named `deck` that refers to a new array of `Card`s that has `NUMBER_OF_CARDS` (52) elements—the `deck` array's elements are `null` by default. Recall from Chapter 3 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.

```
// Fig. 7.10: DeckOfCards.java
// DeckOfCards class represents a deck of playin
import java.security.SecureRandom;

public class DeckOfCards {
   // random number generator
   private static final SecureRandom randomNumbe
   private static final int NUMBER_OF_CARDS = 52

   private Card[] deck = new Card[NUMBER_OF_CARD
   private int currentCard = 0; // index of next

   // constructor fills deck of Cards
   public DeckOfCards() {
      String[] faces = {"Ace", "Deuce", "Three",
         "Seven", "Eight", "Nine", "Ten", "Jack"
      String[] suits = {"Hearts", "Diamonds", "C

      // populate deck with Card objects
      for (int count = 0; count < deck.length; c
         deck[count] =
            new Card(faces[count % 13], suits[co
      }
   }

   // shuffle deck of Cards with one-pass algori
   public void shuffle() {
      // next call to method dealCard should sta
      currentCard = 0;

      // for each Card, pick another random Card
      for (int first = 0; first < deck.length; f
         // select a random number between 0 and
         int second = randomNumbers.nextInt(NUMB

         // swap current Card with randomly sele
         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 de
46          if (currentCard < deck.length) {
47             return deck[currentCard++]; // return c
48          }
49          else {
50             return null; // return null to indicate
51          }
52       }
53    }
```

Fig. 7.10

`DeckOfCards` class represents a deck of playing cards.

`DeckOfCards` Constructor

The class's constructor uses a `for` statement (lines 20–23) to fill the `deck` instance variable with `Card` objects. The loop initializes control variable `count` to `0` and loops while `count` is less than `deck.length`, causing `count` to take on each integer value from 0 through 51 (the `deck` array's indices). Each `Card` is instantiated and initialized with two `String`s—one from the `faces` array (which contains the

`String`s `"Ace"` through `"King"`) and one from the `suits` array (which contains the `String`s `"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 `Card`s 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 `String`s to represent the faces and suits in this example. In Exercise 7.34, 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 `Card`s in the deck. The method loops through all 52 `Card`s (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 7.35 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 Cards

Figure 7.11 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 `Card`s and print them in four columns of 13 `Card`s 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 as a `String`, the `Card`'s `toString` method (Fig. 7.9) is implicitly invoked. Lines 15–17 in Fig. 7.11 start a new line after every four `Card`s.

```
1   // Fig. 7.11: 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 DeckOfCard
8         myDeckOfCards.shuffle(); // place Cards in
9
10        // print all 52 Cards in the order in whic
11        for (int i = 1; i <= 52; i++) {
12           // deal and display a Card
```

```
            System.out.printf("%-19s", myDeckOfCard

            if (i % 4 == 0) { // output a newline a
               System.out.println();
            }
         }
      }
   }
```

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	Queen of	Five of	King of

Diamonds	Diamonds	Clubs	Spades
Five of Diamonds	Ten of Clubs	Jack of Spades	Jack of Clubs

Fig. 7.11

Card shuffling and dealing.

Preventing `NullPointerExceptions`

In Fig. 7.10, we created a `deck` array of 52 `Card` references—by default, each element of a reference-type array created with `new` is initialized to `null`. Similarly, reference-type 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 `NullPointerException`s.